GPU Teaching Kit

Accelerated Computing

Module 11 – Computational Thinking

Objective

- To provide you with a framework for further studies on
  - Thinking about the problems of parallel programming
  - Discussing your work with others
  - Approaching complex parallel programming problems
  - Using or building useful tools and environments

Fundamentals of Parallel Computing

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
  - This is what CUDA (and OpenMP, OpenCL) is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - However, you will find parallels for almost every technique you learned in this course
  - Need to be aware of space-time constraints

Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- Many:Many, One:Many, Many:One, One:One

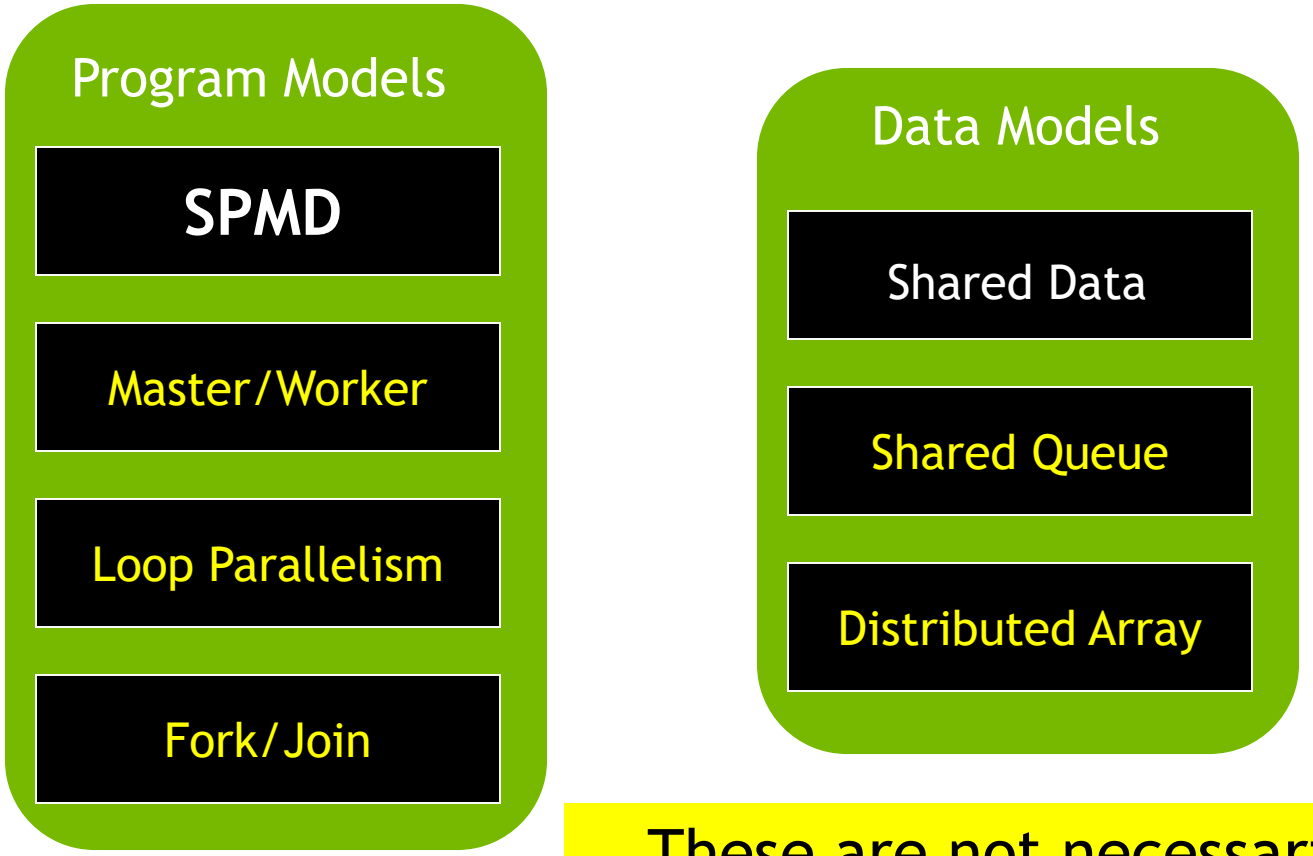
Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
  - Watch out for serialization

- Important: be aware of which items of work are truly independent

6

Parallel Programming Coding Styles – Program and Data Models



These are not necessarily mutually exclusive.

