

# 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)

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.

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

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		

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

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 1

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		

task 2

**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.

MPI

cluster processing using Map-Reduce

Decomposition type ①

Partitioning the output into two parts.

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.**

**(a) Partitioning the transactions among the tasks**

Decomposition type 2

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

task 1

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

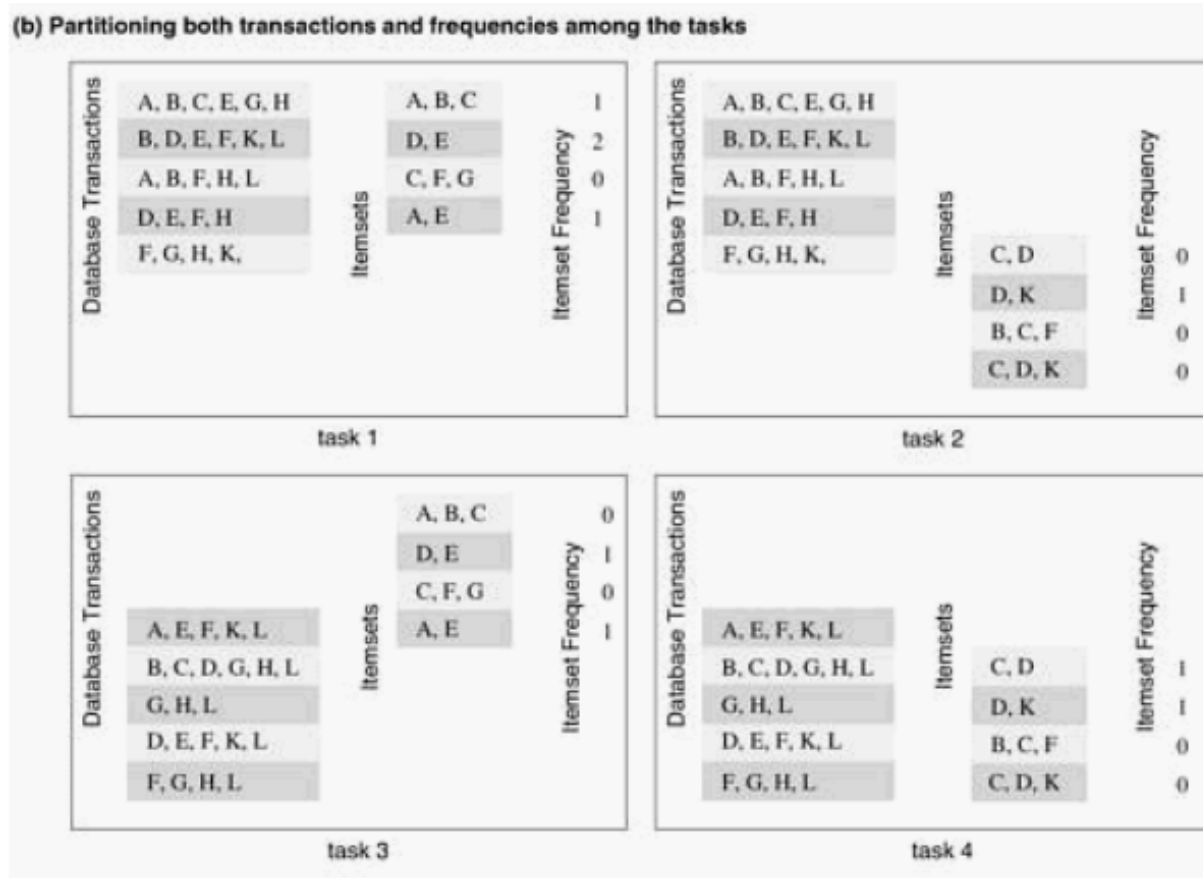
task 2

input data is distributed among tasks running on different nodes.

How Intaiz SuperMart takes its purchasing decision based 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~~ 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.

