MODULE 5: PARALLEL DECOMPOSITION
WEEK 5

Learning Outcomes:

After completing this course you are expected to demonstrate the following:

1. Highlights and tackles the different types of parallel decomposition as well as the need of decomposition in communication and coordination, synchronization, independence and partitioning.

A. Engage

Trivia

Figure 5.0 CUDA NVIDIA makes CUDA Version 9.0

Developers who were excited by Nvidia's May announcement of an upcoming CUDA release can finally hop over to the company's dev portal to download version 11 of the parallel computing platform and programming model for GPUs. The number of those actually able to make the most of CUDA 11 seems to be comparatively small, given that its most notable features can be subsumed under support for the newest generation of Nvidia GPUs. The A100 is one example, built with the new Ampere architecture that should now work well with CUDA. It was developed to help compute some of the more complex tasks that can be found in the realms of AI, data analytics, and high-performance computing and is also central to the company's data centre platform. Amongst other things it is fitted with more streaming multiprocessors, faster memory, and special hardware like Tensor Cores, video decoder units, a JPEG decoder, and optical flow accelerators. - ***towardsdatascience.com.***

B. Explore

Presentation Title: **Parallel Algorithm Design and Decomposition**

Module Presentation Filename**: Week 5 -Parallel Algorithm Design and Decomposition**

C. Explain

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. In this section, we describe some commonly used decomposition techniques for achieving concurrency. This is not an exhaustive set of possible decomposition techniques. Also, a given decomposition is not always guaranteed to lead to the best parallel algorithm for a given problem. Despite these shortcomings, the decomposition techniques described in this section often provide a good starting point for many problems and one or a combination of these techniques can be used to obtain effective decompositions for a large variety of problems.

D. Elaborate

Decomposition Techniques

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

Recursive Decomposition

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

Example 1.0 Quicksort

Consider the problem of sorting a sequence A of n elements using the commonly used quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element x and then partitions the sequence A into two subsequences A0 and A1 such that all the elements in A0 are smaller than x and all the elements in A1 are greater than or equal to x. This partitioning step forms the divide step of the algorithm. Each one of the subsequences A0 and A1 is sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. This is illustrated in Figure 5.1 for a sequence of 12 numbers. The recursion terminates when each subsequence contains only a single element.

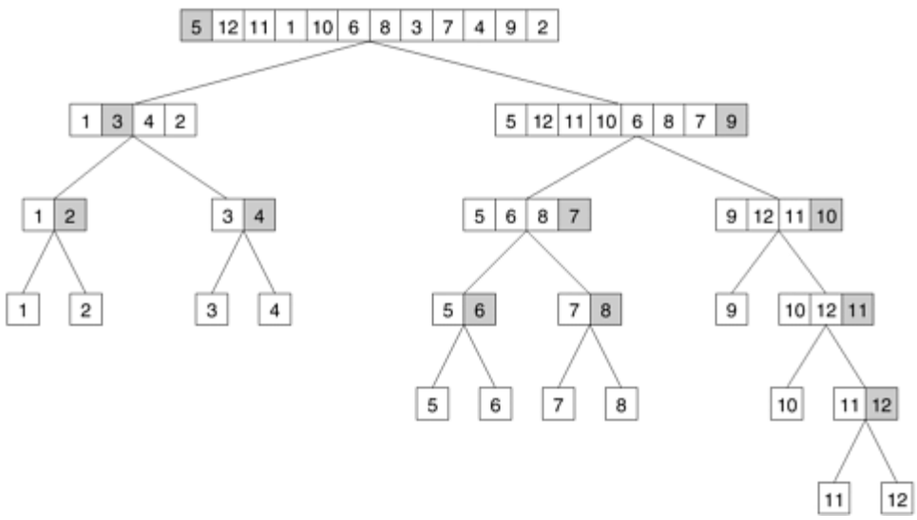

Figure 5.1 The quicksort task dependency graph based on recursive decomposition for sorting a sequence of 12 numbers

In Figure 5.1 we define a task as the work of partitioning a given subsequence. Therefore, Figure 3.8 also represents the task graph for the problem. Initially, there is only one sequence (i.e., the root of the tree), and we can use only a single process to partition it. The completion of the root task results in two subsequences (A0 and A1, corresponding to the two nodes at the first level of the tree) and each one can be partitioned in parallel. Similarly, the concurrency continues to increase as we move down the tree.

