Chapter 3

Dichotomy of Parallel Computing Platforms

Flynn's Classical Taxonomy

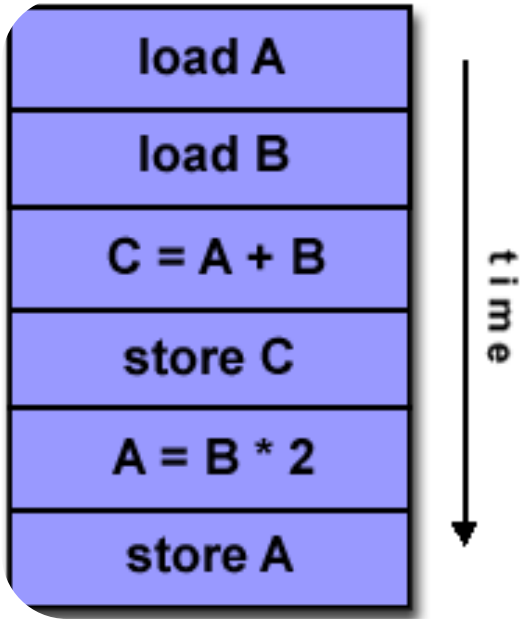
- **Flynn's taxonomy (since 1966) distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of *Instruction* and *Data***

SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

Slide source: LLNL

Single Instruction, Single Data (SISD)

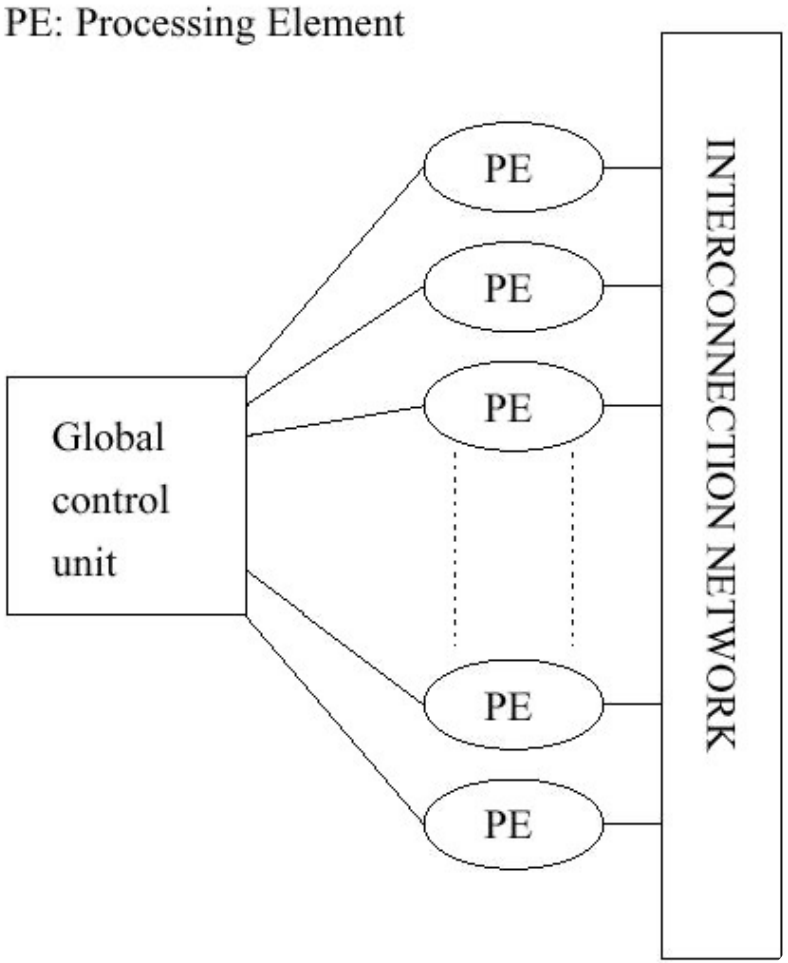
- **A serial (non-parallel) computer**
  - Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
  - Single data: only one data stream is being used as input during any one clock cycle
- **Deterministic execution**
  - This is the oldest and until recently, the most prevalent form of computer
  - Examples: most PCs, single CPU workstations and mainframes



Slide source: LLNL

Single Instruction, Multiple Data (SIMD)

- **A type of parallel computer**
  - Single instruction: All processing units execute the same instruction at any given clock cycle
  - Multiple data: Each processing unit can operate on a different data element
- **This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units**
  - Synchronous (lockstep) and deterministic execution



Slide source: LLNL

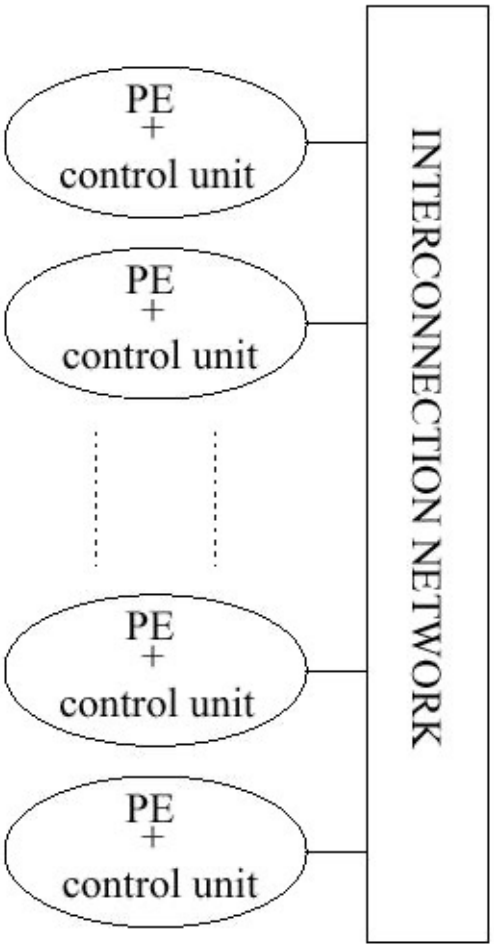
Multiple Instruction, Single Data (MISD)

- **Few actual examples of this class of parallel computer have ever existed**
- **Some conceivable examples might be:**
  - Multiple frequency filters operating on a single signal stream
  - Multiple cryptography algorithms attempting to crack a single coded message

Slide source: LLNL

Multiple Instruction, Multiple Data (MIMD)

- **Currently, the most common type of parallel computer**
  - Multiple Instruction: every processor may be executing a different instruction stream
  - Multiple Data: every processor may be working with a different data stream
- **A variant: single program multiple data (SPMD)**
- **Execution can be synchronous or asynchronous, deterministic or non-deterministic**
  - Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs



Slide source: LLNL

SIMD vs. MIMD

- Require less hardware (only one global control unit and one copy of the program)
- Specialized hardware architectures and application characteristics
- Support irregular problems poorly
- Poor resource utilization in the case of conditional execution

- Store the program and operating system at each processor

- Can be built from inexpensive off-the-shelf components
- Support irregular problems well
- With relatively little effort in a short amount of time

Dichotomy of Parallel Computing Platforms

- **Based on the logical and physical organization of parallel platforms**
- **Logical Organization (from a programmer's perspective):**
  - Control structure: Ways of expressing parallel tasks
  - Communication model: Mechanisms for specifying interaction between tasks
- **Physical Organization (actual hardware organization)**
  - Architecture
  - Interconnection networks

Control Structure of Parallel Platforms

- **Parallel tasks can be specified at various levels of granularity**
  - One extreme: a set of programs
  - The other extreme: individual instructions within a program
  - A range of models between them

- **Example:**

  Adding two vectors:

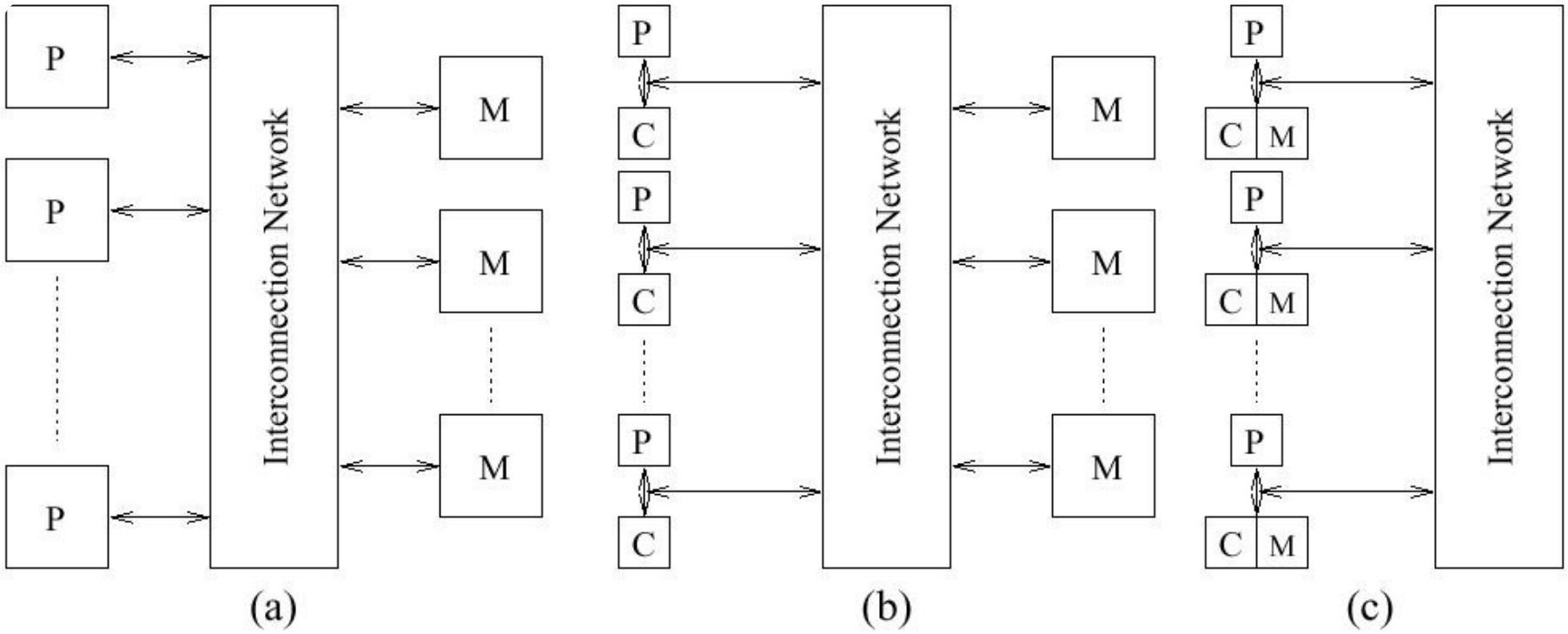
  1. for (i = 0; i< 1000; i++)
  2. c[i] = a[i] + b[i]

