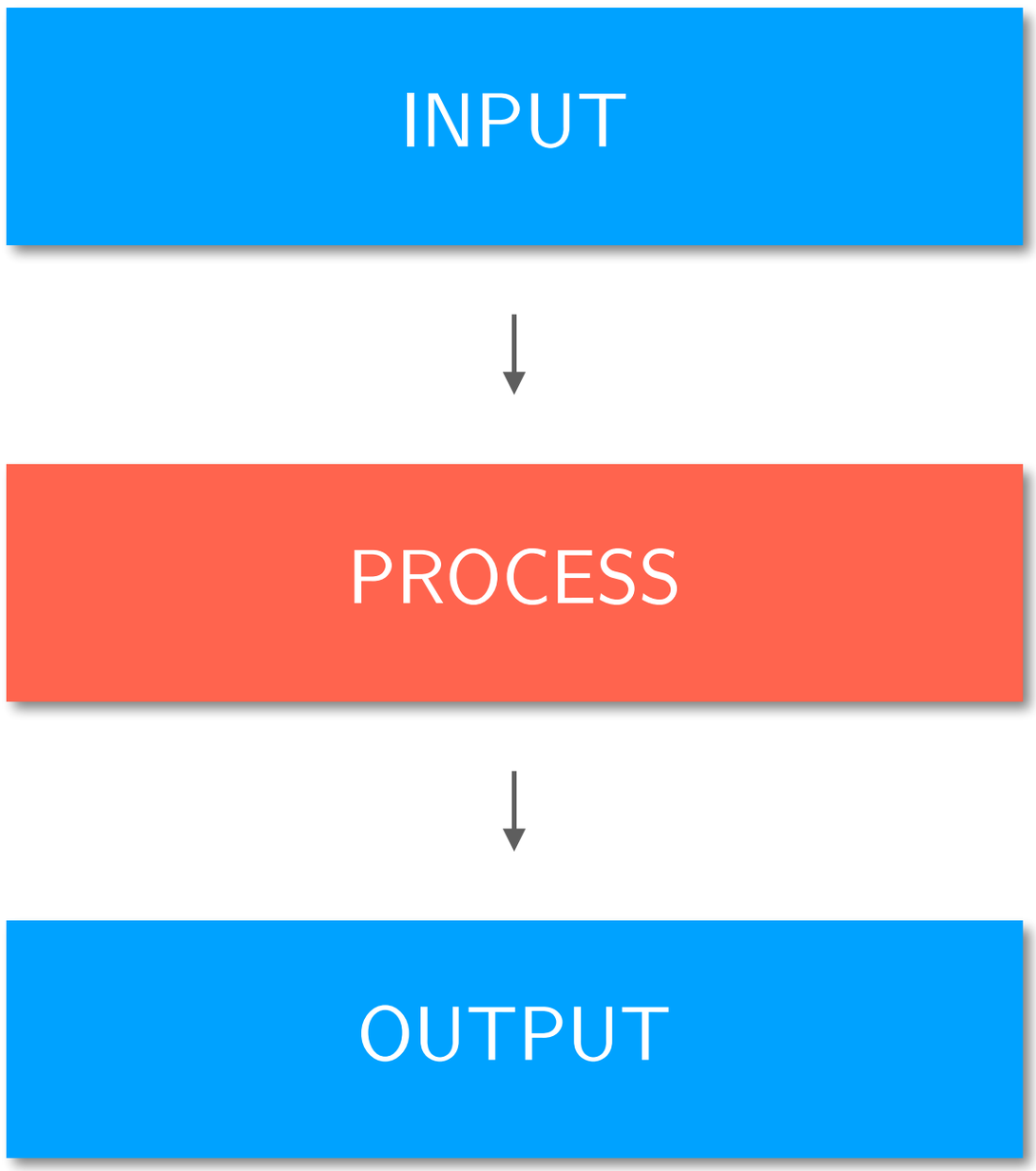
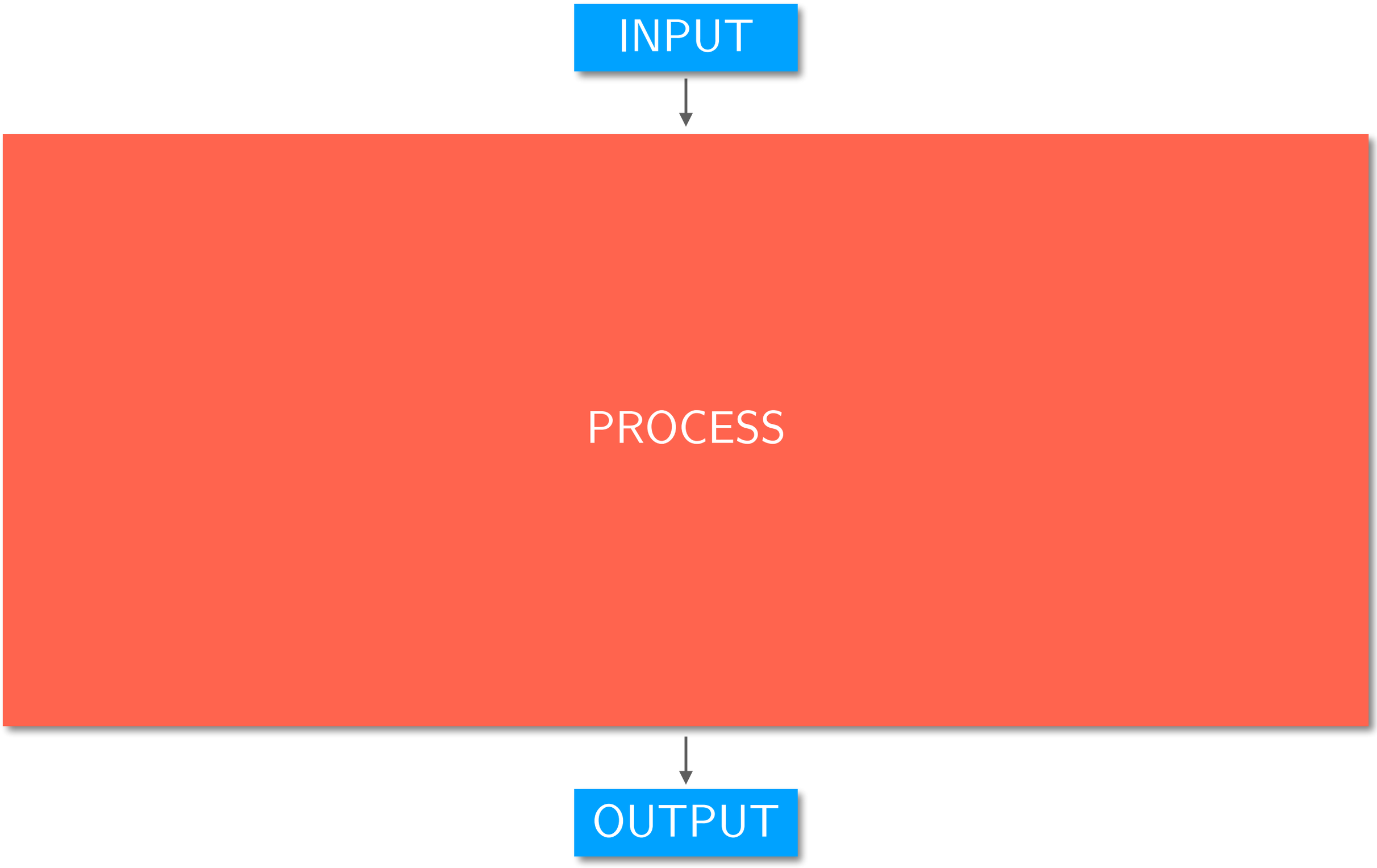
Typical Application

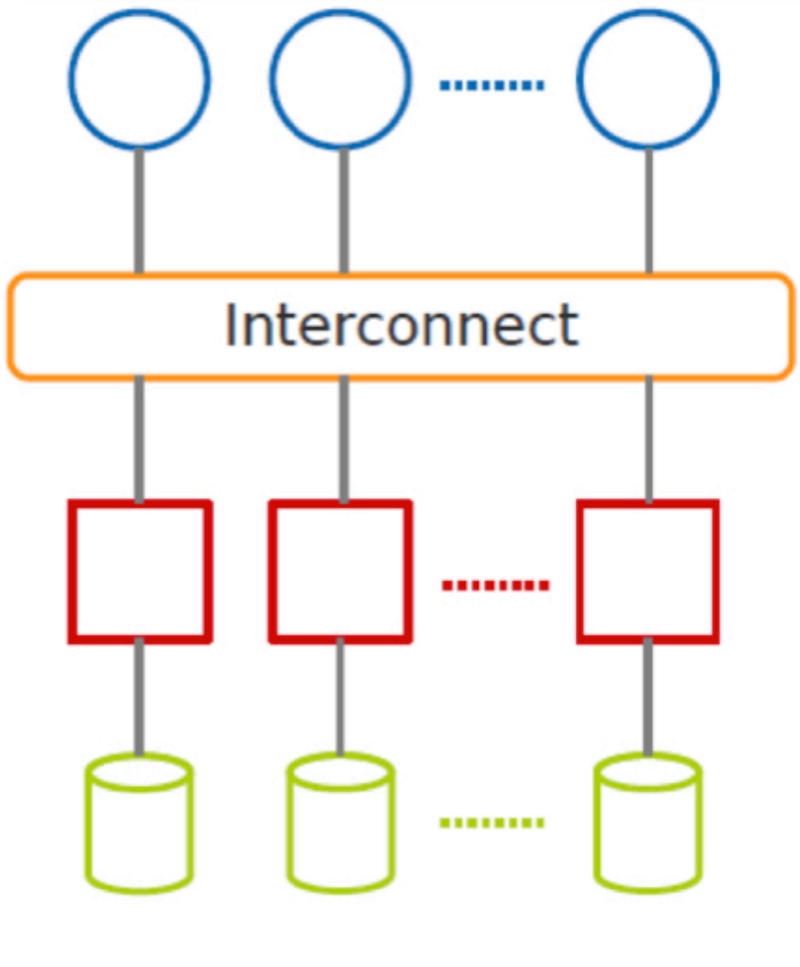


What if?



