Parallel Program Design
(cf. Libro Grama et al.)

Parallel Program Design

- One of the first steps of the design of a parallel program is to divide the problem into "chunks" of discrete job that can be distributed to multiple tasks. This is called **decomposition** or **partitioning**

- There are two main ways to partition the computational load among parallel tasks: **functional (task / work) decomposition** and **data decomposition**

Distributing Work & Data

Work decomposition

- based on loop decomposition

```
do i=1,100
→ i=1,25
  i=26,50
  i=51,75
  i=76,100
```

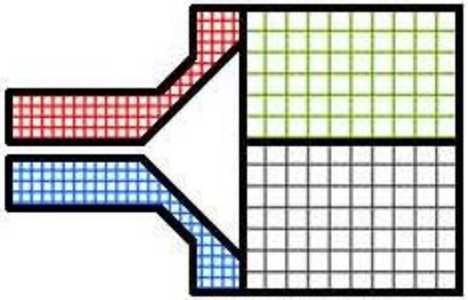
Data decomposition

- all work for a local portion of the data is done by the local processor

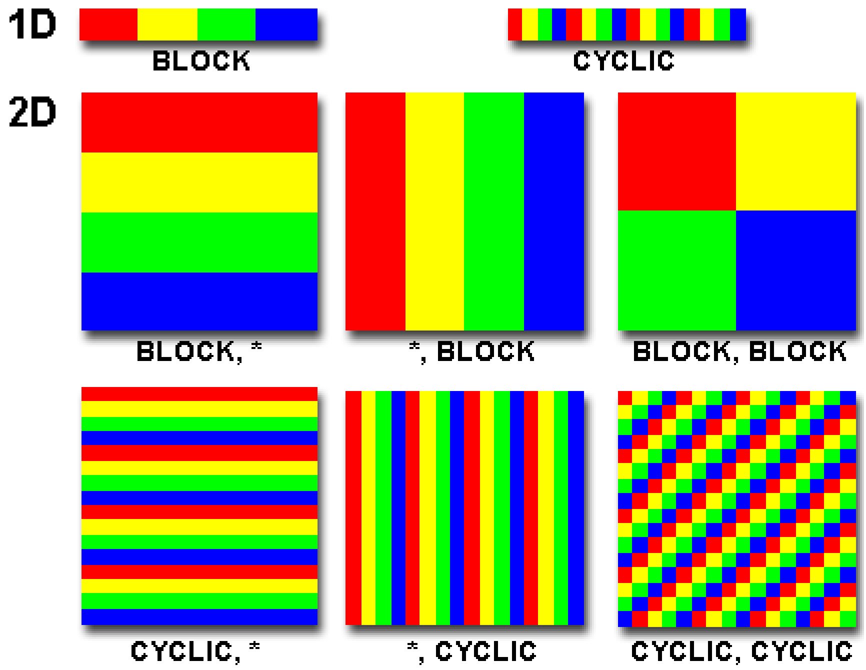
```
A( 1:20,  1: 50)
A( 1:20, 51:100)
A(21:40,  1: 50)
A(21:40, 51:100)
```

