ECE 677: *Distributed Computing Systems*

Salim Hariri

High Performance Distributed Computing Laboratory

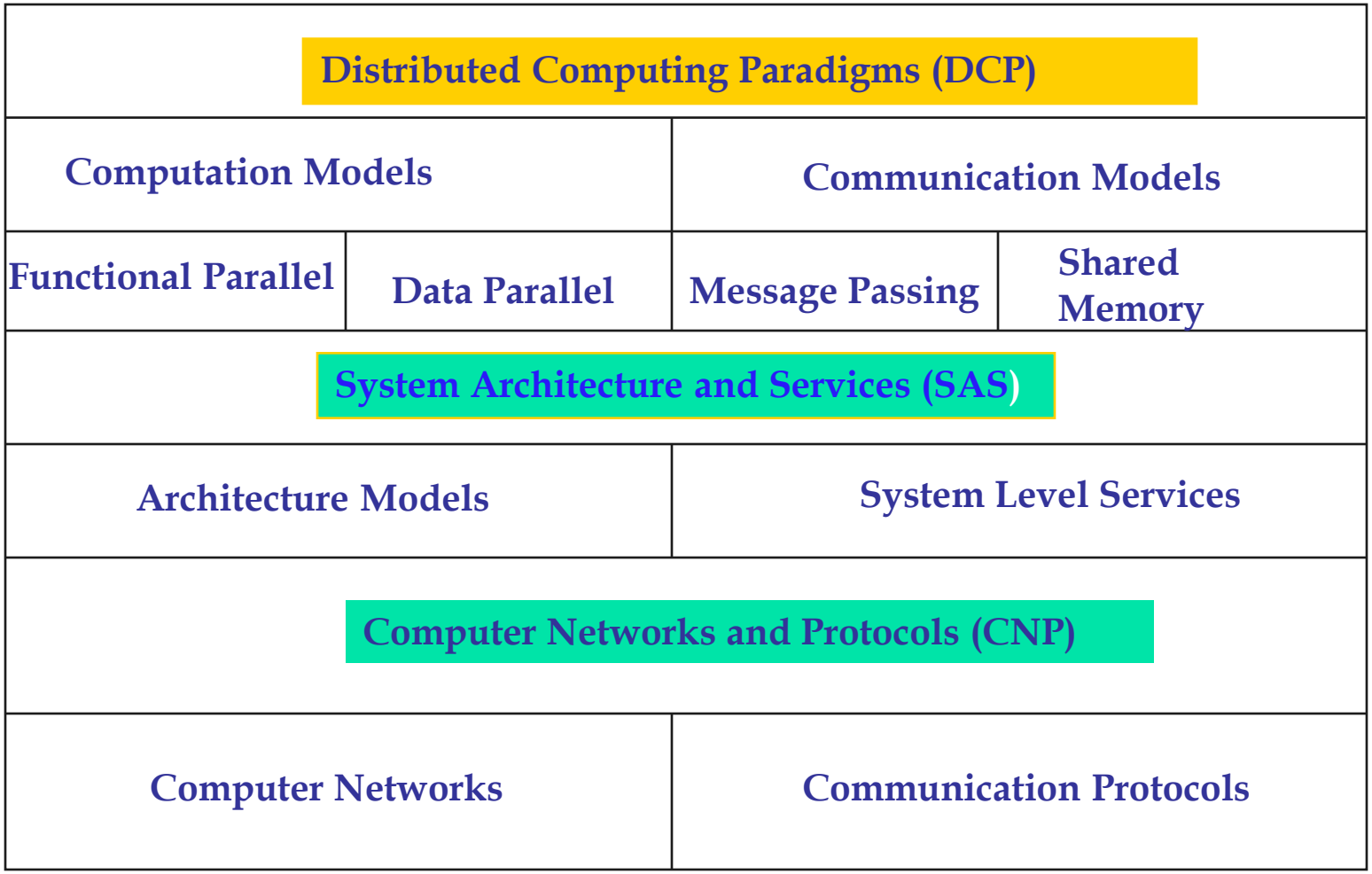
University of Arizona

Tele: (520) 621-4378

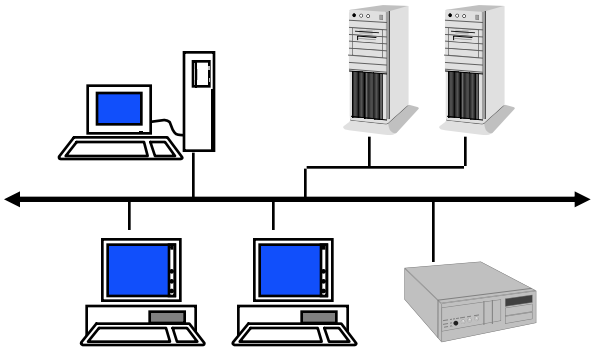
Acl.arizona.edu//classes/ece677

Fall 2013

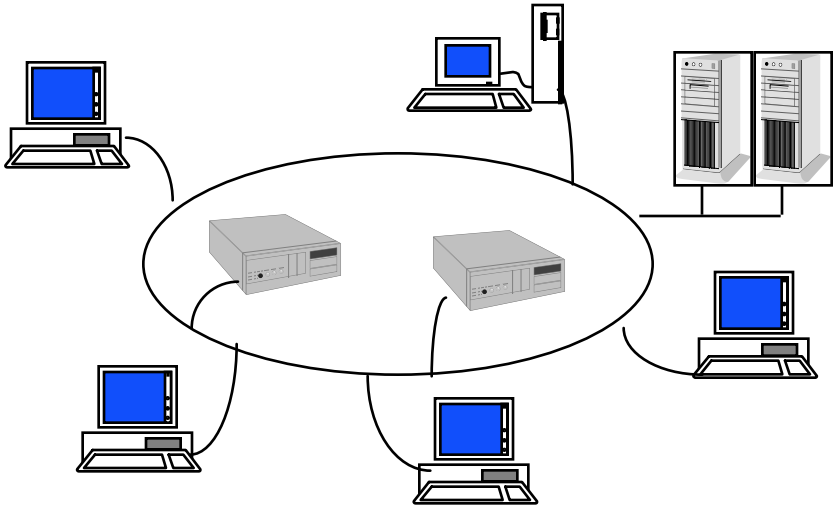
Distributed Systems Design Framework (Cont)



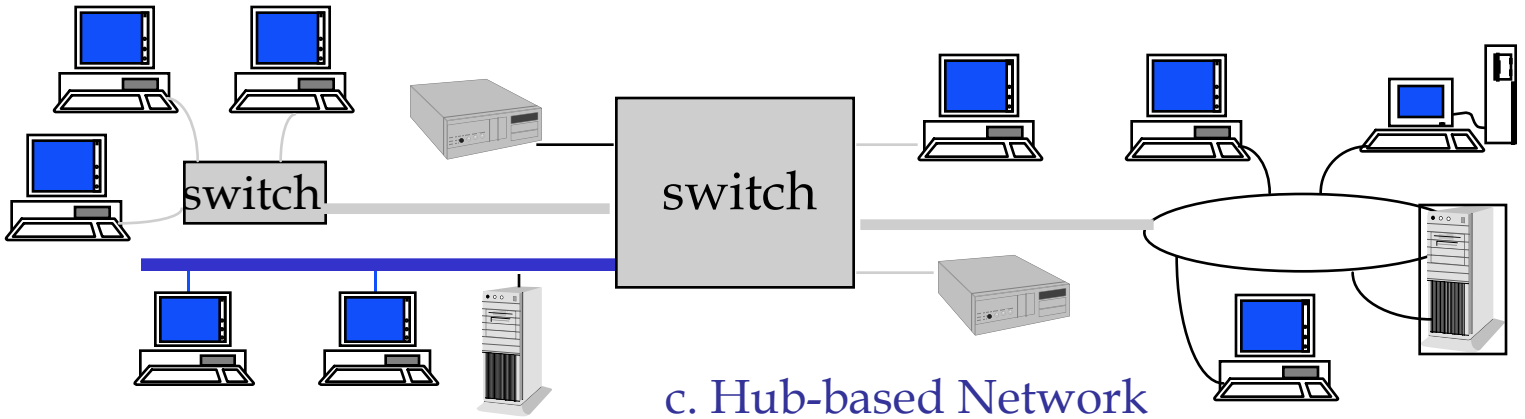
Network Technologies



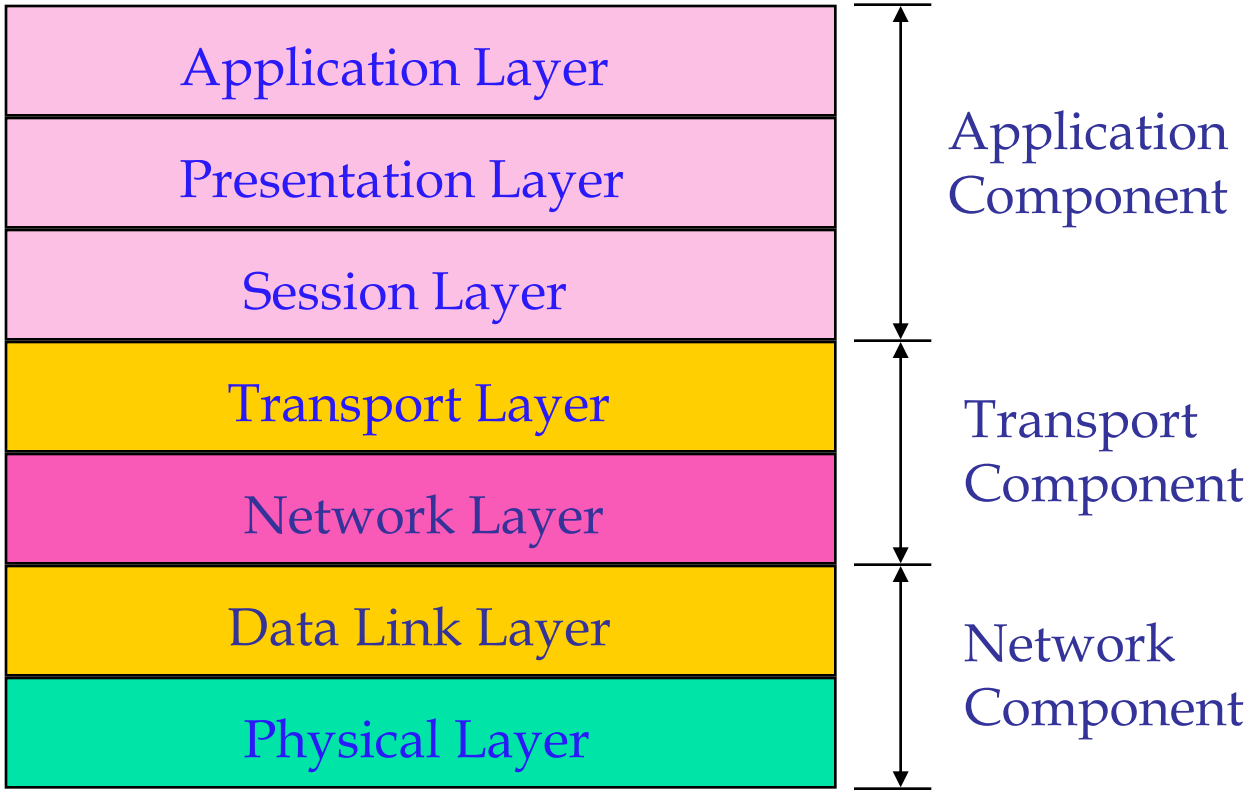
a. BUS Network



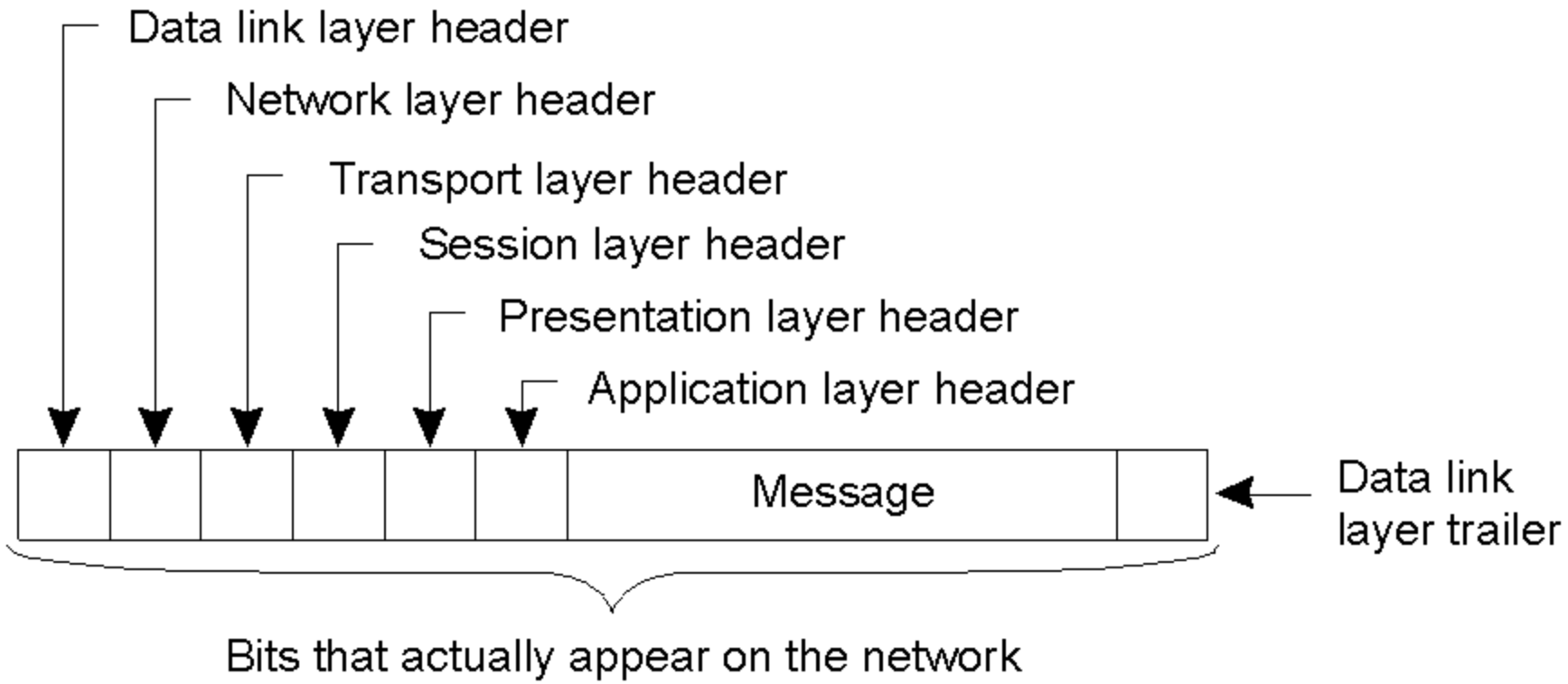
b. RING Network



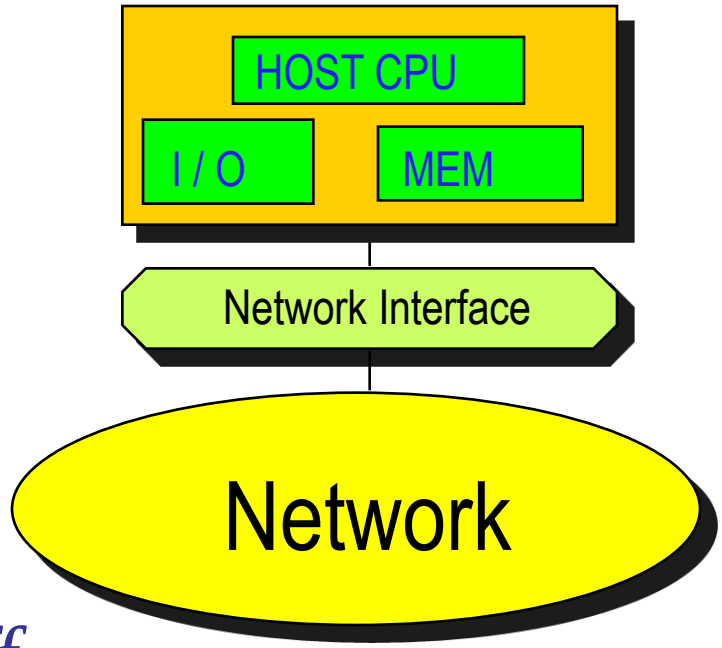
c. Hub-based Network


Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer
Application Component
Transport Component
Network Component

