CS 3006 Parallel and Distributed Computer

Fall 2022

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 # 10 – Lecture # 25, 26

27th, 29th, ??rd Rabi ul Awwal, 1444

24th, 26th, 27th October 2022

Dr. Nadeem Kafi Khan

No Lab this week

Lecture # 25 – Topics

- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Speculative Decomposition

General Ideas

- Identify the portions of code that can be done in parallel.
- Mapping the code onto multiple processes.
- Distributing the input, output, and intermediate data
- Managing the access to shared resources.
- Synchronizing the processes at various stages of the program.

Code Decomposition

- Decomposition: the operation of dividing the computation into smaller parts, some of which may be executed in parallel.
- Task: programmer-defined units of code resulting from decomposition.
- Granularity: the number / size of the tasks.
- Fine-grained decomposition: a large number of tasks
- Coarse-grained decomposition: small number of tasks.
- Degree of concurrency: the maximum number of tasks that can be executed in the same time.

Decomposition Techniques

- Recursive decomposition: used for traditional divide-and-conquer algorithms that are not easy to solve iteratively.
- Data decomposition: the data is partitioned and this <u>induces</u> a partitioning of the code in tasks.
- Functional decomposition: the functions to be performed on data are split into multiple tasks.
- Exploratory decomposition: decompose problems equivalent to a search of a space for solutions.
- Speculative decomposition: when a program may take one of many possible branches depending on results from computations preceding the choice.

Characteristics of Tasks

- Task generation:
  - static - the tasks are known in advance (data decomposition)
  - dynamic - decided at runtime (recursive decomposition)
- Task size:
  - uniform (they require approximately the same amount of time) or
  - non-uniform
  - known/not known.
- Task Interaction
  - Static: it happens at predetermined times and the set of tasks to interact with is known in advance.
  - Dynamic: the timing of the interaction or the set of tasks to interact with are unpredictable. Harder to implement.
  - Regular/irregular: it is regular if the interaction follows a pattern that can be exploited for efficiency.

Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.

Recursive Decomposition: Example

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.

Task generation is dynamic and the task size is non-uniform.