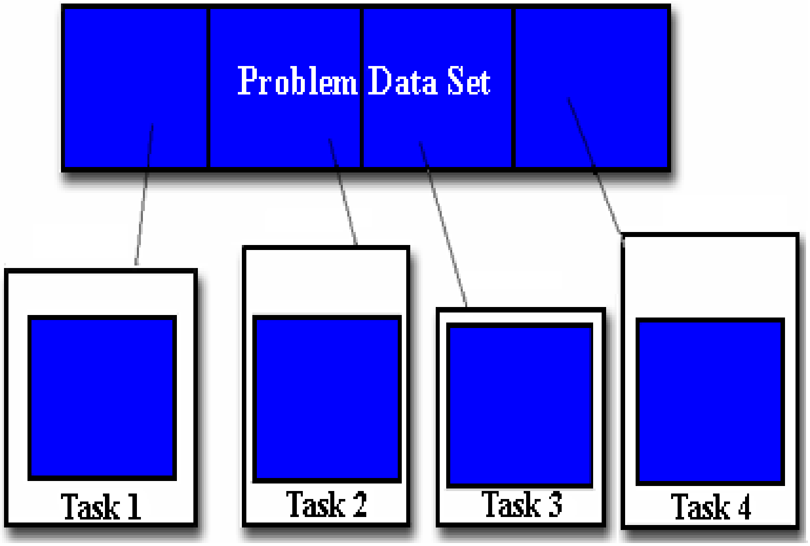
Domain decomposition

- decomposition of work and data is done in a higher model, e.g. in the reality



Data Decomposition

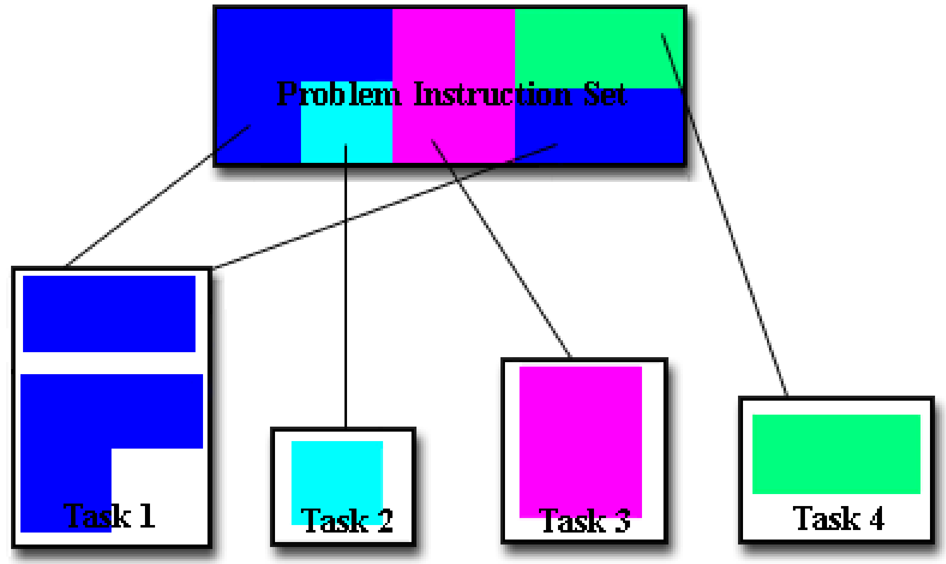


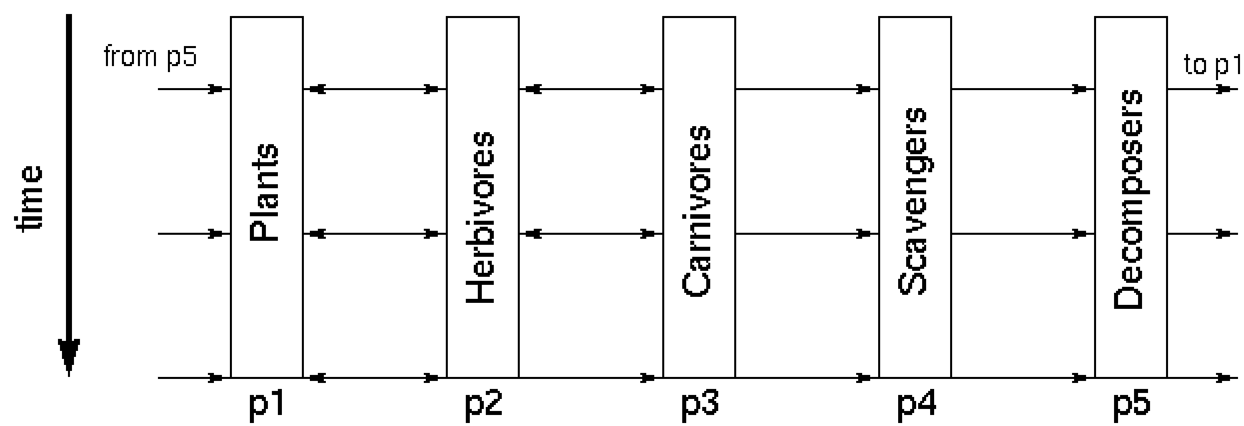


For example, cellular automata lend themselves well to this type parallelization

Task Decomposition

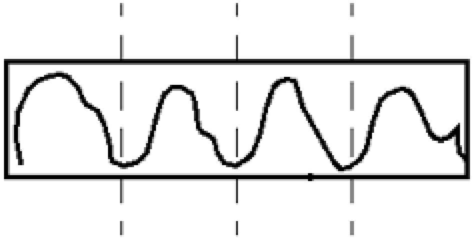
Functional decomposition works well on those problems that can be divided into different tasks, such as:
[Ecosystem Modeling]

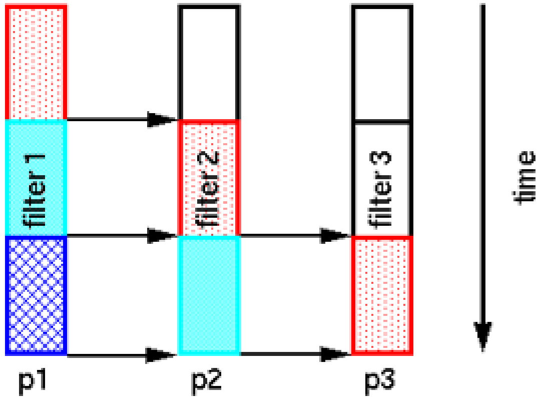




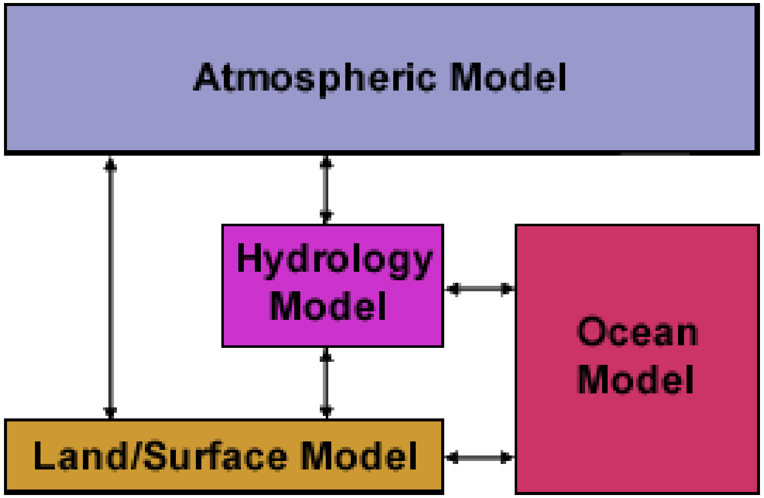
Task Decomposition

- **Signal Processing**





Climate Modeling



… suggestions, advice, etc …

Example of Non-Parallelizable Problem

- **Fibonacci series computations**

(1,1,2,3,5,8,13,21,...)

- The formula: F (k + 2) = F (k + 1) + F (k)
- This problem is **not easily parallelizable** because the calculation of the Fibonacci sequence includes dependent calculation, rather than independent
- The calculation of the value of k + 2 uses both value k + 1 and k. These three terms can not be calculated independently and then, not in parallel
- **In Posix/OpenMP? Thanks to recursion!**

Moral

Identify the hotspots of the program

Try to know where the work is done "really“. Most scientific programs usually run the main part of the work in a few places (typically, **for** loops!)

Focuses on the parallelization of the hotspots and ignore those parts of the program that use little CPU

Identify bottlenecks in the program

There are areas that are disproportionately slow, or cause the work parallelized to stop or be delayed? For example, I / O operations usually slows down the execution of the program!

