University of Central Florida

Patterns for Parallel Programming

School of Electrical Engineering and Computer Science
University of Central Florida

Textbook

T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005, ISBN 0-321-22811-1.

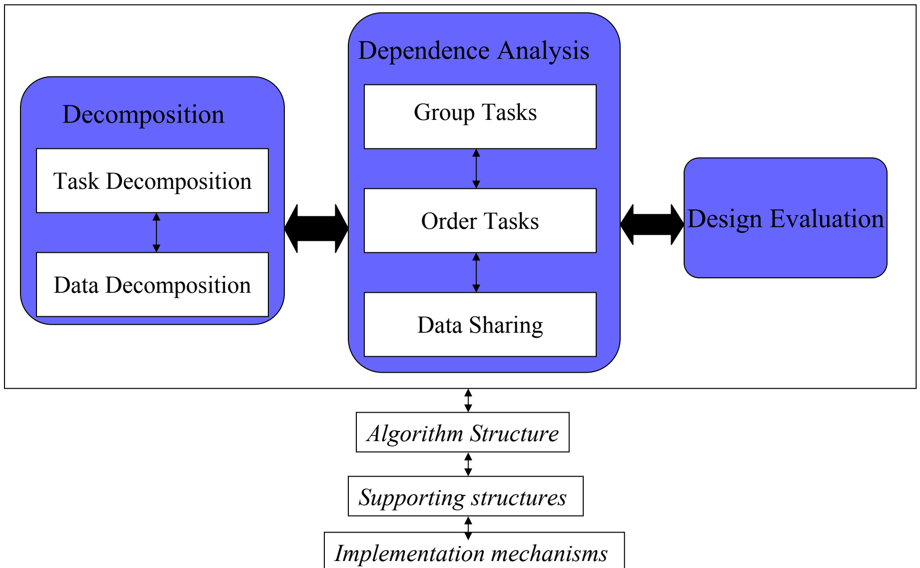
First of all

- Is the problem large enough and the results significant enough to justify the effort to solve it faster?
- If so, what are the most computationally intensive parts? Whether speeding them up provides sufficient performance gains (i.e., Amdahl's law)?

Overview

Finding Concurrency	Exposing exploitable concurrency
↕	
Algorithm Structure	Structuring algorithm to take advantage
↕	
Supporting structures	Structuring program and shared data
↕	
Implementation mechanisms	Mapping to particular programming environment

Finding Concurrency



Decomposition Patterns

- Task decomposition: view problem as a stream of instructions that can be broken into sequences called tasks that can execute in parallel.
  - Key: Independent operations
- Data decomposition: view problem from data perspective and focus on how the can be broken into distinct chunks
  - Key: Data chunks that can be operated upon independently
- Task and data decomposition imply each other. They are different facets of the same fundamental decomposition

Example

- Matrix multiplication
  - Task decomposition
    - Considering the computation of each element in the product matrix as a separate task
    - Performs poorly => group tasks pattern
  - Data decomposition
    - Decompose the product matrix into chunks, e.g., one row a chunk, or a small submatrix (or block) per chunk

Dependency analysis patterns

- Group tasks: group tasks that have the same dependency constraints; identify which tasks must execute concurrently
  - Reduced synchronization overhead – all tasks in the group can use a barrier to wait for a common dependence
  - All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shard Memory)
  - Grouping and merging dependent tasks into one task reduces need for synchronization
- Order task pattern: identifying order constraints among task groups.
  - Control dependency: Find the task group that creates it
  - Data dependency: temporal order for producer and consumer relationship