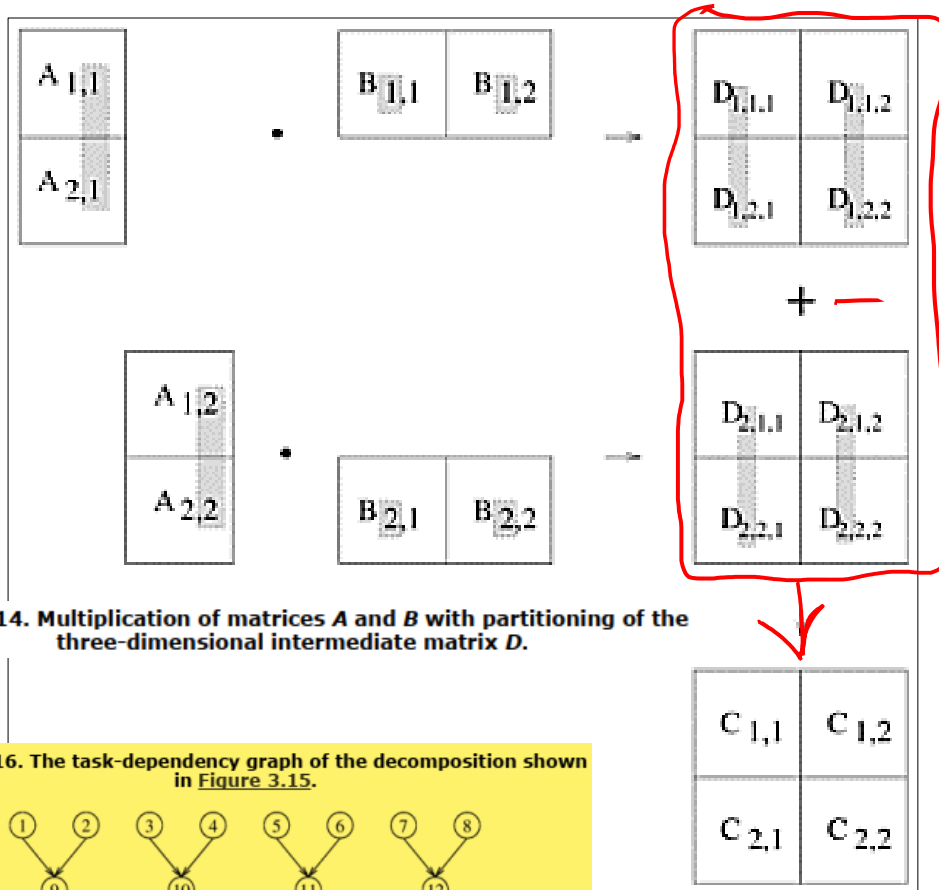
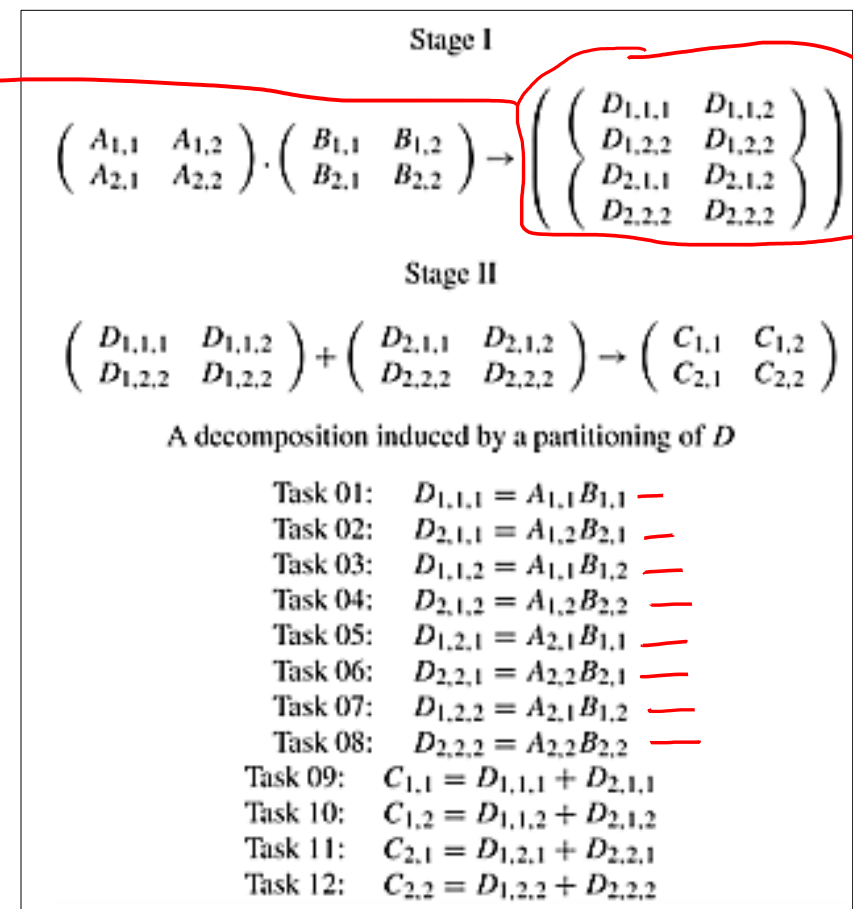