You may need to restructure the program or use a different algorithm to reduce or eliminate areas that are too "slow"

Data Dependencies

- A **data dependency** exists between the instructions of a program when the **order** of execution of instructions **influence** the results of the program
- A data dependence occurs when multiple tasks use several times the same memory locations
- The dependences are important in parallel computing because they are one of the **biggest inhibitors to parallelism**

Moral - bis

Identifies inhibitors of parallelism

A common cause of inhibitor is the data dependence, as demonstrated in the example of the Fibonacci sequence

Investigate other algorithms if possible

This might even be the only alternative when designing a parallel application

How to deal with Data Dependencies

Simple!

- Distributed memory architectures – Communicate the data in sync points

- Shared memory architectures –

  Synchronizes the read / write operations between tasks

Data Dependencies

Example: cycle data dependence

```
for (i=init; i<end; i++)
  a[j] = a[j-1] * 2.0
```

The value of `a[j-1]` must be calculated before the value of `a[j]`, so `a[j]` shows a date dependency on `a[j-1]`. Parallelism is inhibited. If the Task 2 has `a[j]` and Task 1 has `a[j-1]`, the calculation of the corrected value of `a[j]` requires:

- In **Distributed Memory Architectures** - task 2 must obtain the value of `a[j-1]` from task 1 **after** task 1 has finished computing
- In **Shared Memory Architectures** - Task 2 should read `a[j-1]` **after** task 1 has updated

Data Dependencies

Example: Independent-cycle data dependence

```
task 1              task 2
------              ------
X = 2               X = 4
          . . . .
Y = X**2            Y = X**3
```

X, Y are shared variables

As in the previous example, the parallelism is inhibited. The correct value of Y depends on:

In **Distributed Memory Architecture** - If or when the value of X is communicated between tasks

In **Shared Memory Architecture** - which task stores the value of X for last

Principles of Parallel Algorithm Design

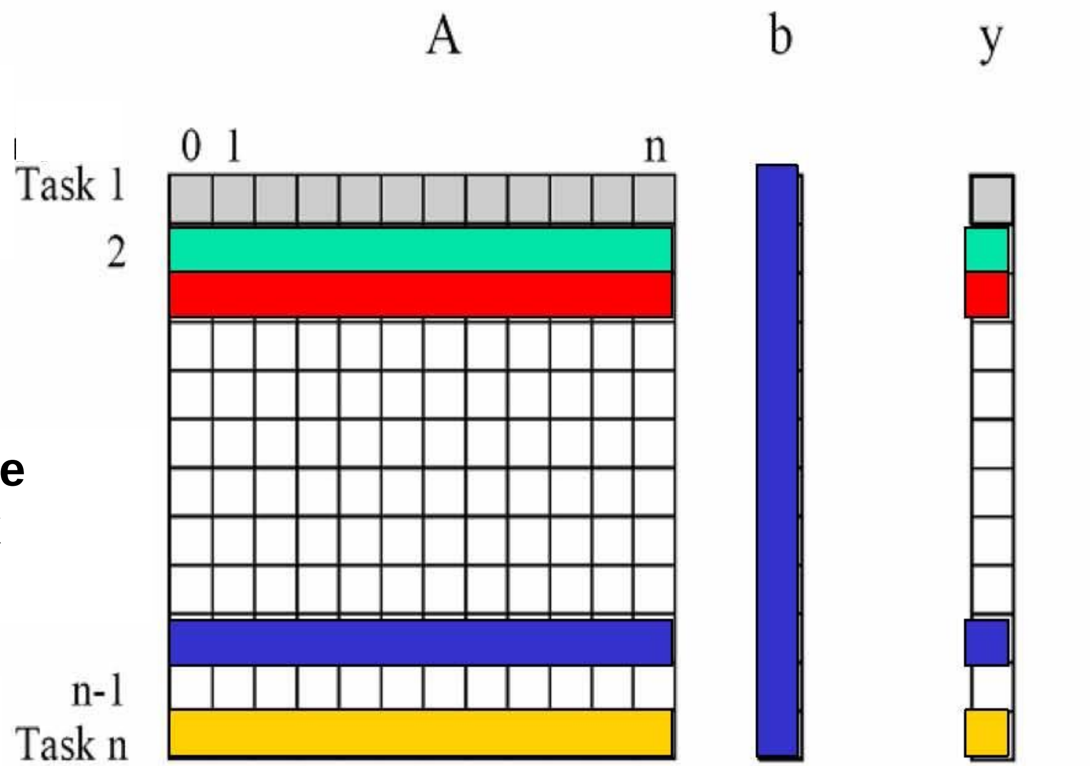
Task Decomposition

Let's consider a matrix-vector product:

$$y[i]=\Sigma_{j=1,n}\ A([i,j]\times b[j])$$

n tasks are considered, as the number of rows of the matrix

Tasks are independent, and can be computed in any order



Task Graph Model (Task Parallelism)

