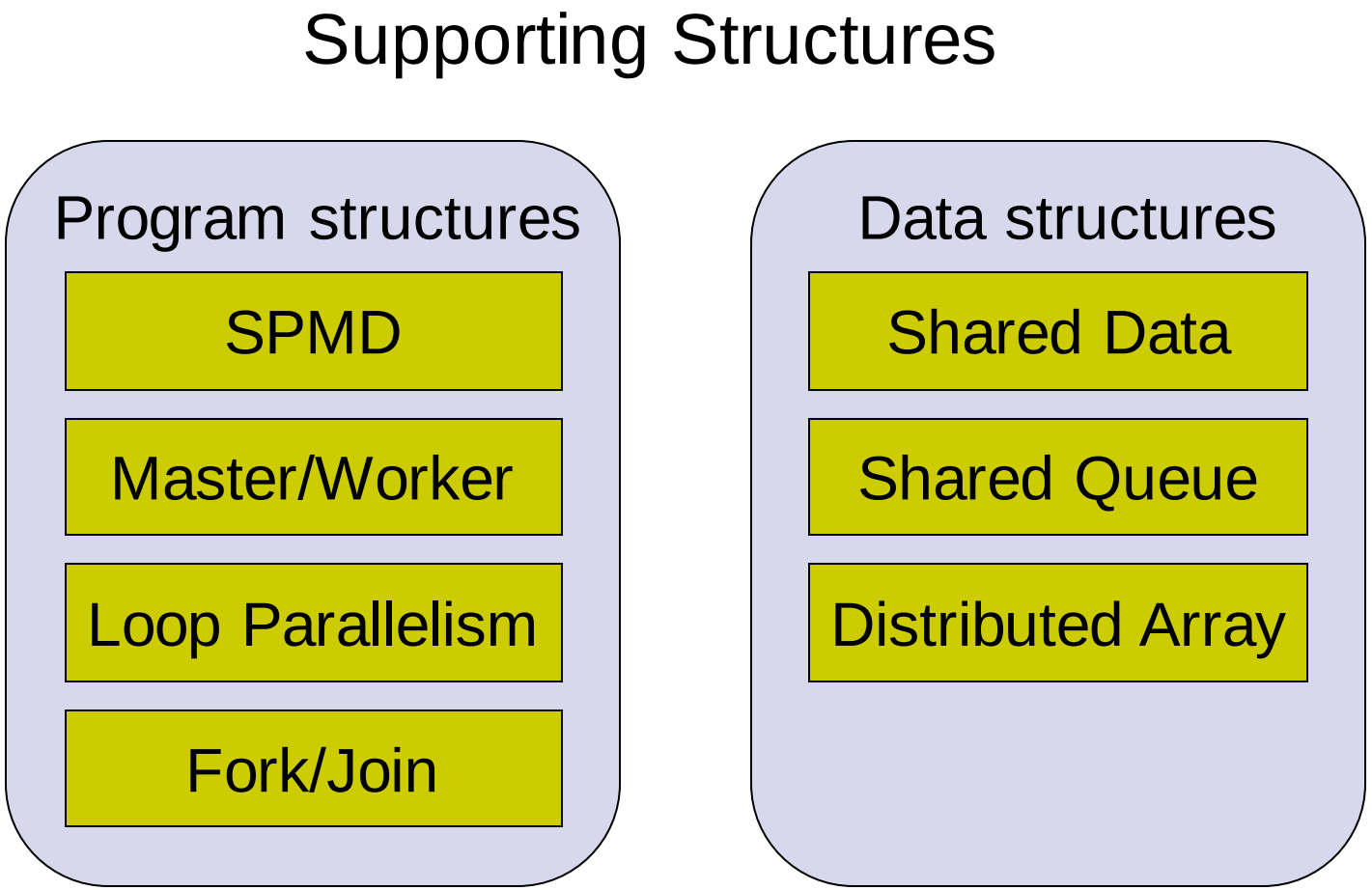
Supporting structures

Jakub Yaghob

The supporting structures design space

- The finding concurrency and algorithm structure design spaces focus on algorithm expression
- At some point, algorithms must be translated into programs
- An intermediate stage between the problem-oriented patterns and the specific programming mechanisms
- How to structure source code to best support algorithm structures of interest?

