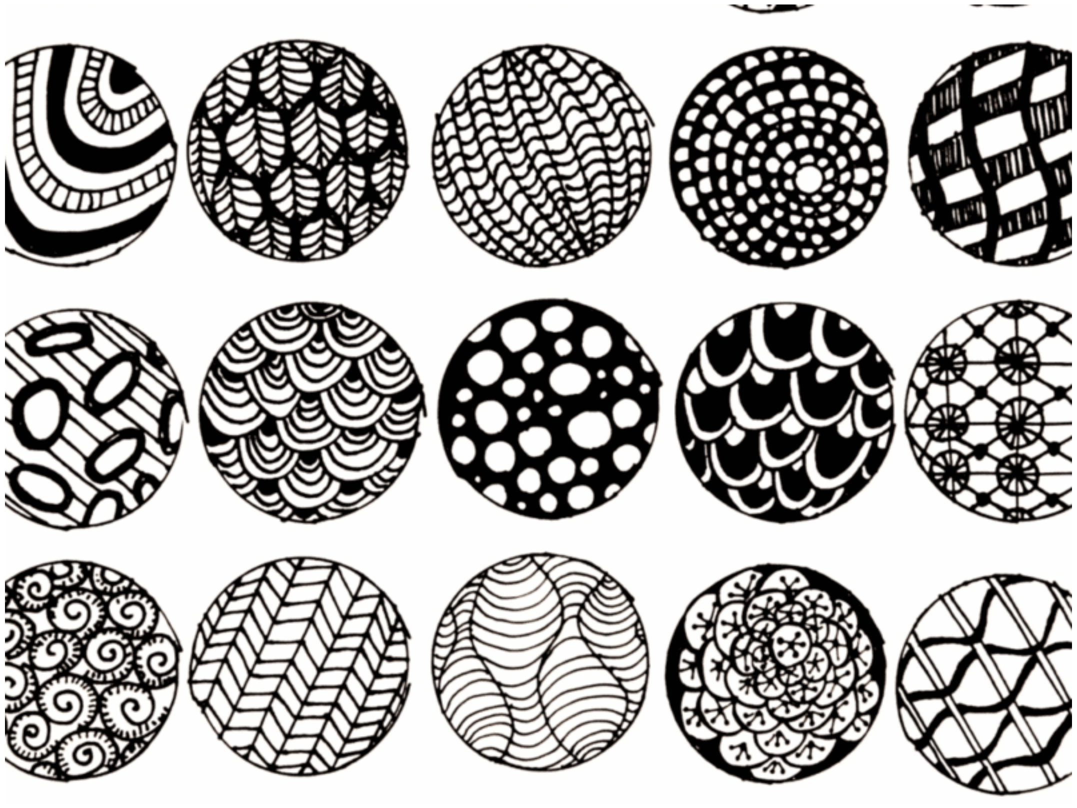
Lecture 12

Parallel Patterns

Pattern Languages

Skeletons

PATTERNS



Parallel Design Patterns

Software engineering setting:

- A **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
  - A design pattern isn't considered a finished design that can be transformed directly into code.
  - It is a description or template for how to solve a problem that can be used in many different situations.

Parallel setting:

Generic patterns of computation and interaction.

- different classifications
- different names

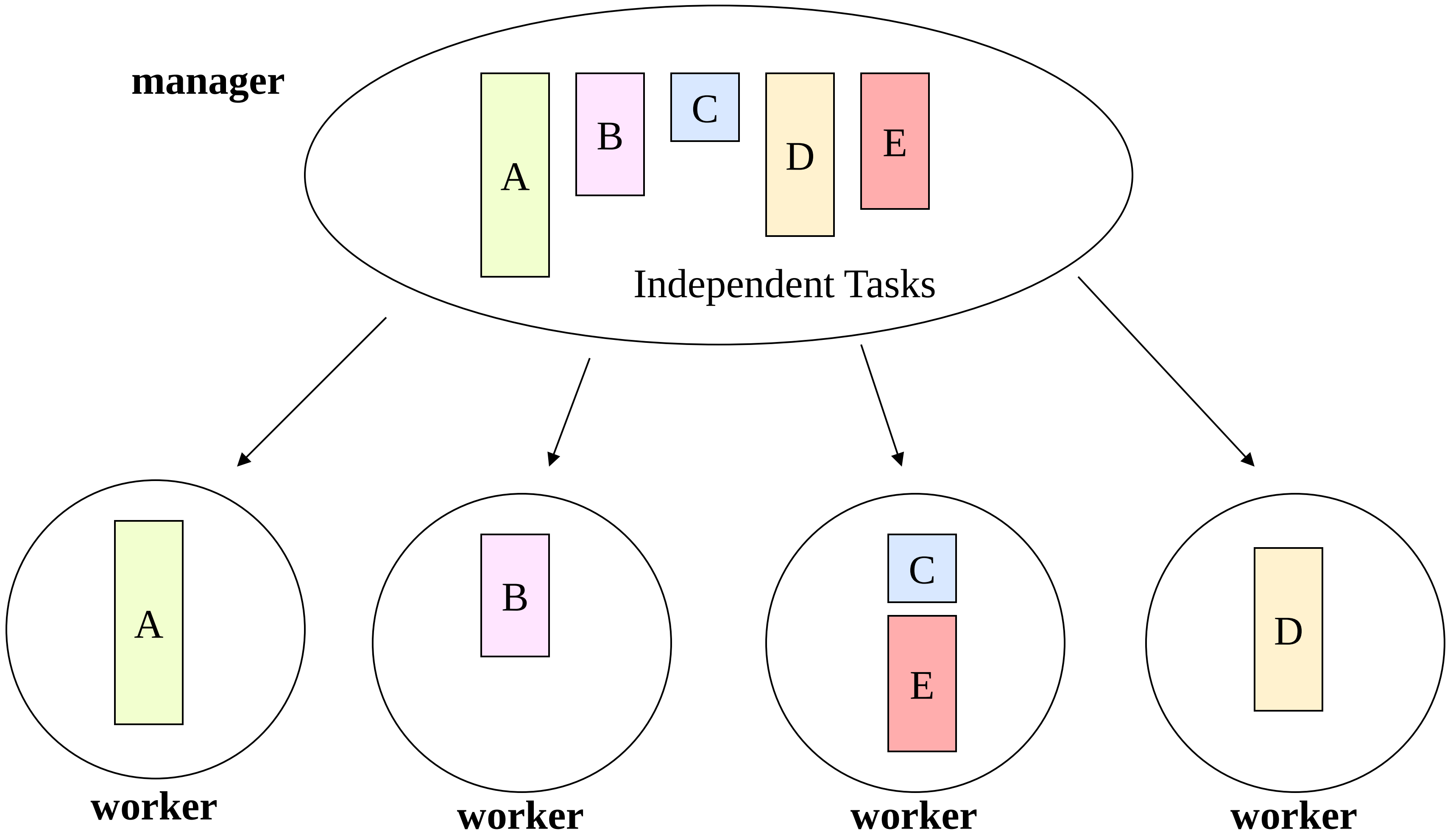
Common Parallel Patterns

- Embarassingly Parallel
- Replicable
- Repository
- Divide&Conquer
- Pipeline
- Recursive Data
- Geometric
- IrregularMesh

Parallel Programming Patterns
Eun-Gyu Kim
2004

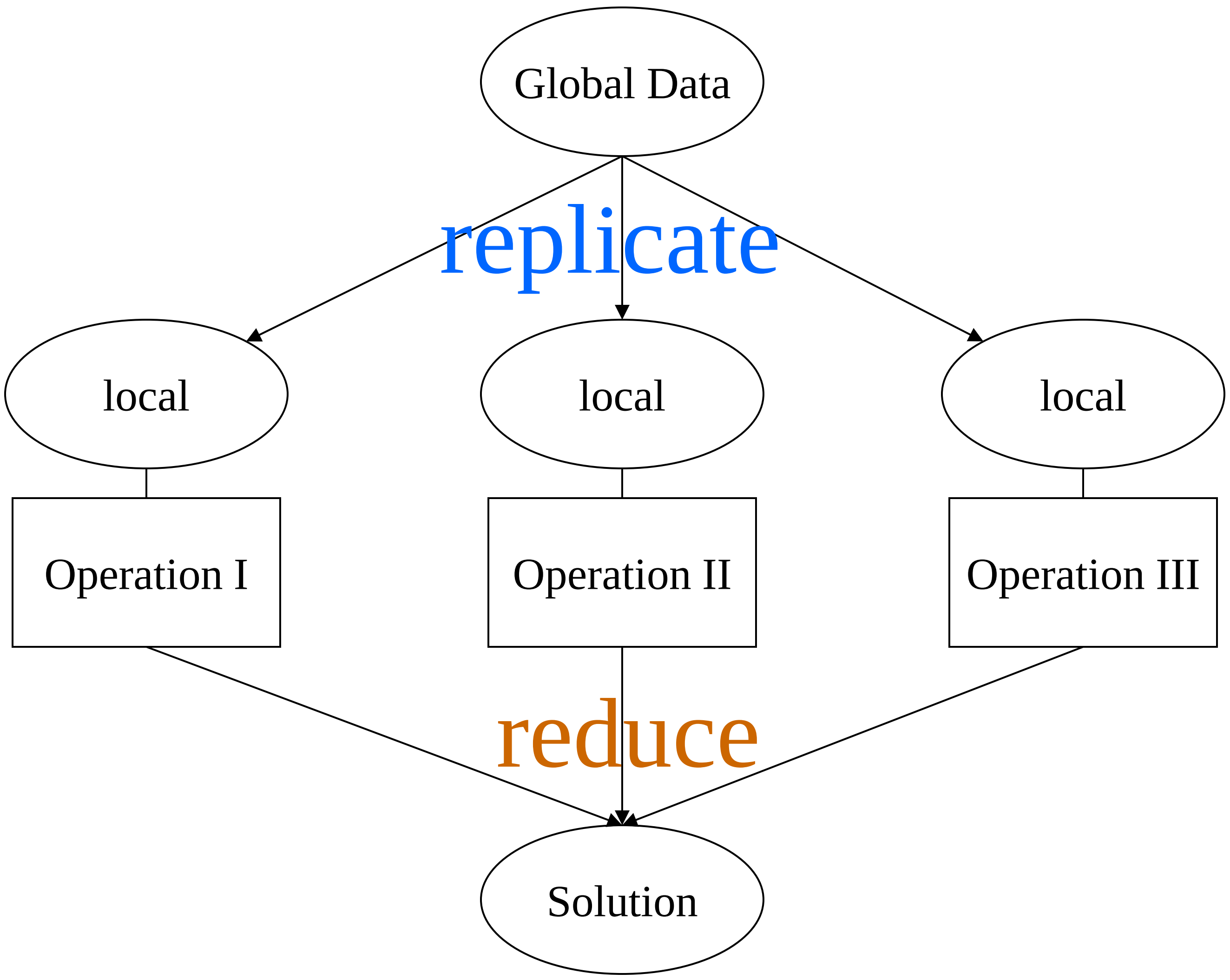
Embarassingly Parallel

Problem: Need to perform same operations to tasks that are independent



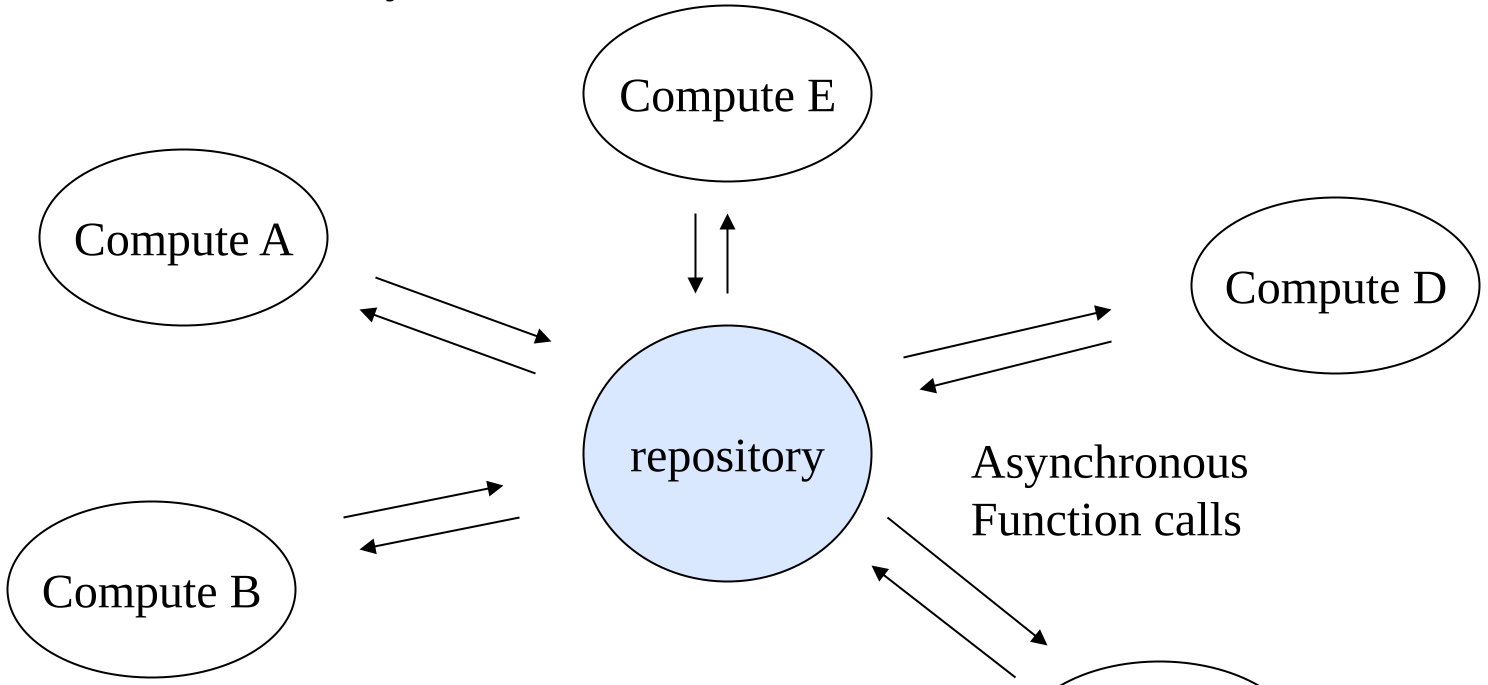
Replicable

Sets of operations need to be performed using global data structure, causing dependency.