Layered Protocols



Network Interface



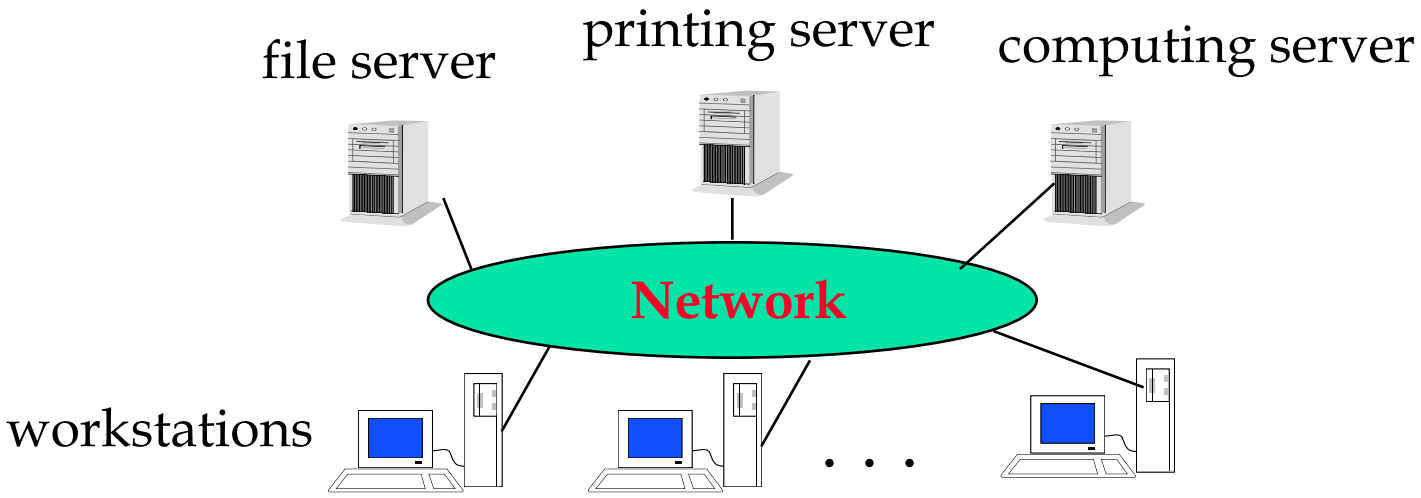
- Responsible for transferring data from host memory to the communication medium and vice versa
- Perform functions related to message assembly, formatting, routing and error control

Tradeoff:

- more functions allocated to the network interface, the less load imposed on the host to process network functions
- however, the cost of the network interface will increase

Architectural Models

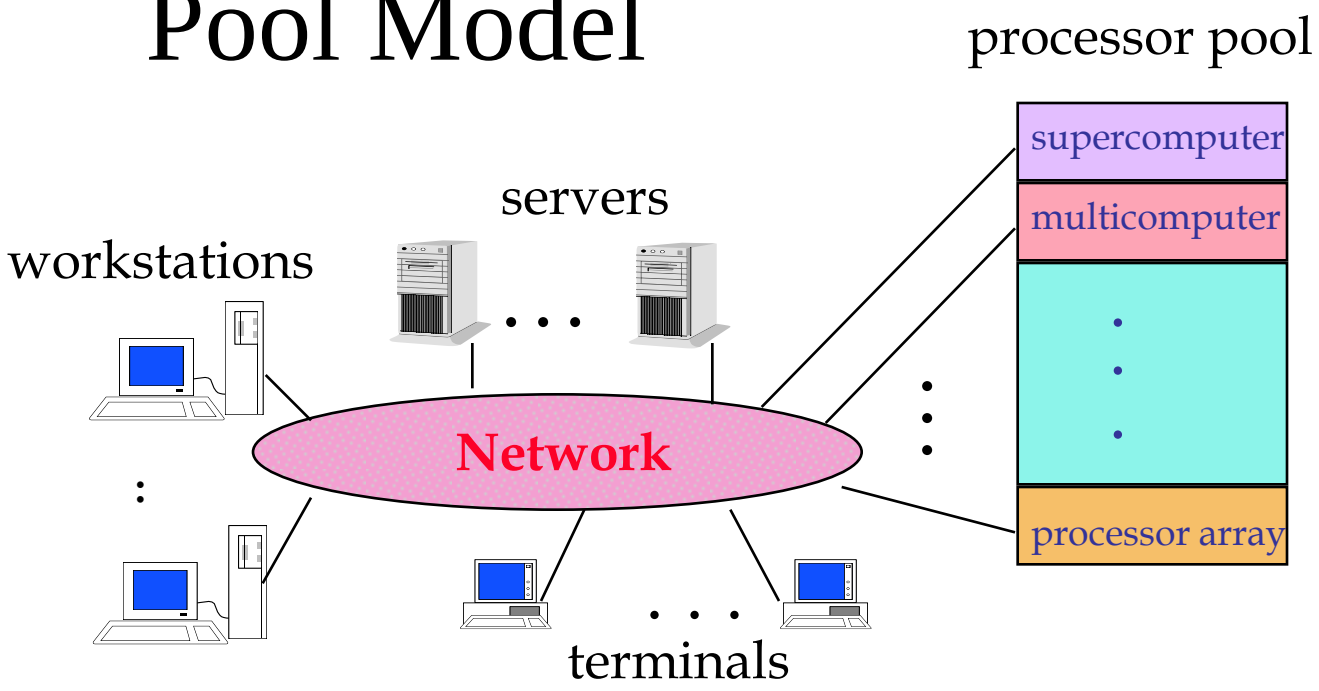
- **Server Model or client/server model**



Workstation/Server Model

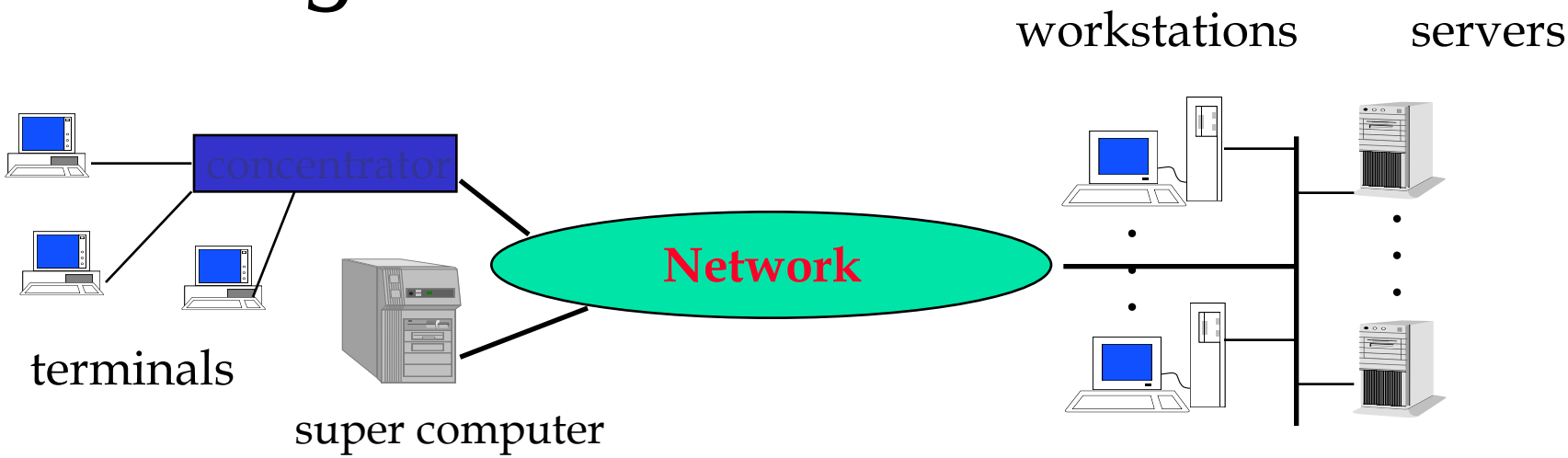
- Majority of dist.systems are based on this model
- Share data between users and applications
- Share file servers and directory servers
- Share expensive peripheral equipment

Pool Model



- Rack full of CPUs which can be dynamically allocated to users on demand
- Users given high-performance graphics terminals, such as X terminals
- All the processors belong equally to everyone
- Advantage: better utilization of resources
- Disadvantages: increased communication between the application program and the terminal, and limited capabilities of terminals

Integrated Model



Has advantages of using networked resources and centralized computing systems
- Each computer performs both the role of a server and the role of a client
- Computing resources managed by a single dist. operating system that makes them appear to the user as a single image system
- Global naming scheme is supported allowing individual computers to share data and files without regard to their location

Hybrid Model
- Viewed as a collection of three architectural models
- Example: *Amoeba* system combines Server and Pool Models

Distributed Computing Paradigms: Layer 3

- Layer 2 describes the architectural models, component properties and services
- It describes what is required to build a distributed system
- Layer 3, it describes how you program distributed applications; what techniques (models) do you use?
- Also, it describes the types of tools that can be used to implement the applications

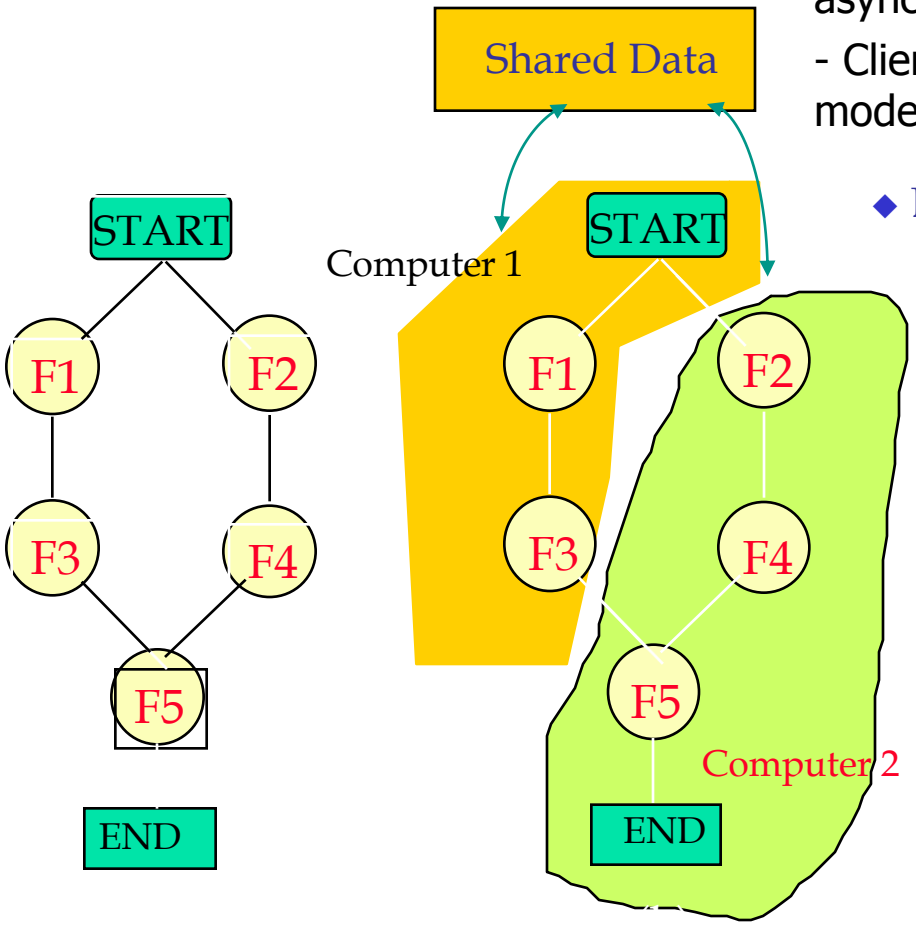
There are two sub-layers:

Computational Model

- **Communication Model**

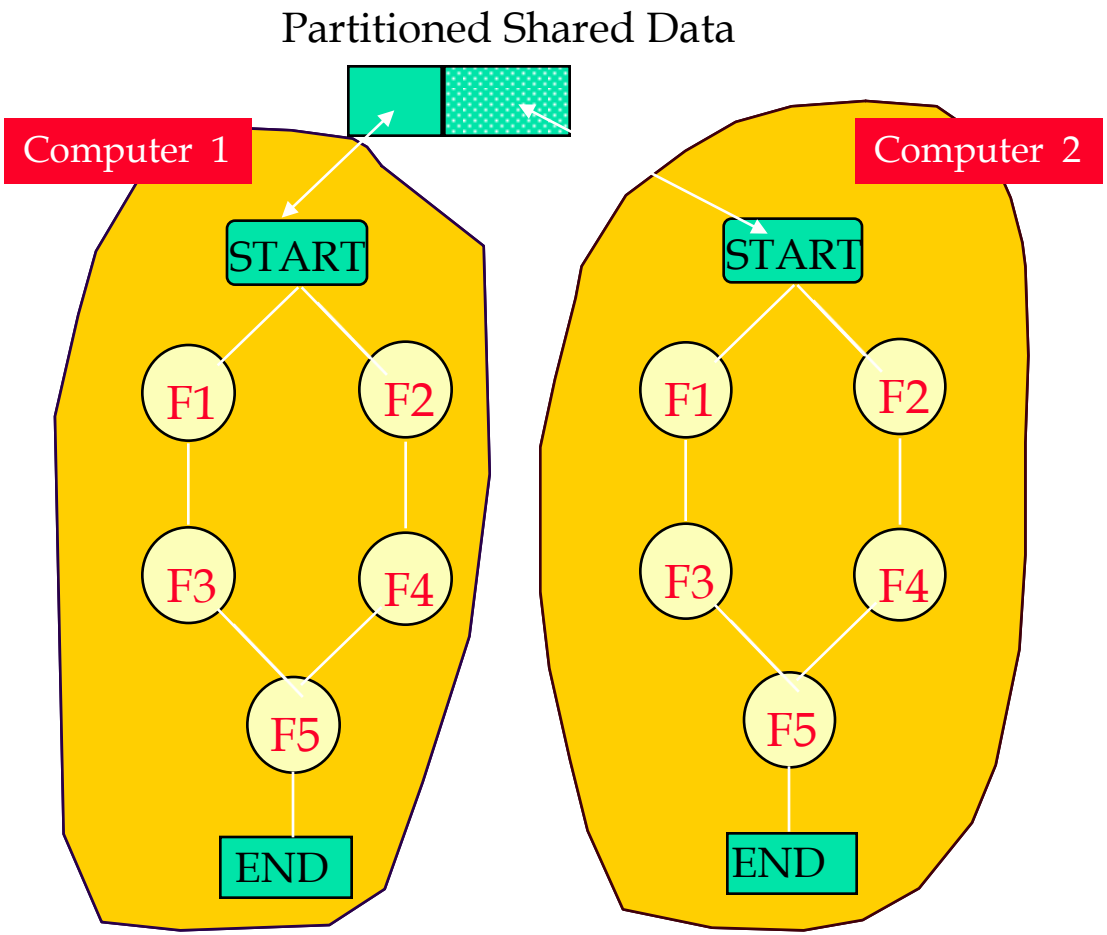
Computational Model: Functional Model

- Functional Parallel Model
  - Computers execute different threads of control,
  - It is referred to as control parallelism, asynchronous parallelism
  - Client-Server Models are variations of this model



