

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 # 7

Dr. Nadeem Kafi Khan

- Limits on Parallel Performance
- Task Interaction Graph
 - Introduction
 - Comparaison with Task dépendancy graph.
- Processes and Mapping
- Decomposition Techniques
 - Recursive Decomposition
 - Data Decomposition
 - Exploratory Decomposition
 - Speculative Decomposition


Limits on Parallel Performance

```
REAL A[n][n], b[n], y[n]; int i, j;  
for (i = 0; i < n; i++) {  
    sum = 0.0;  
    for (j = 0; j < n; j++)  
        sum += A[i][j] * b[j];  
    c[i] = sum;  
}
```

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.
- There is an inherent bound on how fine the granularity of a computation can be. For example, in the case of multiplying a dense matrix with a vector, there can be no more than (n^2) concurrent tasks. *what other factor would limit reaching n^2 ?*
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds. *??*

Task Interaction Graphs

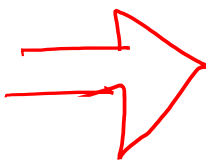
- Subtasks generally exchange data with others in a decomposition.



- For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks, they will have to communicate elements of the vector.



- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.

- 
- *Task interaction graphs* represent data dependencies,
 - *Task dependency graphs* represent control dependencies.

①
②
③
The pattern of interaction among tasks is captured by what is known as a **task-interaction graph**. The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other. 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. The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional. The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph. In the database query example discussed earlier, the task-interaction graph is the same as the task-dependency graph. We now give an example of a more interesting task-interaction graph that results from the problem of **sparse matrix-vector multiplication**.

Example 3.3 Sparse matrix-vector multiplication

Consider the problem of computing the product $y = Ab$ of a sparse $n \times n$ matrix A with a dense $n \times 1$ vector b . 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. Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros.

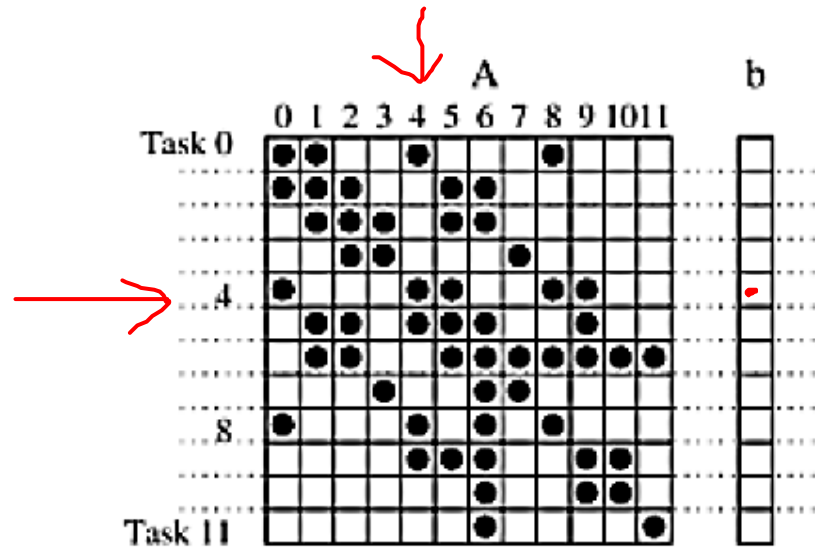
One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it. [Figure 3.6\(a\)](#) illustrates this

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 locations. In the message-passing paradigm, with the ownership of $b[i]$, Task i also inherits the responsibility of sending $b[i]$ 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 $b[0]$, $b[5]$, $b[8]$, and $b[9]$ to perform its own computation. The resulting task-interaction graph is shown in [Figure 3.6\(b\)](#). ■

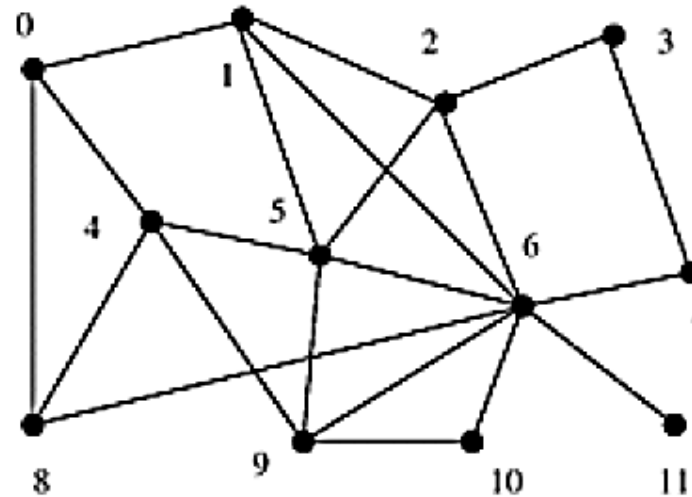
Task Interaction graph- An example

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

Figure 3.6. A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i,j].b[j]$.