Repository

Independent computations needs to applied to centralized data structure in non-deterministic way.

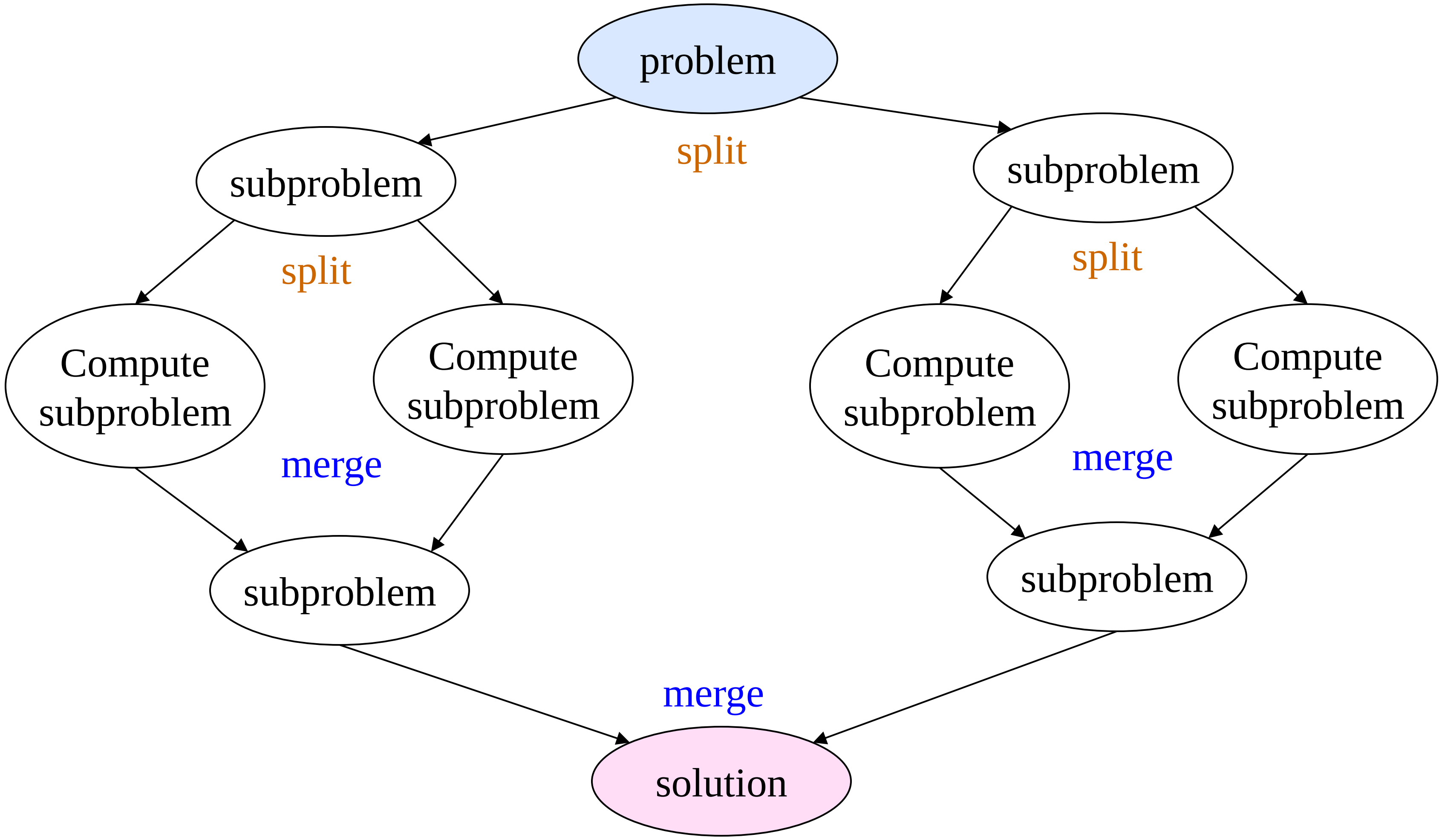


Asynchronous
Function calls

Compute object cannot access same element simultaneously. (Repository controls access)

Divide & Conquer

A problem is structured to be solved in sub-problems independently, and merging them later.



* Split level needs to be adjusted appropriately.

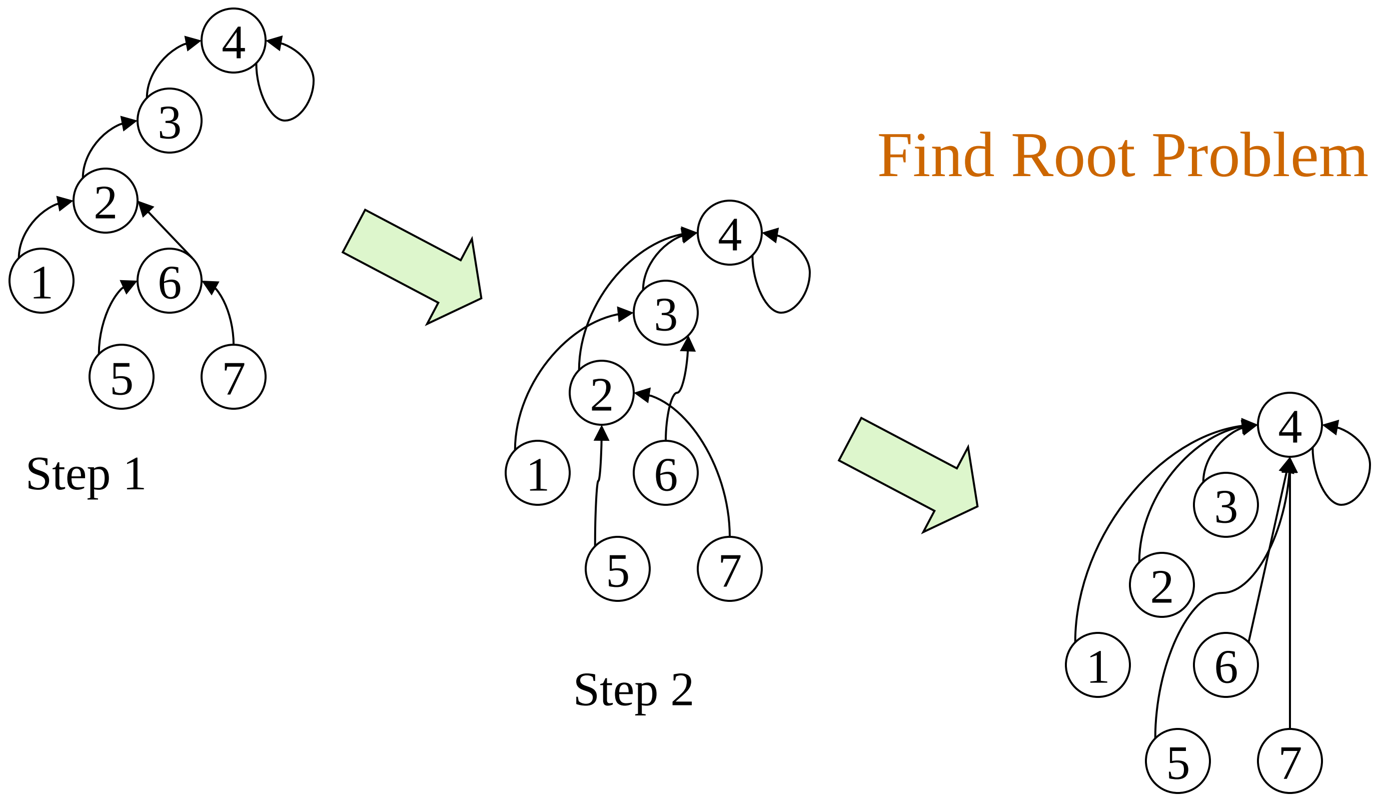
Example: Merge-sort

```
PROCEDURE interSort($A, n$)
  if ($n > 1$) then
    imparte($A, n, A_0, n_0, A_1, n_1$);
    in parallel
      interSort($A_0, n_0$),
      interSort($A_1, n_1$)
    end in parallel
    combina($A_0, n_0, A_1, n_1, A, n$);
  end if
```

Recursive Data

Recursive data structures seem to have little exploitable concurrency.
But in some cases, the structure can be transformed.

Find Root Problem

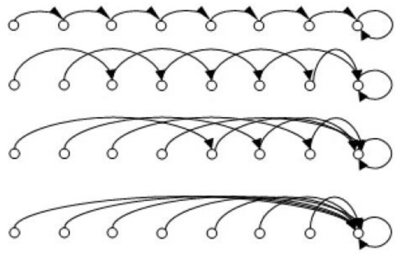


Step 1

Step 2

Step 3

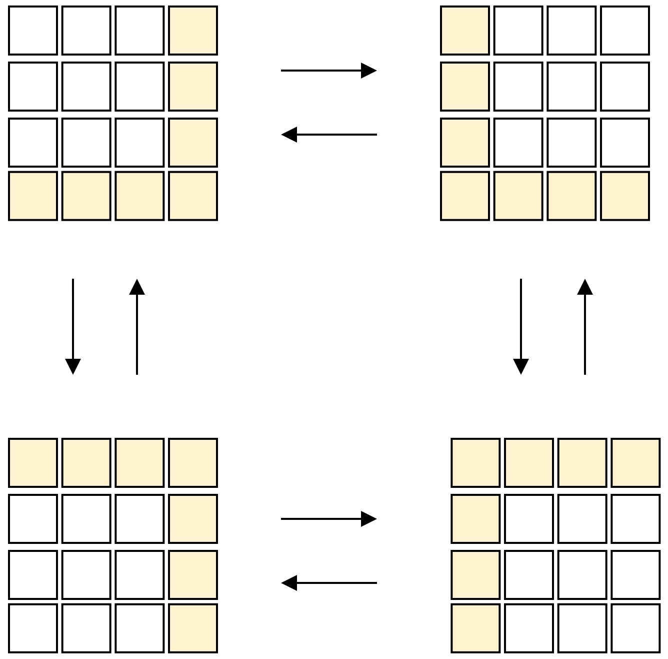
Pointer jumping – recursive doubling
example: tree -> all path length



```
ALGORITHM LUNGIME-DRUM< $n, parinte[0..n-1], distanta[0..n-1]$ >
  for $i = 0, n-1$ in parallel do
    $distanta[i] \leftarrow 1;$
    while $(parinte[i] \neq -1)$ do
      $distanta[i] \leftarrow distanta[i] + distanta[parinte[i]];$
      $parinte[i] \leftarrow parinte[parinte[i]];$
    end while
  end for
```