- Based on the **task dependency graph**
- Useful to reduce the interaction degree
- Used when the quantity of data a task has to compute is large with respect to the computational cost
- Tasks are statically associated, to minimize data exchange among tasks
- Works better if applied for a shared-memory architecture
- Example: Parallel Quicksort

Task-Dependency Graph

- The **dependency graph** is used to explicit which tasks need the result of other tasks and their execution order
- It’s a DAG
- Nodes represent tasks
- Arcs represent the **<u>dependence</u>** among tasks

What’s the dependency graph of the previous example?

- In this case, the graph is disconnected (arc set =0) since all tasks are **<u>independent</u>** from each other

N.B. DAG = Direct Acyclic Graph

Example: Data-Base Query

- **Let's consider a car relational DB:**

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

- **Let's consider the query:**
  - **MODEL="civic" AND YEAR="2001" AND (COLOR="Green" OR "COLOR="Withe")**

Task-Dependency Graph

4 tables

All Civics

All 2001 models

All green models

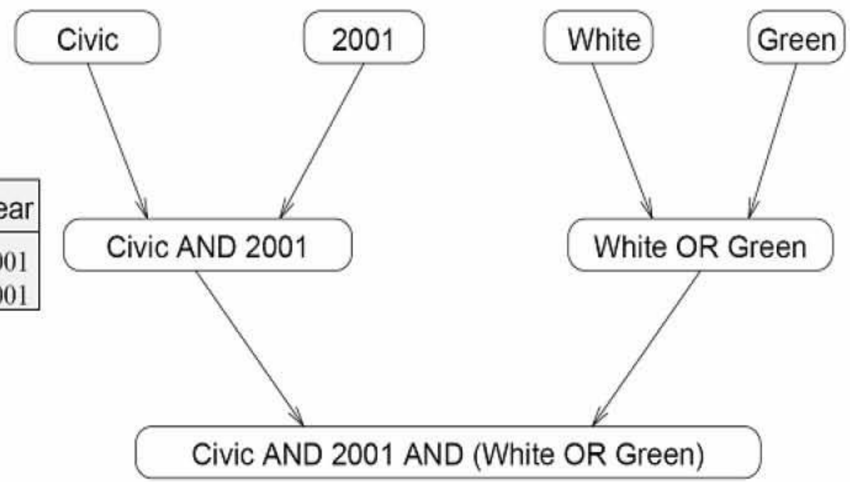
All white models

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green



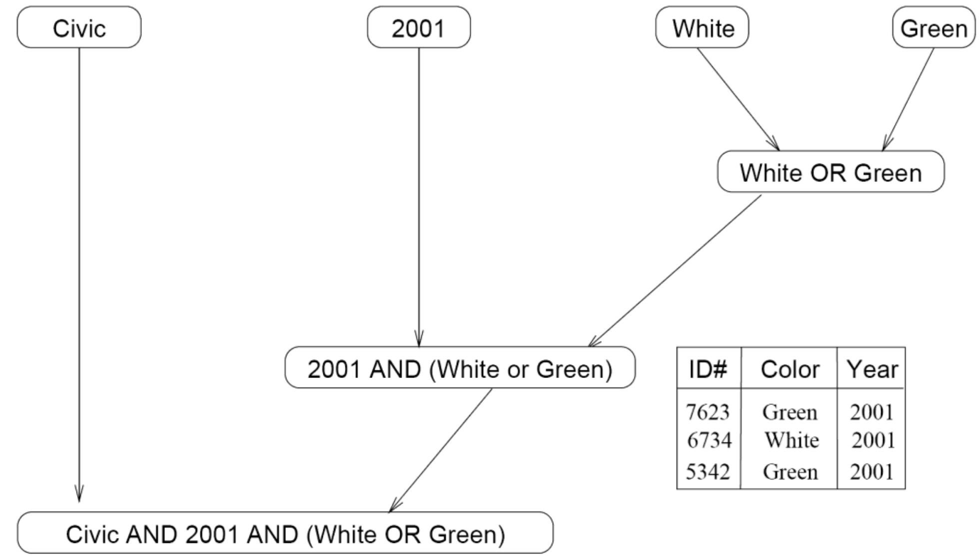
ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