Program Models

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

8

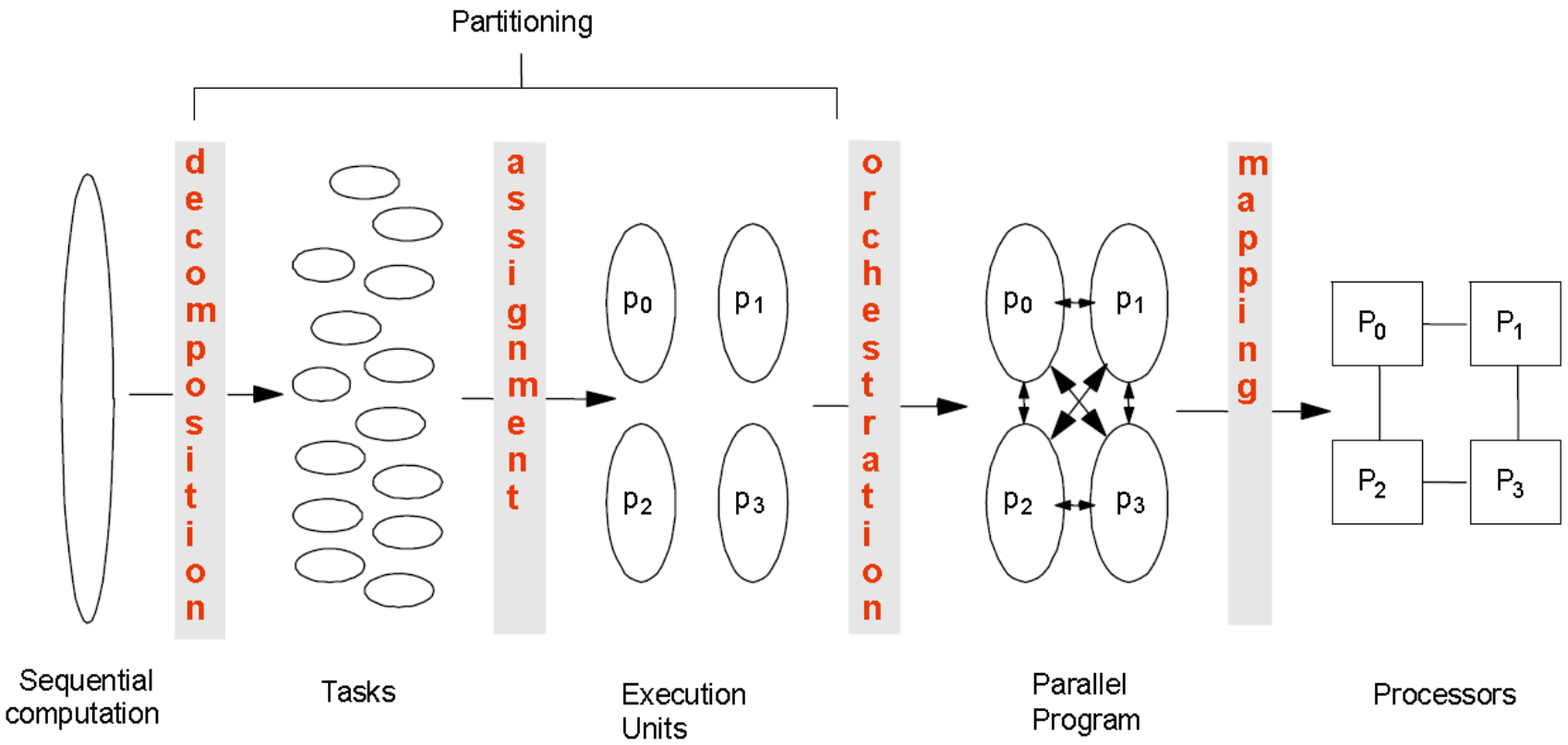
SPMD

- 1. Initialize—establish localized data structure and communication channels.
- 2. Uniquify—each thread acquires a unique identifier, typically ranging from 0 to N-1, where N is the number of threads. Both OpenMP and CUDA have built-in support for this.
- 3. Distribute data—decompose global data into chunks and localize them, or sharing/replicating major data structures using thread IDs to associate subsets of the data to threads.
- 4. Compute—run the core computation! Thread IDs are used to differentiate the behavior of individual threads. Use thread ID in loop index calculations to split loop iterations among threads—beware of the potential for memory/data divergence. Use thread ID or conditions based on thread ID to branch to their specific actions—beware of the potential for instruction/execution divergence.
- 5. Finalize—reconcile global data structure, and prepare for the next major iteration or group of program phases.