Figure 3.14. Multiplication of matrices  $A$  and  $B$  with partitioning of the three-dimensional intermediate matrix  $D$ .

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



## Input Data Decomposition

---

- **Generally applicable if each output can be naturally computed as a function of the input.**
- **In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).**
- **A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.**

# Output vs. Input Data Decompositions

---

From the previous example, the following observations can be made:

- If only the output is decomposed and the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the input database is also partitioned (for scalability), it induces a computation mapping in which each task computes partial counts, and additional tasks are used to aggregate the counts.

Further Explanation Item-set Frequency Example

## From Data Decompositions to Task Mappings: Owner Computes Rule

---

- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.



## Exploratory Decomposition: Example

**A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).**

1	2	3	4		1	2	3	4		1	2	3	4		1	2	3	4
5	6	7	8		5	6	7	8		5	6	7	8		5	6	7	8
9	10	11	12		9	10	11	12		9	10	11	12		9	10	11	12
13	14	15	12		13	14	15	12		13	14	15	12		13	14	15	
(a)					(b)					(c)					(d)			

**Of course, the problem of computing the solution, in general, is much more difficult than in this simple example.**

# Exploratory Decomposition: Example

The state space can be explored by generating various successor states of the current state and viewing them as independent tasks.

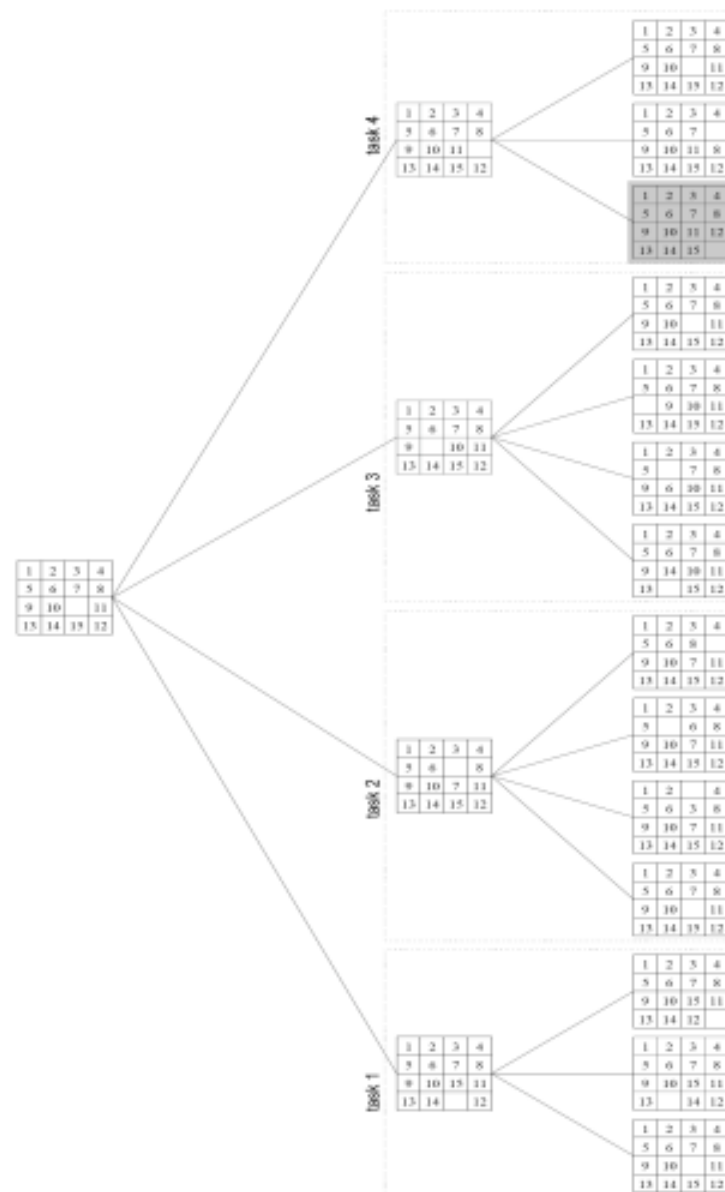
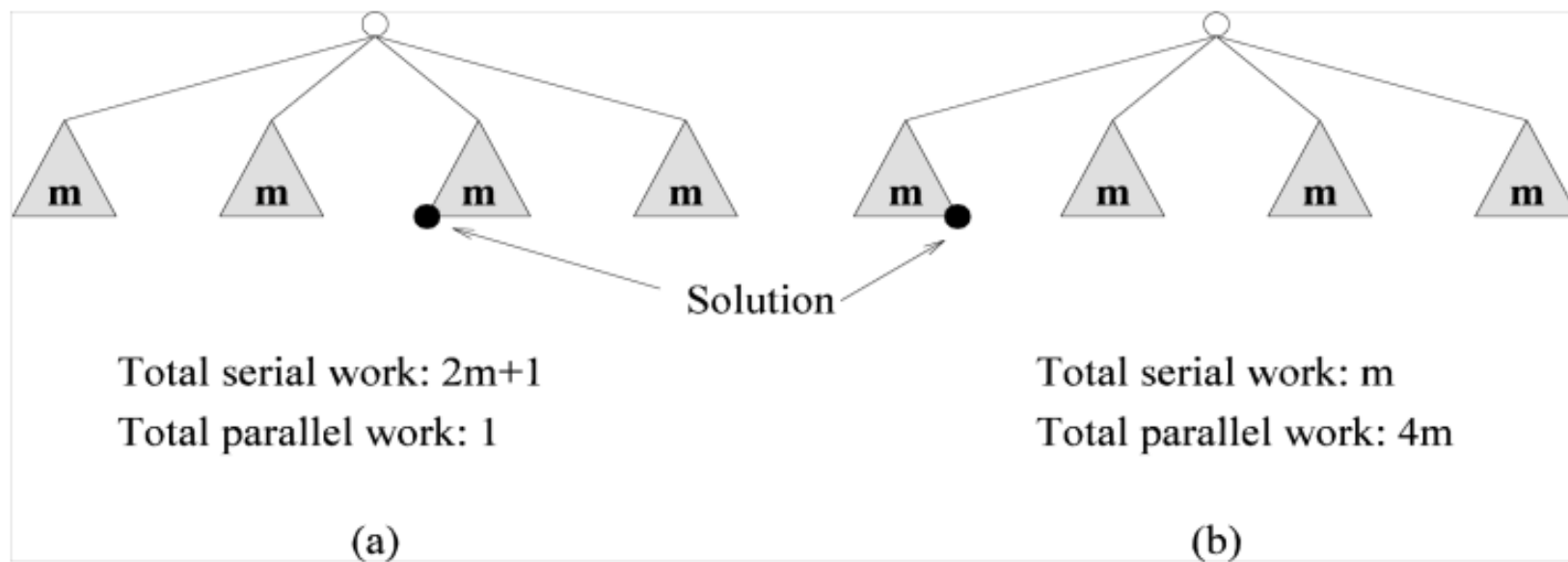


Figure 3.18 The states generated by an instance of the 15-puzzle problem.