Overview


Supporting Structures
Program structures
SPMD
Master/Worker
Loop Parallelism
Fork/Join
Data structures
Shared Data
Shared Queue
Distributed Array

The supporting structures design space

- Groups represent
  - Program-structuring approaches
  - Commonly used shared data structures
- In some programming environments some of the patterns are very well supported
- Understanding the low-level details behind these structures is important for effectively using them
- Describing these structures as patterns provides guidance for implementing them from scratch
- Combine patterns in different ways to meet the needs of a particular problem

Forces

- Clarity of abstraction
  - Is the parallel algorithm clearly apparent from the source code?
    - In a well-structured program, the algorithm leaps from the page
- Scalability
  - How many CPUs can the parallel program effectively utilize?
    - The scalability is restricted by three factors
      - The amount of concurrency available in the algorithm
      - Amdahl's law
      - The parallel overhead contributes to the serial fraction

Forces

- Efficiency
  - How close does the program come to fully utilizing the resources of the parallel computer?

$$E(P) = \frac{S(P)}{P} = \frac{T_{total}(1)}{P\,T_{total}(P)}$$

    - T(1) should be the time of the best sequential algorithm, but it is common to use the runtime for the parallel program on a single PE
      - It inflates the efficiency
- Maintainability
  - Is the program easy to debug, verify and modify?

Forces

- Environmental affinity
  - Is the program well aligned with the programming environment and hardware of choice?
- Sequential equivalence
  - Where appropriate, does a program produce equivalent results when run with many UEs as with one? If not equivalent, is the relationship between them clear?
    - Floating-point operations performed in a different order can produce small (sometimes not) changes in the resulting values
    - Sequential equivalence is a highly desirable goal
      - Debugging on the single-processor version

Choosing the patterns

- In most cases, the programming environment selected for the project and the patterns used from the Algorithm Structure design space point to the appropriate program structure pattern to use

	Task parallelism	Divide and Conquer	Geometric Decom-position	Recursive Data	Pipeline	Event-Based Coordi-nation
SPMD	★★★★	★★★	★★★★	★★	★★★	★★
Loop Parallelism	★★★★	★★	★★★			
Master/ Worker	★★★★	★★	★	★	★	★
Fork/Join	★★	★★★★	★★		★★★★	★★★★

Choosing the patterns

	PThread	TBB	OpenMP	MPI
SPMD		★	★★★	★★★★
Loop Parallelism	★	★★★★	★★★★	★
Master/Worker	★★	★★	★★	★★★
Fork/Join	★★★★	★★★★	★★★	

The SPMD pattern

- Problem
  - The interactions between the various UEs cause most of the problems when writing correct and efficient parallel programs. How can programmers structure their parallel programs to make these interactions more manageable and easier to integrate with the core computations?

The SPMD pattern

- Context
  - For most parallel algorithms, the operations carried out on each UE are similar
  - The tasks and their interactions can be made more manageable by bringing them all together into one source tree
    - The logic for the tasks is side by side with the logic for the interactions between tasks
    - Only one program to manage
    - Important on systems with large number of PEs

The SPMD pattern

- Forces
  - Using similar code for each UE is easier for the programmer
    - Most complex applications require different operations on different UEs with different data
  - SW outlives any given parallel computer
    - Portable programs
  - High scalability and good efficiency
    - The program well aligned with the architecture of the parallel computer

The SPMD pattern

- Solution
  - Single source-code image runs on each of the UEs
  - Basic elements
    - Initialize
      - Establishing a common context
    - Obtain a unique identifier
      - Different decisions during execution
    - Run the same program on each UE, using the unique ID to differentiate behavior on different UEs
      - Branching statements to give specific blocks of code to different UEs
      - Using the ID in loop index calculations to split loop iterations