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

Common Steps to Parallelization



Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to units of execution to exploit parallel architecture

Software Construction

- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

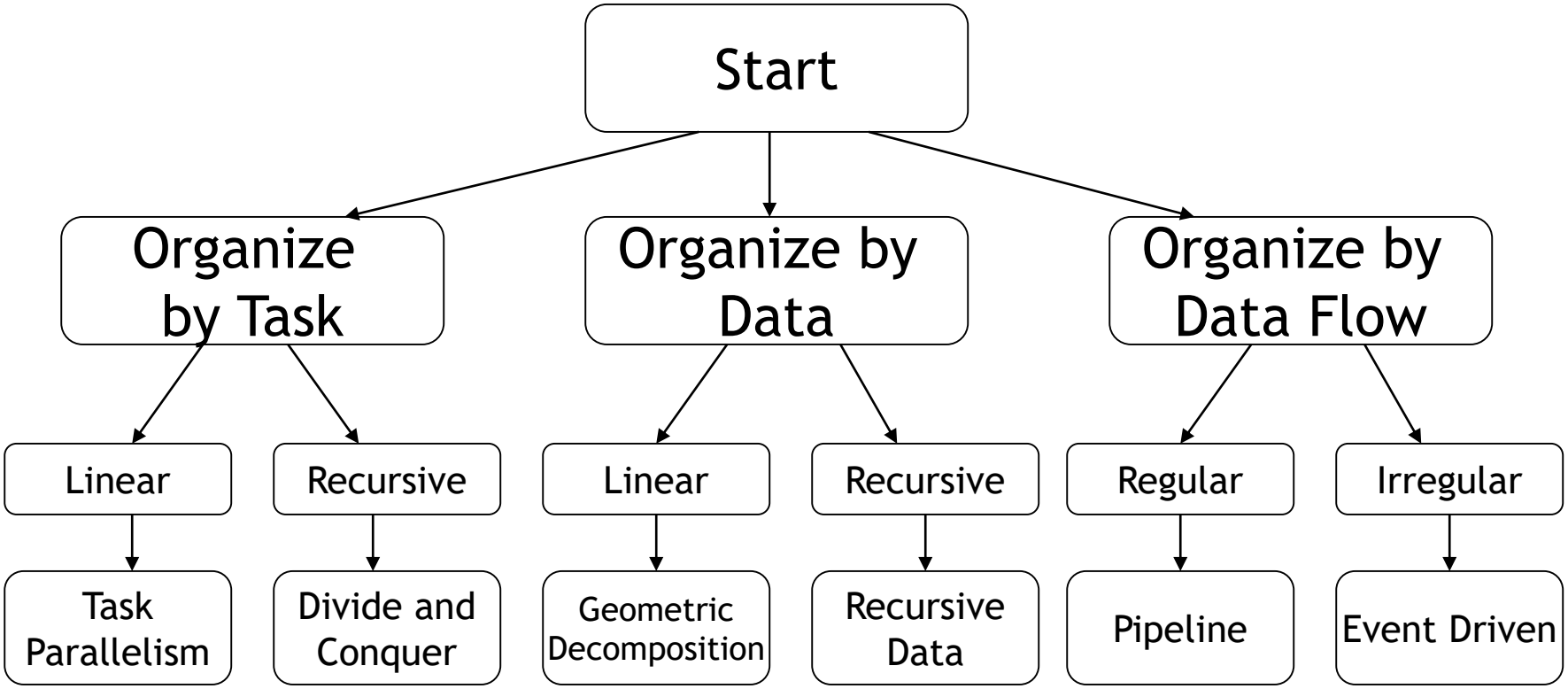
Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).

Major Organizing Principle

- How to determine the algorithm structure that represents the mapping of tasks to units of execution?
- Concurrency usually implies major organizing principle
  - Organize by tasks
  - Organize by data decomposition
  - Organize by flow of data