Sometimes, it is possible to restructure a computation to make it amenable to recursive decomposition even if the commonly used algorithm for the problem is not based on the divide-and-conquer strategy. For example, consider the problem of finding the minimum element in an unordered sequence A of n elements. The serial algorithm for solving this problem scans the entire sequence A, recording at each step the minimum element found so far as illustrated in Algorithm 5.1. It is easy to see that this serial algorithm exhibits no concurrency.

Algorithm 5.1 A serial program for finding the minimum in an array of numbers A of length n.

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

Once we restructure this computation as a divide-and-conquer algorithm, we can use recursive decomposition to extract concurrency. Algorithm 5.1 is a divide-and-conquer algorithm for finding the minimum element in an array. In this algorithm, we split the sequence A into two subsequences, each of size n/2, and we find the minimum for each of these subsequences by performing a recursive call. Figure 5.2 illustrates such a task-dependency graph for finding the minimum of eight numbers where each task is assigned the work of finding the minimum of two numbers.

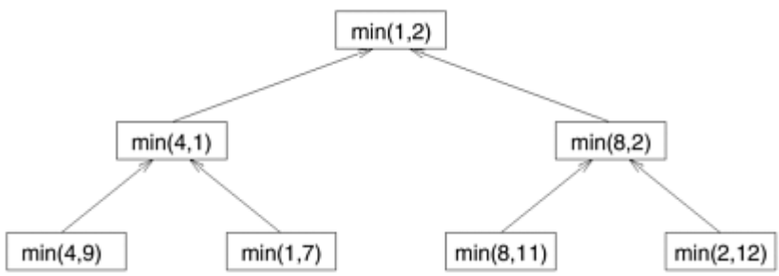

Figure 5.2 The task dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers

Algorithm 5.2 A recursive program for finding the minimum in an array of numbers A of length n.

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

```
12.     endelse;
13.  endelse;
14.  return min;
15.  end RECURSIVE_MIN
```

Data Decomposition

Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks.

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

Partitioning Output Data In many Computations

Each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output. We introduce the problem of matrix-multiplication in Example 3.5 to illustrate a decomposition based on partitioning output data.

Example 5.1 Matrix multiplication

Consider the problem of multiplying two n x n matrices A and B to yield a matrix C. Figure 5.3 shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting each dimension of the matrix into half. The four submatrices of C, roughly of size n/2 x n/2 each, are then independently computed by four tasks as the sums of the appropriate products of submatrices of A and B.

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

(a)

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

(b)

Figure 5.3 (a) Partitioning of input and output matrices into 2 x 2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a)

Most matrix algorithms, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations. In such a formulation, the matrix is viewed as composed of blocks or submatrices and the scalar arithmetic operations on its elements are replaced by the equivalent matrix operations on the blocks.

The decomposition shown in Figure 5.3is based on partitioning the output matrix C into four submatrices and each of the four tasks computes one of these submatrices. The reader must

note that data-decomposition is distinct from the decomposition of the computation into tasks. Although the two are often related and the former often aids the latter, a given data-decomposition does not result in a unique decomposition into tasks.

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 5.4 Two examples of decomposition of matrix multiplication into eight tasks

We now introduce another example to illustrate decompositions based on data partitioning. Example 5.2 describes the problem of computing the frequency of a set of itemsets in a transaction database, which can be decomposed based on the partitioning of output data.

Example 5.2 Computing frequencies of item sets 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.