(a)



(b)

Task #4 Needs
 $b[0], b[4], b[5],$
 $b[8], b[9]$
 It already owns
 $b[4]$ so it needs
 to fetch others from
 respective task nodes.

Processes and Mapping

- **Mapping:** the mechanism by which tasks are assigned to processes for execution.
- **Process:** a logic computing agent that performs tasks, which is an abstract entity that uses the code and data corresponding to a task to produce the output of that task.
- Why use processes rather than processors?
 - We rely on OS to map processes to physical processors.
 - We can aggregate tasks into a process

Criteria of Mapping

1. Maximize the use of concurrency by mapping independent tasks onto different processes
2. Minimize the total completion time by making sure that processes are available to execute the tasks on critical path as soon as such tasks become executable
3. Minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process.

Basis for Choosing Mapping

Task-dependency graph

 Makes sure the max. concurrency

Task-interaction graph

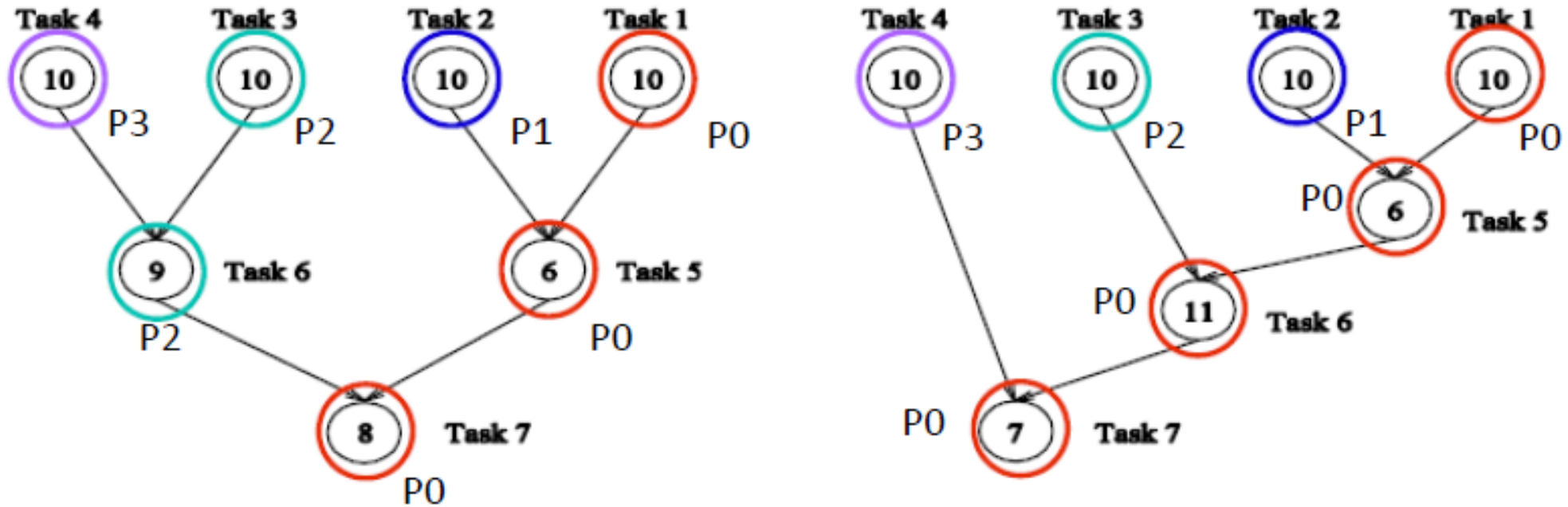
 Minimum communication.

These criteria often conflict with each other.

For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

- Finding a balance that optimizes the overall parallel performance is the key to a successful parallel algorithm.
- Therefore, mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel algorithm is.

Example: Mapping Database Query to Processes



No two nodes in a level have dependencies, therefore, single level tasks are assigned to different processes.

- 4 processes can be used in total since the max. concurrency is 4.
- Assign all tasks within a level to different processes.