- Limitations of Functional Parallelism
  - Asynchronous interactions could lead to data race conditions
  - If the application has large number of parallel tasks, it is difficult to achieve good load balancing

Computational Models: Data Model



- Large number of problems can be solved using this model
- It is easier to develop applications

- Amount of parallelism in functional parallelism is fixed
- Amount of parallelism in data parallelism scales with the data size
- Generally speaking, efficient distributed applications should exploit both types of parallelism

Communications Models

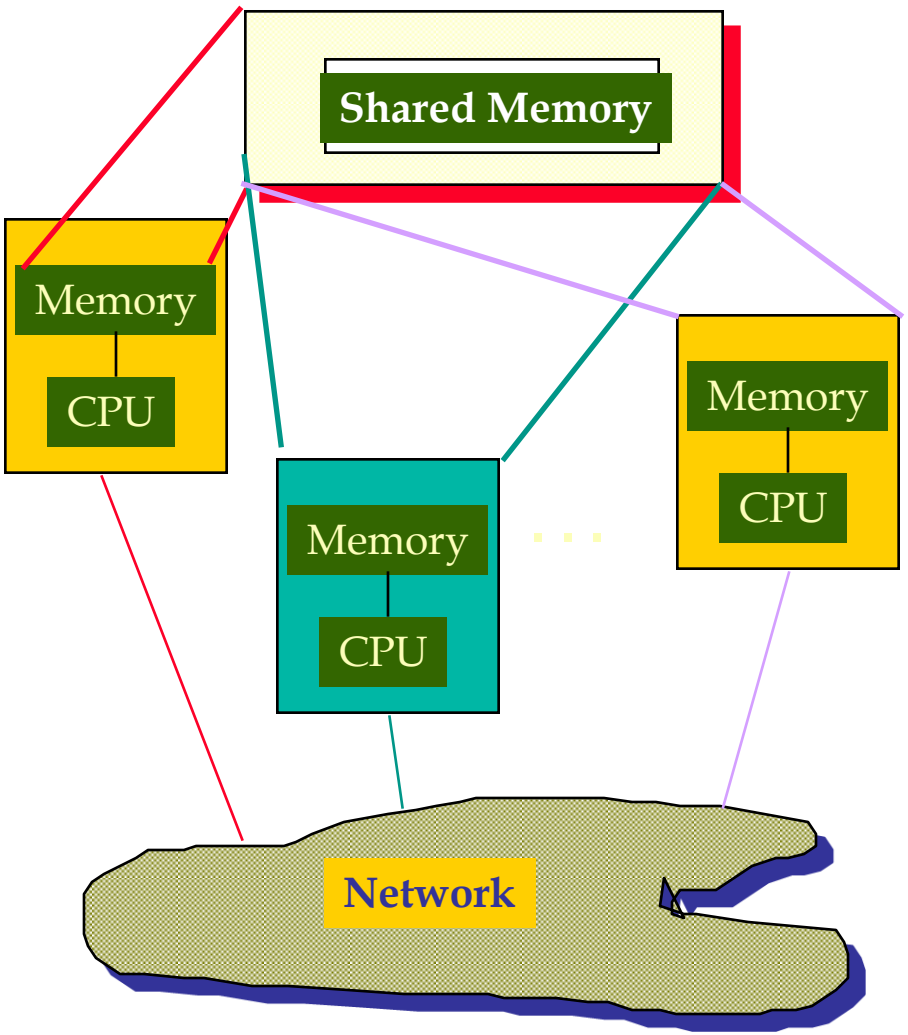
Message passing model

- Messages are used to exchange information between local and remote processes as well as between processes and the operating system
- Application developers need to explicitly involved in writing the communication and synchronization routines
- Users use two basic communications primitives: SEND and RECEIVE
- SEND and RECEIVE primitives have different implementations depending on whether or not they are blocking or Nonblocking, synchronous or asynchronous.
- The main limitations are:

  * synchronizing request and response messages
  * handle data representations
  * machine addresses
  * handle system failures that could be related to communications network or compute failures
  * debugging and testing is difficult

Distributed Shared Memory

- In message passing model, the communication between processes is highly controlled by a protocol and involves explicit cooperation between processes
- Direct shared memory communication is not explicitly controlled and requires the use of a global shared memory
- Message communication resembles the operation of postal service in sending and receiving mail
- The shared memory scheme can also be compared to a bulletin board, found in a grocery store or supermarket; it is a central repository for existing information that can be read or updated by anyone



Distributed Computing Programming Paradigms

- Distributed Computing Models
  - What is Parallelism
  - Parallel Computing Models and Architectures
  - Data Parallel Model
  - Functional Parallel Model
- Distributed Communication Models
  - Shared Memory, Distributed SM
  - Message Passing Model
- Steps for Creating a Parallel/Distributed Program
- Design and Performance Considerations
- Message Passing Interface (MPI)
- Running MPI Programs and Examples

Distributed Programming Paradigms: What is Parallelism?

- A strategy for performing large, complex tasks faster.
- A large task can either be performed serially, one step following another, or can be decomposed into smaller tasks to be performed simultaneously, i.e., in parallel.
- Parallelism is done by:
  - Breaking up the task into smaller tasks
  - Assigning the smaller tasks to multiple workers to work on simultaneously
  - Coordinating the workers
- Parallel problem solving is common. Examples: building construction; operating a large organization; automobile manufacturing plant

Distributed/Parallel Programming Paradigm

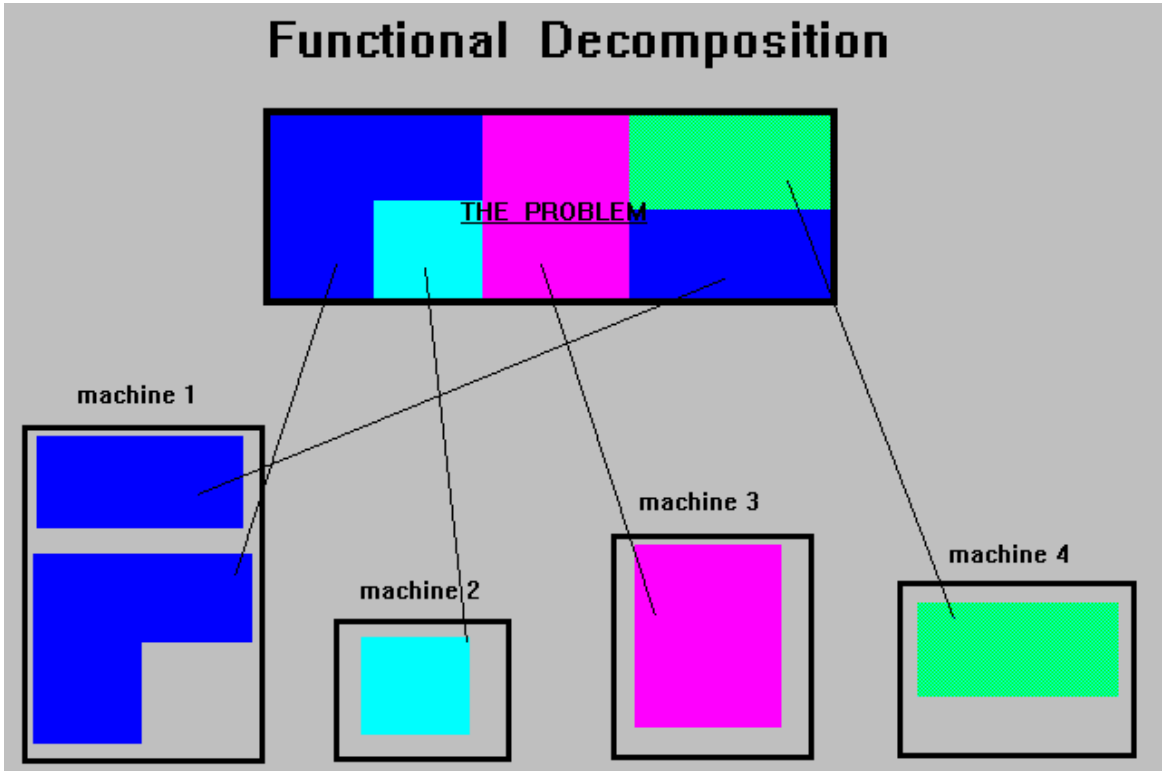
- Parallel programming involves:
  - Decomposing an algorithm or data into parts
  - Distributing the parts as tasks which are worked on by multiple processors simultaneously
  - Coordinating work and communications of those processors
- Parallel/Distributed programming considerations:
  - Type of parallel model being used
  - Type of communication model being used

Steps for Writing a Parallel/Distributed Programs

- If you are starting with an existing serial program, debug the serial code completely
- Identify the parts of the program that can be executed concurrently:
  - Requires a thorough understanding of the algorithm
  - Exploit any inherent parallelism which may exist.
  - May require restructuring of the program and/or algorithm. May require an entirely new algorithm.
- Decompose the program:
  - Functional Parallelism
  - Data Parallelism
  - Combination of both
- Code development
  - Code may be influenced/determined by machine architecture
  - Choose a programming paradigm
  - Determine communication
  - Add code to accomplish task control and communications
- Compile, Test, Debug
- Optimization
  - Measure Performance
  - Locate Problem Areas
  - Improve them

Distributed Computing Models: Functional Parallel Model

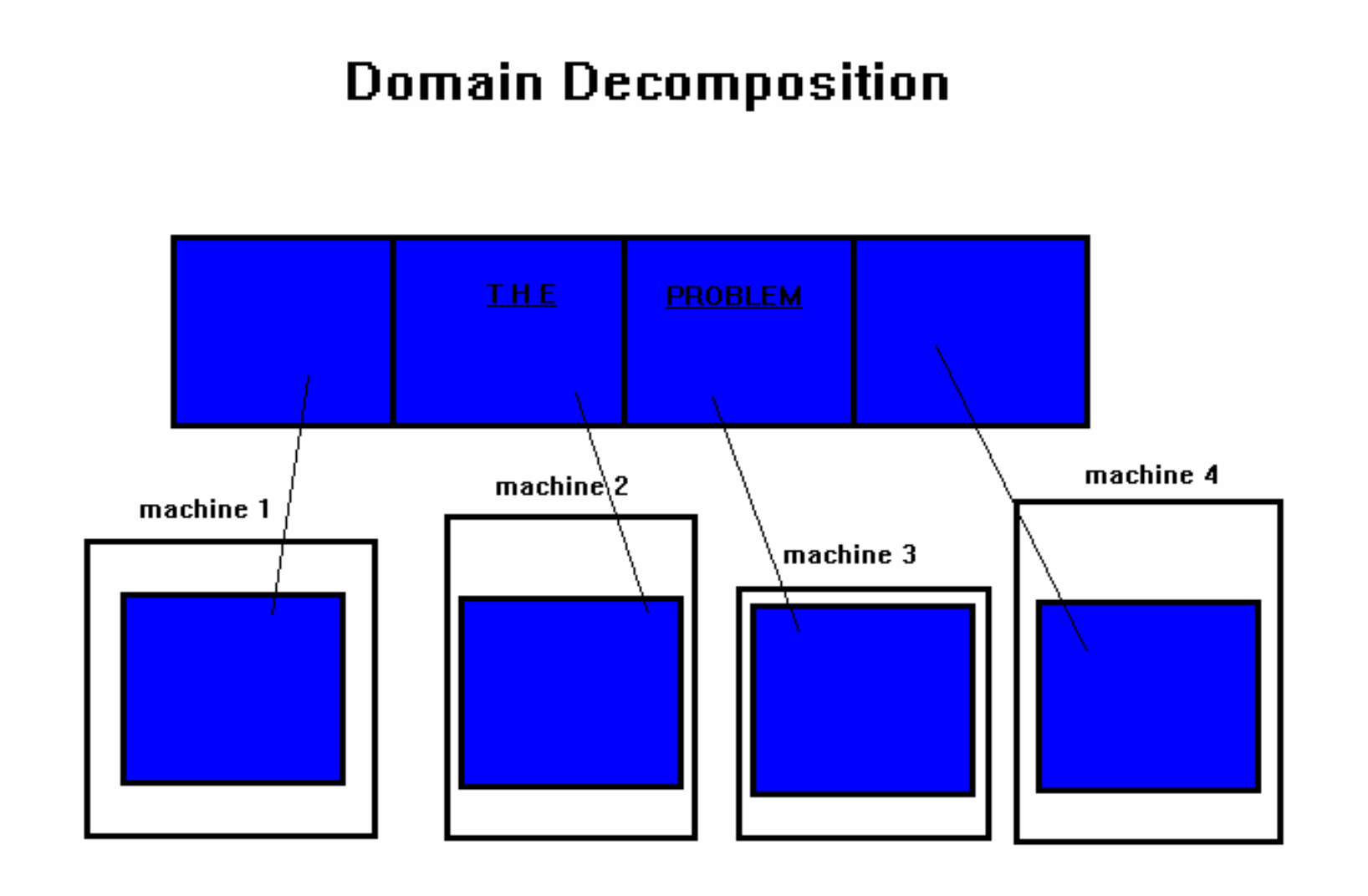
- Functional Decomposition (Functional Parallelism)
  - Decomposing the problem into different tasks which can be distributed to multiple processors for simultaneous execution
  - Good to use when there is not static structure or fixed determination of number of calculations to be performed



Domain Decomposition (Data Parallelism)

- Partitioning the problem's data domain and distributing portions to multiple processors for simultaneous execution
- Good to use for problems where:
  - data is static (factoring and solving large matrix or finite difference calculations)
  - dynamic data structure tied to single entity where entity can be subsetted (large multi-body problems)
  - domain is fixed but computation within various regions of the domain is dynamic (fluid vortices models)

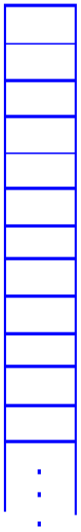
Domain Decomposition (Data Parallelism)



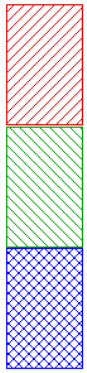
There are many ways to decompose data into partitions to be distributed

- One Dimensional Data Distribution
  - Block Distribution
  - Cyclic Distribution
- Two Dimensional Data Distribution
  - Block Block Distribution
  - Block Cyclic Distribution
  - Cyclic Block Distribution

