Design Patterns for Parallel Computation

Jim Browne

September 21, 2011

Lecture Content

- Concepts of Design Patterns
- Connection with Models of Computation Lecture
- Common Patterns of Parallelism (with some examples)
- Summary

Design Patterns: Definitions

- *Design Patterns* – A set of architecture level specifications for the logical units of a computation and their relationships which are common across multiple application types.
- Design Patterns – Are partial architectural specifications for software systems.

  http://www.sei.cmu.edu/architecture/
- Parallel Design Patterns – A set of architecture level specifications for the parallel structure of the execution behavior of the logical units of a computation which are common across multiple applications.

  Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders; "Patterns for Parallel Application Programs"; *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999.

Parallel Programming Pattern : WHY?

- Follows principle of separation of concerns
- Reuses expert knowledge
  - Shortens development time
- Leads to readable, modular, evolvable and efficent programs.

Multiple Approaches

- **Different authors propose multiple different but partially overlapping lists/sets of patterns**
- **Different authors sometimes give different names to the same pattern**
- **A given application may be classified into or be formulated in multiple patterns**

How To Use Parallel Design Patterns

- Follow the process described in the last lecture to identify the units of computation
- Identify the dependencies among the units of computation you have identified.
- Examine the catalogs of parallel design patterns for structures which match the dependencies in your application.
- Use the templates and examples for the appropriate patterns to structure and implement your code.

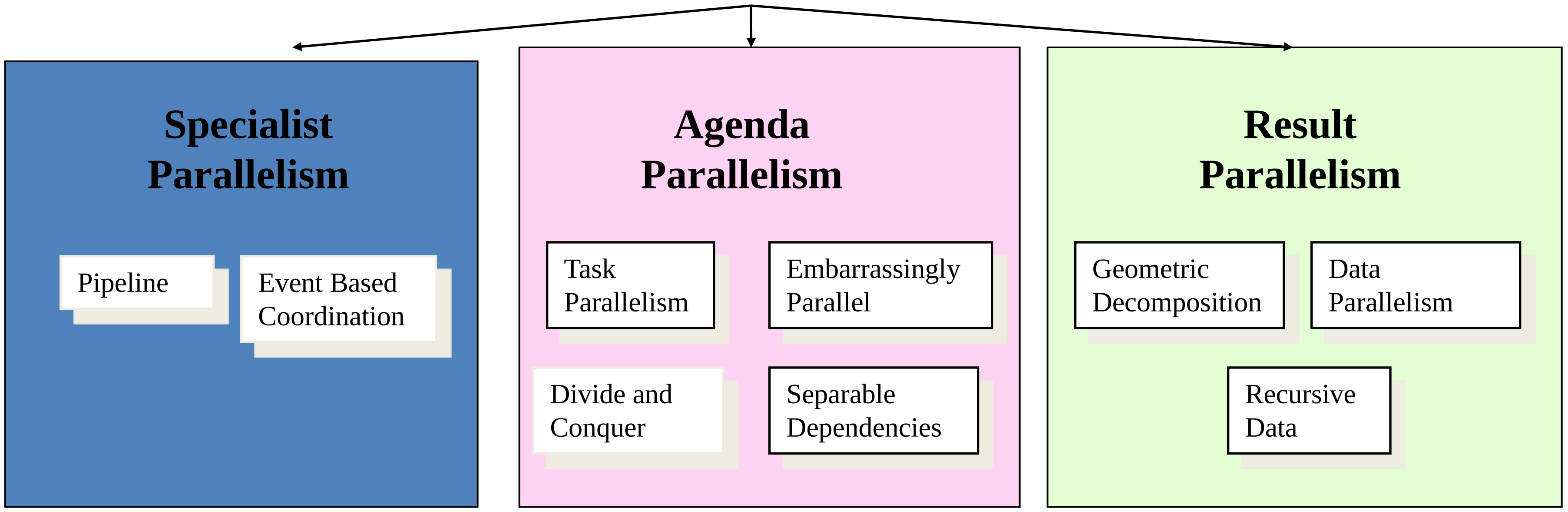
Common Parallel Patterns

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

The Algorithm Design Patterns: Massingill, Sanders and Mattson

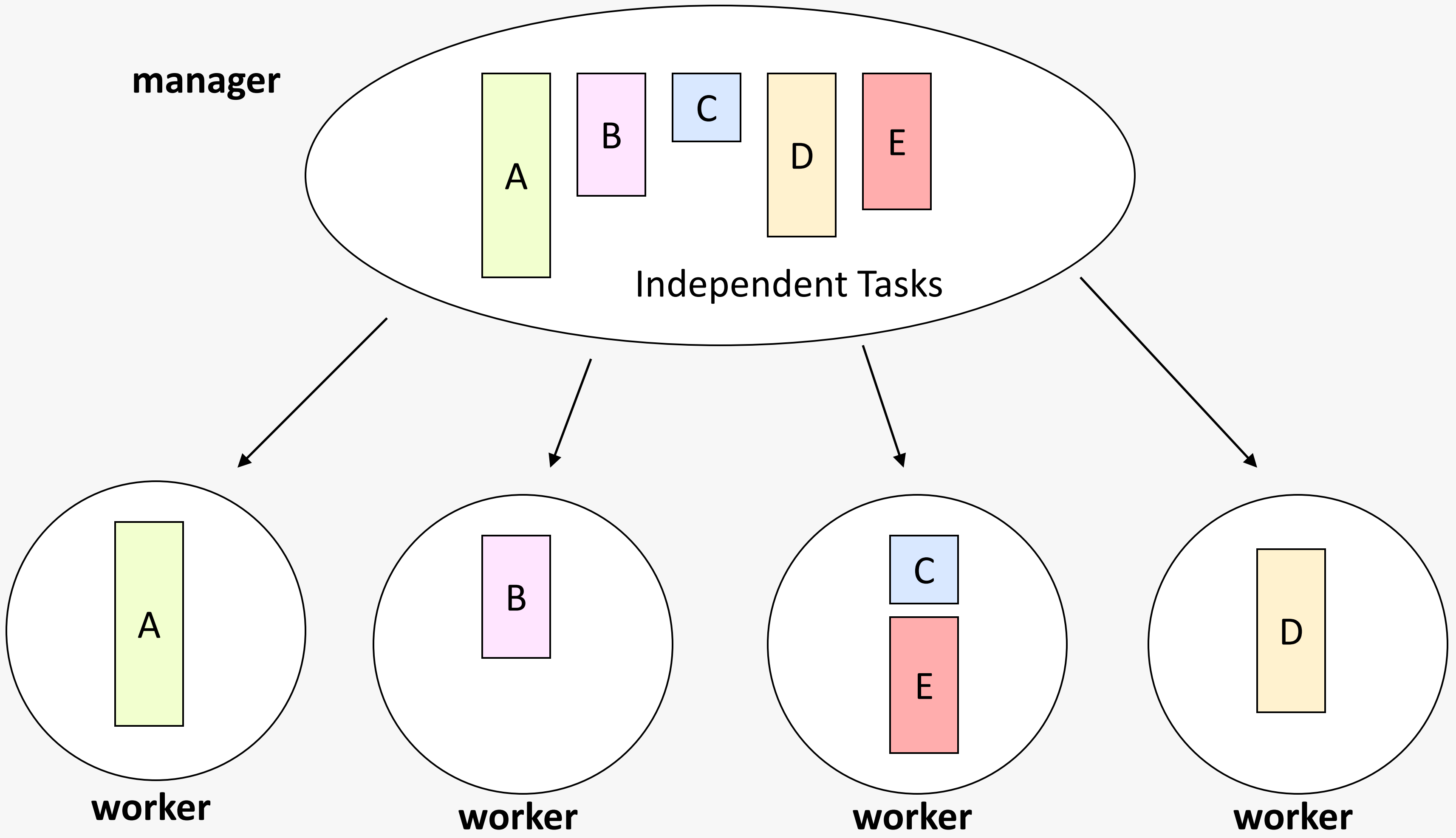
Start with a basic concurrency decomposition

