COMP5426
Parallel and Distributed Computing

Parallel Algorithm Design

Parallel Computing Platforms

- Writing programs for parallel machines is hard ...
- But...one-off efforts if written in a standard language
- Past lack of parallel programming standards ...
  - ... has restricted uptake of technology (to "enthusiasts")
  - ... reduced portability (over a range of current architectures and between future generations)

Parallel Computing Platforms

- Physical organization – actual hardware organization
- Logical organization – programmer's view
  - Control structure: ways of expressing parallel tasks
  - Communication model: mechanisms for specifying interaction between these tasks

Parallel Computing Platforms: Control Structure

- Parallelism can be expressed at various levels of granularity - from instruction level to processes
- Between these extremes exist a range of models, along with corresponding architectural support
- Processing units in parallel computers either operate under
  - SIMD (single instruction stream, multiple data stream) - the centralized control of a single control unit
  - MIMD (multiple instruction stream, multiple data stream) - work independently
  - SPMD (single program multiple data) - A simple variant of MIMD

Parallel Computing Platforms: Communication Model

- There are two primary forms of data exchange between parallel tasks
  - accessing a shared data space
  - exchanging messages
- Platforms that provide a shared data space are called shared-address-space machines or multiprocessors
- Platforms that support messaging are also called message passing platforms or multicomputers

Parallel Computing Platforms: Shared-Address-Space Machine

- A global memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space
- uniform memory access (UMA) platform - If the time taken by a processor to access any memory word in the system global or local is identical
- non-uniform memory access (NUMA) machine - otherwise

Parallel Computing Platforms: Shared-Address-Space Machine

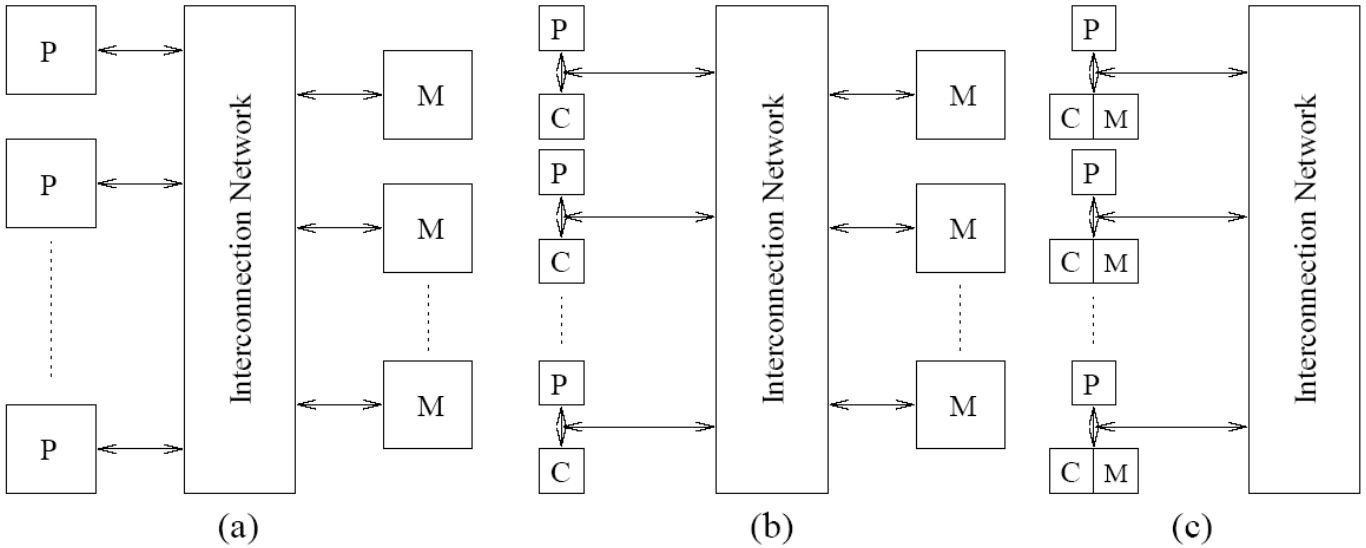


Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

Parallel Computing Platforms: Shared-Address-Space Machine

- 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
- read-write data to shared data must be coordinated
- Shared-address-space programming paradigms, e.g., multithreads (Pthreads) or directives (OpenMP) support synchronization using locks and related mechanisms

Parallel Computing Platforms: Message-Passing Platform

- These platforms comprise of a set of processors and their own (exclusive) memory
- They are programmed using (variants of) send and receive primitives
- Libraries such as MPI and PVM provide such primitives

The Sieve of Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61

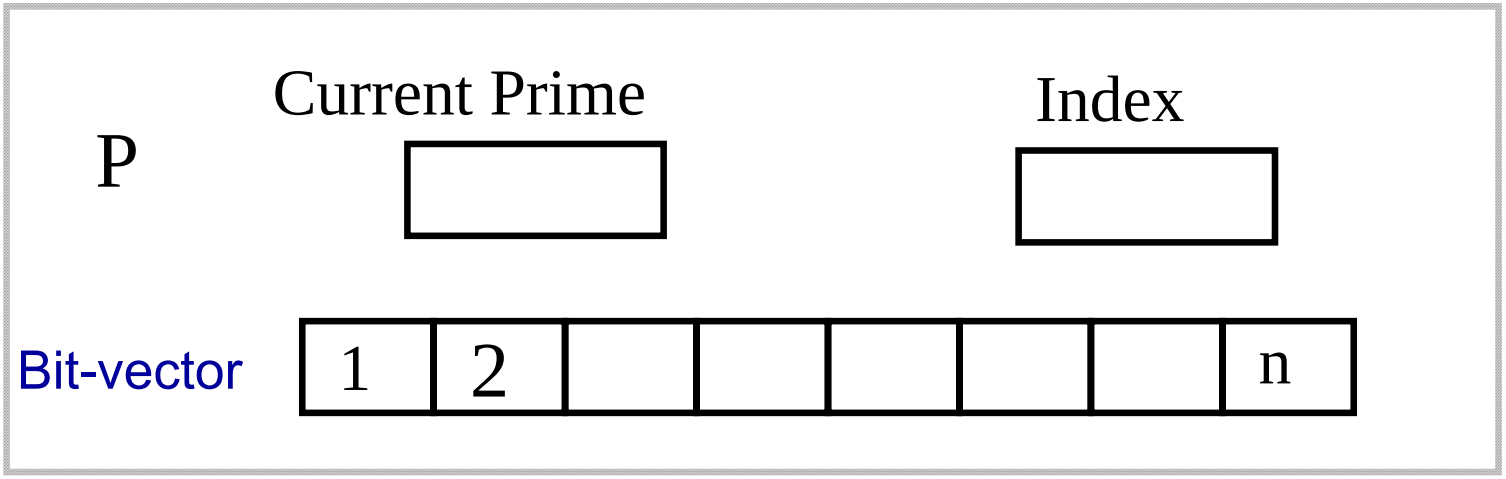
- Any composite number has a prime factor that is no greater than its square root, i.e., the largest prime used to sieve is $\sqrt{n}$
- Complexity: $\Theta(n \ln \ln n)$

Pseudocode

1. Create a list of unmarked natural numbers 2, 3, …, n
2. $k \leftarrow 2$
3. Repeat
   (a) Mark all multiples of k between k^2 and n
   (b) $k \leftarrow$ smallest unmarked number $> k$
   until $k^2 > n$
4. The unmarked numbers are primes

