VGP393 – Week 3

- Agenda:
  - Finding Concurrency
    - Program decompositions
    - Dependency analysis
    - Design evaluation
  - Quiz #1
  - Assignment #1 due
  - Assignment #2 starts

30-July-2008

© Copyright Ian D. Romanick 2008

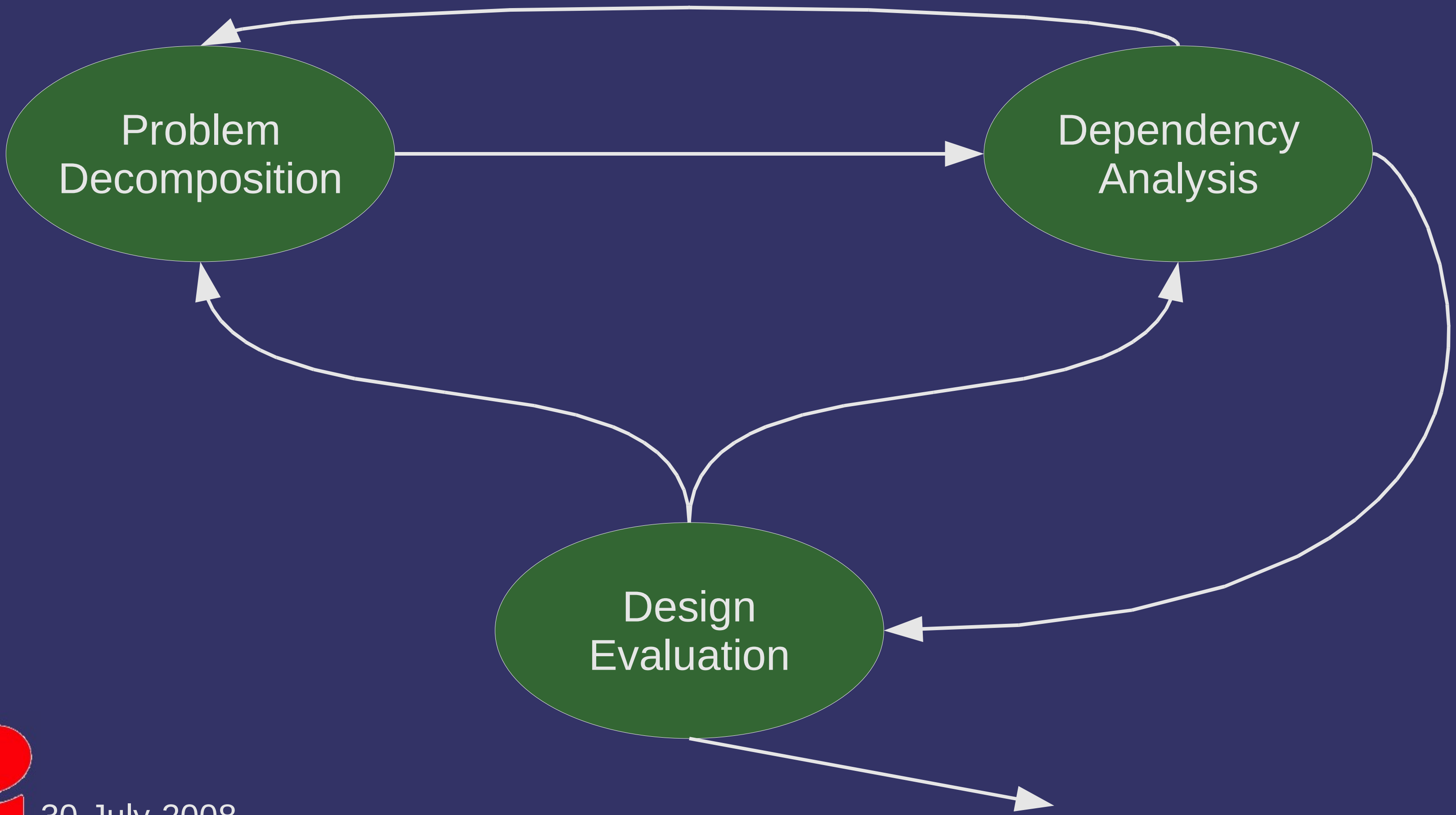
Finding Concurrency

- Parallel programming is about *finding* and *exploiting* concurrency

© Copyright Ian D. Romanick 2008

Finding Concurrency

- Parallel programming is about *finding* and *exploiting* concurrency


Problem Decomposition

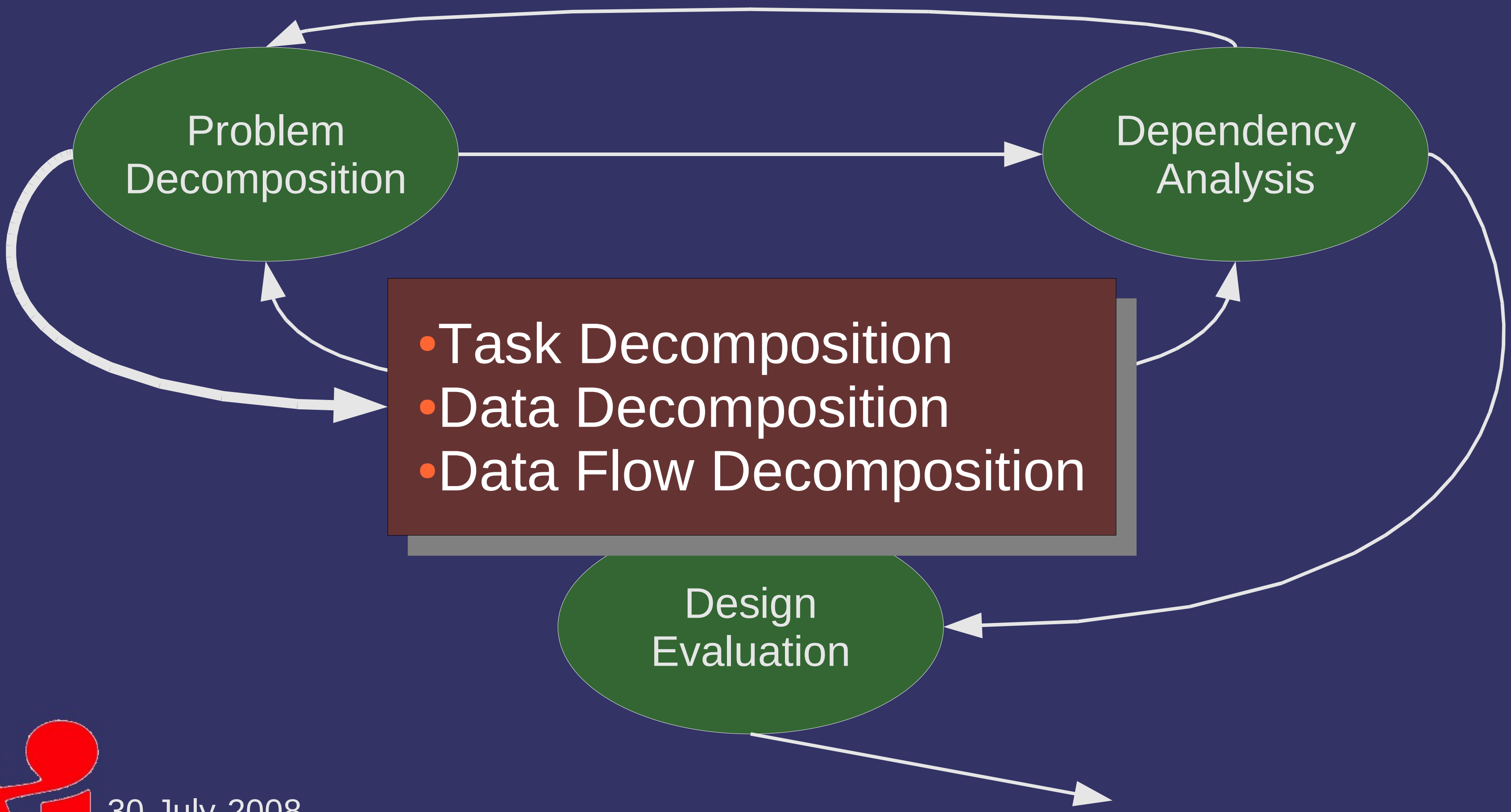
Dependency Analysis

Design Evaluation

© Copyright Ian D. Romanick 2008

Finding Concurrency

- Parallel programming is about *finding* and *exploiting* concurrency



Problem Decomposition

Dependency Analysis

- Task Decomposition
- Data Decomposition
- Data Flow Decomposition

Design Evaluation

30-July-2008

© Copyright Ian D. Romanick 2008

Problem Decompositions

- Must decompose the problem into elements that can execute in parallel
- Decomposition occurs along two primary axes and one secondary axis
  - *Task decomposition* views the problem as a sequence of *tasks* that can be executed concurrently
  - *Data decomposition* views the data problem as separate chunks that can be evaluated concurrently
  - *Data flow decomposition* looks at how data flows through the program as the problem is solved