- A problem decomposed into a set of tasks
- A data decomposition aligned with the set of tasks … designed to minimize interactions between tasks and make concurrent updates to data safe.
- Dependencies and ordering constraints between groups of tasks.

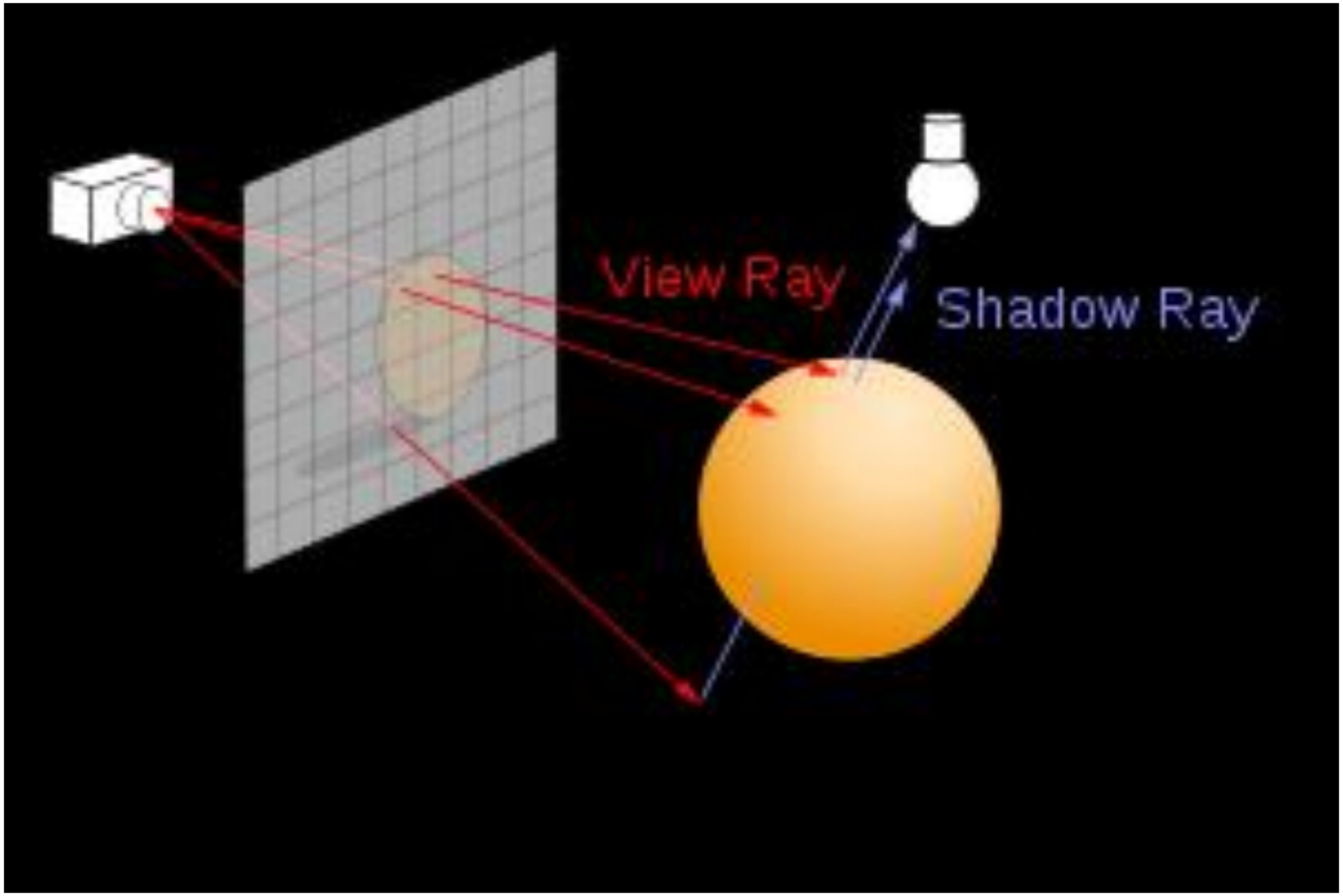


Embarassingly Parallel

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

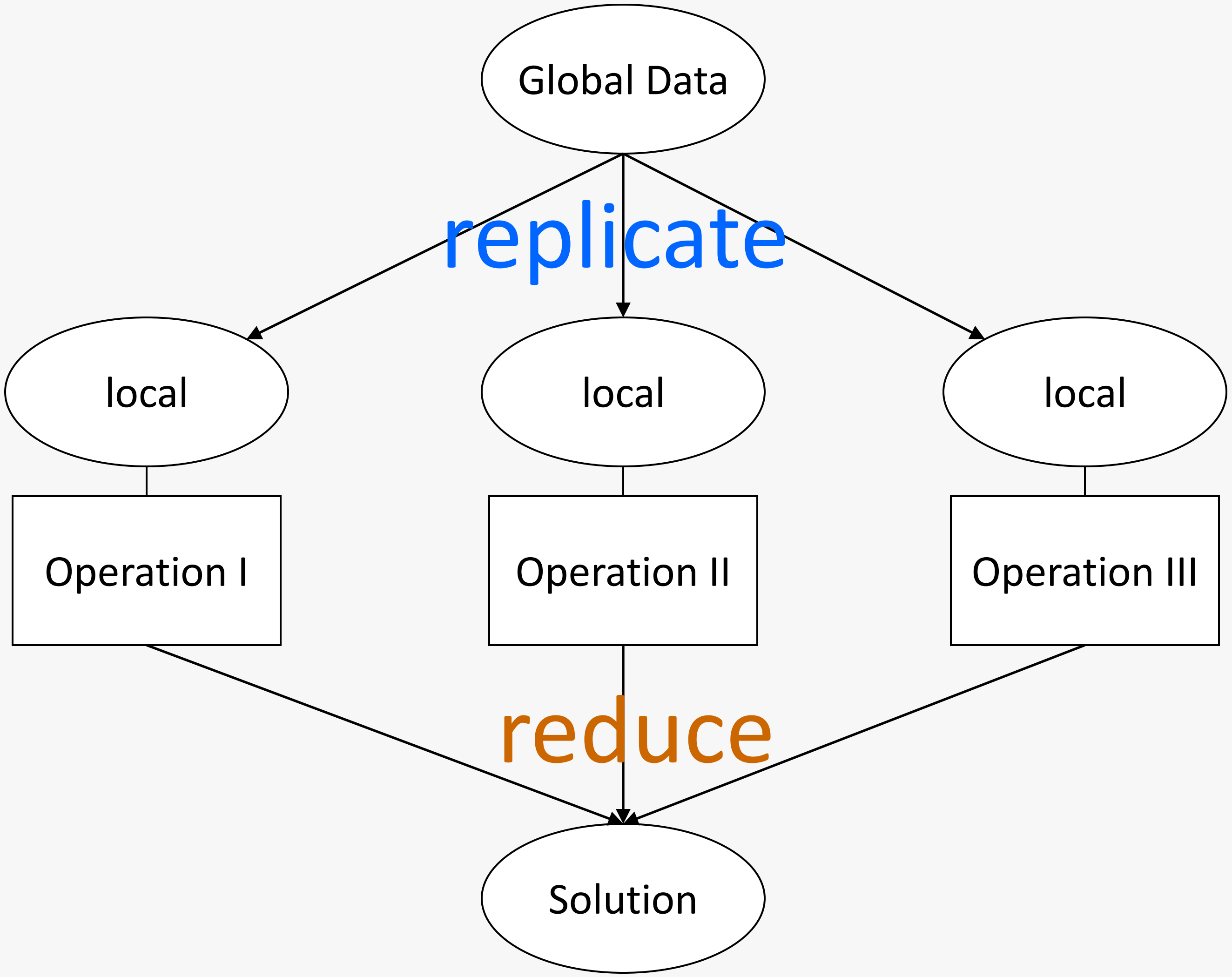


Example Ray Tracing

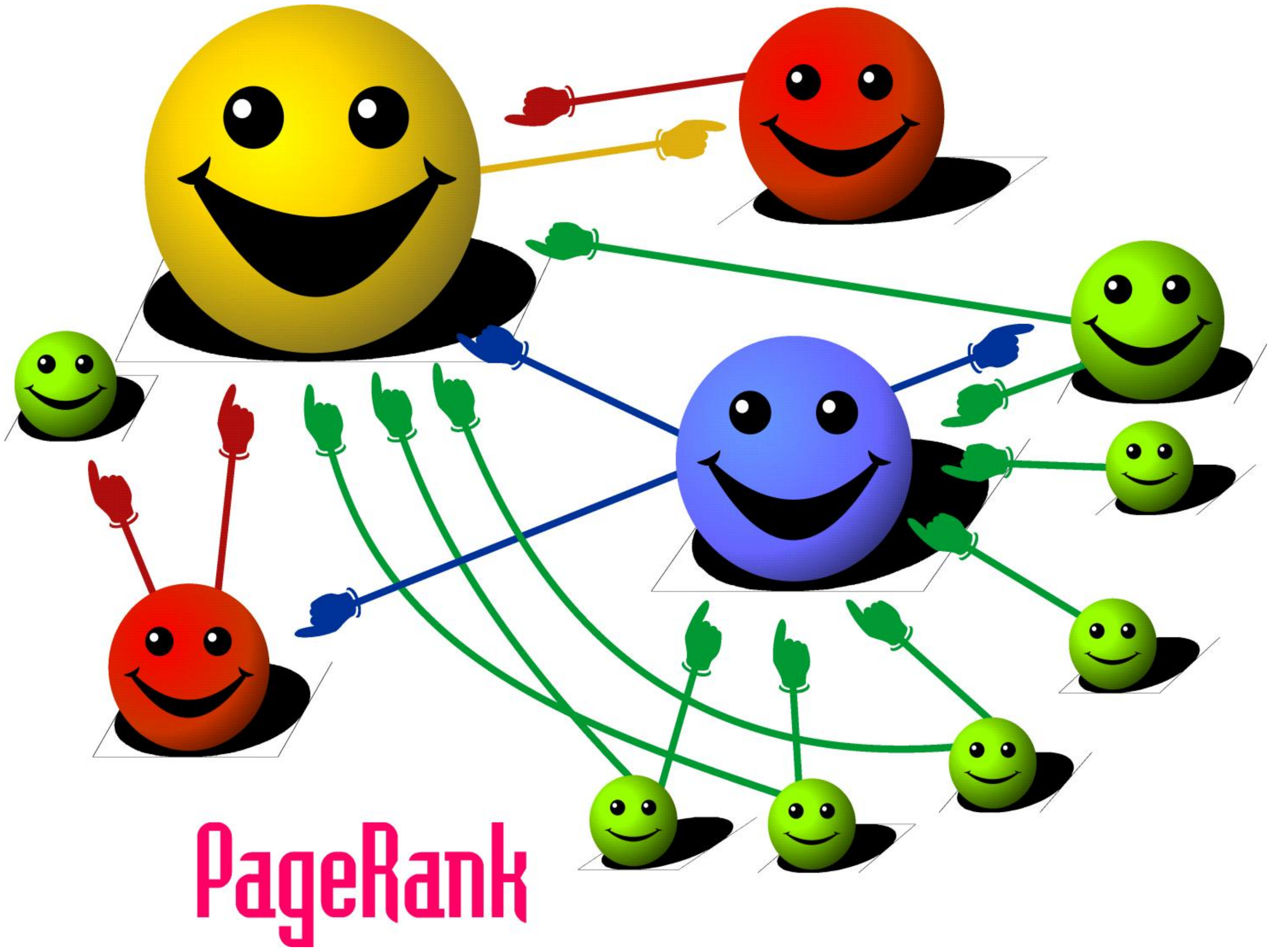


Replicable (Map Reduce)