The SPMD pattern

- Solution – cont.
  - Basic elements – cont.
    - Distribute data
      - Data operated on by each UE is specialized
      - Decomposing global data into chunks and storing them in the local memory, later, if required, recombining them into the global result
      - Sharing or replicating the program's major data structures and using the ID to associate subsets of the data with particular UEs
    - Finalize
      - Cleaning up the shared context

The SPMD pattern

- Discussion
  - Clarity of abstraction
    - From awful to good
    - Complex index algebra
  - SPMD algorithms highly scalable
    - Extremely complicated, complex load-balancing logic
    - Little resemblance to their serial counterparts – common criticism of the SPMD
  - Overhead associated with startup and termination are segregated at the beginning and end of the program
    - An important advantage

The SPMD pattern

- Discussion – cont.
  - SPMD programs closely aligned with programming environments based on message passing
    - SPMD does not assume anything concerning the address space
    - As long as each UE can run its own instruction stream operating on its own data, the SPMD structure is satisfied
      - This generality is one of the strengths of this pattern

The Master/Worker pattern

- Problem
  - How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks among the UEs?

The Master/Worker pattern

- Context
  - Balancing the load is so difficult that it dominates the design
  - Characteristics
    - The workloads associated with the tasks are highly variable and unpredictable
      - Predictable workloads can be sorted into equal-cost bins, statically assigned to UEs, and parallelized using the SPMD or Loop Parallelism
      - Static distributions for unpredictable workloads tend to produce suboptimal load ballance

The Master/Worker pattern

- Context – cont.
  - Characteristics – cont.
    - The program structure for the computationally intensive portions of the problem doesn't map onto simple loops
      - If the algorithm is loop-based, a statistically near-optimal workload achieved by a cyclic distribution or by using a dynamic schedule on the loop
      - If the control structure in the program is more complex than a simple loop, then use the Master/Worker
    - The capabilities of the PEs vary across the system, change over the course of computation, or are unpredictable

The Master/Worker pattern

- Context – cont.
  - When tasks are tightly coupled (they communicate or share RW data) and they must be active at the same time, the Master/Worker is not applicable
  - Particularly relevant for problems using the Task Parallelism pattern when there are no dependencies among tasks (embarrassingly parallel problems)
  - Using with Fork/Join pattern for the cases where the mapping of tasks onto UEs is indirect

The Master/Worker pattern

- Forces
  - The work for each task, and in some cases even the capabilities of the PEs, varies unpredictably
    - The design must balance the load without predicitions
  - Operations to balance the load impose communication overhead and can be very expensive
    - A smaller number of large tasks
      - It reduces the number of ways tasks can be partitioned among PEs
  - Logic to produce an optimal load can be convoluted and require error-prone changes to a program
    - Trade-offs between an optimal distribution of the load and code that is easy to maintain

The Master/Worker pattern

- Solution
  - Two logical elements
    - A master and one or more instances of a worker
  - The master initiates the computation and sets up the problem
  - It then creates a bag of tasks
  - In the classic algorithm, the master then waits until the job is done, consumes the result, and then shuts down the computation