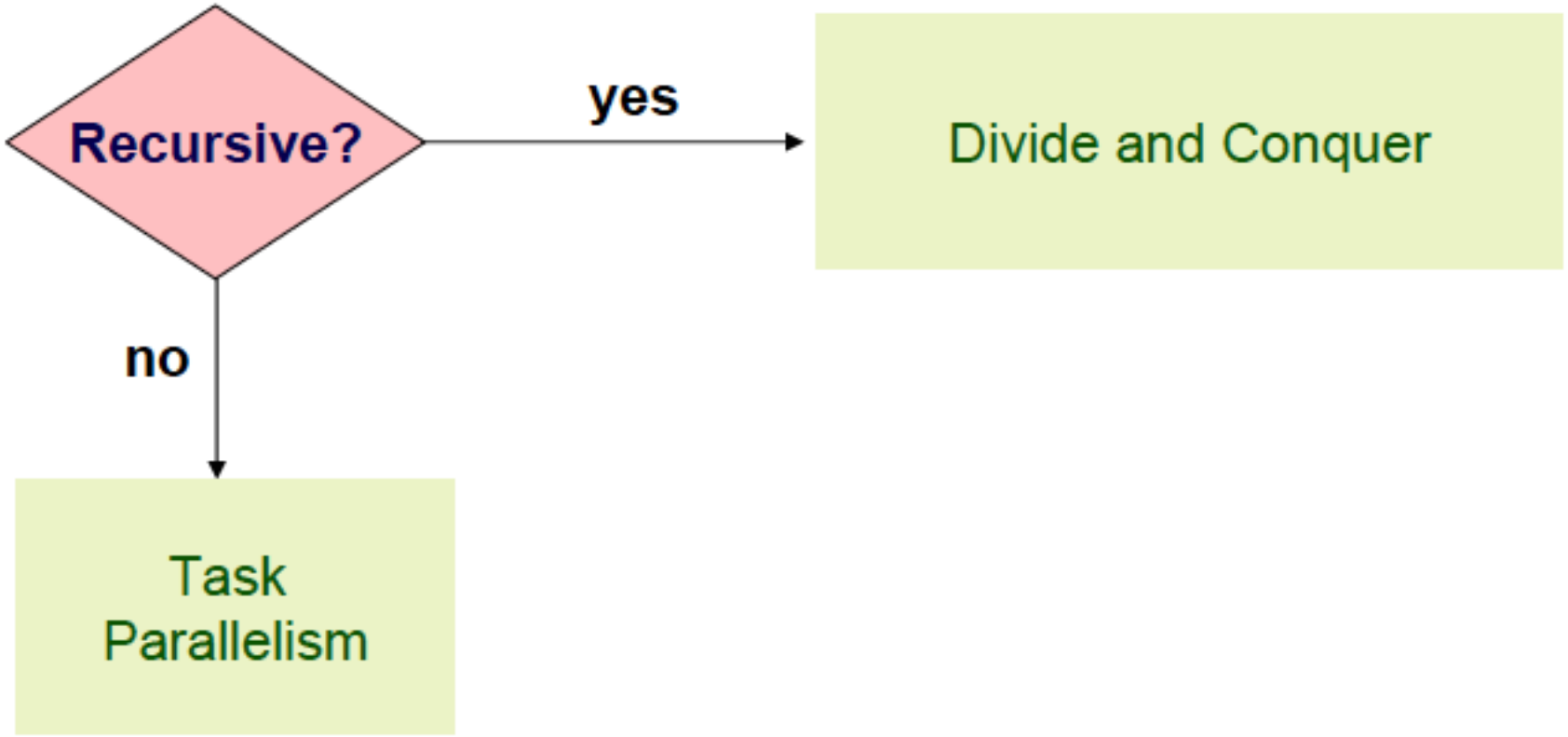
Algorithm Structure

How to determine the algorithm structure that represents the mapping of tasks to units of execution?



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

Organize by Tasks?