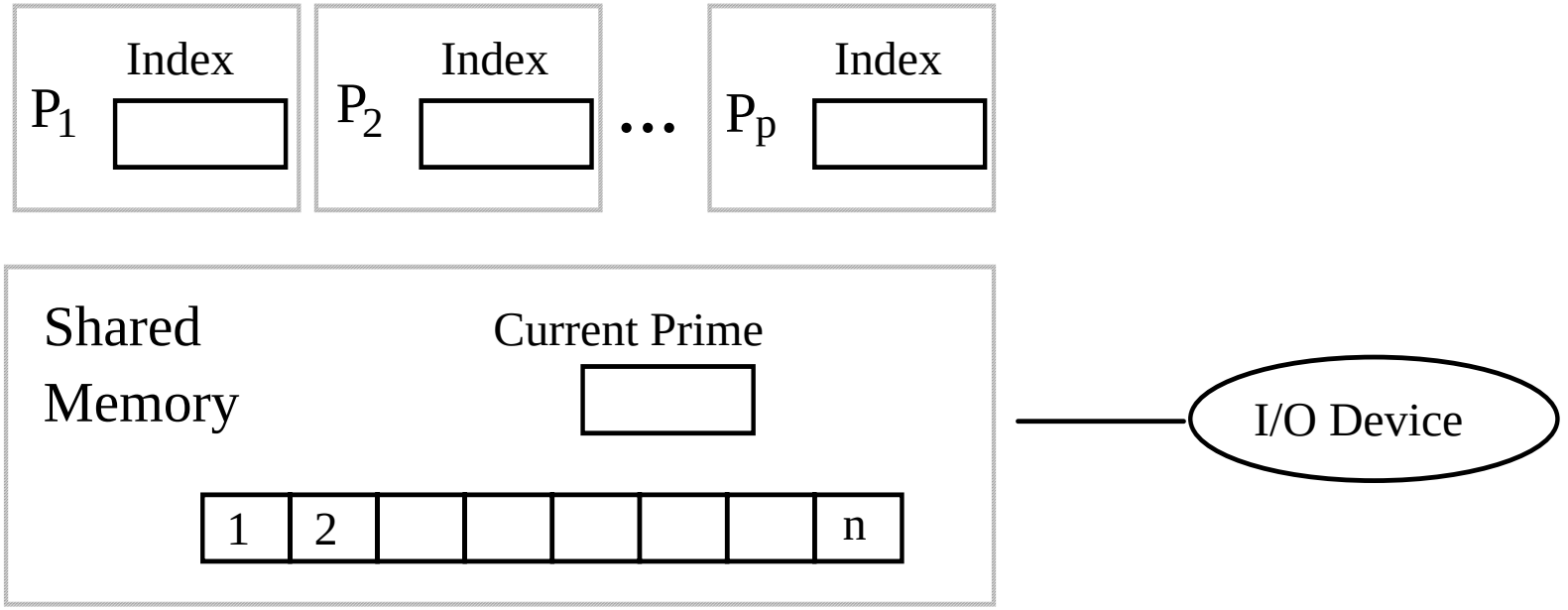
Single-Processor Implementation of the Sieve

- Schematic representation of single-processor solution for the sieve of Eratosthenes:



A Parallel Implementation of the Sieve (I)

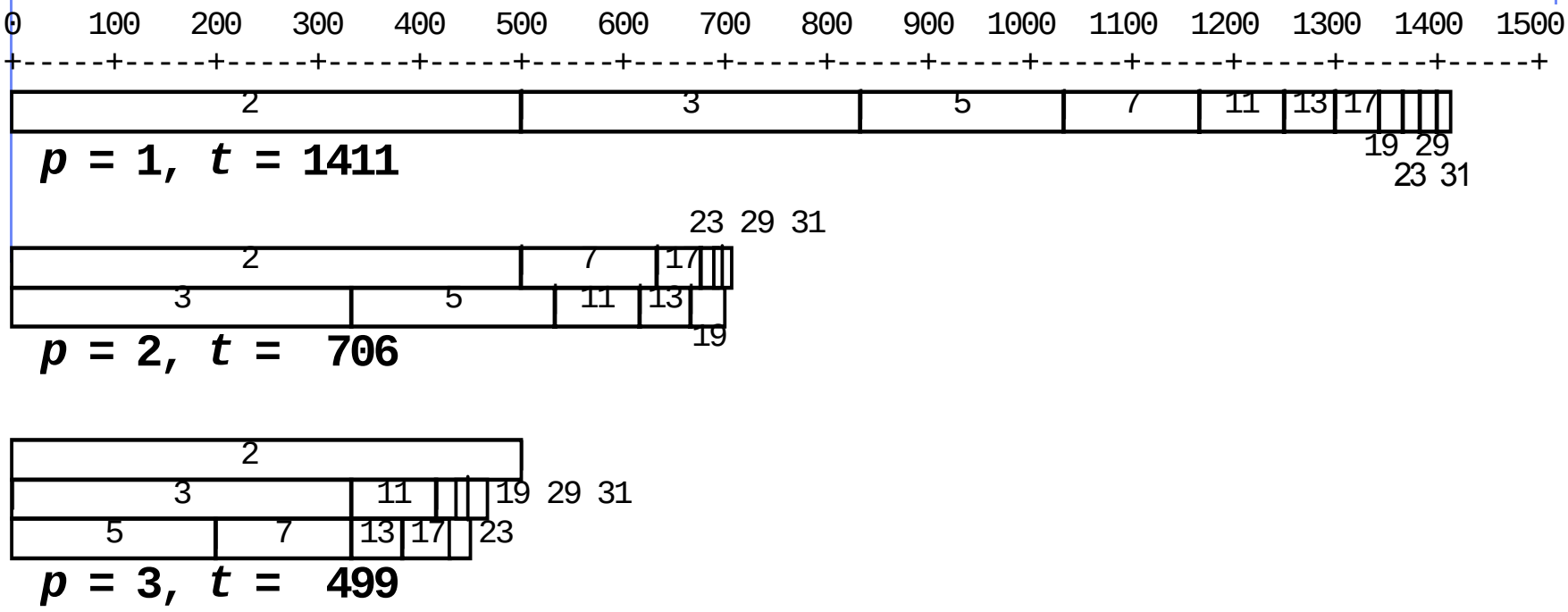
- Schematic representation of parallel implementation on shared-memory machine:



- Synchronization mechanisms – to deal with the shared data

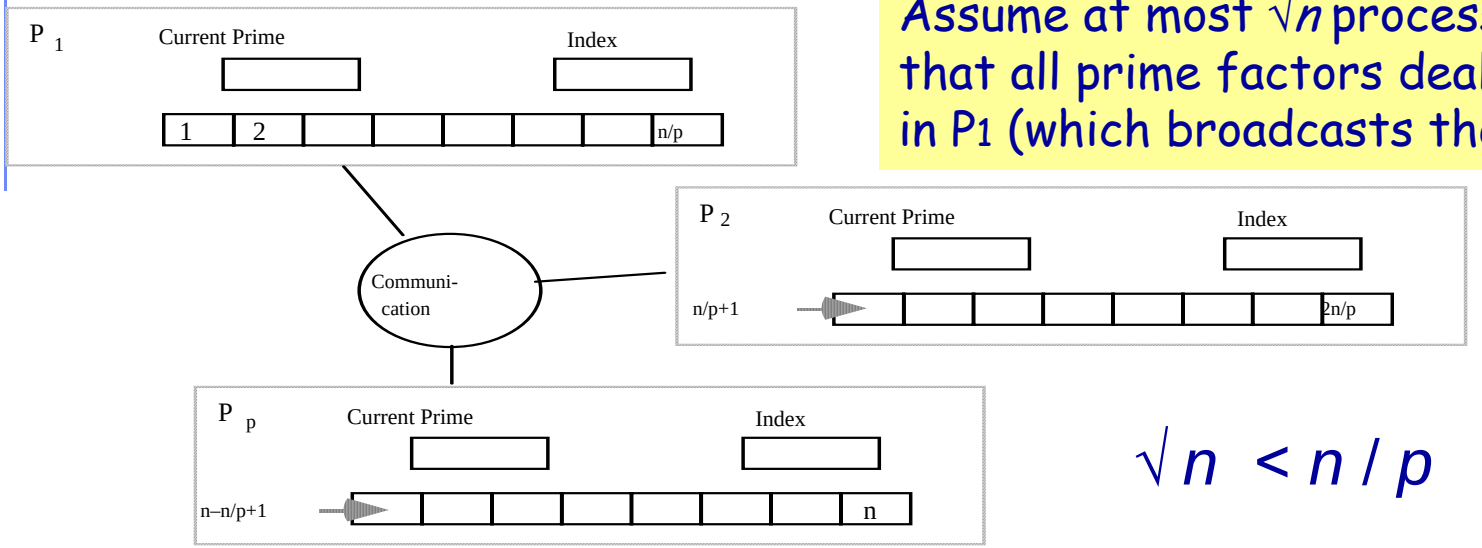
Running Time of the Sequential/Parallel Sieve