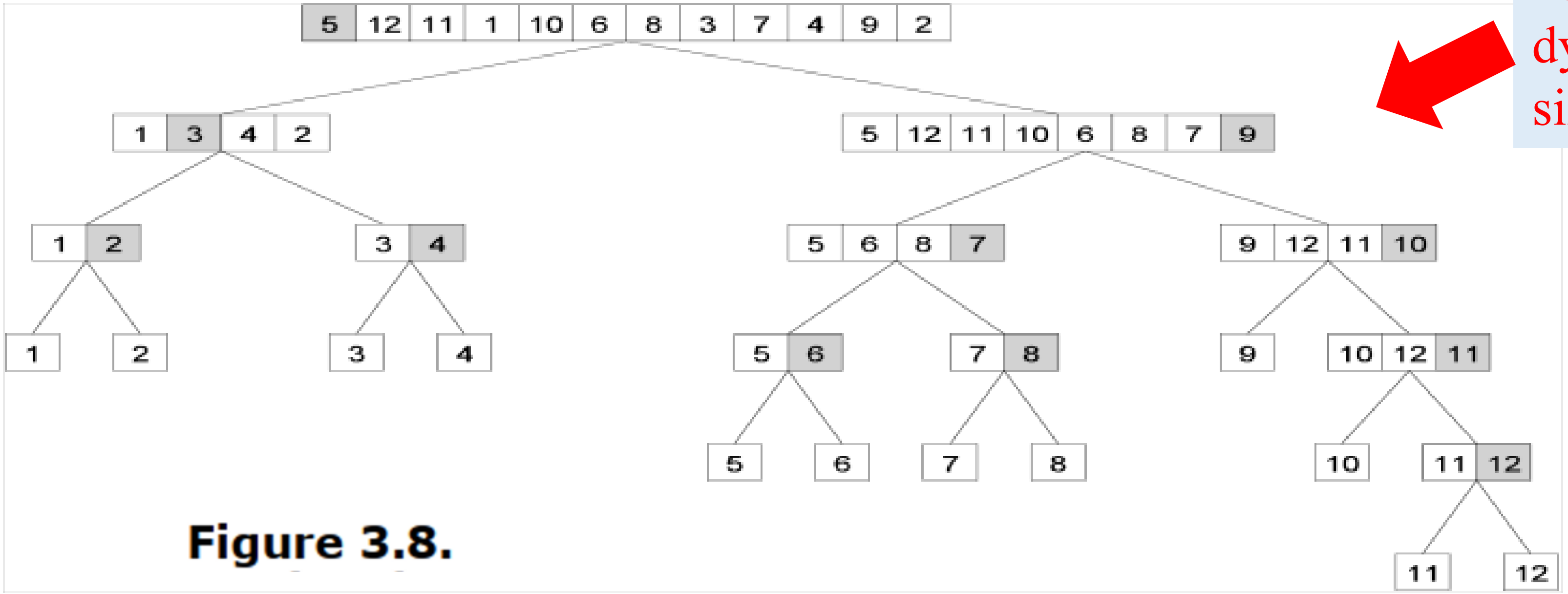


Figure 3.8.

In this example, a task represents the work of partitioning a (sub)array. Note that each subarray represents an independent subtask. This can be repeated recursively.

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

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.

Input Data Decomposition: Example

In the database counting example, the input (i.e., the transaction set) can be partitioned. This induces a task decomposition in which each task generates partial counts for all itemsets. These are combined subsequently for aggregate counts.

Partitioning the transactions among the tasks

task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,

Itemsets	Itemset Frequency
A, B, C	1
D, E	2
C, F, G	0
A, E	1
C, D	0
D, K	1
B, C, F	0
C, D, K	0