Geometric decomposition

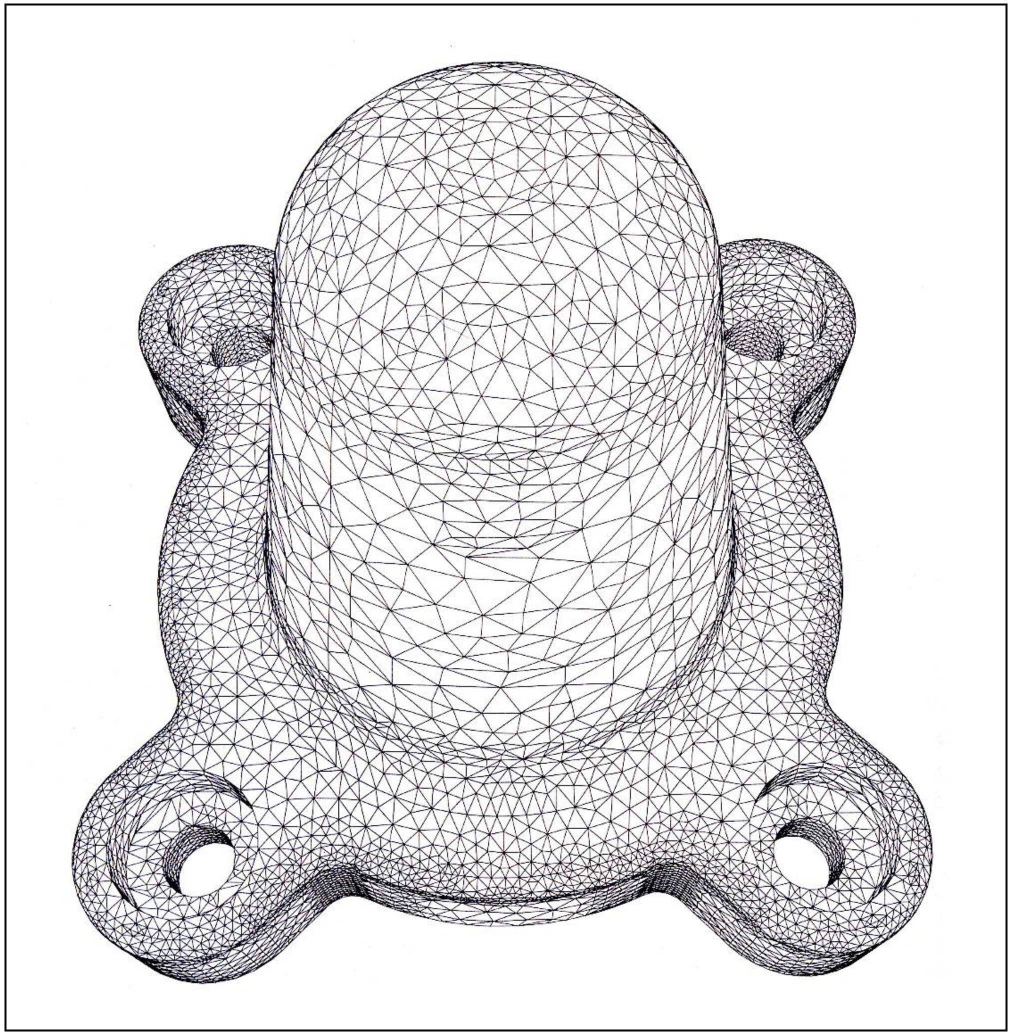
Dependencies exist but the communication is done in predictable (geometric) neighbor-to-neighbor paths.



Neighbor-To-Neighbor communication

Irregular Mesh

Communication in non-predictable paths in mesh topology.



Hard to define due to varying communication patterns.

Start point :
Pattern that constructed this mesh.

all starts from ...

Decomposing a sequential problem to expose concurrency.

- **Data decomposition**
  - Concurrency comes from working on different data elements simultaneously.
- **Functional decomposition**
  - Concurrency comes from working on independent functional tasks simultaneously.
- **Data flow**
  - Concurrency comes from working simultaneously on different streaming data on different stages of computation.

Decomposition Techniques

How does one decompose a task into various subtasks?

- there is no single recipe that works for all problems,
- a set of commonly used techniques that apply to broad classes of problems.

These include:

- recursive decomposition
- data decomposition (geometric decomposition)
- exploratory decomposition
- speculative decomposition

- Pipelines…

Recursive Decomposition

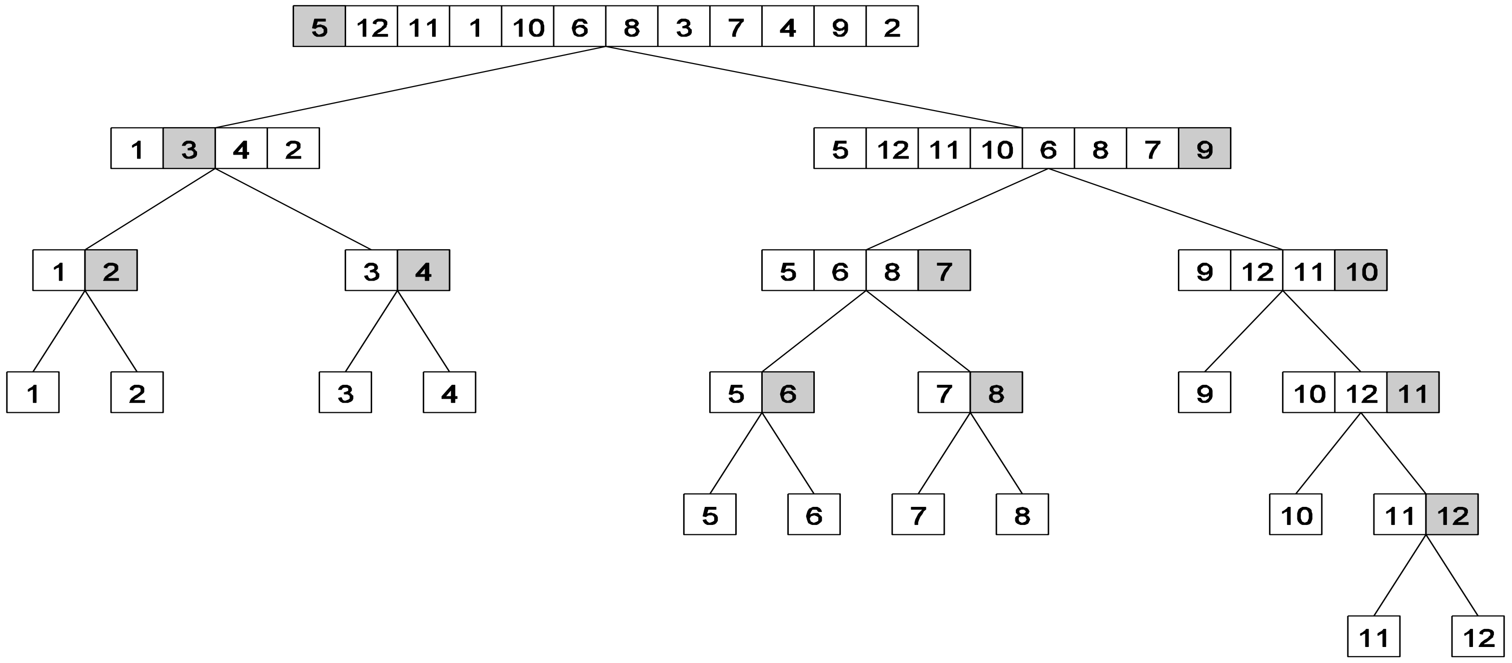
- Generally suited to problems that are solved using the **divide-and-conquer** strategy.

- 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 1

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 2

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
```

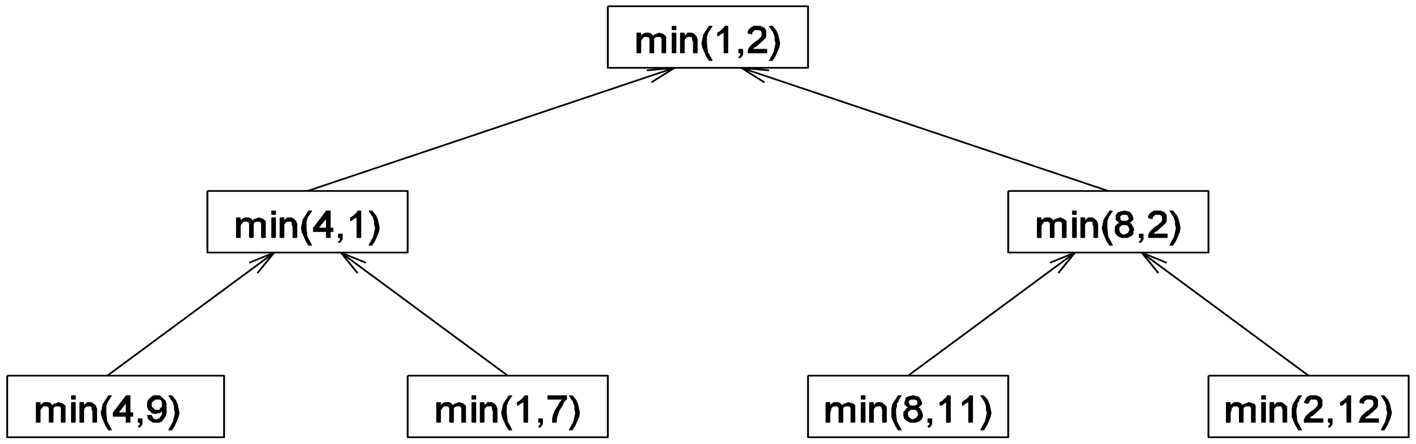
Recursive Decomposition: Example

We can rewrite the loop as follows:

```
procedure RECURSIVE_MIN (A, n)
begin
if ( n = 1 ) then
    min := A [0]  ;
else
    lmin := RECURSIVE_MIN ( A, n/2 );
    rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
    if (lmin  < rmin) then
            min := lmin;
    else
            min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

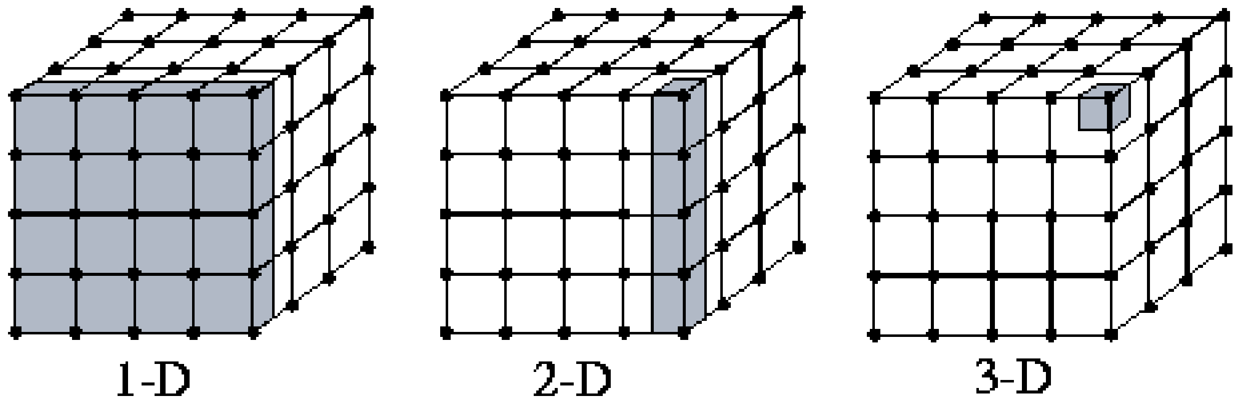
Recursive Decomposition: Example

The code in the previous slide 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:

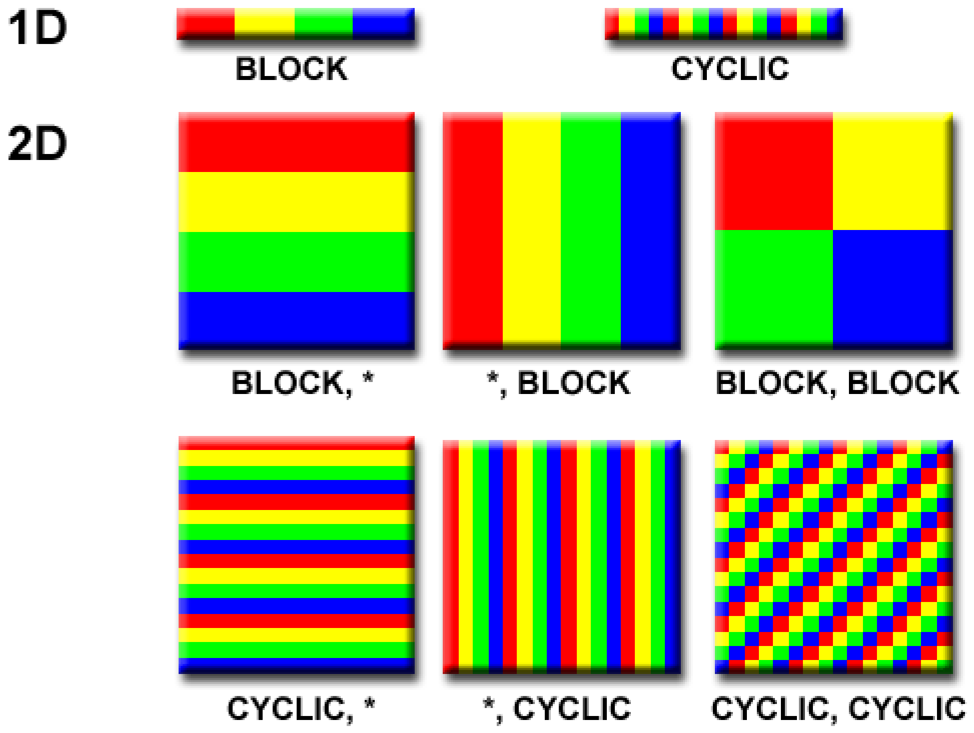


Data Decomposition (geometric)

- 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.



