CS 3006 Parallel and Distributed Computer

Fall 2023

1. Learn about parallel and distributed computer architectures.(1)
2. Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.(4)
3. Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.(2)
4. Use Hadoop or MapReduce programming model to write bigdata applications.(5)

Week # 8

Dr. Nadeem Kafi Khan

Data Decomposition

- *Ideal for problems that operate on large data structures*
- **Steps**
  1. The data on which the computations are performed are partitioned
  2. Data partition is used to induce a partitioning of the computations into tasks.
- Data Partitioning
  - Partition output data
  - Partition input data
  - Partition input + output data
  - Partition intermediate data

Data Decomposition: Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input).
- A partition of the output across tasks decomposes the problem naturally.

1. Input: if each output is described as a function of the input directly. Some combination of the individual results may be necessary.
2. Output data decomposition: if it applies, it can result in less communication.
3. Intermediate data decomposition more rare.
4. Owner computes rules: the process that owns a part of the data performs all the computations related to it.

Example ?

Output Data Decomposition: Example

(Distributed Memory Setup)

Discussed in the context of Intiaz Supunt

- Size of Trasaction
- Why we need item frequency?
- How to select different type of data decomposition
- How data is distributed among nodes as a result?

Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.

MPI cluster processing using Map-Reduce

Problem: Find the number of times that each itemset in I appears in all the transactions; i.e., the number of transactions of which each itemset is a subset of.

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

Decomposition type (1)

(b) Partitioning the frequencies (and itemsets) among the tasks

Partitioning the output into two parts.

task 1

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 2

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	C, D	1
B, D, E, F, K, L	D, K	2
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

Part (b) shows how two tasks can achieve results by partitioning the output into two parts and having each task compute its half of the frequencies.

Figure 3.12. Computing itemset frequencies in a transaction database.

Example 3.6 Computing frequencies of itemsets in a transaction database

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

Understand the problem

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

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

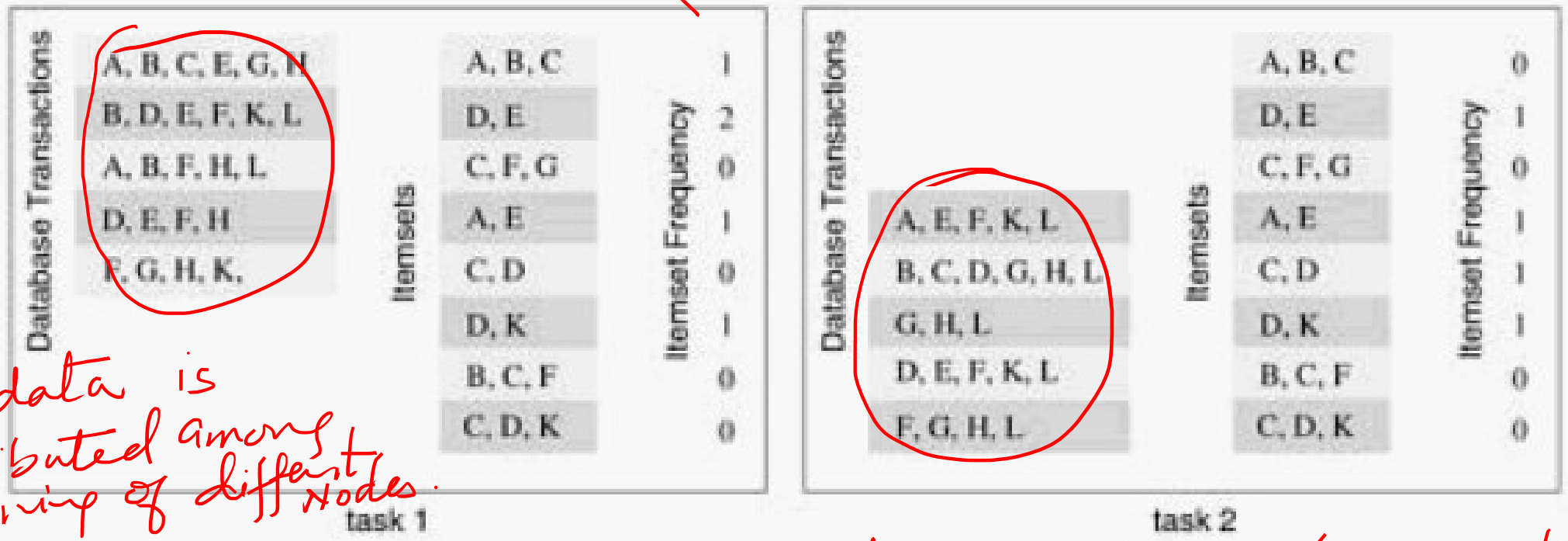
Scatter-gather | Map Reduce

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

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

Decomposition type 2

(a) Partitioning the transactions among the tasks



input data is distributed among tasks running of different nodes.

