Principle of Parallel Algorithm Design

Alexandre David
B2-206

Today

- Preliminaries (3.1).
- Decomposition Techniques (3.2).
- Surprise.

The surprise is during the exercise session so you'd better come.

Overview

- Introduction to parallel algorithms.
  - Tasks and decomposition.
  - Processes and mapping.
  - Processes vs. processors.
- Decomposition techniques.
  - Recursive decomposition.
  - Exploratory decomposition.
  - Hybrid decomposition.

Introduction

- Parallel algorithms have the added dimension of *concurrency*.
- Typical tasks:
  - Identify concurrent works.
  - Map them to processors.
  - Distribute inputs, outputs, and other data.
  - Manage shared resources.
  - Synchronize the processors.

There are other courses specifically on concurrency. We won’t treat the problems proper to concurrency such as deadlocks, livelocks, theory on semaphores and synchronization. However, we will use them, and when needed, apply techniques to avoid problems like deadlocks.

Decomposing Problems

- Decomposition into *concurrent* tasks.
  - No unique solution.
  - Different sizes.
  - Decomposition illustrated as a directed graph:
    - Nodes = tasks.
    - Edges = dependency.

Task dependency graph

Many solutions are often possible but few will yield good performance and be scalable. We have to consider the computational and storage resources needed to solve the problems.

Size of the tasks in the sense of the amount of work to do. Can be more, less, or unknown. Unknown in the case of a search algorithm is common.

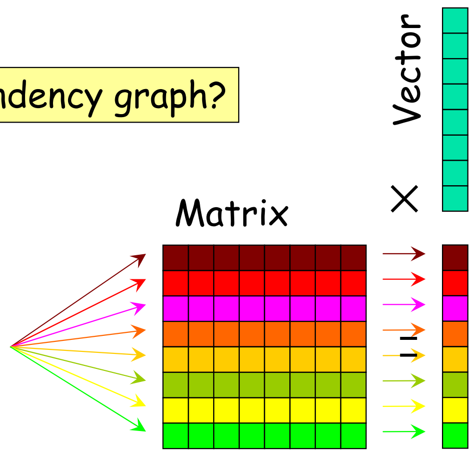
Dependency: All the results from incoming edges are required for the tasks at the current node.

We will not consider tools for automatic decomposition. They work fairly well only for highly structured programs or options of programs.

Example: Matrix * Vector

Task dependency graph?

N tasks, 1 task/row:



Example: Database Query Processing

MODEL = ``CIVIC'' AND YEAR = 2001 AND (COLOR = ``GREEN'' OR COLOR = ``WHITE)

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Table 3.1 A database storing information about used vehicles.

7

The question is: How to decompose this into concurrent tasks? Different tasks may generate intermediate results that will be used by other tasks.