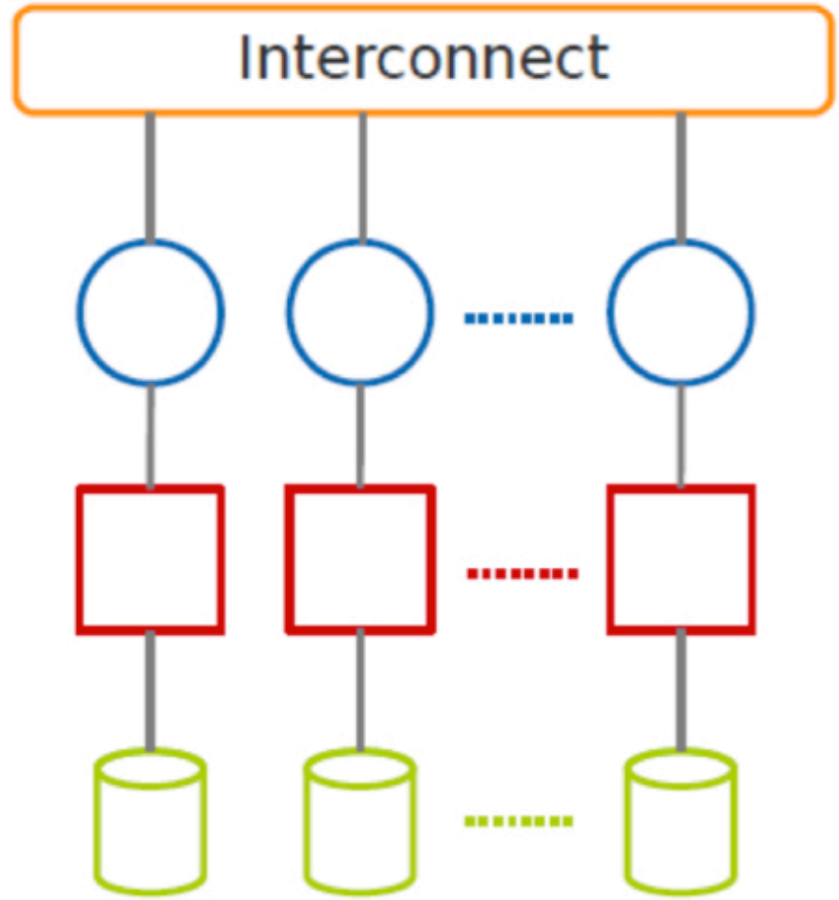
Parallel Architectures

Shared Memory



- Posix Threads
- OpenMP
- Automatic Parallelization (Compiler optimizations)

Message Passing



- Sockets
- PVM - Parallel Virtual Machine (obsolete)
- MPI - Message Passing Interface

DeWitt and Gray, ‘‘Parallel database systems: the future of high performance database systems’’, ACM Communications, 1992

Designing Parallel Algorithms

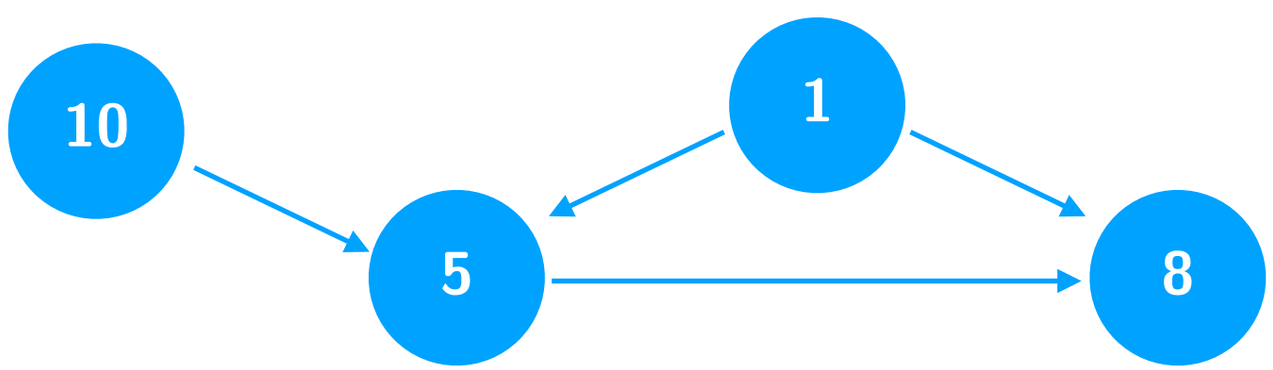
- Typical steps:
  - Identify what pieces of work can be performed concurrently
  - Partition concurrent work onto independent processors
  - Distribute a program's input, output, and intermediate data
  - Coordinate accesses to shared data: avoid conflicts
  - Ensure proper order of work using synchronization
- 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

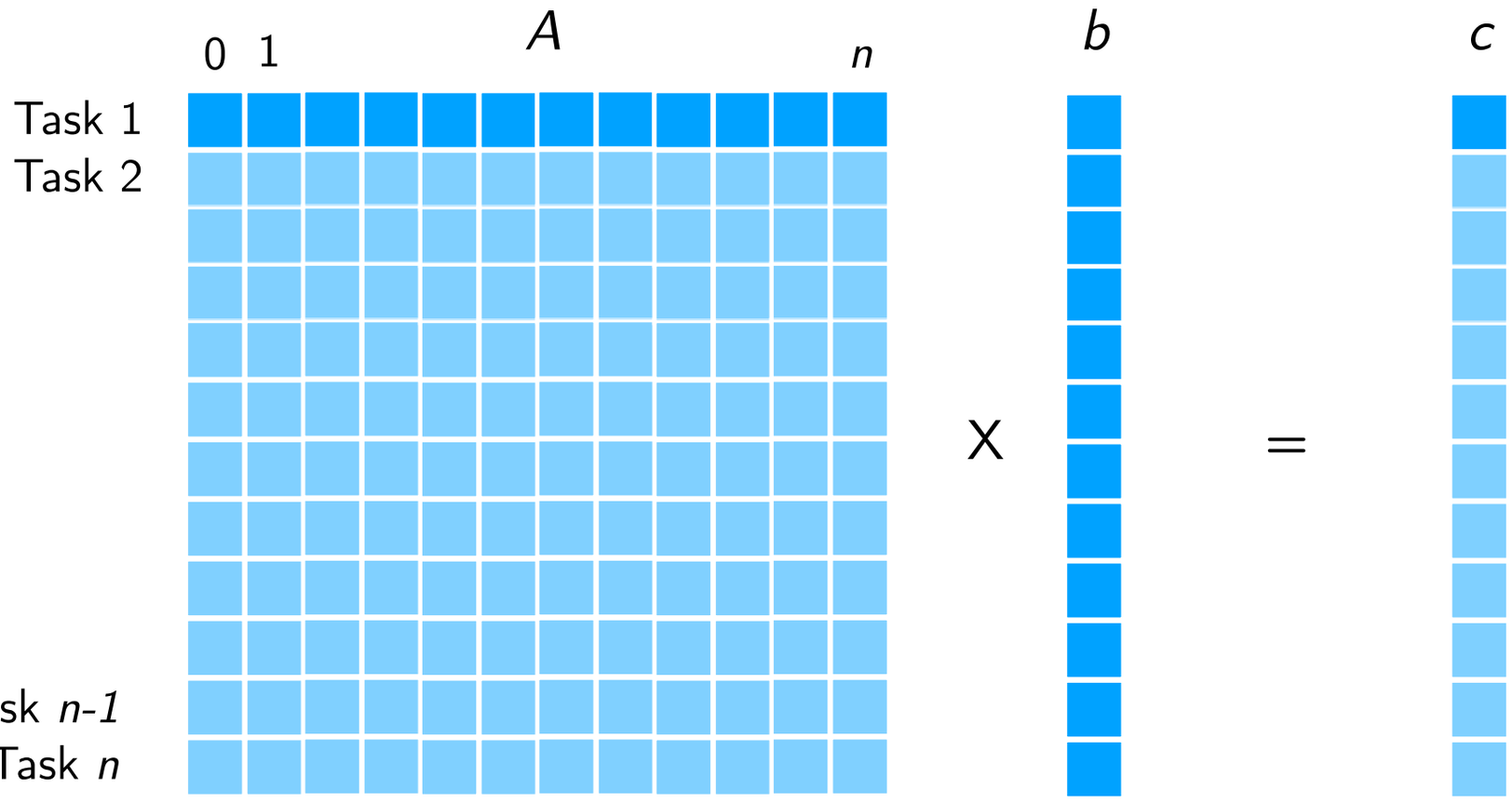
- Each pattern, or **forms of parallelism**, corresponds to:
  - specific structures of the parallel tasks
  - specific techniques of partitioning/allocating data
  - specific structures of communication
- We will see the most common ones, by instantiating them on a message passing model
- Before illustrating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

Tasks Dependency Graph

- 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



Example



- 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.

