HPC - From applications to tasks

Francieli ZANON-BOITO

November 2019

Sources

- Slides from "Introduction to Parallel Computing", by Allen Malony et al. from the University of Oregon
- Slides by George Em Karniadakis, from Brown University
- Slides by Massimiliano Guarrasi, from CINECA
- Slides by Andrea Marongiu
- Slides by Alexandre David, from Aalborg University
- Slides by Anshul Gupta, from IBM Research

HPC - From applications to tasks

Francieli ZANON-BOITO

November 2019

The HPC environment

- Large **supercomputers** composed of multiple **nodes**
- Nodes composed of multiple processors
- Processors composed of multiple cores and accelerators (e.g. GPUs)
- **Network** topology (e.g. fat tree, dragonfly, …)
- **Memory** hierarchy with multiple levels of cache (shared or not)
- Shared **storage** infrastructure: multiple data servers, intermediate I/O nodes

This talk

So far: discussed how to schedule jobs and tasks

But what are jobs and tasks? Where do they come from? What do they eat?

Applications

- Job: instance of **application**
  - An user submits an application to be executed with some parameters
  - Uses a set of resources
- Application to solve a **large problem**
  - Examples: climate/seismic/molecules/brain simulations
  - Separated into **smaller problems, the tasks**

Partitioning

- Application -> set of smaller tasks
  - **Decompose** the computation of the problem and its data
- Goal: to promote **parallelism**
- Try to avoid redundant computation/storage access
- Domain vs. functional decomposition

Domain decomposition

- Domain/data decomposition: focus on the **data**
  - Partition the data associated with the problem
  - **Each task will deal with a part** of the data
  - Associate computations to each piece of data
  - Some operations require data from more than one partition, so communicate
- "Data": input/output/intermediate data
  - In general we focus on the largest data structure, or the most frequently used

Functional decomposition

- Functional decomposition: focus on the **computation**
  - Decompose computation to be performed
  - Then deal with data required to computation
- Ideally: both functional and data decomposition
  - Multi-physics simulations, overall functional + data decomposition for each component

Task decomposition example

Chess Program

- Each task evaluates all moves of a single piece (branch-and-bound)
- Small data (board position) can be replicated
- Dynamic load balancing required

Source: slides by Anshul Gupta

Level of parallelism

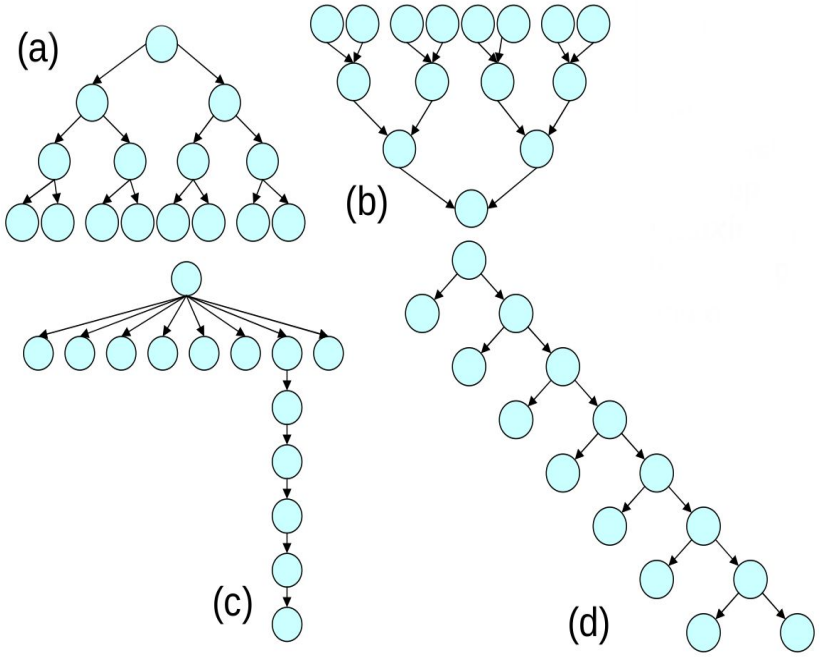
- We try to have more tasks than execution units
  - Flexibility when scheduling
- The **number of tasks should increase with the problem size**
  - If not, it will not **scale**, may not be able to solve larger problems
- Are tasks of similar size?
  - If not, **load imbalance**

Easiest scenario: EP

- EP = **embarrassingly parallel**
- Example: sensitivity analysis, Monte Carlo simulations
- Tasks run in parallel with little or no need for communication
- But most often we have task dependencies!

Task dependencies

- One task cannot start until the other finishes
- Traditionally represented as a **directed acyclic graph (DAG)**
- Concurrency: number of tasks that can run in parallel
- Critical path: longest directed path between start and finish nodes