... alternative

…Note that the same problem can be decomposed in other ways ...



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

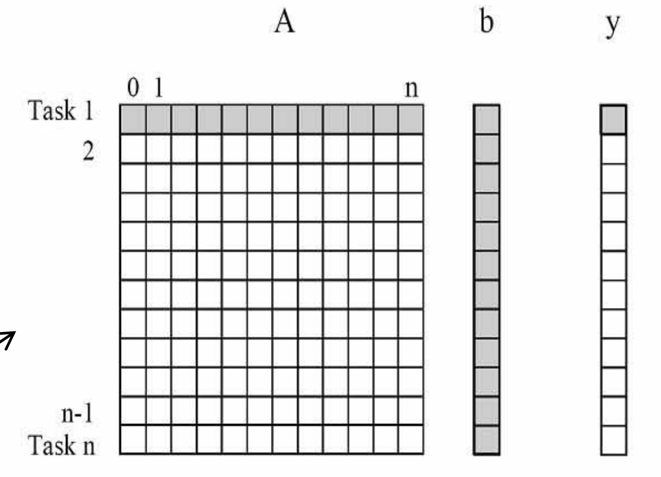
ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

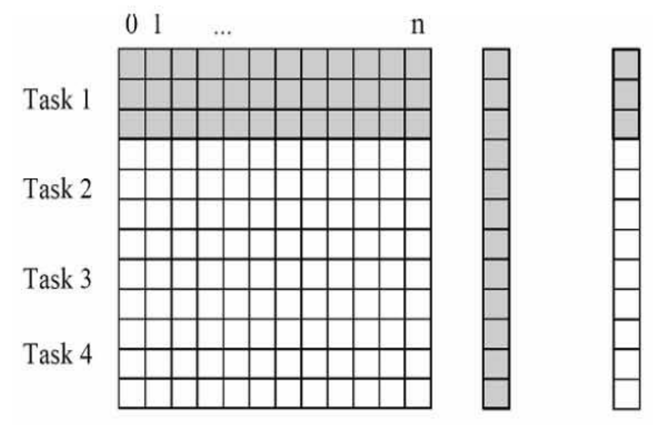
Granularity

- Granularity of the task decomposition
  - Depends both on the number and size of tasks



- Fine grained

- Coarse grained



Concurrency

- It's linked with granularity: when granularity is fine, the concurrency degree among tasks **increases**
- **Maximum concurrency degree**: maximum number of tasks that can be executed simultaneously
- **Average degree of concurrency**: average number of tasks that can be executed simultaneously, computed on the overall duration of the program
- For the same granularity, the concurrency degree is not the same: it depends also on task dependency