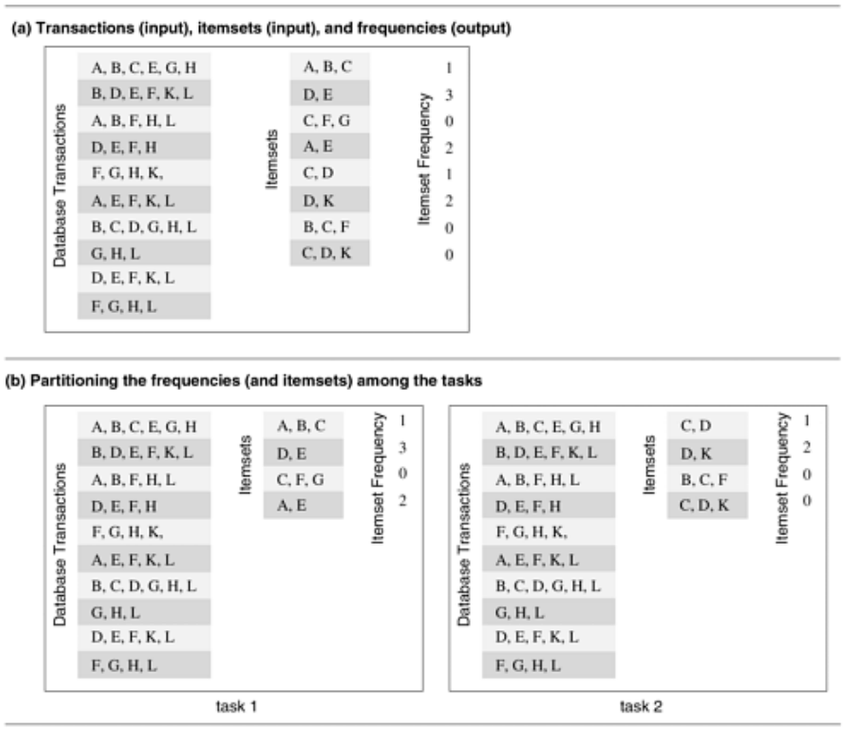

Figure 5.5 Computing itemset frequencies in a transaction database

Figure 5.5 (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 5.5 (b) is to have each task independently compute the subset of frequencies assigned to it.

The problem of computing the frequency of a set of itemsets in a transaction database described in Example 5.2 can also be decomposed based on a partitioning of input data. Figure 5.6 (a) shows 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.

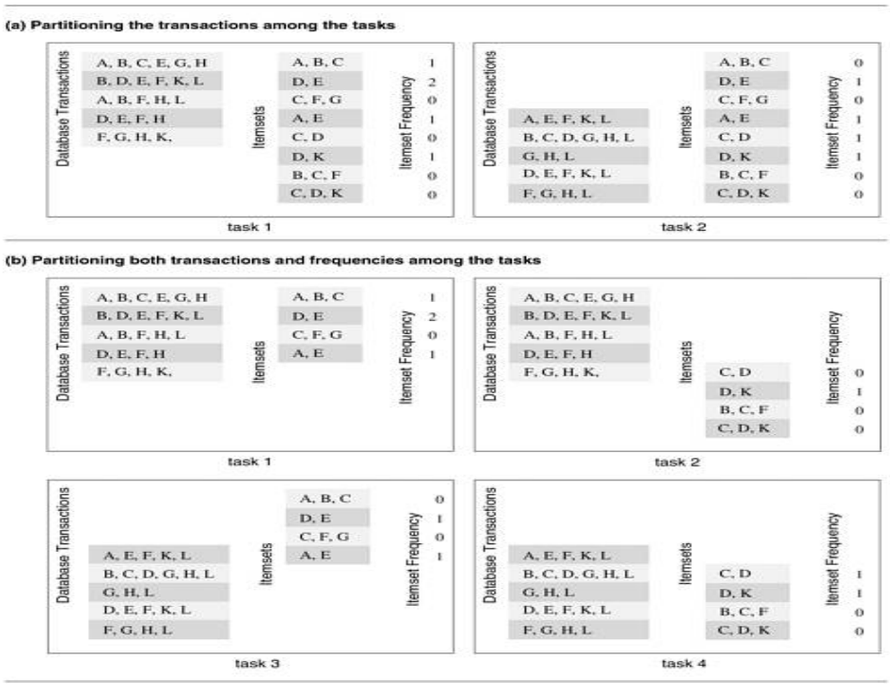

Figure 5.6 Some decompositions for computing itemset frequencies in a transaction database.