Source: slides by Alexandre David

```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```

Source: slides by Allen Malony et al.

Communication

- Even independent tasks may need to **communicate**
- Represented as an interaction graph, or an affinity matrix

Communication

- **Local vs. Global**
  - Local: each task communicates with a small set of other tasks (neighbors)
  - Global: communicates with many or all tasks
- **Structured vs Unstructured**
  - Structured: tasks who communicate follow a regular structure
  - Unstructured: arbitrary communication graph
- **Static vs. Dynamic:**
  - Static: neighbors do not change over the execution
  - Dynamic: neighbors are determined at runtime and variable
- **Synchronous vs. Asynchronous:**
  - Synchronous: coordination between communicating tasks
  - Asynchronous: without coordination

Important aspects for communication

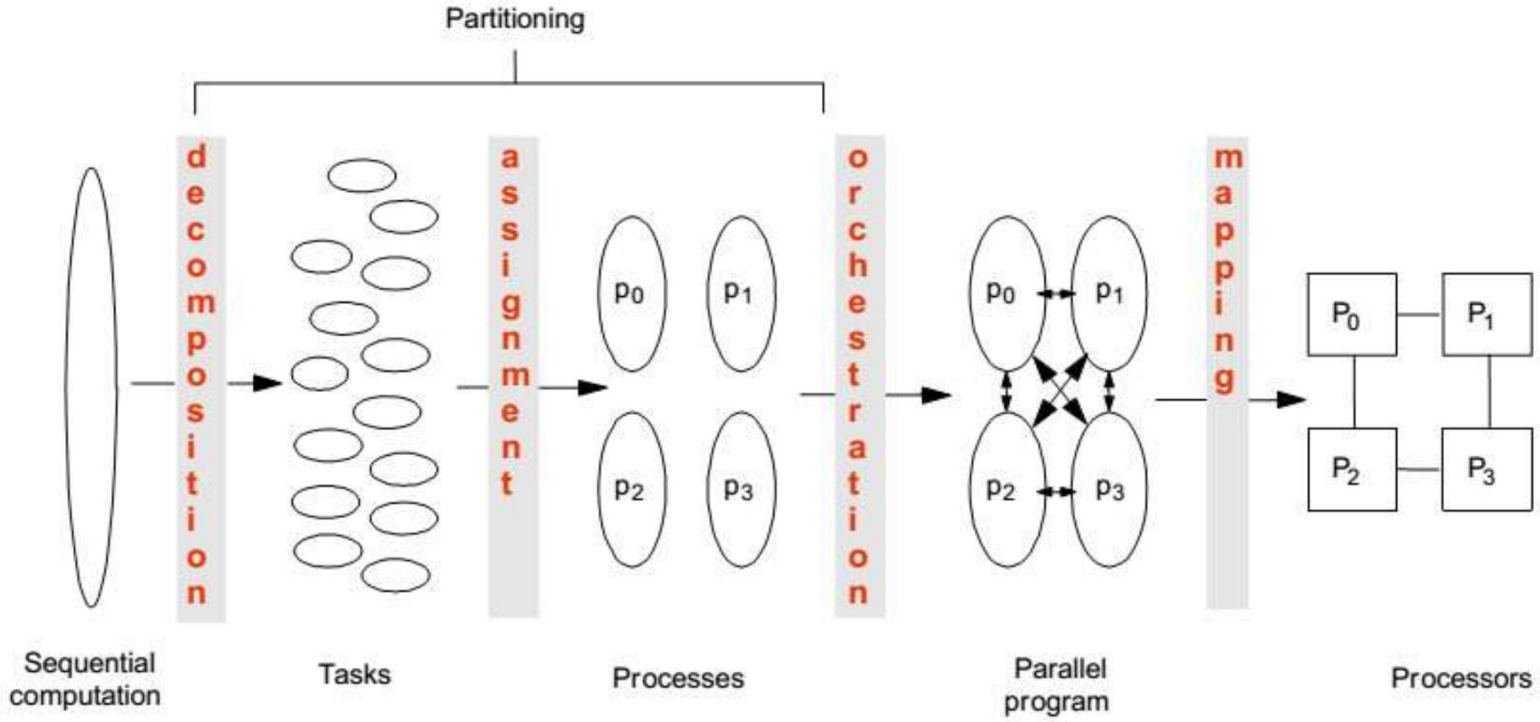
- Do all tasks perform a comparable number of communication operations?
- Can communications proceed concurrently?
- How to alleviate its impact?
  - Try to **overlap** communication and processing
  - Try to place tasks according to their **affinity**

Access to storage (I/O)

- How to alleviate its overhead:
  - Try to overlap I/O and processing
  - Use a "nice" access pattern
  - Sometimes the only way is to avoid it
- More on that later!

Common steps to creating a parallel program

Source: slides by Andrea Marongiu



Guidelines for Task Decomposition

- Algorithms start with a good understanding of the problem being solved
- Programs often naturally decompose into tasks
  - Two common decompositions are
    - Function calls and