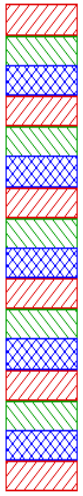
One-dimensional Array A



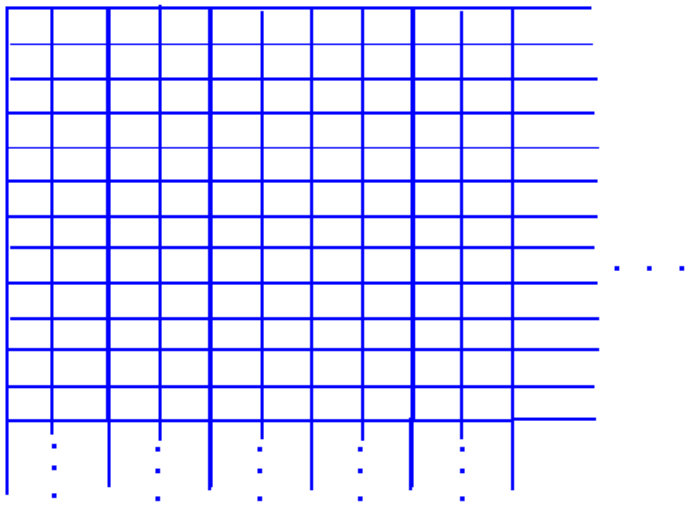
Block Distribution of A



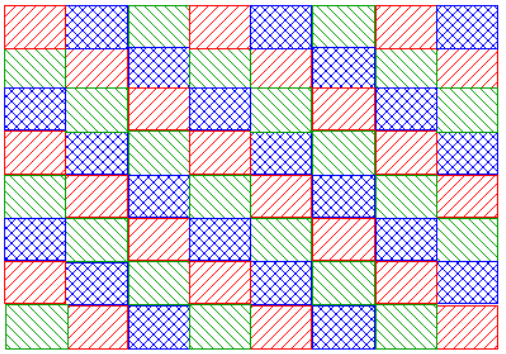
Cyclic Distribution of A



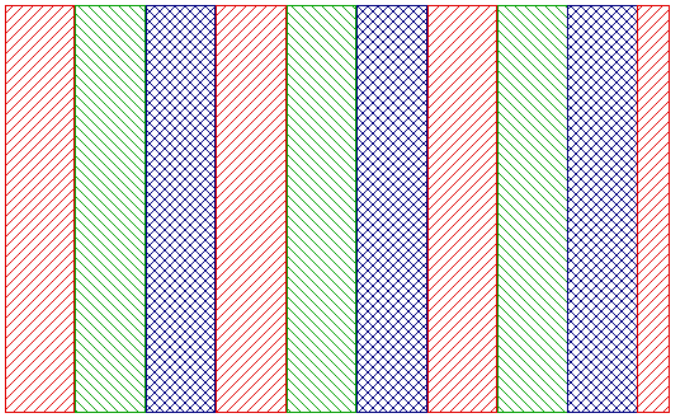
Two-dimensional Array B



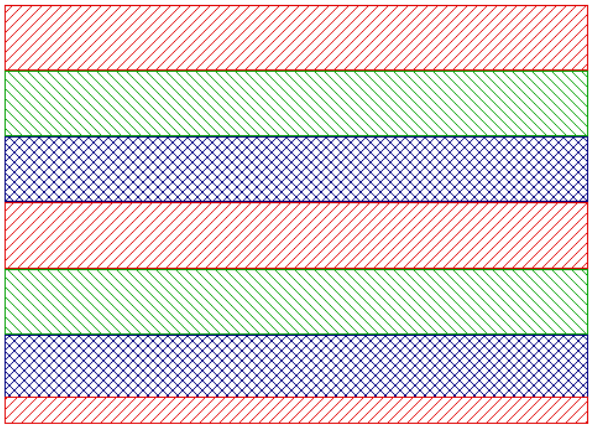
Block Block Distribution of B



Block Cyclic Distribution of B



Cyclic Block Distribution of B



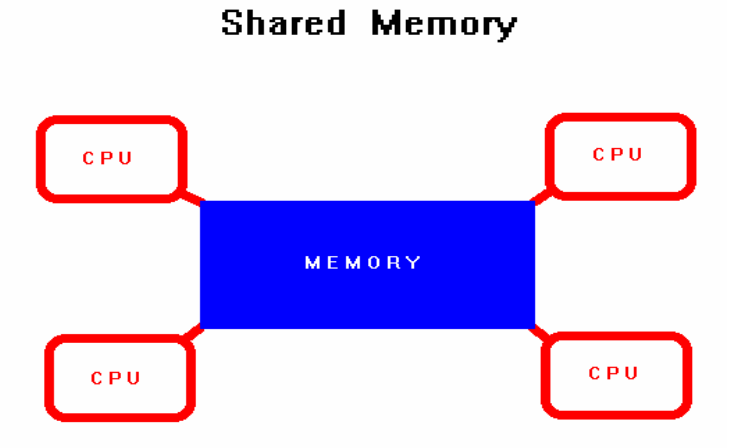
Distributed Communication Models

- Shared Memory
- Message Passing

Memory Architectures

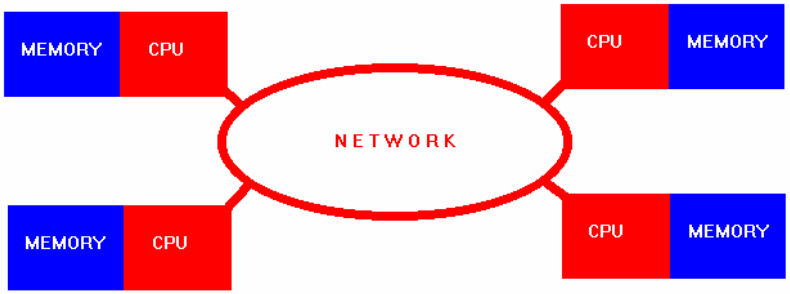
- The way processors communicate is dependent upon memory architecture, which, in turn, will affect how you write your parallel program
- The primary memory architectures are:
  - Shared Memory
  - Distributed Memory

Shared Memory



- Multiple processors operate independently but share the same memory resources
- Only one processor can access the shared memory location at a time
- Synchronization achieved by controlling tasks' reading from and writing to the shared memory
- Advantages
  - Easy for user to use efficiently
  - Data sharing among tasks is fast (speed of memory access)
- Disadvantages
  - Memory is bandwidth limited. Increase of processors without increase of bandwidth can cause severe bottlenecks
  - User is responsible for specifying synchronization, e.g., locks

Distributed Memory



Multiple processors operate independently but each has its own private memory

Data is shared across a communications network using message passing

User responsible for synchronization using message passing

Advantages

- Memory scalable to number of processors. Increase number of processors, size of memory and bandwidth increases.
- Each processor can rapidly access its own memory without interference

Disadvantages

- Difficult to map existing data structures to this memory organization
- User responsible for sending and receiving data among processors
- To minimize overhead and latency, data should be blocked up in large chunks and shipped before receiving node needs it

Memory / Processor Arrangements

- Distributed Memory
  - MPP - Massively Parallel Processor
- Shared Memory
  - SMP - Symmetric Multiprocessor
    - Identical processors
    - Equal access to memory
    - Sometimes called UMA - Uniform Memory Access
    - or CC-UMA - Cache Coherent UMA
    - Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update
  - NUMA - Non-Uniform Memory Access
    - Sometimes called CC-NUMA - Cache Coherent NUMA
    - Often made by linking two or more SMPs
    - One SMP can directly access memory of another SMP
    - Not all processors have equal access time to all memories
    - Memory access across link is slower

Memory / Processor Arrangements

- Combinations
  - Multiple SMPs connected by a network
    - Processors within an SMP communicate via memory
    - Requires message passing between SMPs
    - One SMP can't directly access the memory of another SMP
  - Multiple distributed memory processors connected to a larger shared memory
    - Small fast memory can be used for supplying data to processors and large slower memory can be used for a backfill to the smaller memories
    - Similar to register <= cache memory <= main memory hierarchy
    - Transfer from local memory to shared memory would be transparent to the user
    - Probable design of the future with several processors and their local memory surrounding a larger shared memory on a single board

Communication Models: Message Passing

- The message passing model is defined as:
- set of processes using only local memory
- processes communicate by sending and receiving messages
- data transfer requires cooperative operations to be performed by each process (a send operation must have a matching receive)
- Programming with message passing is done by linking with and making calls to libraries which manage the data exchange between processors. Message passing libraries are available for most modern programming languages.

Message Passing: Message Passing Interface (MPI)

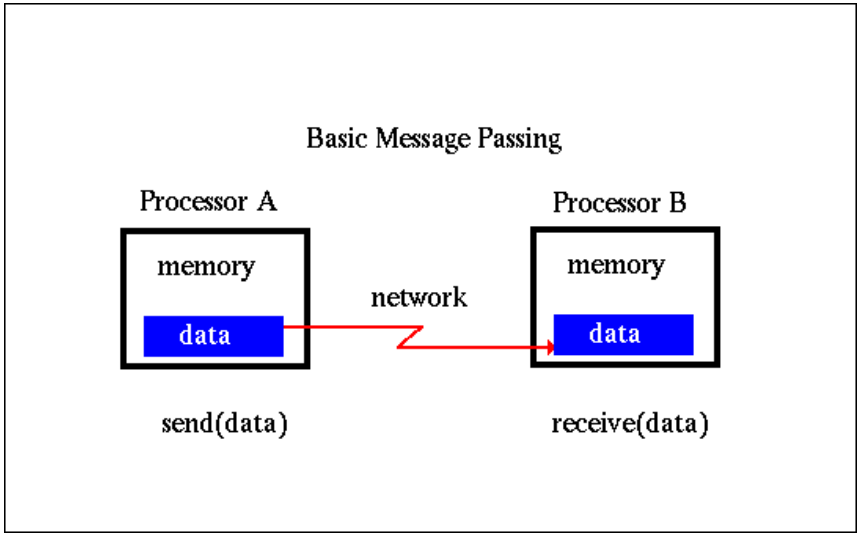
- standard portable message-passing library definition developed in 1993 by a group of parallel computer vendors, software writers, and application scientists.
- Available to both Fortran and C programs.
- Available on a wide variety of parallel machines.
- Target platform is a distributed memory system such as the SP.
- All inter-task communication is by message passing.
- All parallelism is explicit: the programmer is responsible for parallelism the program and implementing the MPI constructs.
- Programming model is SPMD (Single Program Multiple Data)

Communication Primitives

- Communications on distributed memory computers:
  - Point to Point
  - One to All Broadcast
  - All to All Broadcast
  - One to All Personalized
  - All to All Personalized
  - Shifts
  - Collective Computation
-

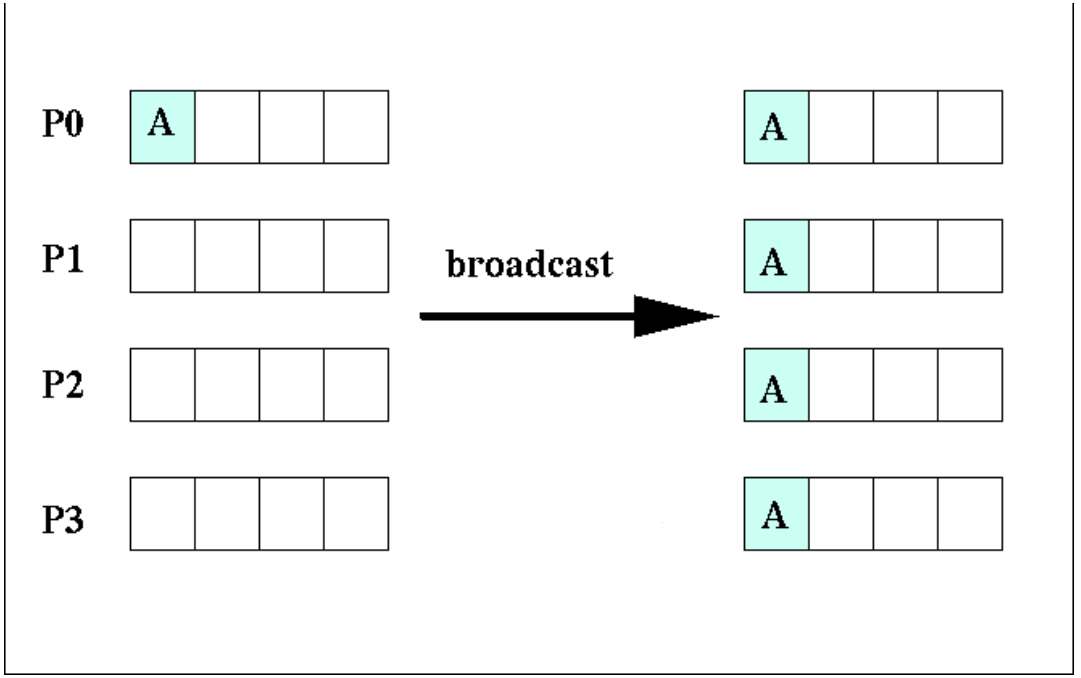
Point to Point

- The most basic method of communication between two processors is the point to point message. The originating processor "sends" the message to the destination processor. The destination processor then "receives" the message.
- The message commonly includes the information, the length of the message, the destination address and possibly a tag.
- Typical message passing libraries subdivide the basic sends and receives into two types:
- blocking - processing waits until message is transmitted
- nonblocking - processing continues even if message hasn't been transmitted yet



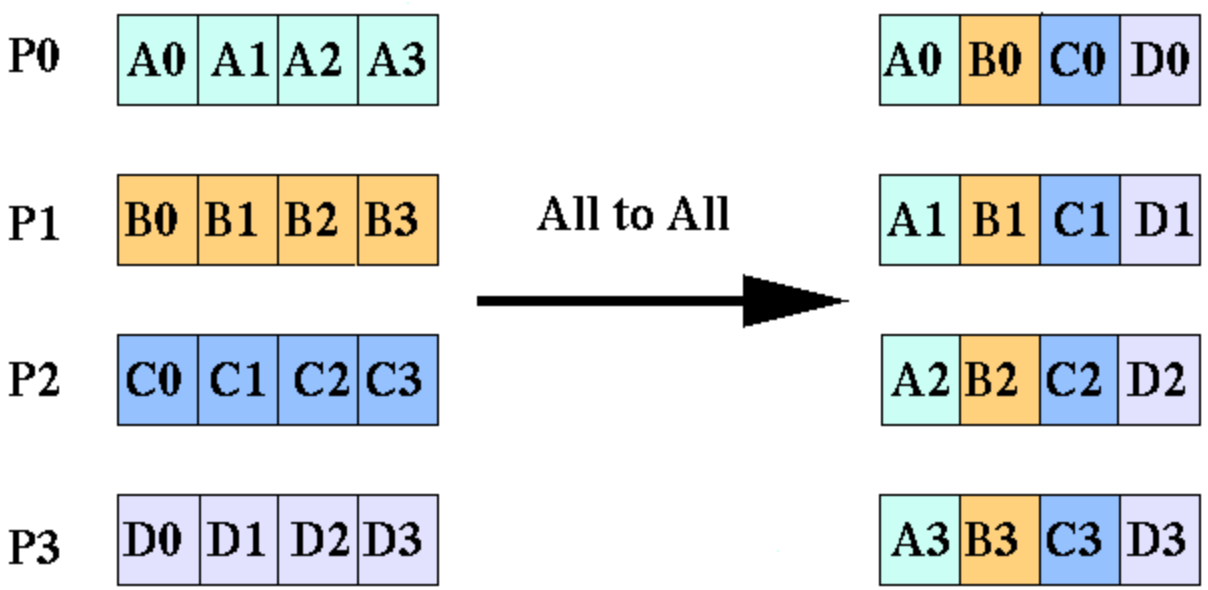
One to All Broadcast

- A node may have information which all the others require. A broadcast is a message sent to many other nodes.
- A One to All broadcast occurs when one processor sends the same information to many other nodes.



All to All Broadcast

- With an All to All broadcast each processor sends its unique information to all the other processors.



Shifts

- Shifts are permutations of information. Information is exchanged in one logical direction or the other. Each processor exchanges the same amount of information with its neighbor processor.
- There are two types of shifts:
  - Circular - Each processor exchanges information with its logical neighbor. When there is no longer a neighbor due to an edge of data the shift "wraps around" and takes the information from the opposite edge.
  - End Off Shift - When an edge occurs, the processor is padded with zero or a user defined value.