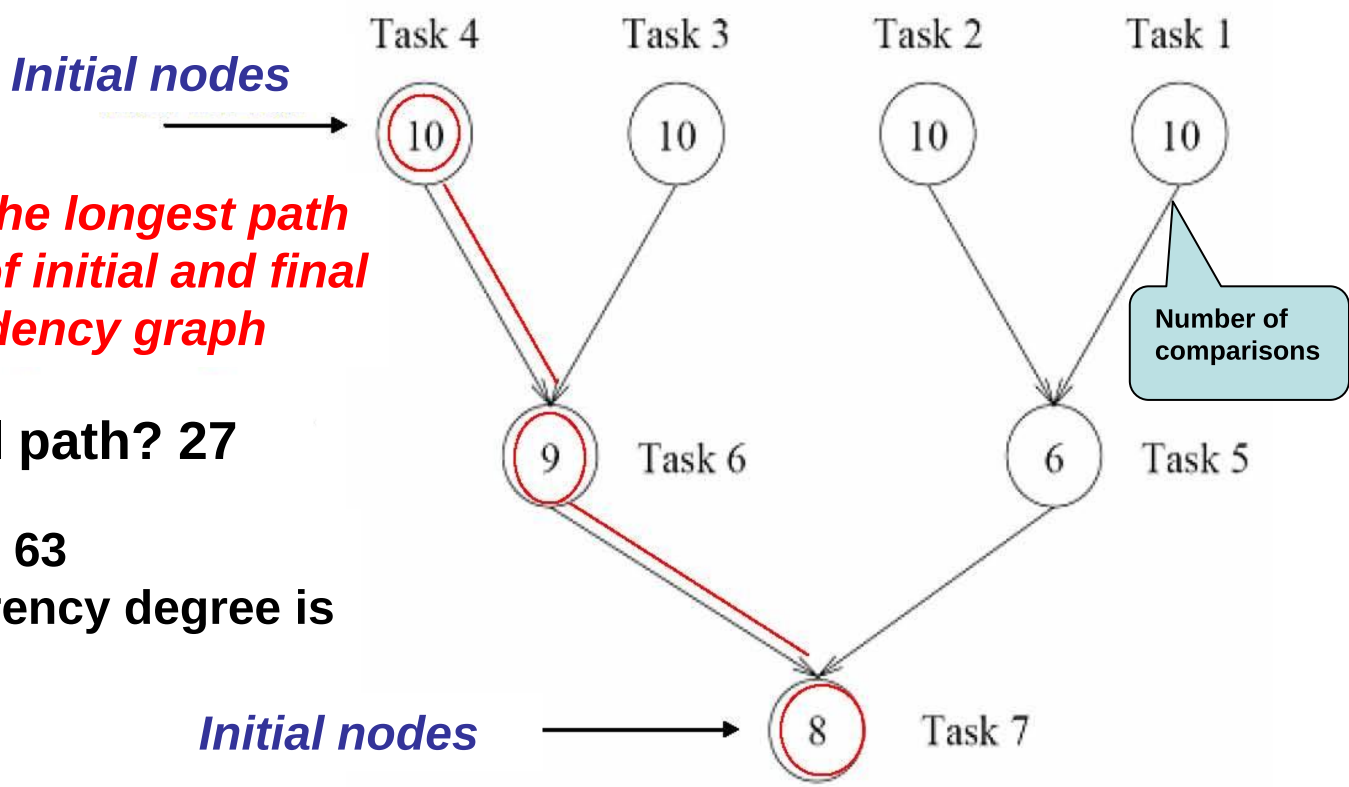
Critical Path

- An aspect of the task dependence that determines the **average degree of concurrency** for a given granularity
- Suppose that in the dependency graph a *weight* at each node is associated that depends con the <u>quantity of work</u> that a task has to carry out

The critical path is the longest path between each pair of initial and final nodes in the dependency graph

Length of critical path? 27

The total work is 63
Average concurrency degree is 63/27=2.33



NB: The average concurrency degree for the 2° decomposition is 1.88

Performance Limits

- It would seem that the parallel time can be reduced in an arbitrary manner by simply making the **granularity finer**

- In practice, there is a lower limit on "how fine" may be the granularity of the computation. For example, in the case of the multiplication of a dense matrix with a vector, <u>it does not make sense to use more than (n^2) concurrent tasks.</u>
- In addition, concurrent tasks may also have the need (obvious!) to exchange data with other tasks. This involves a communication overhead.
- <u>The tradeoff between the granularity of a decomposition and the associated overhead will often determine the limits of performance</u>

- In fact ...