    - Distinct loop iterations
- Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them

Guidelines for Task Decomposition

- Flexibility
  - Program design should afford flexibility in the number and size of tasks generated
    - Tasks should not tied to a specific architecture
    - Fixed tasks vs. Parameterized tasks
- Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn’t become the bottleneck
- Simplicity
  - The code has to remain readable and easy to understand, and debug

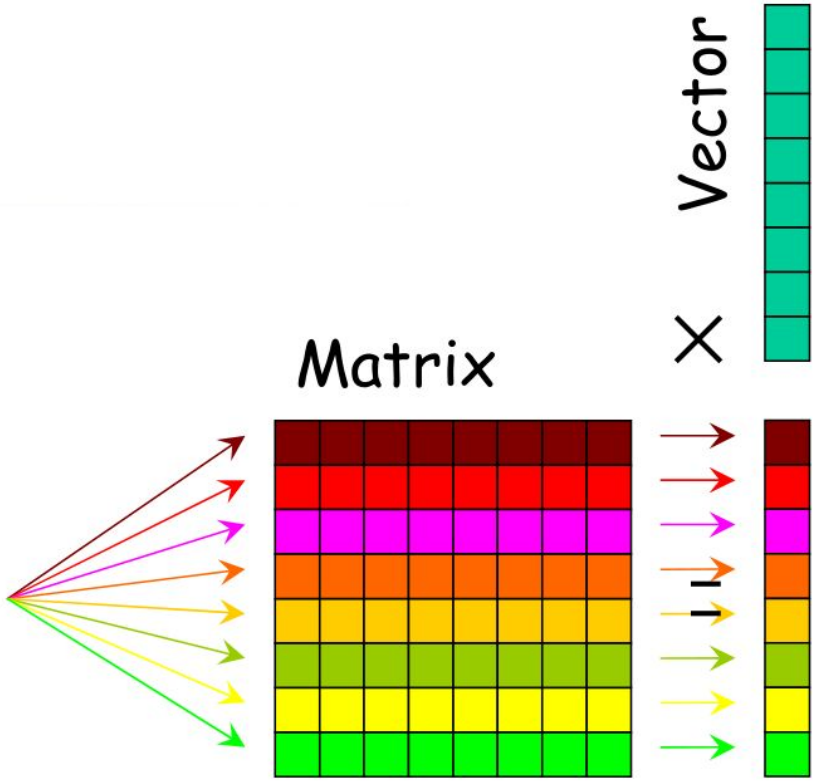
Granularity

- **Fine-grained:** many small tasks
  - Maximize concurrency
  - More flexibility for placement algorithms
  - Cannot go too far: overhead!
- **Coarse-grained:** fewer larger tasks
  - More locality, less communication
  - Surface-to-volume effects
    - Communication cost usually proportional to surface area of domain
    - Computation cost usually proportional to volume of domain
    - As task size increases, amount of communication per unit computation decreases
- Trade-off replication vs communication

Example: matrix * vector

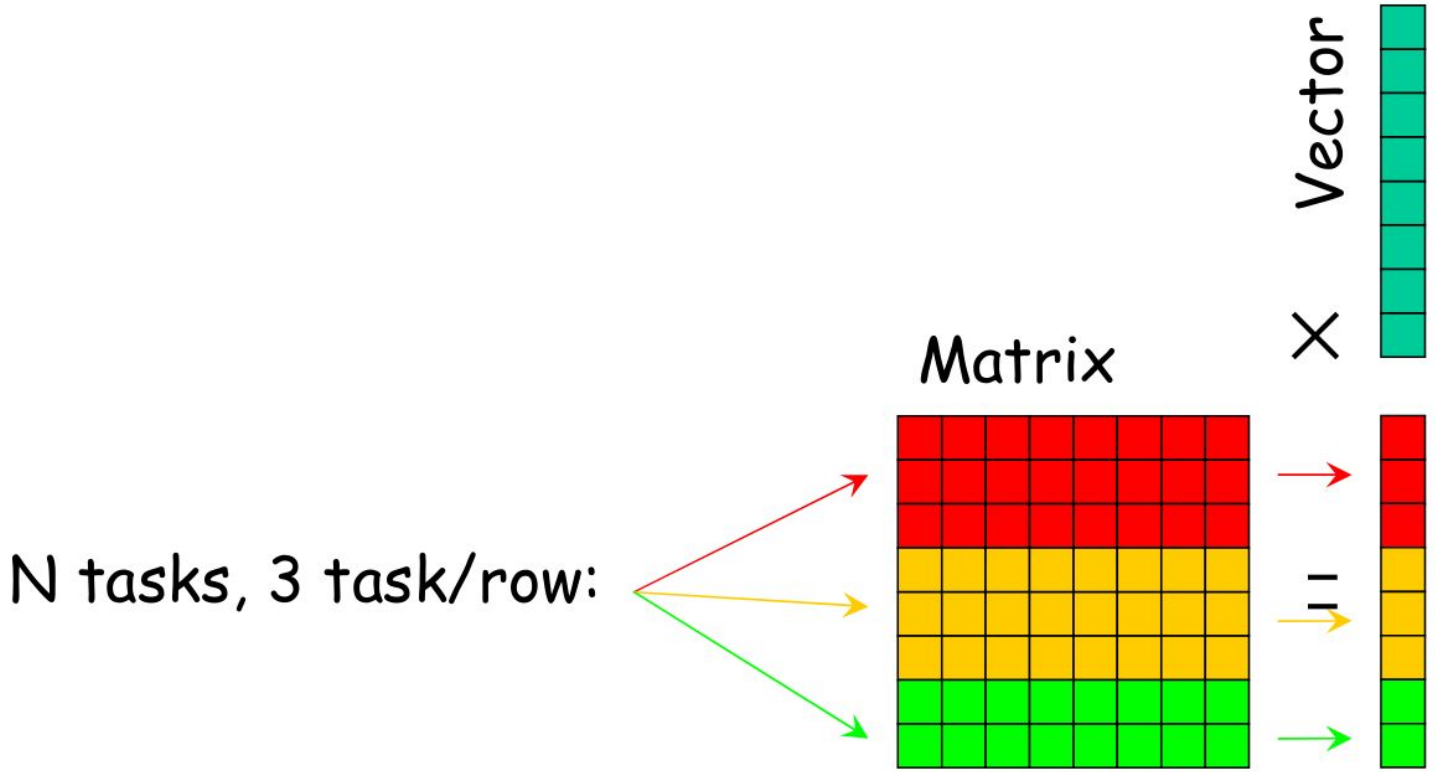
Source: slides by Alexandre David

N tasks, 1 task/row:



Example: matrix * vector

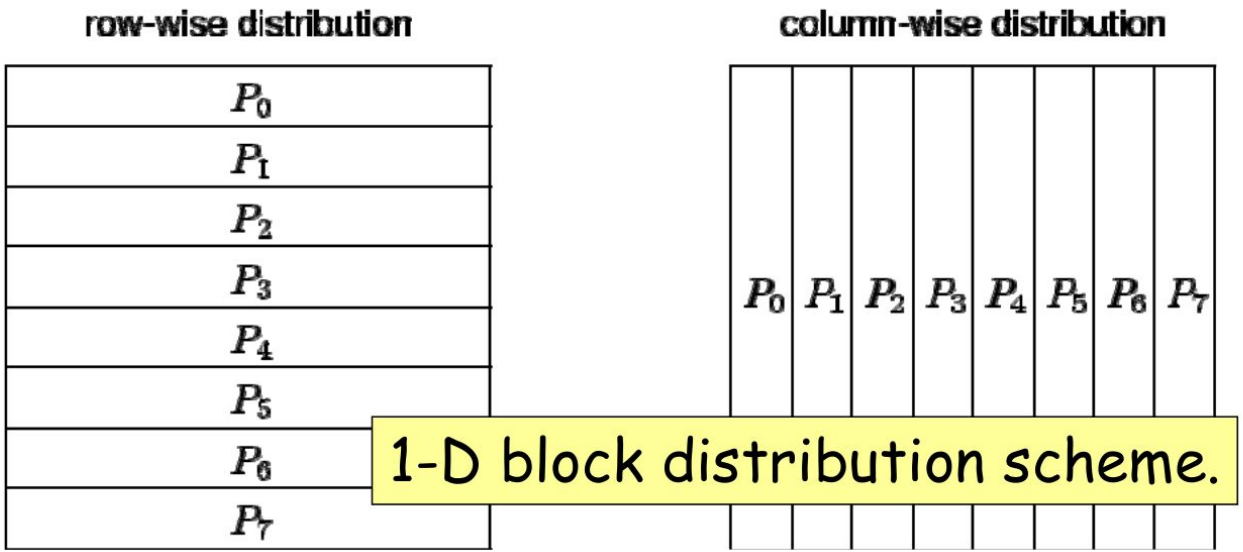
Source: slides by Alexandre David



Source: slides by Alexandre David

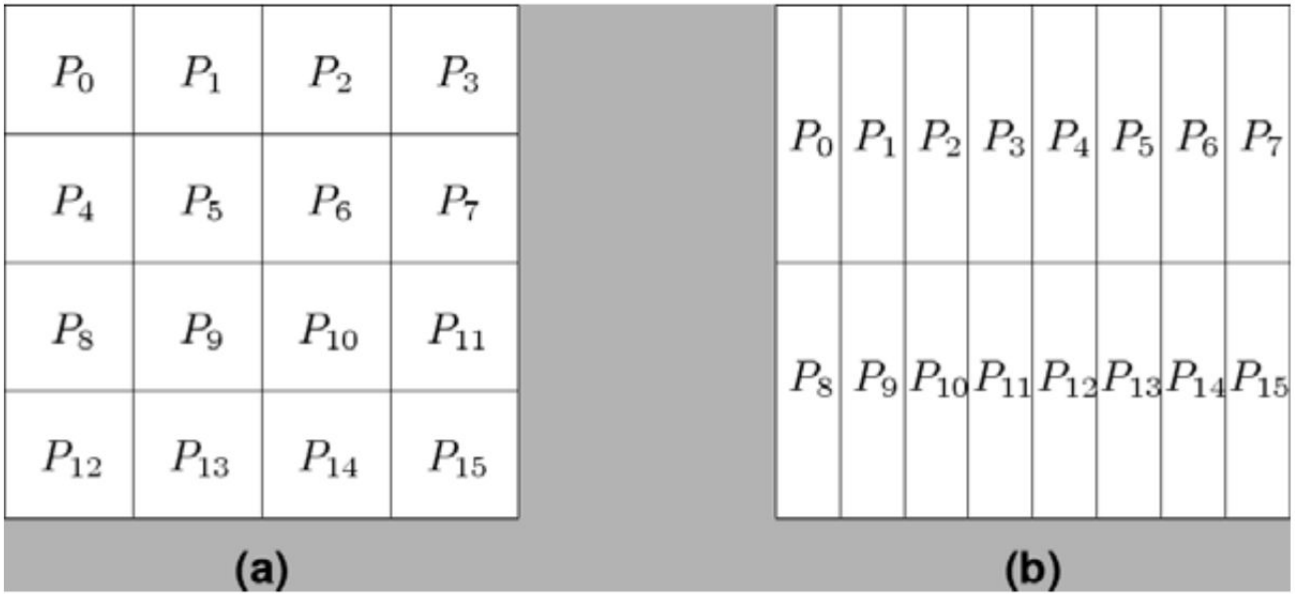
Array distribution scheme

- Combine with “owner computes” rule to partition into sub-tasks.



Source: slides by Alexandre David

Block distribution cont.



Generalize to higher dimensions: 4x4, 2x8.

Source: slides by Alexandre David

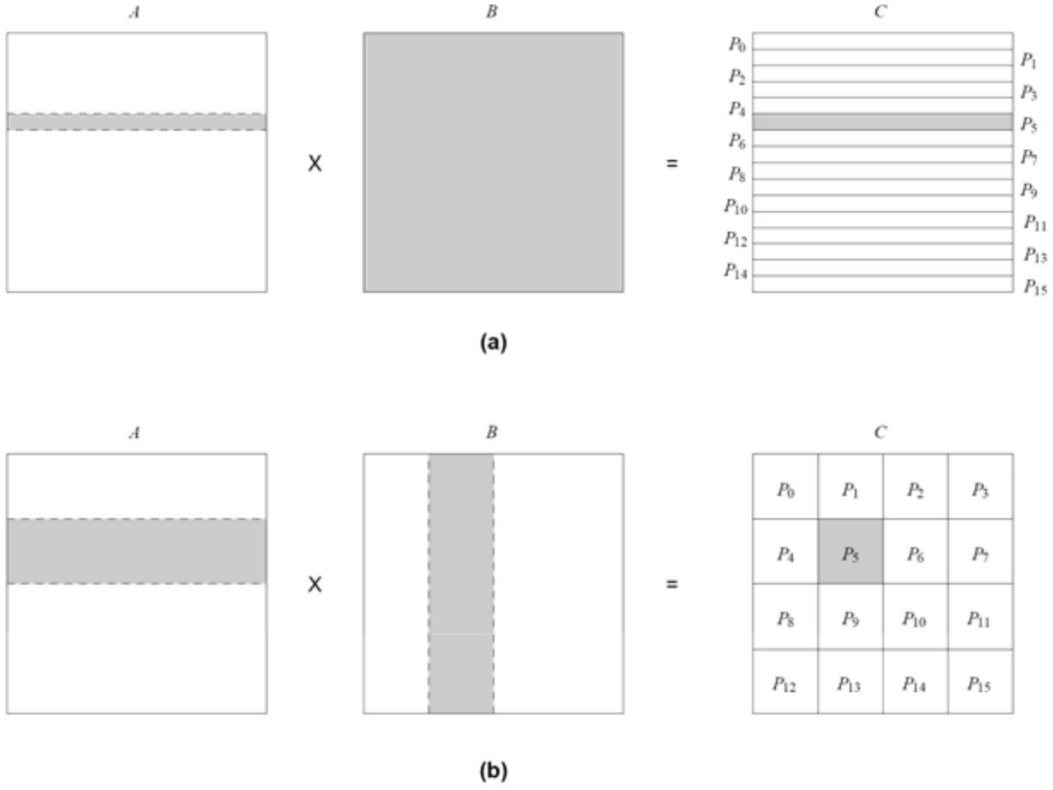


Figure 3.26 Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