Variants


1D
BLOCK
CYCLIC
2D
BLOCK, *
*, BLOCK
BLOCK, BLOCK
CYCLIC, *
*, CYCLIC
CYCLIC, CYCLIC

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 previous slide, with identical output data distribution, we can derive the following two (other) decompositions:

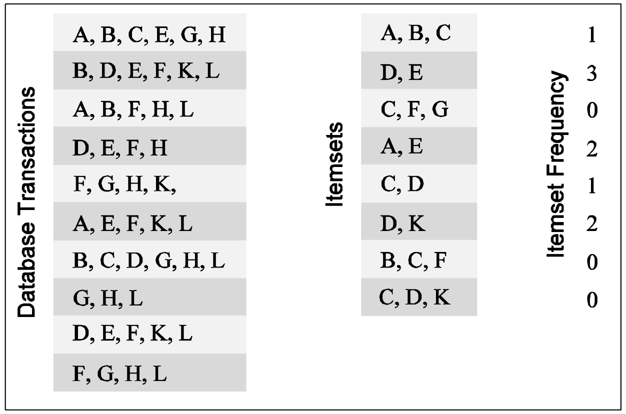
Decomposition I	**Decomposition II**
Task 1: $\boldsymbol{C}_{1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$	Task 1: $\boldsymbol{C}_{1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$
Task 2: $\boldsymbol{C}_{1,1} = \boldsymbol{C}_{1,1} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$	Task 2: $\boldsymbol{C}_{1,1} = \boldsymbol{C}_{1,1} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$
Task 3: $\boldsymbol{C}_{1,2} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$	Task 3: $\boldsymbol{C}_{1,2} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$
Task 4: $\boldsymbol{C}_{1,2} = \boldsymbol{C}_{1,2} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$	Task 4: $\boldsymbol{C}_{1,2} = \boldsymbol{C}_{1,2} + \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$
Task 5: $\boldsymbol{C}_{2,1} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$	Task 5: $\boldsymbol{C}_{2,1} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$
Task 6: $\boldsymbol{C}_{2,1} = \boldsymbol{C}_{2,1} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$	Task 6: $\boldsymbol{C}_{2,1} = \boldsymbol{C}_{2,1} + \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$
Task 7: $\boldsymbol{C}_{2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$	Task 7: $\boldsymbol{C}_{2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$
Task 8: $\boldsymbol{C}_{2,2} = \boldsymbol{C}_{2,2} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$	Task 8: $\boldsymbol{C}_{2,2} = \boldsymbol{C}_{2,2} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

Output Data Decomposition: Example

Consider the problem of counting the instances of given itemsets in a database of transactions.

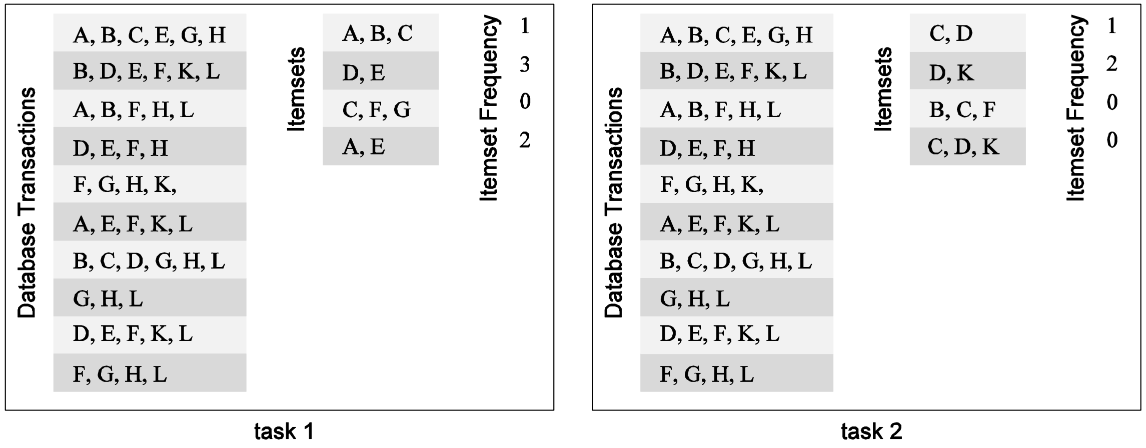
In this case, the output (itemset frequencies) can be partitioned across tasks.

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



Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

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



task 1:

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 2:

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	C, D	1
B, D, E, F, K, L	D, K	2
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

Output Data Decomposition: Example

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

- If the database of transactions is replicated across the processes (or there is a shared access), 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.

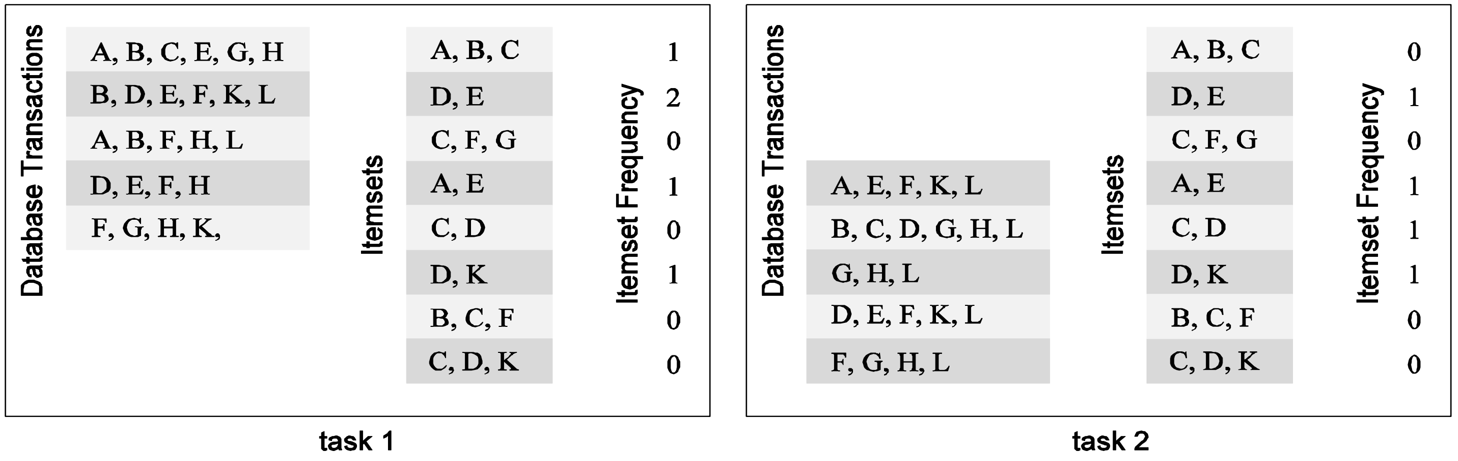
Input Data Partitioning

- 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



task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,

Itemsets	Itemset Frequency
A, B, C	1
D, E	2
C, F, G	0
A, E	1
C, D	0
D, K	1
B, C, F	0
C, D, K	0

task 2

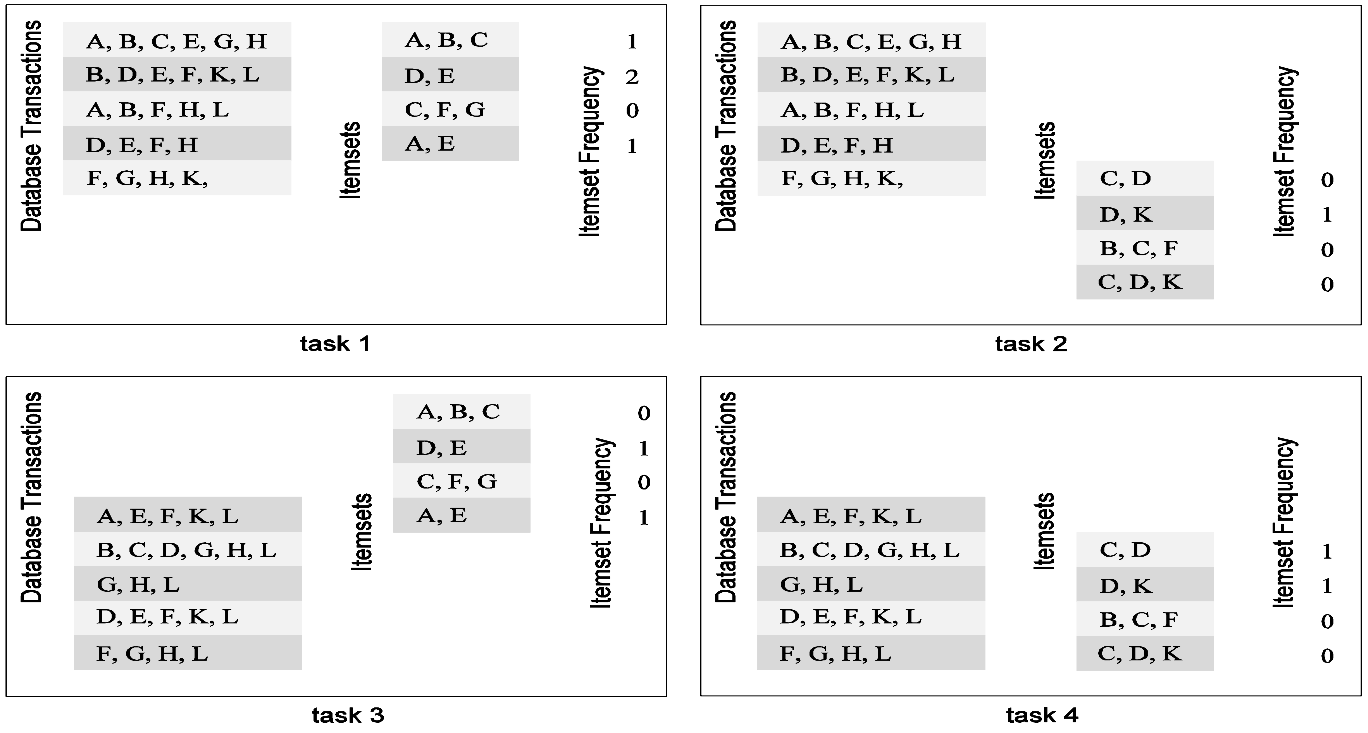
Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Itemsets	Itemset Frequency
A, B, C	0
D, E	1
C, F, G	0
A, E	1
C, D	1
D, K	1
B, C, F	0
C, D, K	0

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:

Partitioning both transactions and frequencies among the tasks



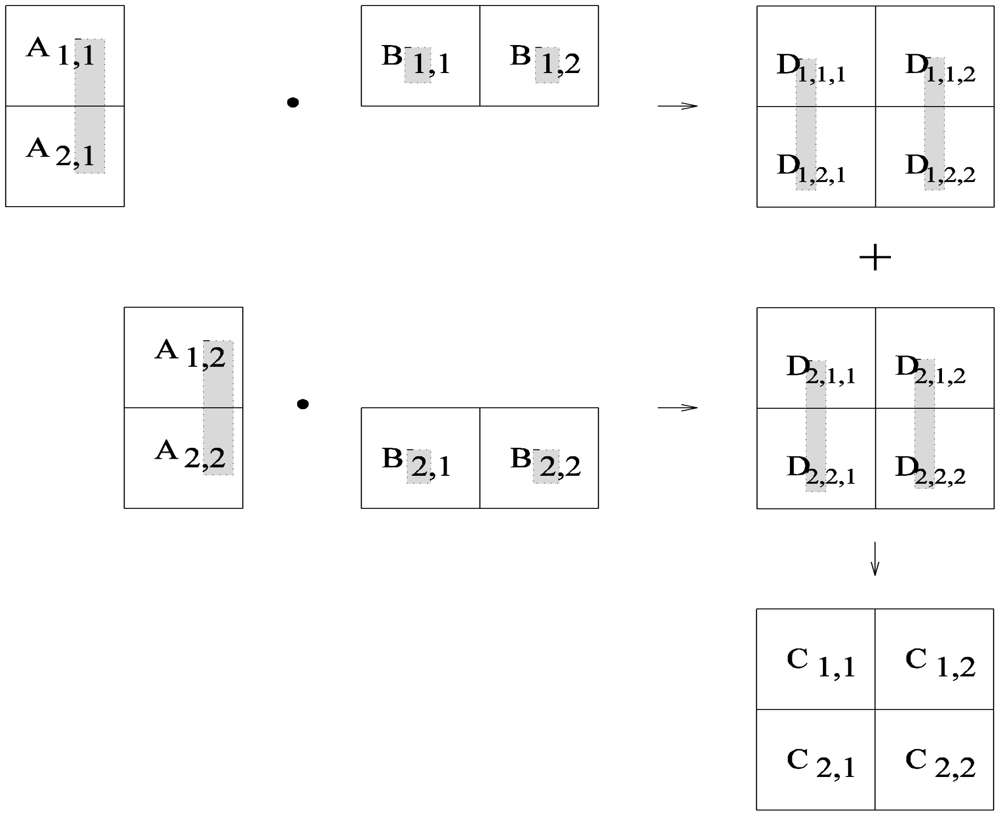
Intermediate Data Partitioning

- 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

Dense matrix multiplication => variant:

-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

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \left(\begin{array}{c} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{array} \right)$$

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

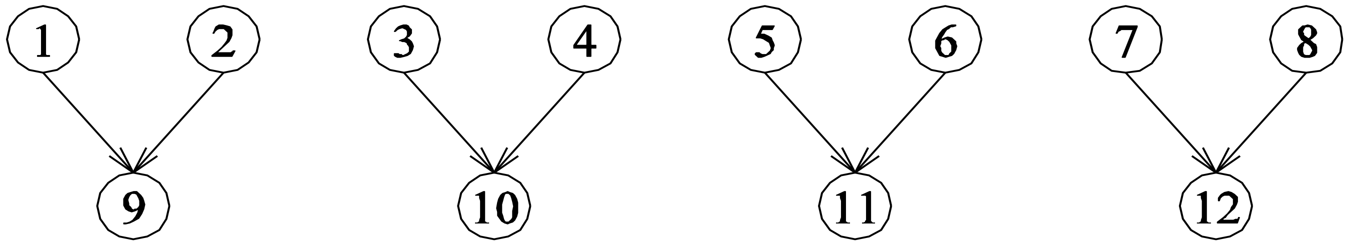
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

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:


$$\begin{tikzcd}
1 \arrow[dr] & & 2 \arrow[dl] & 3 \arrow[dr] & & 4 \arrow[dl] & 5 \arrow[dr] & & 6 \arrow[dl] & 7 \arrow[dr] & & 8 \arrow[dl] \\
& 9 & & & 10 & & & 11 & & & 12 &
\end{tikzcd}$$

The Owner Computes Rule

- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies 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.

Data Replication

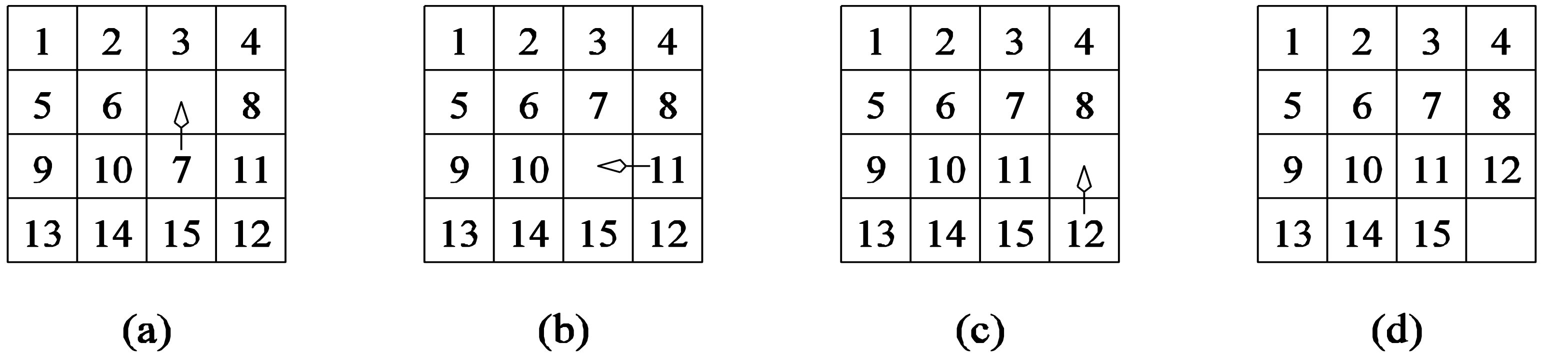
- Replicate data – in order to create independent tasks
- For example, for matrix multiplication: replication on third dimension.

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,
  - search algorithms for finding approximate solutions to hard optimization problems
  - 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).

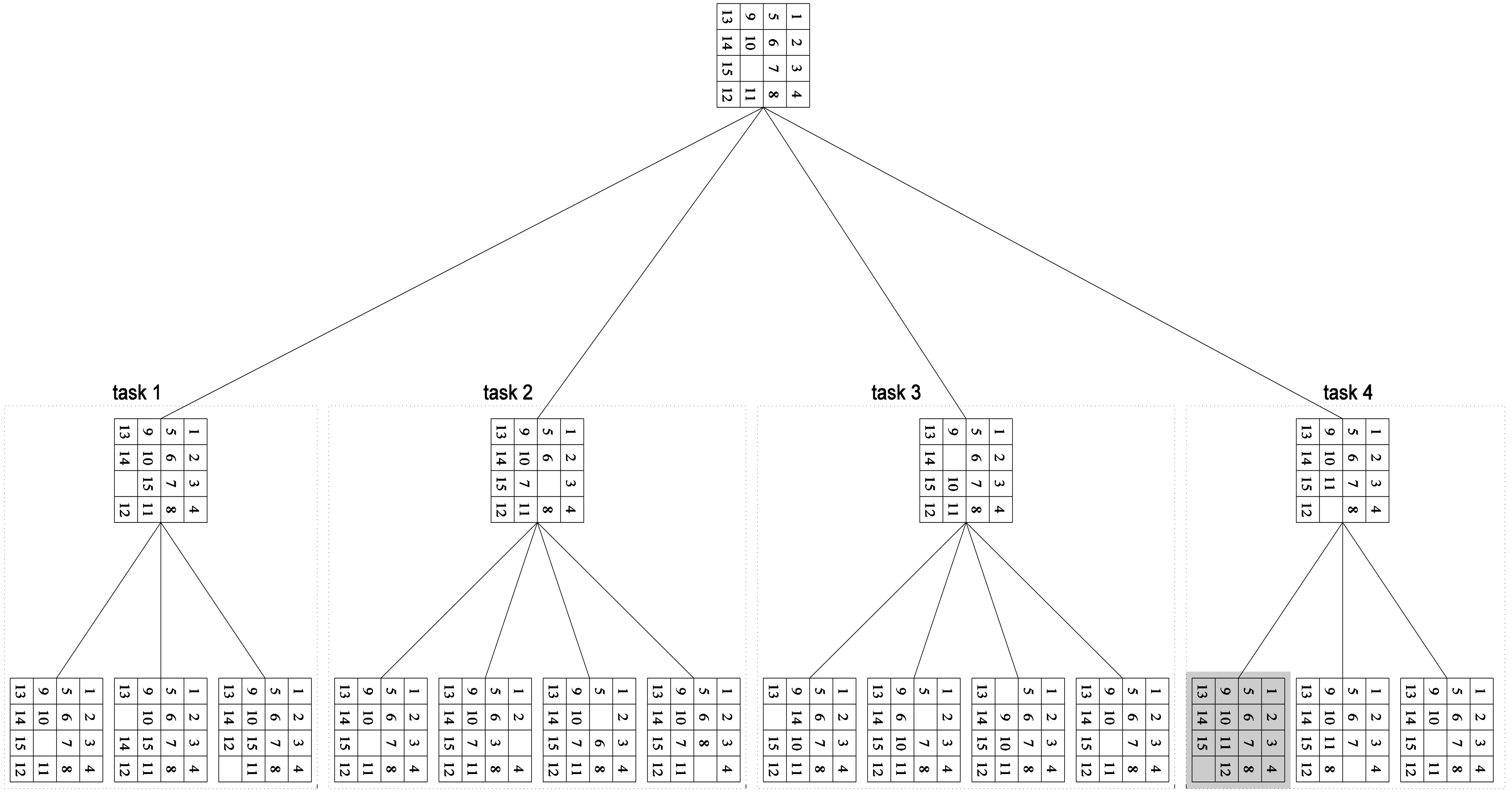


a sequence of three moves that transform a given initial state (a) to desired final state (d).

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

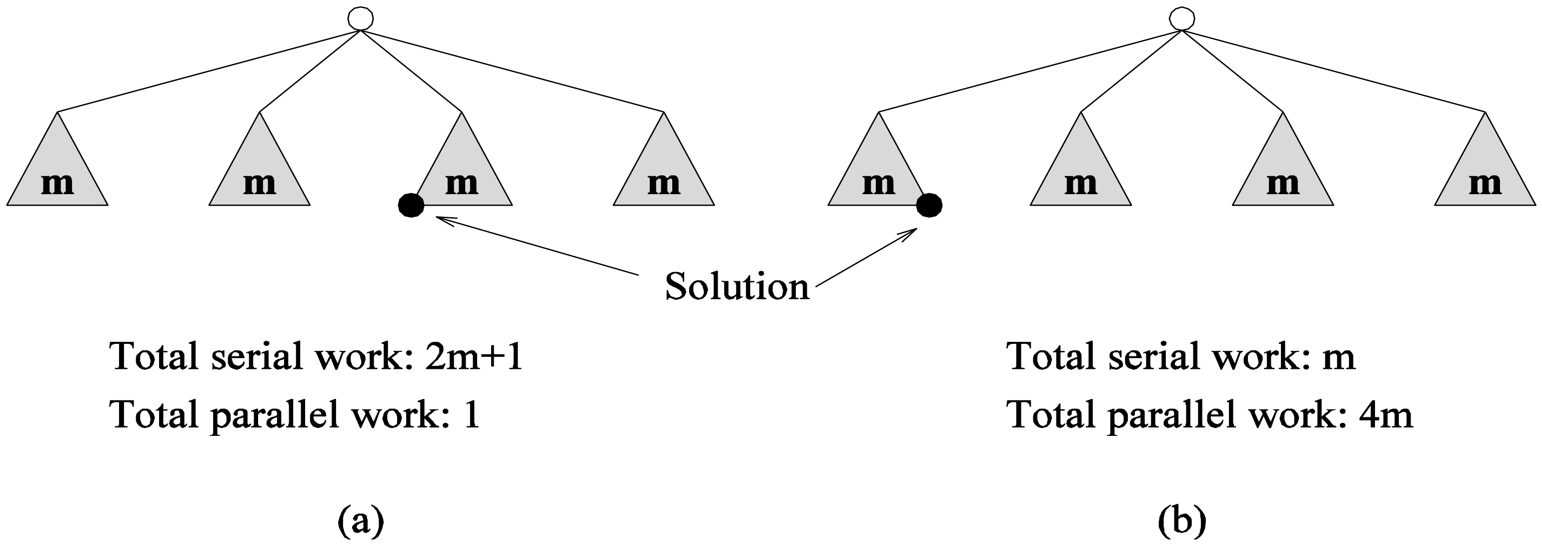
Exploratory Decomposition: Example

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



Exploratory Decomposition: Anomalous Computations

- In many instances of exploratory decomposition, the decomposition technique may change the amount of work done by the parallel formulation.
- This change results in super- or sub-linear speedups.



Speculative Decomposition

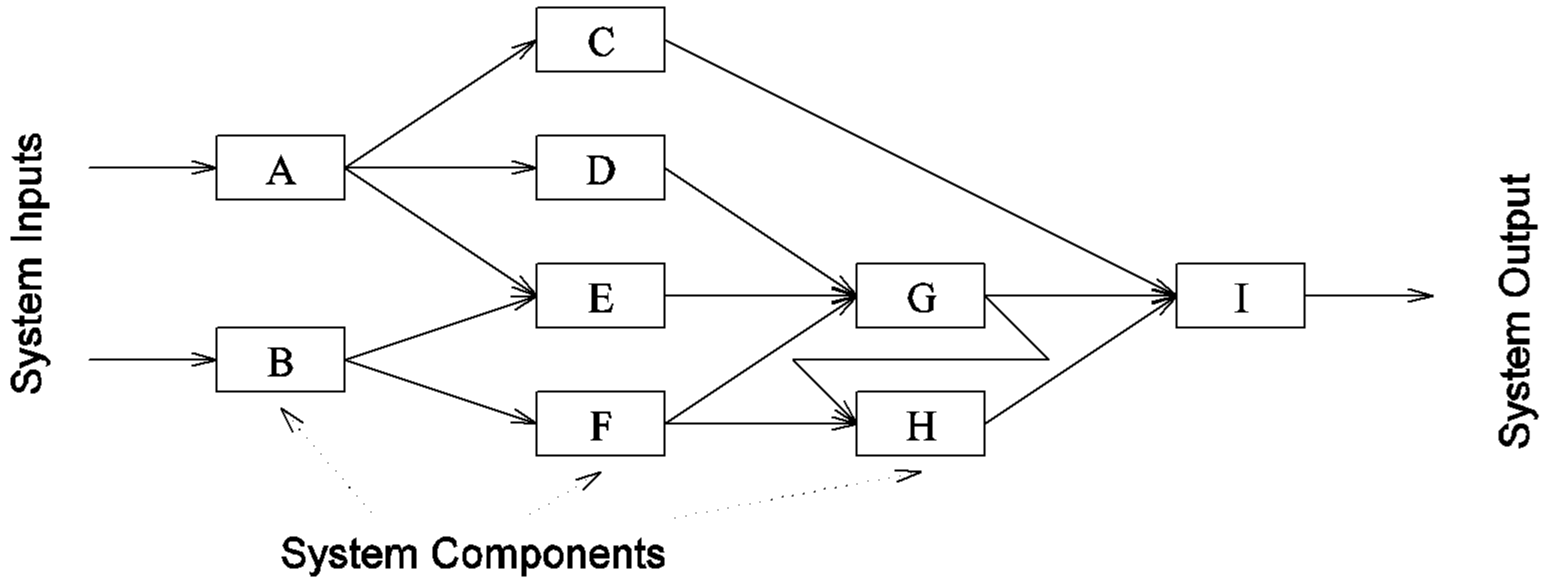
- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications:
  - *conservative* approaches, which identify independent tasks only when they are 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.