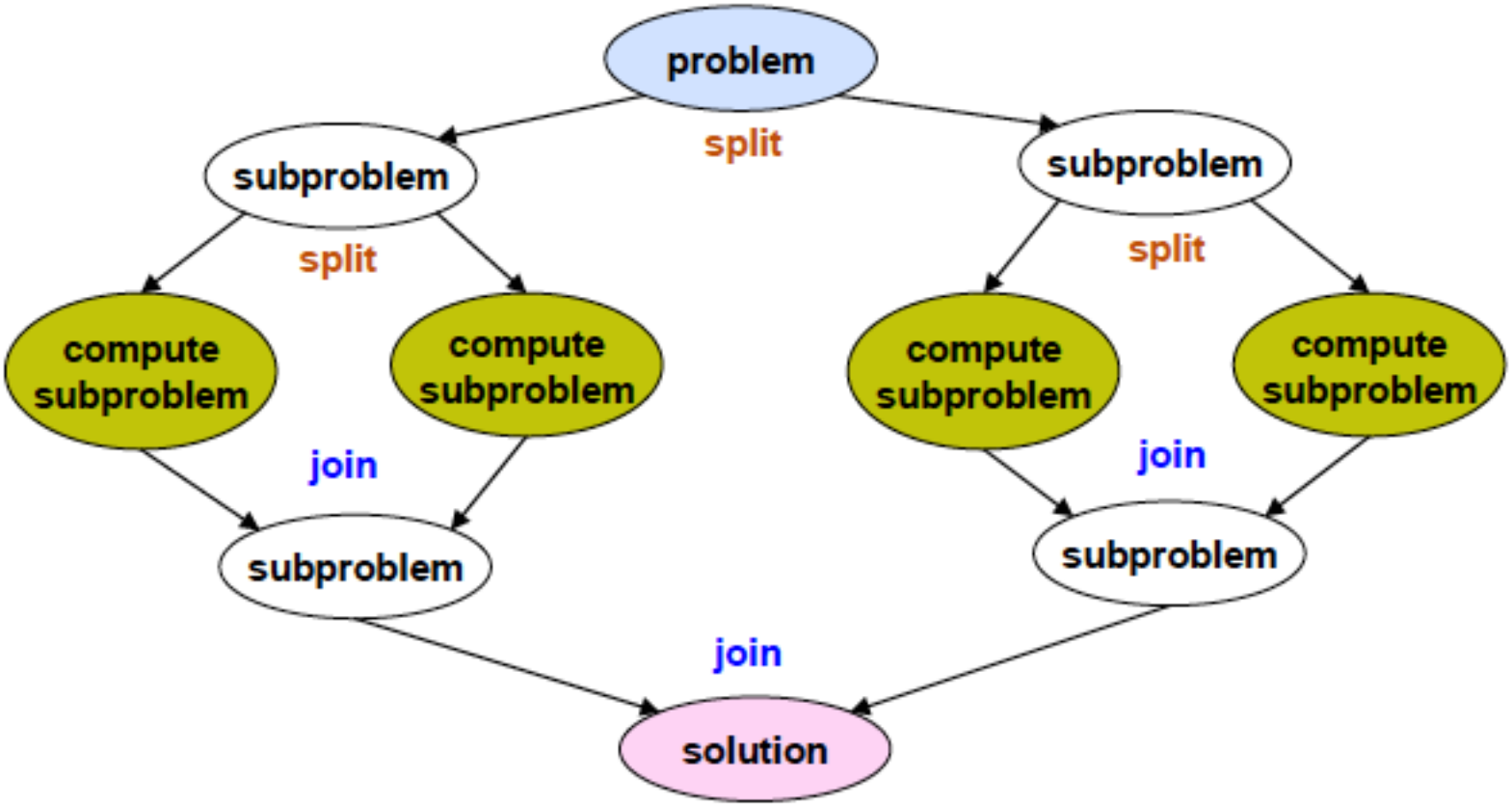


Task Parallelism

- Ray tracing
  - Computation for each ray is a separate and independent
- Molecular dynamics
  - Non-bonded force calculations, some dependencies
- Common factors
  - Tasks are associated with iterations of a loop
  - Tasks largely known at the start of the computation
  - All tasks may not need to complete to arrive at a solution