Decomposition Techniques

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

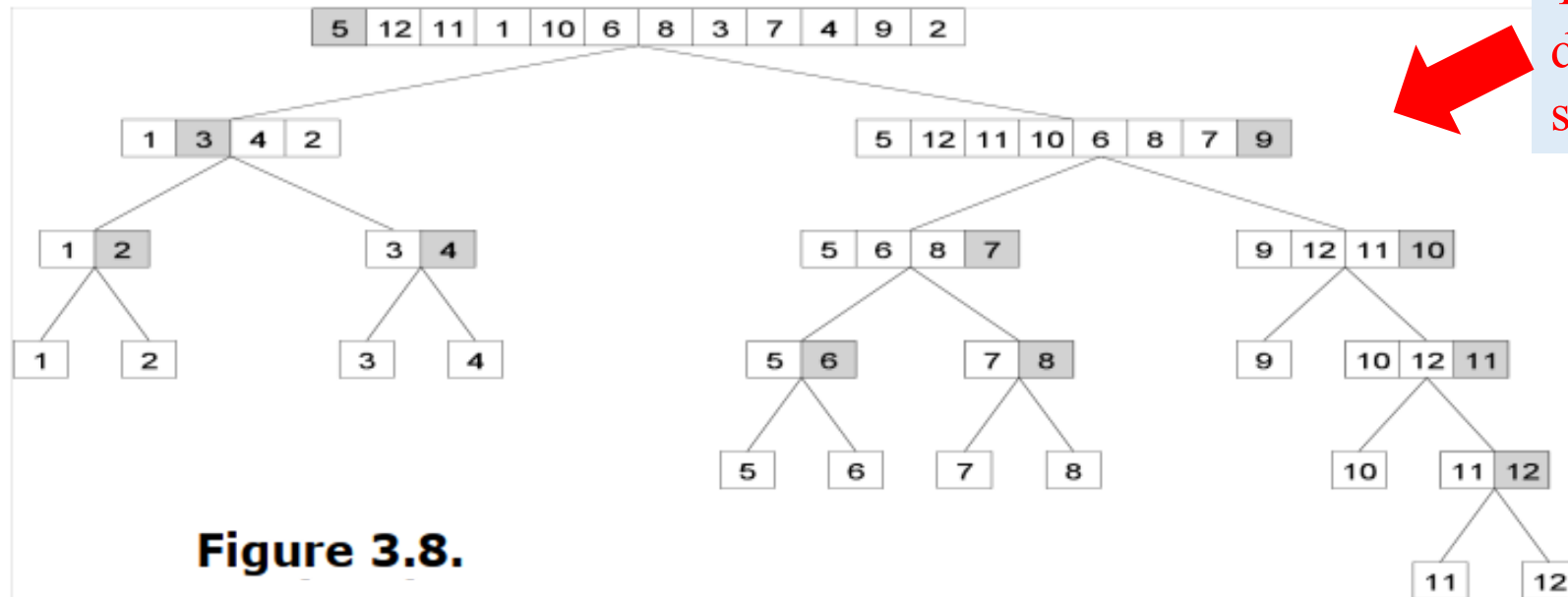
- **recursive decomposition**
- **data decomposition**
- **exploratory decomposition**
- **speculative decomposition**

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.