How Intaiz SuperMart takes it purchasing decision base on processing this data?

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

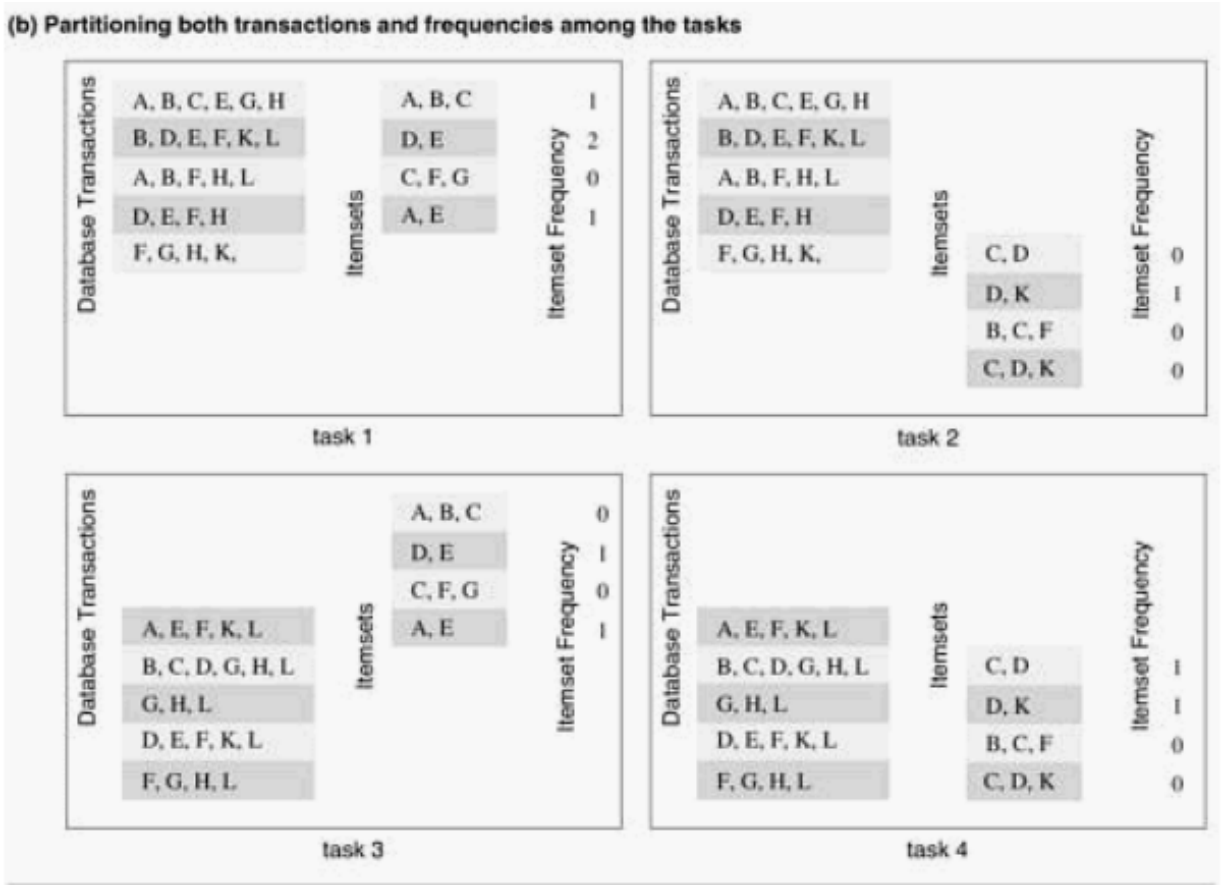


Figure 3.13. Some decompositions for computing itemset frequencies in a transaction database.