Source: slides by Alexandre David

Graph partitioning

- For sparse data structures and data dependent interaction patterns.
  - Numerical simulations. Discretize the problem and represent it as a mesh.
- Sparse matrix: assign equal number of nodes to processes & minimize interaction.
- Example: simulation of dispersion of a water contaminant in Lake Superior.

Source: slides by Alexandre David

Discretization

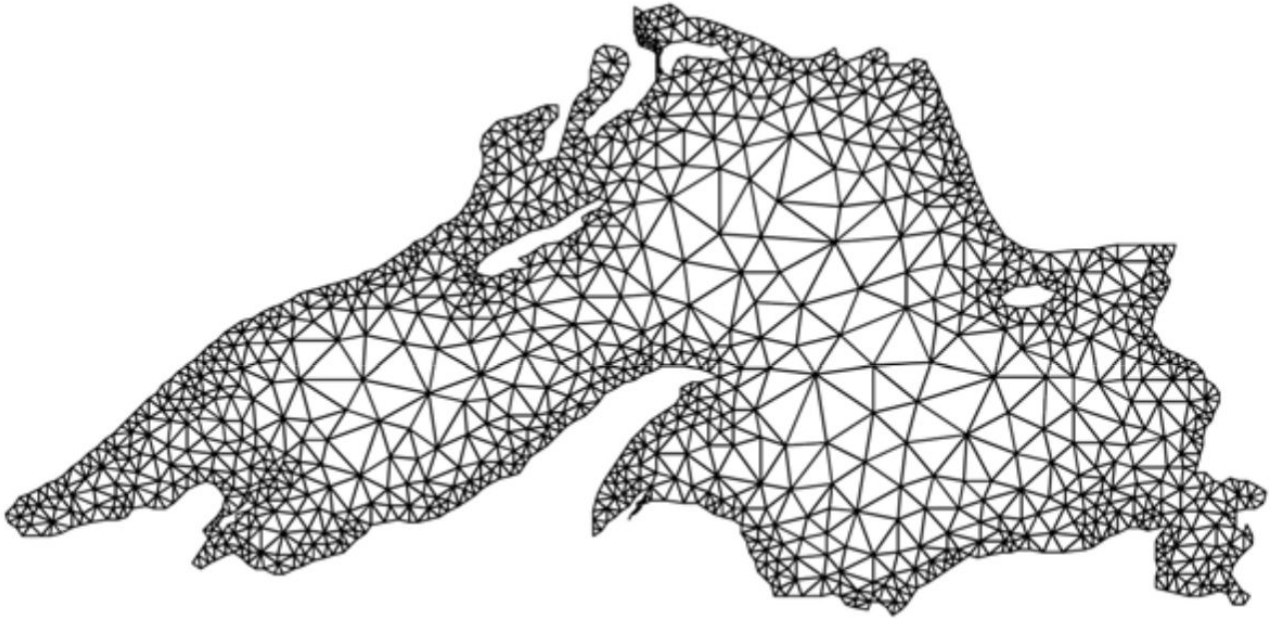
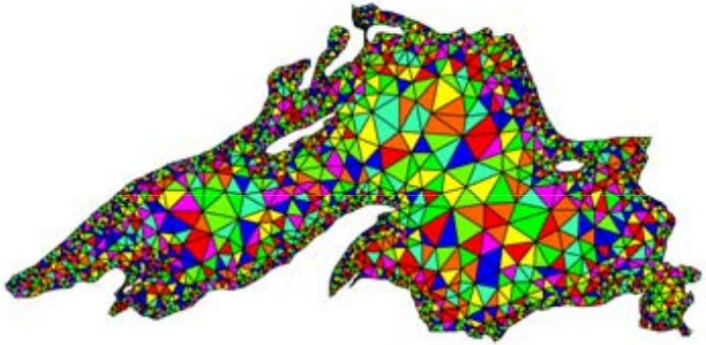


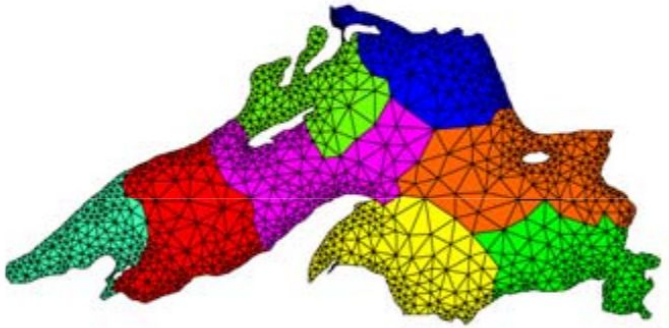
Figure 3.34 A mesh used to model Lake Superior.

Source: slides by Alexandre David

Partitioning Lake Superior



Random partitioning.

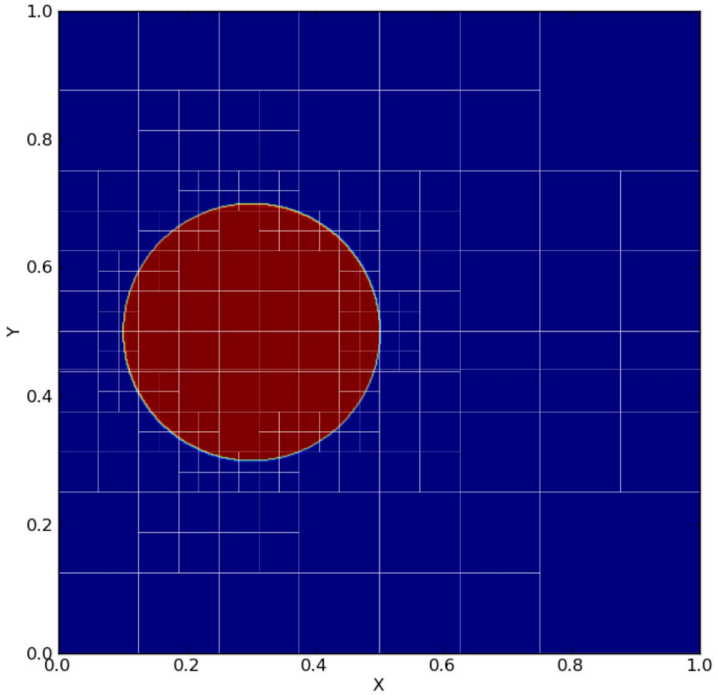


Partitioning with minimum edge cut.

Finding an exact optimal partitioning is an NP-complete problem.

AMR - Introduction

Source: slides by Massimiliano Guarrasi



Uniform meshes

- High resolution required for handling difficult regions (discontinuities, steep gradients, shocks, etc.)
- Computationally extremely costly

Adaptive Mesh Refinement

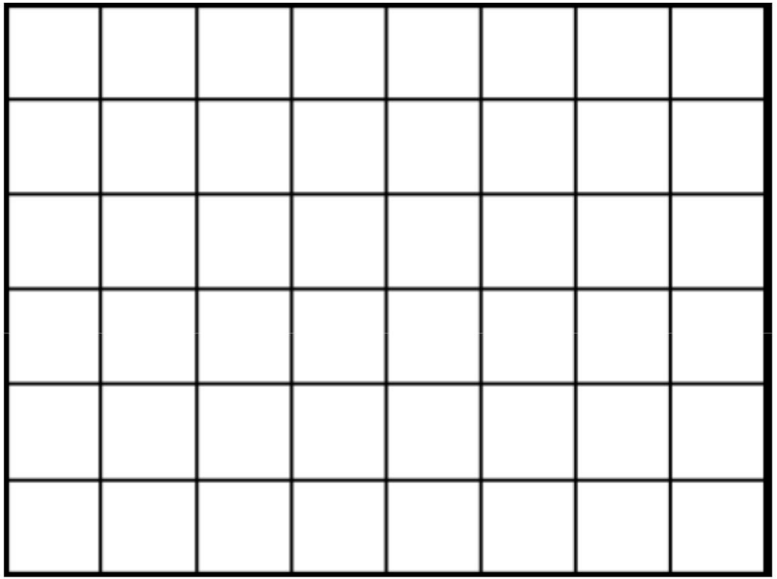
- Start with a coarse grid
- Identify regions that need finer resolution
- Superimpose finer sub-grids only on those regions
- Increased computational savings over a static grid approach.
- Increased storage savings over a static grid approach.
- Complete control of grid resolution, compared to the fixed resolution of a static grid approach.

AMR makes it feasible to solve problems that are intractable on uniform grid

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion

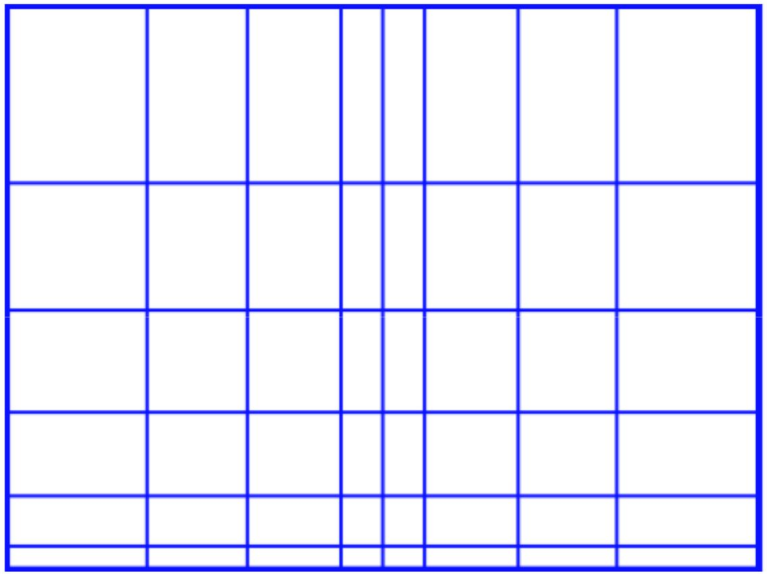