© Copyright Ian D. Romanick 2008

Driving Forces

- Three forces drive all decompositions:
  - Flexibility – Is the design flexible enough to be adapted to changes in requirements?
    - Usually changes in problem size or changes in target system
  - Efficiency – Does the design scale to *at least* the number of processors in the target system?
    - Efficiency for one target system may come at the cost of flexibility to other systems
  - Simplicity – Can the program design be understood, debugged, and maintained?
    - Simplicity can come at a cost to efficiency

© Copyright Ian D. Romanick 2008

Task Decomposition

- Look at the problem as a collection of tasks
  - Look at the individual steps required to solve the problem
  - Determine whether or not these steps are independent
- Find as many tasks as possible
  - Individual function calls
  - Iterations of a loop
  - Updates to portions of large data structures

© Copyright Ian D. Romanick 2008

Task Decomposition

- Evaluate the design...
  - Flexibility – Be flexible in the number of tasks
    - Parametrize the number and size of tasks at run-time
  - Efficiency – Two possibly opposing goals:
    - Tasks should be large enough to outweigh management overhead
    - Should be enough tasks to keep all PEs busy all the time
  - Simplicity – Tasks should be defined in such a way that debugging and maintenance are easy
    - Re-use code from sequential version of program

© Copyright Ian D. Romanick 2008

Data Decomposition

- Works well if...
  - Problem focuses on the manipulation of a large data structure
  - The same or similar operations are performed on different parts of the structure in independent ways
- Focus on data structures that can be broken into chunks that can be operated on concurrently
  - Concurrency can be found in *array-based computations* by looking at updates to different segments of the array
  - Concurrent updates on *recursive data structures* can be performed on different subtrees, etc.

© Copyright Ian D. Romanick 2008

Data Decomposition

- Evaluate the design...
  - Flexibility – Be flexible in the size and number of data chunks
    - *Granularity knobs* are parameters in the program that, at run-time, control the size and number of data chunks
    - Granularity has a major impact on the overhead required to manage dependencies between the chunks
    - Dependencies should scale at a slower rate than effort required to compute each chunk

© Copyright Ian D. Romanick 2008

Data Decomposition

- Evaluate the design...
  - Efficiency – An efficient design should evenly map work to UEs and not create too much additional management work
    - Size of data chunks must be large enough to dominate the amount of work required to manage the dependencies
    - Mapping chunks to UEs must also be considered. If the mapping is poor, some PEs will have much more work to do than others
    - Cache and memory access (NUMA) issues are important for data that must be shared

© Copyright Ian D. Romanick 2008

Data Decomposition

- Evaluate the design...
  - Simplicity – Complex data mappings are difficult to debug
    - Abstract data types to control the mapping of global data to task-local data are useful