Task Interaction Graphs

- **Task interaction** is a limiting factor for having an **infinite speedup**
- Tasks in which an algorithm is decomposed can **<u>share</u> input, output and other intermediate data**
- Tasks that seem independent may need to share data (in which to write, for instance)
- In the case of the matrix-vector multiplication, all tasks must access vector B, so a suitable data exchange is necessary

<u>Obs</u>: The set of edges of a task-interaction graph includes that of task-dependency of the graph (eg, in the previous query they are the same)

Task Interaction Graphs

- **Captures the pattern of interaction between tasks**
- This graph usually contains the **task-dependency graph** as a **subgraph**
- In fact, there may be interactions between tasks even if there are no dependencies
- These interactions usually occur due to accesses on shared data

Task Interaction Graphs: An example

Consider the problem of multiplying a **sparse matrix *A*** with a vector ***b***. The following observations can be made:

- As before, the computation of each element of the result vector can be viewed as an **independent task**.
- Unlike a dense matrix-vector product though, only non-zero elements of matrix ***A*** participate in the computation.
- If, for memory optimality, we also partition ***b*** across tasks, then one can see that the **task interaction graph of the computation is identical to the graph of the matrix *A*** (the graph for which ***A*** represents the adjacency structure).

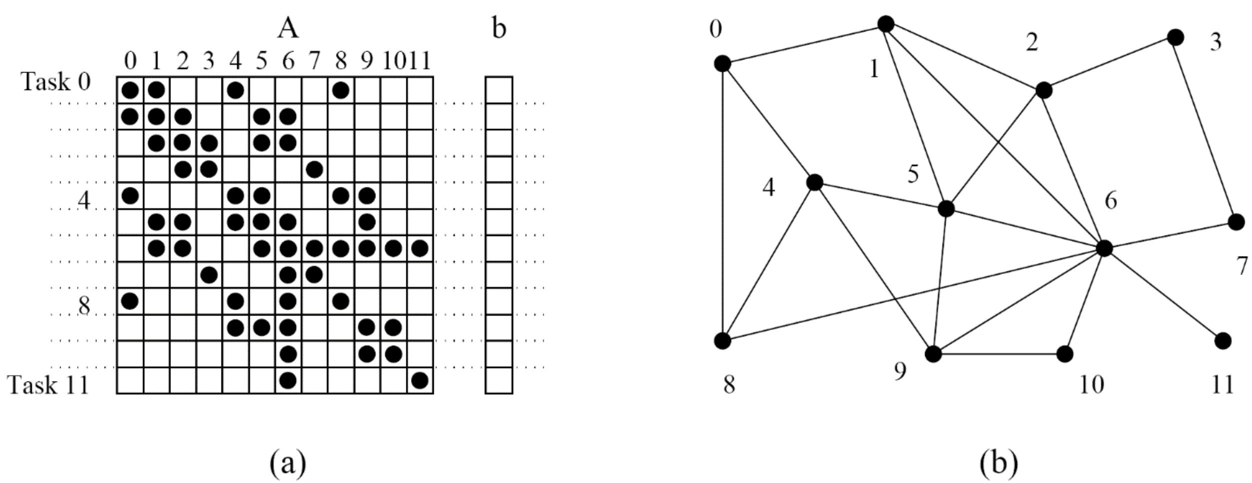


Figure 3.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \ne 0} A[i, j].b[j]$.

Task Interaction Graphs, Granularity, and Communication

In general, if the granularity of a decomposition is finer, the associated overhead (as a ratio of useful work associated with a task) increases

Example: Consider the sparse matrix-vector product example. Assume that each node takes **<u>1 unit time</u>** of computation and each interaction (edge) causes an overhead of **<u>1 unit time</u>**.

- Viewing node 0 as an independent task involves a useful computation of **one time** unit and overhead (communication) of **three time** units (3/1 ratio)
- Now, if we consider nodes 0, 4, and 5 as one task, then the task has useful computation totaling to three time units and communication corresponding to five time units (five edges). Clearly, this is a **<u>more favorable ratio</u>** than the former case (5/3 ratio)

Thus, it seems that using <u>less</u> tasks is better?

At the extreme, one task is <u>**better**</u> than many tasks ?!