Classical example: ***discrete event simulation***

- The central data structure in a discrete event simulation is a time-ordered event list

Speculative Decomposition: Example

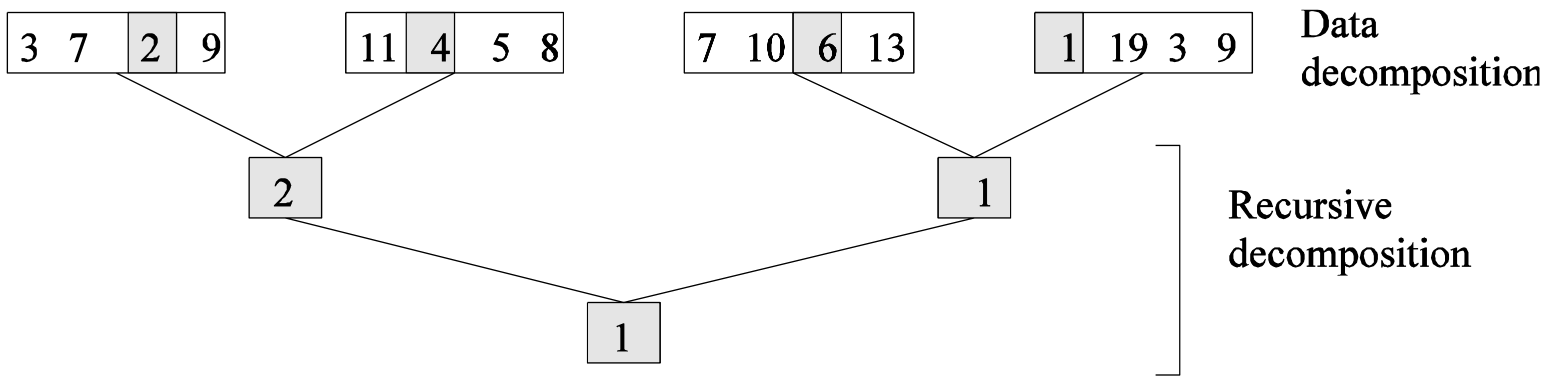
- The simulation of a network of nodes (for instance, an assembly line or a computer network through which packets pass).
- The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue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



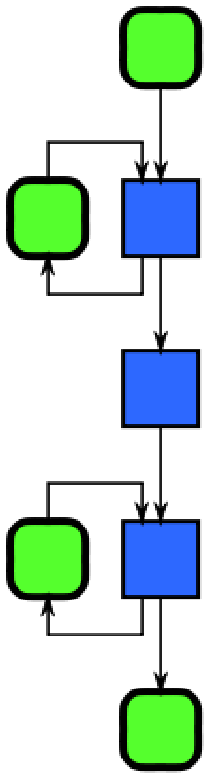
Hybrid Decompositions

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.

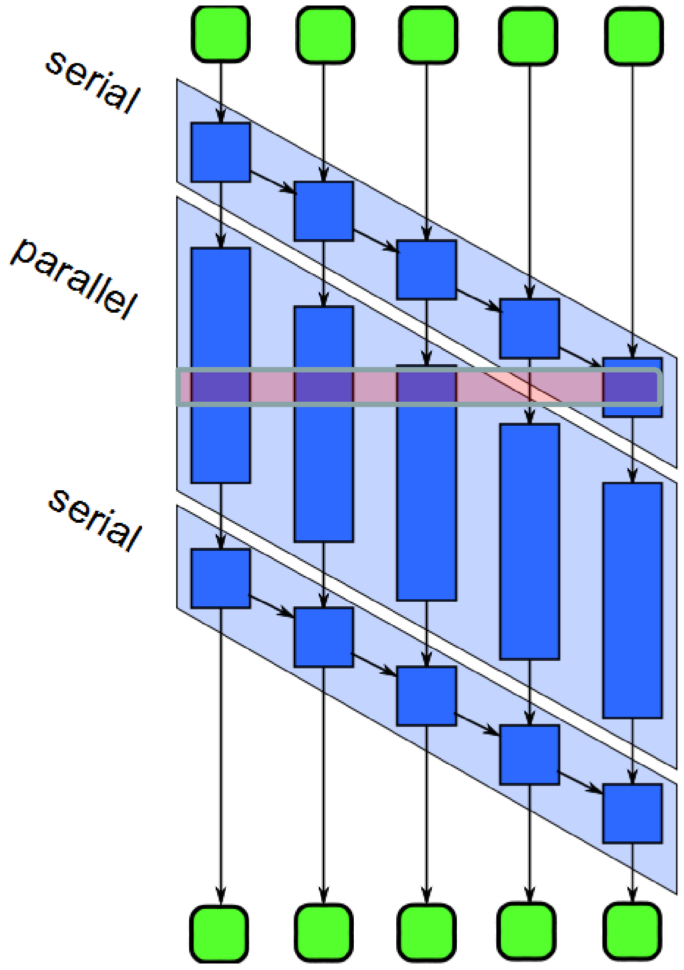


PIPELINE



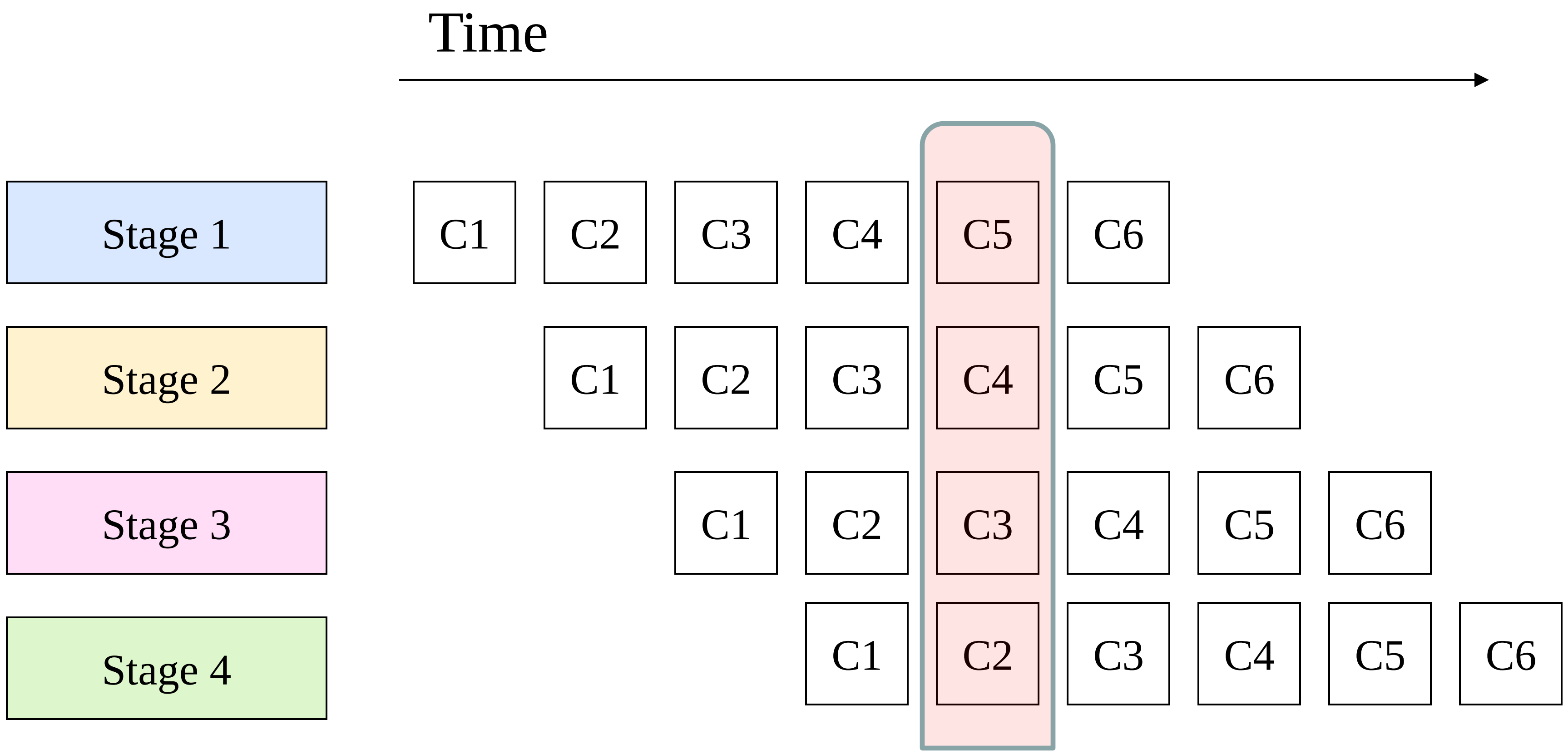
A series of ordered but independent computation stages need to be applied on data,
where each output of a computation becomes input of subsequent computation.

Pipeline

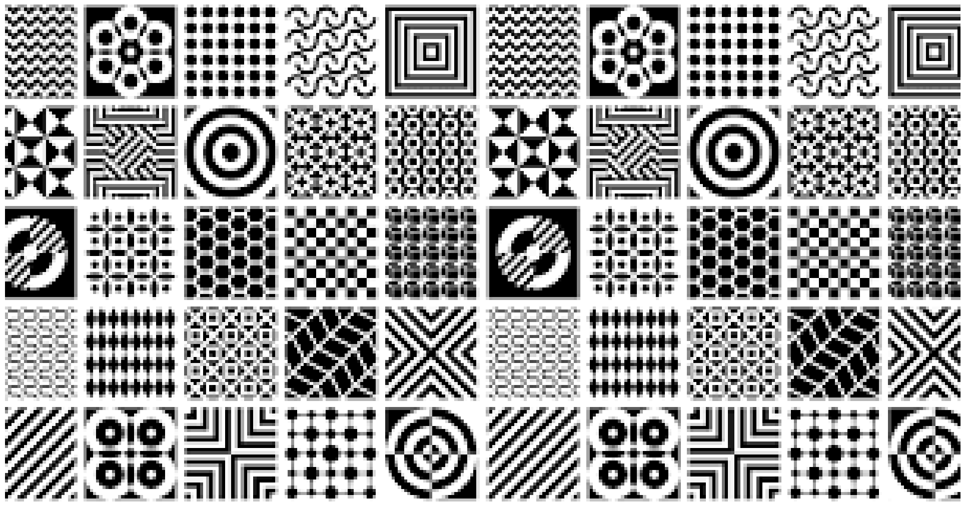


Different stages are executed in parallel!

