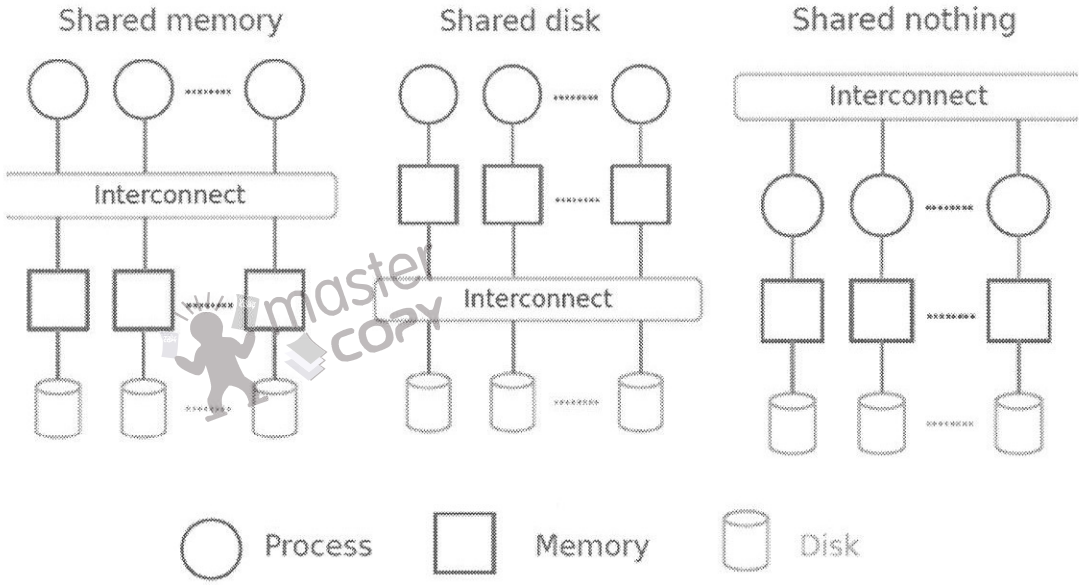
Parallel Architectures

Parallel Architectures are multiple machines interconnected and organized to exchange information.



Depending on how they're interconnected we have a different architecture.
A Shared memory multi-programming system can be obtained with POSIX Threads, OpenMP or Automatic Parallelization given by Compiler optimizations.
In message passing we can use sockets, message passing interface or parallel virtual machines.

Design of Parallel Algorithms

Parallel Algorithms are algorithms thought to be executed in a parallel architecture, more specifically shared-nothing architecture (message-passing model).

- Typical steps:
  - Identify what pieces of work can be performed concurrently
  - Partition concurrent work onto independent processors
  - Distribute a program's input, output, and intermediate data
  - Coordinate accesses to shared data: avoid conflicts
  - Ensure proper order of work using synchronization
- Some steps can be omitted
  - For shared memory parallel programming model, there is no need to distributed data
  - For message passing parallel programming model, there is no need to coordinate shared data
  - Processor partition may be done automatically

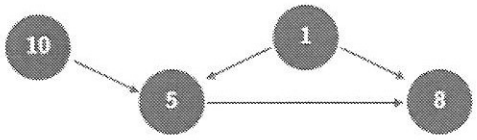
Recurrent Parallel Patterns

- Each pattern, or **forms of parallelism**, corresponds to:
  - specific structures of the parallel tasks
  - specific techniques of partitioning/allocating data
  - specific structures of communication
- We will see the most common ones, by instantiating them on a message passing model
- Before illustrating patterns, some preliminaries about modelling parallel programs

Task Dependency Graph

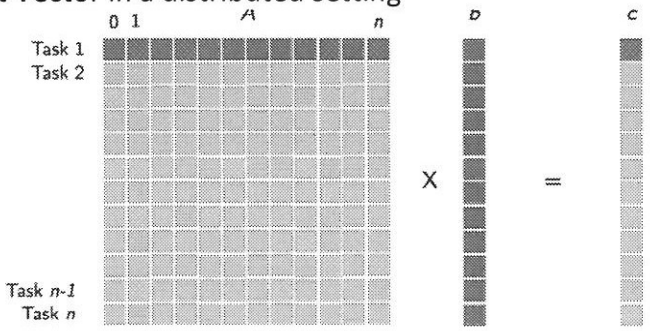
One of the easiest way to model a parallel algorithm, we need to decompose the problem to subproblems, or tasks, solvable concurrently.