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



PageRank

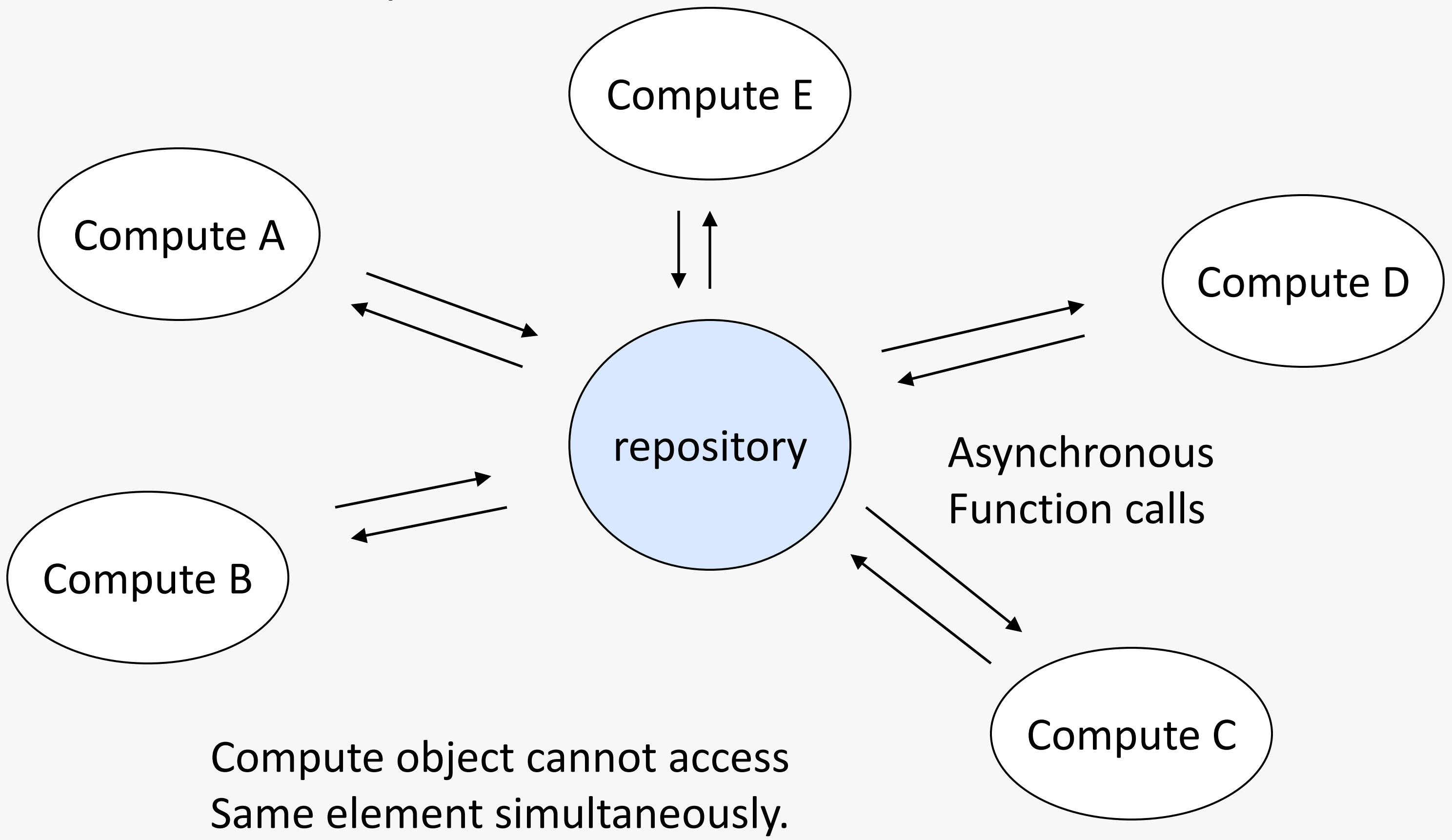


PageRank

- Simulates a “random-surfer”
- Begins with pair (URL, list-of-URLs)
- Maps to (URL, (PR, list-of-URLs))
- Maps again taking above data, and for each *u in list-of-URLs* returns *(u,* PR/|list-of-URLs|), as well as (*u, new-list-of-URLs)*
- Reduce receives (URL, list-of-URLs), and many (URL, value) pairs and calculates (URL, (new-PR, list-of-URLs))

Repository

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



Asynchronous
Function calls

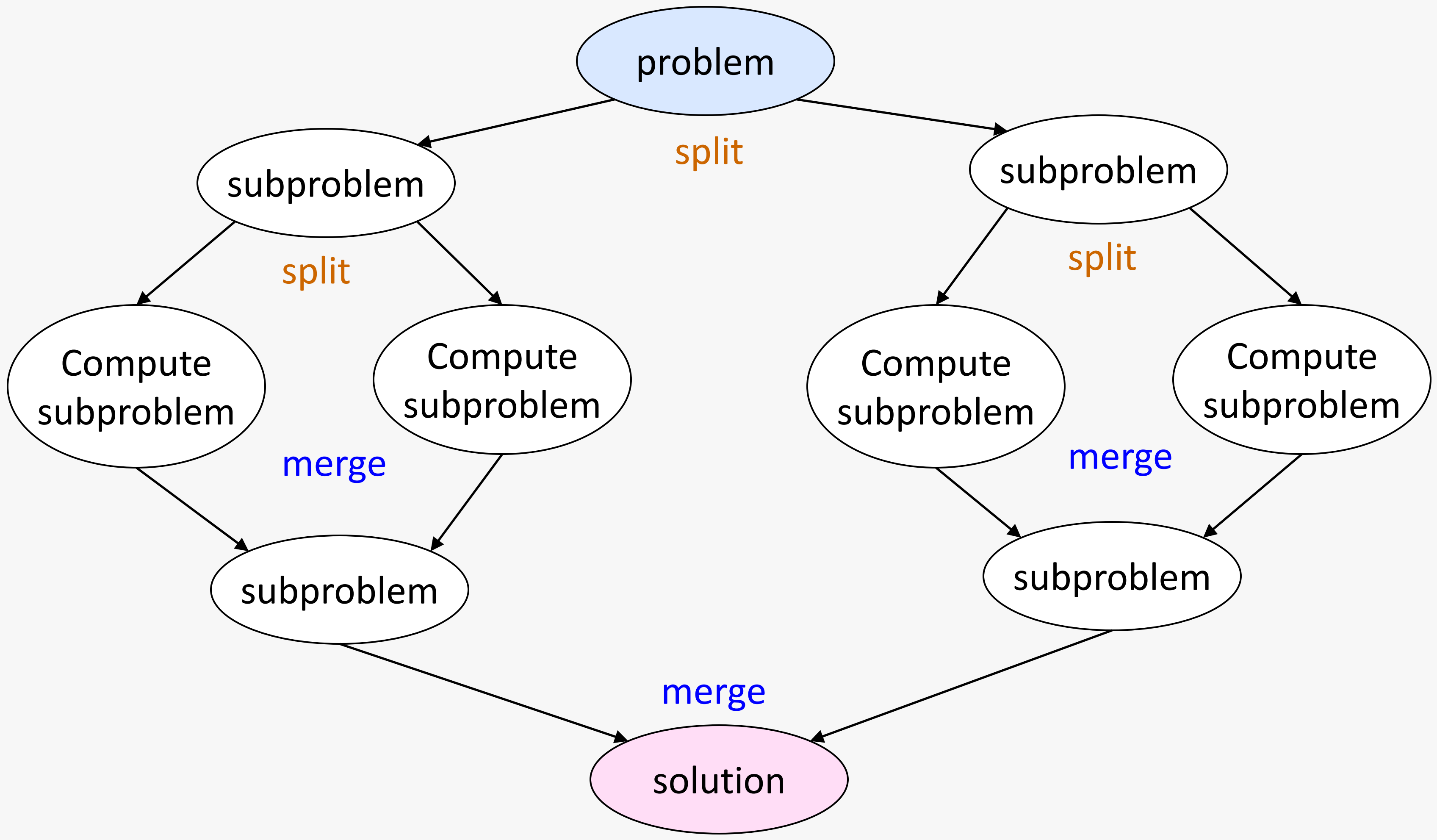
Compute object cannot access
Same element simultaneously.
(Repository controls access)

Example: Parallel Update of Data Base

- Online auctions
- Online merchants
- Etc.