  All iterations of the loop are independent of each other

Communication Model: Shared-Address-Space Platforms

- **Support a common data space that is accessible to all processors**
- **If supporting SPMD programming => multiprocessors**
- **Memory:**
  - Local (exclusive to a processor)
  - Global (common to all processors)
- **Two types of architectures:**
  - Uniform memory access (UMA): the time taken by a processor to access any memory word in the system is identical
  - Non-uniform memory access (NUMA): the time taken to access certain memory words is longer than others
- **NUMA and UMA are defined in terms of memory access times, not cache access times**

Typical Shared-address-space Architectures



UMA without caches

UMA with caches

NUMA

Typical shared-address-space architectures: (a) Uniform-memory access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

NUMA and UMA Shared-Address-Space Platforms

- **The distinction between NUMA and UMA platforms is important from the point of view of algorithm design**
  - NUMA machines require locality from underlying algorithms for performance
- **Programming these platforms is easier since reads and writes are implicitly visible to other processors**
  - However, read-write data to shared data must be coordinated (we have discussed this in greater detail when we talk about threads programming)
- **Caches in such machines require coordinated access to multiple copies**
  - This leads to the cache coherence problem
- **A weaker model of these machines provides an address map, but not coordinated access**
  - These models are called non cache coherent shared address space machines

Global Memory Space

- **Ease programming**
- **Read-only interactions:**
  - Invisible to programmers
  - Same as in serial programs
  - Reduce the burden on programmers
- **Read/write interactions:**
  - Mutual exclusion for concurrent accesses
  - Such as locks and related mechanisms
- **Programming paradigms:**
  - Threads (POSIX, NT)
  - Directives (OpenMP)

Caches in Shared-address-space

- **Two major tasks:**
  - Address translation mechanism to locate a memory word in the system
  - Well-defined semantics over multiple copies (cache coherence)
    - Hardware support
    - Software support (handled by programmers directly, *get* and *put* …)
    - Updated vs. invalidated

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute
- It is possible to provide a shared address space using a physically distributed memory

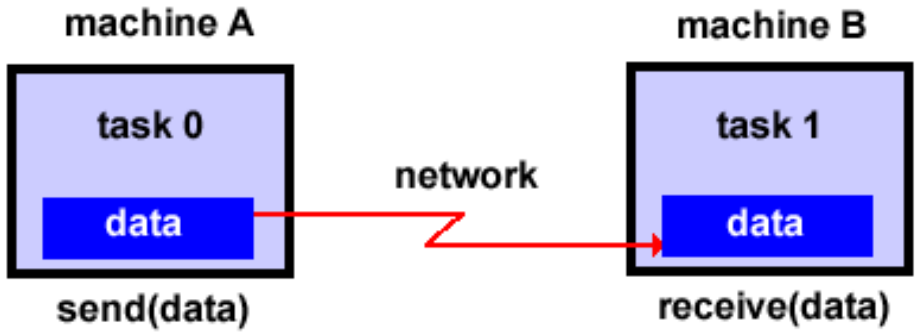
Logical View vs. Physical Organization

- **Logical view:**
  - shared-address-space
  - non-shared-address-space

- **Physical organization:**
  - Shared-memory computers == UMA
  - Distributed-memory computers
    - If shared-address-space => NUMA

Message-Passing Platforms

- **Logical machine view: consisting of *p* processing nodes, each with its own exclusive address space**
  - Example: clustered workstations, non-shared-address-space multicomputers; IBM SP, SGI Origin 2000
- **Message passing: interactions, synchronization, and data & work transfer**
  - Programming paradigm: send and receive
  - APIs: Message Passing Interface (MPI) and Parallel Virtual Machine (PVM)



Shared Memory vs. Distributed Memory

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4	Cray T3E Maspar IBM SP2
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Limited memory bandwidth	New architecture Point-to-point communication	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	10s ISVs

Slide source: LLNL

DSM/SVM

- **Distributed Shared Memory (DSM) or Shared Virtual Memory (SVM)**

- **Page-Based Access Control**
  - Leverage the virtual memory support
  - Manage main memory as a fully associative cache on the virtual address space
  - Embed a coherence protocol in the page fault handler
- **Object-Based Access Control**
  - Flexible
  - No false sharing

Generations of Software DSM

- **Three generations of software DSM:**

1. **Sequential consistency model in single CPU workstation clusters, such as *Ivy***
2. **Relaxed consistency model in single CPU workstation clusters, such as *TreadMarks***
3. **Relaxed consistency model and multi-threading on a network of multiprocessor computers, such as *Brazos* and *Strings***

Parallel Algorithm Design

Steps in Parallel Algorithm Design

- **Identifying portions of the work that can be performed concurrently**
- **Mapping the concurrent pieces of work onto multiple processes running in parallel**
- **Distributing the input, output, and intermediate data associated with the program**
- **Managing accesses to data shared by multiple processors**
- **Synchronizing the processors at various stages of the parallel program execution**

Note:

- Performance benefit vs. Computational and storage resources
- Different choices on different parallel architectures or under different parallel programming paradigms (rely on programmers, not compilers)

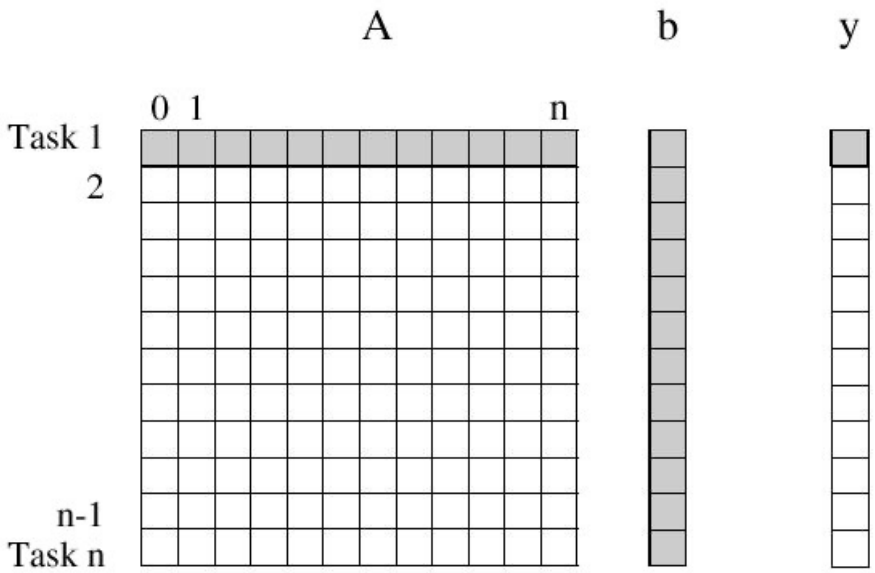
Decomposition

- **Decomposition: dividing a computation into smaller parts, some or all of which may be executed in parallel**
- **Tasks: programmer-defined units (arbitrary size, indivisible)**
- **Reducing execution time: simultaneous execution of multiple tasks**

- **Ideal decomposition:**
  - All tasks have similar sizes
  - Tasks are NOT waiting for each other; NOT sharing resources

Dense Matrix-Vector Multiplication

- The *i*th element *y[i]* of the product vector is the dot-product of the *i*th row of A with the input vector *b*
- A task: the computation of each *y[i]*
- All tasks are independent (performed all together or in any sequence)



$$y[i] = \sum_{j=1}^{n} A[i, j].b[j]$$

Dependency Graphs

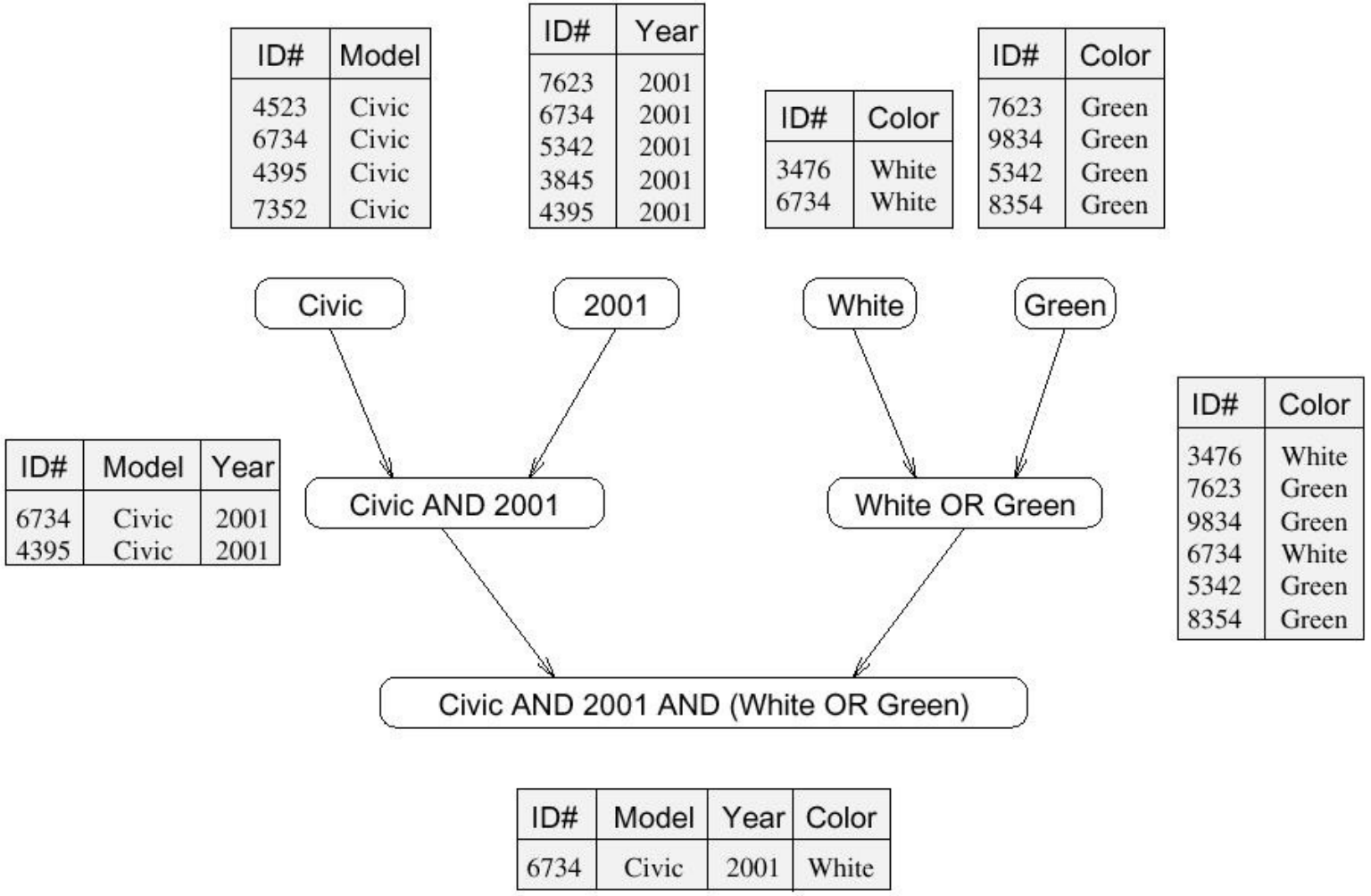
- **Task-dependency graph: an abstraction to express dependencies among tasks and their relative order of execution**
  - Directed acyclic graphs (DAG)
    - Can be disconnected
  - Nodes: tasks
  - Directed edges: dependencies amongst tasks
    - Edge set could be empty
- **Rule: The task corresponding to a node can be executed only when all tasks connected to this node by incoming edges have completed**
- **Different decomposition methods might generate different tasks and their dependency graphs**
  - The fewer directed edges, the better (detect parallelism)