- The first step in developing a parallel algorithm is to **decompose** the problem into tasks that can be executed concurrently
- A given problem may be decomposed into **tasks** in many different ways.
  - Tasks may be of same, different, or even indeterminate sizes/granularities
- Decomposition modelled/illustrated in the form of **Task Dependency Graph** (TDG)
  - Directed Acyclic Graph (DAG)
  - Nodes = tasks
  - Directed edges = control dependency among tasks
  - Node labels = computational size / weight of the task



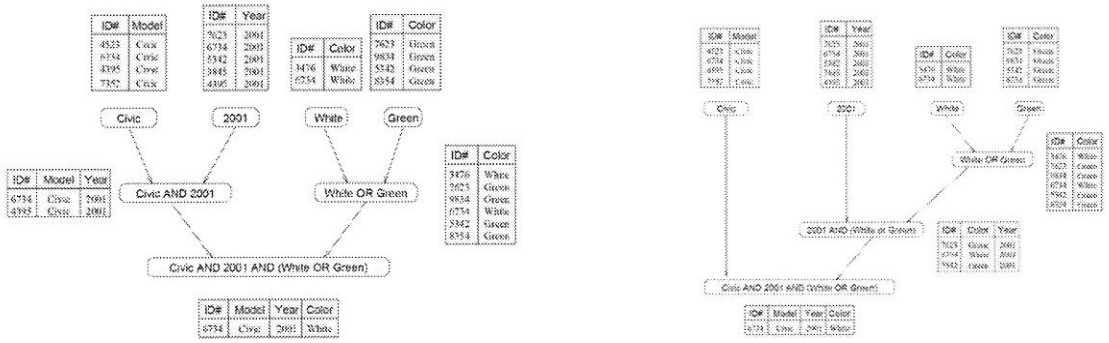
A task cannot be started if all the tasks with incoming edges are completed.
We need to wait for all control dependencies.

Ex. Product Vector in a distributed setting



- The computation of each element of the output vector c is independent of other row in A. Based on this, a dense matrix-vector product can be decomposed into n tasks
- The figure highlights the portion of the matrix and vector accessed by task 1
- While tasks share data (the vector b), they do not have any dependencies
- No task needs to wait for the (partial) completion of any other task
- All tasks are of the same size in terms of number of operations.

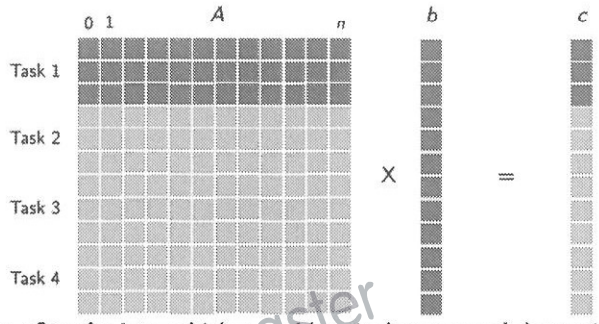
Ex. A query search generates the following graph



These are two ways to perform the same query, the graph changes a lot.

Task Granularity

How much work a task should do?



- The **number of tasks** into which a problem is decomposed determines its **granularity**
- Decomposition into a **large** number of tasks results in **fine-grained decomposition**
- Decomposition into a **small** number of tasks results in a **coarse grained decomposition**

Granularity = task size

- Fine-grain = small tasks, large number of tasks
- Coarse-grain = large tasks, small number of tasks
- Choose the proper granularity based on the problem and hardware

Example: matrix multiplication

- N x N matrix A multiply N x 1 vector b give N x 1 vector y
- Embarrassing parallel: each row of A times vector b gives an element of y

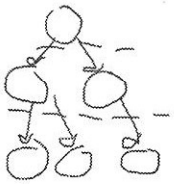
Simple decomposition:

- Task size is uniform
- No dependencies between task – All tasks share b
- Fine-grained: each task process one row of A
- Coarse-grained: each task process three rows of A