task 2

Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

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

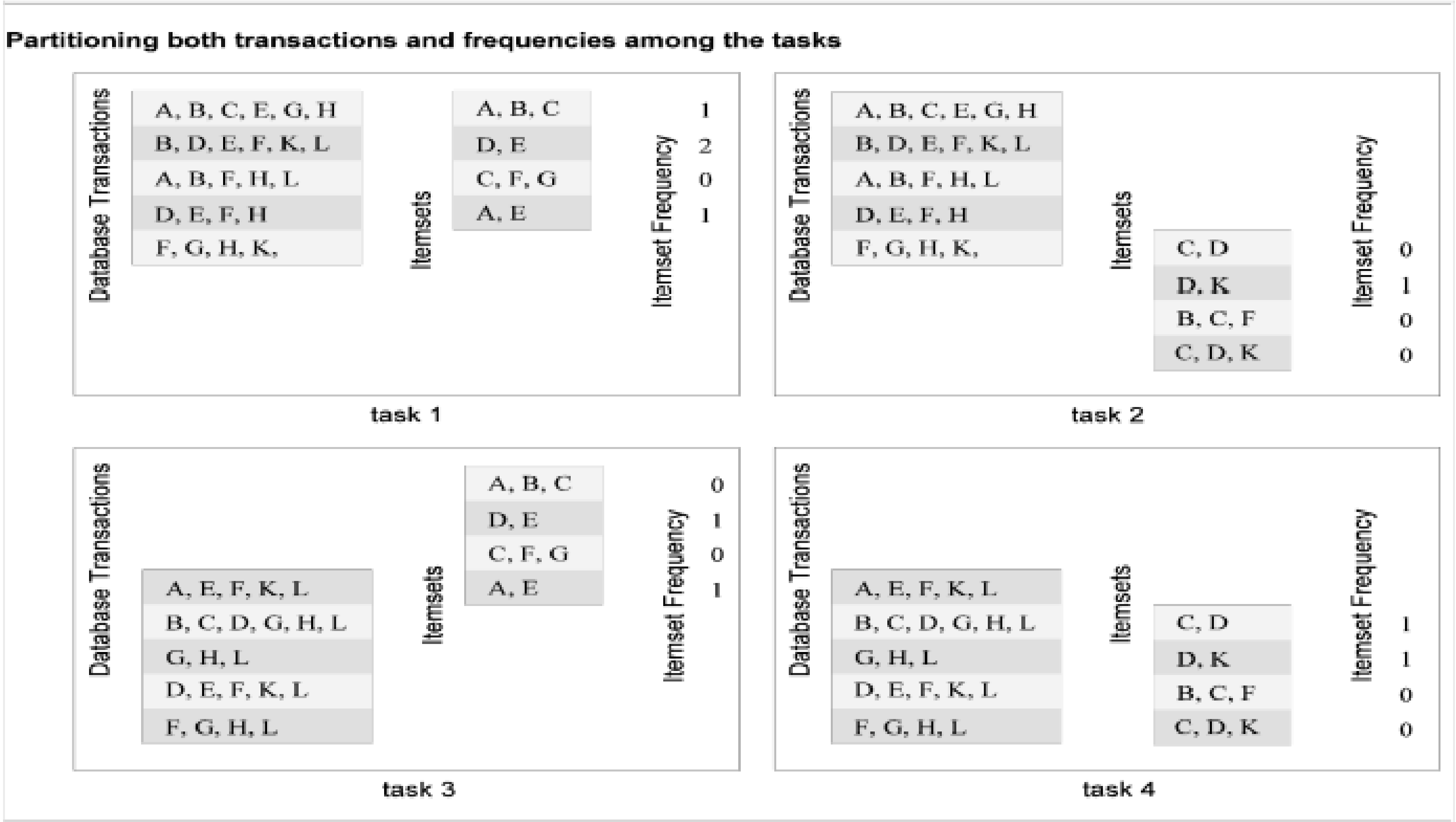
Output vs. Input Data Decompositions

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.

Combining Input *and* Output Data Decompositions

Often input and output data decomposition can be combined for a higher degree of concurrency. For the itemset counting example, the transaction set (input) and itemset counts (output) can both be decomposed as follows:



From Data Decompositions to Task Mappings: Owner Computes Rule

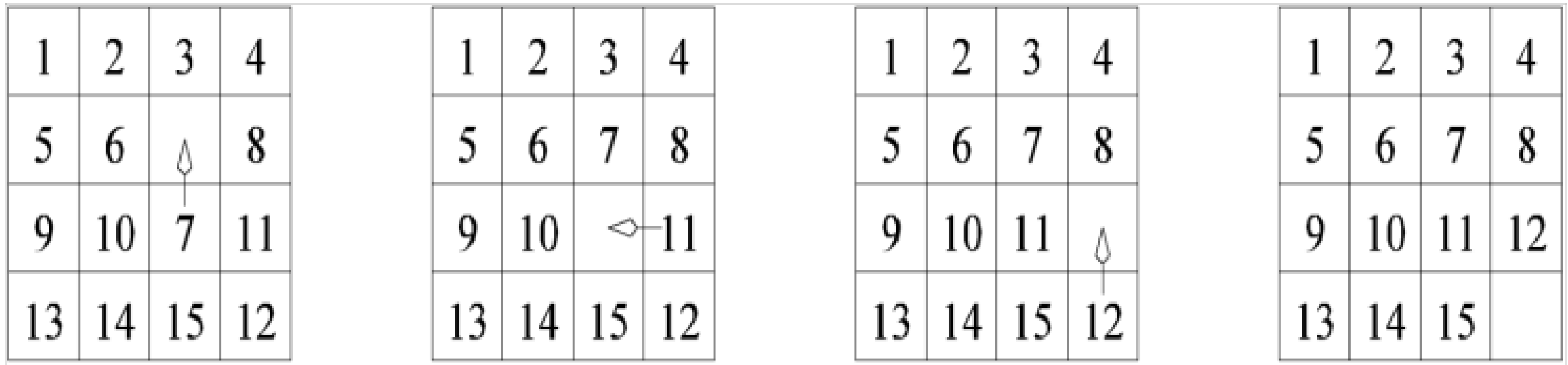
- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
-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

- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.
- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

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).



(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.