A Solution

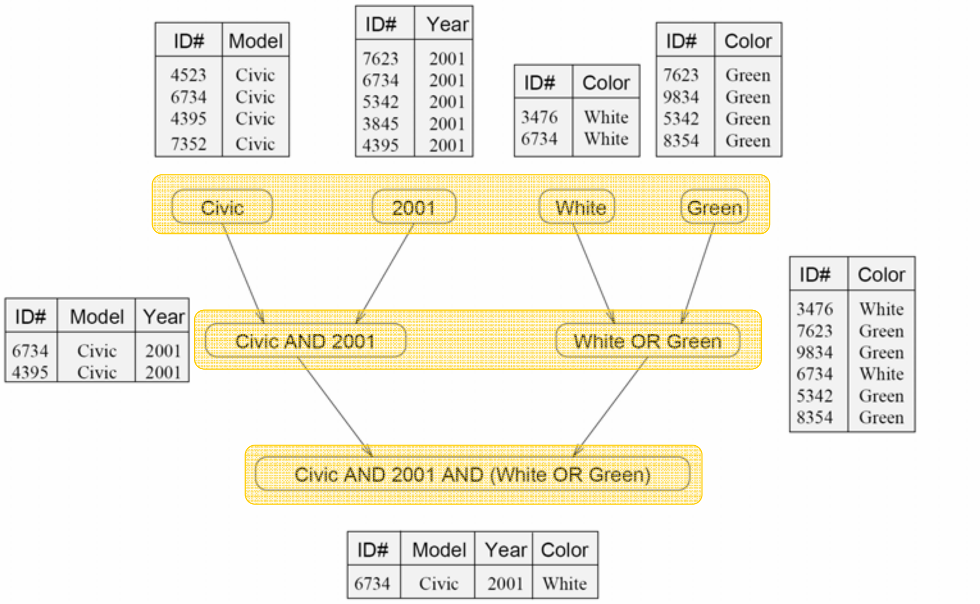
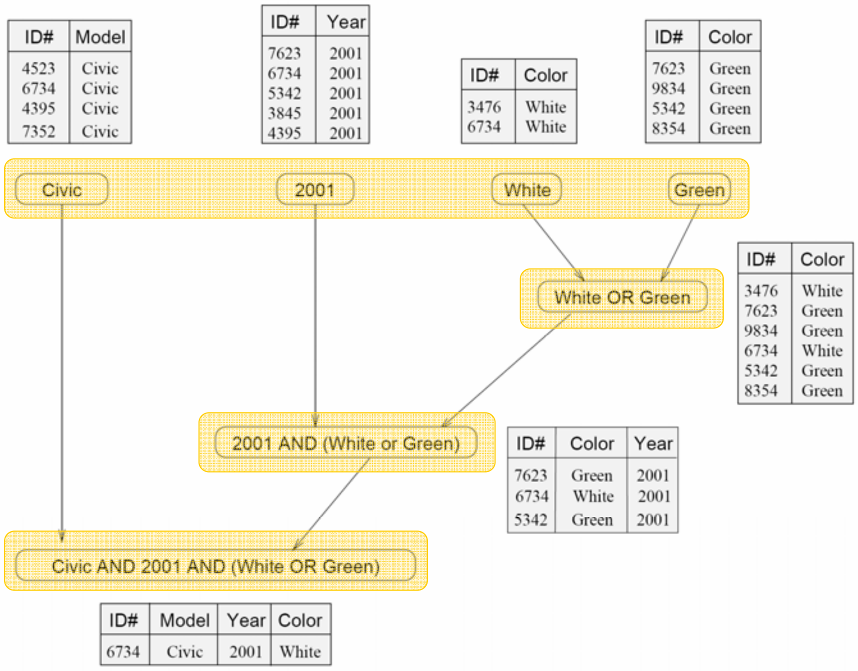


Figure 3.2 The different tables and their dependencies in a query processing operation.

8

How much concurrency do we have here? How many processors to use? Is it optimal?

Another Solution



9

Is it better or worse? Why?

Granularity

- Number and size of tasks.
  - Fine-grained: many small tasks.
  - Coarse-grained: few large tasks.
- Related: *degree of concurrency*.
  - Maximal degree of concurrency.
  - Average degree of concurrency.

•Previous matrix*vector fine-grained.

•Database example coarse grained.

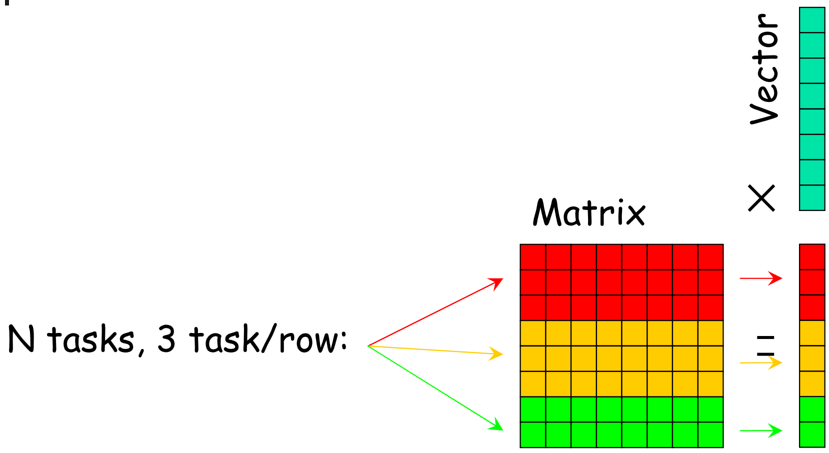
Degree of concurrency: Number of tasks that can be executed in parallel.

Average degree of concurrency is a more useful measure.

Assume that the tasks in the previous database examples have the same granularity. What's their degrees of concurrency? 7/3=2.33 and 7/4=1.75.

Common sense: Increasing the granularity of decomposition and utilizing the resulting concurrency to perform more tasks in parallel increases performance. However, there is a limit to granularity due to the nature of the problem itself.