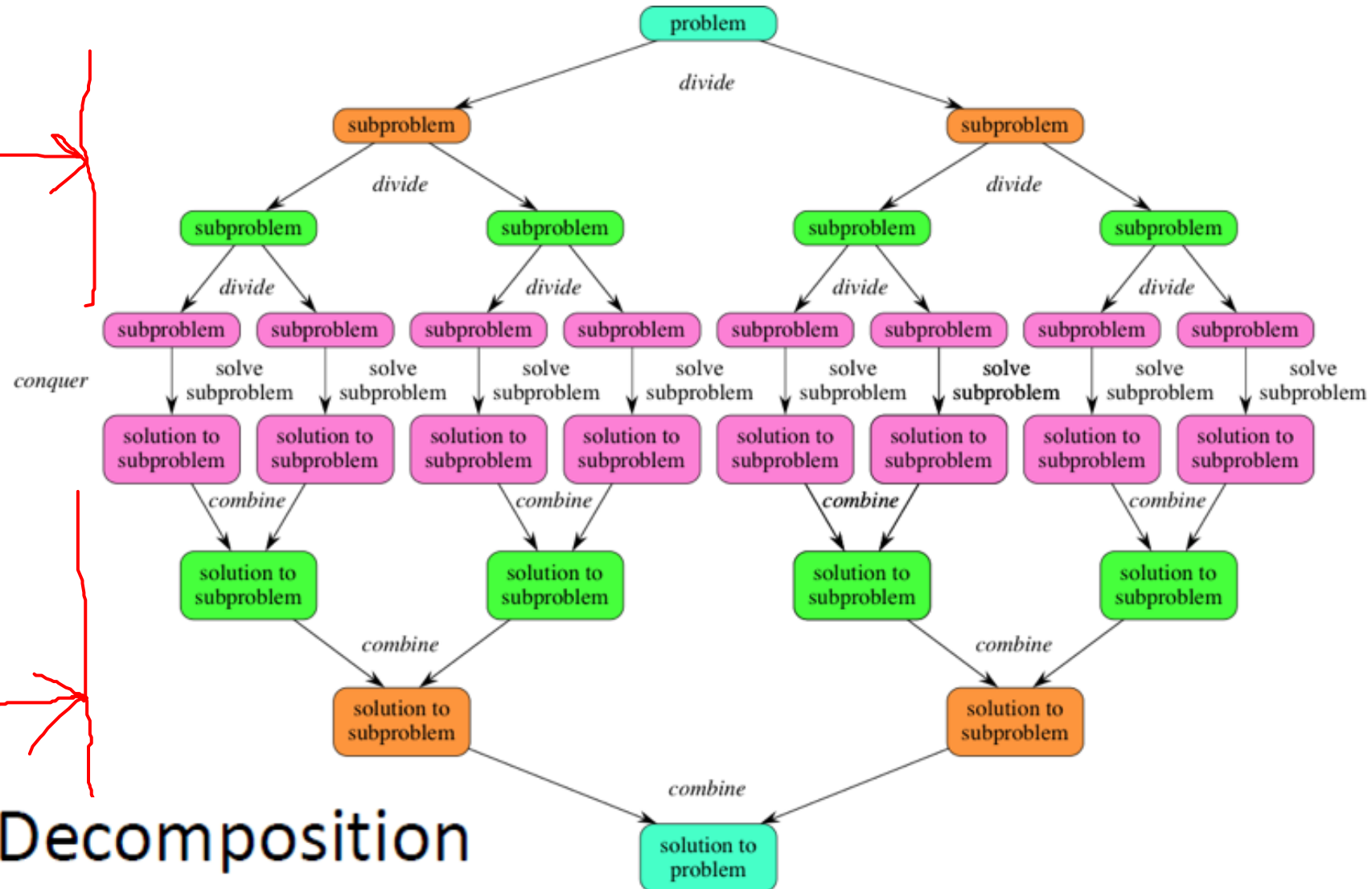


Task generation is dynamic and the task size is non-uniform.

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.

You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



Recursive Decomposition

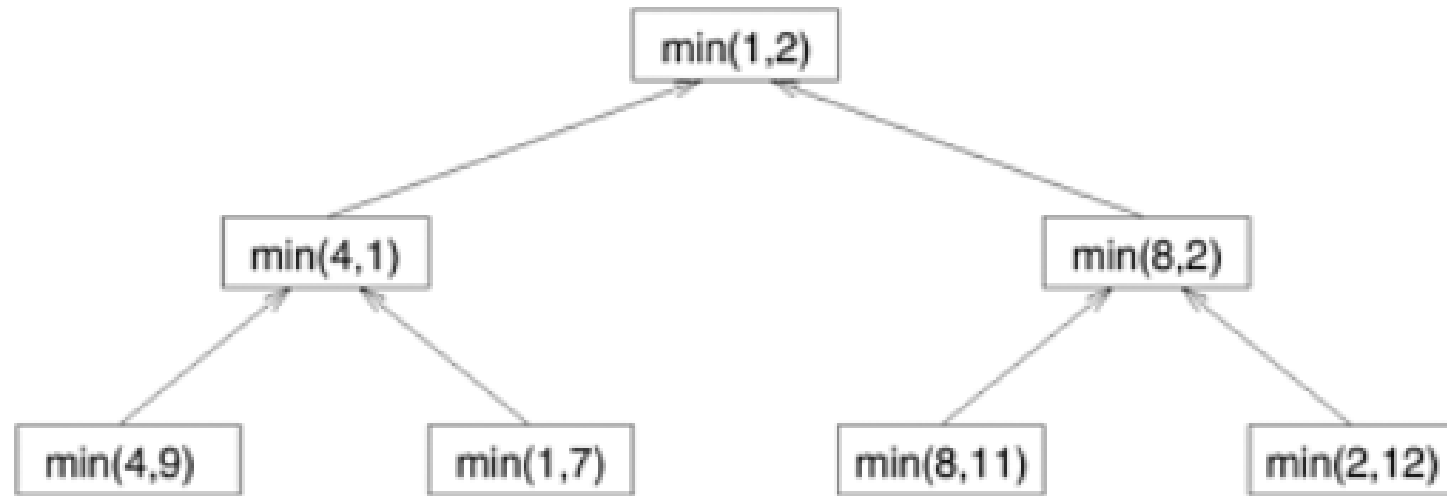
Algorithm 3.1 A serial program for finding the minimum in an array of numbers A of length n .

```
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
```