- The sieve of Eratosthenes with n = 1000 and $1 \leq p \leq 3$:



A Parallel Implementation of the Sieve (II)

- Schematic representation of parallel implementation on a distributed-memory machine:



Assume at most $\sqrt{n}$ processors, so that all prime factors dealt with are in P_1 (which broadcasts them)

$$\sqrt{n} < n / p$$

- communication mechanisms – deal with the remote data

Designing Parallel Algorithms

- How a problem specification is translated into an algorithm that displays concurrency, scalability, and locality
- Parallel algorithm design is not easily reduced to simple recipes
- Rather, it requires the sort of integrative thought that is commonly referred to as ``creativity''
- However, it *can* benefit from a methodical approach that maximizes the range of options considered
  - provides mechanisms for evaluating alternatives
  - reduces the cost of backtracking from bad choices

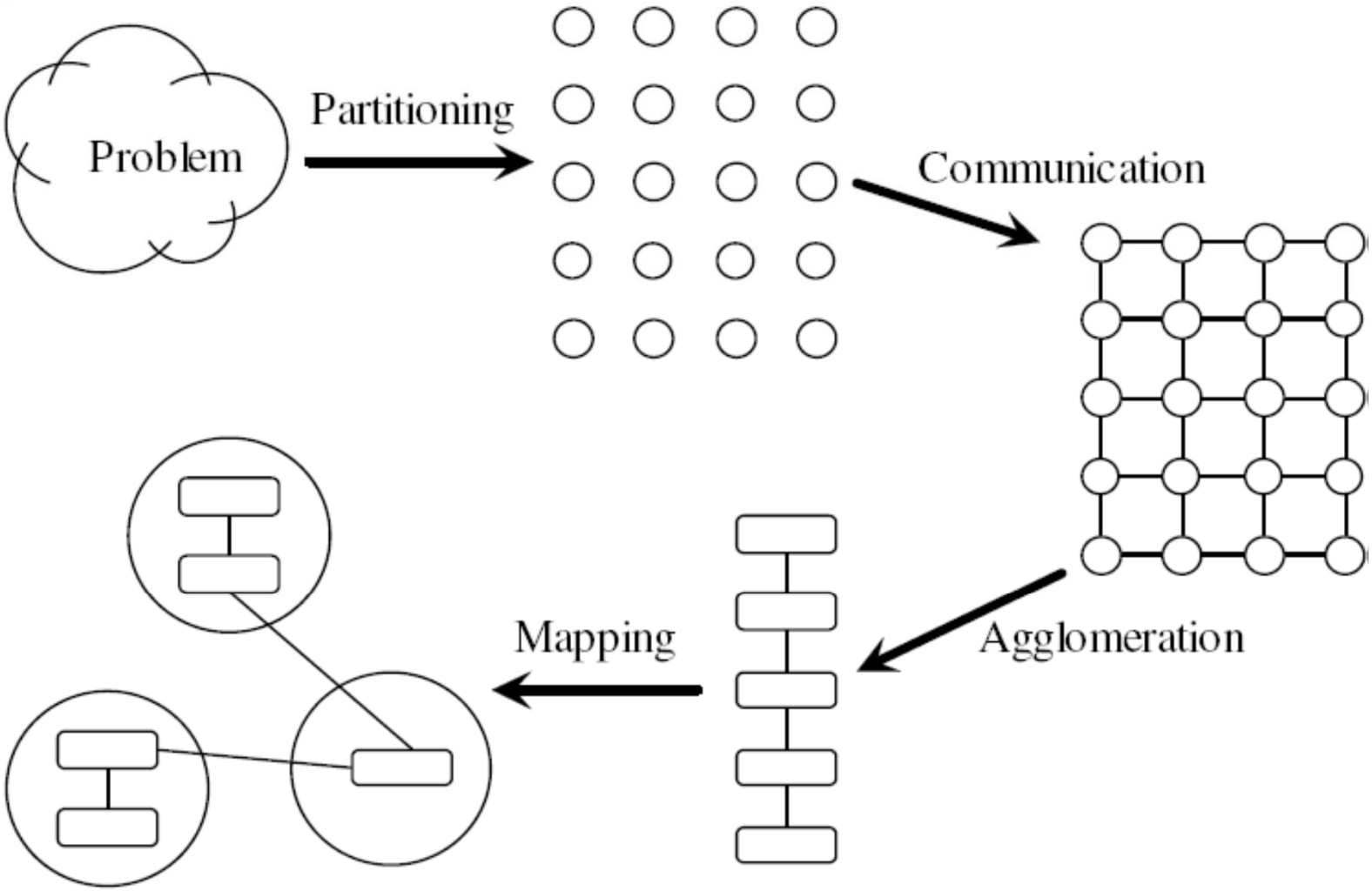
Methodical Design

- Machine-independent issues:
  - concurrency are considered early
  - machine-specific aspects of design are delayed until late in the design process
- Four-stage design process:
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping

Methodical Design

- *Partitioning*. The computation that is to be performed and the data operated on are decomposed into small tasks
- *Communication*. The communication required to coordinate task execution, and appropriate communication structures and algorithms are defined
- In partitioning and communication stages
  - practical issues are ignored
  - focus on recognizing opportunities for parallel execution
- *Agglomeration*. tasks are combined into larger tasks to improve performance or to reduce development costs
- *Mapping*. Tasks are assigned to processes/processors to satisfy the competing goals of
  - maximizing processor utilization
  - minimizing communication/synchronization costs

Methodical Design



Methodical Design: Partitioning

- Expose opportunities for parallel execution
- The focus is on defining a large number of small tasks in order to yield what is termed a *fine-grained* decomposition of a problem
- A partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates

Methodical Design: Partitioning

- Typical types of partitioning:
  - Data decomposition
  - Functional decomposition
- Data decomposition
  - Divide data into pieces
  - Determine how to associate computations with the data
- Functional decomposition
  - Divide computation into pieces
  - Determine how to associate data with the computations