Collective Computation

- In collective computation (reductions), one member of the group collects data from the other members. Commonly a mathematical operation like
  - min, max, add, multiple etc.

Design and Performance Considerations

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) which can be parallelized:
  - **speedup = 1 / (1 – P)**
- If none of the code can be parallelized, p= 0 and the speedup = 1 (no speedup). If all of the code is parallelized, p = 1 and the speedup is infinite (in theory). If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:
  - **speedup = 1 / (P/N +S)**
- where P = parallel fraction, N = number of processors and S = serial fraction.

Scalability of the Problem

It soon becomes obvious that there are limits to the scalability of parallelism.
For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable):

	Speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Scalable Problems

- However, certain problems demonstrate increased performance by increasing the problem size. For example:
  - **2D Grid Calculations 85 seconds 85%**
  - **Serial fraction 15 seconds 15%**
- We can increase the problem size by halving both the grid points and the time step, which is directly proportional to the grid spacing. This results in four times the number of grid points (factor of two in each direction) and twice the number of time steps. The timings then look like:
  - **2D Grid Calculations 680 seconds 97.84%**
  - **Serial fraction 15 seconds 2.16%**
- Problems which increase the percentage of parallel time with their size are more "scalable" than problems with a fixed percentage of parallel time.

Communication Patterns and Bandwidth

- For some problems, increasing the number of processors will:
  - Decrease the execution time attributable to computation
  - But also, increase the execution time attributable to communication
- The time required for communication is dependent upon a given system's communication bandwidth parameters.
- For example, the time (t) required to send W words between any two processors is:

$$t = L + W/B$$

where L = latency and B = hardware bitstream rate in words per second.

- Latency can be thought of as the time required to send a zero byte message

Communication Patterns and Bandwidth

- Communication patterns also affect the computation to communication ratio.
- For example, gather-scatter communications between a single processor and N other processors will be impacted more by an increase in latency than N processors communicating only with nearest neighbors.

I/O Patterns

- I/O operations are generally regarded as inhibitors to parallelism
- Parallel I/O systems are as yet, largely undefined and not available
- In an environment where all processors see the same filespace, write operations will result in file overwriting
- Read operations will be affected by the fileserver's ability to handle multiple read requests at the same time
- I/O which must be conducted over the network (non-local) can cause severe bottlenecks

Improving I/O Performance

- Reduce overall I/O as much as possible
- Confine I/O to specific serial portions of the job
  - For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- Create unique filenames for each tasks' input/output file(s)
- For distributed memory systems with shared filespace, perform I/O in local, non-shared filespace
  - For example, each processor may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.

Principles of Parallel Algorithm Design

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.

Chapter Overview: Algorithms and Concurrency

- Introduction to Parallel Algorithms
  - Tasks and Decomposition
  - Processes and Mapping
  - Processes Versus Processors
- Decomposition Techniques
  - Recursive Decomposition
  - Recursive Decomposition
  - Exploratory Decomposition
  - Hybrid Decomposition
- Characteristics of Tasks and Interactions
  - Task Generation, Granularity, and Context
  - Characteristics of Task Interactions.

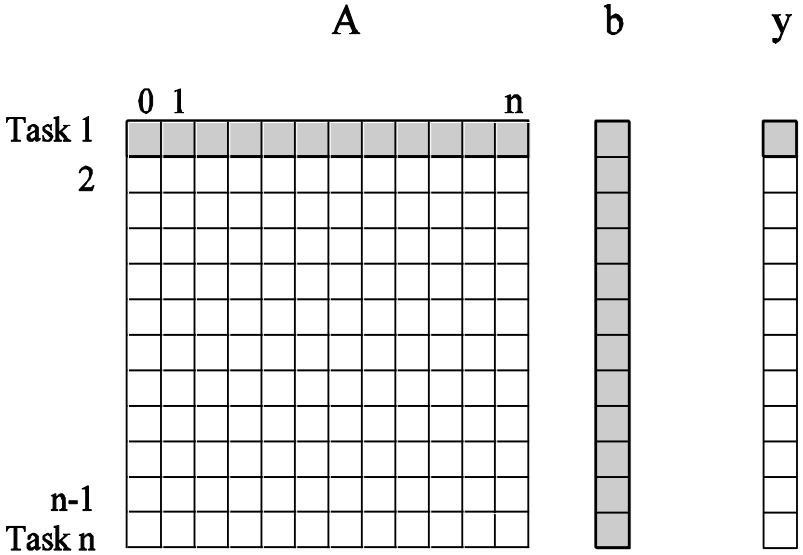
Chapter Overview: Concurrency and Mapping

- Mapping Techniques for Load Balancing
  - Static and Dynamic Mapping
- Methods for Minimizing Interaction Overheads
  - Maximizing Data Locality
  - Minimizing Contention and Hot-Spots
  - Overlapping Communication and Computations
  - Replication vs. Communication
  - Group Communications vs. Point-to-Point Communication
- Parallel Algorithm Design Models
  - Data-Parallel, Work-Pool, Task Graph, Master-Slave, Pipeline, and Hybrid Models

Preliminaries: Decomposition, Tasks, and Dependency Graphs

- 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 docomposed into tasks in many different ways.
- Tasks may be of same, different, or even interminate sizes.
- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next. Such a graph is called a *task dependency graph*.

Example: Multiplying a Dense Matrix with a Vector



Computation of each element of output vector ***y*** is independent of other elements. 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.

Observations: While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other. All tasks are of the same size in terms of number of operations. *Is this the maximum number of tasks we could decompose this problem into?*

Example: Database Query Processing

Consider the execution of the query:

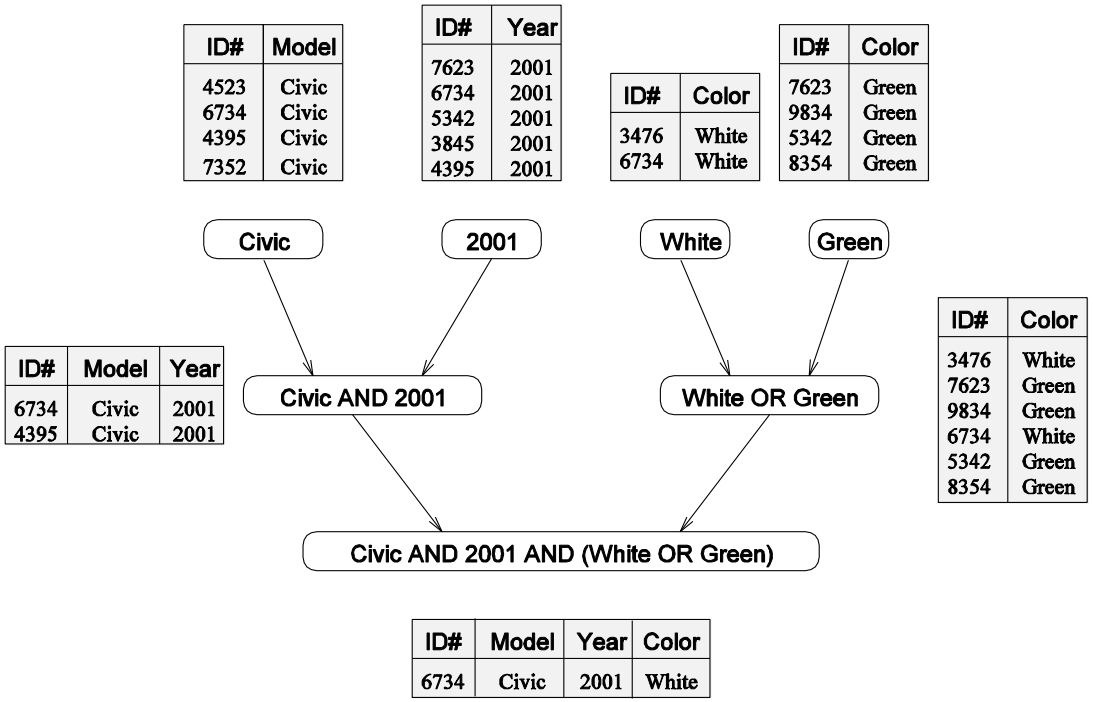
MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

on the following database:

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: Database Query Processing

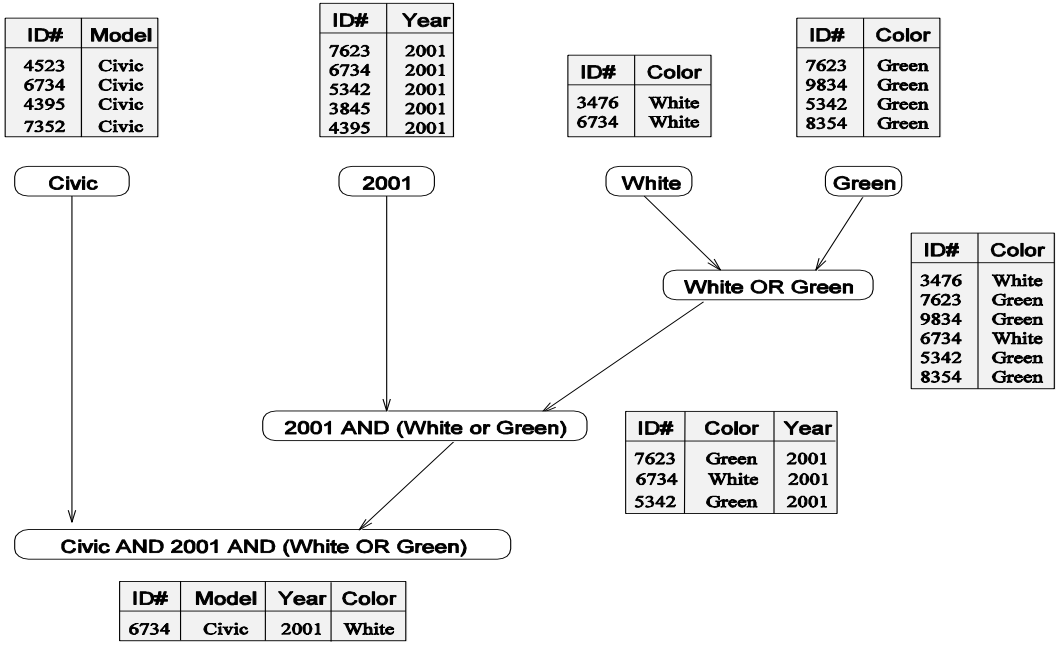
The execution of the query can be divided into subtasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



Decomposing the given query into a number of tasks. Edges in this graph denote that the output of one task is needed to accomplish the next.

Example: Database Query Processing

Note that the same problem can be decomposed into subtasks in other ways as well.

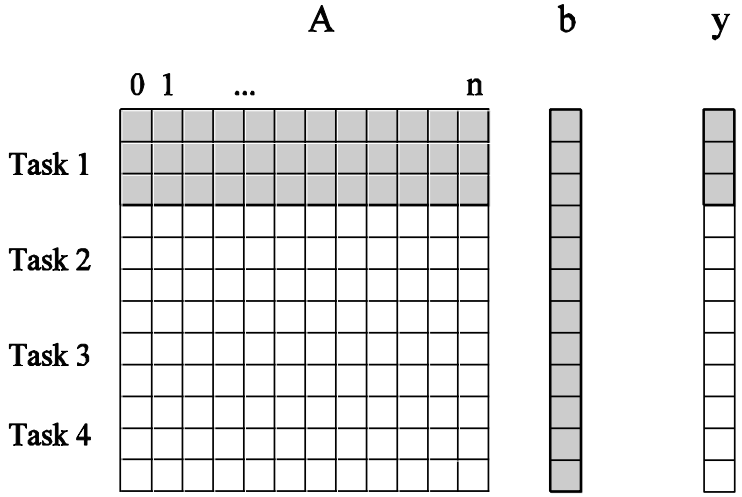


An alternate decomposition of the given problem into subtasks, along with their data dependencies.

Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

Granularity of Task Decompositions

- 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 and that into a small number of tasks results in a coarse grained decomposition.



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

Degree of Concurrency

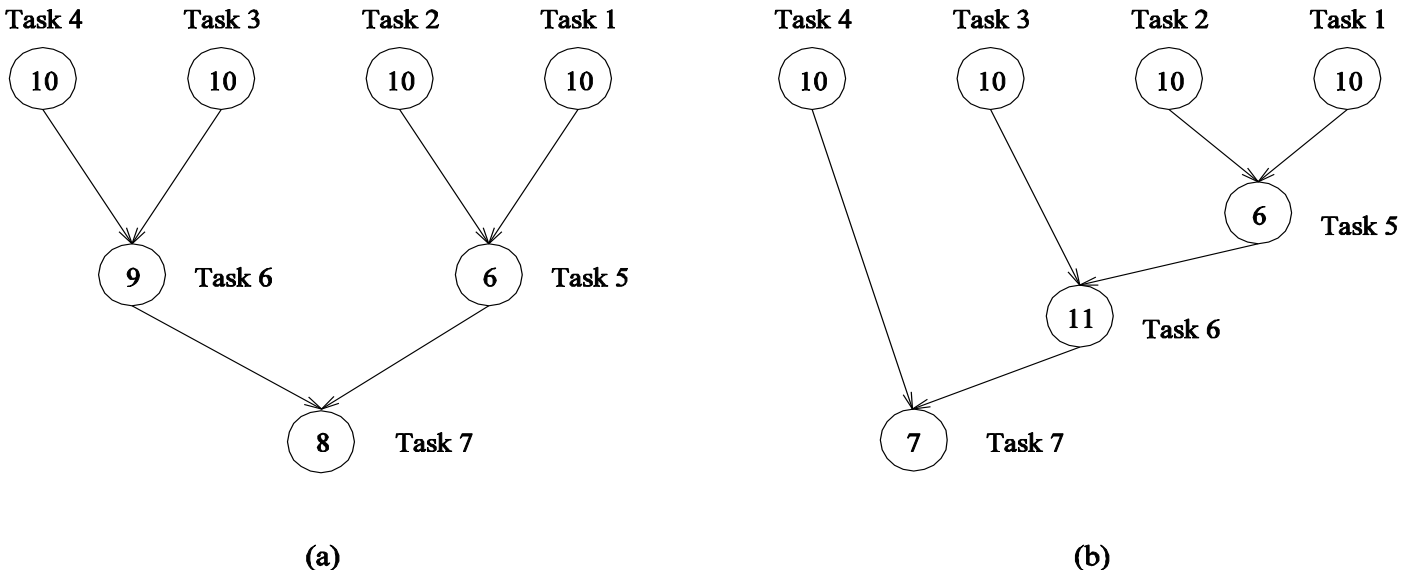
- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution. *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, what is the average degree of concurrency in each decomposition?*
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

Critical Path Length

- 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.
- The length of the longest path in a task dependency graph is called the critical path length.

Critical Path Length

Consider the task dependency graphs of the two database query decompositions:



What are the critical path lengths for the two task dependency graphs? 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?

Limits on Parallel Performance

- 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.*
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.

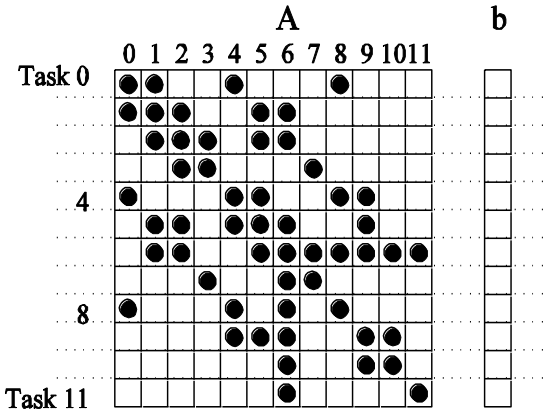
Task Interaction Graphs

- 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, 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*.
- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.