Algorithm 3.2 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
```

Figure 3.9. 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.



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.

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

Output Data Decomposition: Example

Consider the problem of multiplying two $n \times n$ matrices A and B to yield matrix C . The output matrix C can be partitioned into four tasks as follows:

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

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

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

$$\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Output Data Decomposition: Example

A given data decomposition does not result in a unique decomposition into tasks.

Figure 3.11. Two examples of decomposition of matrix multiplication into eight 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: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

Output Data Decomposition: Example

A given data decomposition does not result in a unique decomposition into tasks.

Figure 3.11. Two examples of decomposition of matrix multiplication into eight 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: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

Output Data Decomposition: Example

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)

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	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				

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.

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

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	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

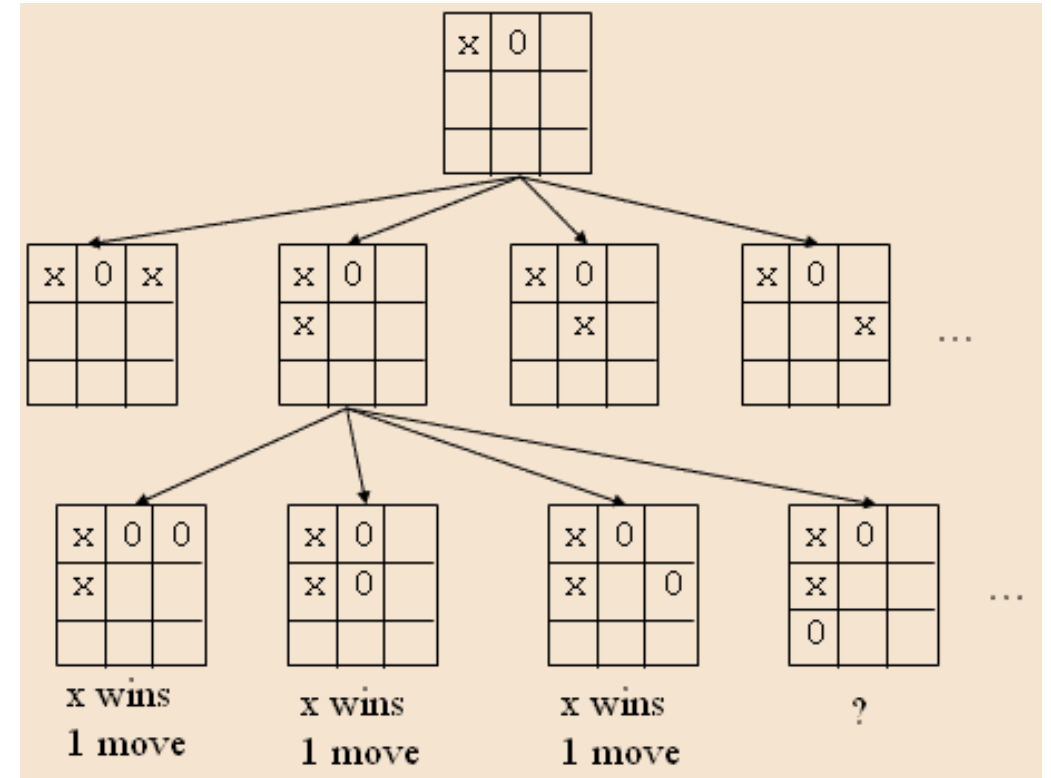
Database Transactions	A, B, C, E, G, H	Itemsets	C, D	Itemset Frequency	1
	B, D, E, F, K, L		D, K		2
	A, B, F, H, L		B, C, F		0
	D, E, F, H		C, D, K		0
	F, G, H, K				
	A, E, F, K, L				
	B, C, D, G, H, L				
	G, H, L				
	D, E, F, K, L				
	F, G, H, L				

task 2

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.

Exploratory Decomposition

- Example: **looking for the best move in a game.**
- Simple case: generate all possible configurations from the starting position.
- Send each of the configurations to a child process.
- *Each process will look for possible best moves for the opponent recursively eventually using more processes.*
- When it finds the result, it sends it back to the parent.
- *The parent selects the best move from all of the results received from the child* (eventually the worst move for the opponent).



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.