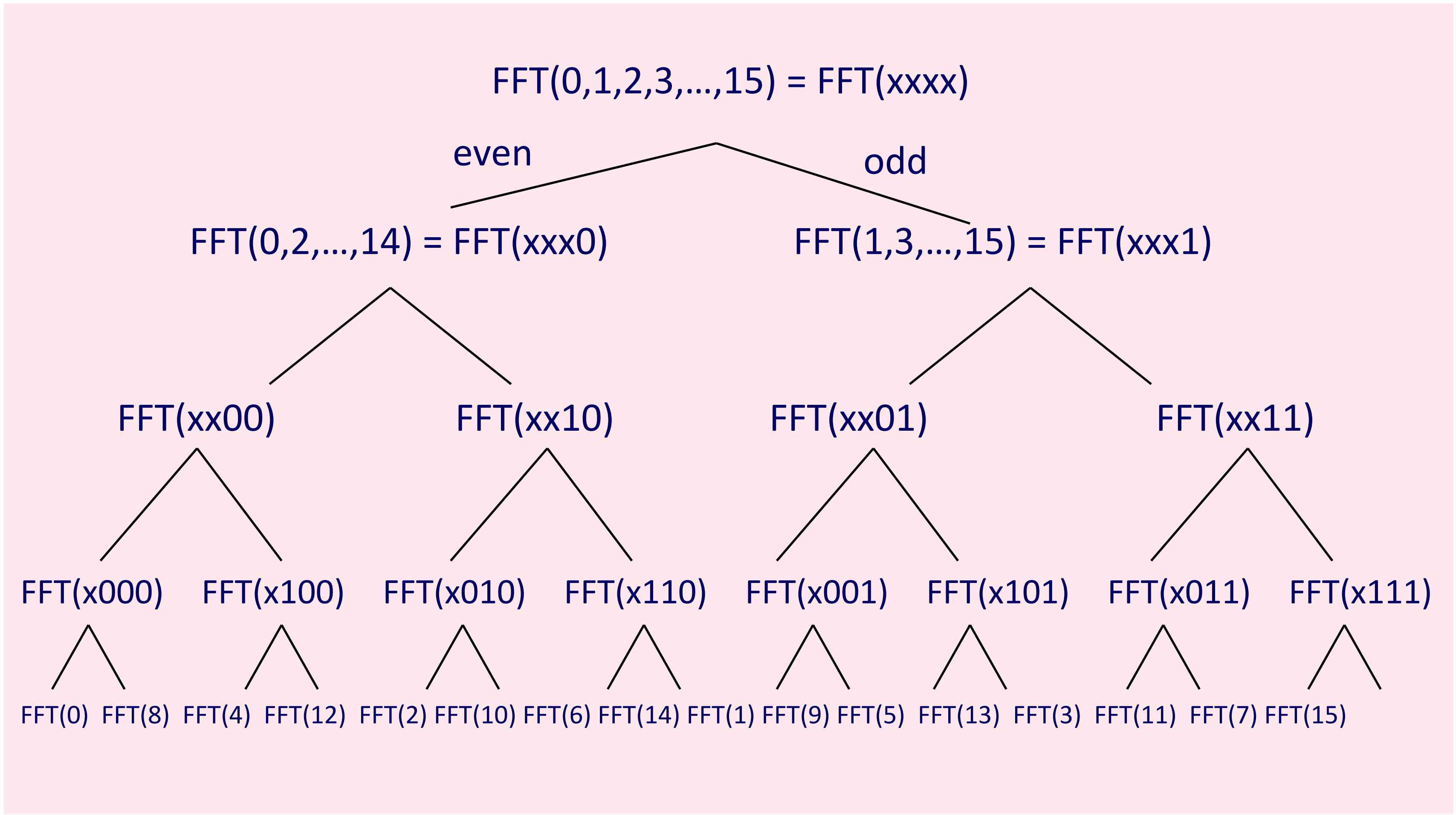
Divide & Conquer

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



* Split level needs to be adjusted appropriately.

Examples: FFT



Merge Sort

- Sorting: An important class of algorithms that take an input list and generate a sorted list.\
- Merge sort
  - Split a list in two
  - Sort each half by a call to merge sort
  - Continue until you his a trivial base case.
  - Unwind the recursive stack to generate final list
- Example
  - Starting point: [3 6 4 1 5 7 3 2]
  - Split in two: [3 6 4 1] [5 7 3 2]
  - Split in two: [3 6] [4 1] [5 7] [3 2]
  - Base case: [3 6] [1 4] [5 7] [2 3]
  - Sort on merge: [1 3 4 6] [2 3 5 7]
  - Sort on merge: [1 2 3 3 4 5 6 7]

Serial Merge Sort:

- Fill stack of recursive calls to merge sort, after base case (n<2) unwind the stack to generate sorted list)

```
void mergesort(int * X, int n, int * tmp)
{
    if (n < 2) return;

    /* recursively sort each half of list */
    mergesort(X, n/2, tmp);
    mergesort(X+(n/2), n-(n/2), tmp);



   /* merge sorted halves into sorted list */
   merge(X, n, tmp);
}
```

tmp points to space equal in size to X and is used as a buffer to sort into

Note: we include the merge function in a later slide … it's the same for both the serial and parallel cases.