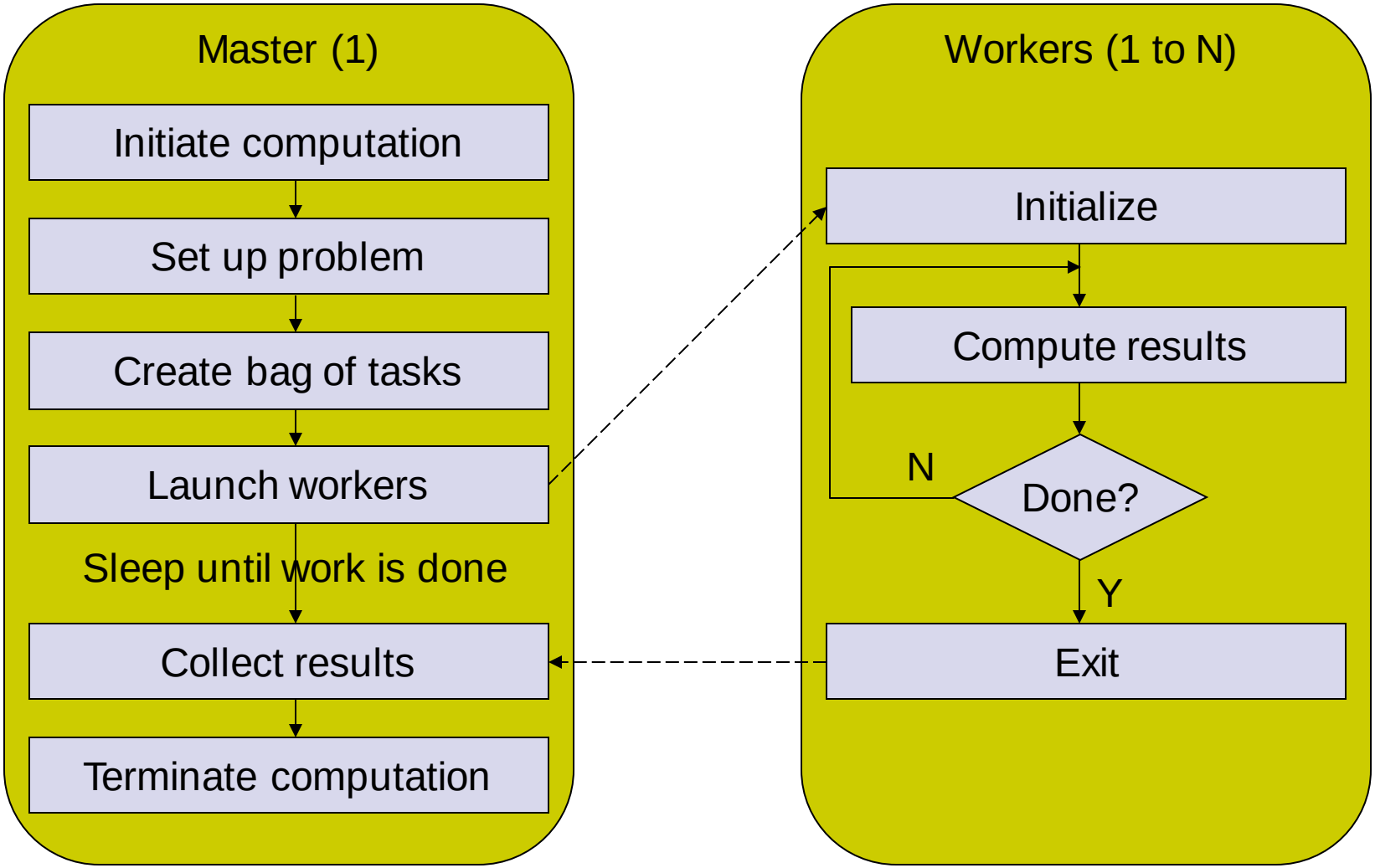
The Master/Worker pattern

- Solution – cont.
  - The bag of tasks can be implemented by the Shared Queue pattern
    - Other implementations are possible: distributed queue, monotonic counter
  - Worker
    - Enters the loop
    - At the top of the loop, the worker takes a task from the bag of tasks
    - Does the indicated work
    - Tests for completion
    - Does it until the termination condition is met

The Master/Worker pattern

- Solution – cont.
  - The master wakes up after the termination condition, collects the result and finishes the computation
  - Master/Worker automatically balance the load
    - The programmer does not explicitly decide which task is assigned to which UE
    - This decision is made dynamically by the master as a worker completes one task and accesses the bag of tasks for more work