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion

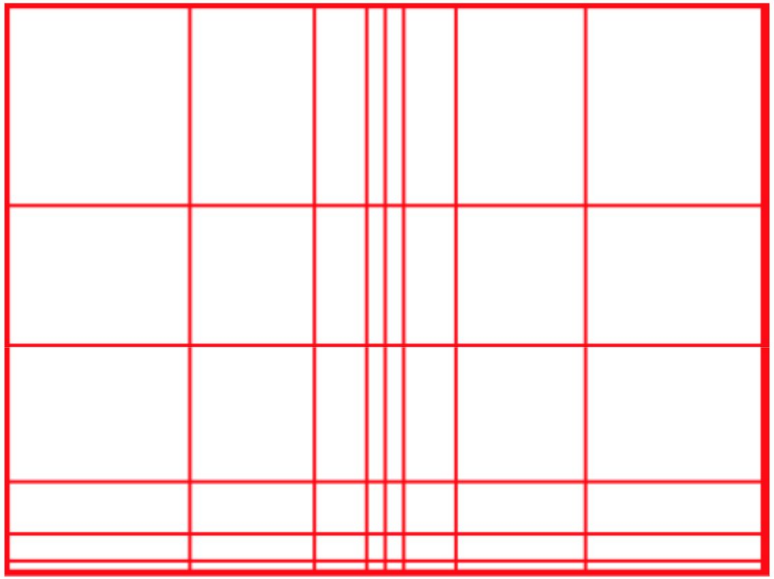


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion

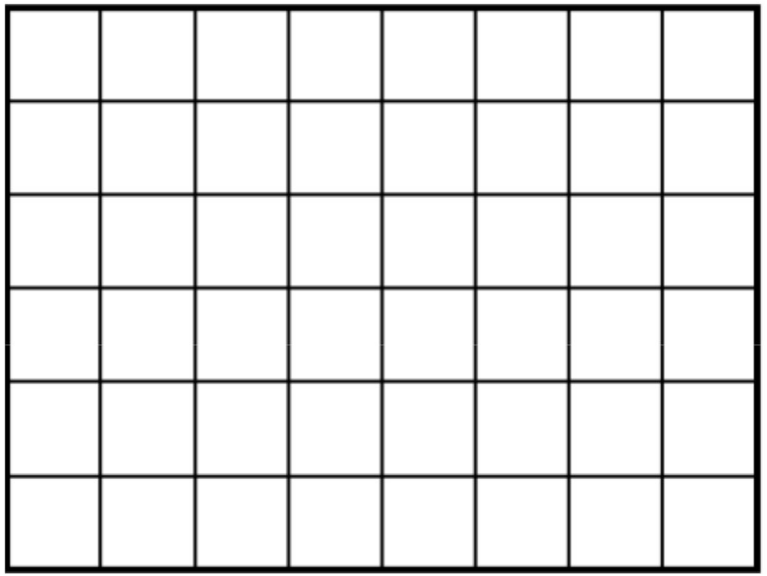


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured refinement (tree-based)

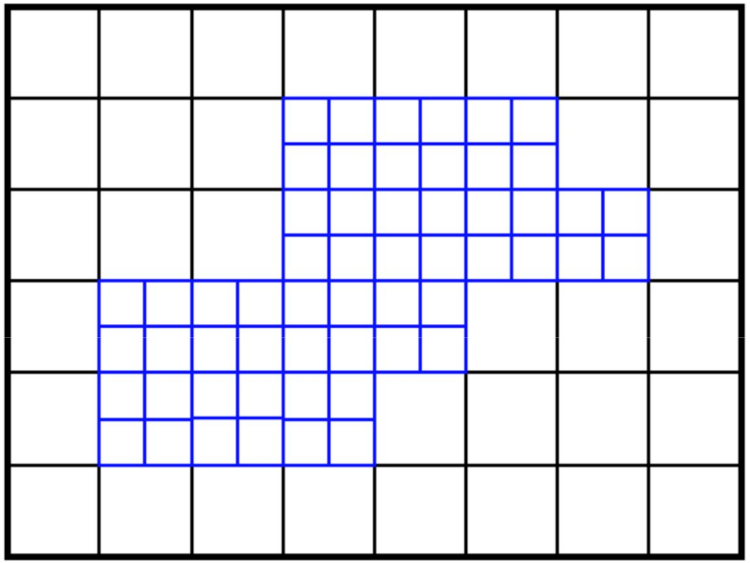


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured refinement (tree-based)

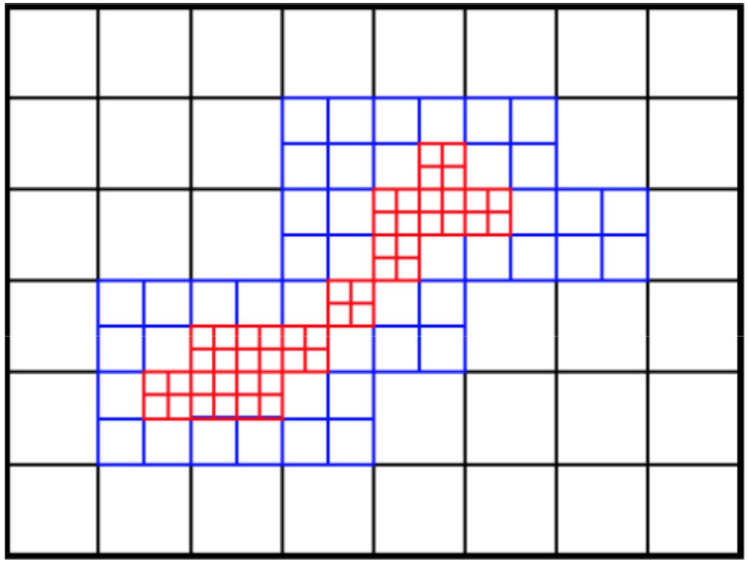


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured (tree-based) refinement

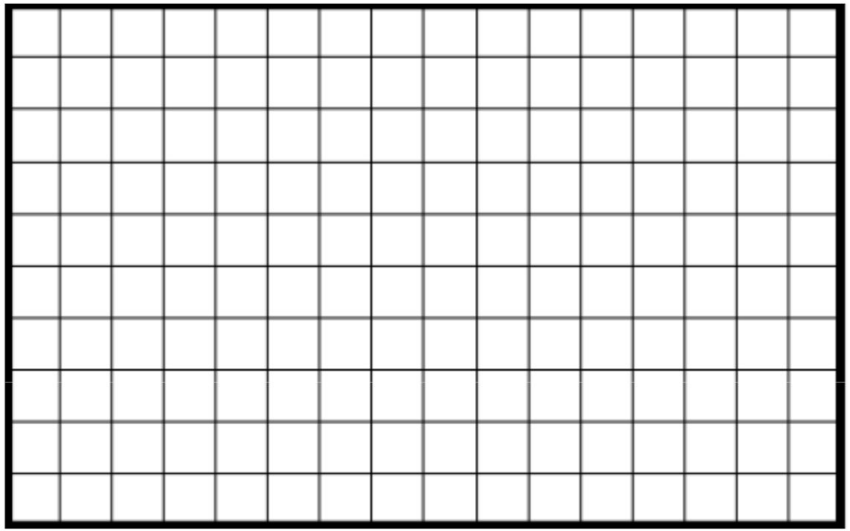


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured (tree-based) refinement
- block structured

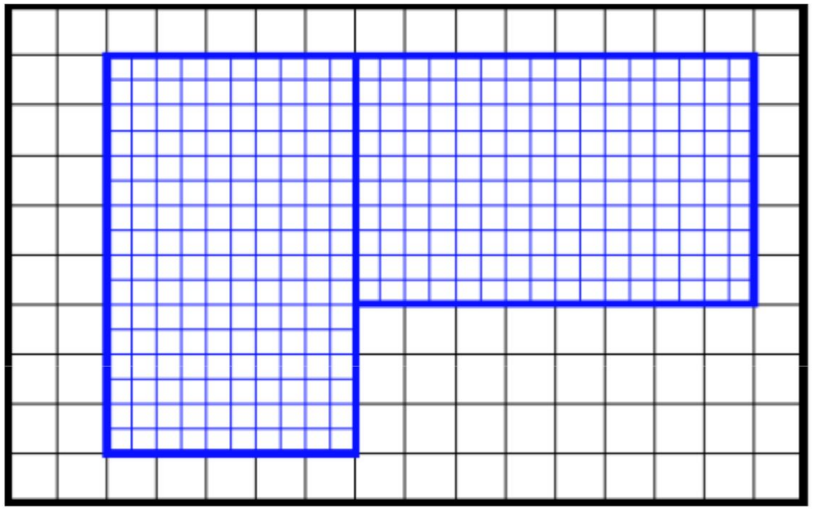


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured (tree-based) refinement
- block structured

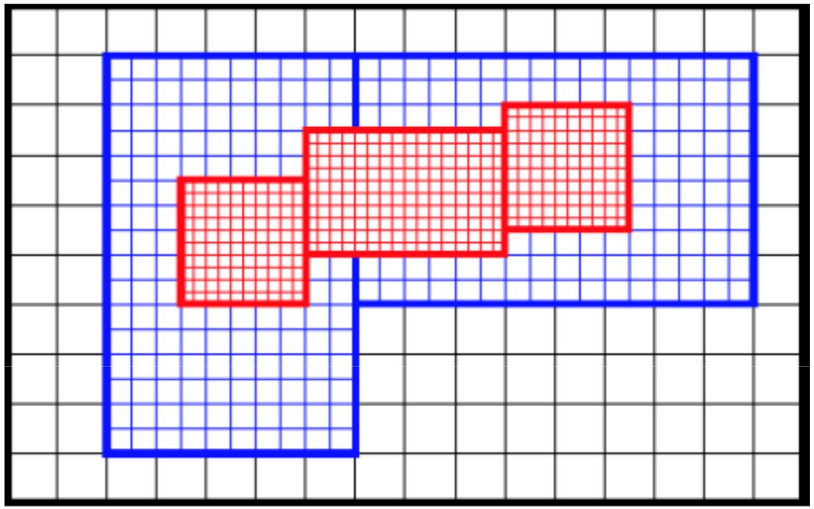


Courtesy of Dr. Andrea Mignone, University of Turin

AMR Techniques

Source: slides by Massimiliano Guarrasi

- mesh distortion
- point-wise structured refinement (tree-based)
- block structured:



Courtesy of Dr. Andrea Mignone, University of Turin

Parallel patterns

Source: slides by Allen Malony et al.

Parallel Patterns

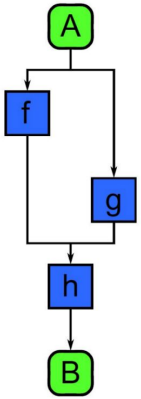
- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a “vocabulary” for algorithm design
- It can be useful to compare parallel patterns with serial patterns
- Patterns are universal – they can be used in *any* parallel programming system

Serial Control Patterns: Sequence

- **Sequence**: Ordered list of tasks that are executed in a specific order
- Assumption – program text ordering will be followed (obvious, but this will be important when parallelized)