Pipeline


Time
Stage 1
Stage 2
Stage 3
Stage 4
C1 C2 C3 C4 C5 C6
C1 C2 C3 C4 C5 C6
C1 C2 C3 C4 C5 C6
C1 C2 C3 C4 C5 C6

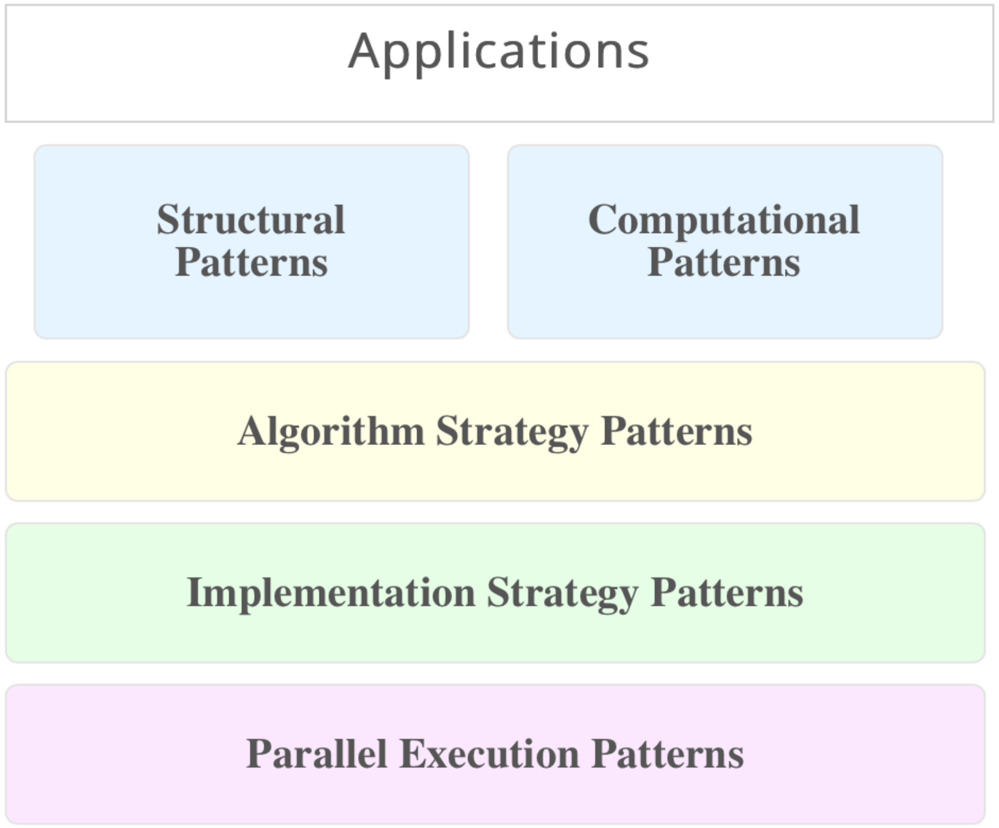
A more general perspective
=
Pattern Languages



A Design Pattern Language for Engineering (Parallel) Software

- https://patterns.eecs.berkeley.edu

Parallel and Distributed together!


Applications
Structural Patterns
Computational Patterns
Algorithm Strategy Patterns
Implementation Strategy Patterns
Parallel Execution Patterns

Structural Patterns

- Model-View-Controller
- Map Reduce
- Layered systems
- Event-based, implicit invocation
- Iterative Refinement
- Process Control
- Arbitrary Static Task Graph
- Pipe and Filter
- Agent and Repository

Computational Patterns

- Backtracking & Branch and Bound
- Monte Carlo Methods
- Circuits
- N-Body Methods
- Dense Linear Algebra
- Sparse Linear Algebra
- Dynamic Programming
- Spectral Methods
- Finite State Machine
- Structured Grids
- Graph Algorithms
- Unstructured Grids
- Graphical Models
- Sorting

Parallel Algorithm Strategy Patterns

- Task Parallelism
- Recursive Splitting
- Discrete Event
- Pipeline
- Geometric Decomposition
- Data Parallelism
- Non-work-efficient Parallelism
- Speculation

Implementation Strategy Patterns

Program Structure

- SPMD
- Master-Worker
- Strict Data Parallelism
- Loop Parallelism
- Fork-Join
- BSP
- Actors
- Graph Partitioning

Data Structure

- Shared Queue
- Shared Hash Table
- Distributed Array
- Shared Data
- Memory Parallelism

Parallel Execution Patterns

Advancing Program Counters

- MIMD
- Task Graph
- SIMD
- Digital Circuits
- Thread Pool
- Speculation
- Data Flow

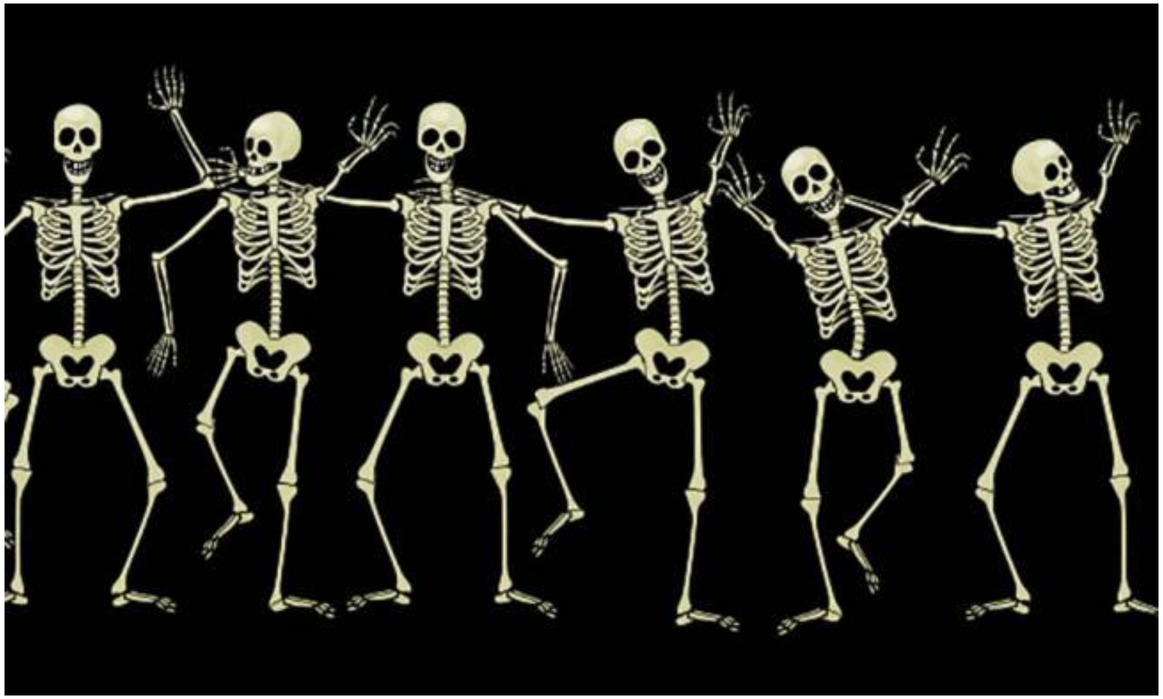
Coordination

- Message Passing
- Collective Communication
- Mutual Exclusion
- P2P Sync
- Collective Synchronization
- Transactional Memory

MODELS

- DIFFERENT LEVELS OF ABSTRACTION

SKELETONS



History

- In late '80s, Murray Cole introduced the **algorithmic skeleton concept**
- A research community was developed around this (mainly build by researchers coming from High Performance Computing community)
- Goal: *the development of programming frameworks suitable to support efficient parallel programming, in particular with respect to the performance achieved in parallel programs.*
- The community evolved and the concept of algorithmic skeleton evolved as well.
- Nowadays there are several research groups around the world, building parallel frameworks based on this concept.

The link with patterns

- In the '00s, software engineering researchers oriented the

design pattern concept onto

parallel programming.

- The parallel design patterns has been and are currently investigated by a community which is fairly disjoint from the one investigating algorithmic skeletons.

- The research on parallel design patterns produced (and it is currently producing) results that are similar to those produced in the algorithmic skeleton framework.

- Sometimes the results only differ in the “jargon” used to illustrate them.

- In more interesting cases they are similar but still
  - more software engineering oriented perspective used in investigation may provide useful hints to algorithmic skeleton designers as well.

General properties

- **Algorithmic skeletons** = pre-defined patterns encapsulating the structure of a parallel computation
- They are provided to user as ***building blocks*** to be used to write applications.

- ***Each skeleton corresponded to a single parallelism exploitation pattern.***

- A skeleton based programming framework was defined as a programming framework providing programmers with algorithmic skeletons that may be used to model the parallel structure of the applications at hand.

- In a skeleton based programming framework the programmer has no other way (but instantiating skeletons) to structure his/her parallel computation.

- Any skeleton application was the result of the instantiation of a single algorithmic skeleton and thus, in turn, of a single parallelism exploitation pattern.

First Definition ***[M. Cole 1988]***

“The new system presents the user with a selection of independent ‘algorithmic skeleton’,

each of which describes the structure of a particular style of algorithm,

in the way in which higher order functions represent general computational frameworks in the context of functional programming languages.

The user must describe a solution to a problem as an instance of the appropriate skeleton.”

A business logic connection

- The term `business logic’ usually denotes the functional code of the application, distinct from the non-functional code modeling.

- Example:
  - an application processing a set of images to apply to each one of the images a filter f
    - the code for the filter is business logic code,
    - the code needed to set up a number of parallel/concurrent/distributed agents that altogether contribute to the parallel computation of the filtered images forms a lower layer.

M. Cole “skeleton manifesto"

- Many parallel algorithms can be characterized and classified by their adherence to one or more of a number of ***generic patterns of computation* and *interaction***.
  - For example, many diverse applications share the underlying control and data flow of the pipeline paradigm.

- Skeletal programming proposes that such patterns be ***abstracted*** and provided as ***a programmer's toolkit***, with :
  - specifications that transcend architectural variations,
  - implementations which recognize these architectural variations to enhance performance.

New definition

An algorithmic skeleton is a parametric, reusable and portable programming abstraction that

models

- a known, common and efficient parallelism exploitation pattern.

Characteristics:

- it models a known, **<u>common</u>** parallelism exploitation **pattern**.
  - (in this context) no interest in patterns that are not commonly found in working parallel applications.
  - un- common patterns could be anyway used with those skeleton frameworks that provide some degree of extensibility
- it is **portable**, that means
  - an efficient implementation of the skeleton should exists on a range of different architectures.
- it models **efficient** parallelism exploitation patterns.
  - Skeletons only model those patterns that have some known, efficient implementation on a variety of target architectures.
- it is **reusable**, that means
  - it could be used in different applications/contexts without actually requiring modifications
- it is **parametric** in that
  - it accepts parameters that can be used to specialize ***what is computed*** by the skeleton, rather than how the parallel computation is performed.

Advantages

- ***the abstraction level*** presented to the programmer by the programming framework ***is higher*** than the one presented by traditional parallel programming frameworks. This in turn has two consequences:
  - The whole process of parallel programming is simplified.
    - Parallelism exploitation mainly consist in properly ***instantiating*** the skeleton abstractions provided by the framework rather than programming your own parallel patterns using the concurrency, communication and synchronization mechanisms available.
  - Application programmers may experiment rapid ***prototyping*** of parallel applications.
    - Different application prototypes may be obtained by instantiating (that is by providing the proper functional/business logic code) ***different skeletons*** among those available or, in case this is allowed, different ***compositions of existing skeletons***.

… advantages

- The **correctness** and the **efficiency** of the parallelism exploitation is not a concern of the application programmers.
  - Rather, the ***system programmers*** that designed and implemented the skeleton framework guarantee both correctness and efficiency.

- Portability on different target architectures is guaranteed by the framework.
  - An algorithmic skeleton application can be moved to a different architecture just by recompiling.
  - Portability has to be intended in this case not only as
    - "***functional portability***", but also as
    - "***performance portability***" as the skeleton framework usually exploits best implementations of the skeletons on the different target architectures supported.

… advantages

- Application debugging is simplified.
  - Only sequential code (that is business logic code) has to be debugged by the application programmers.

- Possibilities for static and dynamic optimizations are much more consistent than in case of traditional, unstructured parallel programming frameworks as the skeleton structure completely exposes to the programming tools the parallel structure of the application.

- Application deployment and (parallel) run, that represent complex activities is completely in charge of the skeleton framework

disadvantages

- In case none of the skeletons provided to the programmer (or none of the possible skeleton nestings) suits to model the parallel structure of the application the application could not be run through the skeleton framework.
- Functional and performance portability across different target architectures requires a huge effort by the skeleton framework designers and programmers.
  - The effort required may be partially reduced by properly choosing the architecture targeted by the skeleton framework, e.g. by considering some “higher level" target to produce the “object" code of the skeleton framework.

Skeletons as higher order functions with associated parallel semantics

- Higher order functions

Stream

```
type 'a stream = EmptyStream | Stream of 'a * 'a stream;;
```

- Pipeline

pipeline :: $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha$ *stream* $\rightarrow \gamma$ *stream*

```
let rec pipeline f g =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> Stream((g(f x)),(pipeline f g y));;
```

- Parallel semantics = informal text

the pipeline stages are computed in parallel onto different items of the input stream. If x_i and x_{i+1} happen to be consecutive items of the input stream, than $f(x_{i+1})$ and $g(f(x_i))$ will be computed in parallel, for any i

Pipeline application

- an application to filter images and then recognize characters string appearing in the images, provided it has available two functions *filter :: image -> image* and *recognize :: image -> string list*, he can simply write a program such as:

  let main = pipeline filter recognize;;

- *The parameters of the skeleton defines only the business logic – e.g. pipeline stages*

- *There could be also non-functional parameters = e.g. parallelism degree*

- The parallelism degree that may be conveniently used was evident in pipeline:
  - Having two stages, a parallelism degree equal to 2 may be used to exploit all the parallelism intrinsic to the skeletons.

the farm skeleton

- the farm skeleton models embarrassingly parallel stream parallelism

$$farm :: (\alpha \to \beta) \to \alpha\ stream \to \beta\ stream$$

```
let rec farm f =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> Stream((f x),(farm f y));;
```

- Parallelism semantics:

the computation of any items appearing on the input stream is performed in parallel

  – according to the parallel semantics a number of parallel agents computing function f onto input data items equal to the number of items appearing onto the input stream could be used.