Coarser Matrix * Vector



Granularity

- Average degree of concurrency if we take into account varying *amount of work?*
- Critical path = longest directed path between any start & finish nodes.
- Critical path length = sum of the weights of nodes along this path.
- Average degree of concurrency = total amount of work / critical path length.

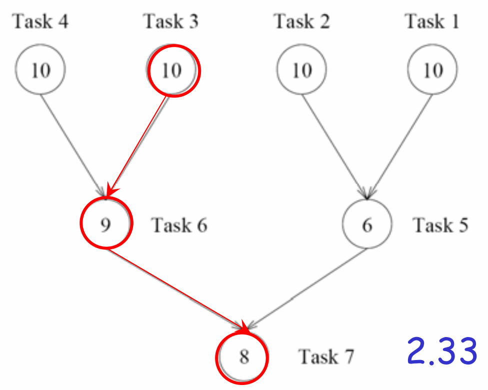
Weights on nodes denote the amount of work to be done on these nodes.
Longest path → shortest time needed to execute in parallel.

Database Example

Critical path (3).

Critical path length = 27.

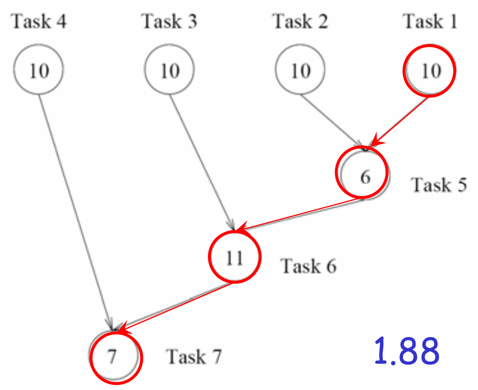
Av. deg. of concurrency = 63/27.



Critical path (4).

Critical path length = 34.

Av. deg. of conc. = 64/34.



Interaction Between Tasks

- Tasks often share data.
- Task interaction graph:
  - Nodes = tasks.
  - Edges = interaction.
  - Optional weights.
- Task dependency graph is a sub-graph of the task interaction graph.

Another important factor is interaction between tasks *on different processors*.

Share data implies synchronization protocols (mutual exclusion, etc) to ensure **consistency**.

Edges generally undirected. When directed edges are used, they show the direction of the flow of data (and the flow is unidirectional).

Dependency between tasks implies interaction between them.

Example: Sparse Matrix Multiplication

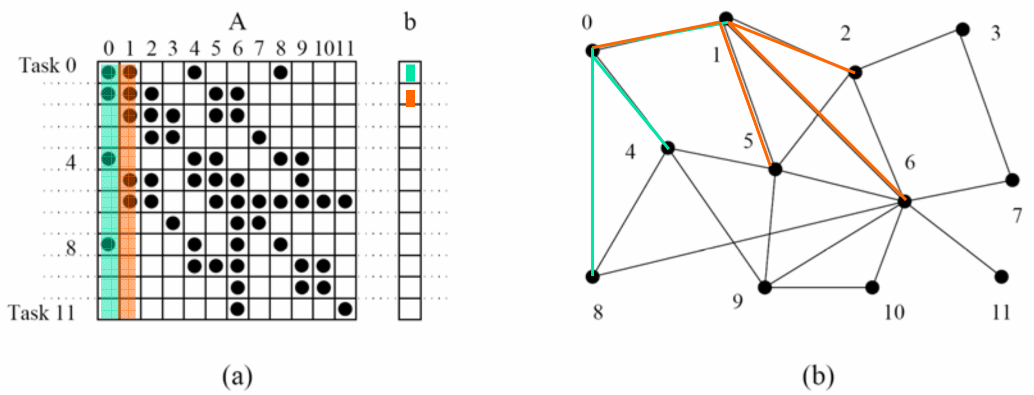


Figure 3.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i,j].b[j]$.

Sparse matrix: A significant number of its entries are zero and the zeros do not conform to predefined patterns. Typically, we do not need to take the zeros into account.

In the example: Task i owns row i of A and b.

Interaction depends on the mapping work to do / task, i.e., granularity, and mapping tasks – processor.

Processes and Mapping

- Tasks run on processors.
- Process: processing agent executing the tasks. Not exactly like in your OS course.
- Mapping = assignment of tasks to processes.

- API expose processes and binding to processors not always controlled.