```
1  T = f(A);
2  S = g(T);
3  B = h(S);
```

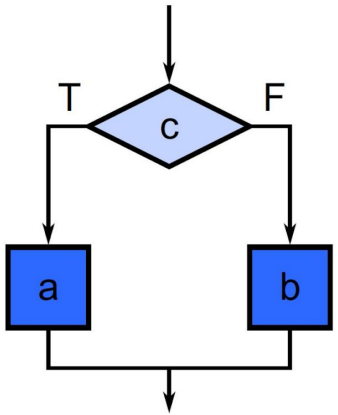

```
1  T = f(A);
2  S = g(A);
3  B = h(S,T);
```



Serial Control Patterns: Selection

- **Selection**: condition *c* is first evaluated. Either task *a* or *b* is executed depending on the true or false result of *c*.
- Assumptions – *a* and *b* are never executed before *c*, and only *a* or *b* is executed - never both

```
1  if (c) {
2    a;
3  } else {
4    b;
5  }
```

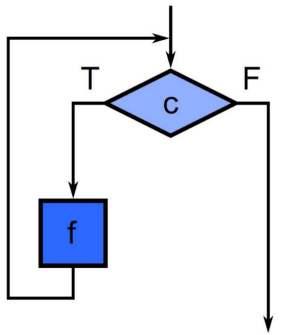


Serial Control Patterns: Iteration

- **Iteration**: a condition *c* is evaluated. If true, *a* is evaluated, and then *c* is evaluated again. This repeats until *c* is false.
- Complication when parallelizing: potential for dependencies to exist between previous iterations

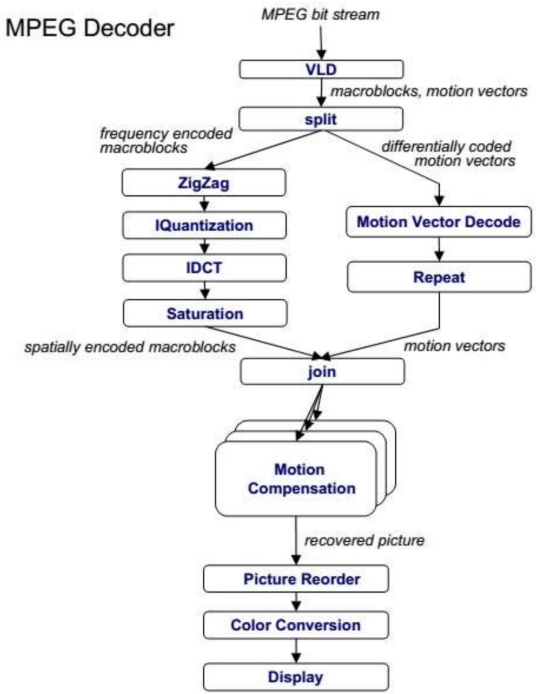
```
1  for (i = 0; i < n;
2    a;
3  }
```