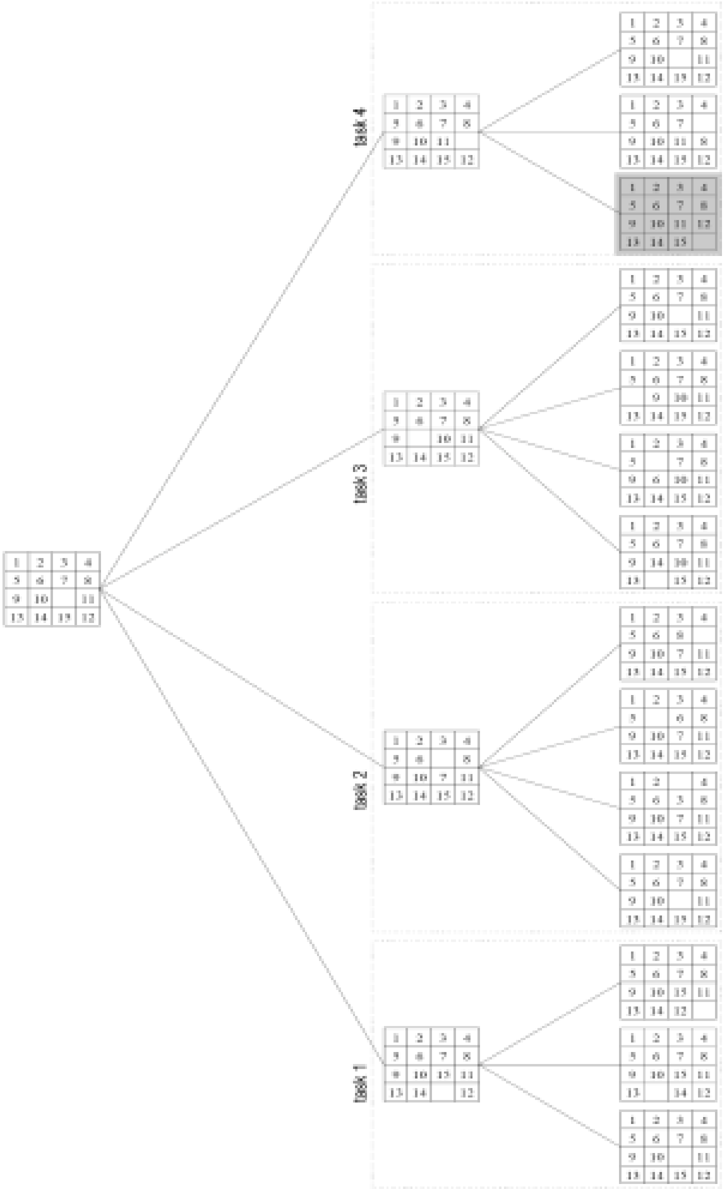
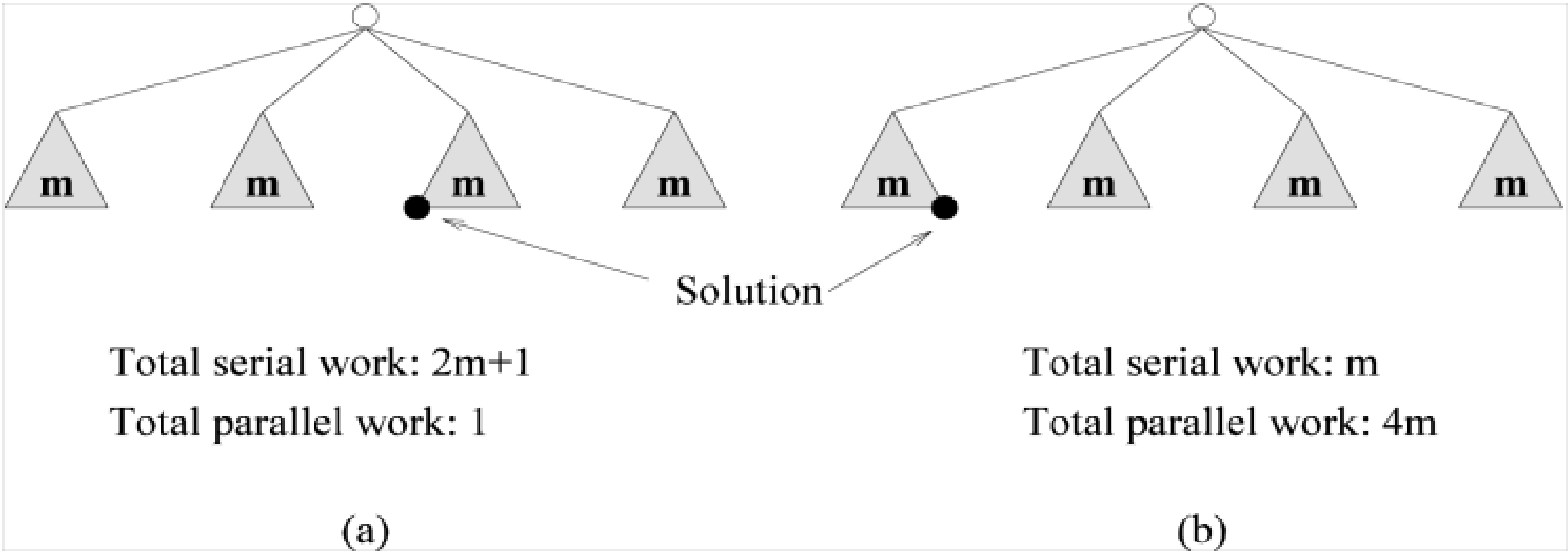


Figure 3.18 The states generated by an instance of the 15-puzzle problem.

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.



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 know, *we keep only the results from the computation to be executed in that case*.
- The *gain in performance comes from anticipating the possible computations*.

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

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

Lecture # 26 – Topics

- **Quiz # 2**