Dependency analysis patterns

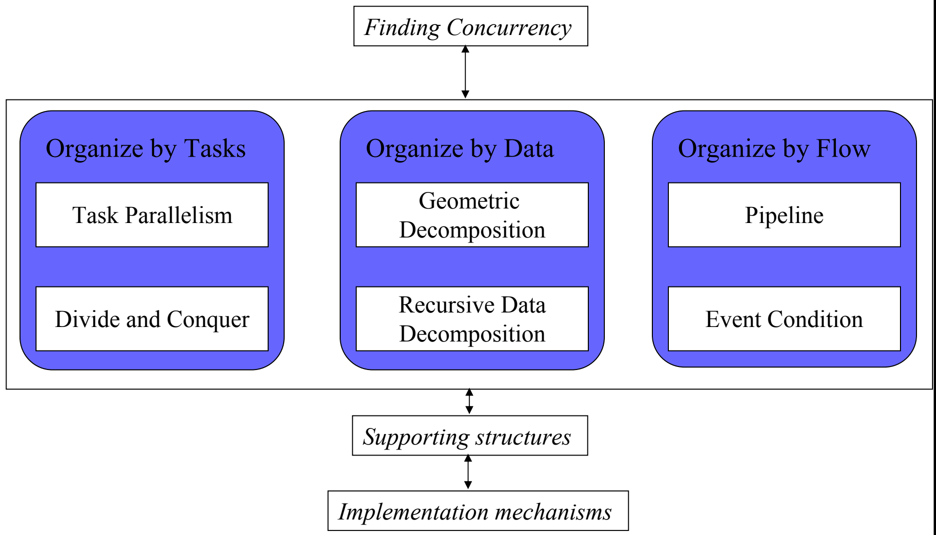
- Data sharing pattern: how data is shared among the tasks?
  - Read only: make own local copies
  - Effectively local: the shared data is partitioned into subsets, each of which is accessed (for read or write) by only one task a time.
  - Read-write: the data is accessed by more than one task. Need exclusive access mechanisms.
  - Example: the use of the shared memory among threads in a thread block.

Design Evaluation Pattern

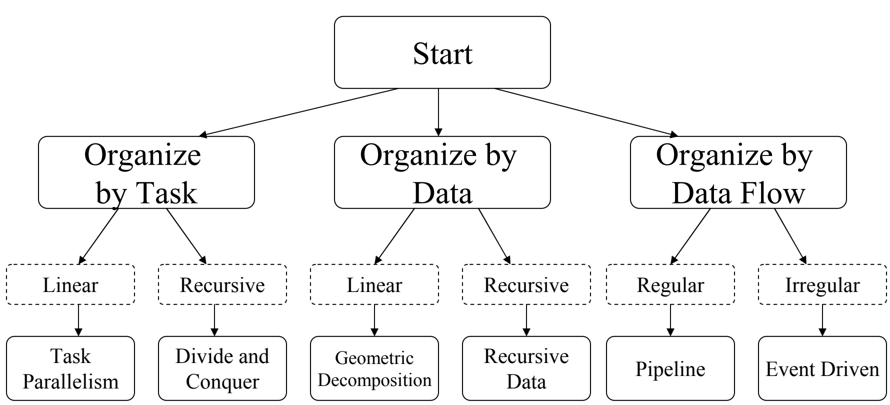
- Whether the partition fits the target hardware platform?

- Key questions to ask
  - How many threads can be supported?
  - How many threads are needed?
  - How are the data structures shared?
  - Is there enough work in each thread between synchronizations to make parallel execution worthwhile?