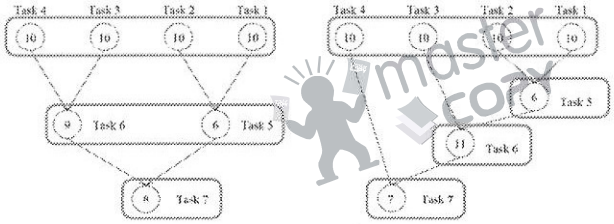
Degree of concurrency

- The number of tasks that can be executed **in parallel** is the **degree of concurrency** of a decomposition
- Since the number of tasks that can be executed in parallel may change over program execution, the **maximum degree of concurrency** is the maximum number of such tasks **at any time** during execution
  - What is the maximum degree of concurrency of the database query examples?
- The **average degree of concurrency** is the average number of tasks that can be processed in parallel **over** the execution of the program.
  - Assuming that each tasks in the database example takes identical processing time, and we have enough processors to execute independent task in parallel, what is the average degree of concurrency in each decomposition?
- If the average degree of concurrency is similar to the maximum, the parallel system is used **very efficiently**
- **The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa**

It depends on the number of incoming edges we have.
The maximum degree of concurrency relates to the maximum number of CPU we can use at limit.
We can distinguish here a first moment in which dc = 1, then dc = 2, then dc = 3.
If avg ~= max, it's better to buy the max. number of CPUs, or we have to choose a tradeoff.



Assuming that all tasks require the same amount of time, we can calculate the degree of parallelism.

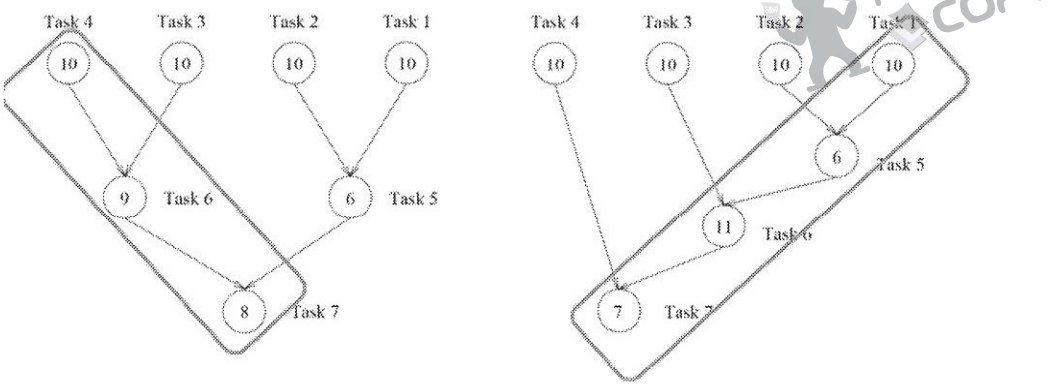


Max = 4 for both
Avg = 7/3, 7/4
In the first case we would need 3 days in which machines are more used, in the other 4 in which in the last 3 we would need to use.
We have 5/16 empty slots in the first, 9/16 empty slots in the second if we plot the table where columns are CPUs and rows are days.

Critical path

- A **directed path** in the task dependency graph represents a **sequence of tasks** that must be processed one after the other
- The **longest** such path determines the **shortest** time in which the program can be executed in parallel
  - measured in terms of **number of tasks**, or **sum of the weights** of the tasks involved
- The **length of the longest path** in a task dependency graph is called the **critical path length**
  - It corresponds to the **minimum execution time** of a parallel program

sequential steps to end the execution



Parallel Performance Limitations

- 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
  - That is, all the multiplications performed in parallel
- Concurrent tasks may also have to **exchange data** with other tasks. This results in **communication overhead**, which increases along with the parallelism increase (finer granularity)
- **The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds**

If we decompose at the finest granularity it's not necessarily convenient.

Tasks Interaction Graph

A second model is the following, devoted to take into account communication.

- 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 but partitioned, 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** (TIG)
  - Nodes = tasks
  - Edges (usually **indirect** ones) = interaction or data exchange
  - Node labels = computation weight of tasks
  - Edge labels = amount of data exchanged

Example of sparse graph, full of 0.
It's useless to compute it in the previous way, a multiplication by 0 gets always 0, also the exchange is kinda useless.