## Exploratory Decomposition: Anomalous Speedups

- In many instances of parallel exploratory decomposition, unfinished tasks can be terminated when the first solution is found
- This can result in “anomalous” super- or sub-linear speedups relative to serial execution.



**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.**

1	2	3	4
5	6	△	8
9	10	7	11
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10	◁	11
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	△
13	14	15	12

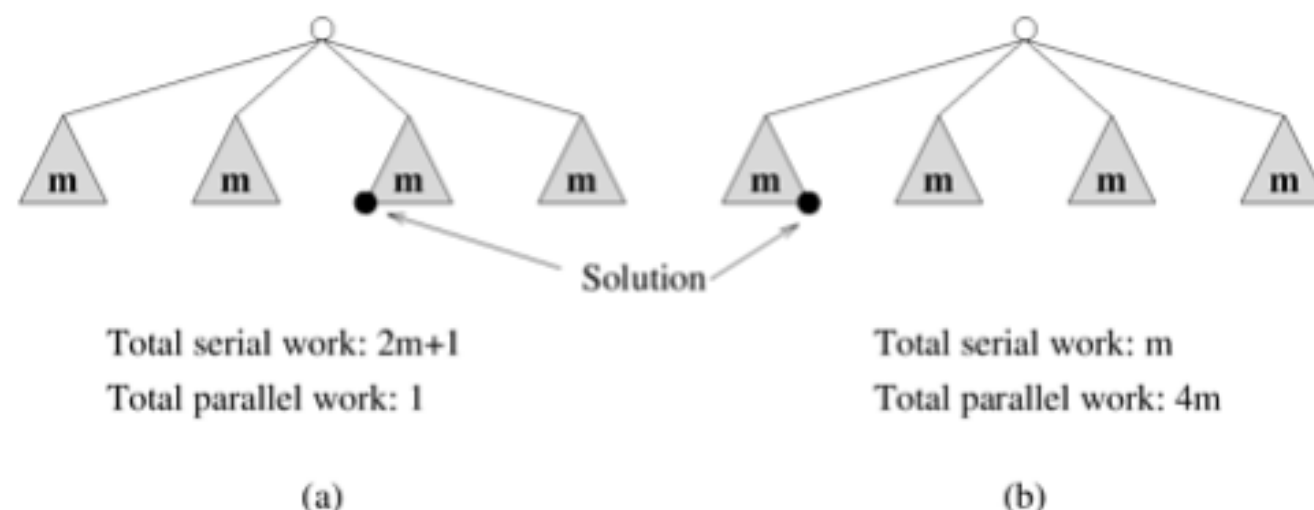
(c)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(d)

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

---

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications: conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies, and, optimistic approaches, which schedule tasks even when they may potentially be erroneous.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.
- Parallel Discrete Event Simulation (Example 3.8) is a motivating example for optimistic approaches

# Speculative Decomposition

- **Switch statement in a program:** We wait to know the value of the expression and execute only the corresponding case.
- In speculative decomposition *we execute some or all of the cases in advance.*
- When the value of the expression is known, *we keep only the results from the computation to be executed in that case.*
- The *gain in performance comes from anticipating the possible computations.*

Sequential version	Parallel version
<pre>compute expr; switch (expr) {   case 1:     compute a1;     break;   case 2:     compute a2;     break;   case 3:     compute a3;     break;... }</pre>	<pre>Slave(i) {   compute ai;   Wait(request);   if (request)     Send(ai, 0); } Master() {   compute expr;   swithch (expr) {     case 1:       Send(request, 1);       Receive(a1, i);     ...   } }</pre>

The difference with the exploratory decomposition is that we can compute the possible states before the next move is performed.