Algorithm Structure

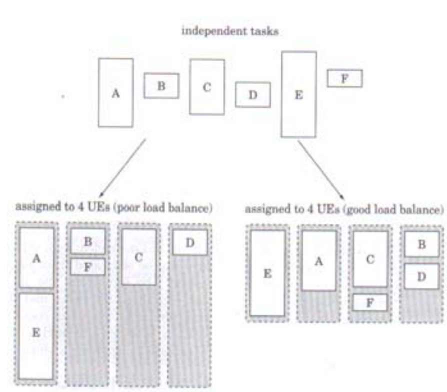


Organizing Principle



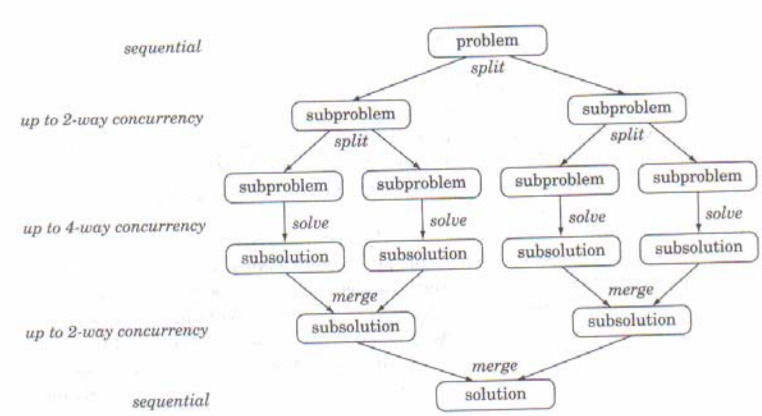
Task Parallelism Pattern

- After the problem is decomposed into a collection of tasks that can execute concurrently, how to exploit this concurrency efficiently?
- Load balancing



Divide and Conquer Pattern

- If the problem is formulated using the sequential divide-and-conquer strategy, how to exploit the potential concurrency?



Divide-and-Conquer Pattern

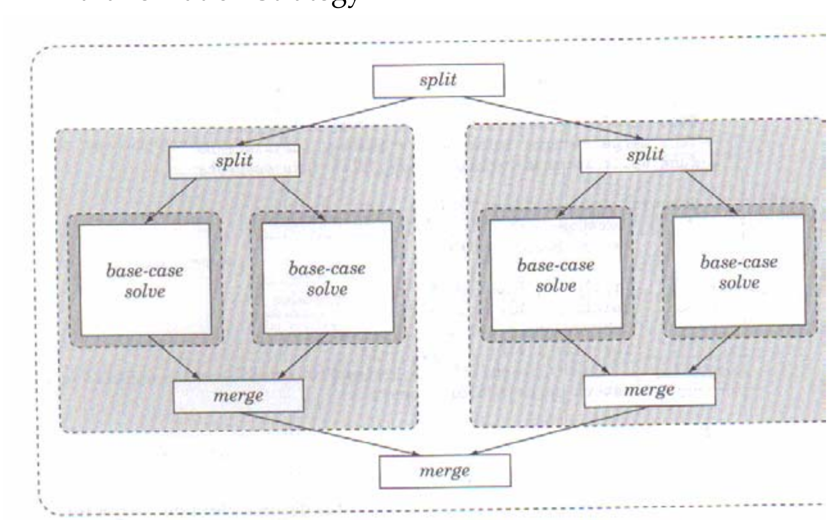
- Sequential code

```
func solve returns Solution; // a solution stage
func baseCase returns Boolean; // direct solution test
func baseSolve returns Solution; // direct solution
func merge returns Solution; // combine subsolutions
func split returns Problem[]; // split into subprobs

Solution solve(Problem P) {
    if (baseCase(P))
        return baseSolve(P);
    else {
        Problem subProblems[N];
        Solution subSolutions[N];
        subProblems = split(P);
        for (int i = 0; i < N; i++)
            subSolutions[i] = solve(subProblems[i]);
        return merge(subSolutions);
    }
}
```


Divide-and-Conquer Pattern

- Parallelization Strategy



Geometric Decomposition Pattern

- How to organize the algorithm after the data has been decomposed into concurrently updatable chunks?