Example

- Consider the execution of the query:

```
SELECT * FROM T WHERE
Model = "CIVIC" AND Year = "2001" AND
(Color = "GREEN" OR Color = "WHITE")
```

- on the following table T:

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

Example

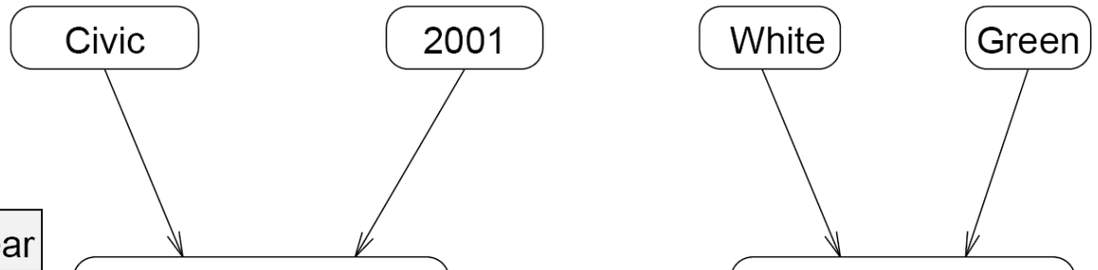
- The execution of the query can be divided into subtasks in various ways
- Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause

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

Example

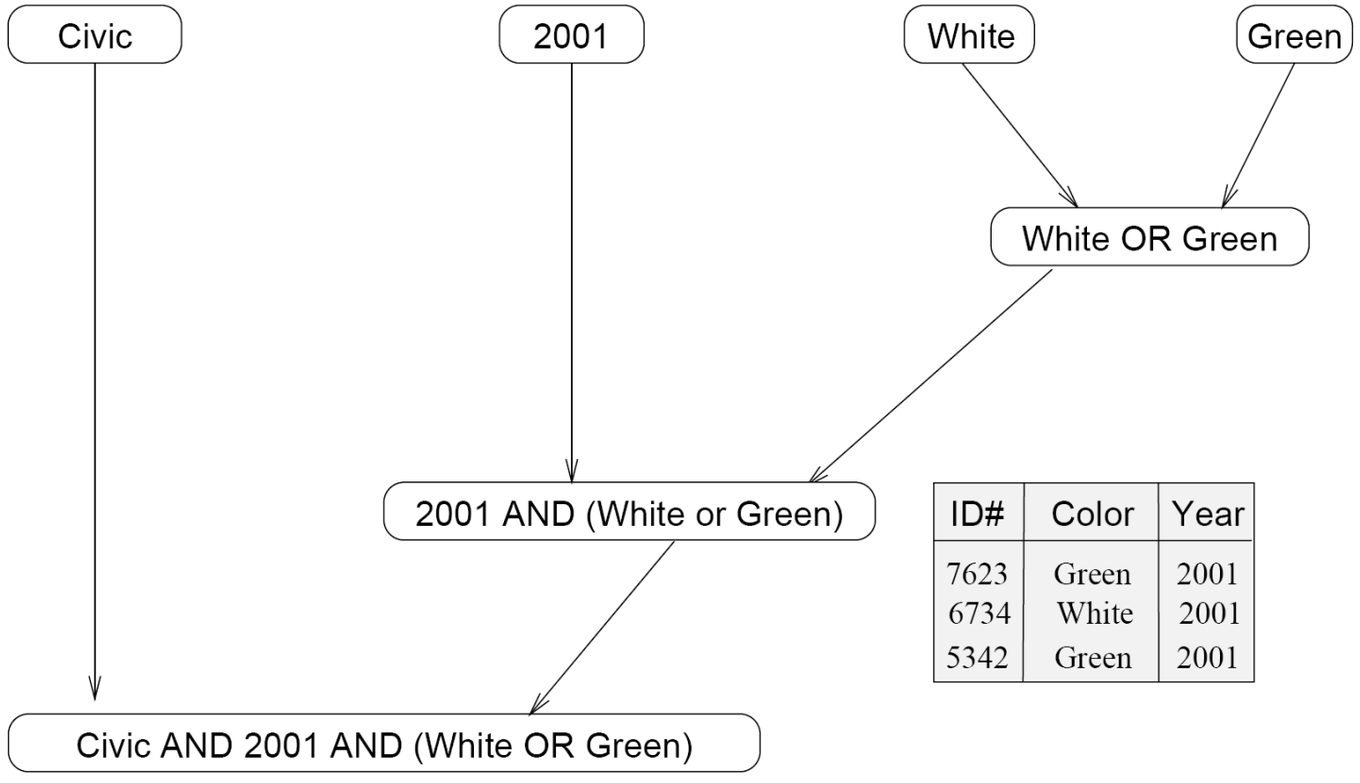
- The same problem can be decomposed into subtasks in other ways as well
- This because the AND operation is associative

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

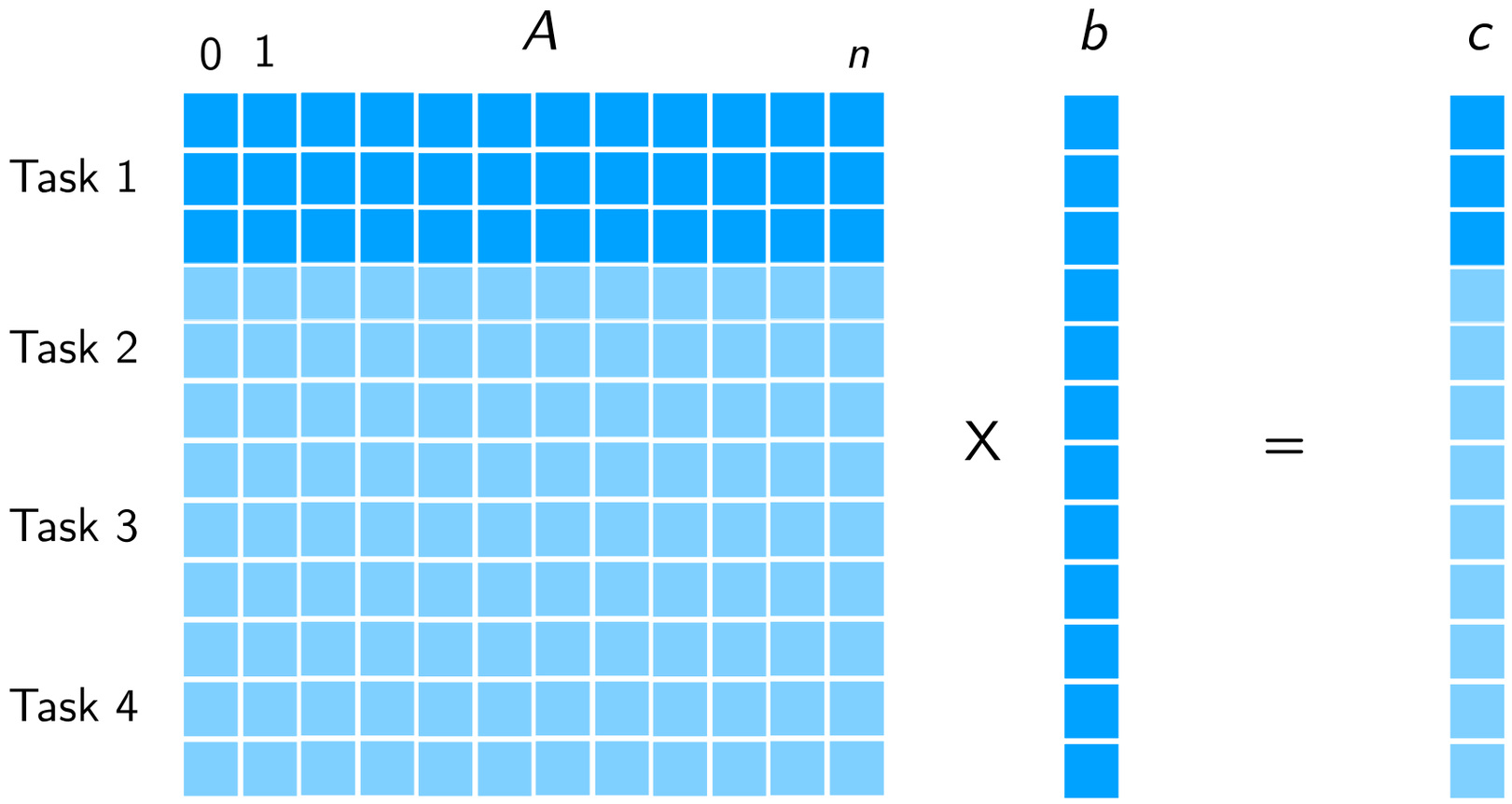


ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

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

ID#	Model	Year	Color
6734	Civic	2001	White

Task Granularity

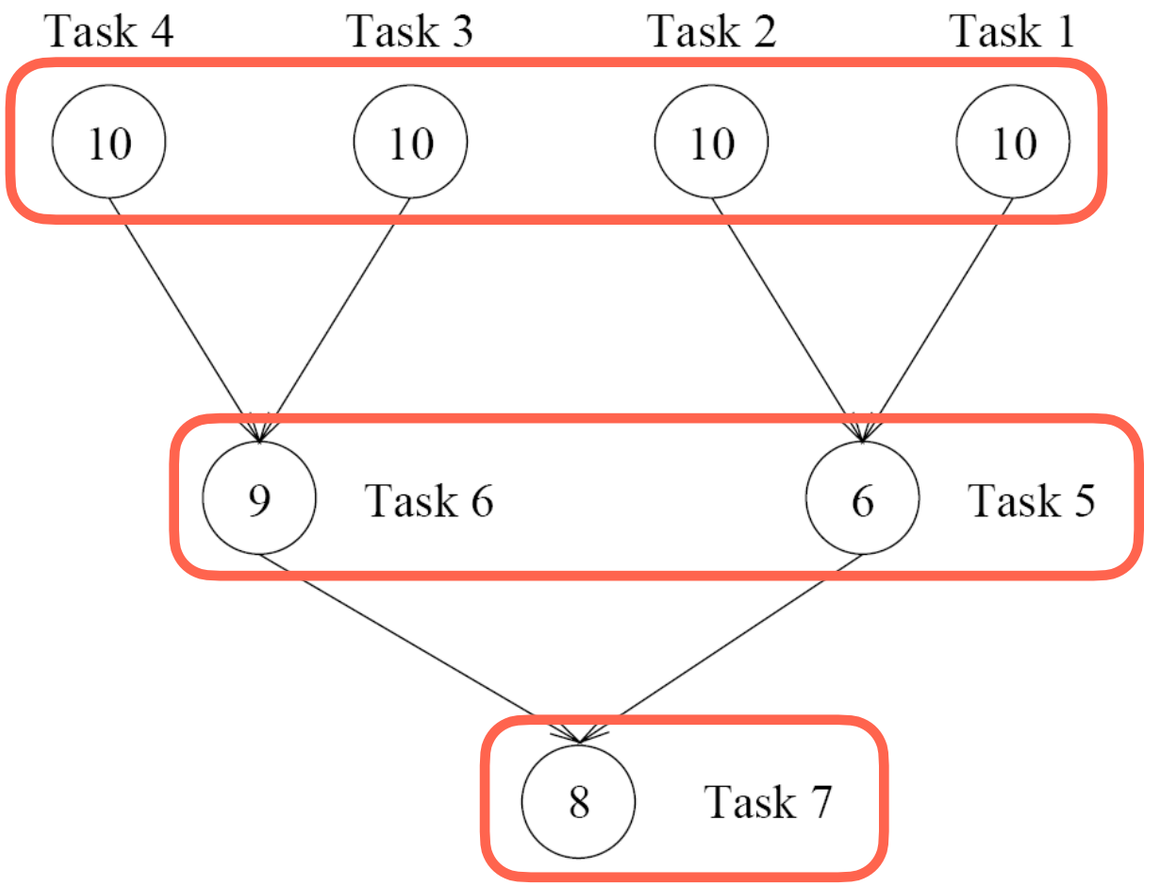


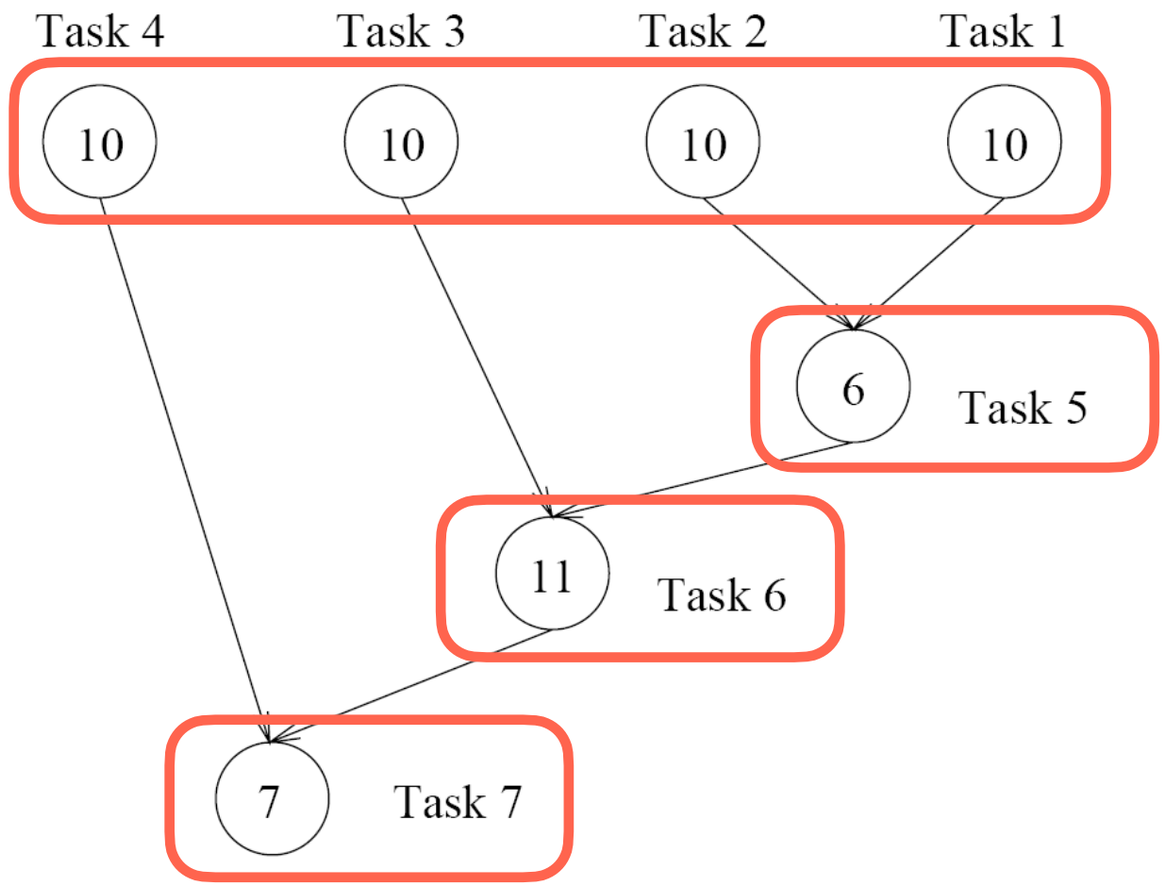
- The **number of tasks** into which a problem is decomposed determines its **granularity**
- Decomposition into a **large** number of tasks results in **fine-grained decomposition**
- Decomposition into a **small** number of tasks results in a **coarse grained decomposition**

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**

Degree of concurrency

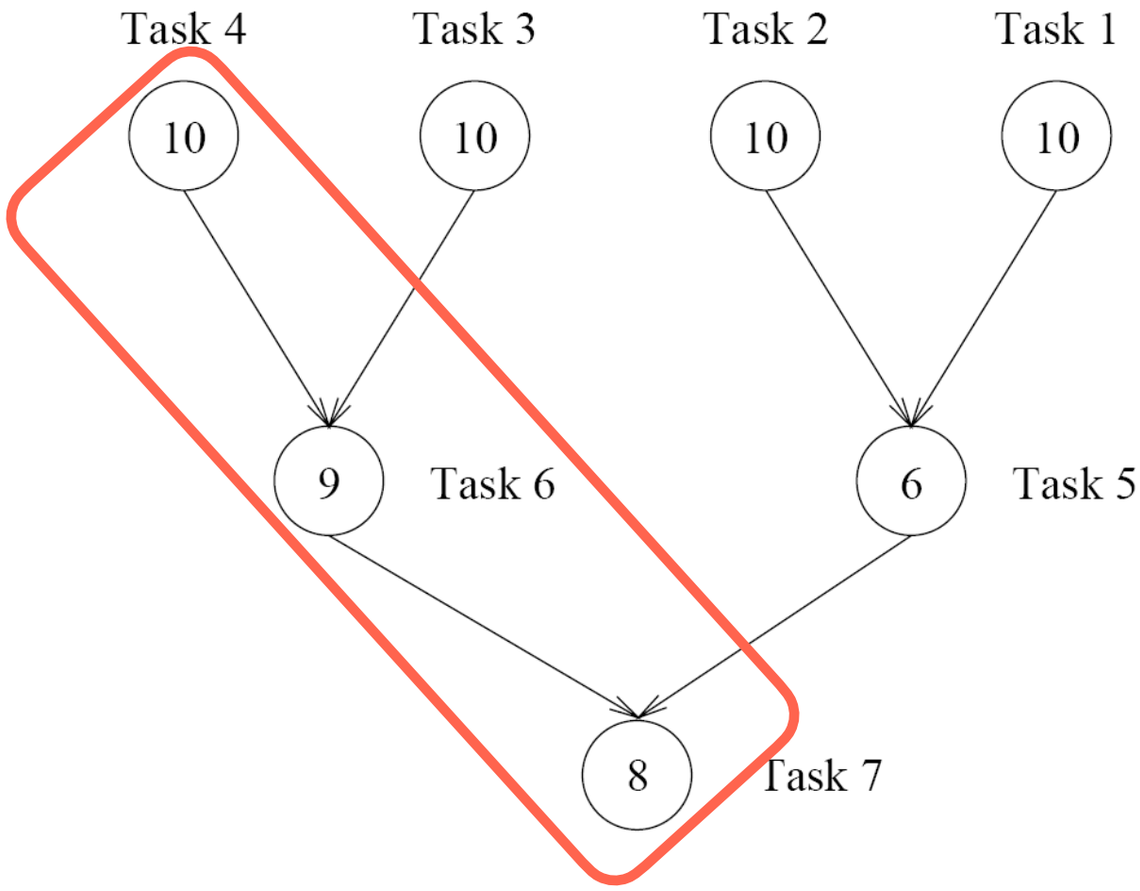


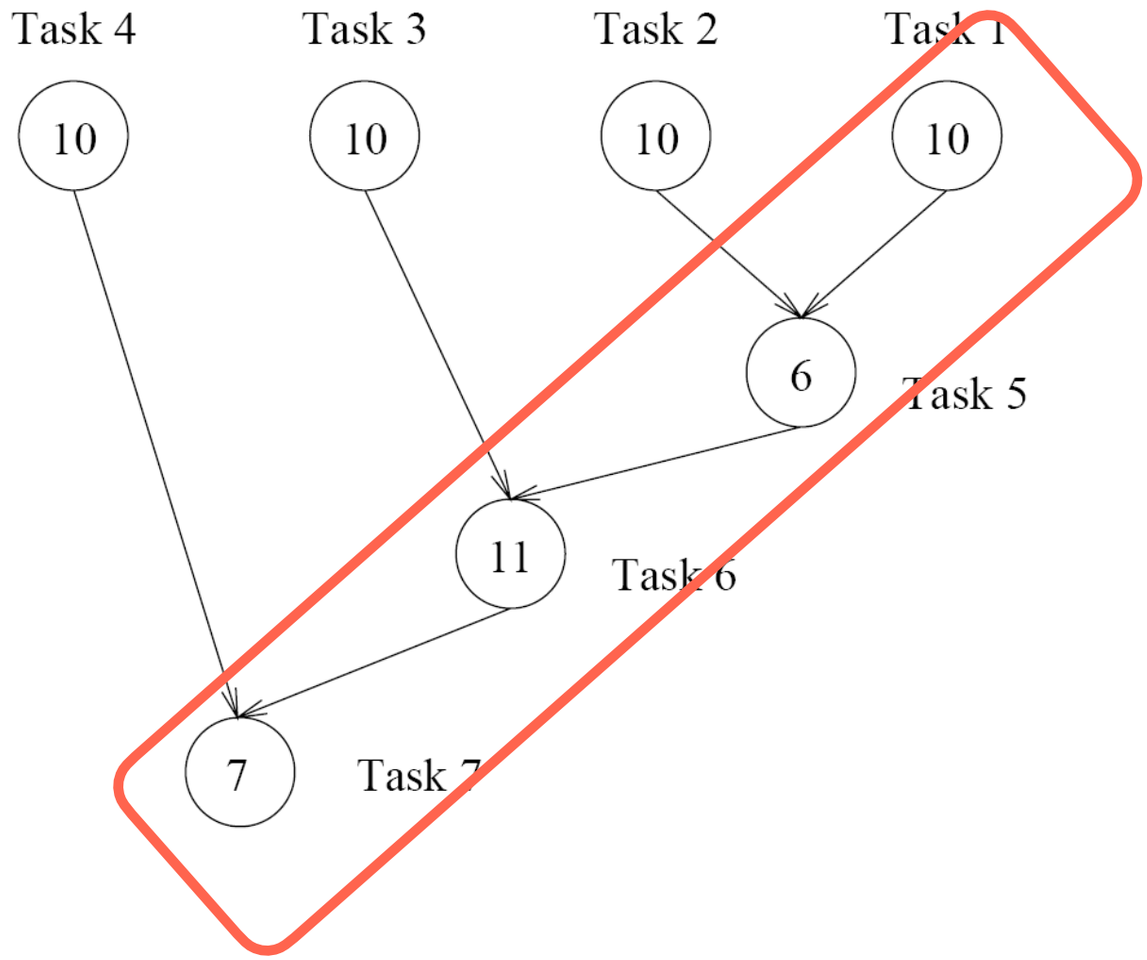


Critical Path

- 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

Critical Path Length





- Task dependency graphs of the two database query
- If each task takes 10 time units, what is the shortest parallel execution time for each decomposition?
- How many processors are needed in each case to achieve this minimum parallel execution time?
- What is the maximum degree of concurrency?

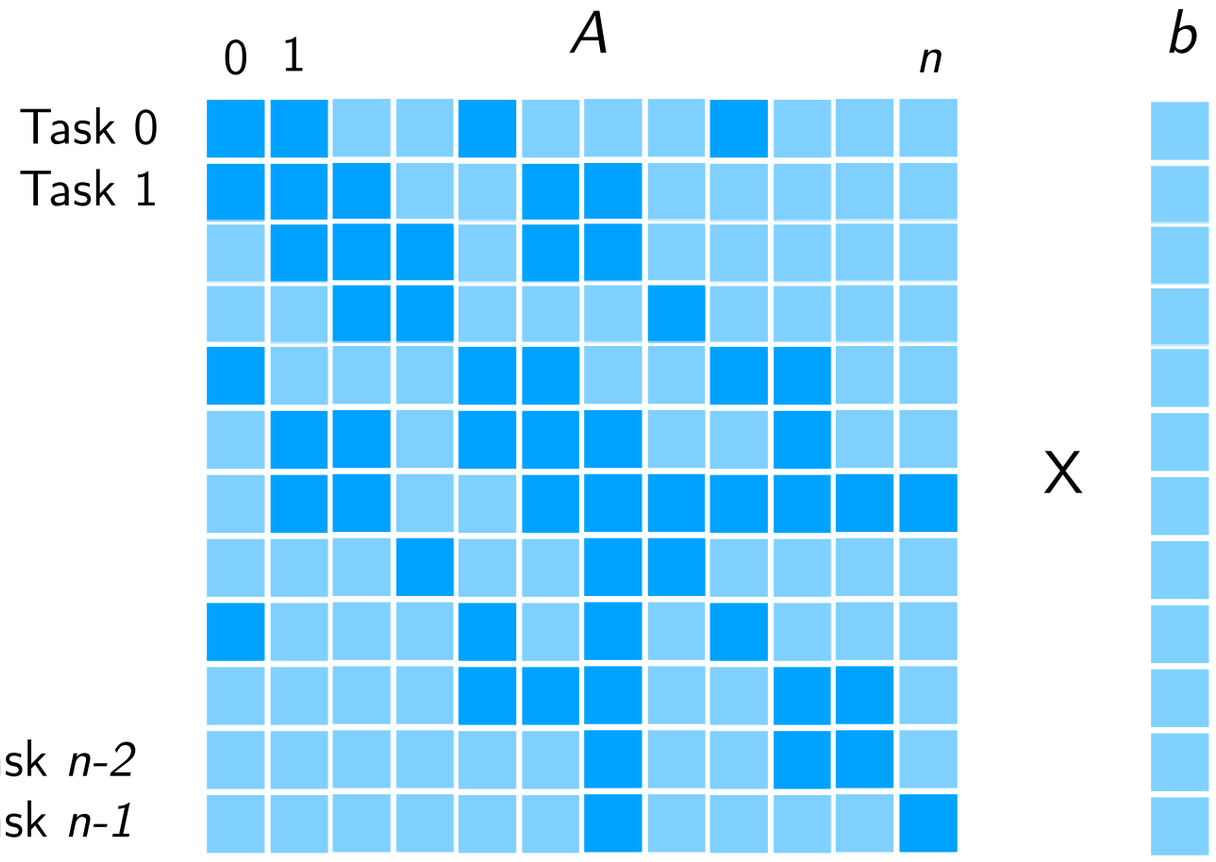
Parallel Performance Limitations

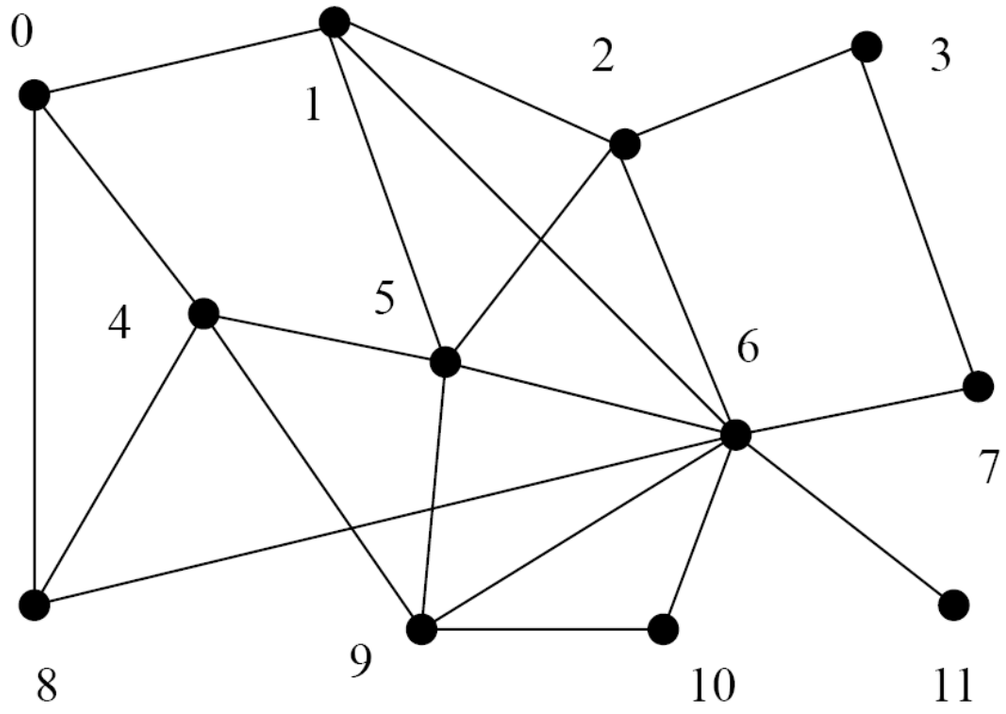
- It would appear that the parallel time can be made **arbitrarily small** by making the decomposition finer in granularity
- There is an **inherent bound** on how fine the granularity of a computation can be
  - For example, in the case of multiplying a dense matrix with a vector, there can be no more 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**

Tasks Interaction Graph

- 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

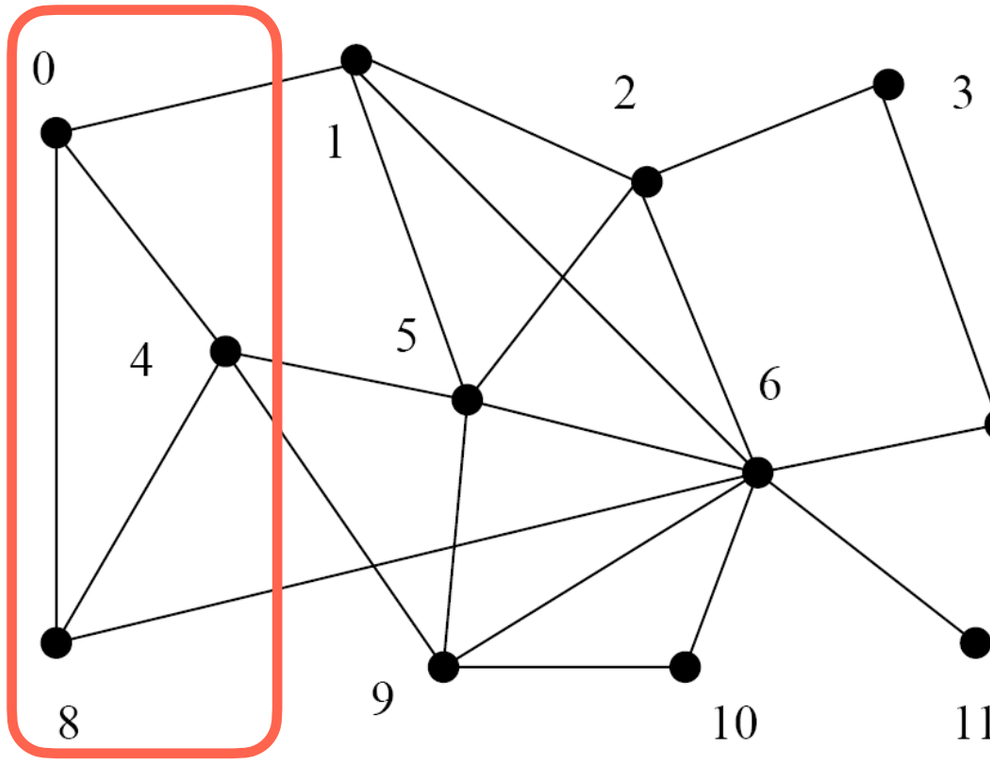




- Matrix-vector multiplication, where the matrix A is **sparse**
- Unlike a dense matrix-vector product though, **only non-zero elements** of matrix A participate in the computation
- Independent task = computation of a single element of the result vector y
- To optimize the usage of memory, ***b* is partitioned among the tasks**
  - b_i assigned to task i (**this is NOT the optimal solution**)
- TIG : models the needed interactions/communications to gather all the missing elements of vector b

TIGs, Granularity, and Communication

- In general, if the granularity of a decomposition is finer, the associated overhead increases
- Example: **sparse matrix-vector multiplication**
  - Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.
  - Thus **node 0 is an independent** task that involves a (useful) computation of one time unit and overhead (communication) of three time units
  - **computation-to-communication ratio: 1/3**
- Now, if we consider **nodes 0, 4, and 8 as one task**, thus increasing task granularity
  - the task has useful computation with a total of **three** time units and communication corresponding to **four** time units (four edges). Clearly, this is a more favourable ratio than the former case (3/4)

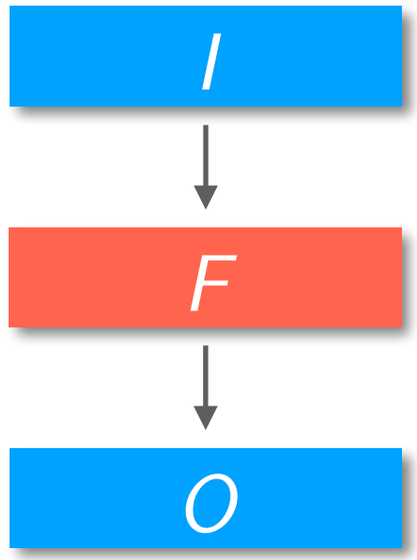


Data parellelism and task parallelism

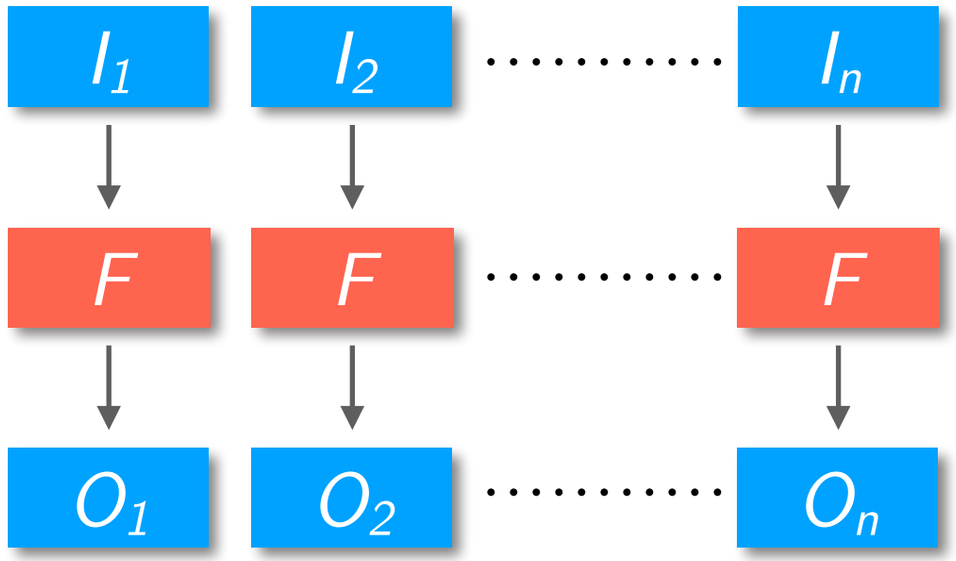
- How do we decompose/partition a problem to obtain the parallel tasks?
- We can decompose on either the **data** or the **control**
- **Data parallelism**
  - Perform the same operation onto different data items in parallel
  - The parallelism comes from a data decomposition (**domain decomposition**), and grows with the size of the data
  - Data parallelism facilitates very high speedups, and is usually very fine-grained (e.g., SIMD and SEE/vectorization)
- **Task parallelism**
  - Perform distinct computations - or tasks - at the same time
  - The parallelism comes from a **control/functional decomposition**

Data parellelism

- Sequential algorithm
  - A function F is applied on a data structure I to yield O
  - $\boldsymbol{O = F(I)}$

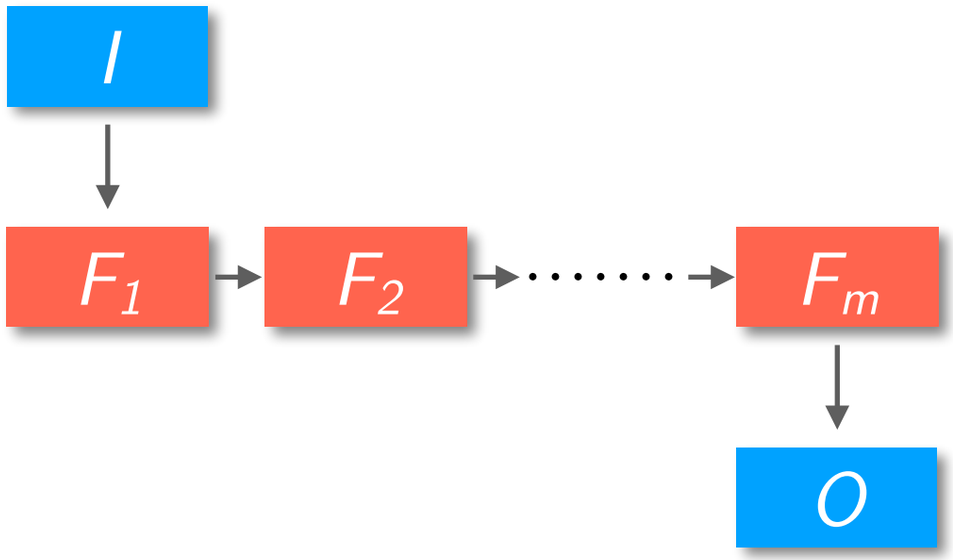


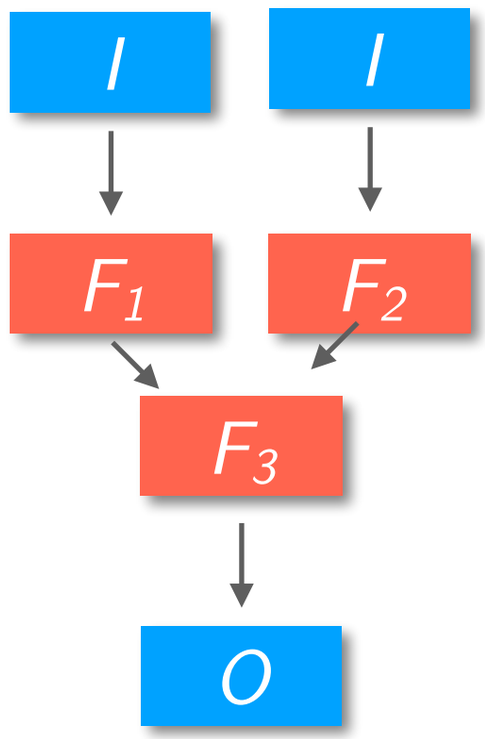
- **Data parallelism**
  - Partition on **data**
  - $I = \{I_1, I_2, \ldots, I_n\}$
  - $\boldsymbol{O_i = F(I_i)}$
  - $O = \{O_1, O_2, \ldots, O_n\}$



Task parallelism

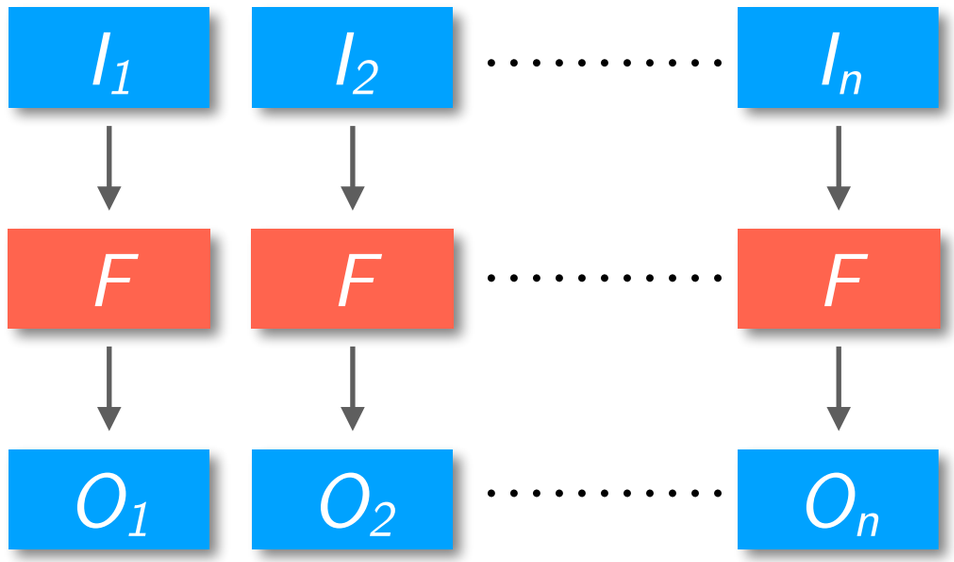
- Sequential algorithm
  - A function F is applied on a data structure I to yield O
  - $O = F(I)$
- **Task parallelism**
  - Partition on **control**
  - **Pipeline** (streaming data)
    - $O = F(I) = F_m(F_{m-1}(\ldots F_1(I)))$
  - **Task graph**
    - $O = F(I) = F_3(F_2(I), F_1(I))$
- **Combination** of task and data parallelism





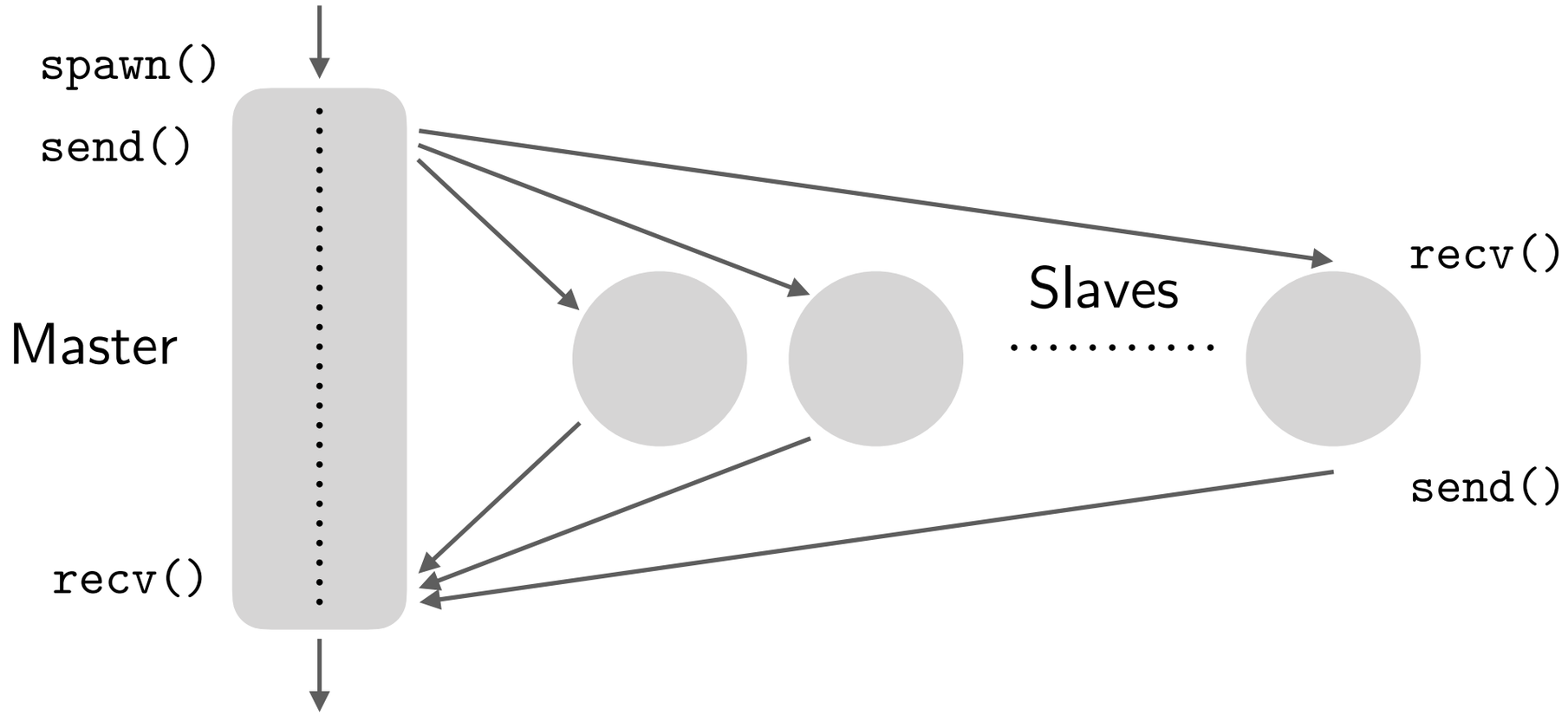
Embarassing Parallel Pattern

- Problems that can be solved with a large set of completely independent sub-tasks
  - Usually **data-parallel**
  - **Completely unconnected** TDG



Master-Slave Implementation

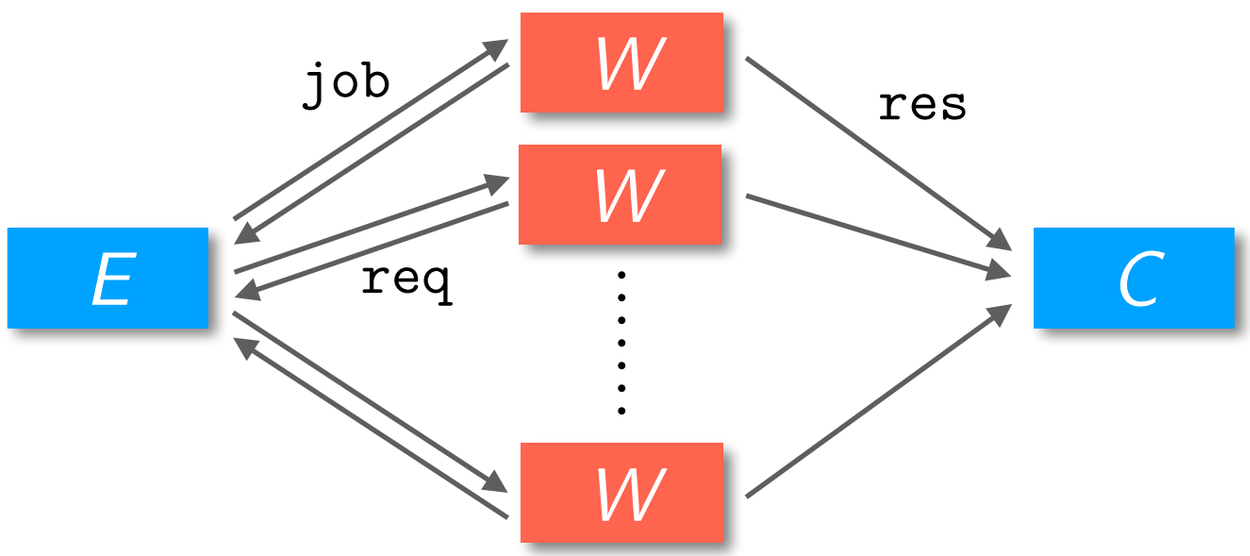
- **Static partitioning** of the input data
- ***N* slaves** are created dynamically
- Each slave is assigned a partition of the input



- This pattern does not perform well if the assigned tasks are **unbalanced**

Farm Pattern

- Three logical entities:
  - **Emitters**: create job **statically** or **dynamically**
  - **Workers**: ask for a job to execute
    - **On-demand**: requests can be issues when previous job has been completed
    - **Pre-fetching**: when the current job has been received, ask for the next one
  - **Collectors**: gather the results of the computation



How many workers?

- What is the **right number of workers** ?
- Given
  - $\boldsymbol{T_{work}}$, time to complete a task
  - $\boldsymbol{T_{comm}}$, communication time (send task + receive results)
- The communication bandwidth is $\boldsymbol{B_{comm}} = 1 \,/\, T_{comm}$
- The computation bandwidth is $\boldsymbol{B_{comp}} = n \,/\, T_{work}$ where n is the number of workers

- The **optimal** $\boldsymbol{n}$ is when $\boldsymbol{B_{comm}} = \boldsymbol{B_{comp}}$:

$$1 \,/\, T_{comm} = n \,/\, T_{work}$$

$$\boldsymbol{n} = \boldsymbol{T_{work}} \,/\, \boldsymbol{T_{comm}}$$

- In addition:
  - We could **use pre-fetching** to overlap communication and computation
  - Use a **single master** to emit jobs and collect results, and interpret a result as a request for the next job

Divide and conquer pattern

- The same as in sequential
  - Decompose into **smaller independent** problems
  - Use **recursion**
  - Parallelism comes **naturally** from task independence

- Sometimes it is used in a parallel setting, even if it may not be a good idea in sequential setting
  - *Example*: find the minimum of a list
  - *Goal*: achieve decomposition of the given task
  - *Pro*: decomposition can be decided **dynamically at run-time**, in order to keep the best parallelism degree
    - Especially when it is difficult to predict the cost of each task

- Issues:
  - **Initial** and **final stages** have small parallelism degree
  - Sub-tasks may **not** be **balanced**

Other data decompositions

- So far, we have only considered **input** data decomposition

- It is also possible to decompose w.r.t. the **output** data
  - A processor produces a partition of the output
  - The input data is partitioned accordingly
  - More often input data is partially/totally replicated

- By partitioning on the **output**
  - We can infer the **computation assignment** (scheduling) from this **data partitioning**
  - Each processor should be in charge of producing its own output partition
  - **Owner computes rule**

Problems in output partitioning

- Suppose a problem
  - Has input I being partitioned in $I = \{I_1, I_2, ..., I_n\}$
  - The algorithm must compute $O = f(I)$
- The output can be partitioned as $O = \{O_1, O_2, ..., O_n\}$
  - In good cases $O_1 = f(I_1)$, $O_2 = f(I_2)$, ...
  - In many cases $O_1 = f(1, I)$, $O_2 = f(2, I)$, ...
    - The output is a function of the **whole input**

- There are cases where the partial outputs should be **aggregated**
  - $O =$ aggregate$(O_1, O_2, ..., O_n)$

Pipeline Pattern

- Pipeline is a special kind of task-parallelism
- The computation is partitioned into stages, that are executed sequentially
- **Asymptotically**, a pipeline achieves a speed-up equal to the number of stages

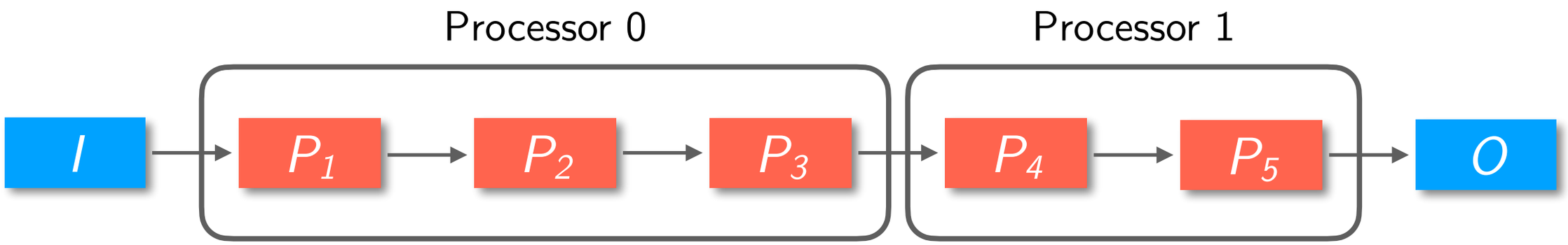
Pipeline in practice

Pipelining is efficient when we have:

- **Job Parallelism**:
  - **Independent** instances of the same problem (different data, same algorithm)
  - The algorithm can be decomposed into a cascading sequence of tasks
  - The various instances (data) may form the input stream
- **Data Parallelism**:
  - The algorithm processing a portion of the input can be decomposed in a cascading sequence of tasks
  - The partitions of the input become the stream of the pipeline
- **Pipeline without stream**:
  - The job can be decomposed in a sequence of tasks generating/working on partial results
  - Each stage **early feeds** the next one

Pipeline issues

- The **throughput** (amount of data per second) of the pipeline depends on the **slowest stage**
- Given a slow stage
  - Next stages are in **idle** waiting
  - Previous stages may have **buffering** problems
- Possible solution:
  - **Coarser grain**: assign many "fast" tasks or a few "slow" tasks to a processor
  - Decrease the parallelism degree



Allocating tasks to processors

- A good design of a parallel algorithm implies to **reduce** communications, and **balance** the process workload

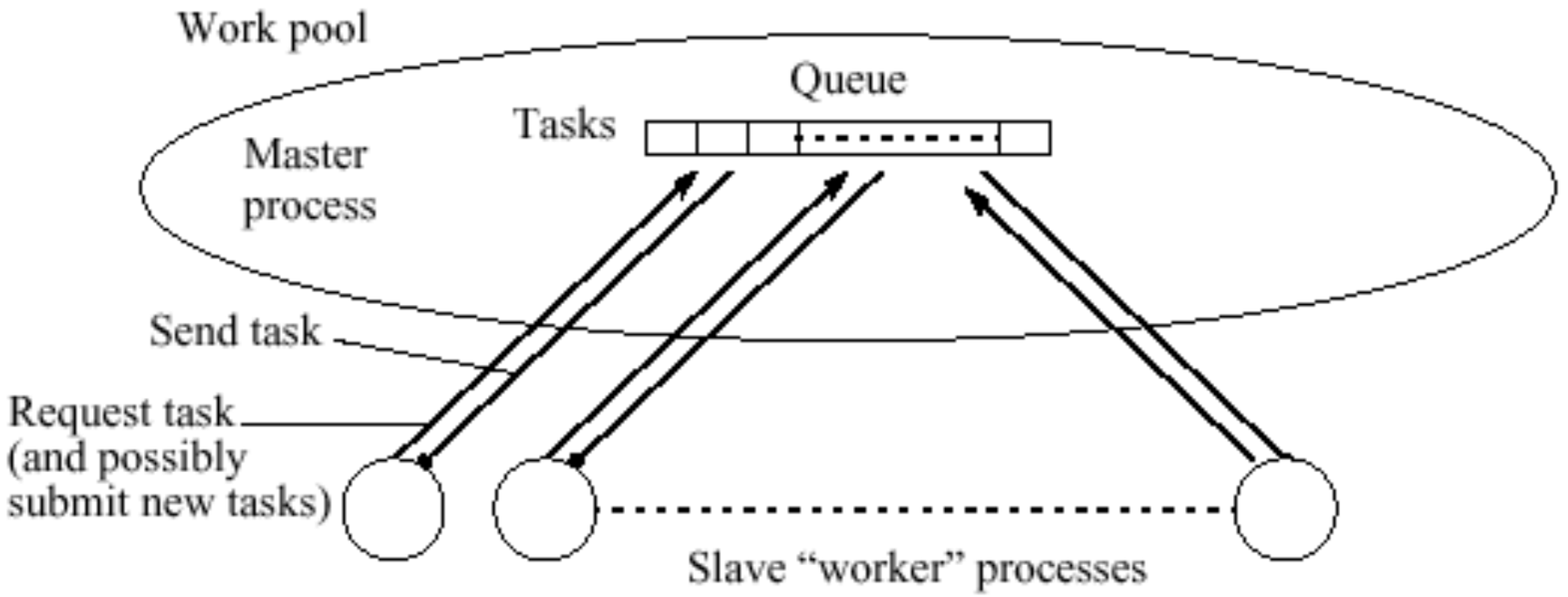
- When interactions are not so regular, we may still exploit a **static** approach
  - Group small tasks to balance the load
  - Map tasks to minimize communications

- When
  - Tasks do not have uniform cost
  - Tasks have unknown dependencies
  - Tasks are created at run-time

Scheduling and mapping must be dynamic!

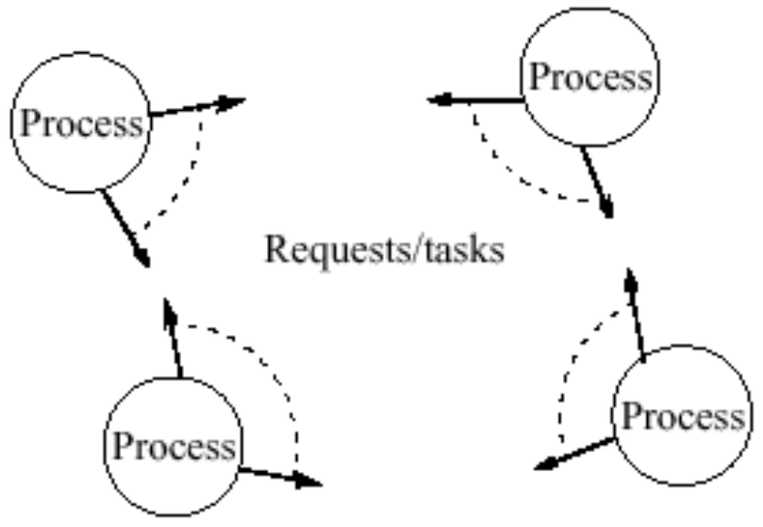
Centralised Load Balancing

- Slaves can submit new tasks
- The master maintains a dynamic queue of available tasks



Distributed Load Balancing

- Goal:
  - Remove centralisation
  - Favour data exchange among neighbours



- **Push/Sender-Initiated**:
  - The worker that generates a new task sends it to another worker (if it cannot handle it)
- **Pull/Receiver-Initiated**:
  - When a worker is idle, it asks other workers for a job to execute

Partner Selection

The partner task can be selected:

- At **random**, among all the workers
- **Global Round Robin** (GRR):
  - A global variable/marker points to the "next" worker
  - A Worker needing a partner reads and increments the global variable
  - Implements a global Round Robin
- **Local Round Robin** (LRR):
  - Every processor keeps a private (un-synchronised) pointer to the next available worker
  - No overhead due to sharing a global variable
  - Approximates a global Round Robin

Static vs Dynamic Mapping