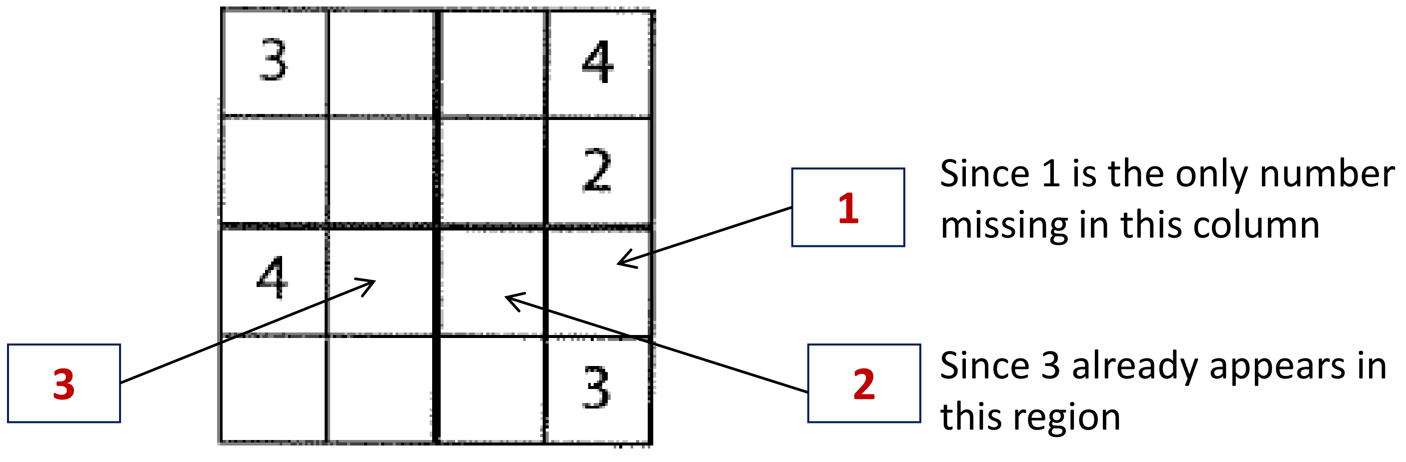
Parallel Merge Sort:

- Each mergesort is independent so parallel version is trivial to create.

```
void mergesort(int * X, int n, int * tmp)
{
    if (n < 2) return;

    /* recursively sort each half of list */
    fork mergesort(X, n/2, tmp);
    fork mergesort(X+(n/2), n-(n/2), tmp);

    join

   /* merge sorted halves into sorted list */
   merge(X, n, tmp);
}
```

Sudoko

- A game where you fill in a grid with numbers
  - A number cannot appear more than once in any column
  - A number cannot appear more than once in any row
  - A number can not appear more than once in any "region"
- Typically presented with a 9 by 9 grid … but for simplicity we'll consider a 4 by 4 grid


3
4
2
4
3
1
Since 1 is the only number missing in this column
2
Since 3 already appears in this region
3
Since 3 is the only number missing in this row

A 4 x 4 Sudoku puzzle with 11 open positions … we show three steps in the solution

Sudoko Algorithm

- The two-dimensional Sudoko grid is flattened into a vector
  - Unsolved locations are filled with zeros
  - The first two rows of the initial 4 x 4 puzzle are shown
  - The current working location [loc=0] is shown in red and the subgrid size is 3
  - Initially call `fork solve(size=3, grid, loc=0)`

grid →

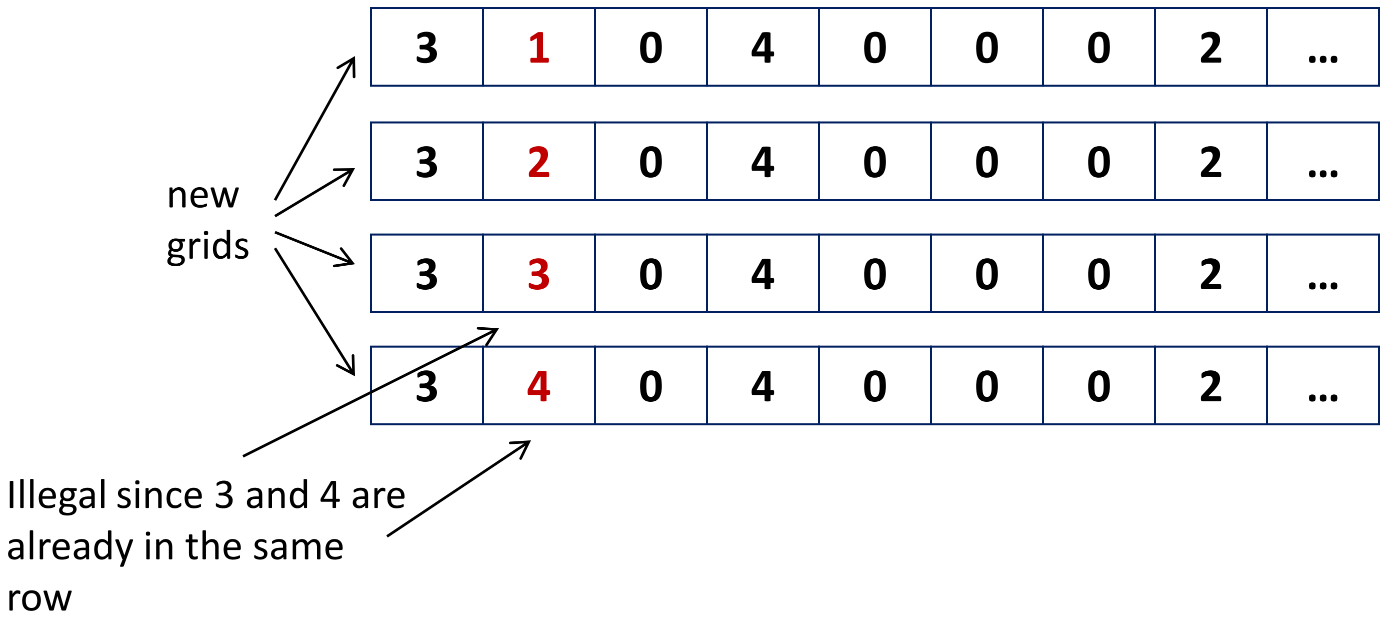
3	**0**	**0**	**4**	**0**	**0**	**0**	**2**	**…**

- The first location has a solution so move to next location
  - Recursively call `fork solve(size=3, grid, loc=loc+1)`