© Copyright Ian D. Romanick 2008

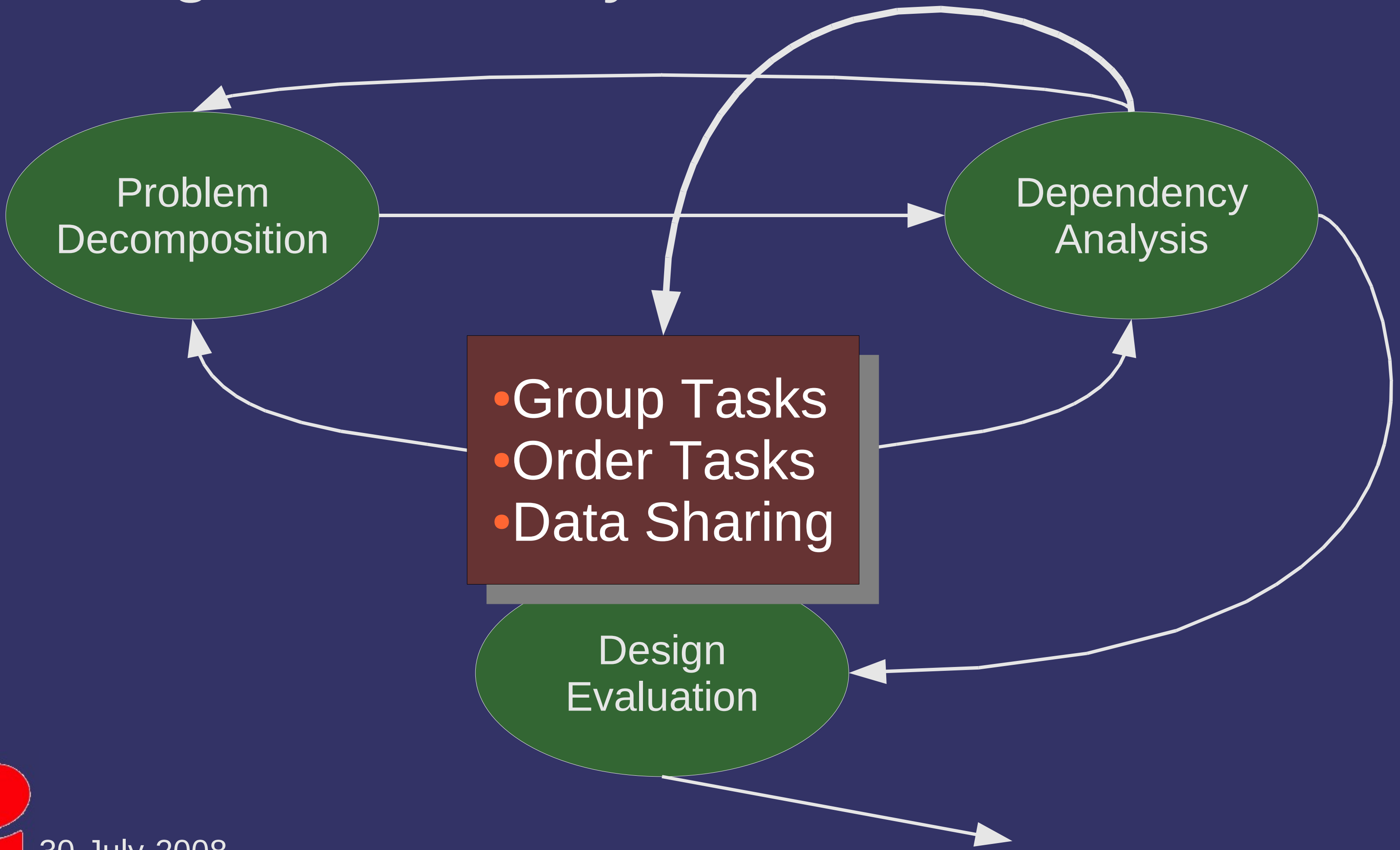
Data Flow Decomposition

- Look at how data flows from one task to another
  - Hybrid of task decomposition and data decomposition
- Key feature is that one task cannot begin until it receives data from another task
  - *Producer-consumer* problems are the classic example
  - Understand the nature of the dependency between tasks
    - Seek to minimize the delay caused by the dependency

© Copyright Ian D. Romanick 2008

Finding Concurrency

- Parallel programming is about *finding* and *exploiting* concurrency



Problem Decomposition

Dependency Analysis

- Group Tasks
- Order Tasks
- Data Sharing

Design Evaluation

© Copyright Ian D. Romanick 2008

Dependency Analysis

- Sometimes task decomposition generates sets of tasks that are entirely independent
    - These problems are often called *embarrassingly parallel*
- *Dependencies* are cases where the execution of one task affects the execution of another
    - *Data-sharing dependencies* can occur when tasks must share or exchange data during execution
    - *Ordering constraints* occur when tasks must execute in a certain order

© Copyright Ian D. Romanick 2008

Dependency Analysis

- Several common ordering constraints:
  - Sequential (or data flow) dependency – One task needs data generated by another task
  - Parallel dependency – A group of tasks *must* execute at the same time
  - Independence – Tasks are truly independent and can execute in any order

© Copyright Ian D. Romanick 2008

Group Tasks

- Grouping tasks simplifies dependency analysis
  - Dependencies between groups can be resolved once per group instead of once per task in each group
  - This principle guides the grouping...pick groupings that simplify the dependency analysis

© Copyright Ian D. Romanick 2008

Group Tasks

- Look at the original problem decomposition
  - High-level operations or loops are central to most task decompositions
    - Tasks that correspond to high-level operations usually group together
    - Tasks within a high-level operation that share a constrain should remain as a separate group
- Merge groups that share a common constraint
  - Larger groups make scheduling and load balancing easier

© Copyright Ian D. Romanick 2008

Group Tasks

- Look at constraints between groups
  - If groups have a clear ordering or a clear data flow, this is easy
  - However, independent task groups may share constraints
    - It may be better to merge these groups