Here we are not talking directly on the mapping to processors. A processor can execute two processes.

Good mapping:

•Maximize concurrency by mapping independent tasks to different processes.

•Minimize interaction by mapping interacting tasks on the same process.

Can be conflicting, good trade-off is the key to performance.

Decomposition determines degree of concurrency.

Mapping determines how much concurrency is utilized and how efficiently.

Mapping Example

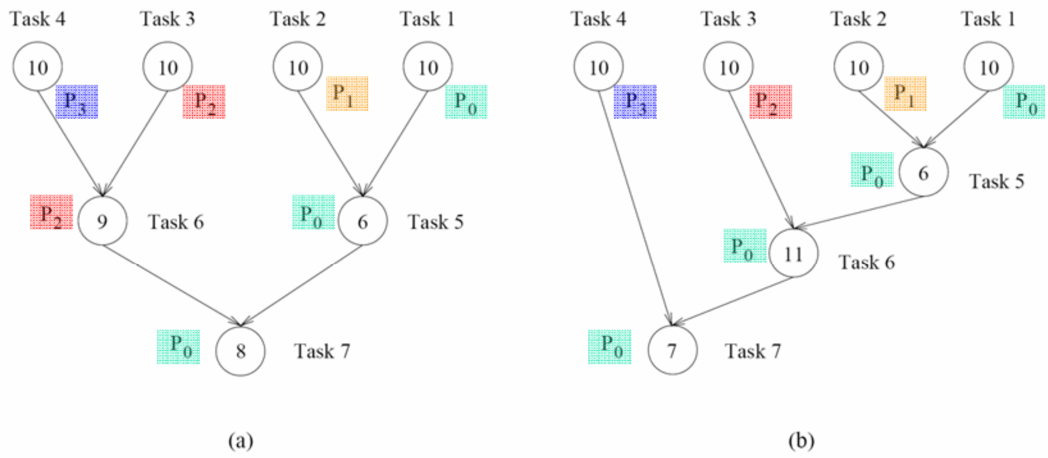


Figure 3.7 Mappings of the task graphs of Figure 3.5 onto four processes.

Notice that the mapping keeps one process from the previous stage because of dependency: We can avoid interaction by keeping the same process.

Processes vs. Processors

- Processes = logical computing agent.
- Processor = hardware computational unit.
- In general 1-1 correspondence but this model gives better abstraction.
- Useful for hardware supporting multiple programming paradigms.

Now remains the question:
How do you decompose?

21-02-2006 Alexandre David, MVP'06 18

Example of hybrid hardware: cluster of MP machines. Each node has shared memory and communicates with other nodes via MPI.

1. Decompose and map to processes for MPI.
2. Decompose again but suitable for shared memory.

Decomposition Techniques

- Recursive decomposition.
  - Divide-and-conquer.
- Data decomposition.
  - Large data structure.
- Exploratory decomposition.
  - Search algorithms.
- Speculative decomposition.
  - Dependent choices in computations.

Recursive Decomposition

- Problem solvable by divide-and-conquer:
  - Decompose into sub-problems.
    - Do it recursively.
  - Combine the sub-solutions.
    - Do it recursively.
- Concurrency: The sub-problems are solved in parallel.

Small problem is to start and finish: with one process only.

Quicksort Example

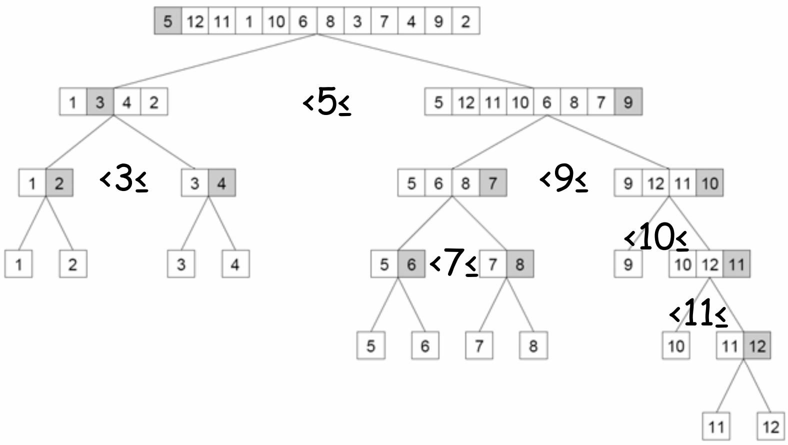


Figure 3.8 The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.