Problems:

1. items of the item stream do not exist all at the same time. A stream is not a vector.

- Items of the stream may appear a different times.
- when we talk of consecutive items xi and xi+1 of the stream we refer to items appearing onto the stream at times ti and ti+1 with ti < ti+1.

=> *it makes no sense to have a distinct parallel agent for all the items of the input stream, as at any given time only a fraction of the input stream will be available*.

2. if we use an agent to compute item xi, presumably the computation will end at some time tk. If item xj appears onto the input stream at a time tj > tk this same agent

can be used to compute item xj rather than picking up a new agent.

Parallelism degree

- Type:

$$farm :: (\alpha \to \beta) \to int \to \alpha\ stream \to \beta\ stream$$

```
let rec farm f n:int =
function
EmptyStream -> EmptyStream
|
Stream(x,y) -> Stream((f x),(farm f n-1 y));;
```

- parallel semantics
  - the computation of consecutive items of the input stream is performed in parallel on n parallel agents.

- Debate:
  - Include or not non-functional parameters

Alternative definitions

assume pipeline and farm just operate on a global input stream of data to produce a global stream of results

- pipeline

```
let pipeline f g = function x -> (g (f x));;
```

- task farm skeleton corresponds to identity:

```
let farm f = function x -> (f x);;
```

- The higher order function taking care of exploiting parallelism can then be defined as follows:

```
let rec streamer f x =
     match x with
     EmptyStream -> EmptyStream
     | Stream (v,s) -> Stream ((f v), (streamer f s));;
```

- Example

```
let main =
          streamer (pipeline filter recognize);;
```

parallel semantics is only associated to the streamer function as:

the processing of independent input stream data items is performed in parallel

refinement

- This way of modelling stream parallelism corresponds to the idea of structuring
  - all stream parallel applications as farms with more coarse grain computations as functional parameter.

Example: Let's suppose that the `recognize` phase is much more computationally expensive with respect to the `filter` phase.

Using the explicit stream handling versions of pipeline and farm skeletons, we would write the program as

```
let main =
        pipeline (filter) (farm recognize);;
```

analysis

- Recalling the parallel semantics of these skeletons we can easily understand that given a stream with data items $x_1; \ldots; x_m$ (with xi appearing on the stream after x_{i-k} for any k) the computations of

recognize(y_i) recognize(y_{i+k})

(being y_i = filter(x_i)) happen in parallel, as well as the computations of

recognize(yi) filter(x_{i+k})

- If we use the alternative modelling of stream parallelism the program instead is written as

```
let main =
        streamer (pipeline filter recognize);;
```

- and in this case the only computations happening in parallel are the

recognize(filter(x_i)) recognize(filter(x_{i+k}))

Skeleton framework basics

Common subset contains skeletons from three different classes of skeletons:

- Stream parallel skeletons
- Data parallel skeletons
- Control parallel skeletons

Stream parallel skeletons

- Stream parallel skeletons are those exploiting parallelism among computations relative to different, independent data items appearing on the program input stream.

- Each independent computation end with the delivery of one single item on the program output stream.

- Common stream parallel skeletons are the pipeline and task farm skeletons

Data parallel skeletons

- Data parallel skeletons are those exploiting parallelism in the computation of ***different sub-tasks*** derived from the ***same input task***.

- the data parallel skeletons are used to speedup a single computation by splitting the computation into parallel sub-tasks.

- Data parallel computation do not concern, *per se*, stream parallelism. But may be the case that a data parallel computation is used to implement the computation relative to a **single stream item** (item could be a complex data structure) in a stream parallel program.

- Typical data parallel skeletons considered in the skeleton based programming frameworks include ***map, reduce*** and ***parallel prefix*** skeletons.

Map

Given a function f and a vector x the map skeleton computes a vector y whose elements are such that for any i: yi = f(xi). The map skeleton is defined as follows:

```
let map1 f x =
        let len = Array.length x in
        let res = Array.create len (f x.(0)) in
                for i=0 to len-1 do
                        res.(i) := (f x.(i))
                done;
        res;;
```

Similar reasoning to task farm.
Difference = the kind of input data passed to the parallel agents: in the farm case=>the input stream; in the map case=>the single input data.

and its type is therefore $map :: (\alpha \rightarrow \beta) \rightarrow \alpha\ array \rightarrow \beta\ array$

The parallel semantics of the map states that
each element of the resulting vector is computed in parallel.

Reduce

- The reduce skeleton computes the “sum" of all the elements of a vector with a function -> associative oper

```
let rec reduce f x =
let len = Array.length x in
        let res = ref x.(0) in
                for i=1 to len-1 do
                        res := (f !res x.(i))
                done;
                !res;;
```

- Type: $(\alpha \to \alpha \to \alpha) \to \alpha\ array \to \alpha$

- Parallel semantics:

the reduction is performed in parallel, using n agents organized in a tree. Each agent computes f over the results communicated by the son agents. Root agent delivers the reduce result. Leaf agents possibly compute locally a reduce over the assigned partition of the vector.

Parallel prefix

- The parallel prefix skeleton computes all the partial sums of a vector, using a function – associative operator
- given a vector x it computes the vector whose elements are:

x1; x1 + x2; x1+x2 +x3; … ; x1+ x2 +x3 … +xlen

```
let parallel_prefix f x =
        let len = Array.length x in
        let res = Array.create len x.(0) in
                res.(0) <- x.(0);
                        for i=1 to len-1 do
                                res.(i) := (f x.(i) res.(i-1))
                        done;
        res;;
```

Type:

$$(\alpha \to \alpha \to \alpha) \to \alpha\ array \to \alpha\ array$$

Parallel prefix

- The parallel semantics states that:

the parallel prefix is computed in parallel by n agents with a schema similar to the

one followed to compute the reduce. Partial results are logically written to the proper

locations in the resulting vector by intermediate nodes.

Remark: there are many possible implementation of parallel prefix – specialized parallel algorithms

Others skeletons

- Stencil (defined using a list of indices)

```
let stencil f stencil_indexes a =
        let n = (Array.length a) in
        let item a i = a.((i+n) mod n) in
        let rec sten a i =
                function
                        [] -> []
                        | j::rj -> (item a (i+j))::(sten a i rj) in
                        let res = Array.create n (f a.(0) (sten a 0 stencil_indexes))
in
                        for i=0 to n-1 do
                                res.(i) <- (f a.(i) (sten a i stencil_indexes))
                        done;
        res;;
```

Divide &Conquer

```
let rec divconq cs dc bc cc x =
        if(cs x) then (bc x)
        else (cc (List.map (divconq cs dc bc cc) (dc x)));;
```

Type:

$$(\alpha \to bool) \to (\alpha \to \alpha\ list) \to (\alpha \to \beta) \to (\beta\ list \to \beta) \to \alpha \to \beta$$

- The parallel semantics associated to the divide&conquer skeleton states that:

computation of Sub problems deriving from the \divide" phase are computed in parallel.

Control parallel skeletons

Control parallel skeletons actually do not express parallel patterns. Rather, they express structured coordination patterns and model those parts of a parallel computation that are needed to support or coordinate parallel skeletons.

- Sequential skeleton
- Conditional skeleton
- Iterative skeleton

- Conditional skeleton which is used to model if-then-else computations. Such a skeleton can be expressed by the following higher order function:

```
let ifthenelse c t e x =
        match(c x) with
            true -> (t x)
            | false -> (e x);;
```

- type is $ifthenelse : (\alpha \rightarrow bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

Iterative skeleton

- Iterative skeleton is used to model definite or indefinite iterations.

- As an example, a *skwhile* iterative skeleton may be represented through the higher order function

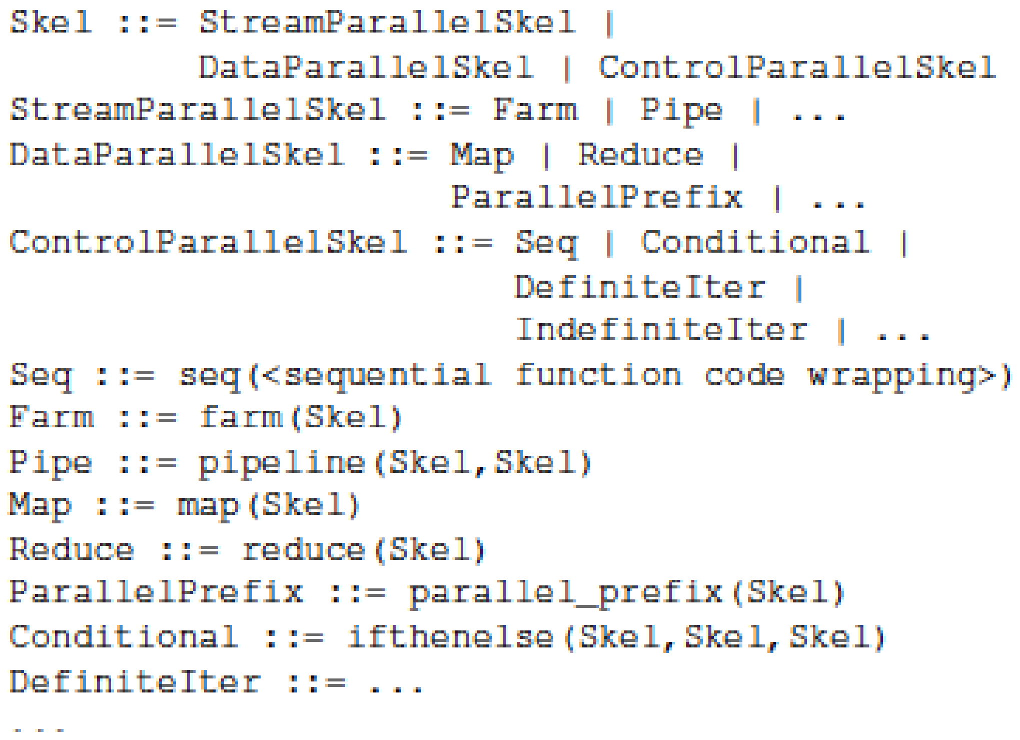
```
let rec skwhile c b x =
        match(c x) with
            true -> (skwhile c b (b x))
            | false -> x;;
```

- type is

$$skwhile : (\alpha \to bool) \to (\alpha \to \alpha) \to \alpha \to \alpha$$

Skeleton framework

- Grammar for the skeletons:



```
Skel ::= StreamParallelSkel |
         DataParallelSkel | ControlParallelSkel
StreamParallelSkel ::= Farm | Pipe | ...
DataParallelSkel ::= Map | Reduce |
                     ParallelPrefix | ...
ControlParallelSkel ::= Seq | Conditional |
                        DefiniteIter |
                        IndefiniteIter | ...
Seq ::= seq(<sequential function code wrapping>)
Farm ::= farm(Skel)
Pipe ::= pipeline(Skel,Skel)
Map ::= map(Skel)
Reduce ::= reduce(Skel)
ParallelPrefix ::= parallel_prefix(Skel)
Conditional ::= ifthenelse(Skel,Skel,Skel)
DefiniteIter ::= ...
...
```

API

- Example: Muskel

```
        Skeleton filter = new Filter(...);
        Skeleton recon = new Recognize(...);
        Skeleton main = Pipeline(filter, recon);

Manager mgr =
        new Manager(main,
                new FileInputStreamManager("datain.dat"),
                        new ConsoleOutputStreamManager());
mgr.start();
```

SkeTo

- the skeleton framework may provide library calls directly implementing the different skeletons.

- Example, in frameworks such as SkeTo the skeletons are declared and executed via suitable library calls: to use a map skeleton followed by a reduce skeleton the user may write a code such as

```
dist_matrix<double> dmat(&mat);
dist_matrix<double> *dmat2 =
matrix_skeletons::map(Square<double>(), &dmat);
double *ss =
matrix_skeletons::reduce(Add<double>(),
Add<double>(), dmat2);
```

Skeleton Nesting

- In the initial form proposed by Cole nesting was not possible

⇒ Lead CISC skeletons library

Alternative:

“RISC" skeletons

- skeleton sets with a small number of items.
  - These skeletons modelled very basic parallelism exploitation patterns.
  - Each of the “code" parameters of the skeletons, however, could be recursively provided as skeletons.
  - The envisioned scenario behind this design choice allowed complex parallelism exploitation patterns to be defined ***as composition of primitive parallelism exploitation patterns***.
- In this case, the application programmer were required to know only a small set of base skeletons plus another small number of \composition rules".
- **composition** rules play a significant role in this case.