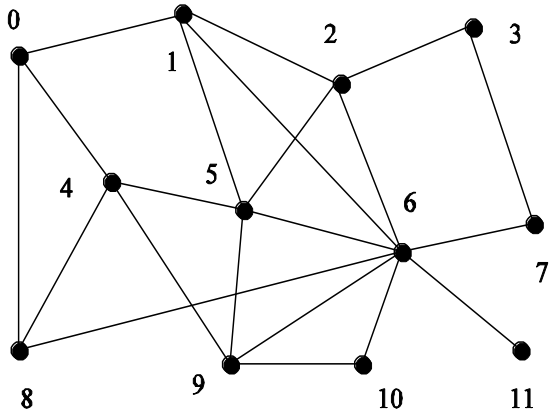
Task Interaction Graphs: An Example

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

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



(a)



(b)

Task Interaction Graphs, Granularity, and Communication

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

Example: Consider the sparse matrix-vector product example from previous foil. Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.

Viewing node 0 as an independent task involves a useful computation of one time unit and overhead (communication) of three time units.

Now, if we consider nodes 0, 4, and 5 as one task, then the task has useful computation totaling to three time units and communication corresponding to four time units (four edges). Clearly, this is a more favorable ratio than the former case.

Processes and Mapping

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.

- For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

Note: We refer to the mapping as being from tasks to processes, as opposed to processors. This is because typical programming APIs, as we shall see, do not allow easy binding of tasks to physical processors. Rather, we aggregate tasks into processes and rely on the system to map these processes to physical processors. We use processes, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.

Processes and Mapping

- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.
- Mappings are determined by both the task dependency and task interaction graphs.
- Task dependency graphs can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).
- Task interaction graphs can be used to make sure that processes need minimum interaction with other processes (minimum communication).

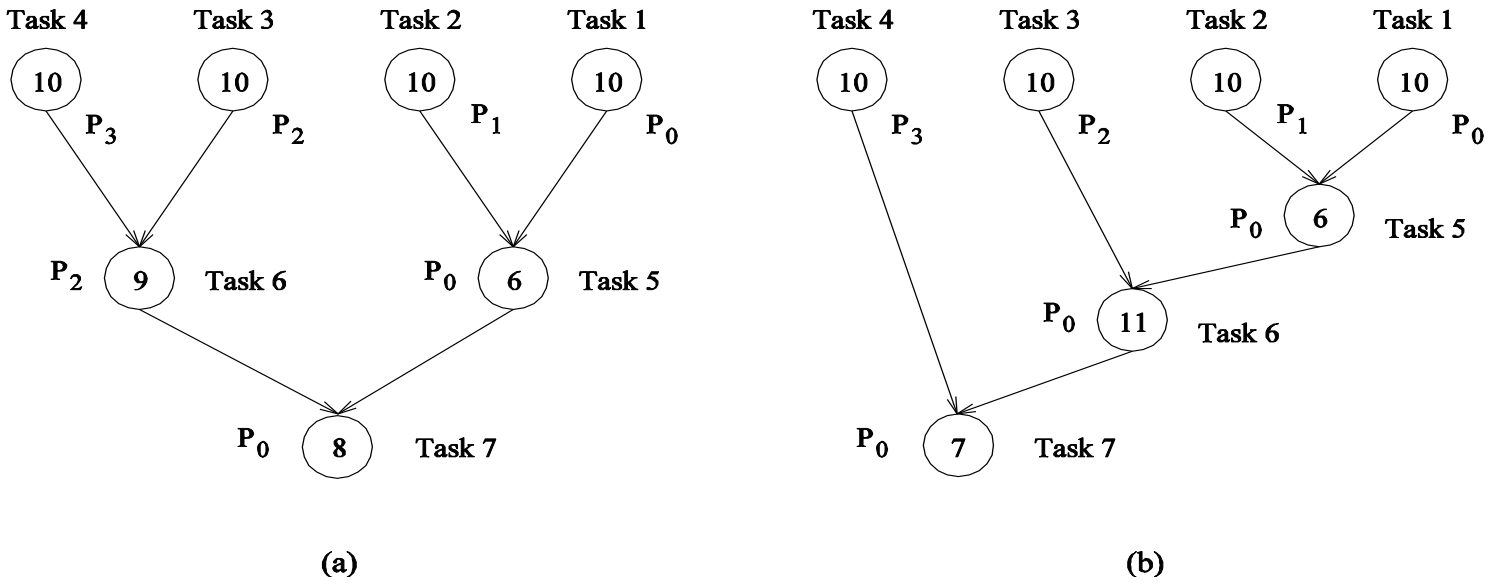
Processes and Mapping

An appropriate mapping must minimize parallel execution time by:

- Mapping independent tasks to different processes.
- Assigning tasks on critical path to processes as soon as they become available.
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

Note: These criteria often conflict eith each other. For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all! Can you think of other such conflicting cases?

Processes and Mapping: Example



Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

Decomposition Techniques

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

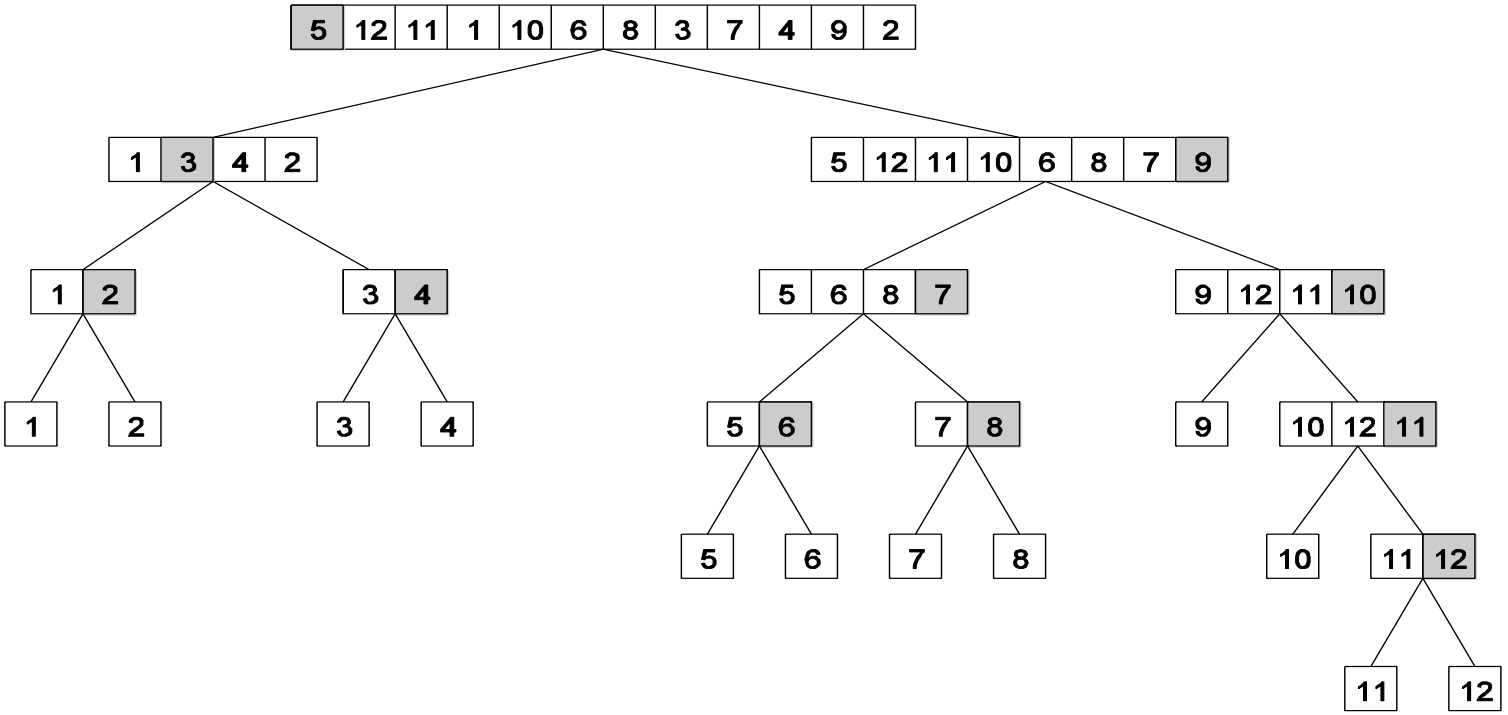
- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.

Recursive Decomposition: Example

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.



In this example, once the list has been partitioned around the pivot, each sublist can be processed concurrently (i.e., each sublist represents an independent subtask). This can be repeated recursively.

Recursive Decomposition: Example

The problem of finding the minimum number in a given list (or indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm. The following algorithm illustrates this.

We first start with a simple serial loop for computing the minimum entry in a given list:

```
1. procedure SERIAL_MIN (A, n)
2. begin
3. min = A[0];
4. for i := 1 to n − 1 do
5.                 if (A[i] < min) min := A[i];
6. endfor;
7. return min;
8. end SERIAL_MIN
```

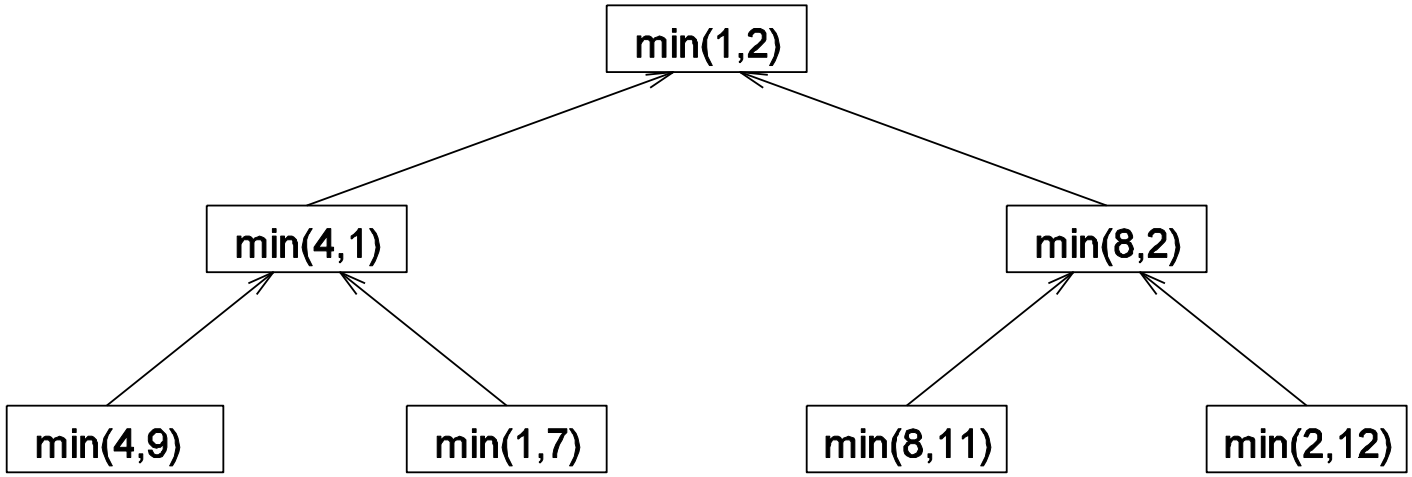
Recursive Decomposition: Example

We can rewrite the loop as follows:

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.     min := A [0]  ;
5. else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.     if (lmin  < rmin) then
9.                 min := lmin;
10.    else
11.                min := rmin;
12.    endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

Recursive Decomposition: Example

The code in the previous foil can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of finding the minimum number in the set {4, 9, 1, 7, 8, 11, 2, 12}. The task dependency graph associated with this computation is as follows:



Data Decomposition

- Identify the data on which computations are performed.
- Partition this data across various tasks.
- This partitioning induces a decomposition of the problem.
- Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.

Data Decomposition: Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input).
- A partition of the output across tasks decomposes the problem naturally.

Output Data Decomposition: Example

Consider the problem of multiplying two ***n*** x ***n*** matrices ***A*** and ***B*** to yield matrix ***C***. The output matrix ***C*** can be partitioned into four tasks as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Output Data Decomposition: Example

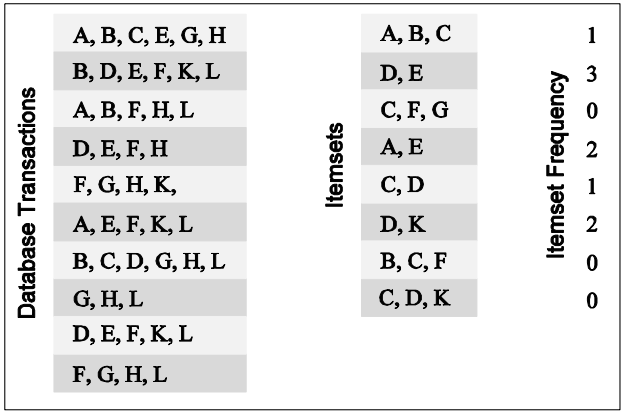
A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previus foil, with identical output data distribution, we can derive the following two (other) decompositions:

Decomposition I	**Decomposition II**
Task 1: $C_{1,1} = A_{1,1} B_{1,1}$	Task 1: $C_{1,1} = A_{1,1} B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Task 3: $C_{1,2} = A_{1,1} B_{1,2}$	Task 3: $C_{1,2} = A_{1,2} B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Task 5: $C_{2,1} = A_{2,1} B_{1,1}$	Task 5: $C_{2,1} = A_{2,2} B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Task 7: $C_{2,2} = A_{2,1} B_{1,2}$	Task 7: $C_{2,2} = A_{2,1} B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

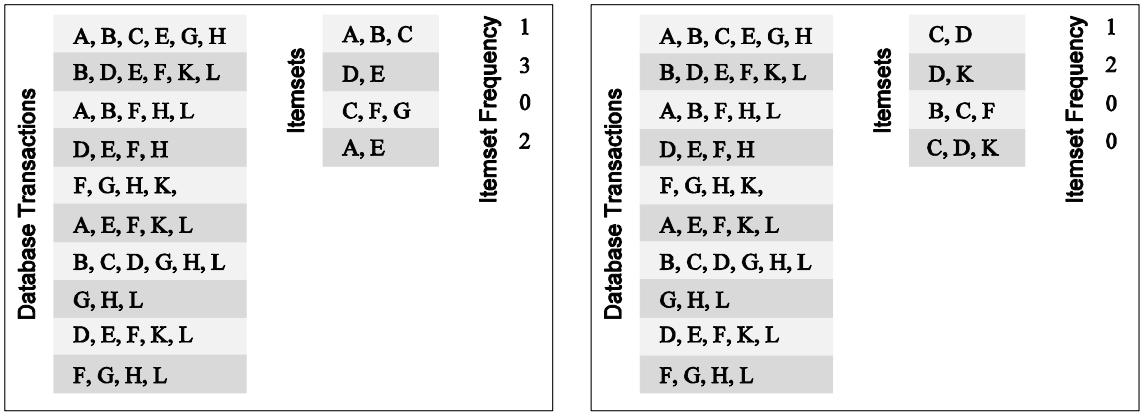
Output Data Decomposition: Example

Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.