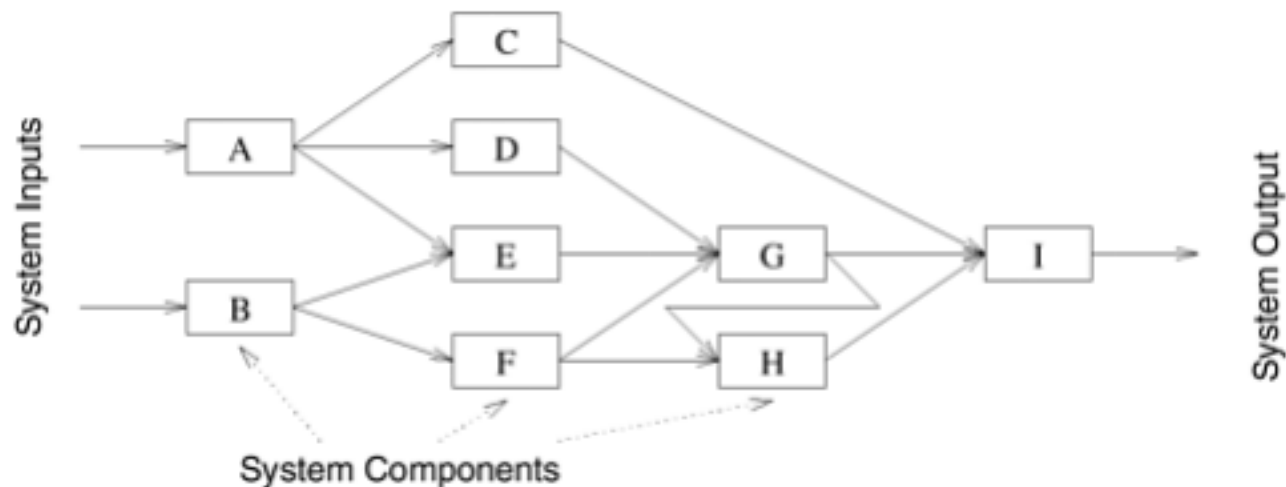
**Speculative decomposition** is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, 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 is 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.

# Exploratory vs Speculative Decomposition

- In speculative decomposition, the input at a branch leading to multiple parallel tasks is unknown.
- In exploratory decomposition, the output of the multiple tasks originating at a branch is unknown.

# Parallel Algorithm Models

- Data parallel
  - Each task performs similar operations on different data
  - Typically statically map tasks to processes
- Task graph
  - Use task dependency graph to promote locality or reduce interactions
- Master-slave
  - One or more master processes generating tasks
  - Allocate tasks to slave processes
  - Allocation may be static or dynamic
- Pipeline/producer-consumer
  - Pass a stream of data through a sequence of processes
  - Each performs some operation on it
- Hybrid
  - Apply multiple models hierarchically, or apply multiple models in sequence to different phases

### 3.6.1 The Data-Parallel Model

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

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

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



### 3.6.2 The Task Graph Model

As discussed in [Section 3.1](#), the computations in any parallel algorithm can be viewed as a task-dependency graph. The task-dependency graph may be either trivial, as in the case of matrix multiplication, or nontrivial (Problem 3.5). However, in certain parallel algorithms, the task-dependency graph is explicitly used in mapping. In the *task graph model*, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs. This model is typically employed to solve problems in which the amount of data associated with the tasks is large relative to the amount of computation associated with them. Usually, tasks are mapped statically to help optimize the cost of data movement among tasks. Sometimes a decentralized dynamic mapping may be used, but even then, the mapping uses the information about the task-dependency graph structure and the interaction pattern of tasks to minimize interaction overhead. Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.

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

Examples of algorithms based on the task graph model include parallel quicksort ([Section 9.4.1](#)), sparse matrix factorization, and many parallel algorithms derived via divide-and-conquer decomposition. This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called *task parallelism*.

### 3.6.4 The Master-Slave Model

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

While using the master-slave model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help

### 3.6.5 The Pipeline or Producer-Consumer Model

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