Database Query Processing

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

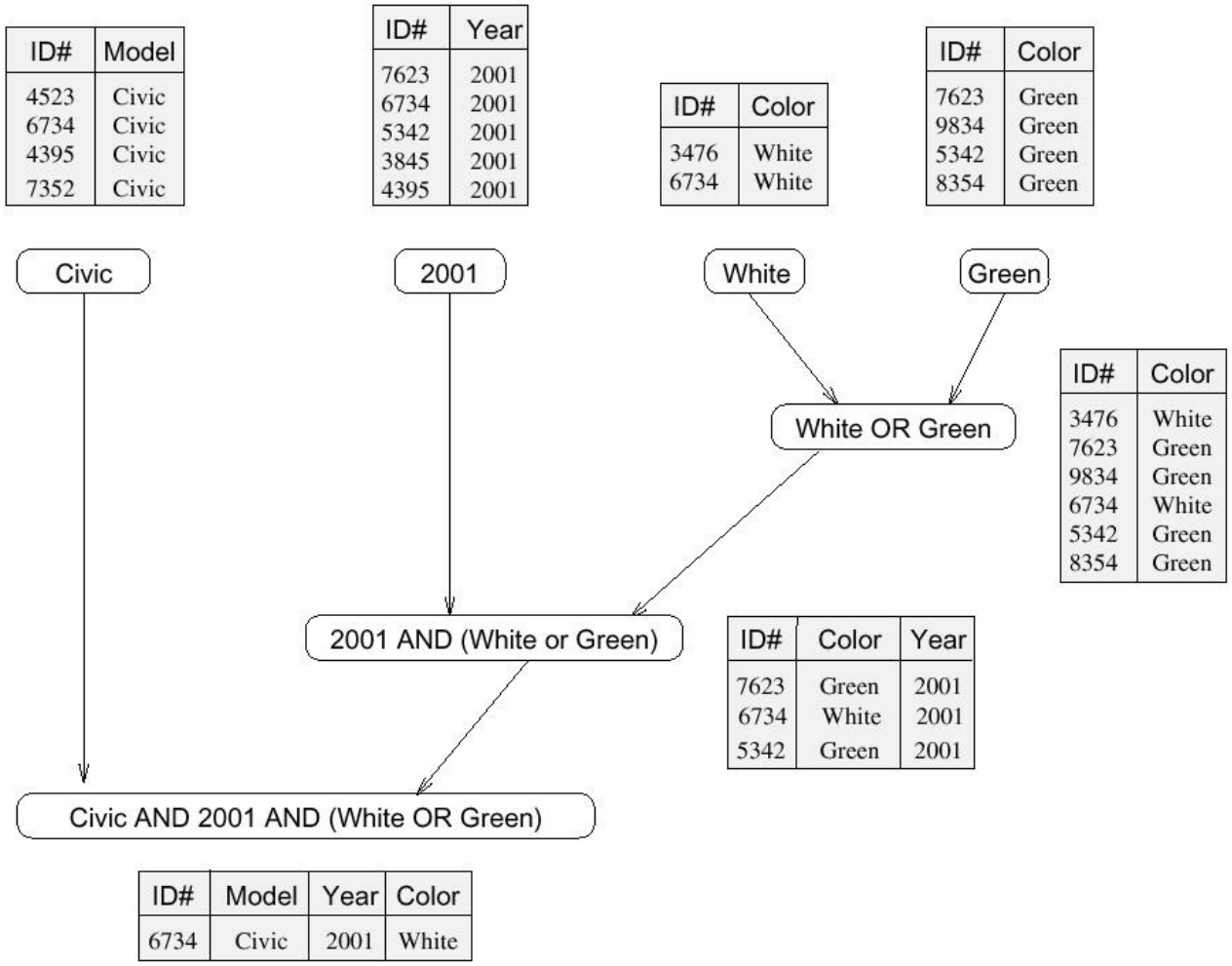
Query: MODEL = “Civic” AND YEAR=“2001” AND (COLOR=“Green” OR COLOR=“White”)

Database Query Processing (cont'd)



Query: MODEL = "Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")

Database Query Processing (cont'd)

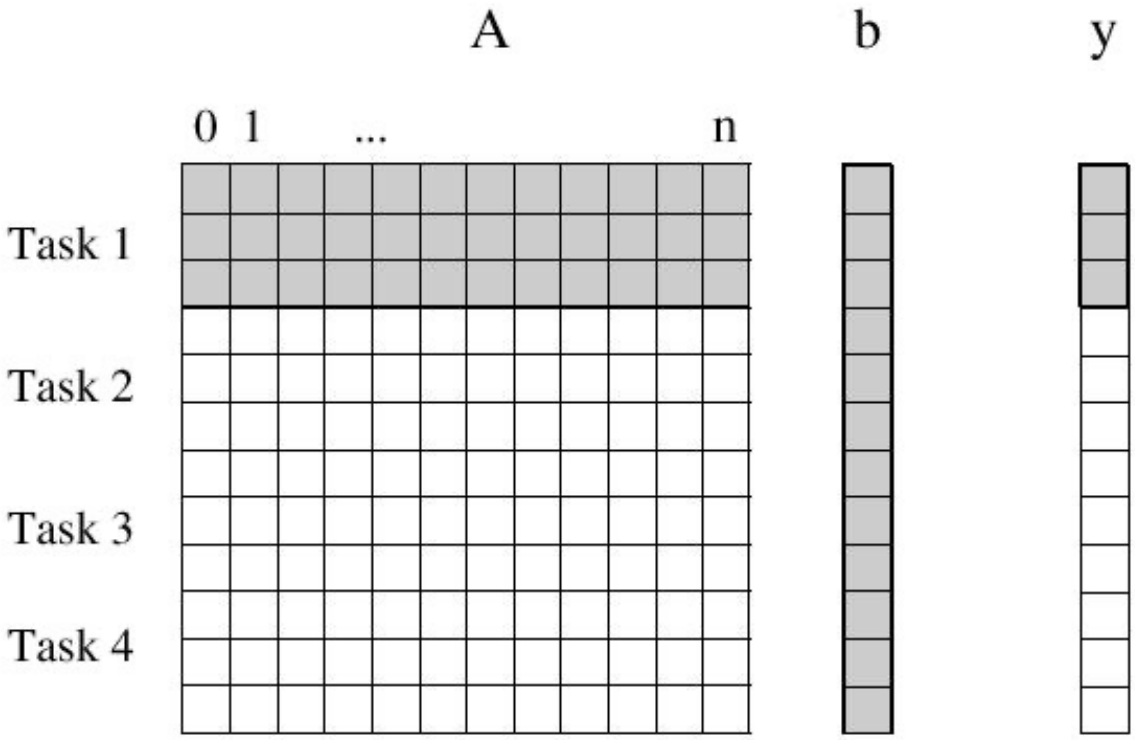


Different ways of arranging computations can lead to different task-dependency graphs with different characteristics

Granularity

- **Granularity of the decomposition: the number and size of tasks into which a problem is decomposed**
- **Fine-grained: a decomposition into a large number of small tasks**
  - Exploit parallelism thoroughly
- **Coarse-grained: a decomposition into a small number of large tasks**

Decomposition of Dense Matrix-Vector into Four Tasks



$$y[i] = \sum_{j=1}^{n} A[i, j].b[j]$$

A coarse-grained decomposition: four tasks, where each task computes n/4 of the entries of the output vector of length n

Concurrency

- **Maximum degree of concurrency: the maximum number of tasks that can be executed simultaneously in a parallel program at any given time**
  - Equal to or less than the total number of tasks due to dependencies among the tasks
  - For database query example: 4
  - If task-dependency graphs are trees: equal to the number of leaves
- **Average degree of concurrency: the average number of tasks that can be run concurrently over the entire duration of execution of the program**
- **Both maximum and average degree of concurrency:**
  - Usually increase as the granularity of tasks becomes smaller (finer)
  - Depend on the shape of the task-dependency graph
  - Same granularity does NOT guarantee the same degree

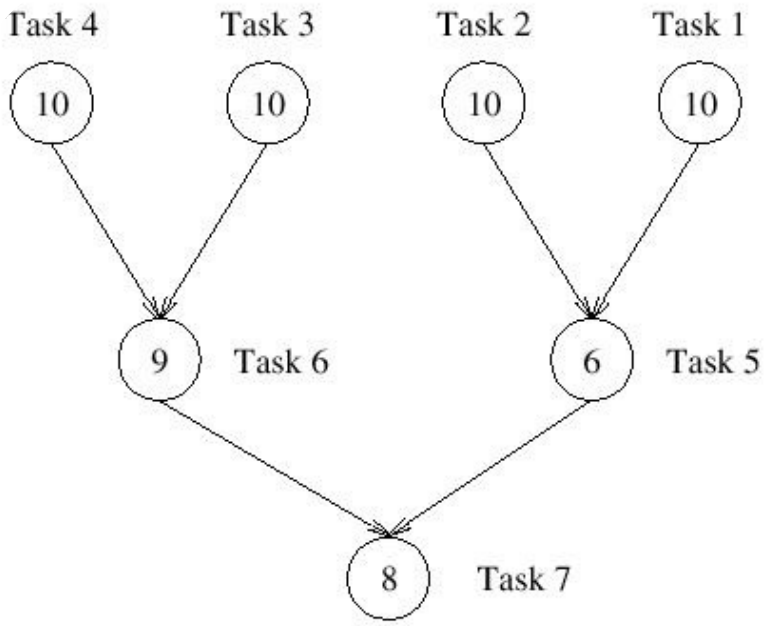
Critical Path

- **Determines the average degree of concurrency**
- **Nodes:**
  - Start nodes: no incoming edges
  - Finish nodes: no outgoing edges
- **Critical path: the longest directed path between any pair of start and finish nodes**
- **Critical path length: the sum of the weights of nodes along the critical path**
- **Average degree of concurrency:**