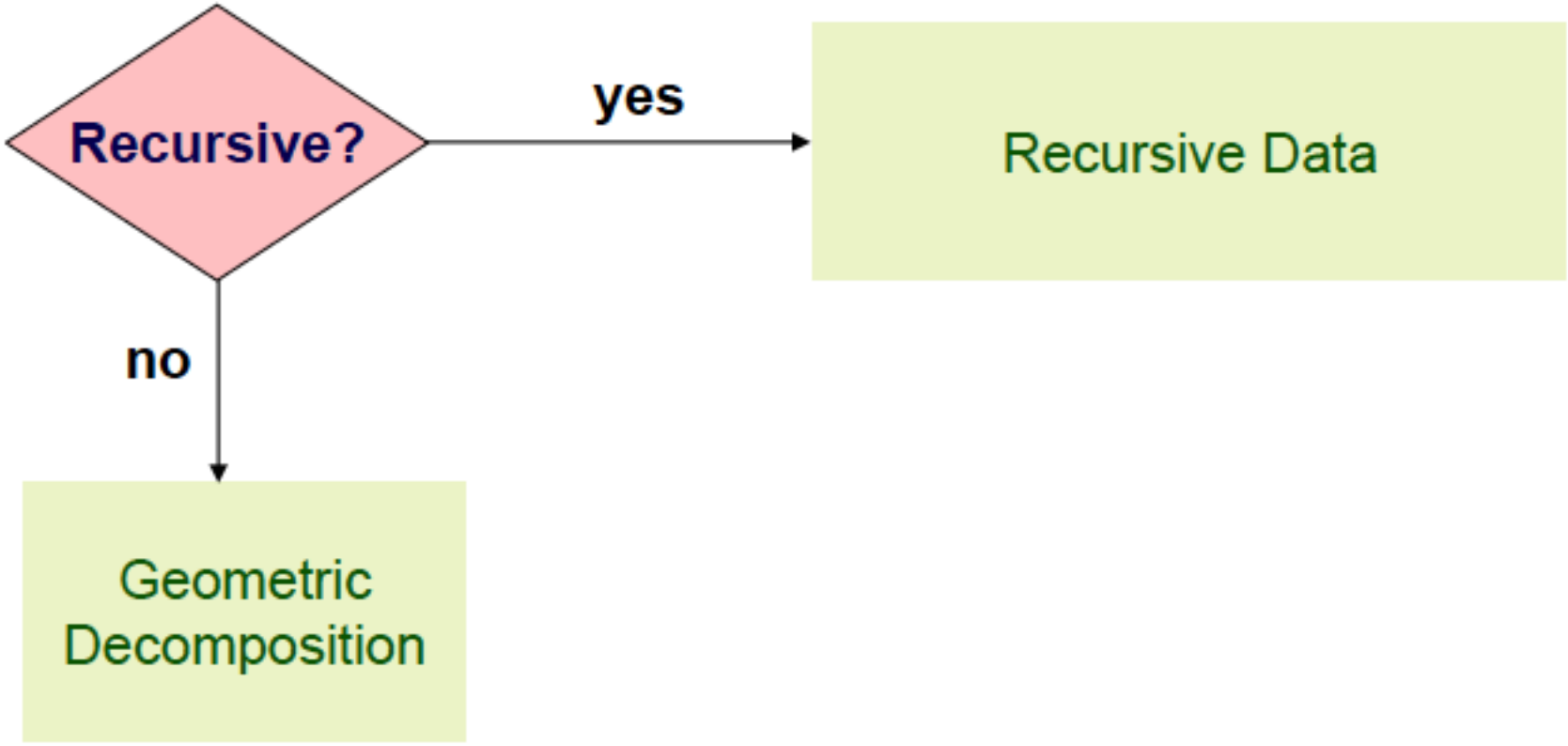
Divide and Conquer

- For recursive programs: divide and conquer
  - Subproblems may not be uniform
  - May require dynamic load balancing



Organize by Data?

- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures



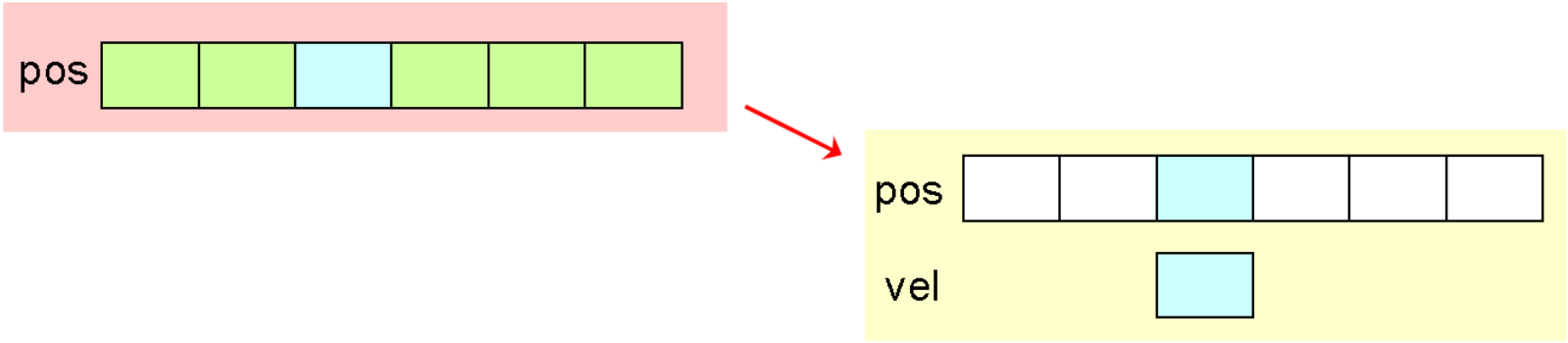
Geometric Decomposition

– Gravitational body simulation

```
VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES - 1; i++) {
  for (j = i + 1; j < NUM_BODIES; j++) {
    // Displacement vector
    VEC3D d = pos[j] - pos[i];
    // Force
    t = 1 / sqr(length(d));
    // Components of force along displacement
    d = t * (d / length(d));

    acc[i] += d * mass[j];
    acc[j] += -d * mass[i];
  }
}
```