21

Recall on the quicksort algorithm:

•Choose a pivot.

•Partition the array.

•Recursive call.

•Combine result: nothing to do.

Minimal Number

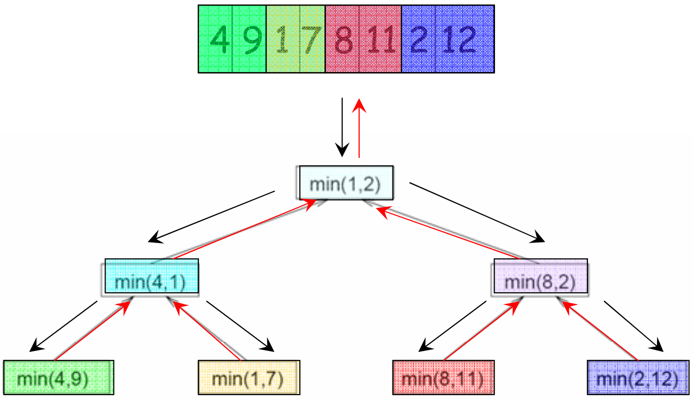


Figure 3.9 The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.

Data Decomposition

- 2 steps:
  - Partition the data.
  - Induce partition into tasks.
- How to partition data?
- Partition output data:
  - Independent “sub-outputs”.
- Partition input data:
  - Local computations, followed by combination.

Partitioning of input data is a bit similar to divide-and-conquer.

Matrix Multiplication

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

(a)

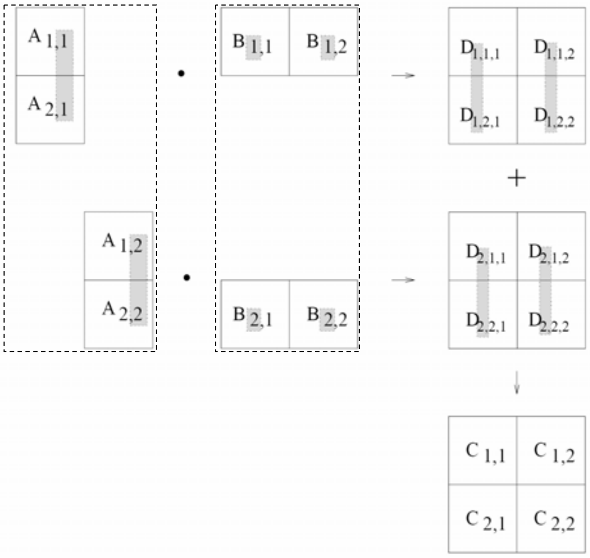
Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figure 3.10 (a) Partitioning of input and output matrices into 2×2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

We can partition further for the tasks. Notice the dependency between tasks.
What is the task dependency graph?

Intermediate Data Partitioning



Linear combination of the intermediate results.

Owner Compute Rule

- Process assigned to some data
  - is responsible for all computations associated with it.
- Input data decomposition:
  - All computations done on the (partitioned) input data are done by the process.
- Output data decomposition:
  - All computations for the (partitioned) output data are done by the process.

Exploratory Decomposition

15-puzzle example

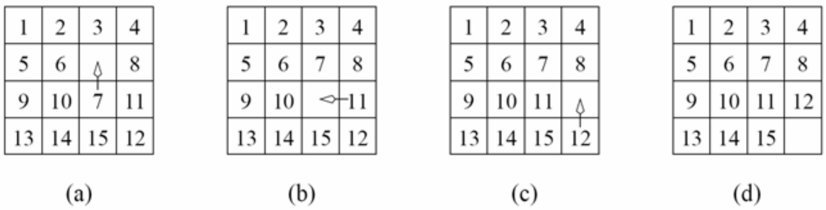
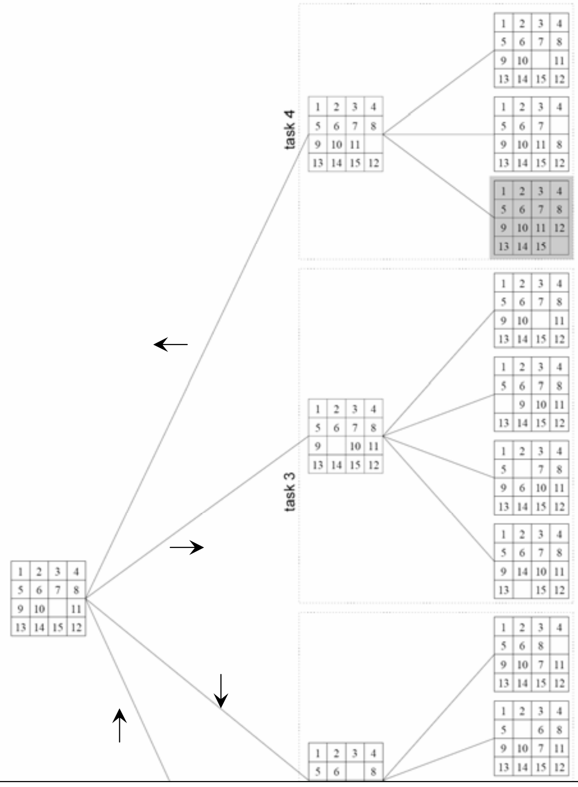