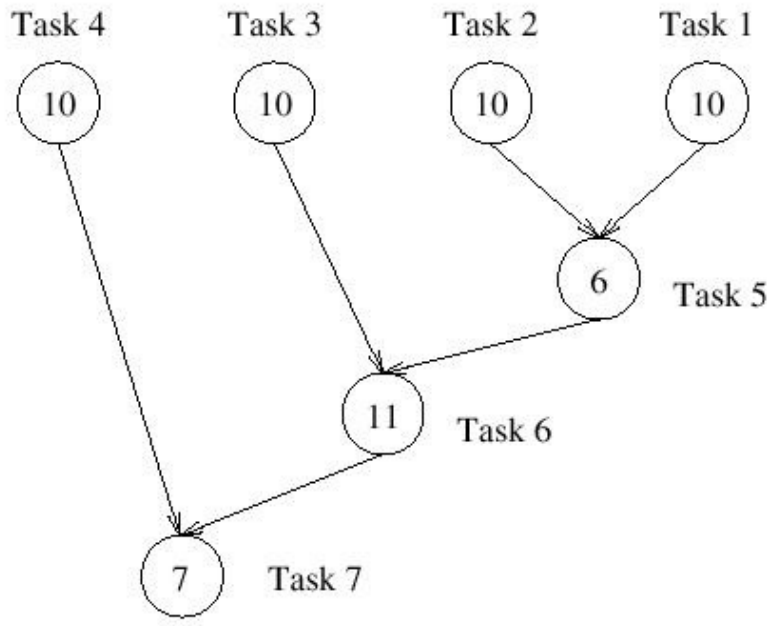
  (the total amount of work) / (the critical path length)
- **A shorter critical path favors a higher degree of concurrency**

Average Degree of Concurrency

- **The shape of the task-dependency graphs can change the degrees of concurrency even if graphs are in the same granularity**



Total work = 63
Critical path length = 27
Max. Deg. Conc. = 4
Ave. Deg. Conc. = 63/27 = 2.33



Total work = 64
Critical path length = 34
Max. Deg. Conc. = 4
Ave. Deg. Conc. = 64/34 = 1.88

Limited Granularity

- **It may appear: increasing the granularity of decomposition and utilizing the resulting concurrency**

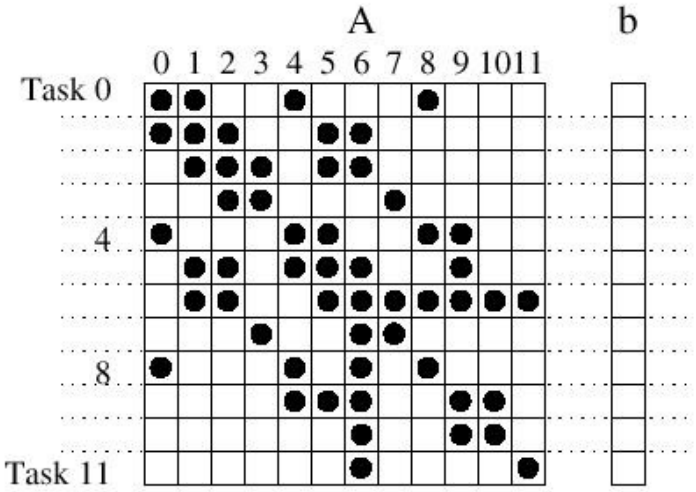
  => perform more tasks in parallel
- **There is an inherent bound on how fine-grained a decomposition a problem permits**
- **Example:**
  - n^2 multiplications and additions in matrix-vector multiplication
  - The problem cannot be decomposed into more than $O(n^2)$ tasks

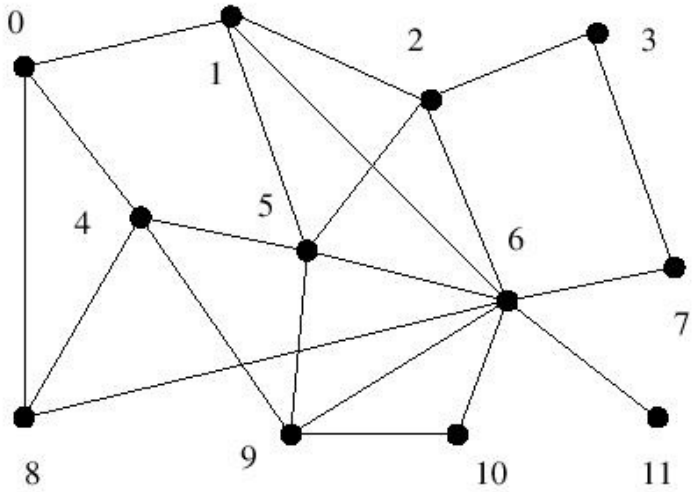
Restrictions on Speedup

- **Speedup: ratio of serial to parallel execution time**
- **Restrictions on obtaining unbounded speedup from parallelization:**
  - Limited granularity
  - Degree of concurrency
  - Interaction among tasks running on different physical processors

Sparse Matrix-Vector Multiplication

- **Compute the product *y* = *Ab* of a sparse *n x n* matrix *A* with a dense *n x 1* vector *b***
- **Can be optimized significantly by avoiding computations involving the zeros**
- **Task *i* :** $\sum_{0 \le j \le 11, A[i,j] \neq 0} A[i, j].b[j]$
  - Computes *y[i]*
  - Owns row *A[i, *]* and *b[i]*
  - Requires access to many elements of *b* owned by other tasks





Processes

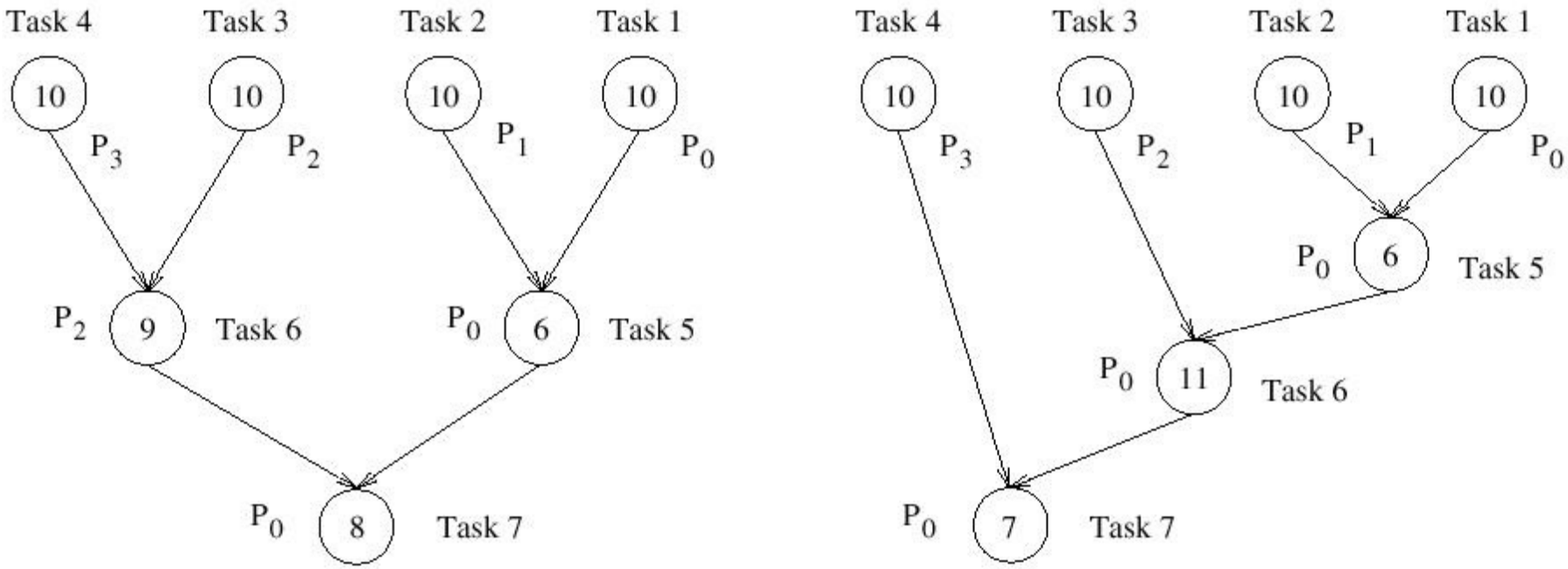
- Process (computing agent that performs tasks) : an abstract entity that uses the code and data of a task to produce the output of the task within a finite amount of time
- A process: Not the rigorous operating system definition of a process
- Synchronize and communicate with other processes
- For speedup: having several processes active simultaneously

Mapping

- **Mapping: the mechanism by which tasks are assigned to processes for execution**
- **The task-dependency and task-interaction graphs play an important role**
- **Good mapping:**
  - Maximize the use of concurrency (mapping independent tasks onto different processes)
  - Minimize the total completion time (Executing tasks on critical path as they're executable)
  - Minimize interaction (mapping tasks with a high degree of mutual interaction onto the same process)
- **Conflicting goals => finding a balance (the key)**
- **Decomposition vs. Mapping**

  Detects concurrency — Determines how much and how efficiently to utilize the concurrency

An Example of Mapping



It makes more sense to map the tasks connected by an edge onto the same process to prevent an inter-task interaction from becoming an inter-processes interaction