Also see word count example for intermediate data decomposition

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

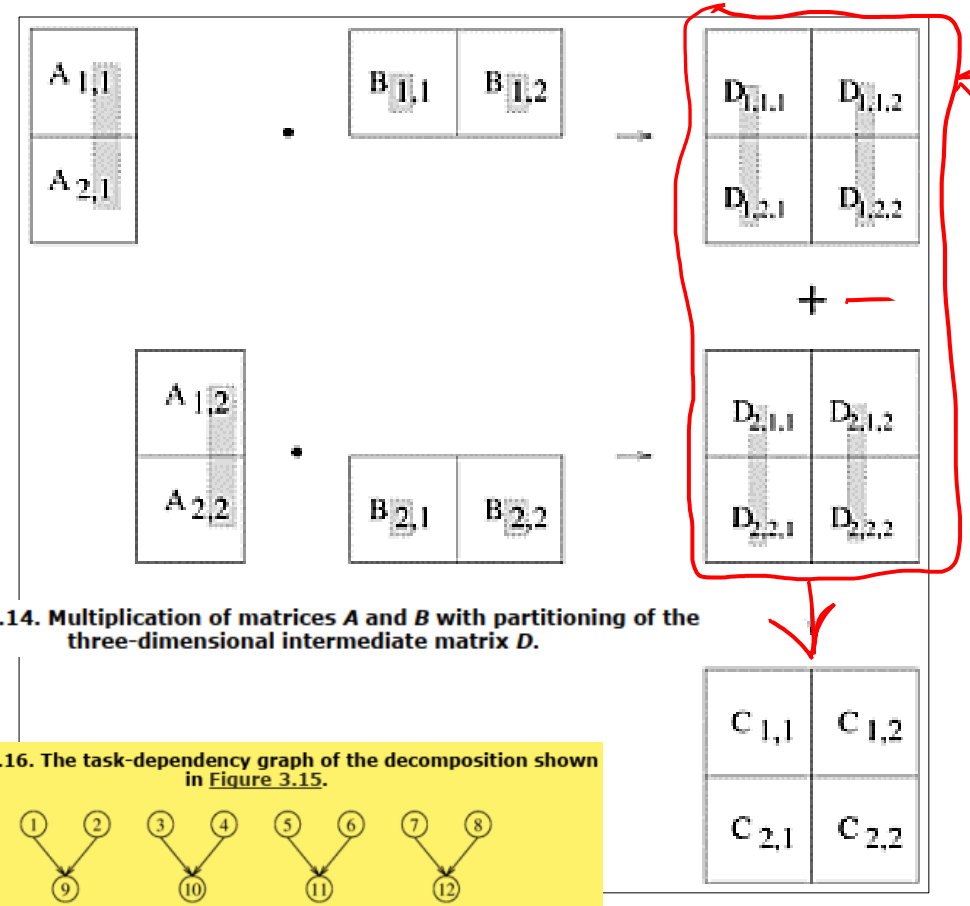


Figure 3.14. Multiplication of matrices *A* and *B* with partitioning of the three-dimensional intermediate matrix *D*.

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

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: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

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

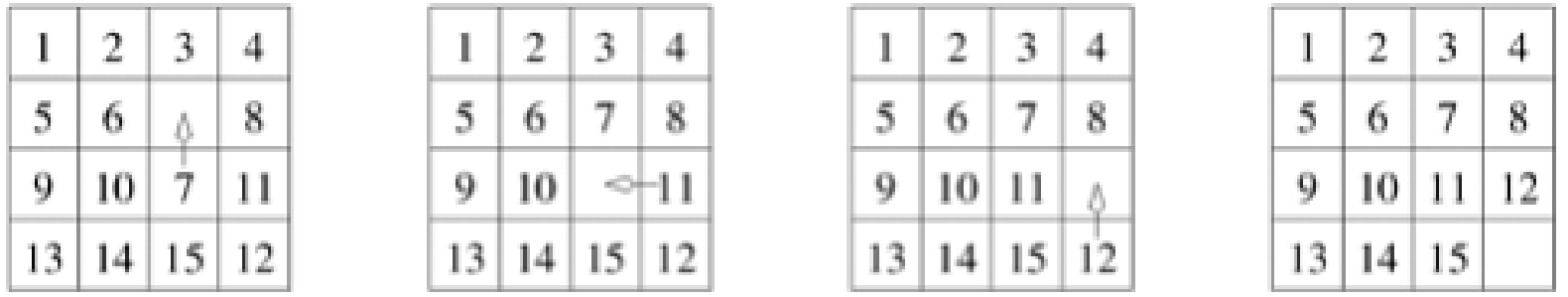
The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successor configurations are generated. A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors.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