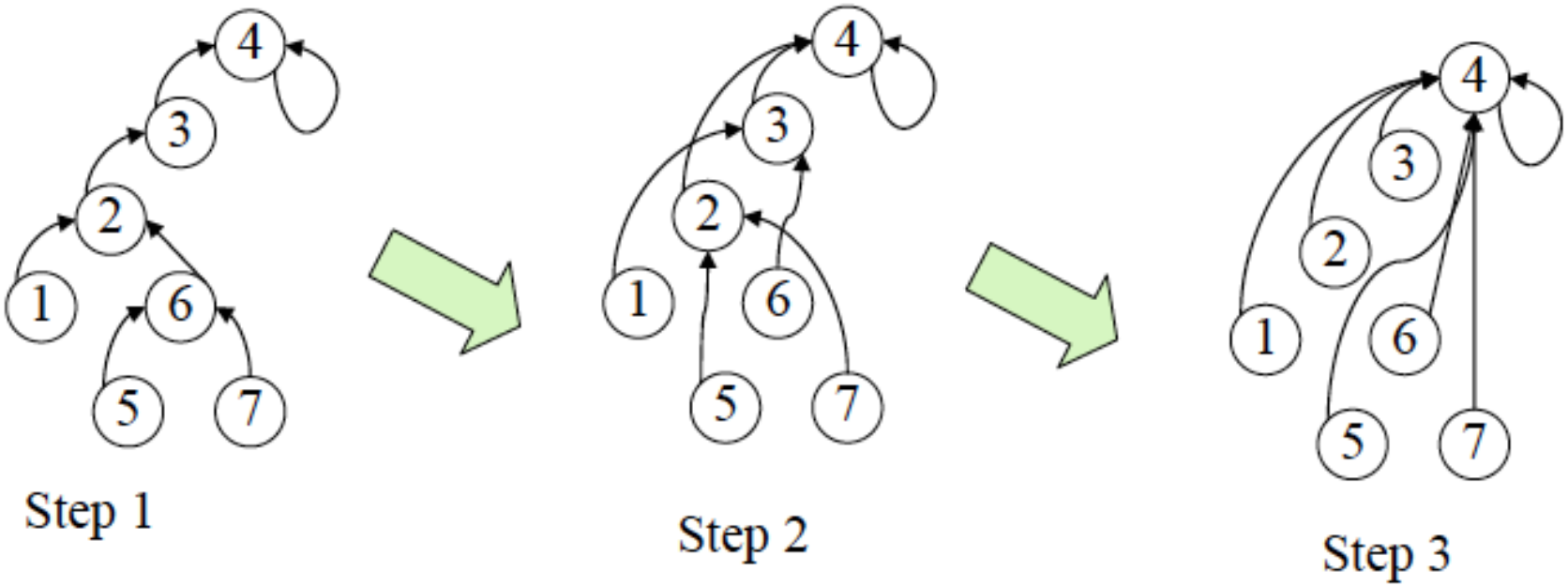


Recursive Data

–

– Computation on a list, tree, or graph
  – Often appears the only way to solve a problem is to sequentially move through the data structure
– There are however opportunities to reshape the operations in a way that exposes concurrency

- Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  - Parallel approach: for each node, find its successor's successor, repeat until no changes
  - O(log n) vs. O(n)