Partitioning both Input and Output Data In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency. For example, consider the 4-way decomposition shown in Figure 5.6 for computing itemset frequencies.

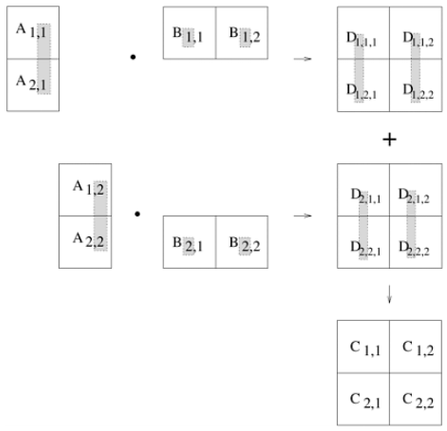

Figure 5.7 Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D

A partitioning of the intermediate matrix D induces decomposition into eight tasks. Figure 5.7 shows this decomposition. After the multiplication phase, a relatively inexpensive matrix addition step can compute the result matrix C. All submatrices D*,i, j with the same second and third dimensions i and j are added to yield Ci, j. The eight tasks numbered 1 through 8 in Figure 5.7 perform O(n3/8) work each in multiplying n/2 x n/2 submatrices of A and B. Then, four tasks numbered 9 through 12 spend O(n2/4) time each in adding the appropriate n/2 x n/2 submatrices of the intermediate matrix D to yield the final result matrix C. Figure 18 shows the task-dependency graph corresponding to the decomposition shown in Figure 5.7.

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

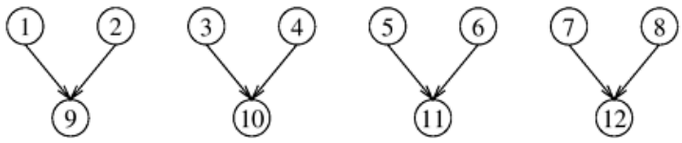

Figure 5.9 The task-dependency graph of the decomposition shown in Figure 5.8

Note that all elements of D are computed implicitly in the original decomposition shown in Figure 5.4, but are not explicitly stored. By restructuring the original algorithm and by explicitly storing D, we have been able to devise decomposition with higher concurrency. This, however, has been achieved at the cost of extra aggregate memory usage.

Exploratory Decomposition

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

Example 5.3 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 5.10 illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration.

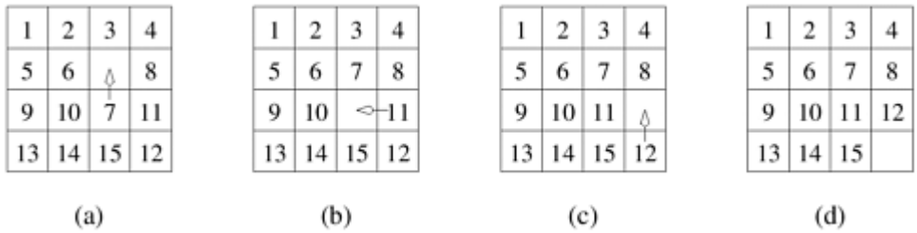

Figure 5.10 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration

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

One method for solving this problem in parallel is as follows. First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes (i.e., configurations of the 15-puzzle). Now each node is assigned to a task to explore further until at least one of them finds a solution. As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches. Figure 5.11 illustrates one such decomposition into four tasks in which task 4 finds the solution.