The Master/Worker pattern



The Master/Worker pattern

- Discussion
  - Good scalability
    - The number of tasks greatly exceeds the number of workers and the costs of the individual tasks are not so variable that some workers take drastically longer then the others
  - Management of the bag of tasks can require global communication
    - The overhead can limit efficiency
    - Not a problem when the work associated with each task is much greater then the time required for management
  - Not tied to any particular HW environment

The Master/Worker pattern

- Detecting completion
  - How to correctly determine when the entire problem is complete?
    - Must be done in an efficient way but also it must guarantee that all of the work is complete before workers shut down
    - All tasks are placed into the bag before the workers begin. Then each task continues until the bag is empty, at which point the workers terminate
    - When termination condition is detected, a poison pill is placed into the bag
      - Taken in the next round by all workers

The Master/Worker pattern

- Detecting completion – cont.
  - The set of tasks is not initially known
    - Workers can add tasks
    - The empty bag is not the termination condition
    - The bag is empty and all workers have finished – correct termination condition
    - Moreover, on systems with asynchronous messaging no messages in transit

The Master/Worker pattern

- Variations
  - The master may turn into a worker after it has created the tasks
    - The termination condition can be detected without explicit action by the master
  - The concurrent tasks map onto a simple loop
    - The master can be implicit
    - The pattern can be implemented as a loop with dynamic iteration assignment

The Master/Worker pattern

- Variations – cont.
  - A centralized task queue can become a bottleneck
    - Especially in a distributed memory environment
    - Random work stealing
      - Each PE maintains a separate double-ended task queue
      - New tasks placed in the front of the local task queue
      - When the task is completed, it is removed from the queue
      - If the local queue is empty, another random PE is chosen, and the task from the end of its queue is “stolen”
      - If that queue is empty as well, try another PE

The Master/Worker pattern

- Variations – cont.
  - The pattern can be modified to provide a modest level of fault tolerance
    - The master maintains 2 queues
      - One for tasks that need to be assigned to workers
      - Another one for tasks that have already been assigned but not completed
    - After the first queue is empty, master can redundantly assign tasks from the “not completed” queue

The Loop Parallelism pattern

- Problem
  - Given a serial program whose runtime is dominated by a set of computationally intensive loops, how can it be translated into a parallel program?

The Loop Parallelism pattern

- Context
  - The overwhelming majority of programs used in scientific and engineering applications are expressed in terms of iterative constructs
    - Well-accepted programs already exist
  - Concurrent tasks identified as iterations of parallelized loops
  - Evolve a sequential program into a parallel program by a series of transformations on the loops

The Loop Parallelism pattern

- Context – cont.
  - Semantically neutral transformation
    - All changes are localized to the loops with transformations that remove loop-carried dependencies and leave the overall program semantics unchanged
  - Not for all problems
    - It only works when the algorithm structure has most, if not all, of the computationally intensive work in a manageable number of distinct loops
    - The body of the loop must result in loop iterations that work well as parallel tasks
      - Computationally intensive, sufficient concurrency, mostly independent

The Loop Parallelism pattern

- Context – cont.
  - Not for all target platforms
    - Needs some level of support for a shared address space
      - Sometimes possible restructuring to effective distributed data structures
    - Often only effective for systems with smaller number of PEs
      - Amdahl's law
  - OpenMP created to support this pattern
    - Using the Task Parallelism and Geometric Decomposition patterns

The Loop Parallelism pattern

- Forces
  - Sequential equivalence
  - Incremental parallelism (or refactoring)
    - The parallelization is introduced as a sequence of incremental transformations, one loop at a time
    - The transformations don't break the program
  - Memory utilization
    - The data access patterns implied by the loops mesh well with the memory hierarchy
      - Sometimes needs to massively restructure loops

The Loop Parallelism pattern

- Solution
  - Find the bottlenecks
    - Most computationally intensive loops
      - Inspecting the source code
      - Use a program performance analysis tools
    - The runtime spend in these loops limits the scalability
  - Eliminate loop-carried dependencies
    - The loop iterations must be nearly independent
    - Transform the code to remove dependencies between iterations or RW accesses
      - Use Task Parallelism and Shared Data patterns