(a) Transactions (input), itemsets (input), and frequencies (output)



(b) Partitioning the frequencies (and itemsets) among the tasks



Output Data Decomposition: Example

From the previous example, the following observations can be made:

- If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

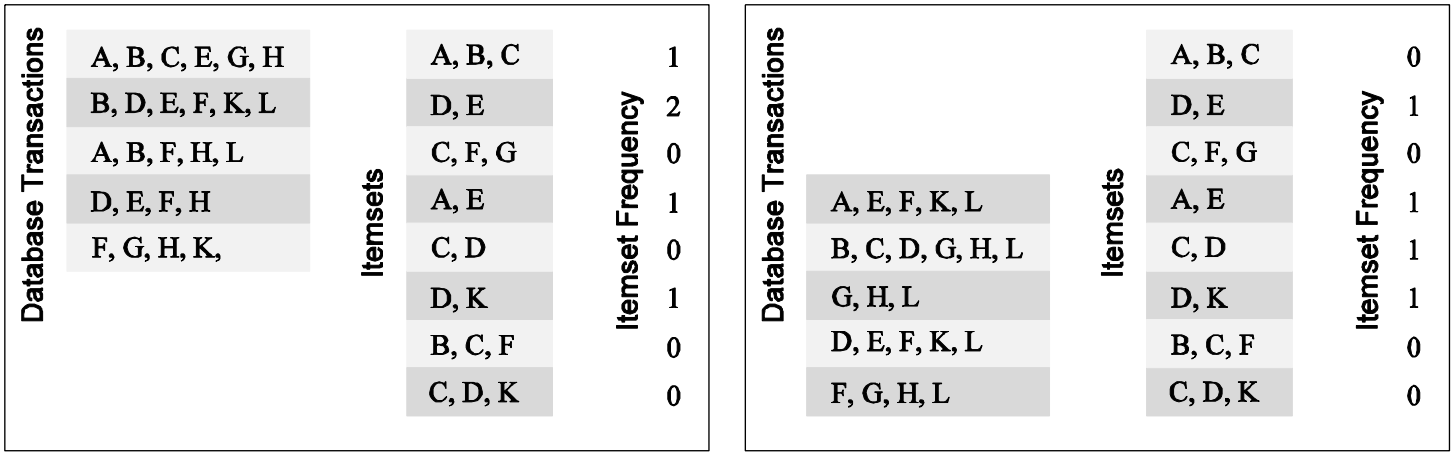
Input Data Partitioning

- Generally applicable if each output can be naturally computed as a function of the input.
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.

Input Data Partitioning: Example

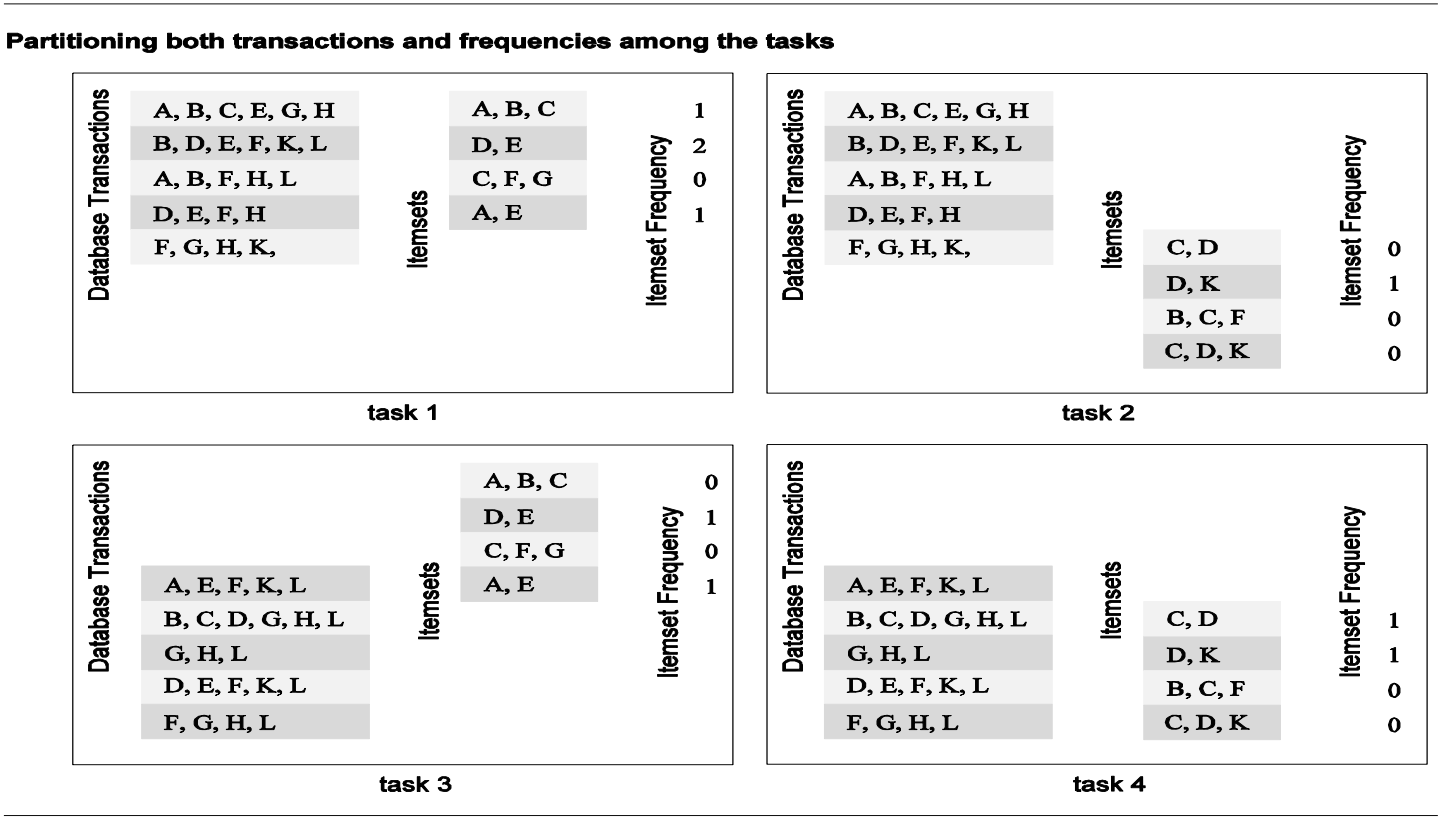
In the database counting example, the input (i.e., the transaction set) can be partitioned. This induces a task decomposition in which each task generates partial counts for all itemsets. These are combined subsequently for aggregate counts.

Partitioning the transactions among the tasks



Partitioning Input *and* Output Data

Often input and output data decomposition can be combined for a higher degree of concurrency. For the itemset counting example, the transaction set (input) and itemset counts (output) can both be decomposed as follows:

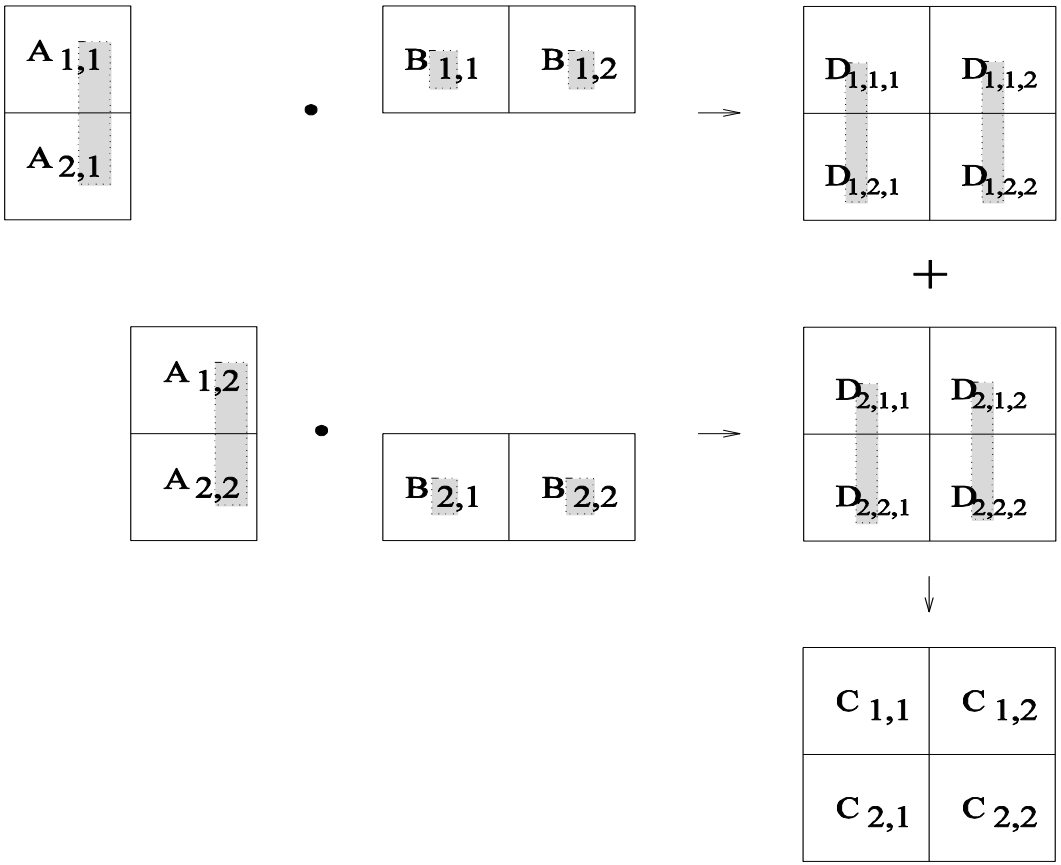


Intermediate Data Partitioning

- Computation can often be viewed as a sequence of transformation from the input to the output data.
- In these cases, it is often beneficial to use one of the intermediate stages as a basis for decomposition.

Intermediate Data Partitioning: Example

Let us revisit the example of dense matrix multiplication. We first show how we can visualize this computation in terms of intermediate matrices ***D***.



Intermediate Data Partitioning: Example

A decomposition of intermediate data structure leads to the following decomposition into 8 + 4 tasks:

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 01: $\boldsymbol{D}_{1,1,1} = \boldsymbol{A}_{1,1}\boldsymbol{B}_{1,1}$

Task 02: $\boldsymbol{D}_{2,1,1} = \boldsymbol{A}_{1,2}\boldsymbol{B}_{2,1}$

Task 03: $\boldsymbol{D}_{1,1,2} = \boldsymbol{A}_{1,1}\boldsymbol{B}_{1,2}$

Task 04: $\boldsymbol{D}_{2,1,2} = \boldsymbol{A}_{1,2}\boldsymbol{B}_{2,2}$

Task 05: $\boldsymbol{D}_{1,2,1} = \boldsymbol{A}_{2,1}\boldsymbol{B}_{1,1}$

Task 06: $\boldsymbol{D}_{2,2,1} = \boldsymbol{A}_{2,2}\boldsymbol{B}_{2,1}$

Task 07: $\boldsymbol{D}_{1,2,2} = \boldsymbol{A}_{2,1}\boldsymbol{B}_{1,2}$

Task 08: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\boldsymbol{B}_{2,2}$

Task 09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

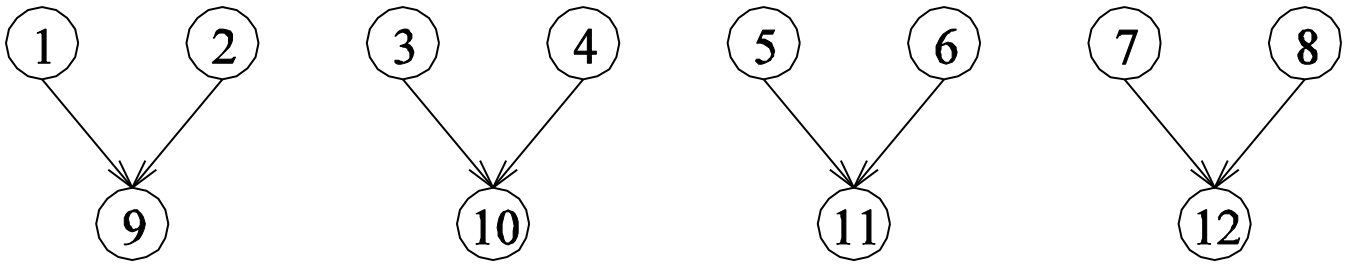
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

Task 11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

Task 12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

Intermediate Data Partitioning: Example

The task dependency graph for the decomposition (shown in previous foil) into 12 tasks is as follows:



The Owner Computes Rule

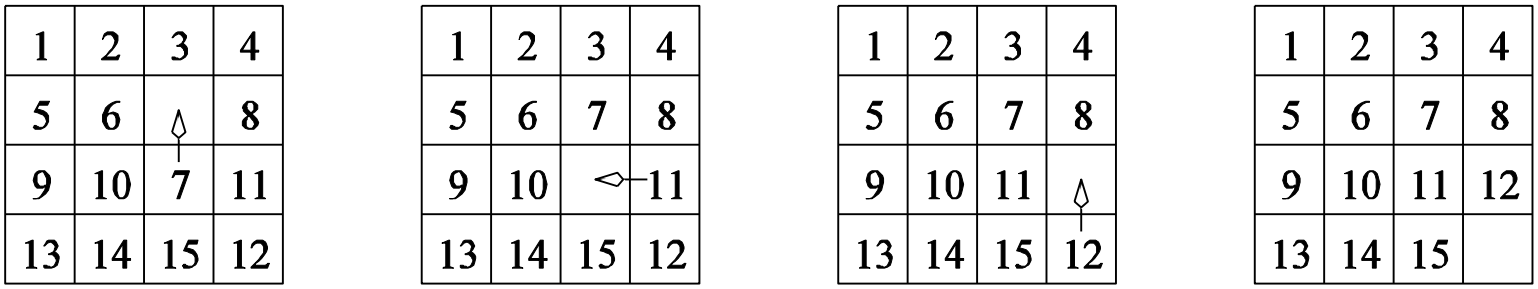
- The *Owner Computes Rule* generally states that the process assined a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule imples that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.

Exploratory Decomposition

- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.
- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

Exploratory Decomposition: Example

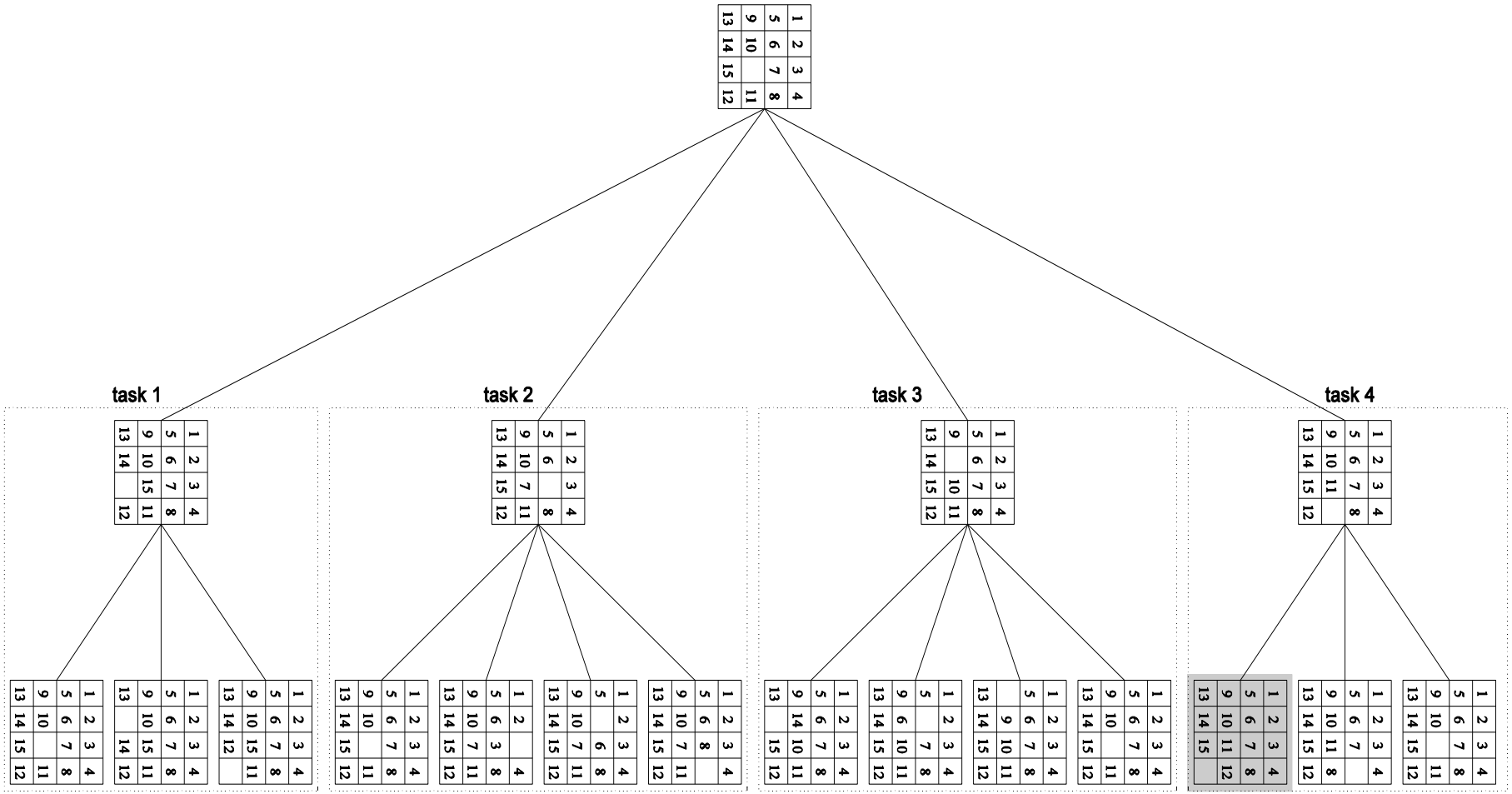
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).



Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.

Exploratory Decomposition: Example

The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



Speculative Decomposition

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications: conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies, and, optimistic approaches, which schedule tasks even when they may potentially be erroneous.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

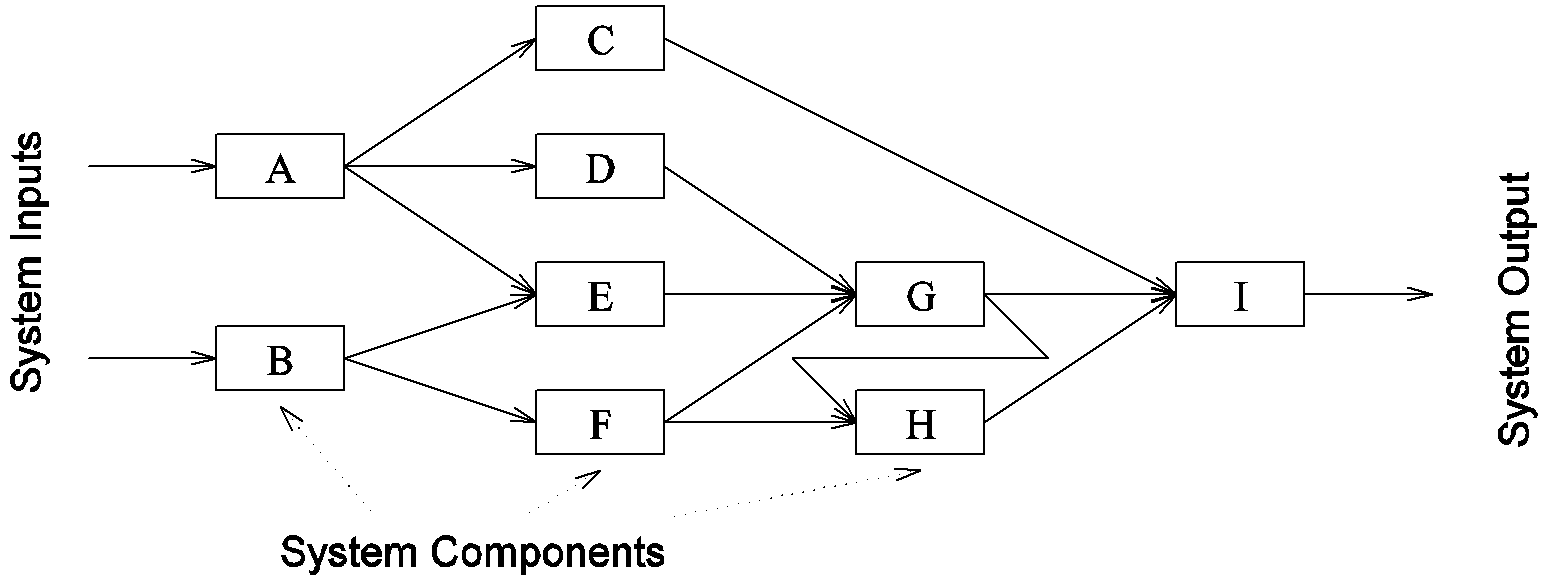
Speculative Decomposition: Example

A classic example of speculative decomposition is in discrete event simulation.

- The central data structure in a discrete event simulation is a time-ordered event list.
- Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- Consider your day today as a discrete event system - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.
- Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.
- Therefore, an optimistic scheduling of other events will have to be rolled back.

Speculative Decomposition: Example

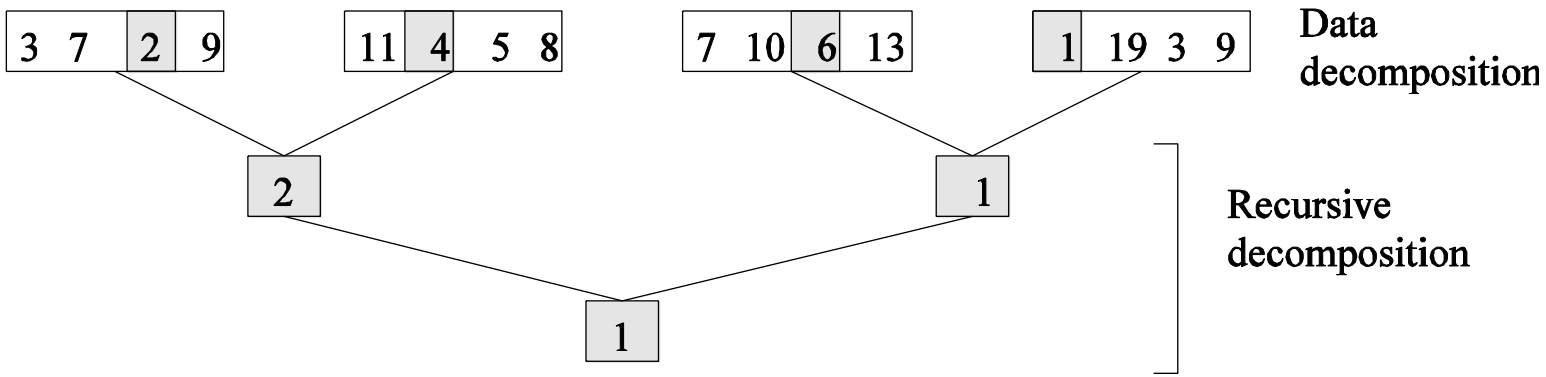
Another example is the simulation of a network of nodes (for instance, an assembly line or a computer network through which packets pass). The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue sizes, etc.).



Hybrid Decompositions

Often, a mix of decomposition techniques is necessary for decomposing a problem. Consider the following examples:

- In quicksort, recursive decomposition alone limits concurrency (Why?). A mix of data and recursive decompositions is more desirable.
- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.



Characteristics of Tasks

Once a problem has been decomposed into independent tasks, the characteristics of these tasks critically impact choice and performance of parallel algorithms. Relevant task characteristics include:

- Task generation.
- Task sizes.
- Size of data associated with tasks.

Task Generation

- Static task generation: Concurrent tasks can be identified a-priori. Typical matrix operations, graph algorithms, image processing applications, and other regularly structured problems fall in this class. These can typically be decomposed using data or recursive decomposition techniques.
- Dynamic task generation: Tasks are generated as we perform computation. A classic example of this is in game playing - each 15 puzzle board is generated from the previous one. These applications are typically decomposed using exploratory or speculative decompositions.

Task Sizes

- Task sizes may be uniform (i.e., all tasks are the same size) or non-uniform.
- Non-uniform task sizes may be such that they can be determined (or estimated) a-priori or not.
- Examples in this class include discrete optimization problems, in which it is difficult to estimate the effective size of a state space.

Size of Data Associated with Tasks

- The size of data associated with a task may be small or large when viewed in the context of the size of the task.
- A small context of a task implies that an algorithm can easily communicate this task to other processes dynamically (e.g., the 15 puzzle).
- A large context ties the task to a process, or alternately, an algorithm may attempt to reconstruct the context at another processes as opposed to communicating the context of the task (e.g., 0/1 integer programming).

Characteristics of Task Interactions

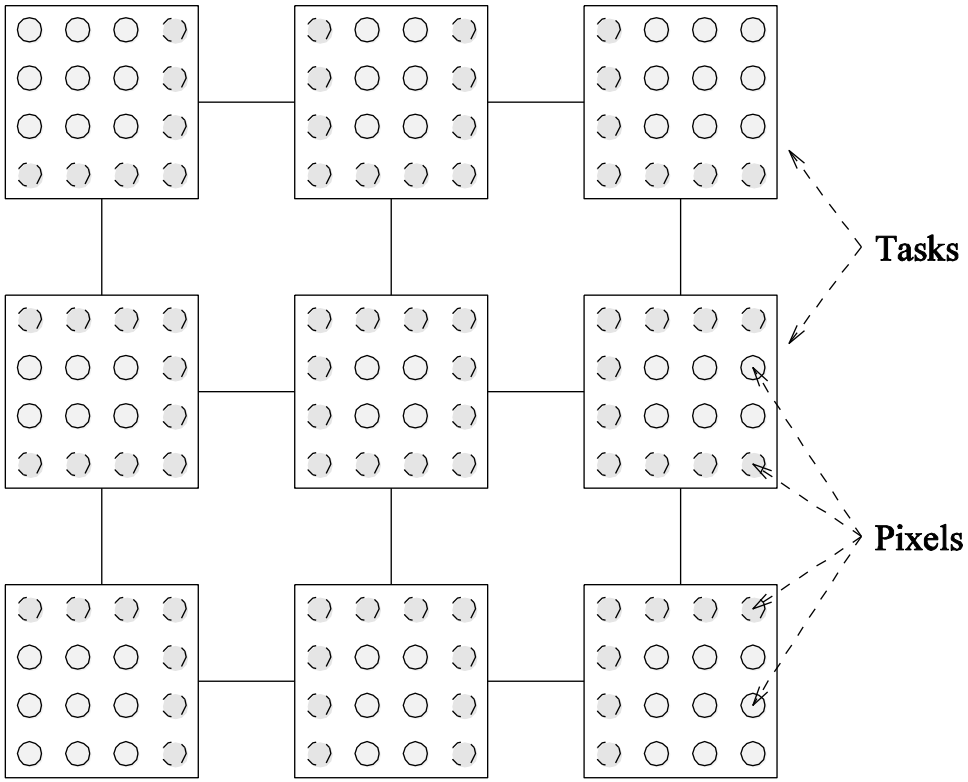
- Tasks may communicate with each other in various ways. The associated dichotomy is:
- Static interactions: The tasks and their interactions are known a-priori. These are relatively simpler to code into programs.
- Dynamic interactions: The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especitally, as we shall see, using message passing APIs.

Characteristics of Task Interactions

- Regular interactions: There is a definite pattern (in the graph sense) to the interactions. These patterns can be exploited for efficient implementation.
- Irregular interactions: Interactions lack well-defined topologies.

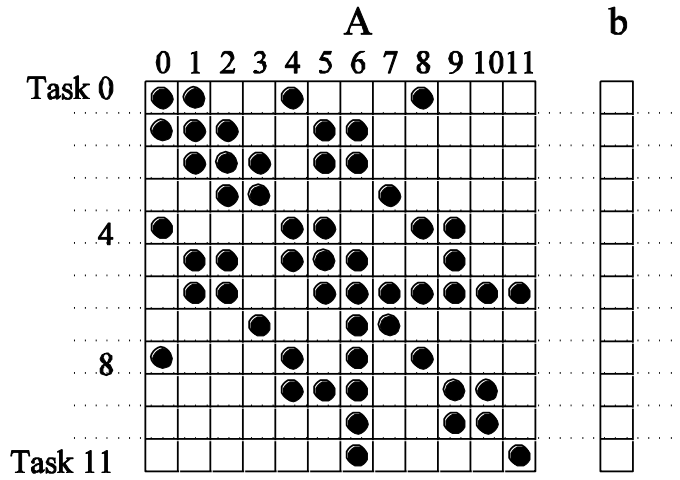
Characteristics of Task Interactions: Example

A simple example of a regular static interaction pattern is in image dithering. The underlying communication pattern is a structured (2-D mesh) one as shown here:

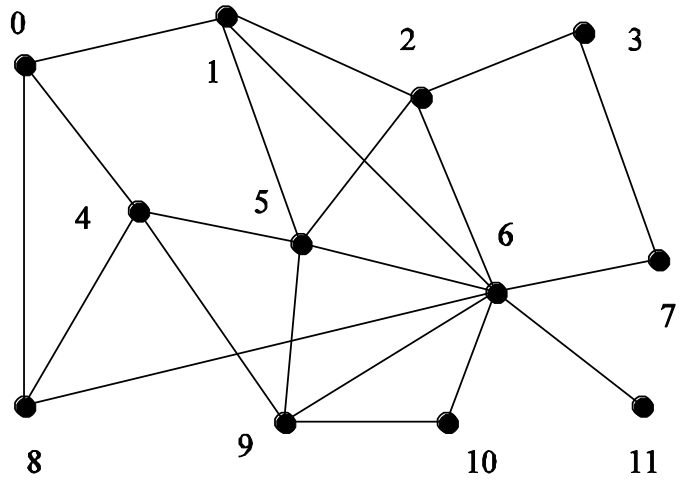


Characteristics of Task Interactions: Example

The multiplication of a sparse matrix with a vector is a good example of a static irregular interaction pattern. Here is an example of a sparse matrix and its associated interaction pattern.



(a)



(b)

Characteristics of Task Interactions

- Interactions may be read-only or read-write.
- In read-only interactions, tasks just read data items associated with other tasks.
- In read-write interactions tasks read, as well as modily data items associated with other tasks.
- In general, read-write interactions are harder to code, since they require additional synchronization primitives.

Characteristics of Task Interactions

- Interactions may be one-way or two-way.
- A one-way interaction can be initiated and accomplished by one of the two interacting tasks.
- A two-way interaction requires participation from both tasks involved in an interaction.
- One way interactions are somewhat harder to code in message passing APIs.

Mapping Techniques

- Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).
- Mappings must minimize overheads.
- Primary overheads are communication and idling.
- Minimizing these overheads often represents contradicting objectives.
- Assigning all work to one processor trivially minimizes communication at the expense of significant idling.

Parallel Algorithm Models

An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- Data Parallel Model: Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.
- Task Graph Model: Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.

Parallel Algorithm Models (continued)

- Master-Slave Model: One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- Pipeline / Producer-Comsumer Model: A stream of data is passed through a succession of processes, each of which perform some task on it.
- Hybrid Models: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

Main Considerations to Develop Efficient Parallel/Distributed Programs

- identification of parallelism
- program decomposition
- load balancing (static vs. dynamic)
- task granularity in the case of dynamic load balancing
- communication patterns - overlapping communication and computation