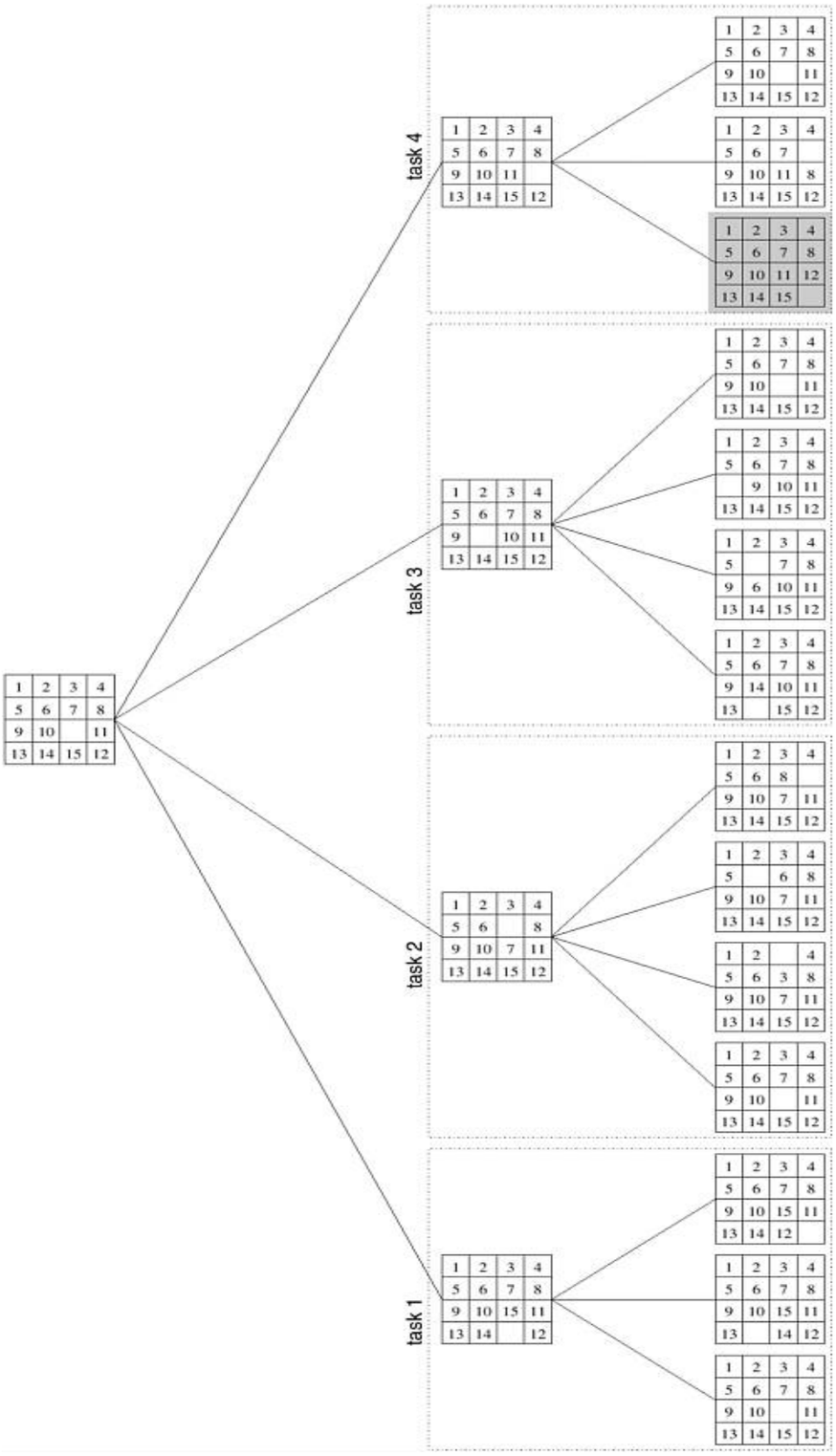

Figure 5.11 The states generated by an instance of the 15-puzzle problem

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.

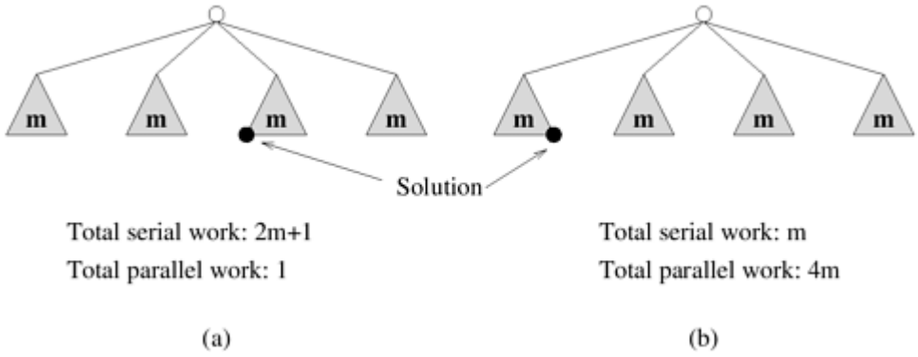

Figure 5.12 An illustration of anomalous speedups resulting from exploratory decomposition

Speculative Decomposition

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

Example 5.4 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. There are 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.