The Loop Parallelism pattern

- Solution – cont.
  - Parallelize the loops
    - Split up the iterations among the UEs
    - Use semantically neutral directives
      - OpenMP
    - One loop at a time with testing and careful inspection for introduced race conditions
  - Optimize the loop schedule
    - Load balancing

The Loop Parallelism pattern

- Solution – cont.
  - Previous approach effective only when the computing times for the loop iterations large enough to compensate for parallel loop overhead
  - Another problem is the number of iterations per loop
    - Many iterations per UE provides greater scheduling flexibility

The Loop Parallelism pattern

- Solution – cont.
  - Transformations addressing these issues
  - Merge loops
    - Sequence of loops with consistent loop limits
    - Merge into a single loop
  - Coalesce nested loops
    - Combine nested loops into a single loop with a larger combined iteration count
    - Overcomes parallel loop overhead
      - Create more concurrency to better utilize larger number of UEs
      - Provide additional options for scheduling the iterations

The Loop Parallelism pattern

- Solution – cont.
  - It is difficult to define sequentially equivalent programs when the code uses either a thread ID or the number of threads
    - These algorithms favor particular numbers of threads

The Loop Parallelism pattern

- Performance considerations
  - The pattern assumes that multiple PEs share a single address space with equal-time access to every element of memory
    - NUMA
  - False sharing
    - Variables not shared between UEs, but they are on the same cache line

The Fork/Join pattern

- Problem
  - In some programs, the number of concurrent tasks varies as the program executes, and the way these tasks are related prevents the use of simple control structures such as parallel loops. How can a parallel program be constructed around such complicated sets of dynamic tasks?

The Fork/Join pattern

- Context
  - General and dynamic parallel control structure
    - Tasks created dynamically – forked
    - Later terminated – joined with the forking task
  - When relationship between tasks simple, task creation handled with
    - Loop parallelism
    - Task queue (Master/Worker)
  - Relationship between the tasks within the algorithm must be captured in the way the tasks are managed
    - Recursively generated task structures
    - Highly irregular sets of connected tasks
    - Different functions mapped onto different concurrent tasks

The Fork/Join pattern

- Context – cont.
  - Particularly relevant for
    - Java programs running on shared memory
    - Divide and conquer
    - Recursive data pattern
    - OpenMP with nested parallel regions

The Fork/Join pattern

- Forces
  - Algorithms imply relationships between tasks
    - Complex and recursive relations between tasks
    - Relations created and terminated dynamically
  - A one-to-one mapping tasks onto UEs natural
    - Balanced against the number of UEs a system can handle
  - UE creation and destruction are costly operations
    - The algorithm might need to be recast to decrease these operations

The Fork/Join pattern

- Solution
  - Mapping tasks onto UEs
  - Direct task/UE mapping
    - New subtasks forked – new UEs created
    - A synchronization point where the main task waits for its subtasks to finish
    - After a subtask terminates, the UE handling it will be destroyed
    - Standard programming model in OpenMP

The Fork/Join pattern

- Solution – cont.
  - Indirect task/UE mapping
    - Repeated creation and destruction of UEs
    - Many more UEs then PEs
    - Thread pool
      - A static set of UEs before the first fork
      - The number of UEs the same as the number of PEs
      - Mapping of tasks to UEs occurs dynamically using a task queue
      - Efficient programs with good load balancing

The Shared Data pattern

- Problem
  - How does one explicitly manage shared data inside a set of concurrent tasks?