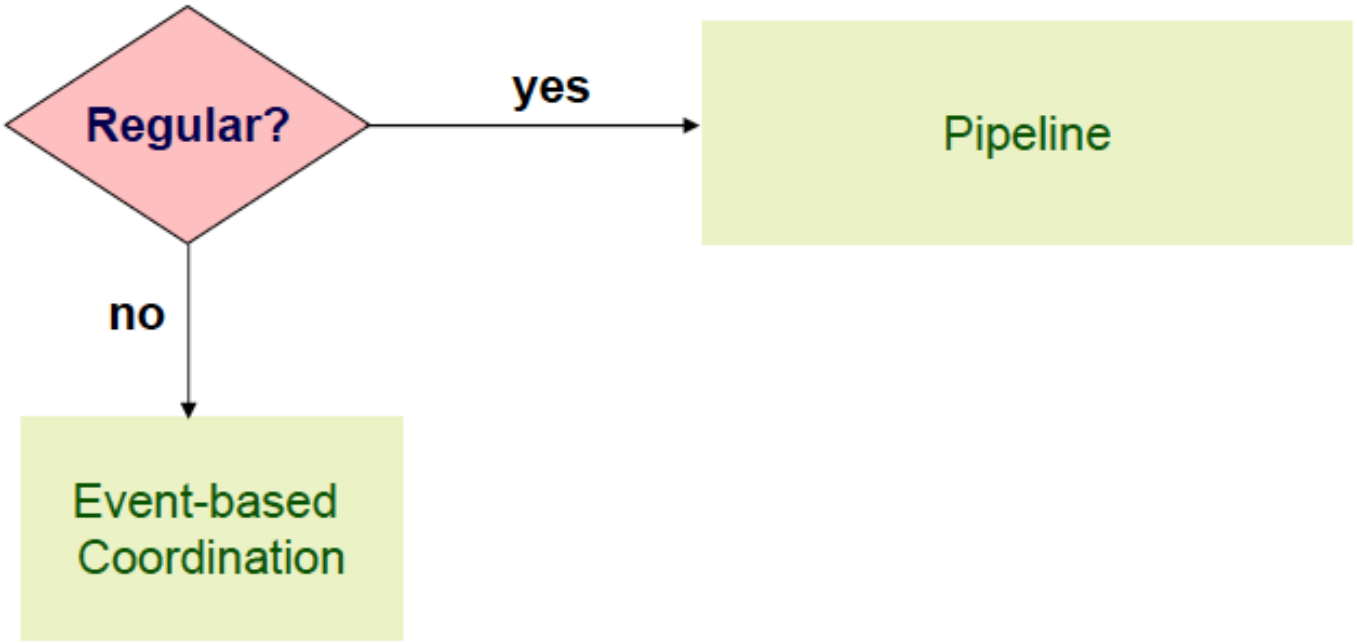
Work vs. Concurrency Tradeoff

–

– Parallel restructuring of find the root algorithm leads to O(n log n) work vs. O(n) with sequential approach
– Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency

Organize by Flow of Data?

- In some application domains, the flow of data imposes ordering on the tasks
- Regular, one-way, mostly stable data flow
- Irregular, dynamic, or unpredictable data flow



Pipeline Throughput vs. Latency

–

– Amount of concurrency in a pipeline is limited by the number of stages
– Works best if the time to fill and drain the pipeline is small compared to overall running time
– Performance metric is usually the throughput
  – Rate at which data appear at the end of the pipeline per time unit (e.g., frames per second)
– Pipeline latency is important for real-time applications
  – Time interval from data input to pipeline, to data output

Event-Based Coordination

–

– In this pattern, interaction of tasks to process data can vary over unpredictable intervals
– Deadlocks are likely for applications that use this pattern

More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

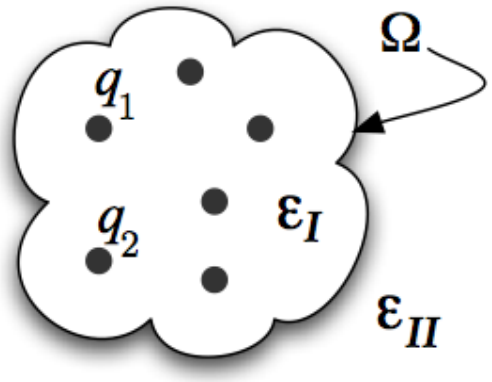
Both can have very different performance results and code complexity depending on the way they are done.

Making Science Better, not just Faster

or… in other words:
There will be no Nobel Prizes or Turing Awards awarded for “just recompile” or using more threads

As in Many Computation-hungry Applications

- Three-step approach:
  - Restructure the mathematical formulation
  - Innovate at the algorithm level
  - Tune core software for hardware architecture



Conclusion: Three Options

- Good: “Accelerate” Legacy Codes
  - Recompile/Run
  - Call CUBLAS/CUFFT/thrust/matlab/PGI pragmas/etc.
  - => good work for domain scientists (minimal CS required)
- Better: Rewrite / Create new codes
  - Opportunity for clever algorithmic thinking
  - => good work for computer scientists (minimal domain knowledge required)
- Best: Rethink Numerical Methods & Algorithms
  - Potential for biggest performance advantage
  - => Interdisciplinary: requires CS and domain insight
  - => Exciting time to be a computational scientist

Think, Understand… then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part ☺)

Future Studies

- More complex data structures
- More scalable algorithms and building blocks
- More scalable math models

- Thread-aware approaches
  - More available parallelism
- Locality-aware approaches
  - Computing is becoming bigger, and everything is further away