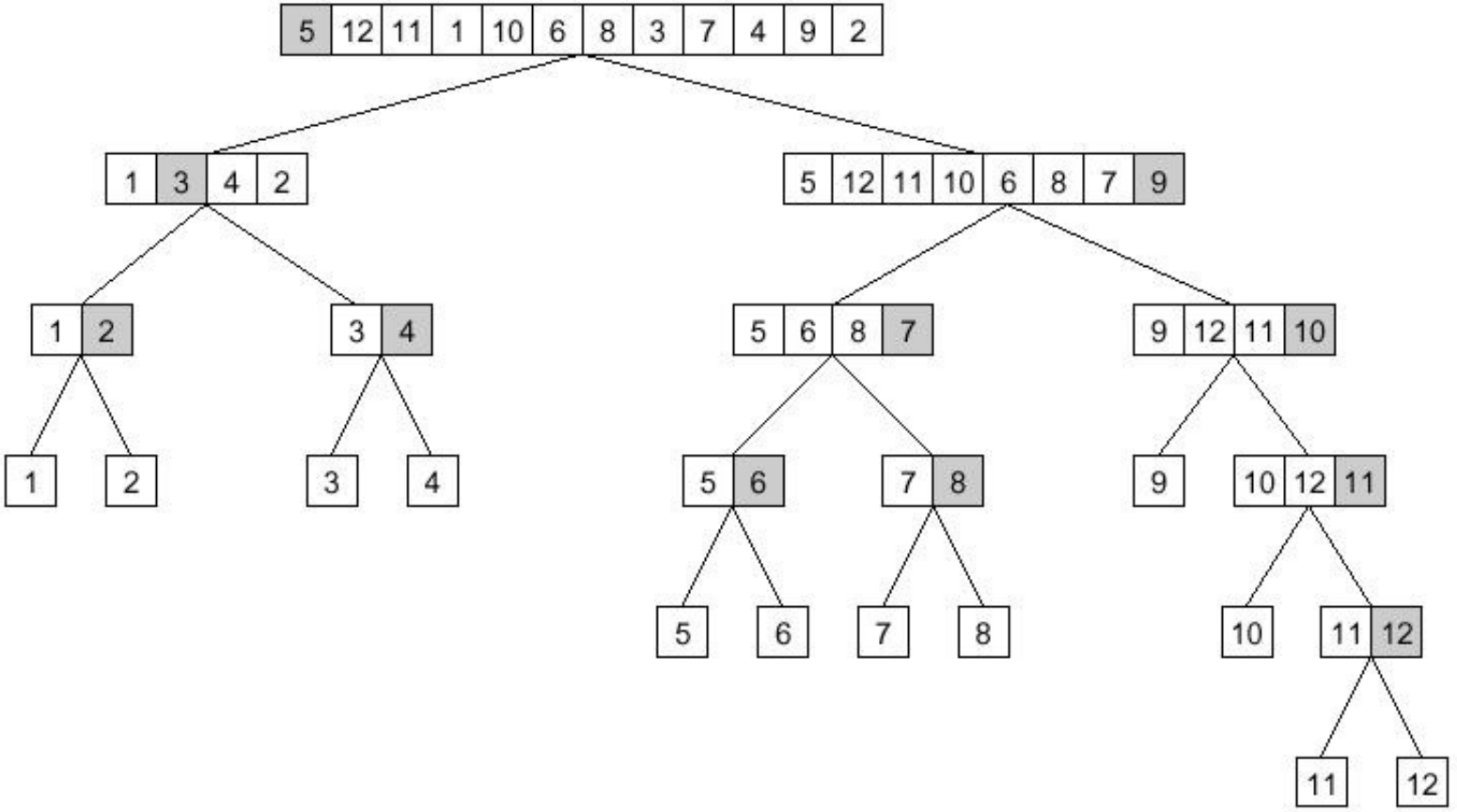
Decomposition Techniques

- **Fundamental steps: split the computations to be performed into a set of tasks for concurrent execution (decomposition)**
- **Classification:**
  - Recursive decomposition (general purpose)
  - Data decomposition (general purpose)
  - Exploratory decomposition (special purpose)
  - Speculative decomposition (special purpose)

Recursive Decomposition

- **A method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy:**
  - Divide a problem into a set of independent subproblems
  - Each subproblem is solved by recursively applying a similar division into smaller subproblems
  - Combine the results of the smaller subproblems
- **Example: Quicksort**
  - A sequence A of n elements
  - Select a pivot element x
  - Partition A into A_0 (smaller) and A_1 (equal to x or greater than x)
  - A_0 and A_1 are sorted by calling Quicksort recursively
  - The recursion terminates when each subsequence contains only a single element

Quicksort Example



A task: the work of partitioning a given subsequence

Data Decomposition

- **A method for deriving concurrency in algorithms that operate on large data structures**
  - Partition the data (input, output, both input and output, intermediate)
  - Partition the computations into tasks based on the data partitioning
- **The operations performed by these tasks on different data partitions:**
  - Usually similar (matrix multiplication)
  - Chosen from a small set of operations (LU factorization)

Partitioning Output Data

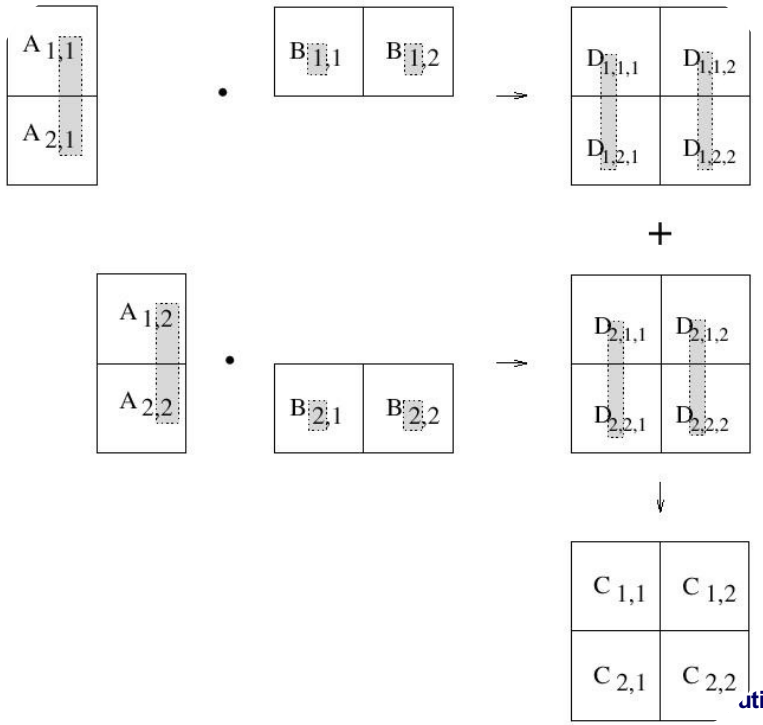
- **Each element of the output can be computed independently of others as a function of the input**
- **Each task is assigned the work of computing a portion of the output**
- **Example: Matrix Multiplication**
  - The matrix is viewed as composed of blocks
  - Scalar arithmetic operations on the elements are replaced by the ones on the blocks
  - Block versions of matrix algorithms are used to aid decomposition

Partitioning Input Data

- **Partitioning of output data: only if each output can be naturally computed as a function of the input**
- **Partitioning of input data**
  - Induce concurrency
  - Each task performs as much computation as possible using local data
  - Solutions to tasks may not directly solve the original problem
    - A follow-up computation is needed to combine the results
    - Such as summation and sorting of a sequence

Partitioning Intermediate Data

- **Algorithms are often structured as multi-stage computations**
  - The output of one stage is the input to the subsequent stage
- **Higher concurrency: partitioning the input or the output data of an intermediate stage of the algorithm**
  - Sometimes: restructuring of the original algorithm
- **Example: Matrix Multiplication**
  - Eight tasks compute their respective product submatrices and store the results in a 3-D matrix D
  - $D_{k,i,j}$ is the product of $A_{i,k}$ and $B_{k,j}$

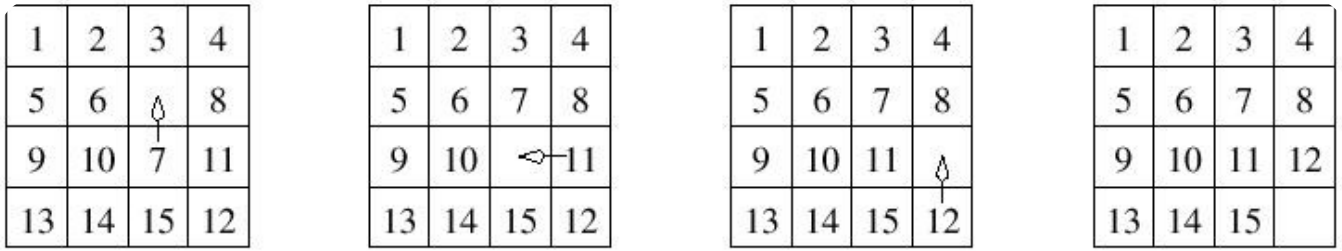


The Owner-Computes Rule

- **The owner-computes rule: a decomposition based on partitioning output or input data**
- **Each partition performs all the computations involving data that it owns**
- **Variant meanings:**
  - Partitioning input data: a task performs all the computations that can be done using these data
  - Partitioning output data: a task computes all the data in the partition assigned to it

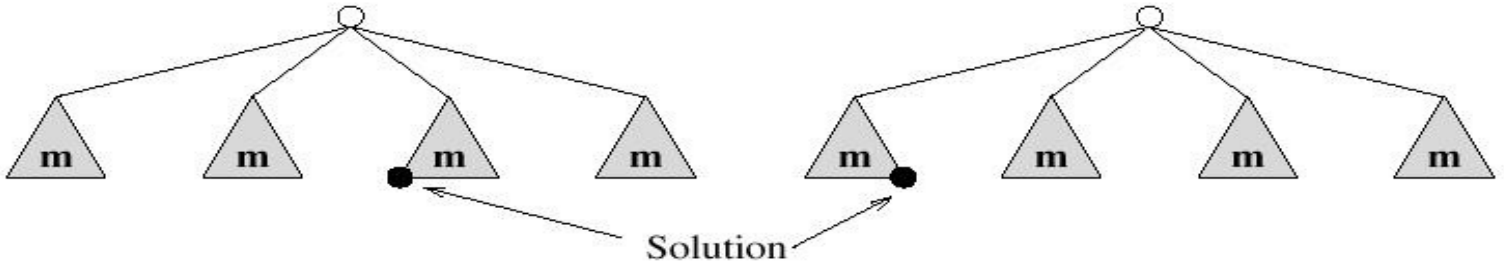
Exploratory Decomposition

- **Decompose problems whose underlying computations correspond to a search of a space for solutions**
  - Partition the search space into smaller parts
  - Search each one of them until solutions are found
- **Example: The 15-puzzle problem**
  - 15 tiles numbered 1 through 15 in a 4 x 4 grid
  - One blank tile
  - Four possible moves: up, down, left, and right
  - The initial and final configurations are specified
  - The objective: determine any sequence or a shortest sequence of moves



Exploratory vs. Data Decomposition

- **Exploratory decomposition appears similar to data decomposition**
  - The search space can be thought of as being the data partitioned
- **Differences:**
  - Data decomposition: each task performs useful computations towards the solution of the problem
  - Exploratory decomposition: unfinished tasks can be terminated as soon as an overall solution is found
    - The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm



Speculative Decomposition

- **Used when a program may take one of many possible computationally significant branches depending on the output of preceding computations**
  - One task performs the computations whose output will be used in deciding the next computation
  - Other tasks can concurrently start the computations of the next stage
- **Similar to evaluating branches in a *switch* statement in C**
  - Evaluate multiple branches in parallel
  - Correct branch will be used and other branches will be discarded
- **The parallel run time is smaller than the serial run time by the amount of time to evaluate the condition**
  - It is used to perform next stage's computation
  - At least some wasteful computation
    - Only the most promising branch is taken up a task in parallel
      - If different, roll back and take the correct one

Speculative vs. Exploratory Decomposition

- **What is unknown**
  - In speculative one: the input at a branch leading to multiple parallel tasks is unknown
  - In exploratory one: the output of the multiple tasks originating at a branch is unknown
- **The amount of work**
  - In speculative one: performs more aggregate work than its serial counterpart
  - In exploratory one: perform more, less, or the same amount of aggregate work depending on the location of the solution in the search space

Designing a Parallel Algorithm

1. **Identify the concurrency available in a problem and decompose it into tasks (executed in parallel)**
2. **Design a parallel algorithm to assign (map) tasks onto the available processes**
   - The nature of the tasks
   - The interactions among tasks

Characteristics of Tasks

- **Task generation**
  - Static: all the tasks are known before the algorithm starts execution
    - Data decomposition: matrix-multiplication, LU factorization
    - Recursive decomposition: finding the minimum of a list of numbers
  - Dynamic: the actual tasks and the task-dependency graph are not explicitly available a priori, although the high level rules or guidelines are known
    - Recursive decomposition: quicksort
      - Tasks are generated dynamically
      - The size and shape of the task tree are determined by the input array
  - Either static or dynamic:
    - Exploratory decomposition: 15-puzzle problem
      - A preprocessing task expands the search tree in a breadth-first manner to generate predefined number of configurations
      - These configurations are mapped and run on processes in parallel, and they can generate dynamic tasks later

Characteristics of Tasks (cont'd)

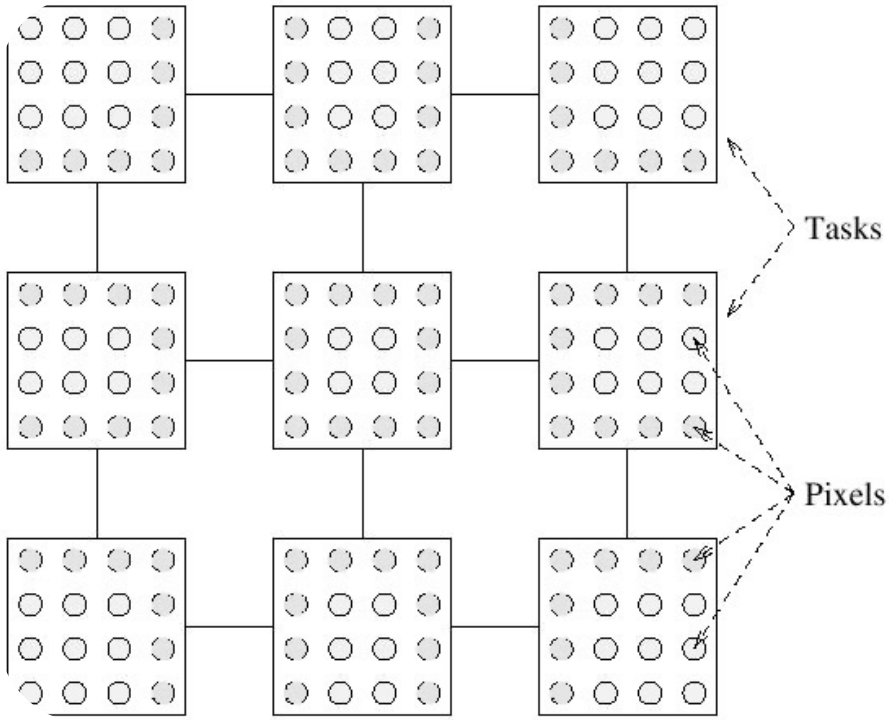
- **Task sizes: the relative amount of time required to complete the task**
  - Uniform: the tasks require roughly the same amount of time
    - Matrix multiplication
  - Non-uniform: the amount of time required by the tasks varies significantly
    - Quicksort
- **Knowledge of task sizes: influences the choice of mapping scheme**
  - Known: matrix multiplication
  - Unknown: 15-puzzle problem (how many moves to lead to the solution)
- **Size of data associated with tasks: (location) determines if excessive data-movement overhead will be incurred**
  - Small input: 15-puzzle
  - Small output: computing the minimum of a sequence
  - Same order of input/output: Quicksort

Characteristics of Inter-Task Interactions

- **Different parallel algorithms => different tasks**
  => different types of interactions
- **The nature of interactions => programming paradigms and mapping schemes**
- **Static versus Dynamic**
  - Static: the task-interaction graph and the stage of the computation at which each interaction occurs are known
    - Programmed easily in shared-address-space and message-passing paradigms
    - Matrix multiplication
  - Dynamic: the timing of interactions or the set of tasks to interact with cannot be determined prior to the execution
    - Hard to synchronize senders and receivers in message-passing
      - Additional synchronization or polling responsibility
    - 15-puzzle problem
      - The finished task can pick up an unexplored state from the queue of another busy task and start exploring it

Characteristics of Inter-Task Interactions (cont'd)

- **Regular versus Irregular (spatial structure)**
  - Regular: an interaction pattern has some structure that can be exploited for efficient implementation
    - Image dithering (each pixel weight: values of original one and neighbors)
  - Irregular: no such regular pattern exists
    - Harder to handle, particularly in message-passing paradigm
    - Sparse matrix-vector multiplication (the access pattern for the vector depends on the structure of the sparse matrix)



Characteristics of Inter-Task Interactions (cont'd)

- **Read-only versus Read-Write**
  - Sharing of data among tasks => inter-task interaction
  - Type of sharing => the choice of the mapping
  - Read-only: tasks require only a read-access to the data shared among many concurrent tasks
    - Matrix multiplication
  - Read-Write: read and write on some shared data
    - 15-puzzle problem (an exhaustive search)
    - Heuristic search: use a heuristic to provide a relative approximate indication of each state from the solution (potential number of moves)
      - The number of tiles that are out of place
      - Priority queue: shared data and tasks (read/write)
        - Put the states resulting from an expansion into the queue
        - Pick up the next most promising state for the next expansion

Characteristics of Inter-Task Interactions (cont'd)

- **One-way versus Two-way**
  - Two-way: the data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks
    - Predefined producers and consumers
    - Read-write
  - One-way: only one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one
    - Read-only, read-write
  - Shared-address-space: supports both one-way and two-way interactions equally easily
  - Message-passing: does NOT support one-way interactions
    - The source must explicitly send the data to the recipient
    - Converting one-way to two-way interactions via program restructuring
      - Static: known *a priori* => introducing matching interaction operations
      - Dynamic: restructuring (polling, checking for pending requests after regular intervals)

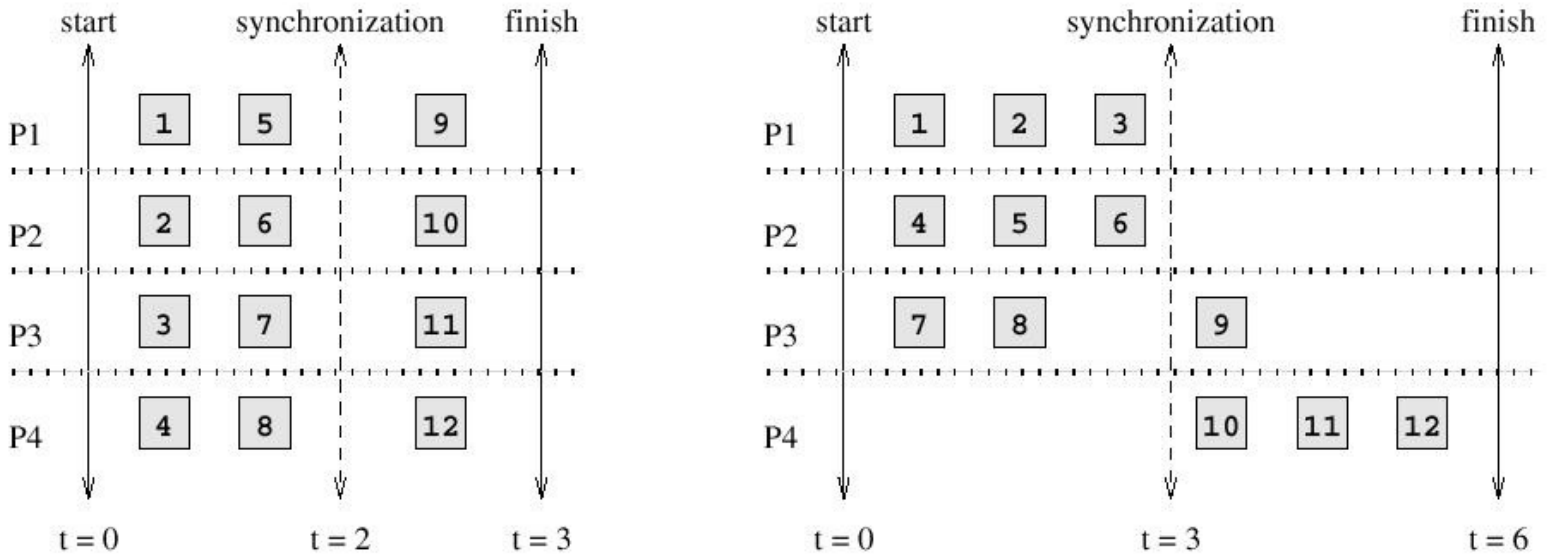
Mapping Techniques

Mapping Techniques for Load Balancing

- **To achieve a small execution time => minimize overheads**
- **Overheads:**
  - Interaction: inter-process interaction
  - Idling: some processes may spend being idle
    - To satisfy the constraints imposed by the task-dependency graph
- **Overheads => functions of mapping**
- **Good mapping:**
  - Reducing interaction time
  - Reducing idle time
- **Conflicting objectives**
  - Mapping tasks onto the same process => unbalanced workload (against concurrency)
  - Balance the load among processes => may cause heavy interactions

Mapping Techniques for Load Balancing (cont'd)

- **Assigning a balanced aggregate load of tasks to each process is necessary but not sufficient condition for reducing process idling**
- **Poor synchronization can lead to idling**
  - One task waits to send or receive data from others
- **A good mapping: balance both computations and interactions at each stage**



Static Mapping

- **Mapping: determined by programming paradigm and the characteristics of tasks and interactions**
- **Statically generated tasks: either static or dynamic**
- **Static mapping: distribute the tasks among processes prior to the execution of the algorithm**
- **A good mapping:**
  - The knowledge of task sizes
  - The size of data associated with tasks
  - The characteristics of inter-task interactions
  - Parallel programming paradigm
- **Optimal mapping for non-uniform tasks: NP-complete**
  - Heuristics

Dynamic Mapping

- **Distribute the work among processes during the execution**
- **If tasks are generated dynamically => mapped dynamically**
- **Unknown task sizes => dynamic mappings are more effective**
- **Large data associated with the computation**
  - Data-movement cost may outweigh other advantages => static
  - May work well in shared-address-space paradigm (read-only)
    - Physical data movement on NUMA and cc-UMA

Schemes for Static Mapping

- **Static mapping is often used in conjunction with**
  - Data partitioning
  - Task partitioning

- **Static mapping is used for mapping certain problems that are expressed naturally by a static task-dependency graph**

Schemes for Dynamic Mapping

- **Dynamic mapping: when a static mapping generates imbalanced work distribution or the task-dependency graph is dynamic**
- **Referred as dynamic load-balancing**
- **Classification:**
  - Centralized
  - Distributed

Centralized Schemes

- **All executable tasks are maintained**
  - In a common central data structure
  - By a special process or a subset of processes
    - Master: the special process
    - Slaves: other processes that depend on the master to obtain work
- **No work => the central data structure or the master process**
- **Easier to implement**
- **Limited scalability**
  - The common data structure and the master process become the bottleneck

Distributed Schemes

- **The set of executable tasks are distributed among processes which exchange tasks at run time to balance work**
  - Each process can send work to or receive work from any other process
  - Do not suffer from the bottleneck
- **Critical parameters:**
  - How are the sending and receiving processes paired together?
  - Is the work transfer initiated by the sender or the receiver?
  - How much work is transferred in each exchange?
    - Too little: frequent transfers (receiver)
    - Too much: frequent transfers (sender)
  - When is the work transfer performed?
    - In receiver initiated load balancing:
      - Out of work
      - Too little work left and anticipated being out of work soon

Minimize Frequency of Interactions

- **There is a relatively high startup cost associated with each interaction on many architectures**
- **Restructure the algorithm such that shared data are accessed and used in large pieces**
  - Amortize the startup cost over large accesses (not the volume)
    - Increase the spatial locality of data access
      - Ensure the proximity of consecutively accessed data locations
      - On a shared-address-space architecture:
        a word => an entire cache line => fewer cache lines
      - On a message-passing system:
        fewer messages => larger messages
    - Example: sparse matrix-vector multiplication
      - Collect all the nonlocal entries of the input vector that it requires
      - Then perform an interaction-free multiplication
        (not trying to access a nonlocal element of the input vector when required)

Minimizing Contention and Hot Spots

- **Contention occurs when multiple tasks try to access the same resources concurrently (interaction pattern)**
  - Multiple simultaneous transmissions of data over the same link
  - Multiple simultaneous accesses to the same memory block
  - Multiple processes sending messages to the same process
- **Only one of the multiple operations can proceed at a time (critical section & mutual exclusion)**
  - Others are queued and proceed sequentially
- **Example: matrix multiplication (2-D distribution)**

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

  - All tasks that work on the same row (column) of C will accessing the same block of A (B)
  - The need to concurrently access these blocks of matrices A and B will create contention on both NUMA shared-address-space and message-passing parallel architectures

Minimizing Contention and Hot Sports (cont'd)

- **Eliminate contention in matrix multiplication**
  - Modify the order in which the block multiplications are performed

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

  - All the tasks $P_{*\,i}$ that work on the same row of C will be accessing block $A_{*,(*+j+k)\%\sqrt{p}}$, which is different for each task

  - All the tasks $P_{i,*}$ that work on the same column of C will be accessing block $B_{(i+*+k)\%\sqrt{p},*}$, which is also different for each task

- **Centralized schemes for dynamic mapping are a frequent source of contention => distributed mapping schemes**

Overlapping Computations with Interactions

- **When wait for shared data => do some useful computations**
- **Techniques:**
  - Initiate an interaction early enough to complete before it is needed
    - Identify irrelevant computations
    - Restructure programs to initiate the interaction at an earlier point
    - Possible if
      - The interaction pattern is spatially and temporally static
      - Multiple tasks are ready for execution
        - Reducing the granularity of tasks => increase overheads
  - In dynamic mapping schemes, the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance

Overlapping Computations with Interactions (cont'd)

- **Overlapping computations with interaction requires support from the programming paradigm, the operating system, and the hardware**
  - Disjoint address-space paradigm:
    - Non-blocking message passing primitives
    - Functions for sending and receiving messages return control to the program before they have actually completed
    - Hardware permits computation to proceed concurrently with message transfers
  - Share-address-space paradigm:
    - Prefetching hardware: anticipate the memory addresses and initiate the access in advance of when they are needed
    - Compilers can detect the access pattern and place pseudo-references to certain key memory locations

Replicating Data or Computations

- **Multiple processes may require frequent read-only access to shared data structure, such as a hash-table**
- **After replicating a copy of the shared data on each process, all subsequent accesses are free of interaction overhead**
- **For different paradigms:**
  - Shared-address-space: cache
  - Message-passing:
    - Remote data accesses are more expensive or harder than local accesses
      - Replication reduces interaction overhead and significantly simplifies the writing of the parallel program

Replicating Data or Computations (cont'd)

- **Cost: data replication increases the memory requirements**
  - Linearly with the number of concurrent processes
  - Limit the size of the problem that can be solved

    => only replicate small mount of data

- **To share intermediate results**
  - In some situations, it may be more cost-effective to compute these intermediate results than to get them from another process
    - Interaction overhead can be traded for replicated computation

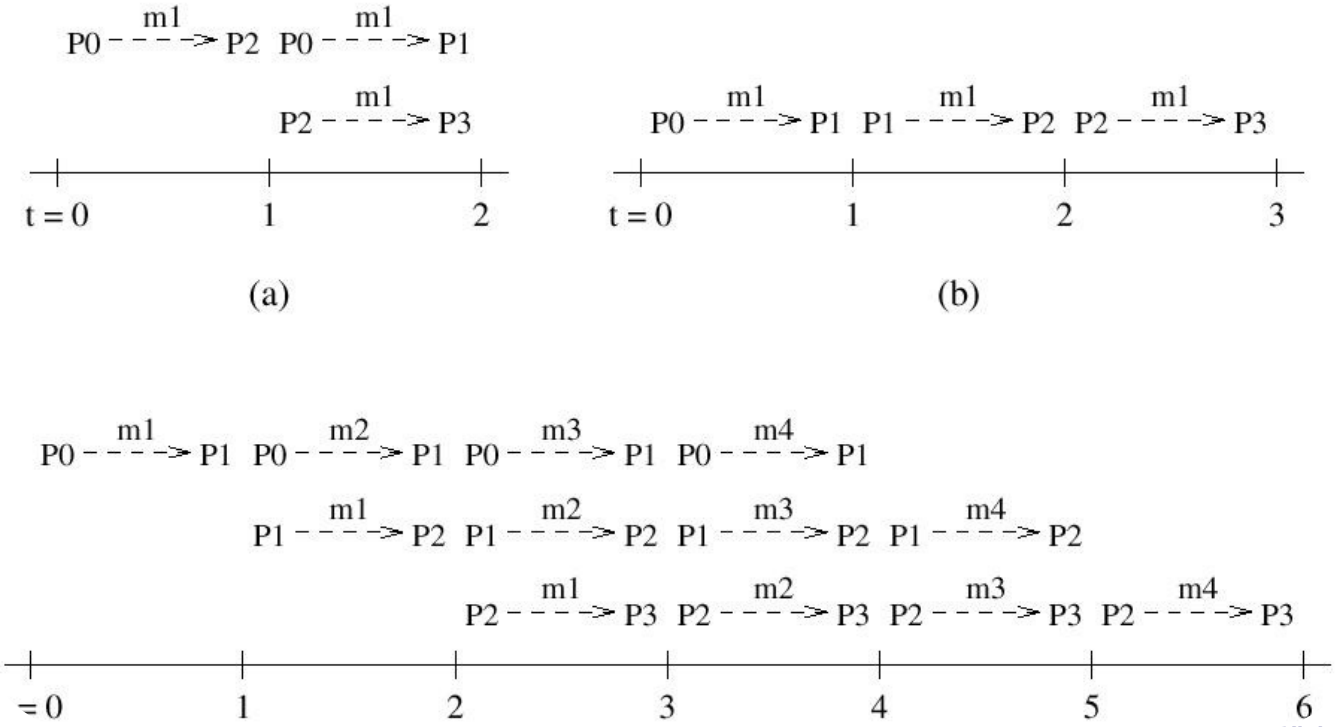
Using Optimized Collective Interaction Operations

- **Collective operations: such as**
  - Broadcasting some data to all processes
  - Adding up numbers, each belonging to a different process
- **Categories:**
  - Used by the tasks to access data
  - Used to perform some communication-intensive computations
  - Used for synchronization

- **Collective operations are highly optimized to minimize the overheads due to data transfer as well as contention**
  - Available in library form from the vendors (MPI)

Overlapping Interactions with Other Interactions

- **Overlapping interactions between multiple pairs of processes can reduce the effective volume if communication**
- **Pipeline fashion (using naive broadcast algorithm) can increase the amount of overlap**
  - Unlikely to be included in a collective communication library (Expensive for a single broadcast operation)



Parallel Algorithm Models

Parallel Algorithm Models

- **The way of structuring a parallel algorithm by**
  - Selecting a decomposition
  - Selecting a mapping technique
  - Applying the appropriate strategy to minimize interactions

The Data-Parallel Model

- **The tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data**
  - Data parallelism: a result of identical operations on different data items
  - Similar computations => uniform partitioning => load balance
- **Programming paradigms:**
  - Message-passing: a better handle on locality
  - Shared-address-space: ease the programming effort
- **Minimize interaction overheads:**
  - Choose a locality preserving decomposition
  - Overlap computation and interaction
  - Use optimized collective interaction routines
- **The degree of data parallelism increases with the size of the problem => more processes to solve larger problem**
- **Example: matrix multiplication**

The Task Graph Model

- **The interrelationships among the tasks are utilized to promote locality or to reduce interaction costs**
  - Task-dependency graph is explicitly used in mapping for task parallelism
- **Solved problems: tasks with large data**
  - Tasks are mapped statically to help optimize the cost of data movement among tasks
- **Interaction-reducing techniques:**
  - Reduce the volume and frequency of interaction by promoting locality
  - Asynchronous interaction methods for overlapping
- **Example: quicksort, sparse matrix factorization, many algorithms using divide-and-conquer decomposition**