The Shared Data pattern

- Context
  - The common elements
    - At least one data structure accessed by multiple tasks
    - At least one task modifies the shared data structure
    - The tasks potentially need to use the modified value during the concurrent computation

The Shared Data pattern

- Forces
  - The results of the computation must be correct for any ordering of the tasks that could occur during the computation
  - Explicitly managing shared data can incur parallel overhead, which must be kept small if the program is to run efficiently
  - Techniques for managing shared data can limit the number of tasks that can run concurrently, thereby reducing the potential scalability of an algorithm
  - If the constructs used to manage shared data are not easy to understand, the program will be more difficult to maintain

The Shared Data pattern

- Solution
  - One of the more error-prone aspects of designing a parallel algorithm
  - Start with a solution that emphasizes simplicity and clarity of abstraction, then try more complex solutions if necessary to obtain acceptable performance
  - The solution reflects this approach

The Shared Data pattern

- Solution – cont.
  - Be sure this pattern is needed
    - It might be worthwhile to revisit decisions made earlier in the design process
      - The decomposition into tasks
    - Different decisions might lead to a solution that fits one of the Algorithm Structure patterns without the need to explicitly manage shared data

The Shared Data pattern

- Solution – cont.
  - Define an abstract data type
    - View the shared data as an abstract data type (ADT) with a fixed set of (possibly complex) operations on the data
    - Each task typically perform a sequence of these operations
    - These operations should have the property that if they are executed serially, each operation will leave the data in a consistent state
    - The implementation of the individual operations will most likely involve a sequence of lower-level actions, the result of which should not be visible to other UEs

The Shared Data pattern

- Solution – cont.
  - Implement an appropriate concurrency-control protocol
    - One-at-a-time execution
      - Operations are executed serially
      - In a shared-memory environment, each operation as part of a single critical section
        - Implemented by the facilities of the target programming environment
      - In a message-passing environment, assign the shared data structure to a particular UE
      - Easy to implement
      - Overly conservative, can produce a bottleneck

The Shared Data pattern

- Solution – cont.
  - Concurrency-control protocol – cont.
    - Noninterfering sets of operations
      - Analyze the interference between the operations
      - Operation A interferes with operation B if A writes a variable that B reads
        - An operation may interfere with itself – more than one task executes the same operation
      - Sometimes the operations fall into two disjoint sets, where the operations in different sets do not interfere with each other
        - Different critical sections

The Shared Data pattern

- Solution – cont.
  - Concurrency-control protocol – cont.
    - Readers/writers
      - The type of interference – some of the operations modify the data, but others only read it
      - More readers can execute the operation concurrently
      - Greater overhead than that of simple mutex locks
        - The length of the read operation should be long enough
        - Larger number of readers than writers

The Shared Data pattern

- Solution – cont.
  - Concurrency-control protocol – cont.
    - Reducing the size of the critical section
      - Analyzing the implementations of the operations in more detail
      - Only part of the operation involves actions that interfere with other operations
      - The size of the critical section can be reduced to that smaller part
      - Very easy to get wrong
        - Only if it gives significant performance improvement
        - The programmer completely understands it

The Shared Data pattern

- Solution – cont.
  - Concurrency-control protocol – cont.
    - Nested locks
      - Two operations are almost noninterfering
      - Use two locks
      - Example
        - Operation A does some work with x, then updates y
        - Operation B does some work with y
        - LockA for x, LockB for y
        - B holds LockB
        - A holds LockA, then for the update of y holds LockB
      - Deadlocks

The Shared Data pattern

- Solution – cont.
  - Concurrency-control protocol – cont.
    - Application-specific semantic relaxation
      - Partially replicate shared data
      - Even allowing the copies to be inconsistent
        - Without affecting the results

The Shared Data pattern

- Solution – cont.
  - Review other considerations
    - Memory synchronization
      - Caching and compiler optimization can result in unexpected behavior with respect to the shared variables
      - Memory synchronization techniques very platform-specific
      - OpenMP – flush
      - `volatile` in C/C++ should be synchronized
    - Task scheduling
      - How the shared variables affect task scheduling?
      - Take into account that tasks might be suspended waiting for access to shared data
        - Assign multiple tasks to each UE in the hope there will be at least one task per UE that is not waiting

The Shared Queue pattern

- Problem
  - How can concurrently-executing UEs safely share a queue data structure?

The Shared Queue pattern

- Context
  - Effective implementation of many parallel algorithms requires a queue shared among UEs.
  - Master/Worker pattern
  - Pipeline pattern

The Shared Queue pattern

- Forces
  - Simple concurrency-protocol provide greater clarity of abstraction
  - Concurrency-control protocols that encompass too much of the shared queue in a single synchronization construct increase the chances UEs will remain blocked waiting to access the queue
    - Limits available concurrency
  - A concurrency-control protocol finely tuned to the queue increases the available concurrency
    - At the cost of much more complicated, and more error-prone, synchronization constructs
  - Maintaining a single queue for systems with complicated memory hierarchies (NUMA) can cause excess communication and increase parallel overhead
    - Break the single-queue abstraction, use multiple or distributed queues

The Shared Queue pattern

- Solution
  - Ideally the shared queue implemented as part of the target programming environment
    - Explicitly as an ADT
    - Implicitly as support for the higher-level patterns
  - Implementing the shared queue can be tricky
    - Appropriate synchronization
    - Performance considerations
      - Large number of UEs accessing the queue
  - Sometimes noncentralized queue might be needed to eliminate performance bottlenecks

The Shared Queue pattern

- Solution – cont.
  - The abstract data type (ADT)
    - The values are ordered lists of 0 or more objects of some type
    - Operations put and take
    - What happens when a take is attempted on an empty queue?

The Shared Queue pattern

- Solution – cont.
  - Queue with “one at a time” execution
    - Nonblocking queue
      - Take operation returns with an indication
      - Straightforward implementation with mutual exclusion
    - Block-on-empty queue
      - Take operation is blocking
      - Waiting thread must release its lock
      - Reacquire the lock before trying it again

The Shared Queue pattern

- Solution – cont.
  - Concurrency-control protocols for noninterfering operations
    - If the performance is inadequate, look for more efficient concurrency-control protocols
    - Look for noninterfering sets of operations in ADT
  - Concurrency-control protocols using nested locks
    - Very complicated
    - Error-prone
    - Potential deadlocks

The Distributed Array pattern

- Problem
  - Arrays often need to be partitioned between multiple UEs. How can we do this so the resulting program that is both readable and efficient?

The Distributed Array pattern

- Context
  - CPUs much faster than large memory subsystems
    - NUMA
  - Networks connecting nodes much slower than memory buses
  - Access time vary substantially depending on which PE is accessing which array element
  - The challenge is to organize the arrays so that the elements needed by each UE are nearby at the right time in the computation
    - The arrays must be distributed, so that the array distribution matches the flow of the computation

The Distributed Array pattern

- Forces
  - Load balance
    - The computational load among the UEs must be distributed so each UE takes nearly the same time to compute
  - Effective memory management
    - Good use of the memory hierarchy
    - The memory references implied by a series of calculations are close to the CPU
  - Clarity of abstraction

The Distributed Array pattern

- Solution
  - Simple to state at a high level, details make it complicated
  - Partition the global array into blocks
  - Map those blocks onto the UEs
  - Equal amount of work for each UE
  - Each UE's blocks stored in a local array
    - Unless all UEs share a single address space
    - The code will access elements of the distributed array using indices into a local array
  - Solution is based on indices into the global array
    - How to transform global indices to local and back using UE's identification

The Distributed Array pattern

- Solution – cont.
  - Array distributions
    - 1D block (column block, row block)
      - Decomposed in one dimension
      - Distributed one block per UE
    - 2D block
      - A rectangular subblock
      - One block to each UE
    - Block-cyclic
      - The array decomposed into blocks (using 1D or 2D partition)
      - More blocks than UEs
      - Blocks assigned round-robin to UEs

The Distributed Array pattern

- Solution – cont.
  - Choosing a distribution
    - Consider, how the computational load on the UEs changes as the computation proceeds?
  - Mapping indices
    - The relationship between global indices and the combination of UE and local indices transparent as possible
    - Use macros or inline functions for transformations
  - Aligning computation with locality
    - The loops that update local data should be organized in a way that gets as much use as possible out of each memory reference