3	**0**	**0**	**4**	**0**	**0**	**0**	**2**	**…**

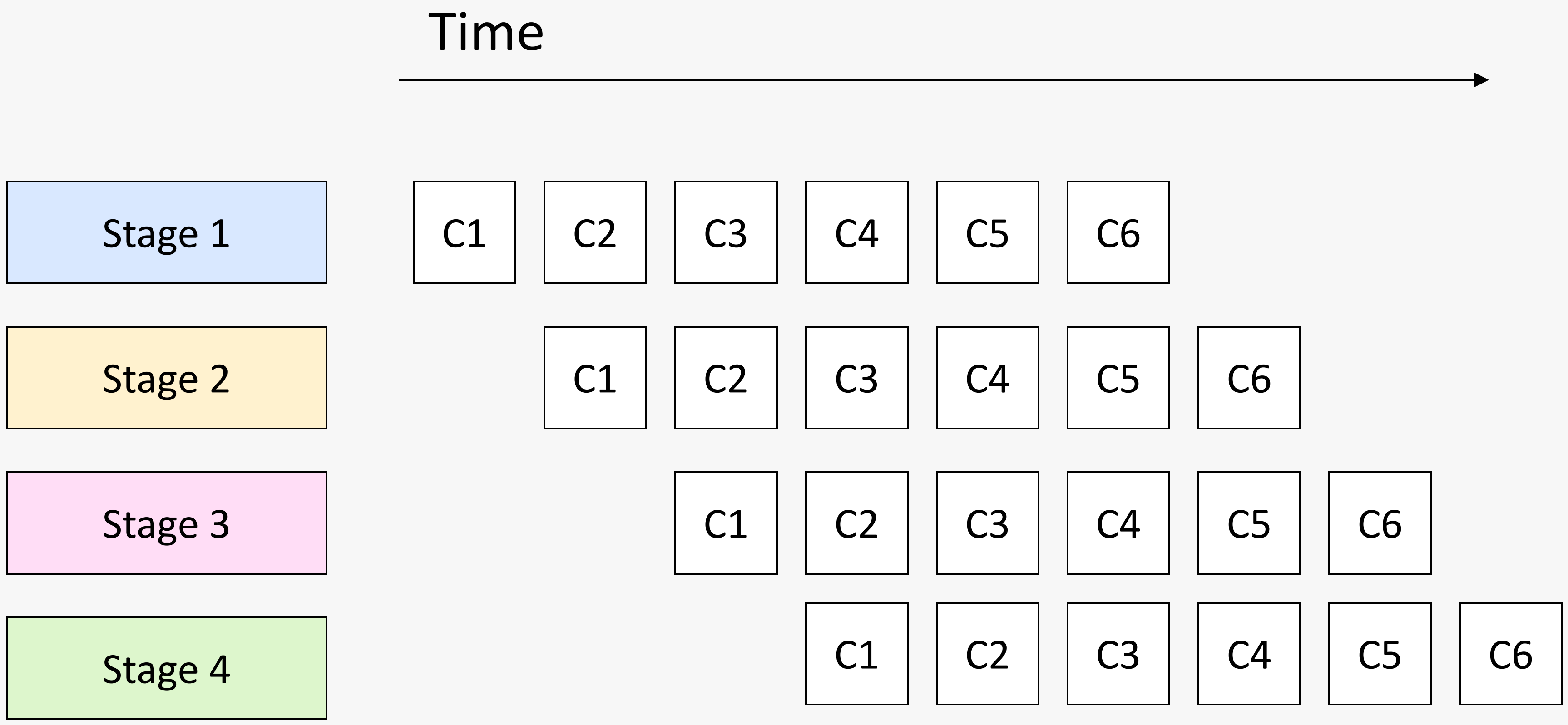
Exhaustive Search

- The next location [loc=1] has no solution (‘0’ in the current cell) so …
  - Create 4 new grids and try each of the 4 possibilities (1,2,3,4) concurrently
  - Note: the search goes much faster if the guess is first tested to see if it is legal
  - Spawn a new search tree for each guess k
  - Call: `fork solve(size=3, grid[k], loc=loc+1)`



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.



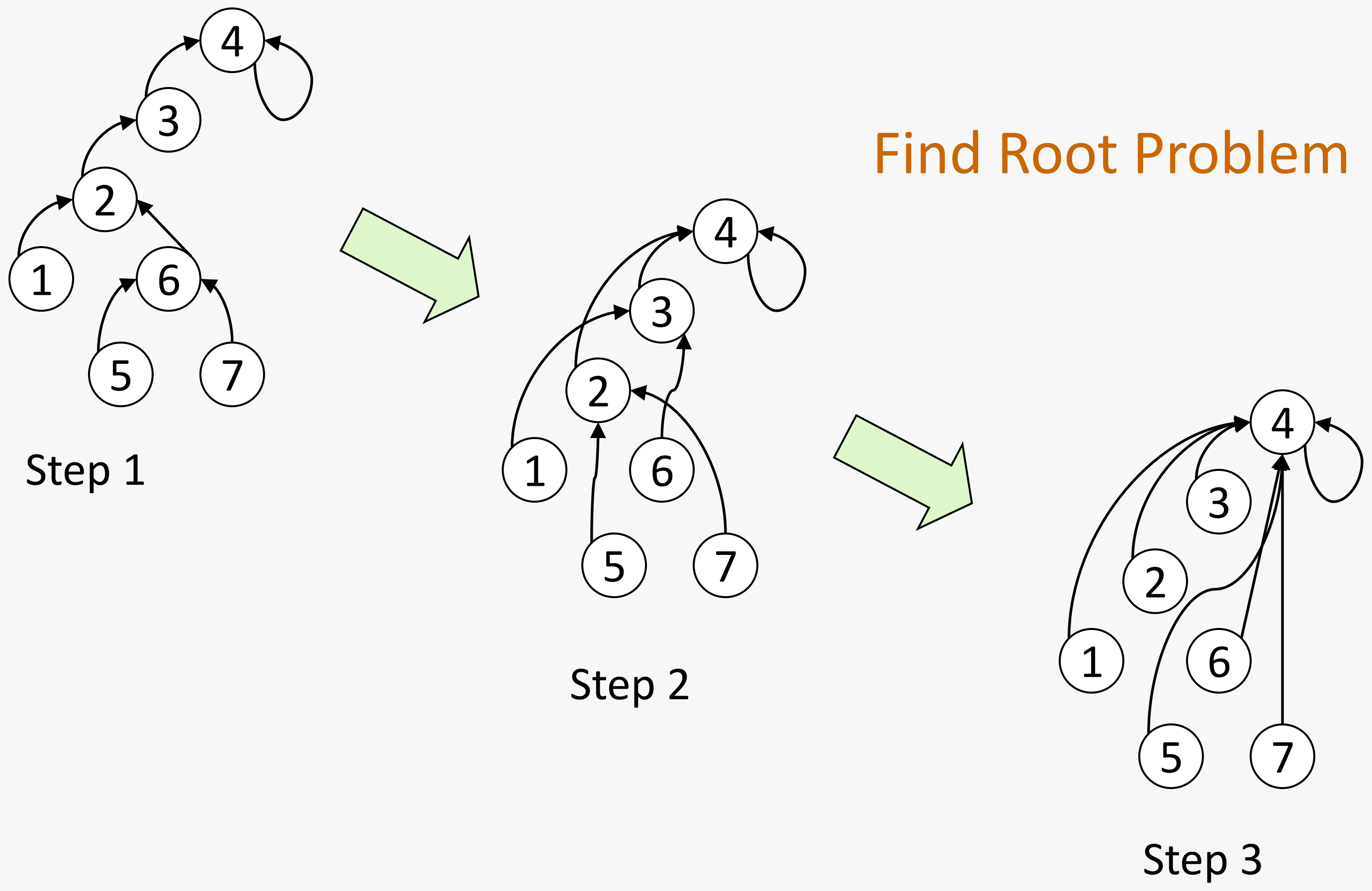
Example: Microlevel Execution of Instructions

- Stages of execution of instruction
- Fetch instruction
- Fetch data
  - Check L1 cache
  - Check L2 cache,
  - Etc.
- Execute instruction
- Store result in memory
  - Check for cache invalidation
  - Etc.

Recursive Data

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

Find Root Problem



Example: Graph Algorithm – Node Roots

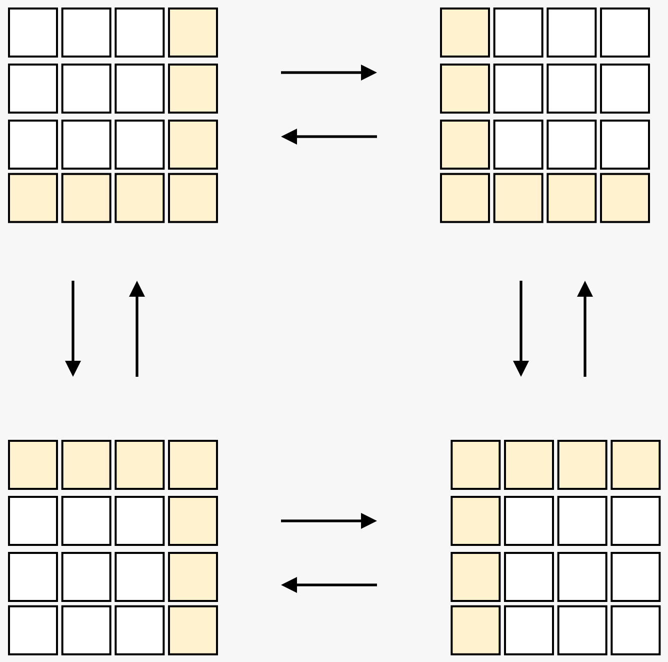
- Consider a forest of rooted directed trees (defined by specifying, for each node, its immediate ancestor, with a root node's ancestor being itself)
- compute, for each node in the forest, the root of the tree containing that node.
- Sequential program, we would trace depth-first through each tree from its root to its leaf nodes; as we visit each node, we have the needed information about the corresponding root. Total running time of such a program for a forest of N nodes would be O(N).
- There is some potential for concurrency (operating on subtrees concurrently), but there is no obvious way to operate on all elements concurrently, because it appears that we cannot find the root for a particular node without knowing its parent's root.

Example: Graph Algorithm – Node Roots

- Define for each node a "successor", which initially will be its parent and ultimately will be the root of the tree to which the node belongs.
- Calculate for each node its "successor's successor".
- For nodes one "hop“ from the root, this calculation does not change the value of its successor (because a root's parent is itself).
- For nodes at least two "hops" away from a root, this calculation makes the node's successor its parent's parent. We repeat this calculation until it converges (that is, the values produced by one step are the same as those produced by the preceding step), at which point every node's successor is the desired value.
- At each step we can operate on all N nodes in the tree concurrently, and the algorithm converges in at most log N steps.

Geometric

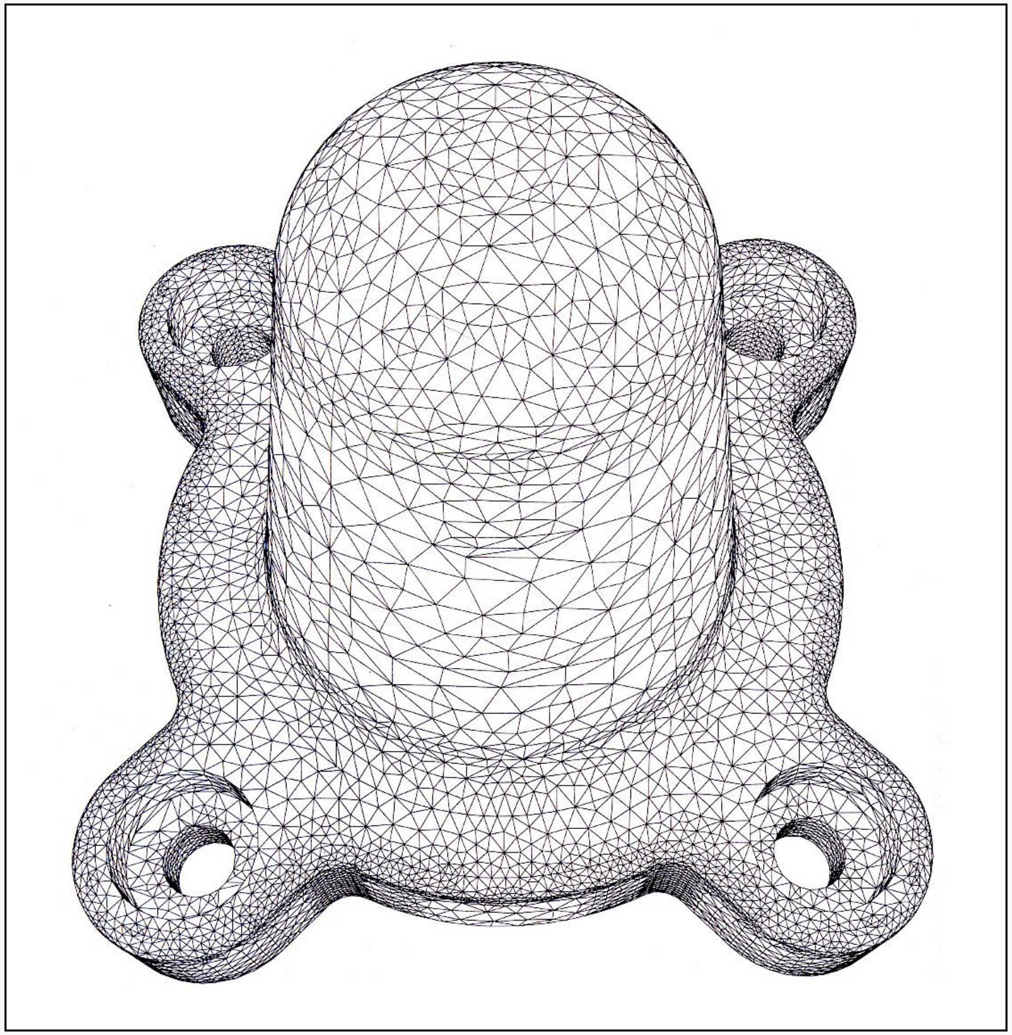
Dependencies exist, but communicate in predictable (geometric) neighbor-to-neighbor paths.



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.

Still Unrecovered

- The corrupted file has not yet been recovered.
- We will repost these notes with comments added to the currently missing slides as soon as I can recover or recreate the lost slides.