The Work Pool Model

- **Characterized by a dynamic mapping of task onto processes for load balancing**
  - No desired premapping
  - Centralized or decentralized
  - Work (task) pool: shared list, priority queue, hash table, or tree
  - If the work is generated dynamically and a decentralized mapping is used => termination detection
- **In message-passing paradigm: this model is used when the amount of data is relative small (compared to computation)**
  - Tasks can move around without much interaction overhead
  - The granularity of tasks: tradeoff between load-imbalance and the overhead for adding and extracting tasks
- **Example: parallelization of loops by chunk scheduling**

The Master-Slave Model

- **One or more master processes generate work and allocate it to worker processes**
  - Static: if the manager can estimate the size of the tasks
  - Dynamic: for load balancing
    - When time-consuming for the master to generate work
- **Synchronization: each phase must finish before work in the next phases can be generated**
- **Hierarchical or multi-level manager-worker model**
  - Workers can further subdivide the tasks
- **The granularity of tasks**
  - Bottleneck: too small
  - Criteria: the cost of doing work dominates the cost of transferring work and cost of synchronization
- **Asynchronous interaction: overlapping**

The Pipeline or Producer-Consumer Model

- **A stream of data is passed on through a succession of processes, each of which performs some tasks**
- **Stream parallelism: simultaneous execution of different programs on a data stream**
- **A pipeline is a chain of producers and consumers**
  - A linear chain
  - A directed graph
- **Task granularity:**
  - Too large: longer time to fill up the pipeline
  - Too fine: more interaction overheads
- **Interaction reduction technique: overlapping**
- **Example: LU factorization**

Hybrid Models

- **More than one model may be applied**
  - Multiple models are applied hierarchically
  - Multiple models are applied sequentially to different phases of a parallel algorithm
- **Example: quicksort**

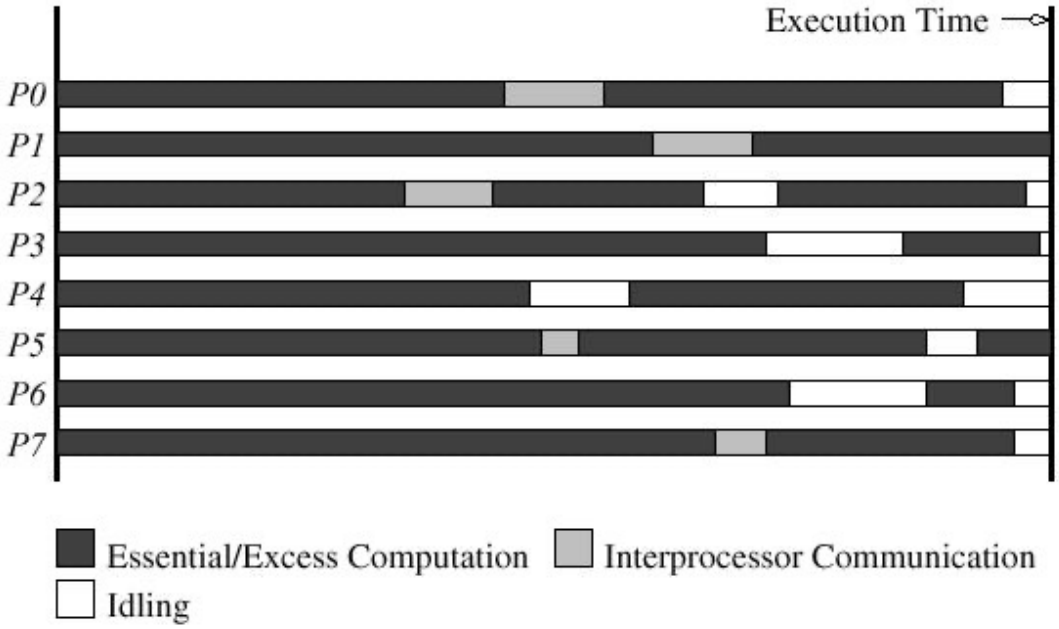
Analytical Modeling of Parallel Programs

Performance Evaluation

- **Evaluation in terms of execution time**
  - A sequential algorithm: a function of the size of its input
  - A parallel algorithm:
    - Input size
    - The number of processing elements (PEs) used
    - The relative computation and interprocess communication speeds
- **A parallel system: the combination of**
  - An algorithm
  - The parallel architecture on which it is implemented
- **Intuitive performance measures:**
  - Wall-clock time: taken to solve a given problem on a given parallel platform
    - Cannot be extrapolated to other problem instances or larger machine configurations
  - Quantify the benefit of parallelism: how much faster the parallel program runs with respect to the serial program
    - A poorer serial algorithm might be more amenable to parallel processing?

Sources of Overhead in Parallel Programs

- **A typical execution includes:**
  - Essential computation
    - Computation that would be performed by the serial program for solving the same problem instance
  - Interprocess communication
  - Idling
  - Excess computation
    - Computation not performed by the serial formulation



Sources of Overhead in Parallel Programs (cont'd)

- **Interprocess interaction: processing elements (PEs) interact and communicate data (e.g., intermediate results)**
  - Usually the most significant source of parallel processing overhead
- **Idling: processing elements become idle due to**
  - Load imbalance
    - Dynamic task generation: hard/impossible to predict the size of the subtasks
      - The problem cannot be subdivided statically to maintain uniform workload
        - Some PEs are idle while others are working on the problem
  - Synchronization
    - PEs might synchronize at certain points
      - PEs which are ready sooner will be idle until all the rest are ready
  - Presence of serial components in a program
    - Only one PE is allowed to work on it
    - All the other PEs must wait

Sources of Overhead in Parallel Programs (cont'd)

- **Excess computation: the difference in computation performed by the parallel program and the best serial program**
  - The fastest known sequential algorithm may be difficult or impossible to parallelize
  - A parallel algorithm is developed based on a poorer but easily parallelizable sequential algorithm
  - A parallel algorithm based on the best serial algorithm may still perform more aggregate computation than the serial algorithm
    - Example: FFT (Fast Fourier Transform)
      - In the serial version, the results of certain computations can be used
      - In the parallel version, they are not reusable (generated by different PEs)
        - Be performed multiple times on different PEs

Performance Metrics for Parallel Systems

- **Performance metrics are useful for:**
  - Determining the best algorithm
  - Evaluating hardware platforms
  - Examining the benefits from parallelism
- **Execution time**
  - The serial runtime of a program (T_S): the time elapsed between the beginning and the end of its execution on a sequential computer
  - The parallel runtime (T_P): the time that elapses from the moment a parallel computation starts to the moment the last PE finishes execution

Performance Metrics for Parallel Systems (cont'd)

- **Overhead function: the overheads incurred by a parallel program are encapsulated into a single expression**
- **Total overhead (T_o): the total time collectively spent by all the PEs over and above that required by the fastest known sequential algorithm for solving the same problem on a single PE**
  - The total time summed over all PE is pT_P
  - Overhead $T_o = pT_P - T_S$
- **Speedup (S): the ratio of the time taken to solve a problem on a single PE to the time required to solve the same problem on a parallel computer with p identical PEs**
  - Capture the relative benefit of solving a problem in parallel
  - The p PEs are identical the one used by the sequential algorithm

Computing Speedups of Parallel Programs

- **For a given problem, more than one sequential algorithm may be available**
  - Natural to use the one that solves the problem in the least amount of time
  - The asymptotically fastest sequential algorithm is unknown or its runtime has a large constant that makes it impractical to implement
    - Take the fastest known algorithm
- **Example: parallelizing bubble sort (10^5 records)**
  - The serial bubble sort: 150 seconds
  - The serial quick sort: 30 seconds
  - A parallel version of bubble sort (odd-even sort): 40 seconds
  - Speedup
    - Using serial bubble sort: 150/40 = 3.75
    - Using serial quick sort: 30/40 = 0.75

Computing Speedups of Parallel Programs (cont'd)

- **Theoretically, speedup can never exceed the number of PE, p**
- **To achieve speedup p:**
  - None of the PEs spends more than T_S/p
- **A speedup greater than p:**
  - Only if each PE spends less than time T_S/p
    - A single PE could emulate the p PEs and solve the problem in fewer than T_S units of time
      - Contradiction: speedup is computed with respect to the best sequential algorithm
  - Superlinear speedup
    - The work performed by a serial algorithm is greater than its parallel formulation
    - Hardware features that put the serial implementation at a disadvantage
    - Example: the data might be too large for the cache of a single PE
      - Degrading performance due to the use of slower memory elements
      - Partitioned data can be small enough to fit into respective PE's caches

Amdahl's Law

- **The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used**

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

Furthermore,

$$\begin{cases} S_{\text{latency}}(s) \leq \dfrac{1}{1-p} \\ \lim\limits_{s \to \infty} S_{\text{latency}}(s) = \dfrac{1}{1-p}. \end{cases}$$

- **The theoretical speedup is always limited by the part of the task that cannot benefit from the improvement**
  - Excess computation and communication are captured in the serial component

Efficiency

- **Only an ideal parallel system containing p PEs can deliver a speedup equal to p**
  - In practice, not achievable
    - PEs cannot devote 100% of their time to the computations of the algorithm
- **Efficiency: a measure of the fraction of time for which a PE is usefully employed**
  - The ratio of speedup to the number of PEs E = S / p
  - In practice, speedup is less than p and efficiency is between 0 – 1
- **Example: adding n numbers on n PEs**

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n}$$
$$= \Theta\left(\frac{1}{\log n}\right)$$

Cost

- **Cost (work or processor-time product) : the product of parallel runtime and the number of processing elements used**
  - Reflect the sum of the time that each PE spends solving the problem
- **Efficiency: the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p PEs**
  - The cost of solving a problem on a single PE ➔ time of the fastest known sequential algorithm
- **Cost-optimal: the cost on a parallel computer has the same asymptotic growth as a function of the input size as the fastest-known sequential algorithm on a PE**
  - For such systems, the efficiency should be $\theta(1)$
    - Known as pT_P-optimal systems
  - Example: adding n numbers on n PEs
    - Total cost (processor-time product): $\theta(n \log n)$
    - Serial time: $\theta(n)$
    - Not cost-optimal

Variation of Efficiency

- **Two observations**
  - For a given problem size, as we increase p, the overall efficiency goes down
    - Common to all parallel systems
  - Keeping p constant, the efficiency increases if the problem size is increased
- **To keep the efficiency fixed: the problem size increases at a rate with respect to p**
  - A lower rate is more desirable (in problem size)