- Decomposition to minimize the data communication and dependency among tasks
- Care needs to be taken when update non-local data, e.g., exchange operations

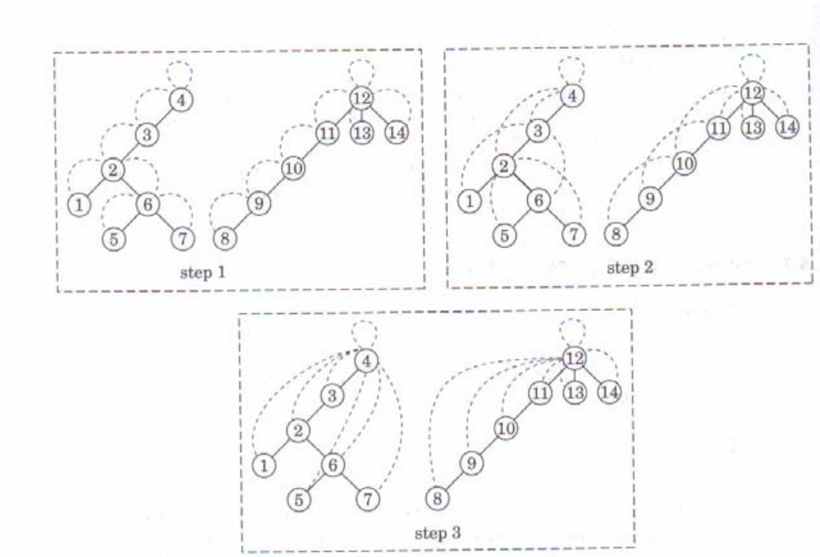
Recursive data pattern

- Suppose the problem involves an operation on a recursive data structure that appears to require sequential processing. How to make the operations on these data structures parallel?

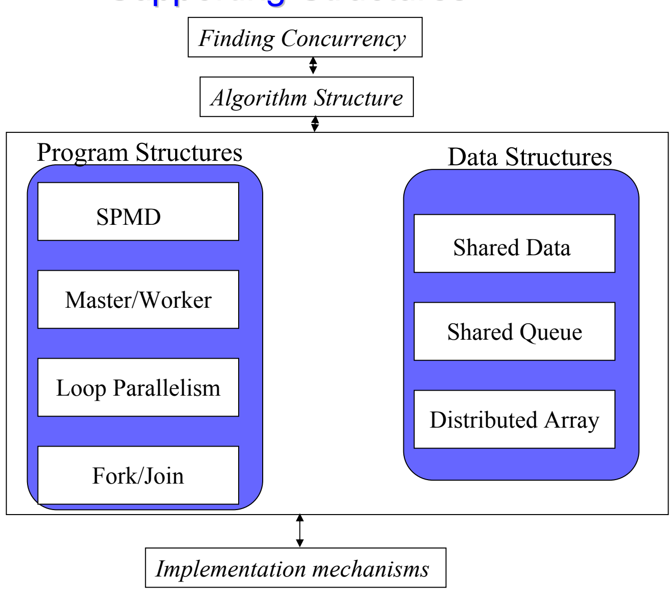
- Check whether divide-and-conquer pattern works

- If not, may need to transform the original algorithm.

Example: Finding root in a forest



Supporting Structures



Relationship between Supporting Program Structure Patterns and Algorithm Strcture Patterns

	Task Parallel.	Divide/Conquer	Geometric Decomp.	Recursive Data	Pipeline	Event-based
SPMD	☺ ☺ ☺ ☺	☺ ☺ ☺	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺	☺ ☺
Loop Parallel	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺			
Master /Worker	☺ ☺ ☺ ☺	☺ ☺	☺	☺	☺	☺
Fork/ Join	☺ ☺	☺ ☺ ☺ ☺	☺ ☺		☺ ☺ ☺ ☺	☺ ☺ ☺ ☺

Relationship between Supporting Program Structure Patterns and Programming Environment

	OpenMP	MPI	Java	Brook+/ CUDA	Cell
SPMD	☺ ☺ ☺	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺ ☺ ☺	☺ ☺ ☺ ☺
Loop Parallel	☺ ☺ ☺ ☺	☺	☺ ☺ ☺		☺ ☺ ☺
Master/ Slave	☺ ☺	☺ ☺ ☺	☺ ☺ ☺		☺ ☺ ☺ ☺
Fork/Join	☺ ☺ ☺		☺ ☺ ☺ ☺		☺ ☺

Implementation Mechanisms