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

Example 3.7 The 15-puzzle problem

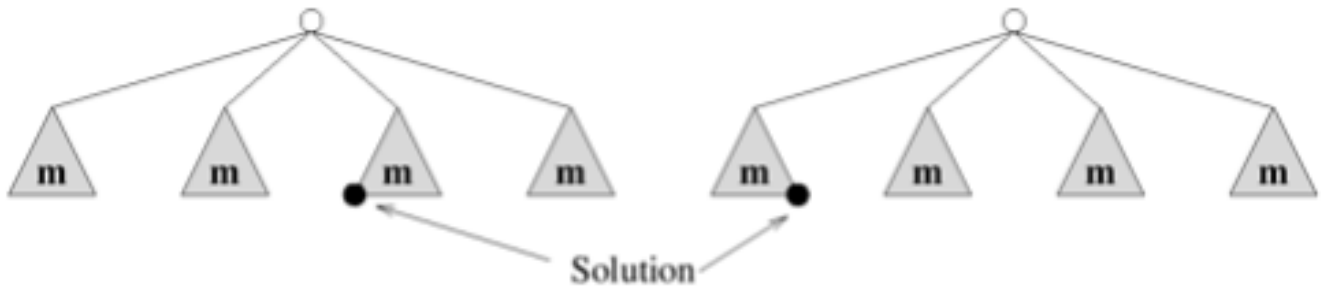
The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: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

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



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

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



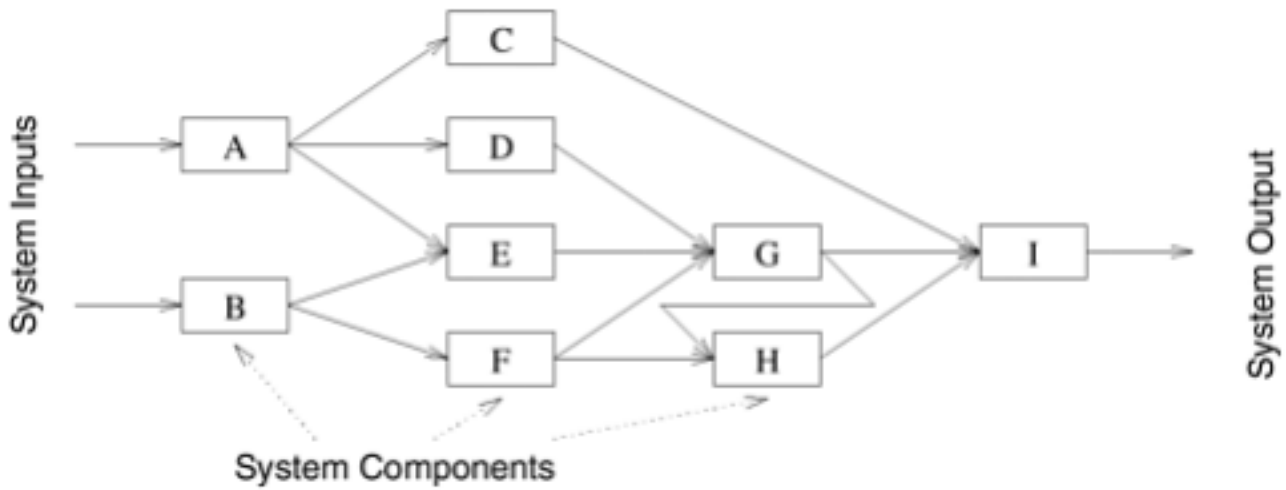
Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage. This scenario is similar to evaluating one or more of the branches of a *switch* statement in C in parallel before the input for the *switch* is available. While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel. When the input for the *switch* has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded. The parallel run time is smaller than the serial run time by the amount of time required to evaluate the condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel. However, this parallel formulation of a switch guarantees at least some wasteful computation. In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others. In this case, only the most promising branch is taken up a task in parallel with the preceding computation. In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.

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

Example 3.8 Parallel discrete event simulation

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

Figure 3.20. A simple network for discrete event simulation.



The problem of simulating a sequence of input jobs on the network described in Example 3.8 appears inherently sequential because the input of a typical component is the output of another. However, we can define speculative tasks that start simulating a subpart of the network, each assuming one of several possible inputs to that stage. When an actual input to a certain stage becomes available (as a result of the completion of another selector task from a previous stage), then all or part of the work required to simulate this input would have already been finished if the speculation was correct, or the simulation of this stage is restarted with the most recent correct input if the speculation was incorrect.

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