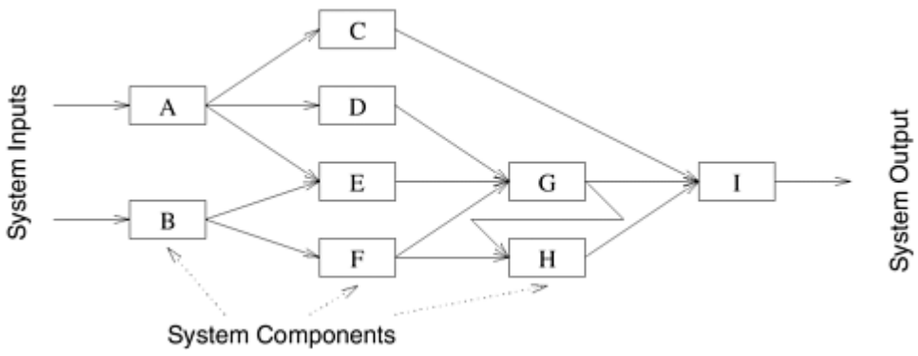

Figure 5.13 Simple network for discrete event simulation

The problem of simulating a sequence of input jobs on the network 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.

Hybrid Decompositions

So far we have discussed a number of decomposition methods that can be used to derive concurrent formulations of many algorithms. An efficient decomposition would partition the input into P roughly equal parts and have each task compute the minimum of the sequence assigned to it. The final result can be obtained by finding the minimum of the P intermediate results by using the recursive decomposition shown in Figure 5.14.

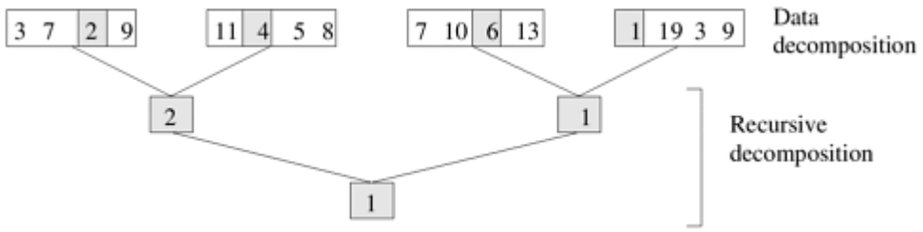

Figure 5.14 Hybrid decomposition for finding the minimum of an array of size 16 using four tasks

As another example of an application of hybrid decomposition, consider performing quicksort in parallel. In Example 5.3, we used a recursive decomposition to derive a concurrent formulation of quicksort. This formulation results in O(n) tasks for the problem of sorting a sequence of size n. But due to the dependencies among these tasks and due to uneven sizes of the tasks, the effective concurrency is quite limited..

E. Evaluate

ASSESSMENT:
Instruction: Answer the questions below using the Answer Sheet (AS) provided in this module.

CONTENT FOR ASSESSMENT:
(2-points each)

1. A method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy.
2. A powerful and commonly used method for deriving concurrency in algorithms that operates on large data structures.
3. It is used to decompose problems whose underlying computations correspond to a search of a space for solutions.
4. It is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it.
5. It is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages.

References:

1. *Decomposition Techniques, http://parallelcomp.uw.hu/ch03lev1sec5.html*
2. *Introduction to Parallel Computing UsingAdvanced Architectures and Algorithms, D. R. EMERSON, Head of Computational Engineering, Daresbury Laboratory, Keckwick Lane. Daresbury, Warrington WA4 4AD, Cheshire, United Kingdom*

Facilitated By:		
Name	:	
MS Teams Account (email)	:	
Smart Phone Number	:	