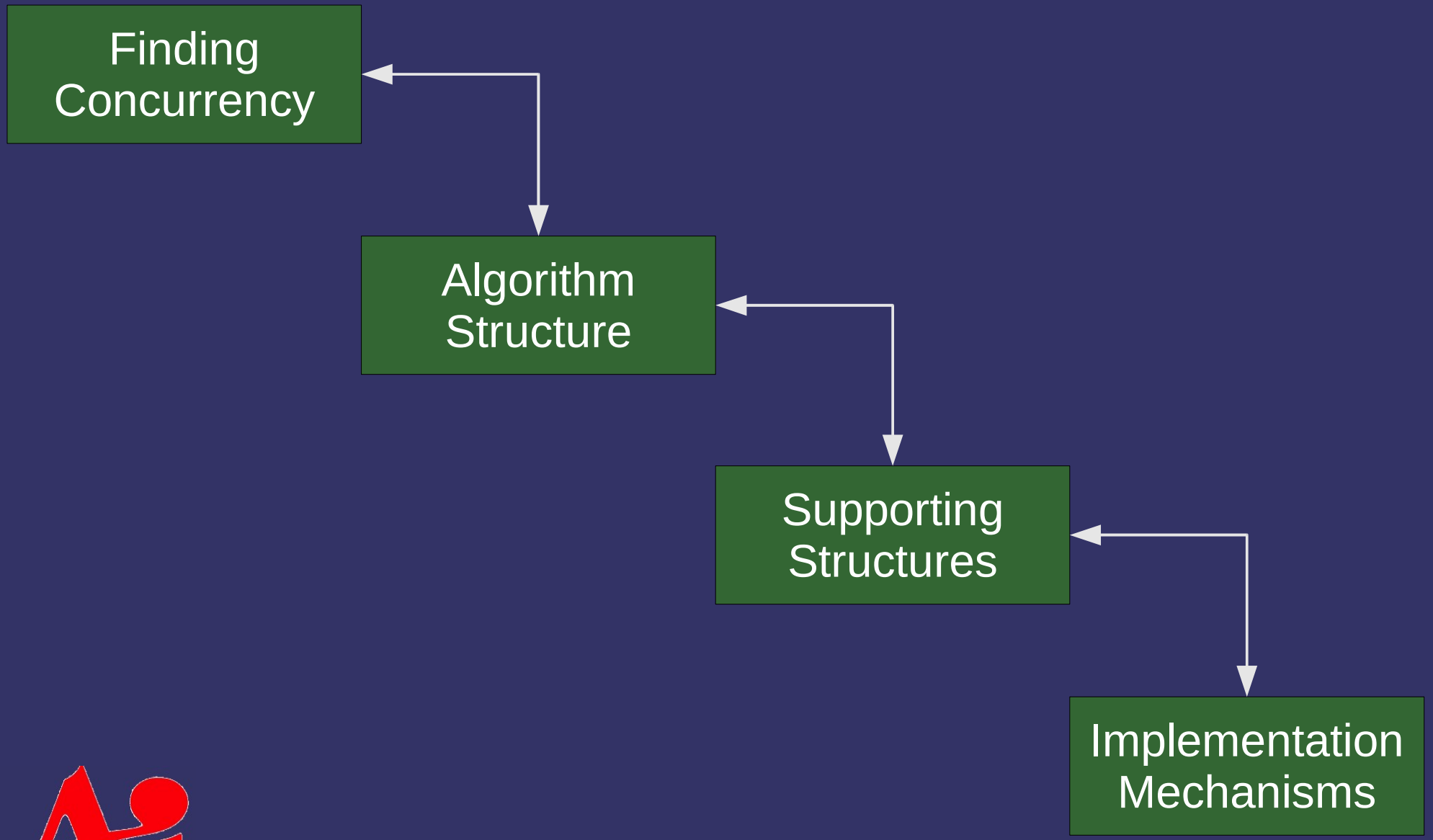
VGP393C – Week 4

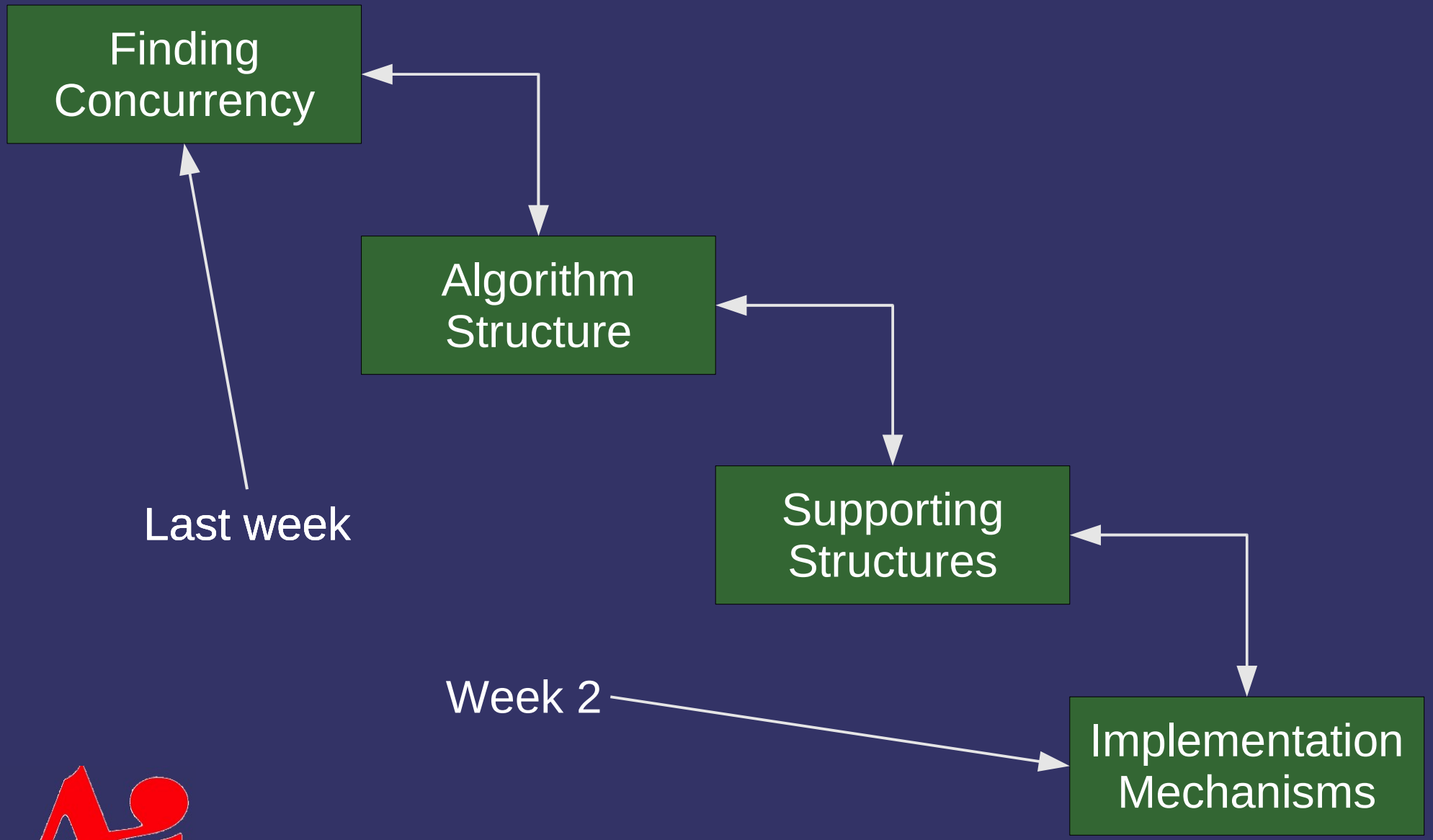
- Agenda:
  - Algorithm Structure
    - Task Parallelism
    - Divide and Conquer
    - Geometric Decomposition
    - Recursive Data
    - Pipeline
    - Event-Based Coordination

Algorithm Structure



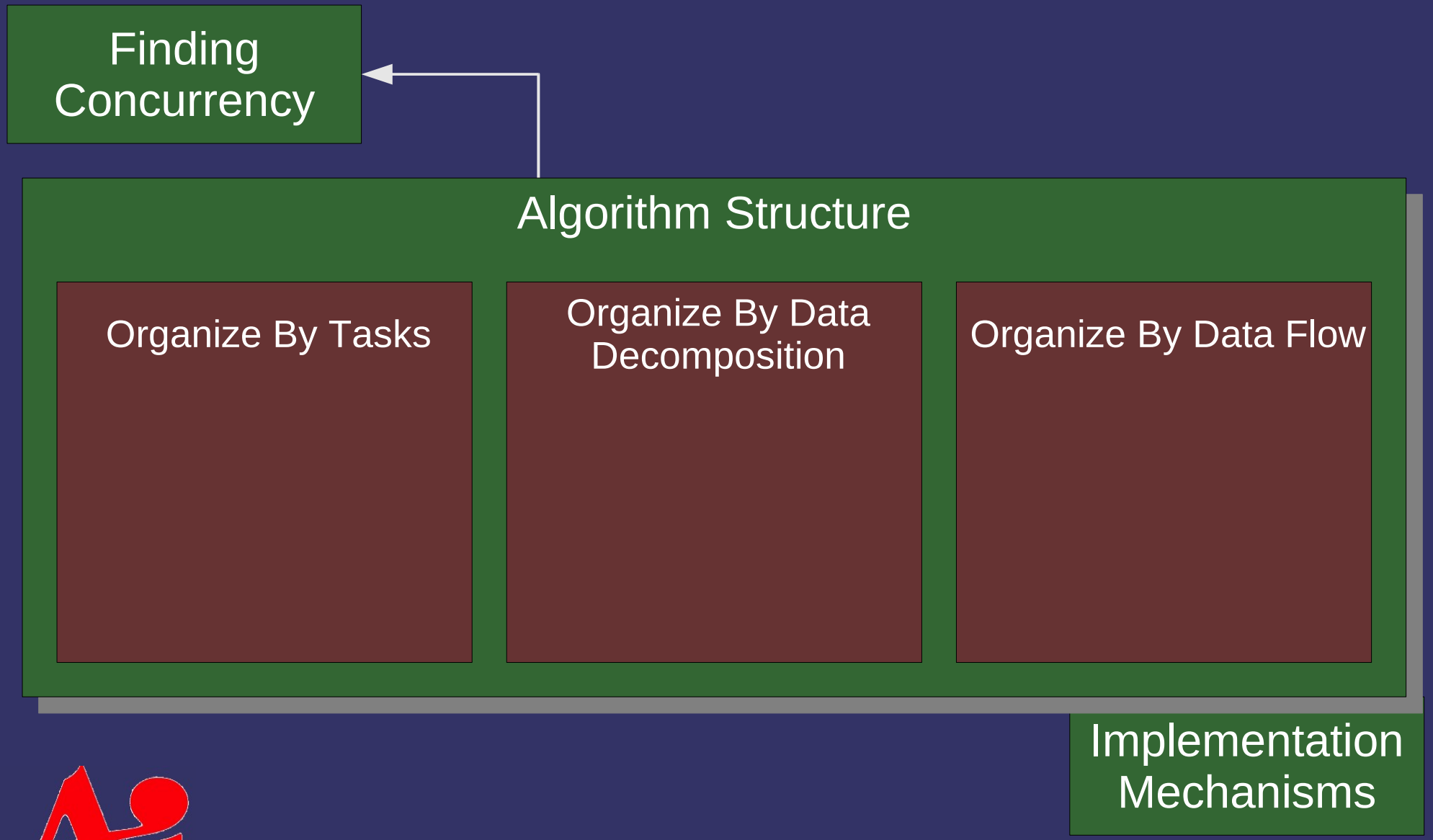
6-August-2008

Algorithm Structure

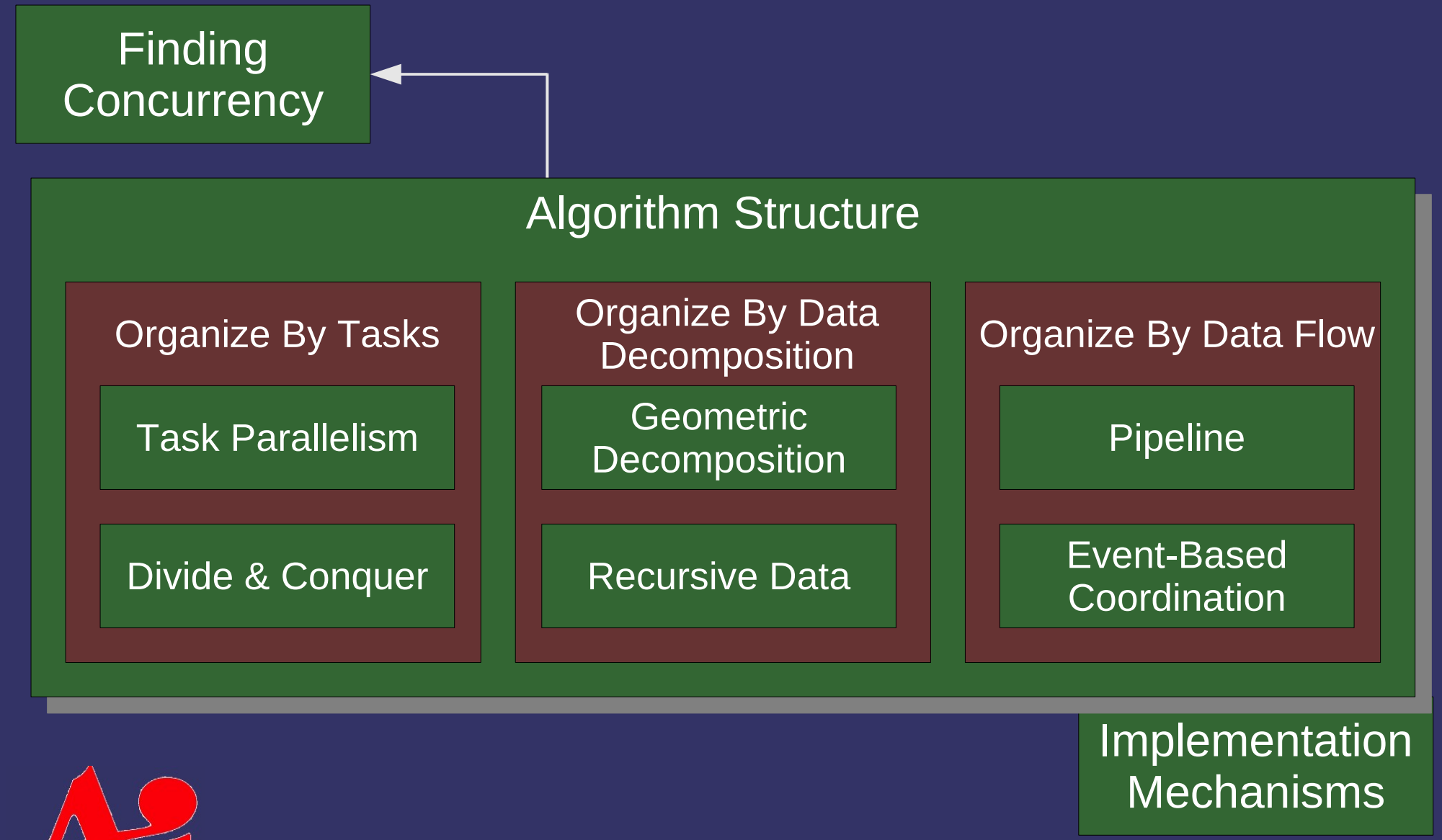


6-August-2008

Algorithm Structure



Algorithm Structure



6-August-2008

Task Parallelism

- Three primary elements:
  - Tasks
  - Dependencies between tasks
  - Scheduling of tasks

Task Parallelism

- Tasks
  - At least as many tasks as UEs...preferably *many* more
  - Computation of each task should outweigh the overhead of managing the task and dependencies

Task Parallelism

- Dependencies
  - Ordering constraints – handled by forcing tasks to execute in a particular order
  - Shared data dependencies – more complex
    - In some cases there are none
    - *Removable dependencies* can be removed by reworking the code (next slide)
    - *Separable dependencies* involve accumulations of partial results into a larger data structure
      - Each UE works in a local, temporary copy and the subresults are accumulated at the end
      - If the partial results are combined to a single element, it is called a *reduction*

Task Parallelism

```
int ii = 0;
int jj = 0;

for (int i = 0; i < N; i++) {
    /* Loop-carried dependencies: the value in
     * iteration X+1 requires knowledge of the
     * value at iteration X.
     */
    ii = ii + 1;
    jj = jj + i;
    d[ii] = first_big_calculation(ii);
    a[jj] = second_big_calculation(jj);
}
```

Task Parallelism

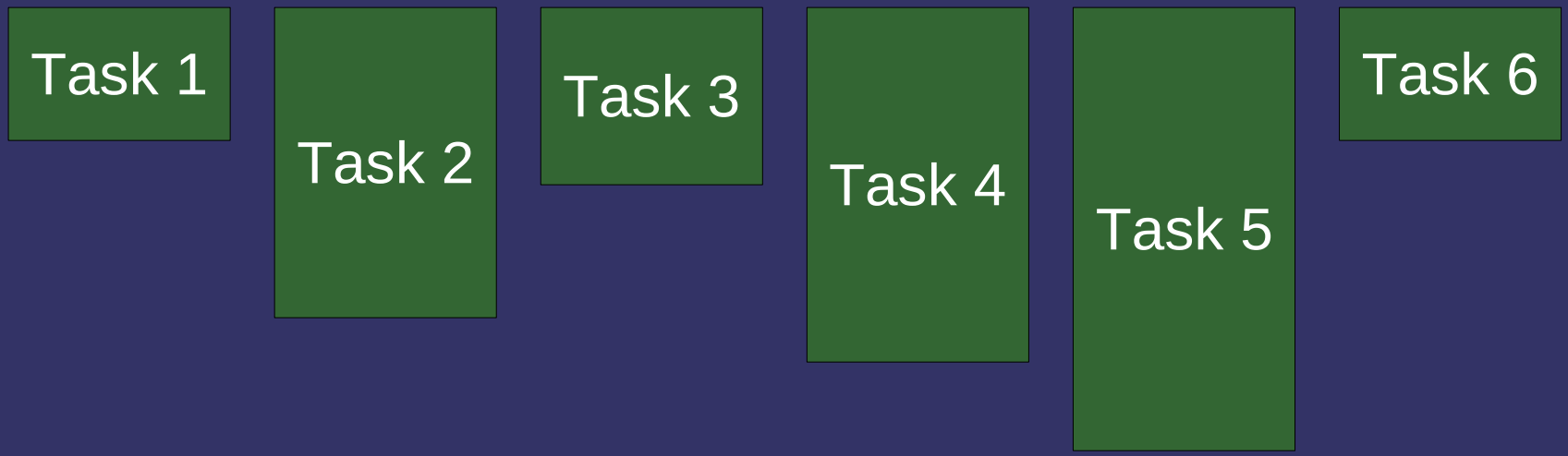
```
for (int i = 0; i < N; i++) {
    /* The values of ii and jj depend only on the
     * loop iteration count...no dependency!
     */

    const int ii = i;
    const int jj = (i * i + i) / 2;
    d[ii] = first_big_calculation(ii);
    a[jj] = second_big_calculation(jj);
}
```

Task Parallelism

- Dependencies
  - “Other” dependencies have to be managed by hand using synchronization primitives
  - We'll talk more about doing this in a sensible way later

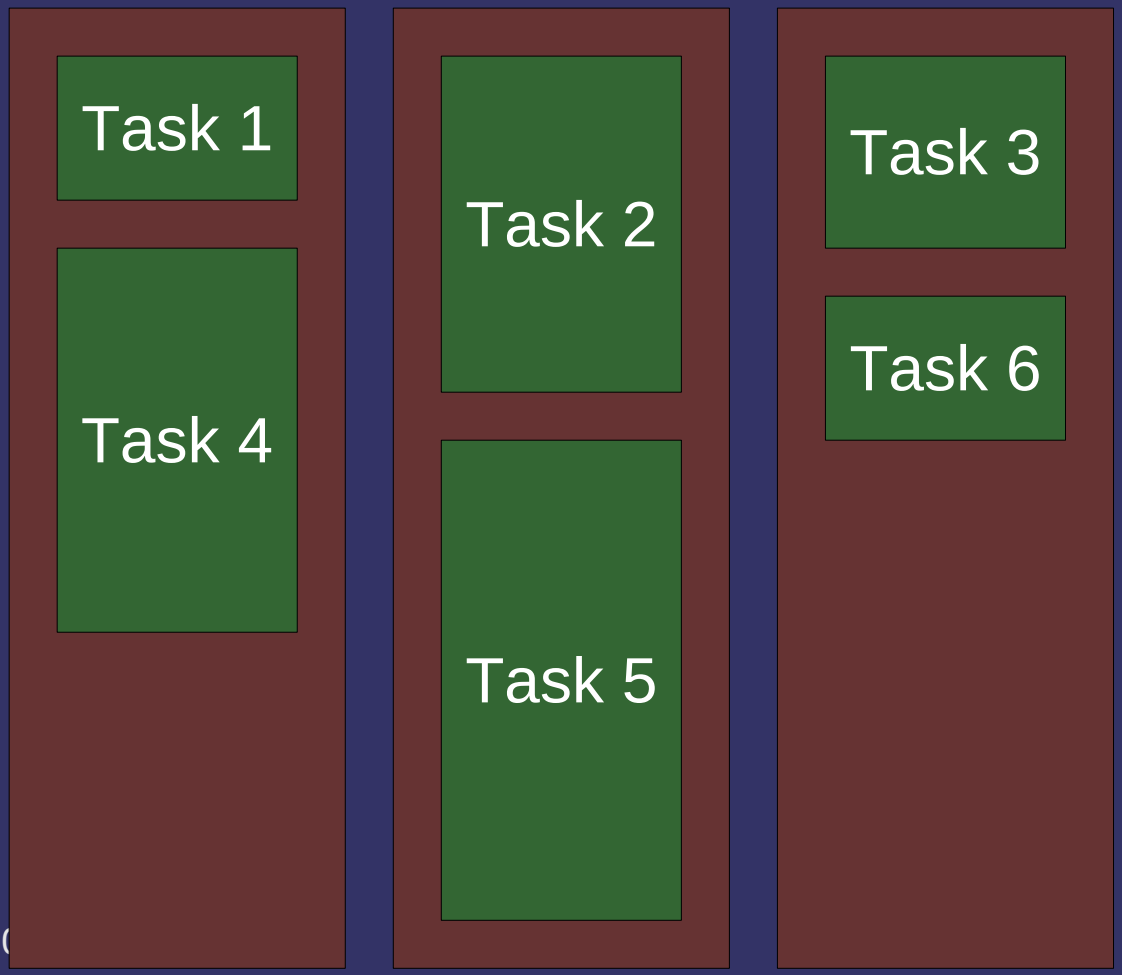
Task Parallelism

- Scheduling, especially in task-parallel programs, can make or break performance



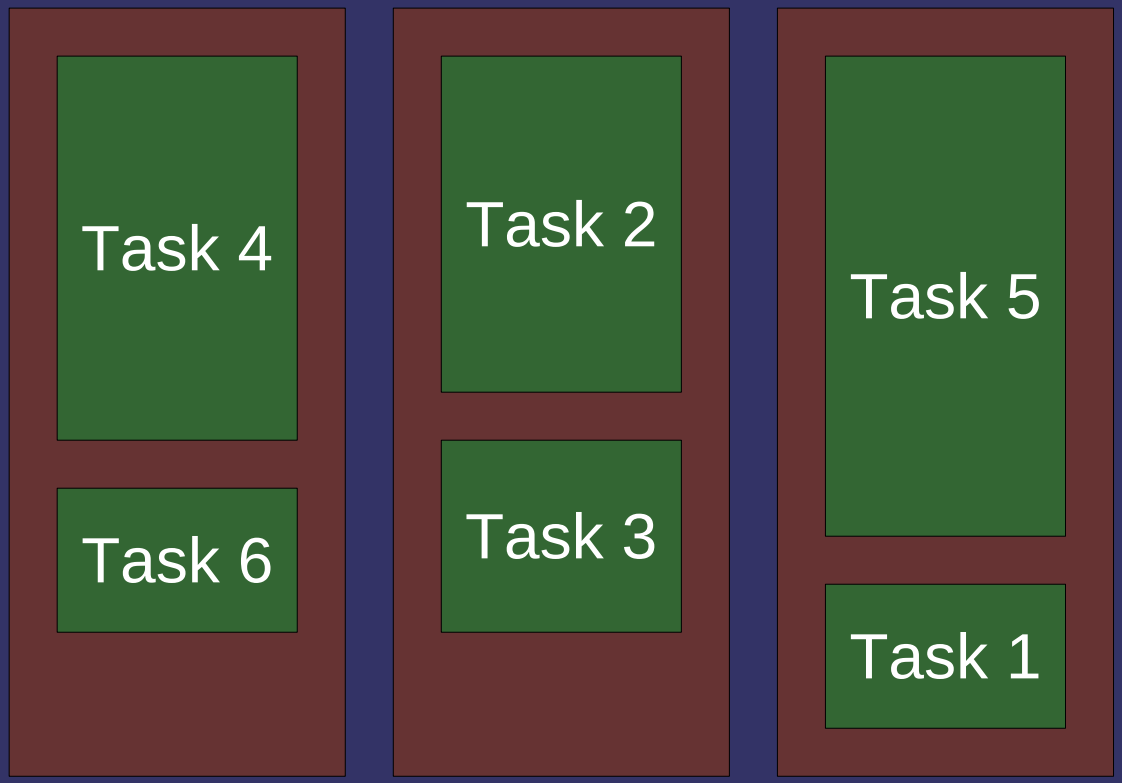
Task Parallelism

- Scheduling, especially in task-parallel programs, can make or break performance



Task Parallelism

- Scheduling, especially in task-parallel programs, can make or break performance



Task Parallelism

- Two general scheduling techniques
  - Static – Tasks are partitioned in the relatively equal sized chunks and statically assigned to UEs
  - Dynamic – Used when either the size of each chunk varies a lot or when the performances of the PEs differ
    - The most common technique is to use a single *task queue* where tasks are added and removed by UEs
    - *Work stealing* enhances this technique

Task Parallelism

- Work stealing attempts to solve two problems with the single task queue
  - Contention on the task queue mutex
  - Poor cache performance

Task Parallelism

- Work stealing attempts to solve two problems with the single task queue
  - Contention on the task queue mutex
  - Poor cache performance
- Each UE has its own task queue
  - UE adds tasks to the head of its TQ
    - Smaller tasks typically end up at the head of the TQ
  - UE removes tasks from the head of its TQ
    - Improves cache performance
  - If a UEs TQ is empty, it *steals* work from the tail of another UE's TQ

Task Parallelism

- Many task-parallel programs are loop-based
  - The primary tasks are individual iterations of a loop
  - Many parallel programming environments have special constructs for this form
  - Known as the *loop parallelism* pattern
- Some jobs don't fit the loop parallelism model
  - Particularly if all tasks are not known in advance
  - Either *master / worker* or *SPMD* is usually a better fit

Divide and Conquer

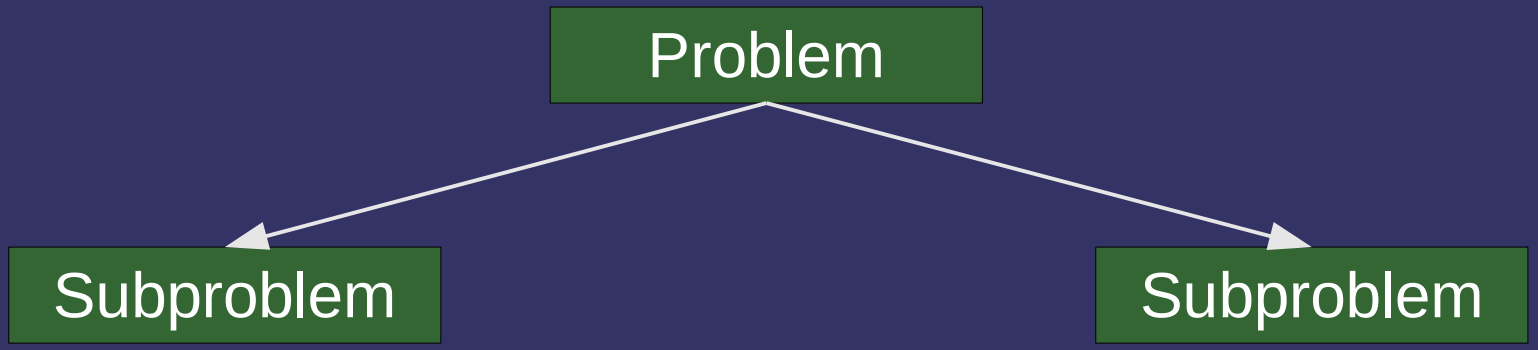
- *Divide and conquer* recursively subdivides problem space in to multiple subproblems. Subproblems are, eventually, solved, and the results combined
  - Very common design method in sequential algorithms
  - Can anyone think of any?
    - Merge sort, QuickSort, *Mandelbrot generators*
  - Divide and conquer algorithms dynamically generate tasks
    - This necessitates dynamic scheduling

Divide and Conquer

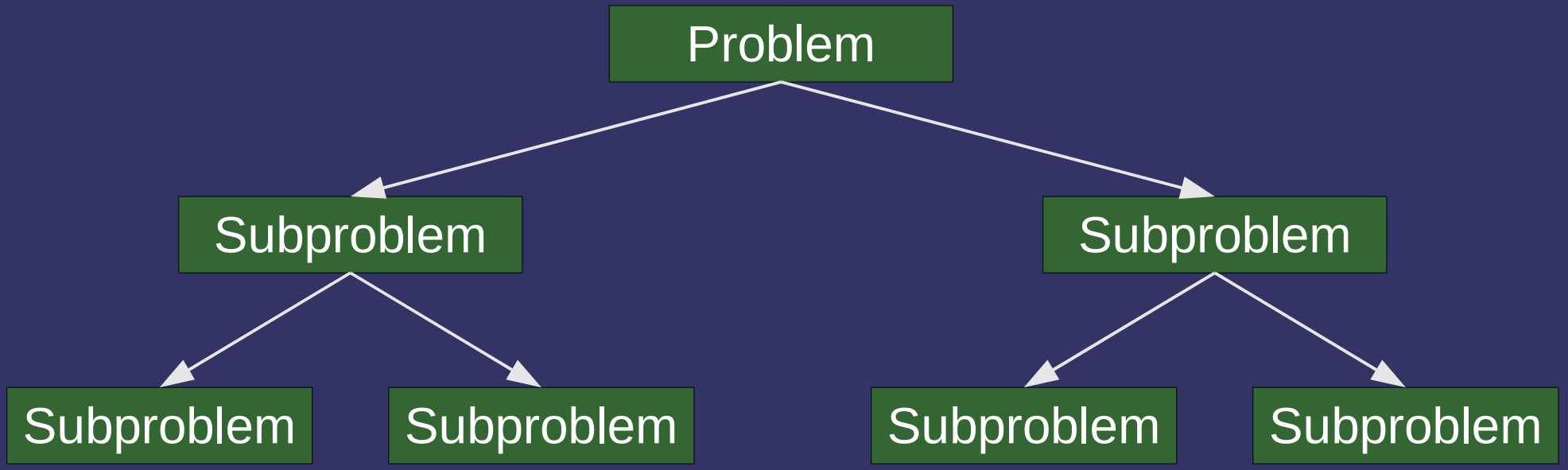
Problem

6-August-2008

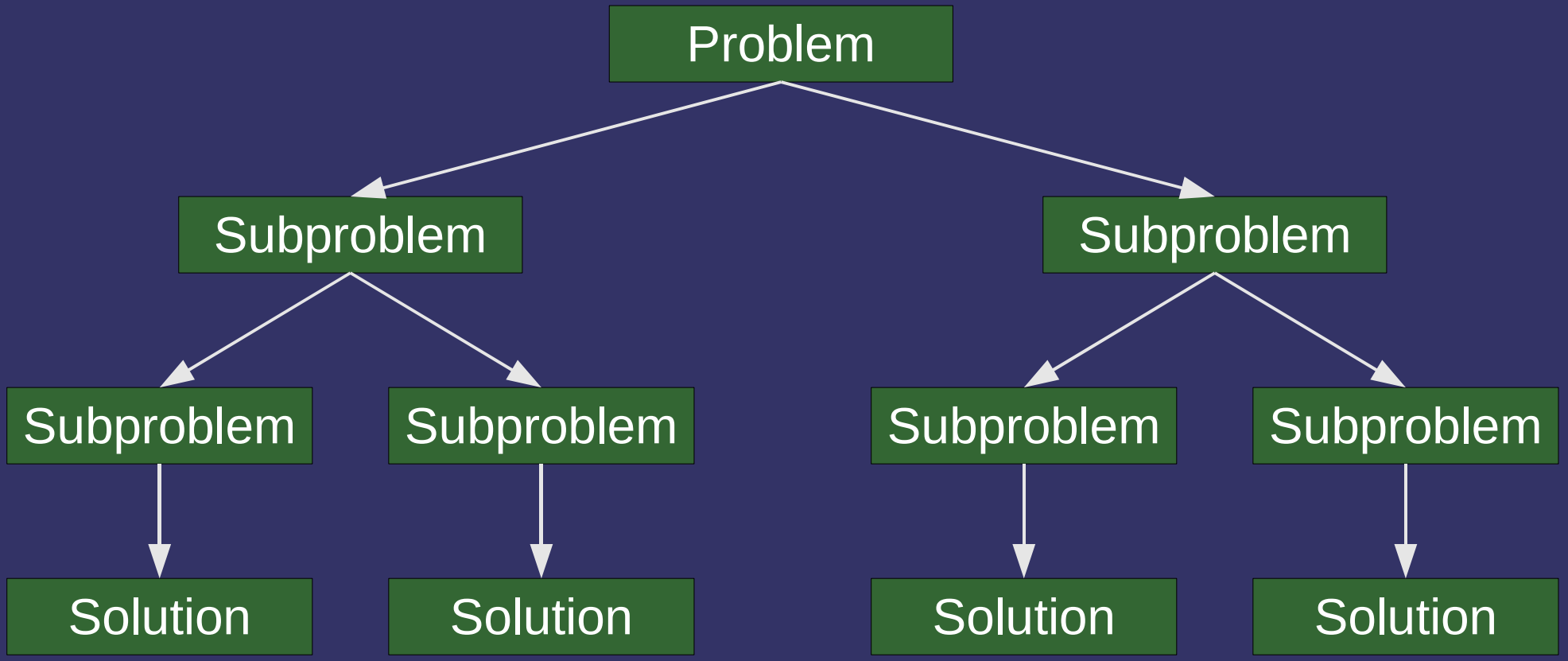
Divide and Conquer



Divide and Conquer

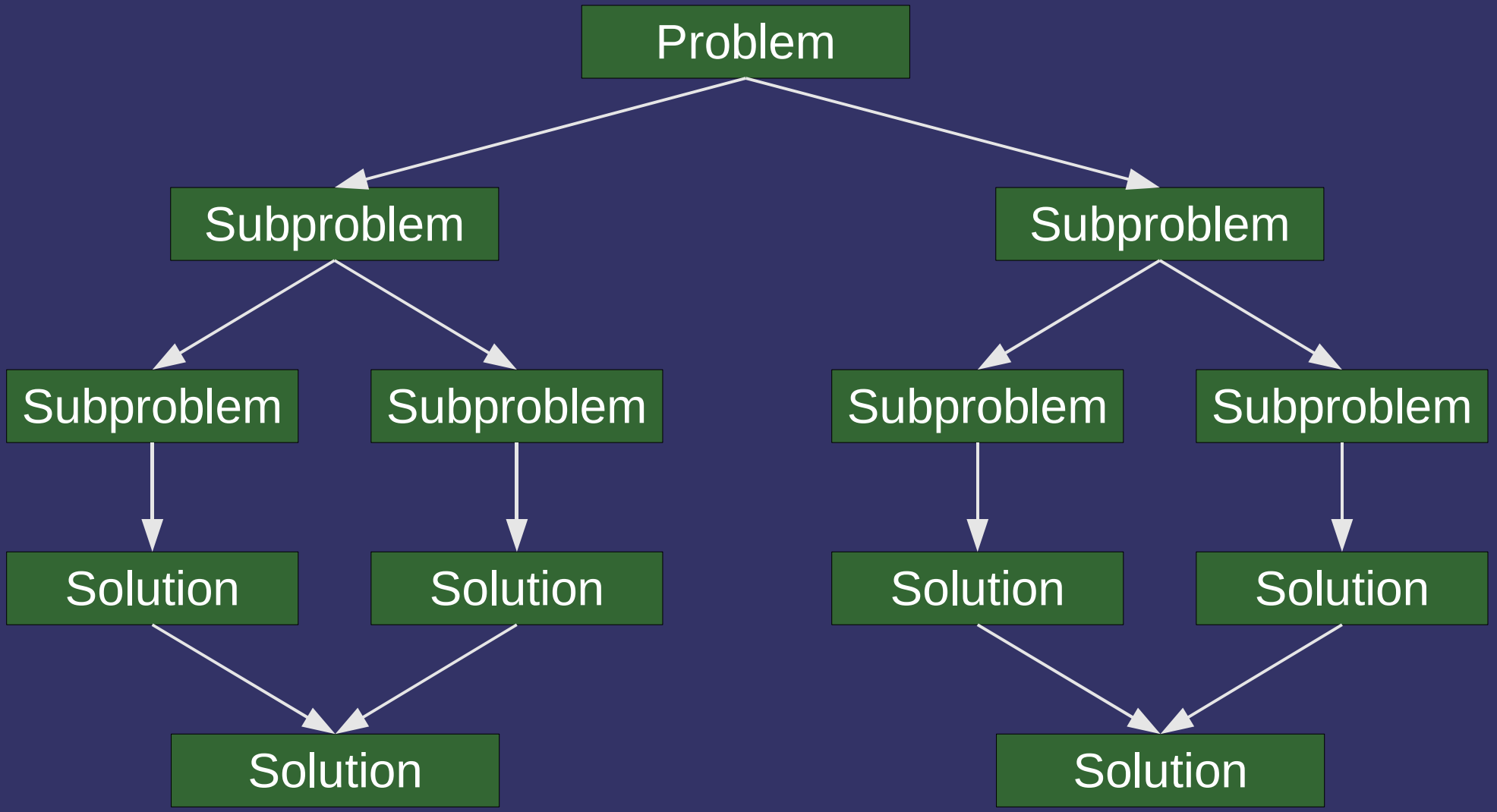


Divide and Conquer



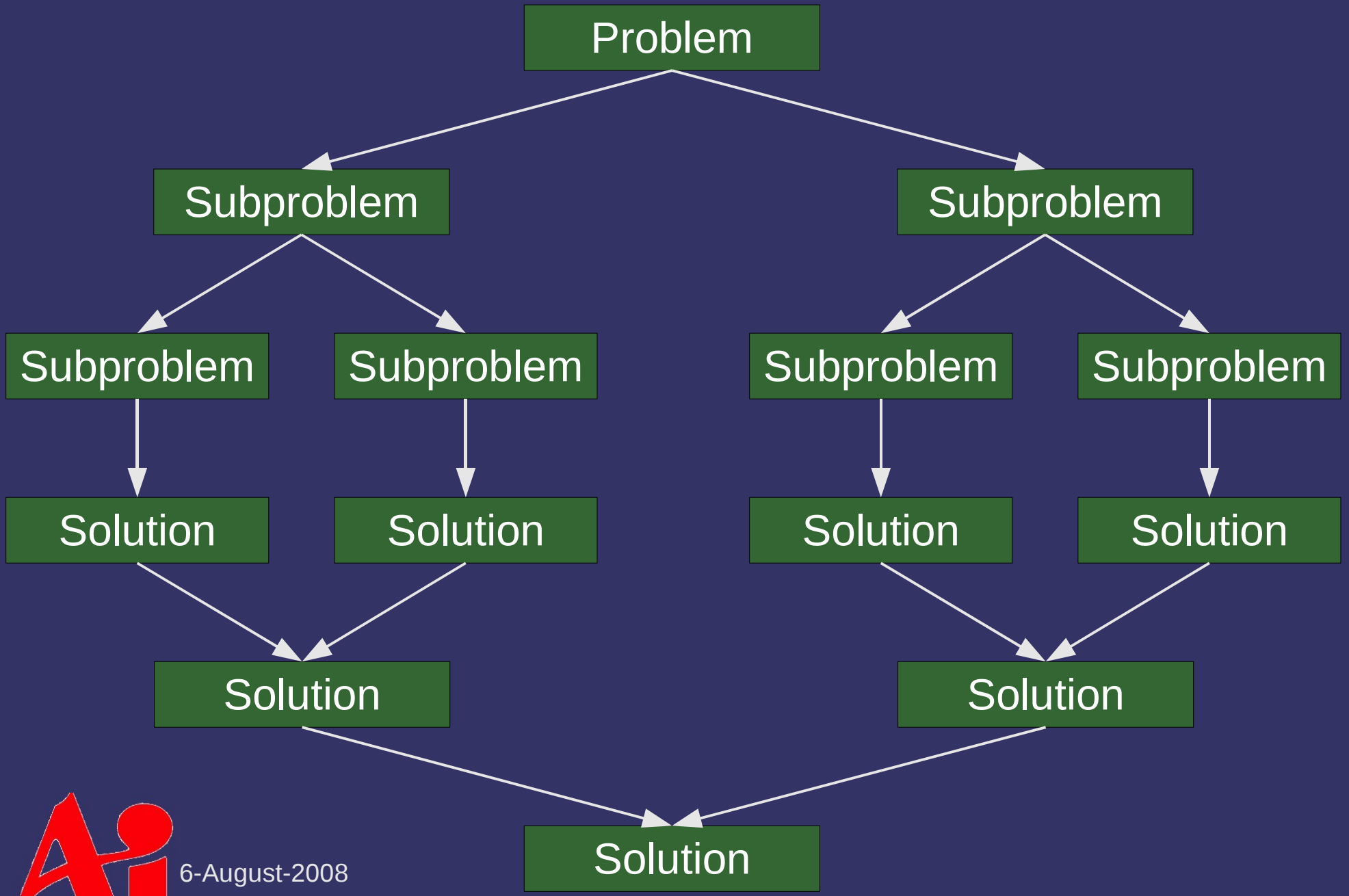
6-August-2008

Divide and Conquer



6-August-2008

Divide and Conquer



6-August-2008

Divide and Conquer

- Concurrency is possible when subproblems can be solved independently
  - As we have found, a sequential D&C algorithm becomes parallel by defining a task for each call to the primary “solve” function
  - At some point the subproblems are small enough that just solving them is faster than creating new tasks
    - May happen *before* it is beneficial to stop subdividing
    - This threshold, or *granularity knob*, should be tunable at run-time

Divide and Conquer

- Can be implemented using the *Fork / Join* pattern
  - Subproblems at each split are roughly the same size
  - Assign each task to a UE
  - Stop splitting when the number of tasks matches the number of PEs
- Can also be implemented using the *Master / Worker* pattern
  - One (or slightly more) UE per PE
  - Queue of tasks

Geometric Decomposition

- Many problems are decomposed by subdividing a large data structure into chunks
  - Arrays and array-like structures can be divided “geometrically” into regions
  - If all subregions are independent, *task parallelism* can be used
  - Many computations require access to data in *neighboring regions*
    - Computed data must be shared between regions for the tasks to complete
    - This is where *geometric decomposition* comes into play

Geometric Decomposition

- Data decomposition granularity is important to overall efficiency
  - Larger chunks results in fewer, larger messages between tasks
    - Reduces messaging overhead
  - Smaller chunks results in more, smaller messages between tasks
    - Increases messaging overhead
    - Simplifies scheduling...especially if there are many more chunks than PEs
  - Experimentation is usually required to find a balanced chunk size

Geometric Decomposition

- Chunk “shape” is also important
  - Data is usually only shared along common boundaries between chunks
  - A 2D array divided in long, thin rectangles will have more boundary regions than one divided into squares
    - The so-called *surface-to-volume* effect
- Chunk shape may be determined by other factors
  - Reuse of sequential code
  - Other portions of the parallel program
  - etc.

Geometric Decomposition

- Data duplication can improve communication performance
  - Extra copies of boundary data can be kept for neighbor tasks to read
    - May be called *ghost boundaries* or *shadow copies*
  - Double buffer can also be used

Geometric Decomposition

- Non-local data required for a computation must be available before that computation can begin
  - If all shared data is ready at the beginning of a computation “phase,” it can be exchanged all at once, up front
  - Data exchange and computation can also proceed concurrently
    - Updating the “interior” data that does not rely on the neighbor's boundary data
    - In cases where some data is not yet available at the start of the computation phase

Geometric Decomposition

- Partition data into chunks, distribute chunks to UEs
  - Simple
  - Can lead to poor performance if per-chunk work is unbalanced or becomes unbalanced as computation progresses
  - Generating many more chunks that UEs and assigning multiple “random” chunks to a single UE can help
- Can dynamically redistribute chunks among UEs
  - Can cause a lot of overhead
  - Can increase cache-miss rate

Recursive Data

- Recursive data structures are often difficult to operate on concurrently
  - Serial traversal of the structure must be converted to one that allows concurrent operation
    - Usually *increases* the total amount of work
  - Problem conversion may be difficult in the first place
    - Requires looking at well-known problems is odd ways
    - May result in a really complex algorithm
  - May be difficult to exploit the exposed concurrency
    - Communication overhead may be difficult to overcome

Recursive Data

- The data structure is decomposed into one element per task
  - Simplest method is to assign one task per UE
  - If there are too many UEs per PE, the performance will be poor
- Result *usually* looks like a loop that operates on every element of the structure simultaneously
  - Good fit for classic vector computers!
  - Can cause synchronization headaches
    - "Double buffering" pointers (i.e., `next` pointer in a linked list) is often helpful

Recursive Data

- Example

Pipeline

- The classic “assembly line”
  - Improves *throughput* not *latency*
  - Requires many more work items than pipeline stages to be efficient

Pipeline

- One pipeline stage per task
  - Concurrency is limited by the number of stages
  - Task size should be relatively equivalent
    - Otherwise some stages will finish and sit idle
    - More time consuming stages can also be parallelized
  - *Fill time* and *drain time* should be relatively small compared to total running time
- Program structure is important
  - SPMD (next week) with a switch statement
  - OOP where each stage is a subclass with a `do_work` method

Pipeline

- The pipeline is all about *data flow*
  - How data flows from one stage to the next will dominate the program design
  - Several common techniques:
    - Buffered, ordered message passing
    - Shared queue
- Flow is more complex if stages are also parallel
  - Consider a stage with 4 parallel units sending data to a stage with 5 parallel unit
  - Usually have an aggregation / disaggregation stage in between
    - May be necessary to ensure data flows in the correct order

Pipeline

- Typically assign one stage per PE
  - Some stages can also operate on special purpose hardware
    - Encryption accelerators, graphics accelerators, etc.
  - If there are fewer PEs than stages, assign stages with different resource uses to the same PE
    - Assign compute intensive stage and an I/O intensive stage to the same PE
  - Otherwise assign adjacent stages to the same PE
    - More cache friendly

Event-Based Coordination

- Collection of semi-independent tasks that operate in a non-linear order
  - Think of the pipeline as a directed graph without loops
  - Event-based coordination is a directed graph *with* loops
- Each task receives an event, processes it, and *possibly* sends out other events
  - Asynchronous communication is *required*
    - Shared queue is your friend

Event-Based Coordination

- Events must be processed in the proper order
  - Tasks may not be able to process events in the order received
    - The oldest event in the system may need to be processed first
    - Tasks may have to wait to process one event until *after* receiving a different event

Event-Based Coordination

- Out-of-order events can be handled either *optimistically* or *pessimistically*
  - Optimistic assumes it's okay to process events in the order received
    - May need a way to “back out” events processed out of order
  - Pessimistic ensures that events are only processed in order
    - Can add extra latency waiting for missing events
    - Can add extra communication to be sure that no events are on the way

Next week...

- NO CLASS NEXT WEEK!
  - Meet again on 8/20
- Quiz #2
- Assignment #2 due
- Supporting Structures
  - SPMD
  - Master / worker
  - Loop parallelism
  - Shared Queue
  - etc.

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.

6-August-2008