```
1  while (c) {
2    a;
3  }
```

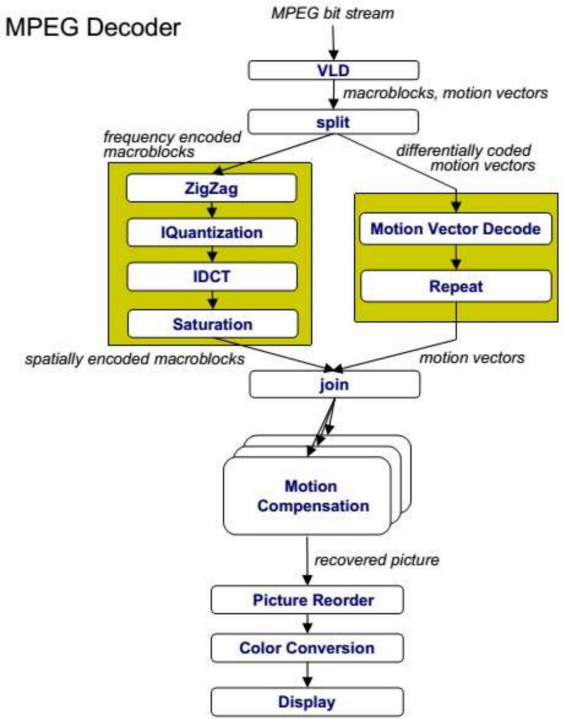


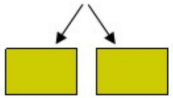
Source: slides by Andrea Marongiu

Where is the parallelism?



Where is the parallelism?

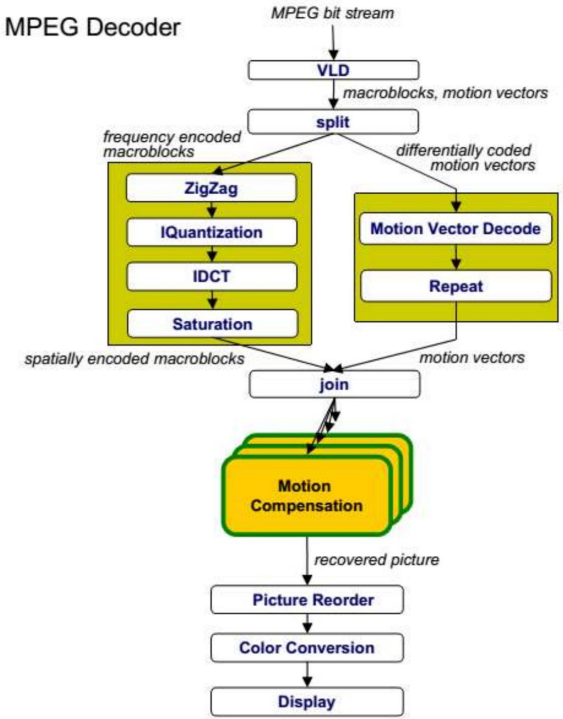




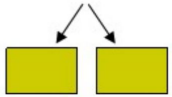
- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

Source: slides by Andrea Marongiu

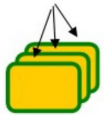
Where is the parallelism?



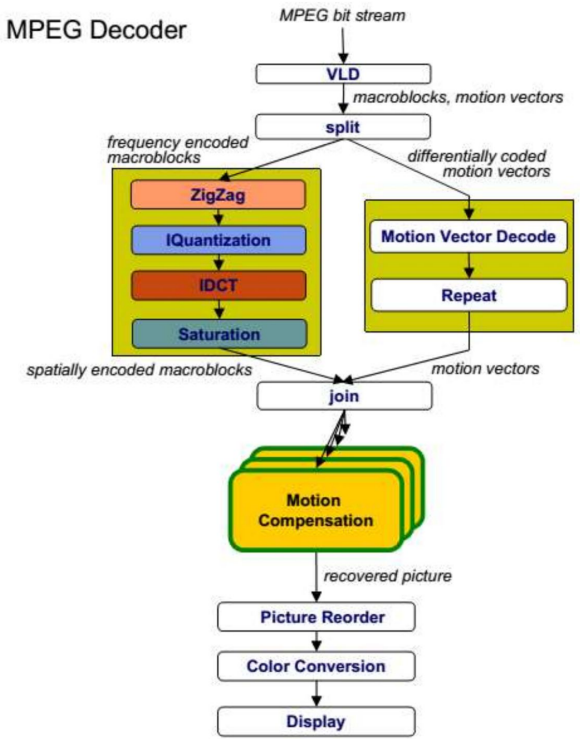
- Task decomposition
  - Parallelism in the application



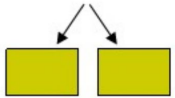
- Data decomposition
  - Same computation is applied to small data chunks derived from large data set



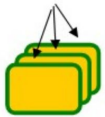
Where is the parallelism?



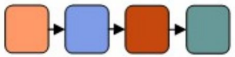
- Task decomposition
  - Parallelism in the application



- Data decomposition
  - Same computation is applied to small data chunks derived from large data set



- Pipeline decomposition
  - Data assembly lines
  - Producer-consumer chains

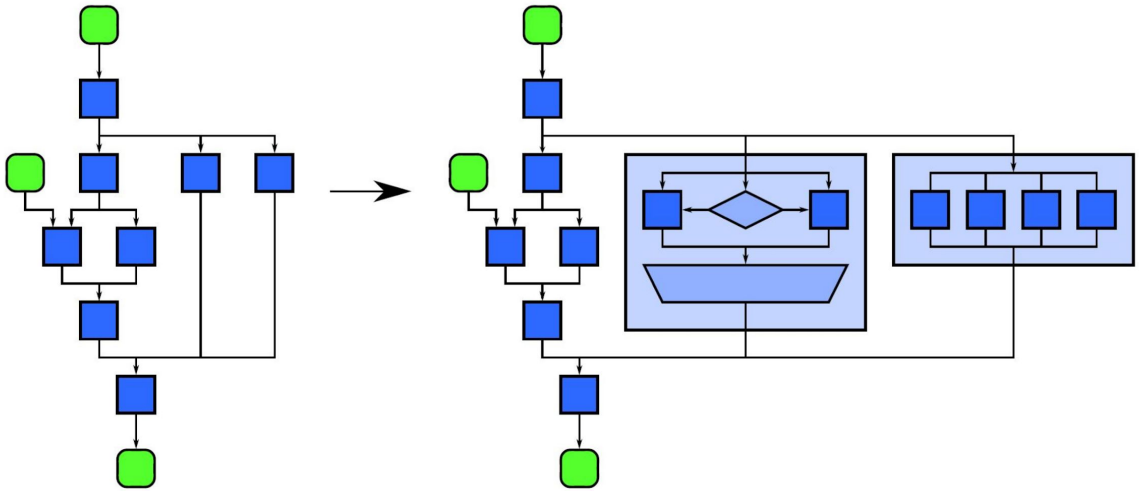


Source: slides by Allen Malony et al.

Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns
- This pattern appears in both serial and parallel algorithms
- “Pattern diagrams” are used to visually show the pattern idea where each “task block” is a location of general code in an algorithm
- Each “task block” can in turn be another pattern in the **nesting pattern**

Nesting Pattern



Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.

Source: slides by Allen Malony et al.

Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- Cilk Plus implements this with **spawn** and **sync**
  - The call tree is a parallel call tree and functions are spawned instead of called
  - Functions that spawn another function call will continue to execute
  - Caller *syncs* with the spawned function to join the two
- A "join" is different than a "barrier
  - Sync – only one thread continues
  - Barrier – all threads continue

Source: slides by Allen Malony et al.

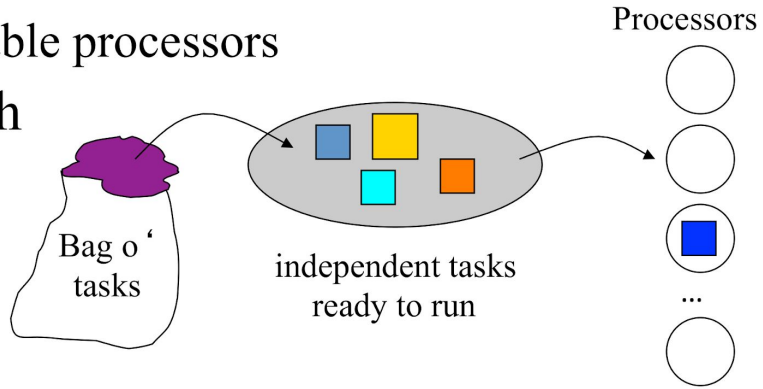
Master-Worker Parallelism

- One or more master processes generate work
- Masters allocate work to worker processes
- Workers idle if have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but most be told to do so
- Master may become the bottleneck if not careful
- What are the performance factors and expected performance behavior
  - Consider task granularity and asynchrony
  - How do they interact?

Source: slides by Allen Malony et al.

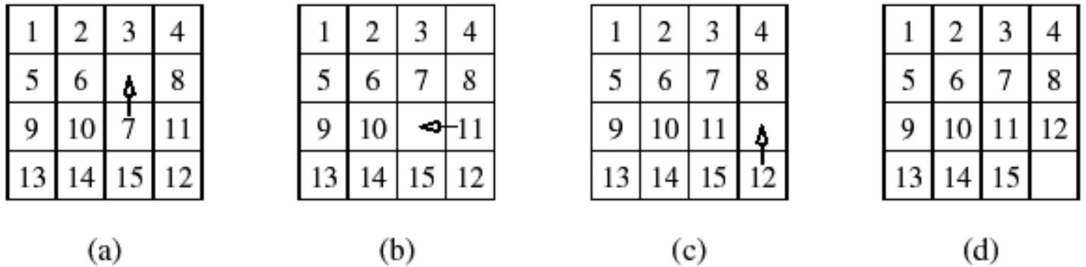
Bag o' Tasks Model and Worker Pool

- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o' Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisified, it is moved to a ready to run queue
  - Scheduler assigns a task to a free processor
- Dynamic approach that is effective for load balancing



Search-Based (Exploratory) Decomposition

- 15-puzzle problem
- 15 tiles numbered 1 through 15 placed in 4x4 grid
  - Blank tile located somewhere in grid
  - Initial configuration is out of order
  - Find shortest sequence of moves to put in order

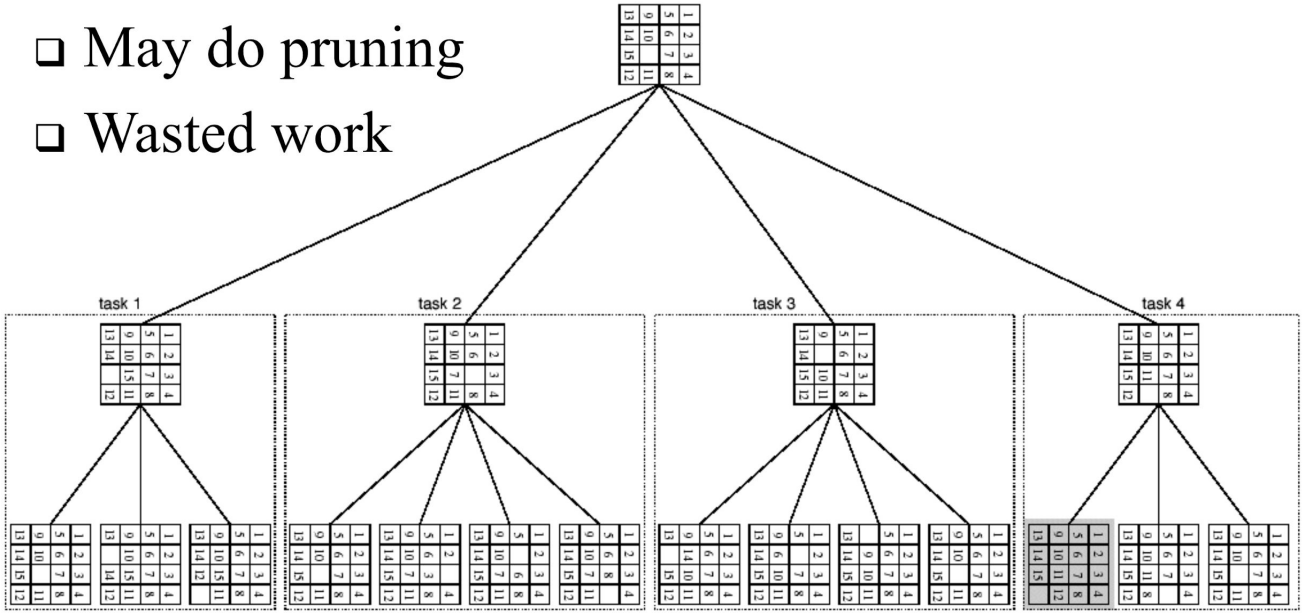


- Sequential search across space of solutions
  - May involve some heuristics

Source: slides by Allen Malony et al.

Parallelizing the 15-Puzzle Problem

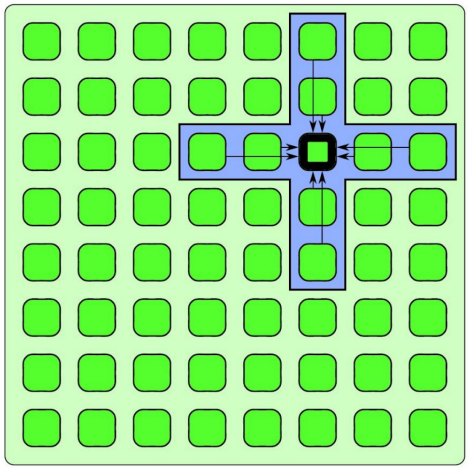
- Enumerate move choices at each stage
- Assign to processors
- May do pruning
- Wasted work



Source: slides by Allen Malony et al.

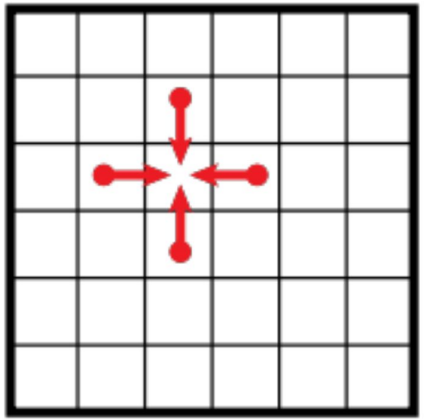
Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of "neighbors", stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…



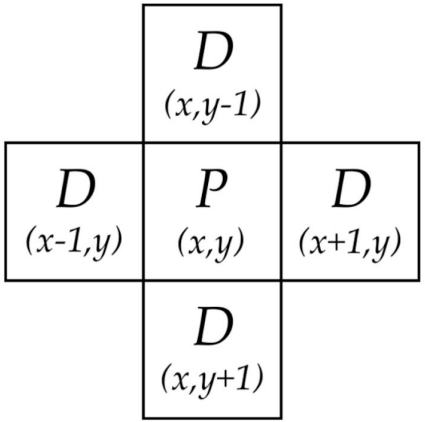
Source: slides by Allen Malony et al.

2-Dimensional Stencils



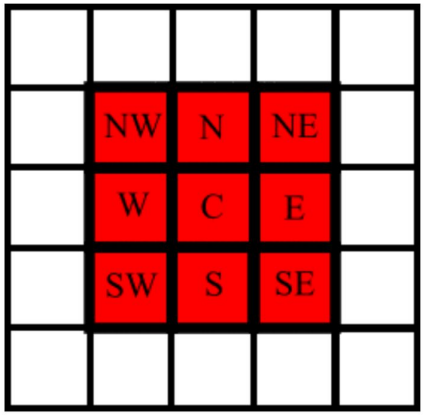
4-point stencil

Center cell (P)
is not used



5-point stencil

Center cell (P)
is used as well

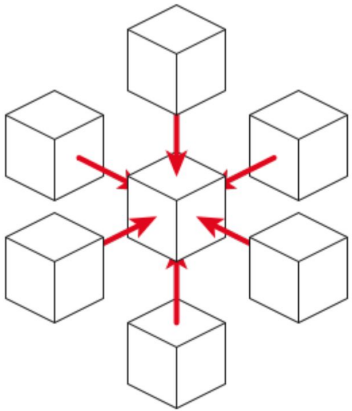


9-point stencil

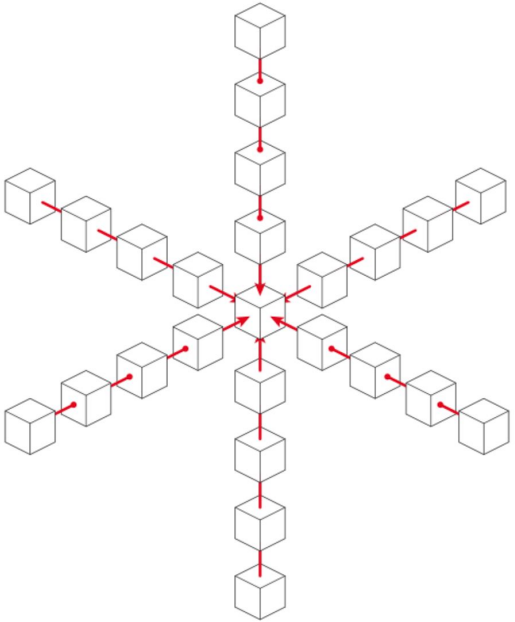
Center cell (C)
is used as well

Source: slides by Allen Malony et al.

3-Dimensional Stencils



6-point stencil
(7-point stencil)

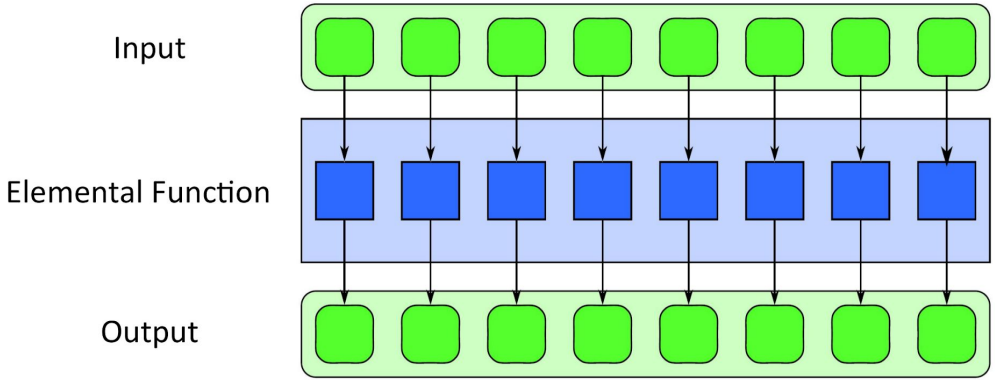


24-point stencil
(25-point stencil)

Source: slides by Allen Malony et al.

Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an "elemental function"



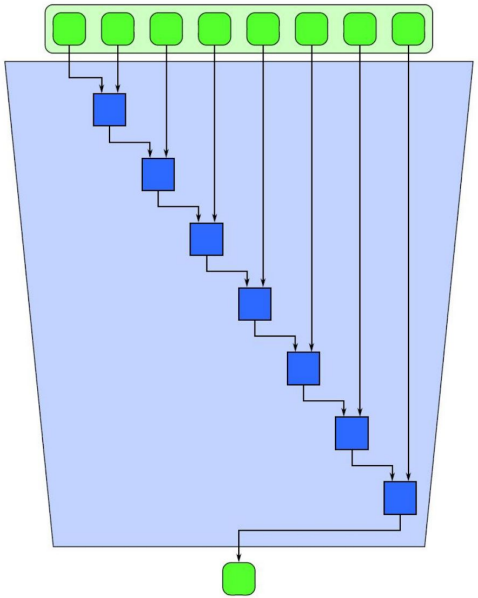
Parallel Control Patterns: Reduction

- **Reduction**: Combines every element in a collection using an associative "combiner function"
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

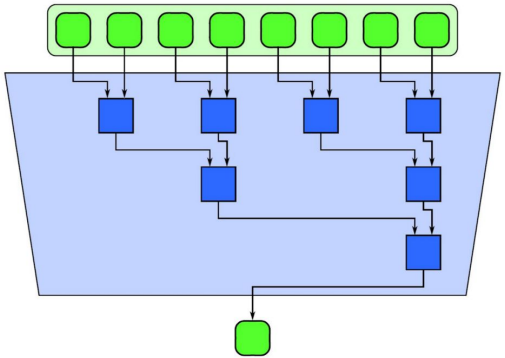
Source: slides by Allen Malony et al.

Parallel Control Patterns: Reduction

Serial Reduction



Parallel Reduction



Source: slides by Allen Malony et al.

Big-Data and Map-Reduce

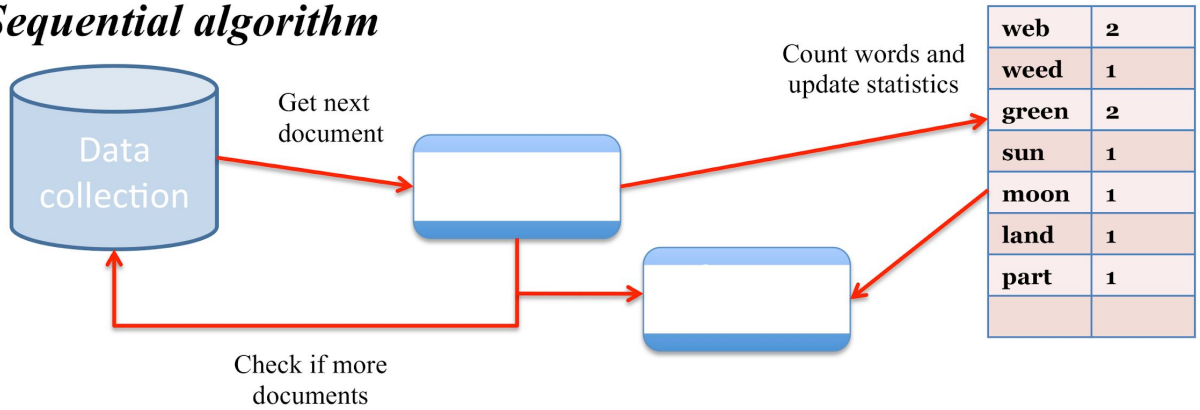
- Big-data deals with processing large data sets
- Nature of data processing problem makes it amenable to parallelism
  - Looking for features in the data
  - Extracting certain characteristics
  - Analyzing properties with complex data mining algorithms
- Data size makes it opportunistic for partitioning into large # of sub-sets and processing these in parallel
- We need new algorithms, data structures, and programming models to deal with problems

Source: slides by Allen Malony et al.

A Simple Big-Data Problem

- Consider a large data collection of text documents
- Suppose we want to find how often a particular word occurs and determine a probability distribution for all word occurrences

Sequential algorithm

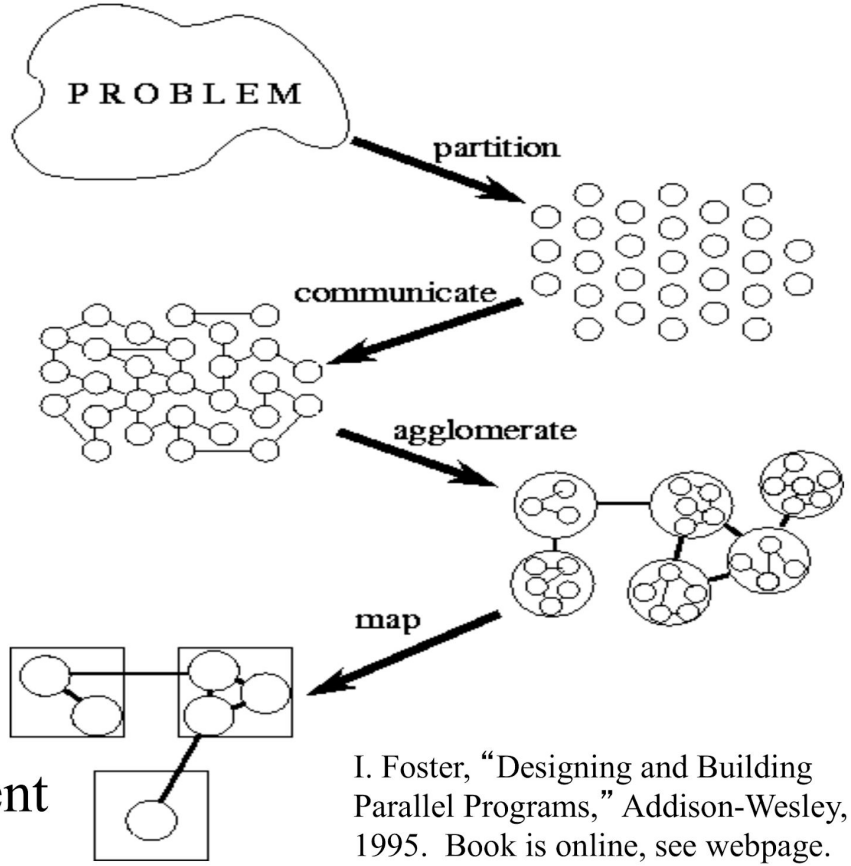


Quick recap

Methodological Design

Source: slides by Allen Malony et al.

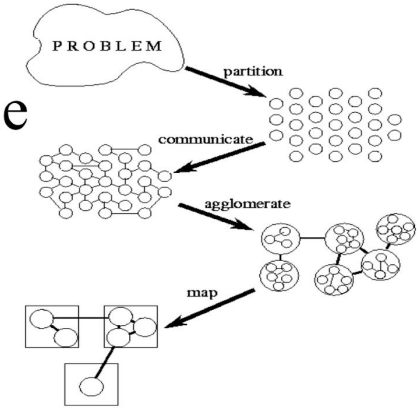
- Partition
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995. Book is online, see webpage.

Source: slides by Allen Malony et al.

Partitioning

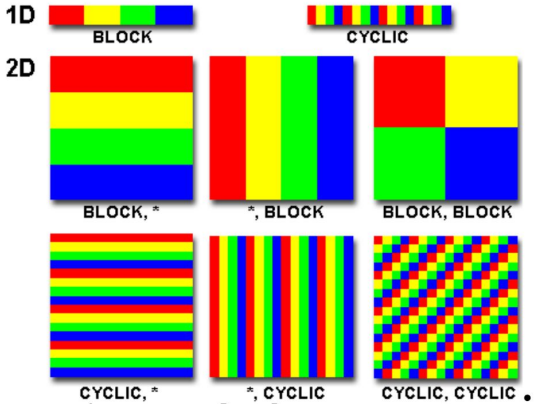


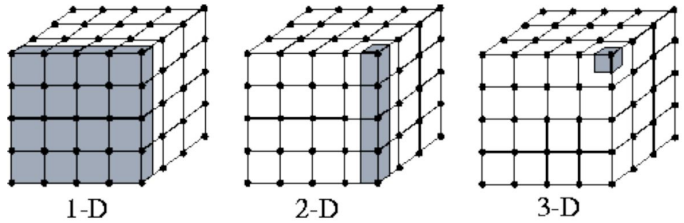
- Partitioning stage is intended to expose opportunities for parallel execution
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the computational *tasks* associated with a problem and the *data* on which the tasks operates
- *Domain decomposition* focuses on computation data
- *Functional decomposition* focuses on computation tasks
- Mixing domain/functional decomposition is possible

Source: slides by Allen Malony et al.

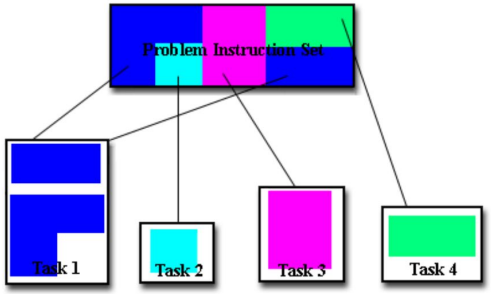
Domain and Functional Decomposition

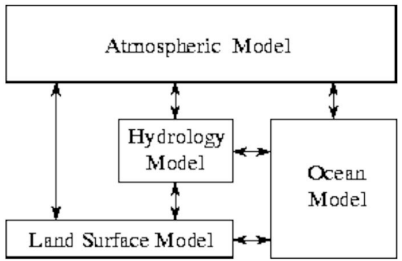
- Domain decomposition of 2D / 3D grid



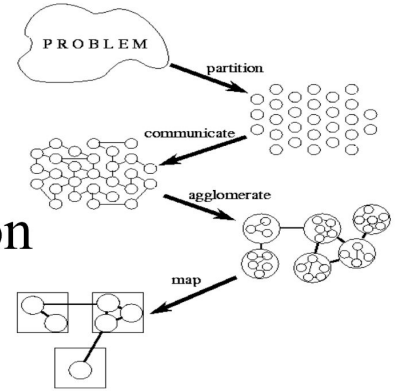


- Functional decomposition of a climate model





Agglomeration



- Move from parallel abstractions to real implementation
- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to *agglomerate*?
  - What happens when tasks are combined?
- Is it useful to *replicate* data and/or computation?
- Changes important algorithm and performance ratios
  - *Surface-to-volume*: reduction in communication at the expense of decreasing parallelism
  - *Communication/computation*: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap