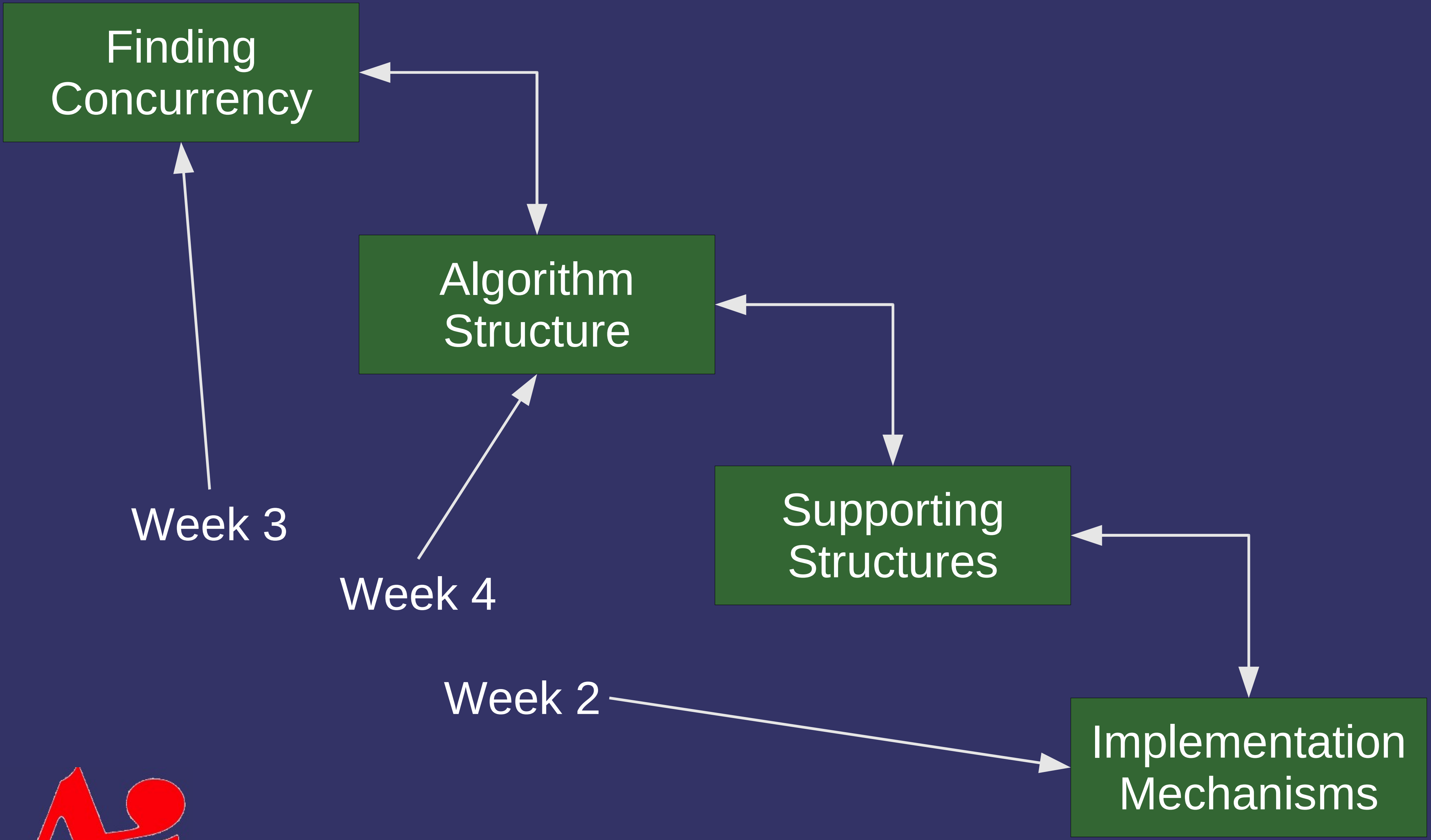
VGP393C – Week 5

- Agenda:
  - Quiz #2
  - Supporting Structures
    - SPMD
    - Master / worker
    - Loop parallelism
    - Shared Queue
    - etc.
  - Assignment #2 due
  - Assignment #3 started

20-August-2008

Supporting Structures



Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

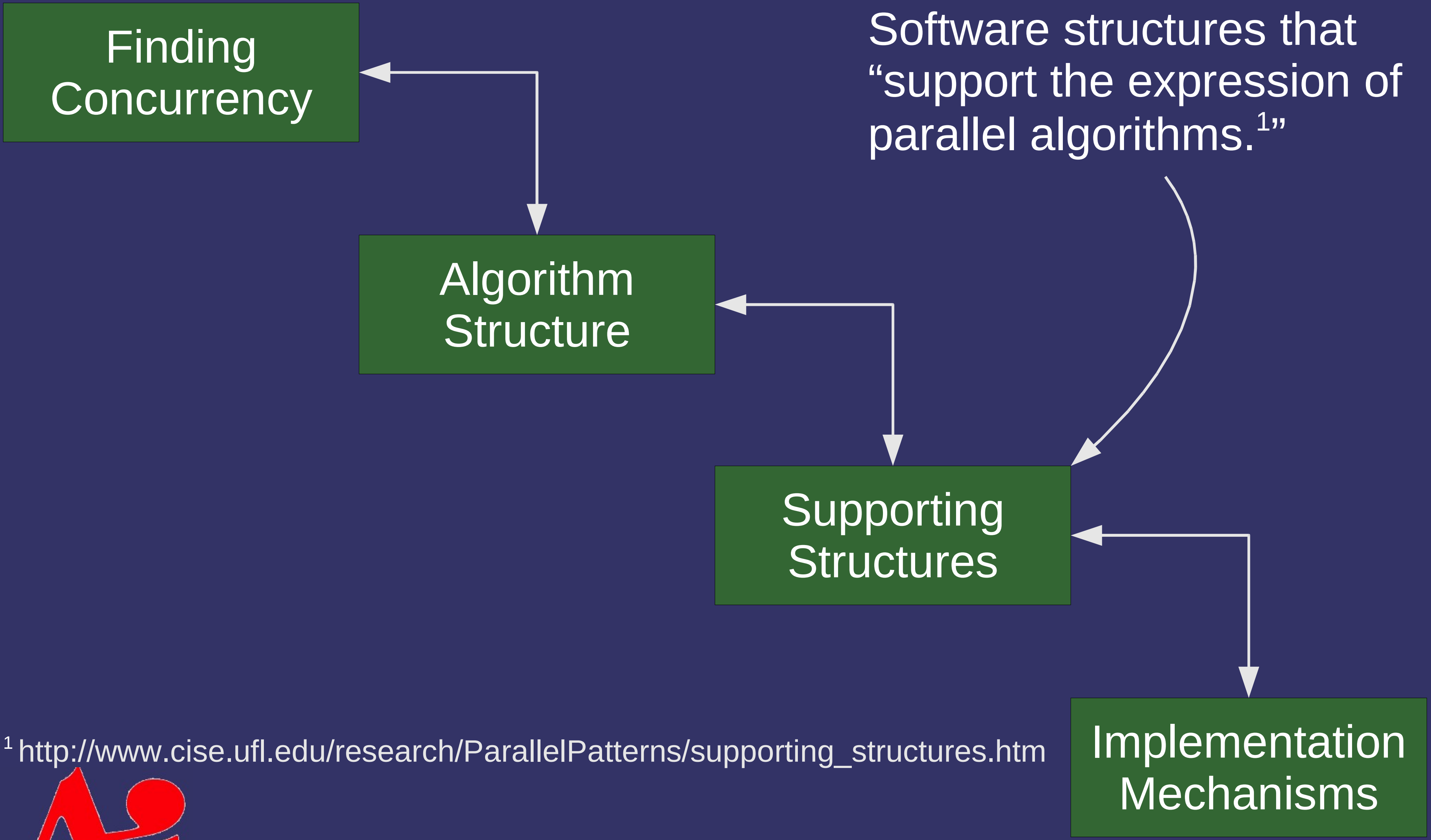
Week 3

Week 4

Week 2

20-August-2008

Supporting Structures

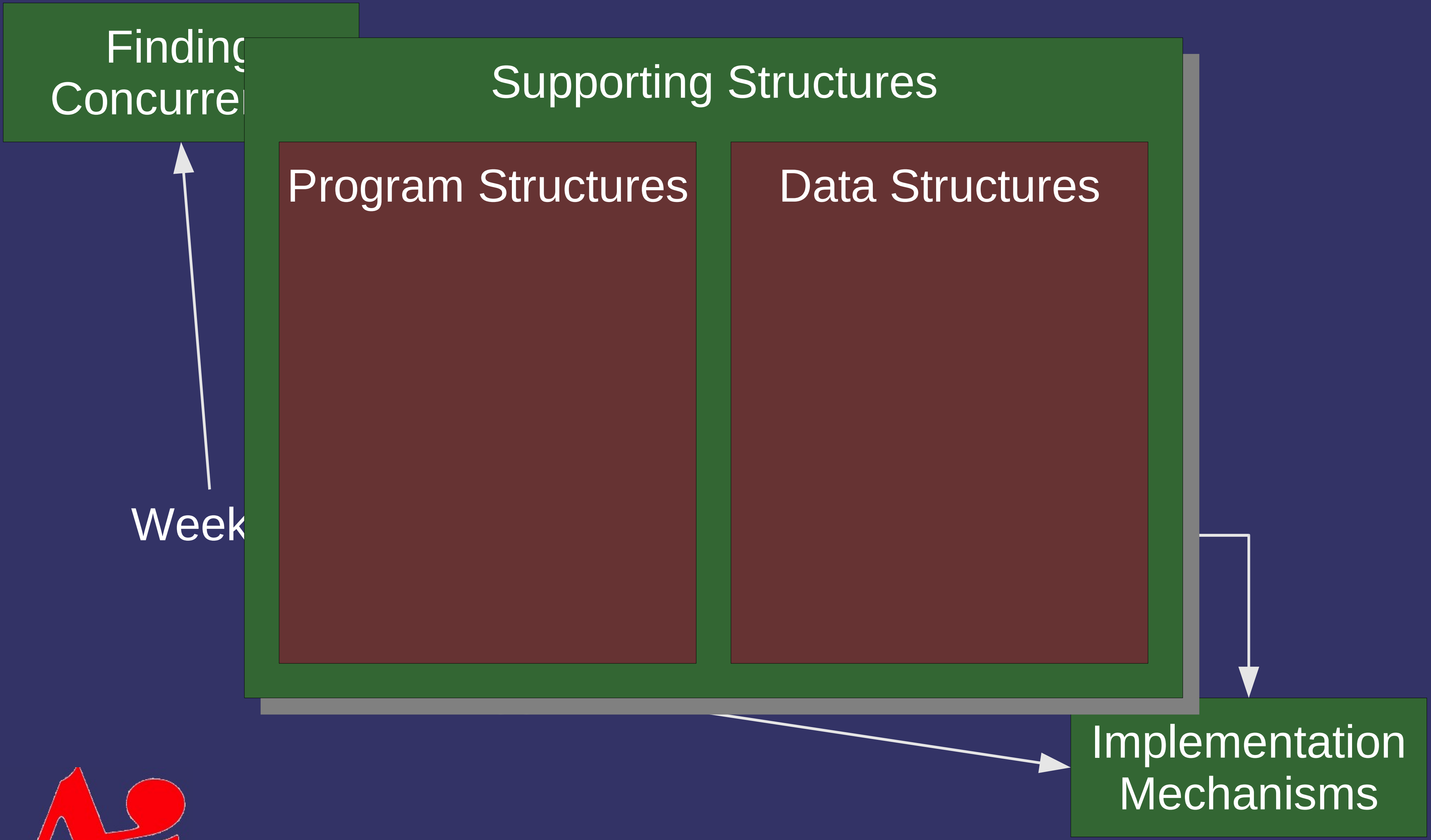


Software structures that "support the expression of parallel algorithms.[1]"

[1] http://www.cise.ufl.edu/research/ParallelPatterns/supporting_structures.htm

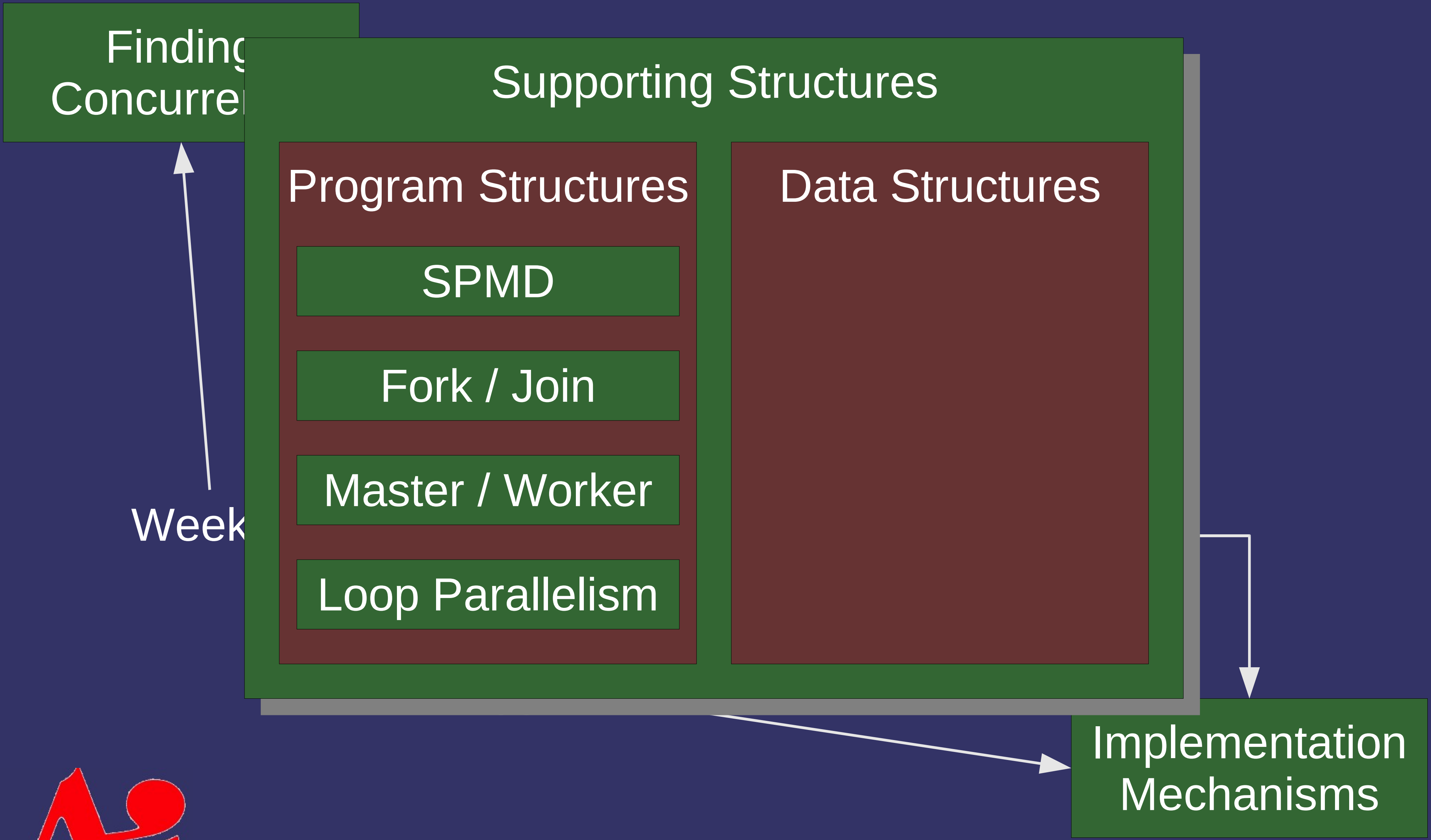
20-August-2008

Supporting Structures



20-August-2008

Supporting Structures



Finding
Concurre

Week

Supporting Structures

Program Structures

- SPMD
- Fork / Join
- Master / Worker
- Loop Parallelism

Data Structures

Implementation Mechanisms

SPMD

- In the *single program, multiple data* (SPMD) pattern, N UEs execute the same code concurrently on different data
  - Each UE may have a unique ID that is also considered to be "different data"

SPMD

- Fundamental question: can the computation be structured so that UEs can be setup at the start and persist through the life of the program?
  - Implicitly requires that the same (or nearly same) code can be used on all data
  - Plays well with concurrency based on data decompositions

SPMD

- Advantages:
  - Avoids thread creation / destruction costs implicit in other patterns
  - Easy reuse of sequential code
    - Each thread is, basically, a copy of the sequential version

SPMD

- Common structure of SPMD programs:
  - Bootstrap – perform the “serial” initialization operations
  - Set unique IDs – Each UE gets some sort of unique identifier. This is usually passed in, and is often derived from the thread ID.
  - Run program on each UE – Each UE can use its unique ID to achieve different behavior
  - Distribute data – Each UE receives its unique data using its unique ID
  - Finalize – perform the “serial” shutdown procedures

Fork / Join

- A single, “master” UE creates additional UEs (forks) and waits for them to complete (joins)
  - Names “fork” and “join” come from the name of the old Unix process creating and wait-for-completion functions
  - *Implies* that threads are created and terminated, but this is not strictly necessary
  - The join can be implemented as a true join or a barrier

Fork / Join

- Fundamental questions:
  - How is data partitioned into local and global blocks?
  - How do the UEs interact?
  - What does the “master” thread do while waiting?
  - How are tasks mapped to UEs?

Fork / Join

- Two common task mappings:
  - “Direct” mapping – UEs have one task mapped
  - “Indirect” mapping – Tasks are dynamically assigned to threads
    - Thread creation and destruction is expensive
    - This cost is mitigated by creating a static pool of threads
    - Threads are “mapped” as needed by sending them tasks
    - Usually one UE per PE

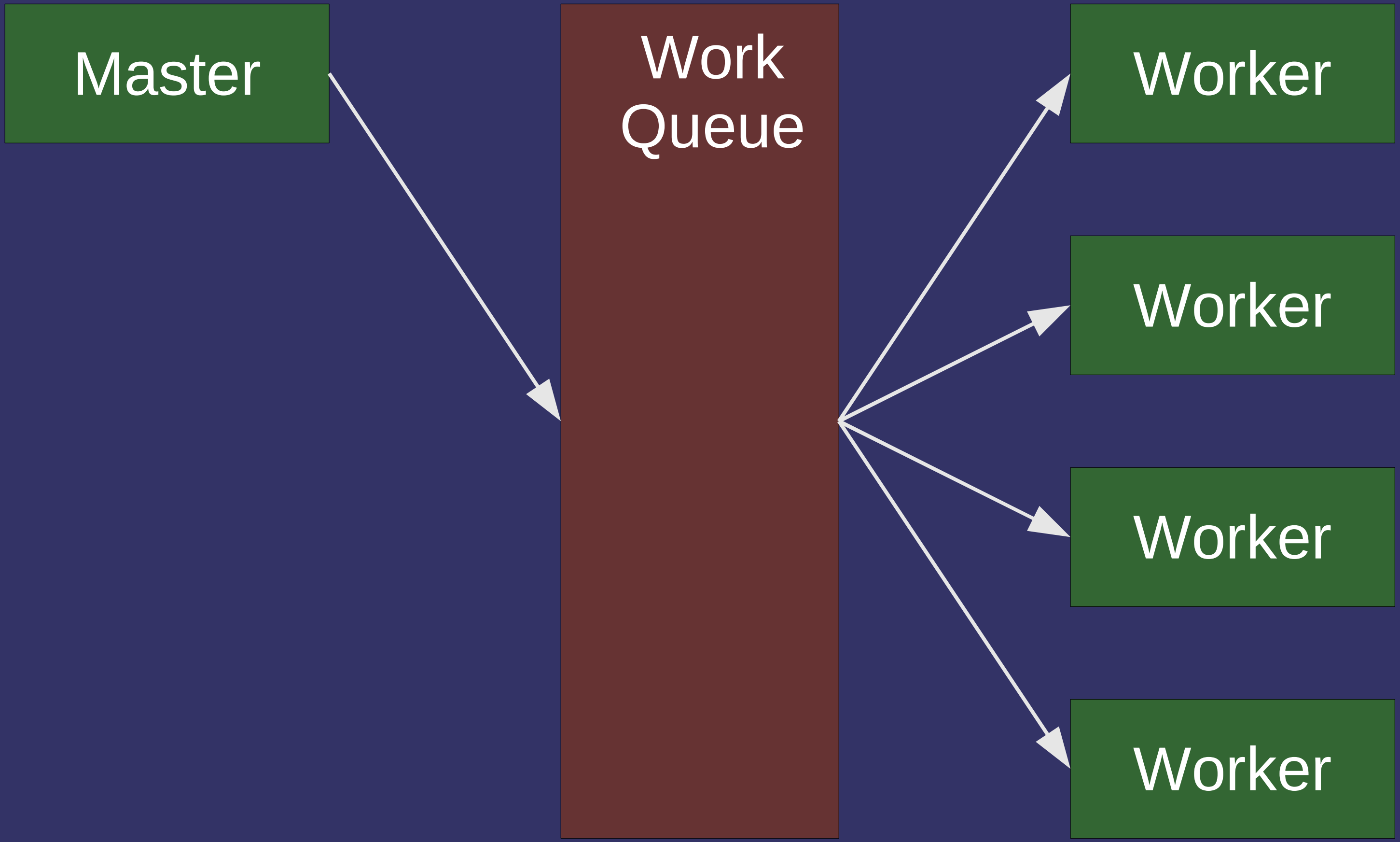
Fork / Join

- Conceptually similar to SPMD
  - Fork / Join can be used at multiple levels within the larger program, but SPMD is a top-level structure
  - SPMD fixes the number of UEs at the start
  - All UEs in SPMD perform the same computation

Master / Worker

- Master / worker pattern works well when:
  - Per-task work loads are variable and unpredictable
    - i.e., static scheduling doesn't work well
  - Computationally intensive part of the program isn't a loop or loop-like
  - Computer power of available PEs varies
    - As is the case with some SMT implementations

Master / Worker



Master

Work Queue

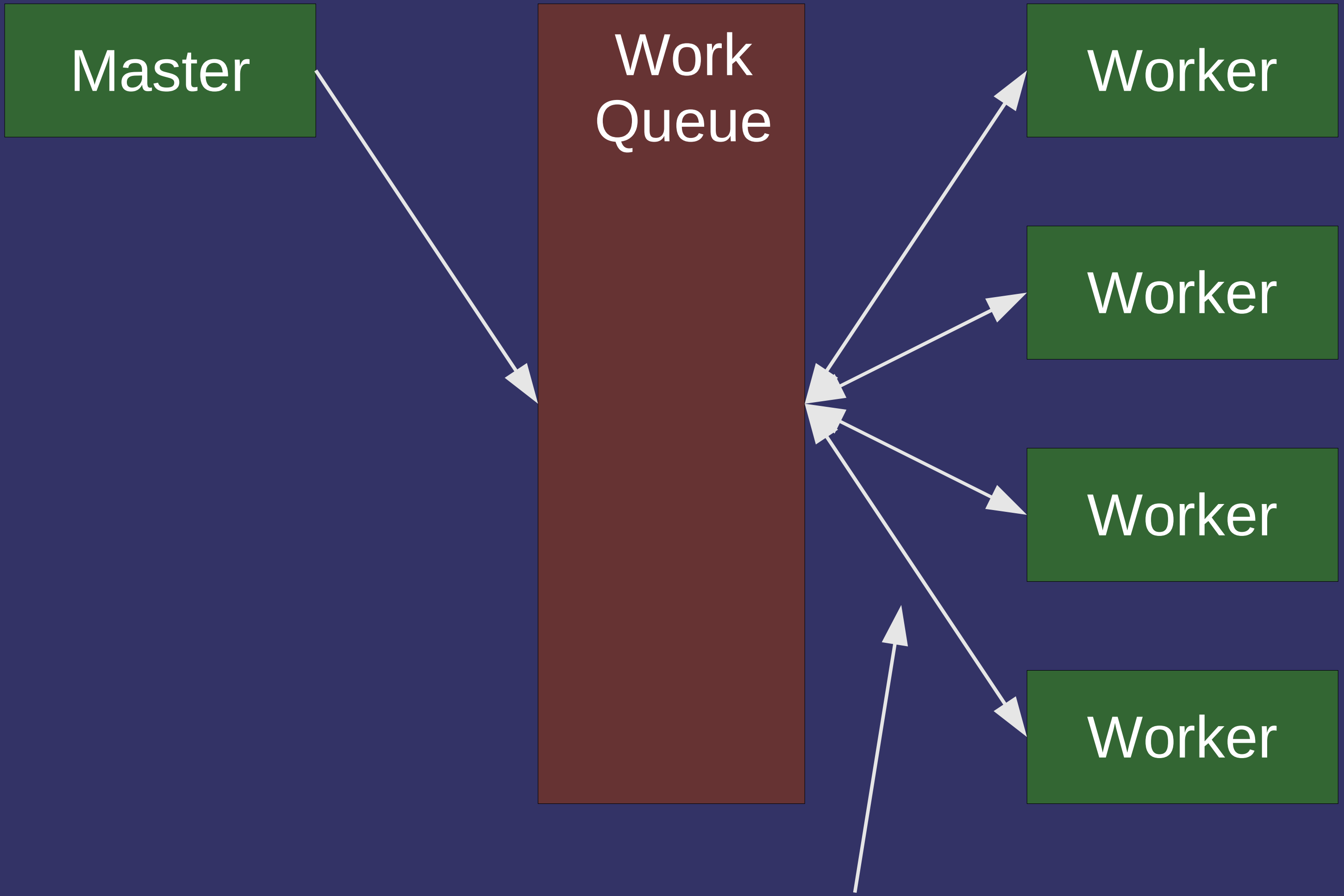
Worker

Worker

Worker

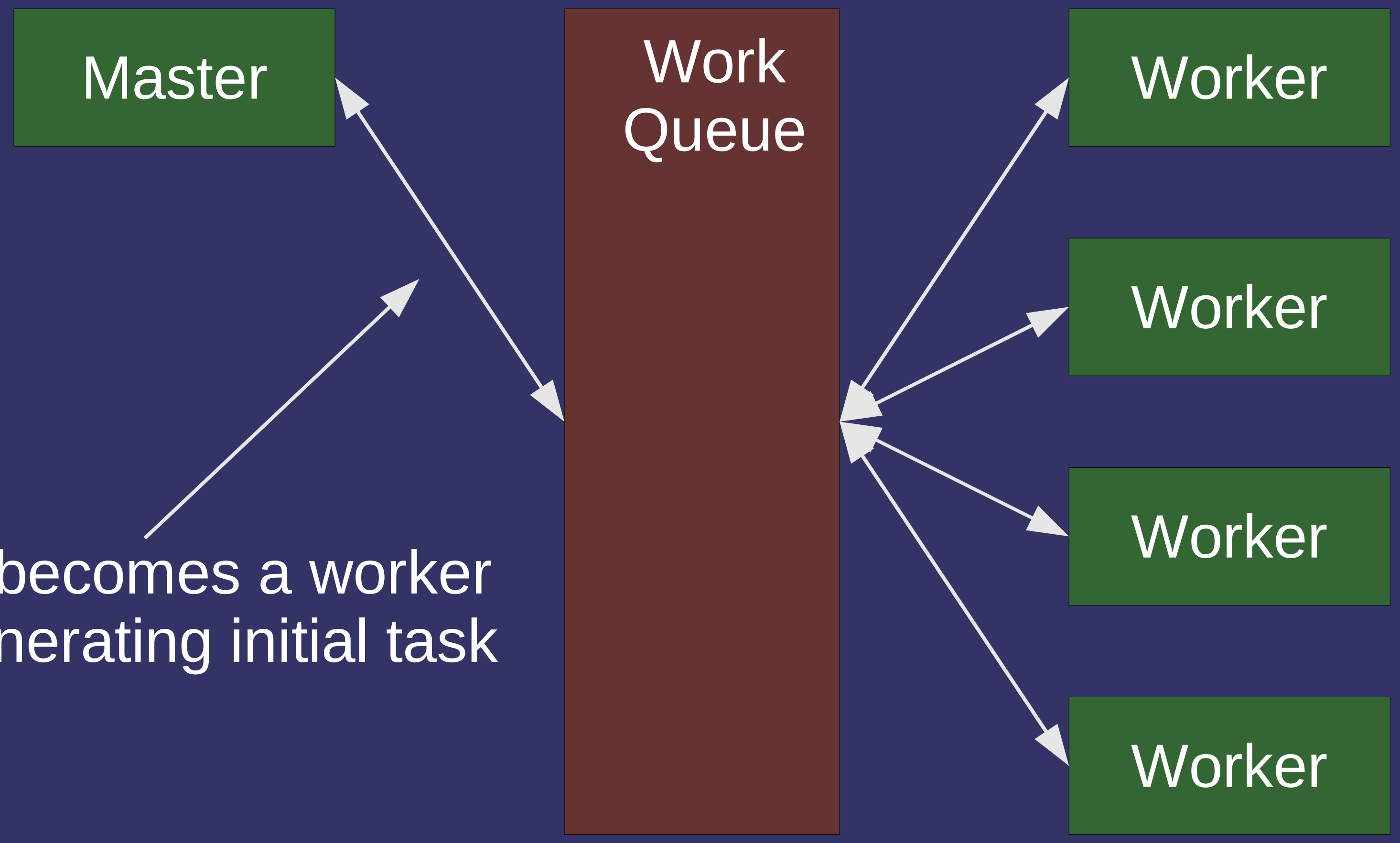
Worker

Master / Worker



Workers generate additional tasks

Master / Worker



Master

Work Queue

Worker

Worker

Worker

Worker

Master becomes a worker after generating initial task

Master / Worker

- Fundamental question: How do workers determine computation is complete?
  - We've already encountered this problem in the Mandelbrot fractal generator
- Several possible strategies for simpler cases:
  - If all work is known at the start, workers can terminate when the work queue is empty
  - Master or a worker can detect completion and add a *poison pill* task to the queue
  - Tree-like computation can hierarchically determine that computation has completed
    - Propagate completion “up” the tree

Loop Parallelism

- Many programs have a small number of computationally expensive loops

Loop Parallelism

- Advantages:
  - Sequential equivalence – Parallelized loops can easily be serialized. This makes code easier to test, debug, and maintain
  - Incremental parallelization – One loop can be parallelized at a time. Step-by-step parallelization allows incremental test and allows parallelization efforts to stop when the program is “fast enough.”

Loop Parallelism

- Initial steps:
  - Find the “hot spots”
  - Eliminate loop-carried dependencies
  - Parallelize the loops
  - Optimize scheduling
- Additional transformations:
  - Merge loops
  - Coalesce nested loops

Loop Parallelism

```
for (i = 0; i < N; i++) {
    do_some_work(i);
}

/* code that does not depend on the results of
 * the above loop and that the following loop
 * does not depend on
 */
...

for (i = 0; i < N; i++) {
    do_other_work(i);
}
```

Loop Parallelism

```
for (i = 0; i < N; i++) {
    do_some_work(i);
    do_other_work(i);
}

...
```

This transformation can happen *before* making the loop parallel...much easier to test!

- More work in each iteration (task) reduces the total parallel overhead

Loop Parallelism

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        do_some_work(i, j);
    }
}
```

Loop Parallelism

```
for (i = 0; i < N * M; i++) {
    do_some_work(i / M, i % M);
}
```

This transformation can happen *before* making the loop parallel...much easier to test!

- More iterations (tasks) simplifies scheduling and improves load balancing

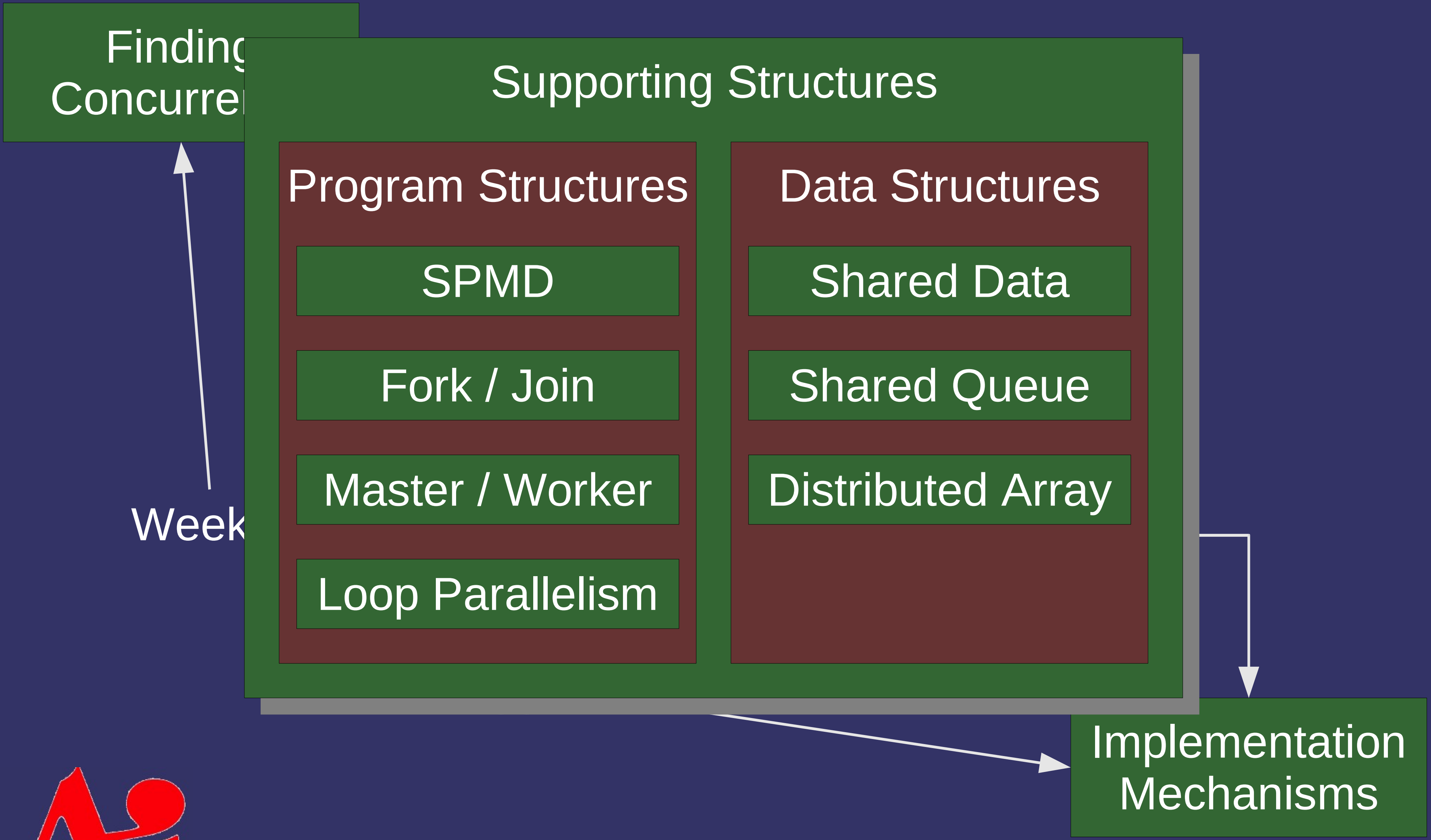
Pattern Selection

	Task Parallelism	Divide and Conquer	Geometric Decomp.	Recursive Data	Pipeline	Event-Based Coord.
SPMD	☺☺☺☺	☺☺☺	☺☺☺☺	☺☺	☺☺☺	☺☺
Loop Parallelism	☺☺☺☺	☺☺	☺☺☺			
Master / Worker	☺☺☺☺	☺☺	☺	☺	☺	☺
Fork/ Join	☺☺	☺☺☺☺	☺☺		☺☺☺☺	☺☺☺☺

Table from “Patterns for Parallel Programming,” p. 125.

Break

Supporting Structures



Finding Concurrer

Supporting Structures

Program Structures

- SPMD
- Fork / Join
- Master / Worker
- Loop Parallelism

Data Structures

- Shared Data
- Shared Queue
- Distributed Array

Week

Implementation Mechanisms

20-August-2008

Shared Data

- Many techniques exist to reduce data shared by tasks
  - Careful partitioning
  - Replication
  - Etc.

Shared Data

- Warning signs:
  - Some data structure is accessed by multiple tasks during computation
  - Some task modifies the data structure
  - Some task needs the modified value in the computation
- Example: the task queue in the master / worker pattern

Shared Data

- Verify that the data really is shared
  - Much effort is required to ensure proper arbitration of shared data and correct results
  - Synchronization adds overhead
  - Many synchronization methods implicitly limit scalability
  - Resulting code can be difficult for other to understand and maintain
    - And for the original developer to debug!

Shared Data

- Start with an abstract data type
  - Abstracting the interface to the data keeps all of the synchronization in one place
  - Makes it easier to change synchronization methods
    - We did this with the work queue in the Mandelbrot fractal generator

Shared Data

- Define, implement, *and document* a synchronization protocol
  - One-at-a-time execution
  - Non-interfering operations
  - Readers / writers
  - Reduced critical section size
  - Nested locks
  - Application-specific semantic relaxation

Increasing
Complexity

Shared Data

- Memory synchronization
  - Compiler handles *most* of this
  - Use `volatile` keyword
- Task scheduling
  - Synchronization can affect scheduling
  - Consider ways to schedule tasks to minimize waiting

Shared Queue

- “Thread-safe” queue with additional design considerations:
  - In what order are items removed from the queue?
    - FIFO? LIFO? Priority order? Other?
  - Should the queue size be fixed or grow?
  - What happens when an element is removed from an empty queue?
    - Related question: What happens when an element is added to a full queue?
  - How critical is the performance of the queue?
    - Related question: What is the level of contention on the queue?

Shared Queue

- Start with the simplest implementation that will work, and work from there
- Many parallel programming environments have built-in shared queue primitives

Distributed Array

- Parallel programs often operate on *massive* data sets
  - Adding more processors often allows larger data sets rather than decreased processing time
  - Data may be so large that it won't fit into main memory
    - Even if it fits in memory, it certainly won't fit in the cache
    - ...even the 12MB L2 cache on some modern processors

Distributed Array

- Common array distributions:
  - 1D block – Array is partitioned into 1D sub-arrays, and each partition is distributed to a UE
    - This is a 1-to-1 block-to-UE mapping
  - 2D block – Array is partitioned into 2D sub-arrays, and each partition is distributed to a UE
    - This is also a 1-to-1 block-to-UE mapping
  - Block-cyclic – Array is partitioned into either 1D or 2D blocks and block are distributed round-robin to UEs
    - This is a many-to-1 block-to-UE mapping

Distributed Array

- Mapping array indexes
  - Original problem is formulated in terms of *global* indexes
  - Each UE “wants” to operate in terms of *local* indexes
- Solution?

Distributed Array

- Mapping array indexes
  - Original problem is formulated in terms of *global* indexes
  - Each UE “wants” to operate in terms of *local* indexes
- Solution?
  - Create an ADT to map local indexes to global indexes

Distributed Array

- Locality of reference
  - Accessing data “hot” in the cache is fastest
  - Accessing data on the local NUMA node is fastest
- Choose the partition wisely
  - Partition data to maximize cache usage
  - Partition data to fit on a single NUMA node
  - etc.

References

Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders; "Some Algorithm Structure and Support Patterns for Parallel Application Programs": *Proceedings of the Ninth Pattern Languages of Programs Workshop (PLoP 2002)*, 2002; http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

Next week...

- ...and by "next week" I mean ***this Friday*** (8/22)
- Atomic Operations
- Lockless Algorithms

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.