Methodical Design: Data Decomposition

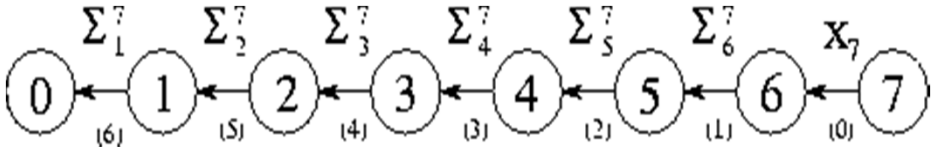
- First decompose the data associated with a problem. If possible, we divide these data into small pieces of approximately equal size
- Next partition the computation that is to be performed, typically by associating each operation with the data on which it operates.
- This partitioning yields a number of tasks, each comprising some data and a set of operations on that data
  - An operation may require data from several tasks

Methodical Design: Data Decomposition

- The data that are decomposed may be
  - the input to the program
  - the output computed by the program
  - intermediate values maintained by the program
- Different partitions may be possible. Good rules of thumb are to focus first
  - on the largest data structure or
  - on the data structure that is accessed most frequently

Methodical Design: Data Decomposition

- Summation of N numbers $S = \sum_{i=0}^{N-1} X_i$



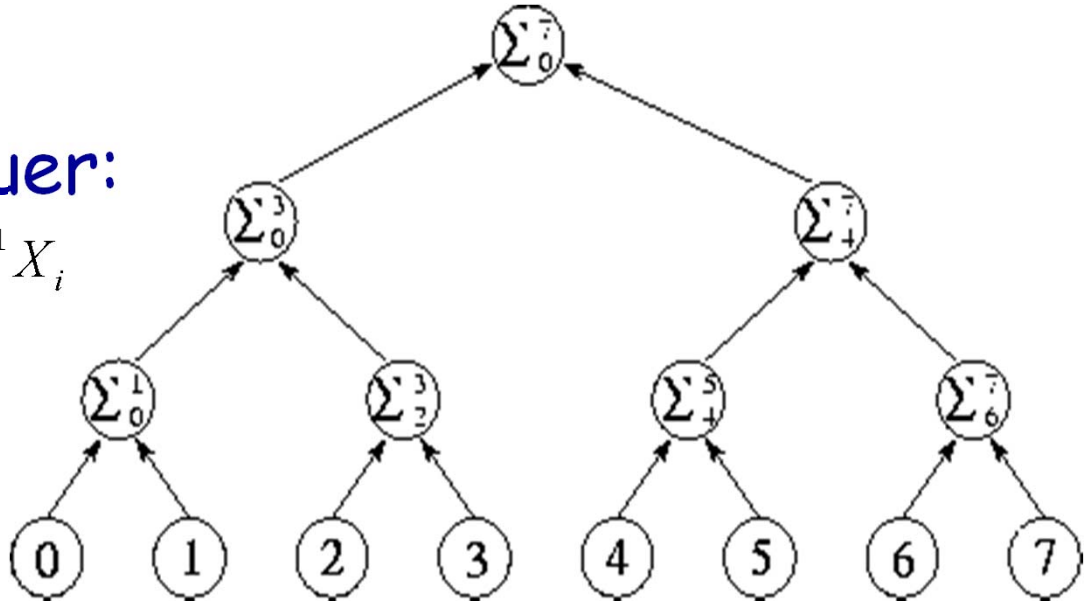
Serial computation

Use recursion:

$$S_i = S_{i-1} + X_{N-i}$$

Use divide & conquer:

$$\sum_{i=0}^{2N-1} X_i = \sum_{i=0}^{N-1} X_i + \sum_{i=N}^{2N-1} X_i$$

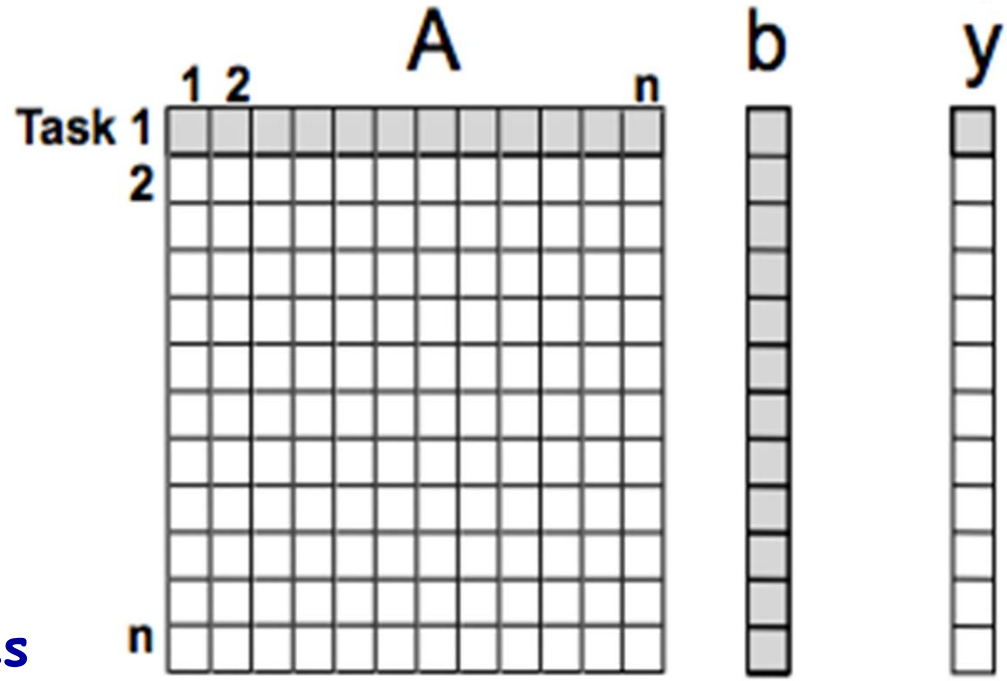


Parallel computation

Methodical Design: Data Decomposition

- Computing each element of output vector y is independent
- Easy to decompose dense matrix-vector product into tasks
  - one per element in y
- Observations
  - task size is uniform
  - no control dependences between tasks
  - tasks share *b*

Matrix-vector product



Methodical Design: Functional Decomposition

- First partition the computation that is to be performed rather than on the data manipulated by the computation
- Potential problem: Limited degree of parallelism
- The case in computer models of complex systems, which may be structured as collections of simpler models (each of which may be computed in parallel) connected via interfaces (i.e., Modular Design)

Methodical Design: Functional Decomposition

- **Database query processing:**

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

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

Table 3.1 A database storing information about used vehicles.

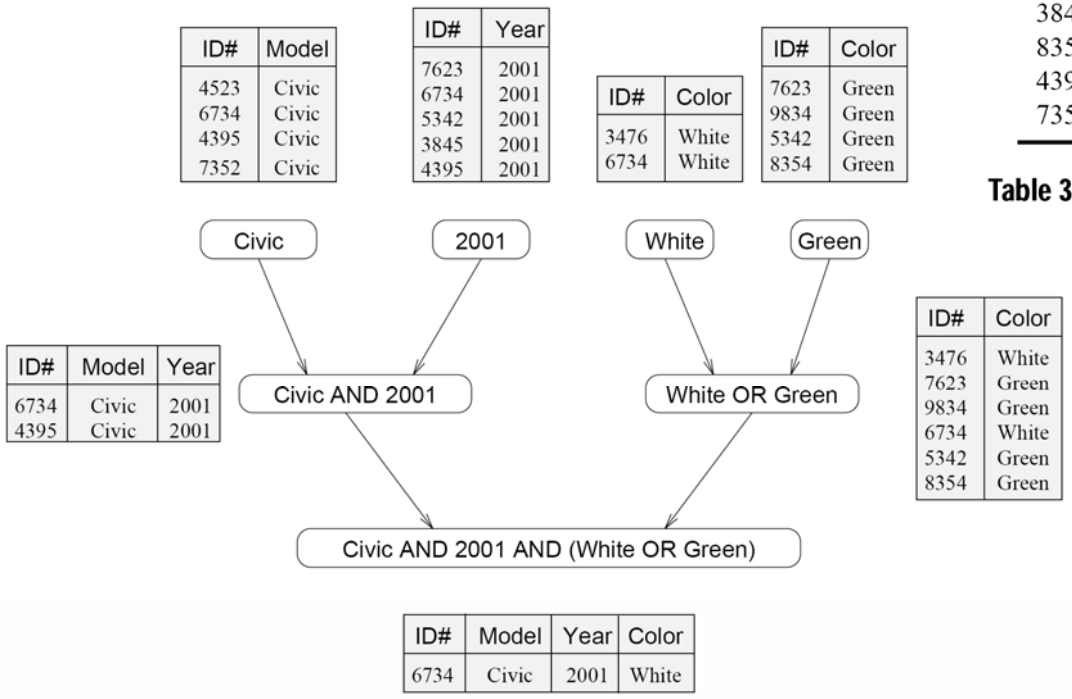


Figure 3.2 The different tables and their dependencies in a query processing operation.

Methodical Design: Partitioning

- Good partitioning:
  - partition at least an order of magnitude more tasks than there are processors in your target computer
  - tasks should be of comparable size
  - the number of tasks should scale with problem size.
  - avoid unnecessary redundant computation and storage requirements.
  - try several alternative partitions

Methodical Design: Communication

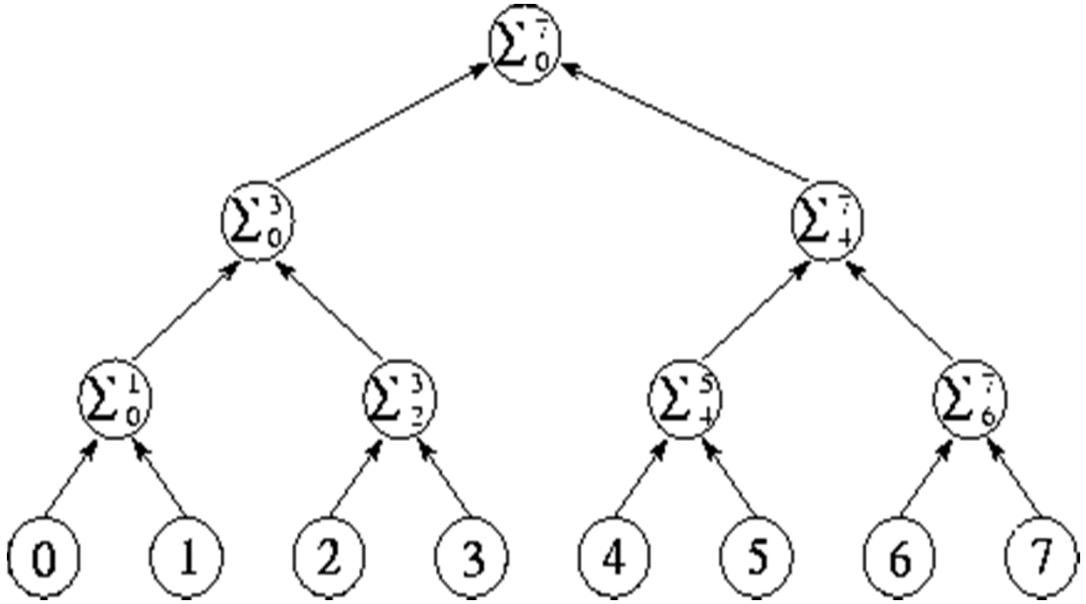
- The computation to be performed in one task will typically require data associated with another task
- Data must then be transferred between tasks so as to allow computation to proceed
- The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently
- Communication is then required to manage the data transfer necessary for these tasks to proceed
- Organizing this communication in an efficient manner can be challenging. Even simple decompositions can have complex communication structures

Methodical Design: Communication

- Communication patterns:
  - local vs global
    - In *local* communication, each task communicates with a small set of other tasks (its ``neighbours'')
    - *global* communication requires each task to communicate with many tasks
  - static vs dynamic
    - In *static* communication, the identity of communication partners does not change over time
    - communication partners in *dynamic* communication structures may be determined at runtime and may be highly variable.
  - structured vs unstructured
    - In *structured* communication, a task and its neighbours form a regular structure, such as a tree or grid
    - *unstructured* communication networks may be arbitrary graphs

Methodical Design: Communication

- Simple communication patterns may be represented using task-dependency graphs
- In a task-dependency graph each node represents a task and the arrows between nodes indicate dependencies between tasks



Methodical Design: Communication

- Preferred communication patterns:
  - Communication operations balanced among tasks
  - each task communicates only with a small number of neighbours
  - the computation associated with different tasks should be able to proceed concurrently
  - communication operations should be able to proceed concurrently

Partitioning & Communication

- The two steps are often considered simultaneously
- The main purpose is to identify the maximum degree of parallelism
- Understand the nature of the problem is essential

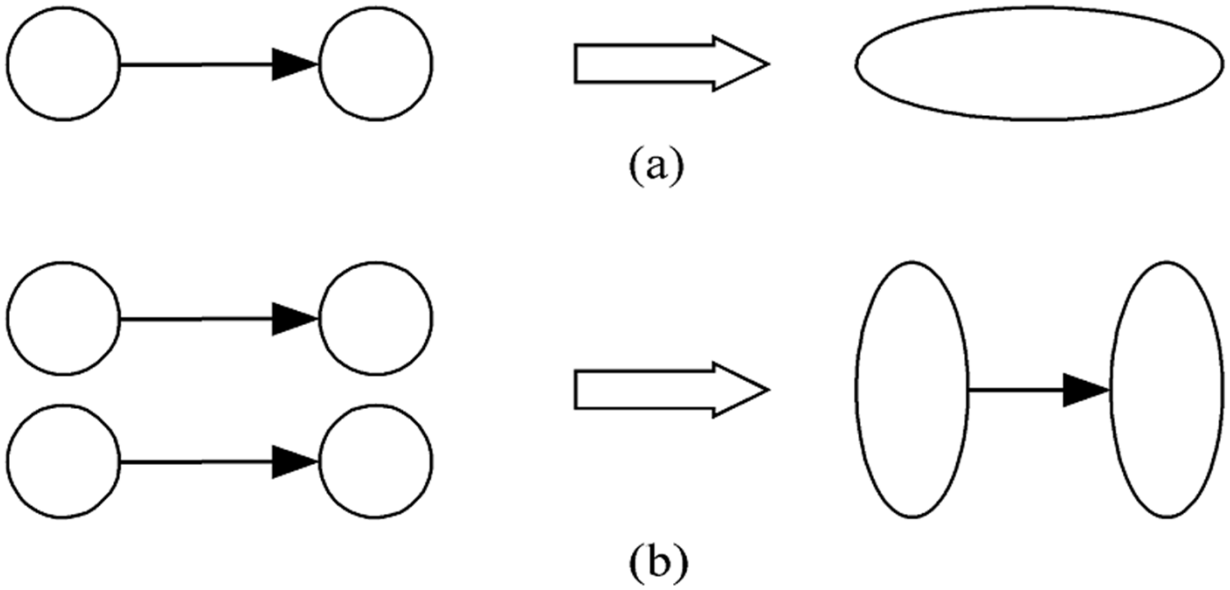
Methodical Design: Agglomeration

- In the partitioning phase of the design process, our efforts are focused on defining as many tasks as possible (fine-grained parallelism)
- However, defining a large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm
- In the agglomeration stage, we move from the abstract toward the concrete
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming

Methodical Design: Agglomeration

- The outcome can impact the choice and performance of parallel algorithms
- Important issues:
  - Load balancing
  - Communication costs
- Improve performance by eliminating communication between primitive tasks agglomerated into consolidated task
  - sending less data
  - using fewer messages, even if the same amount of data

Methodical Design: Agglomeration



Combine groups of sending and receiving tasks

Methodical Design: Agglomeration

- There are other ways to reduce communication costs, e.g., replicate data and/or computation
- Good agglomeration:
  - reduce communication costs and increase locality
  - yield tasks with similar computation and communication costs
  - without introducing load imbalances
  - the number of tasks should still scale with problem size

Methodical Design: Mapping

- To specify where each task is to execute
- This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling
- In these computers, a set of tasks and associated communication requirements is a sufficient specification for a parallel algorithm and operating system or hardware mechanisms to rely on to schedule executable tasks to available processors
- Unfortunately, general-purpose mapping mechanisms have yet to be developed for scalable parallel computers

Methodical Design: Mapping

- The goal in developing mapping algorithms is normally to minimize total execution time
- Mappings must minimize overheads – communication and idling
- Strategies:
  - place tasks that communicate frequently on the *same* process, so as to increase locality
  - place tasks that are able to execute concurrently on *different* processes, so as to enhance concurrency
- Minimizing the overheads often represents contradicting objectives
  - Assigning all work to one processor trivially minimizes communication at the expense of significant idling

Methodical Design: Mapping

- Mapping must simultaneously minimize idling and load balance. Merely balancing load does not minimize idling

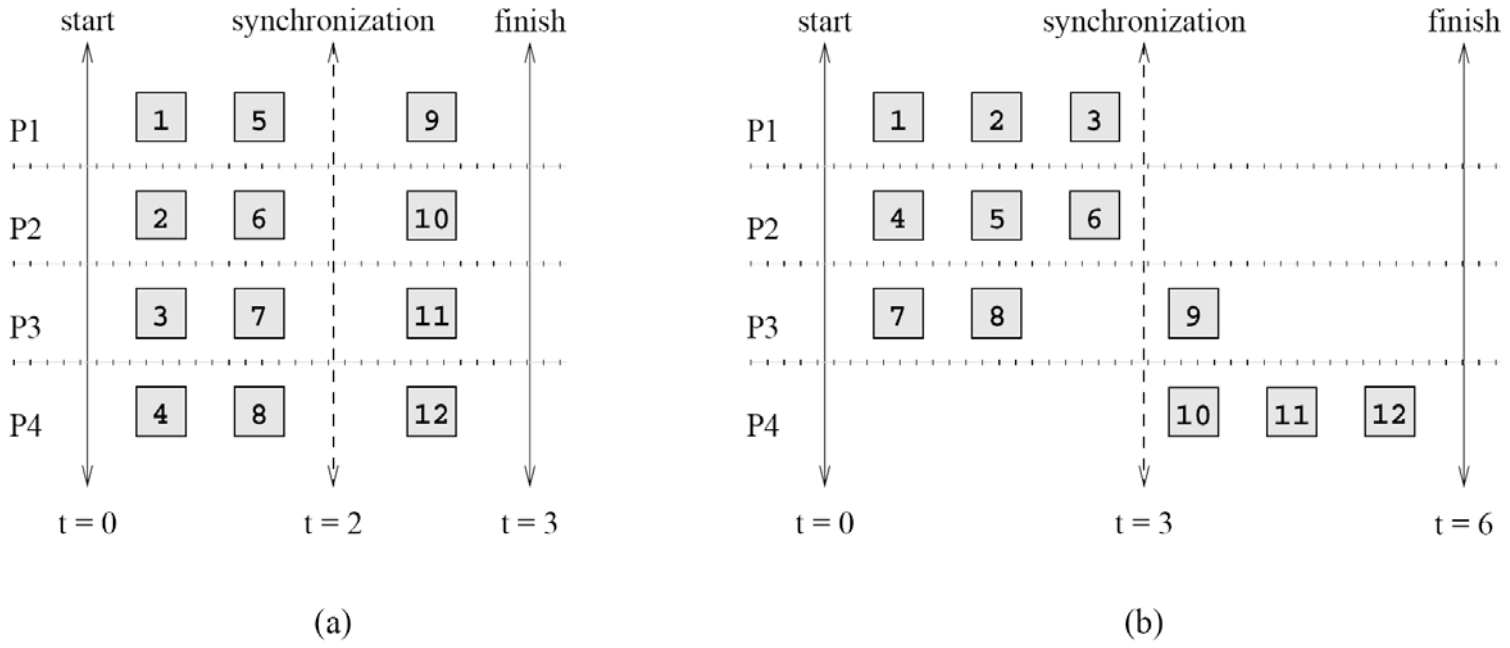


Figure 3.23 Two mappings of a hypothetical decomposition with a synchronization.

Methodical Design: Mapping

- The mapping problem is known to be *NP -complete*, meaning that no computationally tractable (polynomial-time) algorithm can exist for evaluating these tradeoffs in the general case
- However, considerable knowledge has been gained on specialized strategies and heuristics which are very effective for many practical problems
- Mapping techniques can be static or dynamic
  - Static Mapping: Tasks are mapped to processors a-priori. For this to work, we must have a good estimate of the size of each task
  - Dynamic Mapping: Tasks are mapped to processors at runtime. This may be because the tasks are generated at runtime, or that their sizes are not known

Methodical Design: Static Mapping

- Based on data partitioning - widely used in dense matrix computation
- The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes

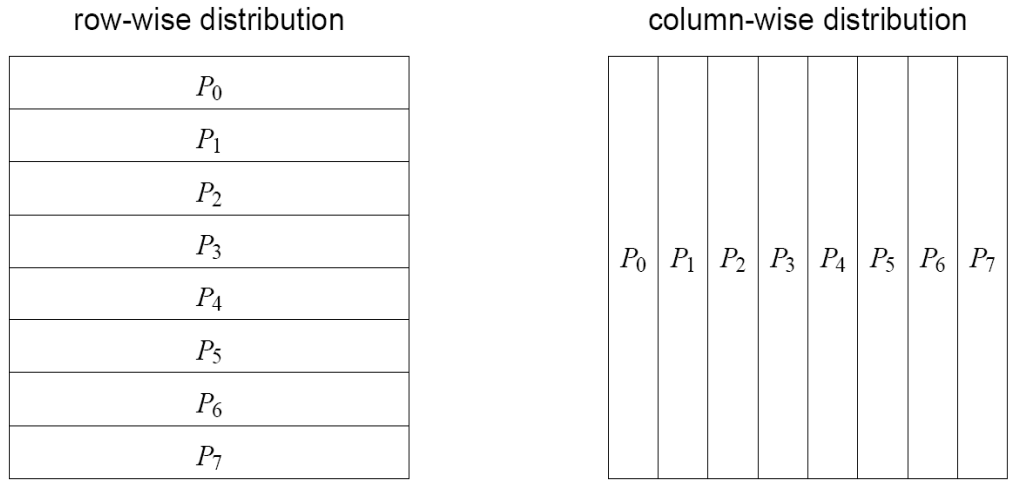


Figure 3.24 Examples of one-dimensional partitioning of an array among eight processes.

Methodical Design: Static Mapping

- Block distribution scheme can be generalized to higher dimensions:

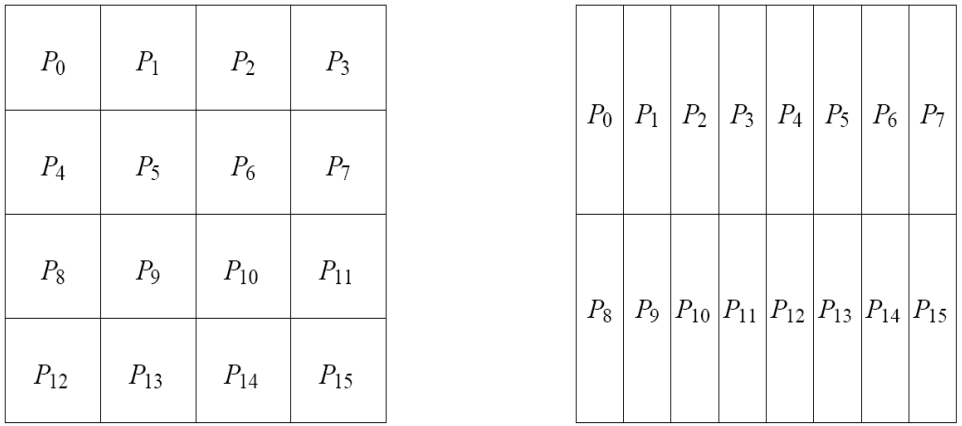
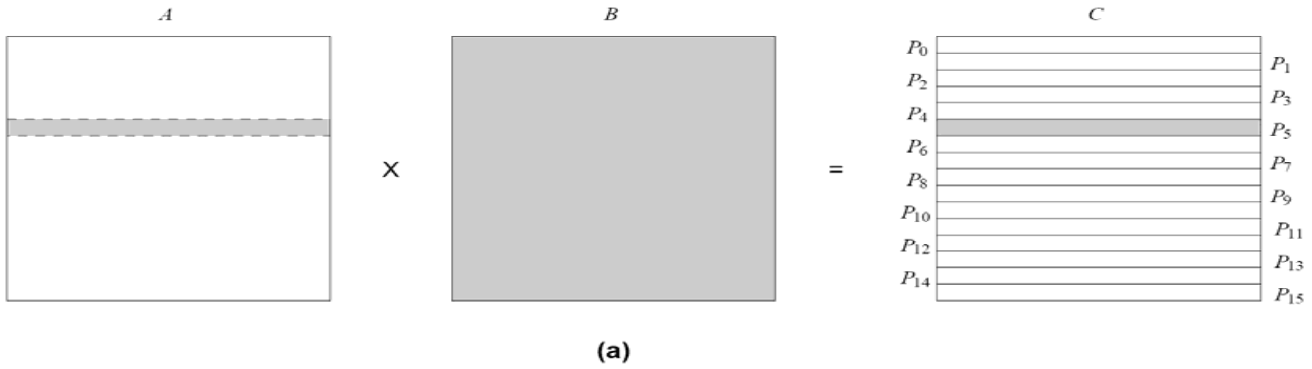


Figure 3.25 Examples of two-dimensional distributions of an array, (a) on a 4×4 process grid, and (b) on a 2×8 process grid.

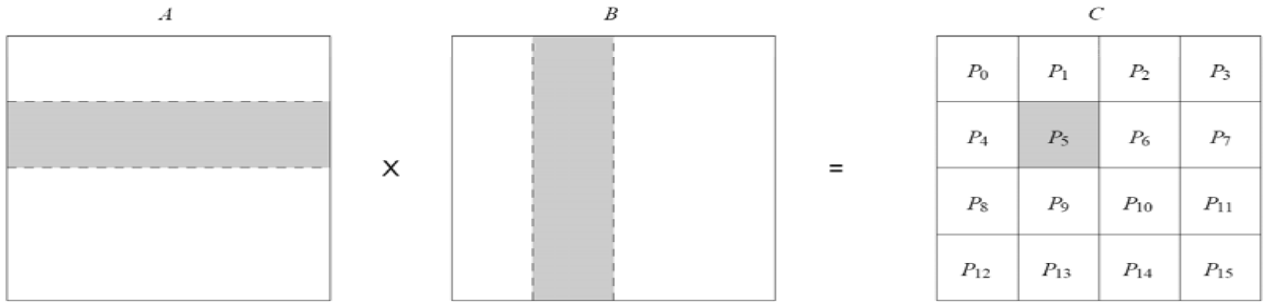
Methodical Design: Static Mapping

- Matrix multiplication $C = A \times B$:
- Partition the output matrix C using a block decomposition
- Give each task the same number of elements of C
  - each element of C corresponds to a dot product
  - even load balance
- Obvious choices: 1D or 2D decomposition
- Select to minimize associated communication overhead

Methodical Design: Static Mapping



(a)

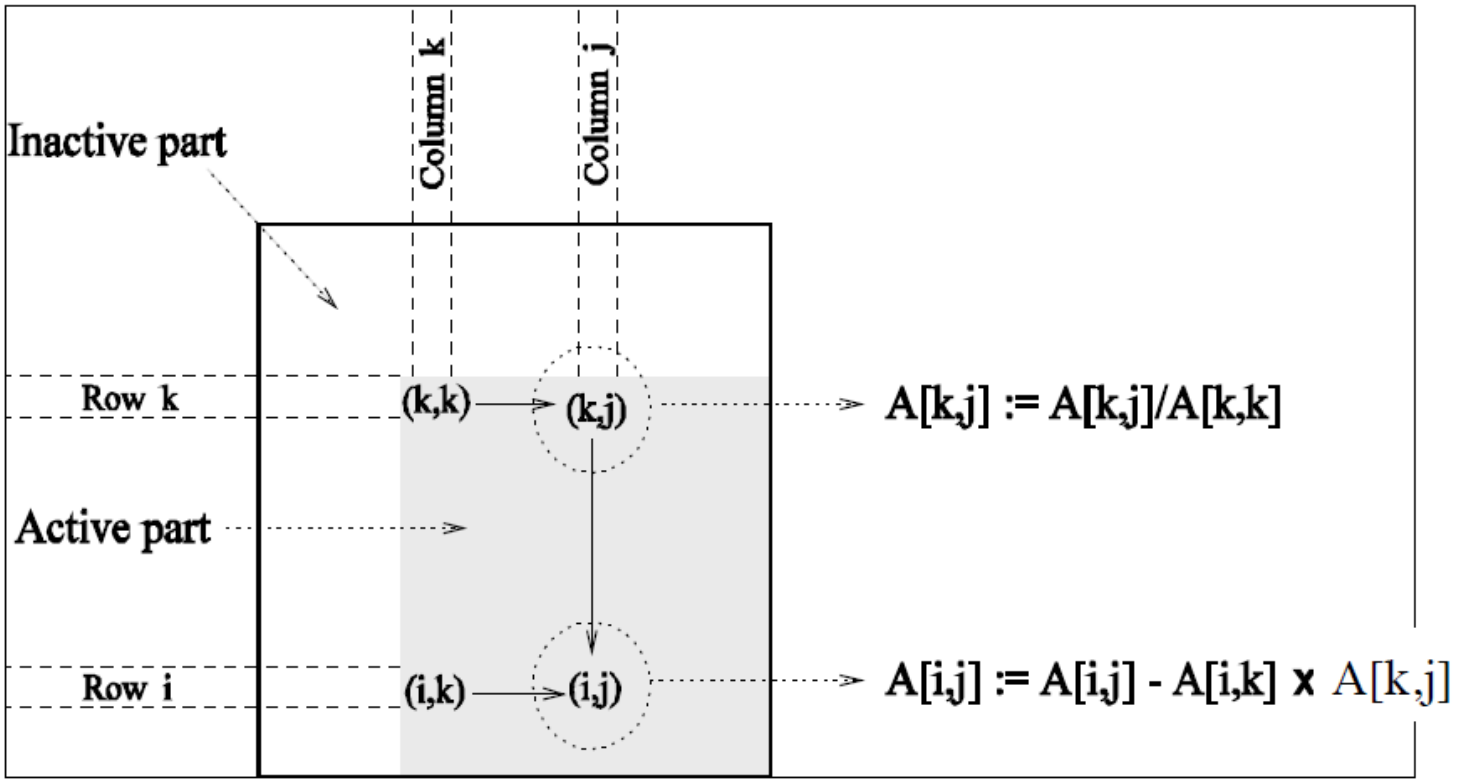


(b)

- Question: how are A and B stored on a distributed memory machine?

Methodical Design: Static Mapping

- Gaussian Elimination:
- Active submatrix shrinks as elimination progresses

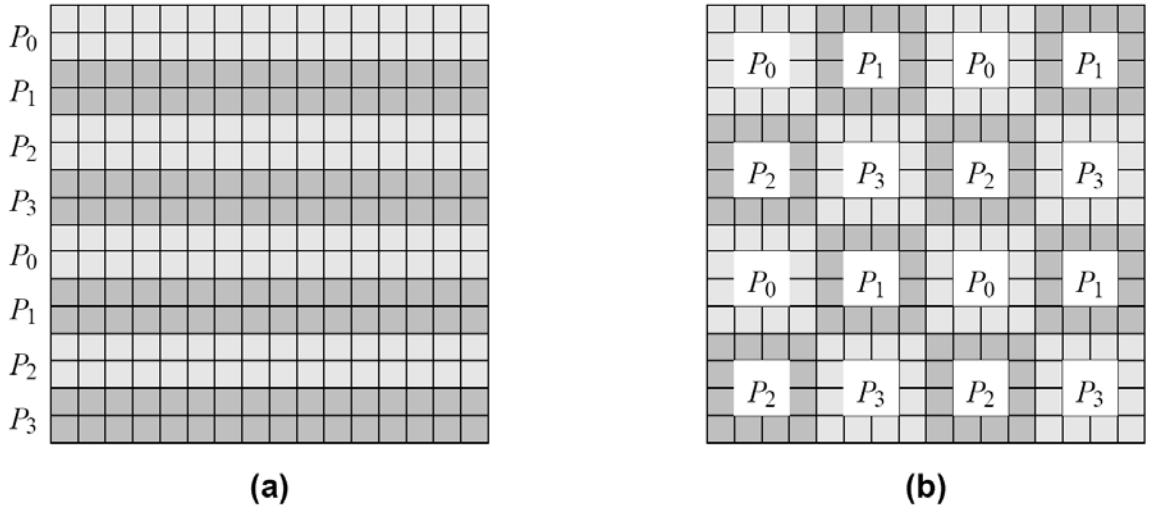


Methodical Design: Static Mapping

- Consider a block distribution for Gaussian Elimination
  - —amount of computation per data item varies
  - —a block decomposition would lead to significant load imbalance
- Another computation with similar distribution challenges
  - —LU decomposition of a dense matrix
- **Block Cyclic Distribution**
  - 1. partition an array into many more blocks than the number of available processes
  - 2. assign blocks to processes in a round-robin manner
    - each process gets several non-adjacent blocks

Methodical Design: Static Mapping

- Block-cyclic distribution scheme:



- Cyclic distribution: special case with block size = 1
- Block distribution: special case with block size is *n/p*,
  - —*n* is the dimension of the matrix; *p* is the # of processes

Methodical Design: Static Mapping

- Other static mapping schemes:
  - Based on task partitioning
  - Based on graph partitioning
  - Hierarchical mapping

Methodical Design: Dynamic Mapping

- Dynamic mapping is sometimes also referred to as dynamic load balancing, since load balancing is the primary motivation for dynamic mapping
- Dynamic mapping schemes can be centralized or distributed

Methodical Design: Centralized Dynamic Mapping

- Processes are designated as masters or workers
- When a worker process runs out of work, it requests the master for more work
- When the number of processes increases, the master may become the bottleneck
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling
- Selecting large chunk sizes may lead to significant load imbalances as well
- Gradually decrease chunk size as the computation progresses

Methodical Design: Distributed Dynamic Mapping

- Executable tasks are distributed among processes which exchange tasks at run time to balance work
- Each process can send or receive work from other processes
- This alleviates the bottleneck in centralized schemes.
- There are four critical questions:
  - how are sending and receiving processes paired together?
  - who initiates work transfer?
  - how much work is transferred?
  - when is a transfer triggered?
- Answers to these questions are generally application specific

Minimizing Interaction Overheads

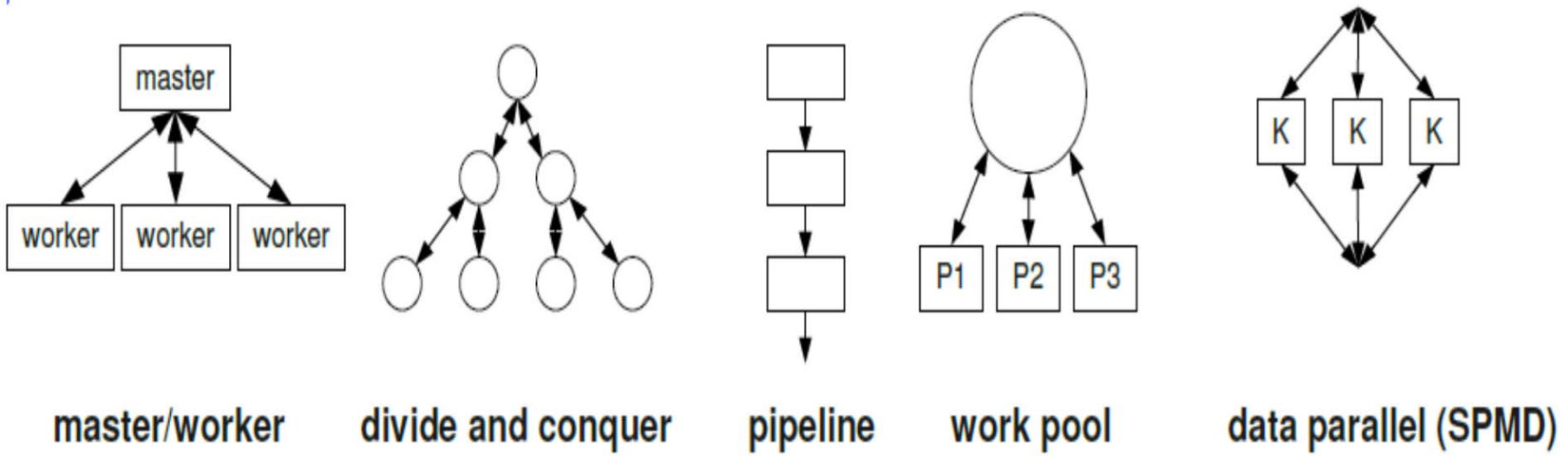
- Maximize data locality: Where possible, reuse intermediate data
- Minimize volume of data exchange: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated
- Minimize frequency of interactions: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible

Minimizing Interaction Overheads

- Overlapping computations with communications: Use non-blocking communications, multithreading, and pre-fetching to hide latencies
- Replicating data or computations

Common Parallel Programming Models

- There are many parallel programming models
- In reality applications usually combine different models



master/worker divide and conquer pipeline work pool data parallel (SPMD)

Additional References

- Ian Foster, Designing and Building Parallel Programs, http://www.mcs.anl.gov/~itf/dbpp/