Figure 3.17 A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

Suitable for search algorithms. Partition the search space into smaller parts and search in parallel. We search the solution by a tree search technique.

Search



Anomalous Behavior Possible

Work depends on the order of the search!

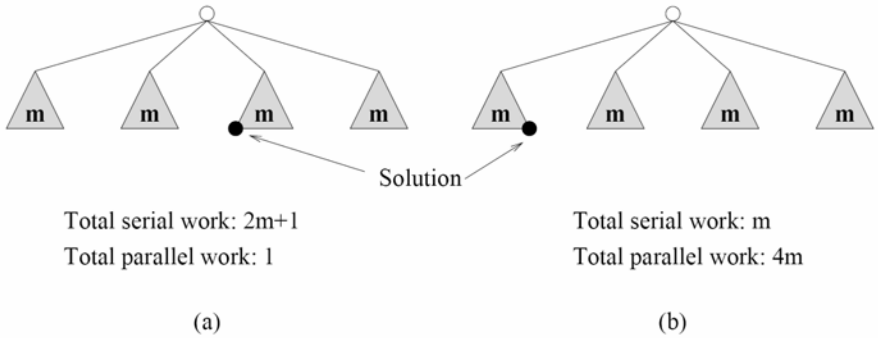


Figure 3.19 An illustration of anomalous speedups resulting from exploratory decomposition.

Speculative Decomposition

- Dependencies between tasks are not known a-priori.
  - How to identify independent tasks?
  - Conservative approach: identify tasks that are *guaranteed* to be independent.
  - Optimistic approach: schedule tasks even if we are not sure – may roll-back later.

Not possible to identify independent tasks in advance. Conservative approaches may yield limited concurrency. Optimistic approach = speculative. Optimistic approach is similar to branch prediction algorithms in processors.

Speculative Decomposition Example

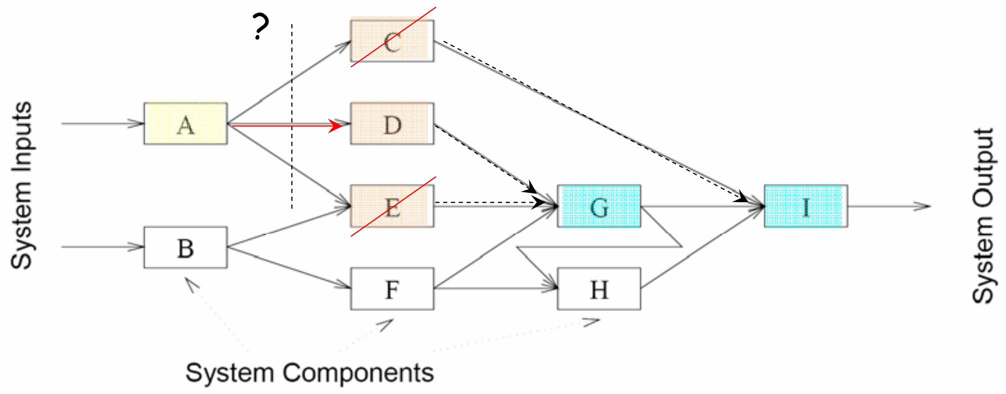


Figure 3.20 A simple network for discrete event simulation.

More aggregate work is done. Problem is to send inputs to the next stages speculatively. Could be the case that two different kinds of outputs are possible for A and A could start C,D,E twice.

Other approaches are possible that combine different techniques: hybrid decompositions.