© Copyright Ian D. Romanick 2008

Order Tasks

- How must task groups be ordered to satisfy all constraints?
- Create a partial ordering of tasks by identifying ordering constraints among groups
  - Ordering must be restrictive enough to satisfy all constraints
    - Design is not correct otherwise!
  - No more restrictive than necessary
    - Additional constraints limit flexibility in load balancing

© Copyright Ian D. Romanick 2008

Order Tasks

- For each group, identify data required before that group can execute
  - To identify the ordering constraint, find the task / group that creates that data
- Determine if external services impose addition ordering constraints
  - Classic example is file I/O...different tasks may have to write data to a file in a particular order
- Also note when there is *no* constraint
  - Makes it more clear that potential interactions have been examined

© Copyright Ian D. Romanick 2008

Data Sharing

- Determine *how* and *when* data is shared among tasks
  - Data that is statically partitioned to particular tasks is task-local
  - Data that cannot be strictly associated with a particular task is shared
    - This is the source of most dependencies
  - Task may also need access to a *portion* of another task's data
    - Usually boundary data that neighbors that tasks local data

© Copyright Ian D. Romanick 2008

Data Sharing

- Impacts both correctness and efficiency of the program
  - Incorrect sharing can lead to some tasks getting incorrect data (reading before data is written)
  - Synchronization on global data can incur a lot of overhead
  - Excessive communication can also incur a lot of overhead
    - Condition variables, message queues, etc.

© Copyright Ian D. Romanick 2008

Data Sharing

- Identify data that is shared among tasks
  - Look back at the original program decomposition for clues
- Classify each shared data
  - Read-only – Data the is not modified does not need to be protected
  - Effectively-local – Global data that is partitioned into per-UE subsets needs limited, if any, protection
  - Read/write – Data that is both read and written arbitrarily needs the most synchronization

© Copyright Ian D. Romanick 2008

Data Sharing

- Special cases of read / write data:
  - Accumulate – Partial results are accumulated together to form a final result
    - Typically each task has a copy of the data where it accumulates partial results
    - When all tasks are complete, each local copy is accumulated into the final result
  - Multiple-read / single-write – Data is read by multiple tasks, but only updated by one
    - All readers need the initial value
    - The writer can modify the data arbitrarilly
    - Two copies of the data are required (constant initial and modifyable)

© Copyright Ian D. Romanick 2008

Design Evaluation

- Evaluate the design *so far*
  - Decide whether or not to return to earlier steps or move on to the next step
  - The earlier design flaws are caught, the easier they are to fix!

© Copyright Ian D. Romanick 2008

Design Evaluation

- Suitability for target platform
  - Does the design match the number of PEs available?
  - How is data shared among the PEs?
    - Different data partitionings fit SMP, NUMA, etc.
  - Are there sufficient UEs to mask I/O latency, etc?
  - Ratio of time spent doing useful work vs. overhead
    - Synchronization primitives (and available atomic operations) vary from platform to platform

© Copyright Ian D. Romanick 2008

Design Evaluation

- Flexibility
  - Flexible in the number of tasks generated?
  - Is the definition of tasks independent of scheduling?
  - Is the size of data chunks parameterizable?
  - Does the algorithm handle boundary cases?

© Copyright Ian D. Romanick 2008

Design Evaluation

- Efficiency
  - Can the load be balanced amont PEs?
  - Is overhead minimized?
    - Thread creation?
    - Synchronization?
    - Message passing?
    - etc.

© Copyright Ian D. Romanick 2008

Design Evaluation

- Simplicity
  - Is the design as simple as possible without missing necessary components?

© Copyright Ian D. Romanick 2008

Design Evaluation

- Other issues:
  - How regular are tasks and their dependencies?
  - Are interactions between tasks synchronous or asynchronous?
  - Are tasks grouped in the best way?

© Copyright Ian D. Romanick 2008

References

Much of this lecture comes from the following two sources:

Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders; "Patterns for Finding Concurrency for Parallel Application Programs"; *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP 2000)*, 2000; http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/proceedings.html

Beverly Sanders, “A Pattern Language for Parallel Programming”; http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt

See also http://www.cise.ufl.edu/research/ParallelPatterns/

© Copyright Ian D. Romanick 2008

Next week...

- Algorithm structure
  - Task Parallelism
  - Divide and Conquer
  - etc.
- Supporting Strcutures
  - SPMD
  - Master / worker
  - Loop Parallelism
  - etc.

© Copyright Ian D. Romanick 2008

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.

© Copyright Ian D. Romanick 2008