Additional Patterns for Parallel Application Programs*

Berna L. Massingill† Timothy G. Mattson‡
Beverly A. Sanders§

Abstract

We are developing a pattern language to guide the programmer through the entire process of developing a parallel application program. The pattern language includes patterns that help find the concurrency in the problem, patterns that help find the appropriate algorithm structure to exploit the concurrency in parallel execution, and patterns describing lower-level implementation issues. Other patterns in the pattern language can be seen at `http://www.cise.ufl.edu/research/ParallelPatterns`.

In this paper, we briefly outline the overall structure of the pattern language and present selected patterns from the group of patterns that represent different strategies for exploiting concurrency once it has been identified.

1 Introduction

We are developing a pattern language for parallel application programs. The goal of the pattern language is to lower the barrier to parallel programming by guiding the programmer through the entire process of developing a parallel program. In our vision of parallel program development, the programmer brings into the process a good understanding of the actual problem to be solved, then works through the pattern language, eventually obtaining a detailed design or even working code. The pattern language is organized into three *design spaces*, each corresponding to one major phase in the design-and-development process:

- The *FindingConcurrency* design space includes high-level patterns that help find the concurrency in a problem and decompose it into a collection of tasks. These

†Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL; `blm@cise.ufl.edu` (current address: Department of Computer Science, Trinity University, San Antonio, TX; `bmassing@trinity.edu`).

‡Intel Corporation; `timothy.g.mattson@intel.com`.

§Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL; `sanders@cise.ufl.edu`.

patterns, presented in [MMS00], help the designer determine (i) how to decompose the problem into tasks that can execute concurrently, (ii) which data is local to the tasks and which is shared among tasks, and (iii) what ordering and data-dependency constraints exist among tasks.

- The *AlgorithmStructure* design space contains patterns that help find an algorithm structure to exploit the concurrency that has been identified. These patterns, most of which are presented in [MMS99] and [MMS02], capture recurring solutions to the problem of turning problems into parallel algorithms; with one exception, each pattern represents a high-level strategy for exploiting available concurrency. (The exception is *ChooseStructure*, which helps the designer select an appropriate pattern from among the other patterns in this design space.) Examples of patterns in this design space are *EmbarrassinglyParallel* [MMS99] and *DivideAndConquer* [MMS02].
- The *SupportingStructures* design space includes patterns that represent an intermediate stage between the problem-oriented patterns of the *AlgorithmStructure* design space and the APIs needed to implement them. Examples of pattern in this design space are *ForkJoin* [MMS02] and *SharedQueue* [MMS02].

The whole pattern language can be seen at `http://www.cise.ufl.edu/research/ParallelPatterns`. It consists of a collection of extensively hyperlinked documents, such that the designer can begin at the top level and work through the pattern language by following links. In this paper, we present the complete text of the *AlgorithmStructure* patterns not presented in [MMS99] and [MMS02]: *EventBasedCoordination*, *InseparableDependencies*, and *RecursiveData* (Sections 2, 3, and 4 respectively). Each of these sections represents one document in the collection of hyperlinked documents making up our pattern language; each document represents one pattern. To make the paper self-contained, we replace hyperlinks with text formatted like this and footnotes or citations. To make it easier to identify patterns and pattern sections, we format pattern names as *SomePattern* and pattern section names as **Some Section**. We also include an appendix (Appendix A) outlining the pattern language as a whole and sketching an example of its use in developing an application.

2 The *EventBasedCoordination* Pattern

Problem

If your application can be decomposed into a collection of semi-independent tasks interacting in an irregular fashion, how can you implement these tasks and their interaction?

Context

Some problems are most naturally represented as a collection of semi-independent entities interacting in an irregular way. As a real-world analogy, consider a newsroom, with reporters, editors, fact-checkers, and other employees collaborating on stories. As

reporters finish stories, they send them to the appropriate editors; an editor can decide to send the story to a fact-checker (who would then eventually send it back) or back to the reporter for further revision. Each employee is a semi-independent entity, and their interaction (e.g., a reporter sending a story to an editor) is irregular.

Many other examples can be found in the field of discrete-event simulation, i.e., simulation of a physical system consisting of a collection of objects whose interaction is represented by a sequence of discrete "events". An example of such a system is the car-wash facility described in [Mis86]: The facility has two car-wash machines and an attendant. Cars arrive at random times at the attendant. Each car is directed by the attendant to a non-busy car-wash machine if one exists, or queued if both machines are busy. Each car-wash machine processes one car at a time. The goal is to compute, for a given distribution or arrival times, the average time a car spends in the system (time being washed plus any time waiting for a non-busy machine) and the average length of the queue that builds up at the attendant. The "events" in this system include cars arriving at the attendant, cars being directed to the car-wash machines, and cars leaving the machines. Figure 1 sketches this example. Notice that it includes "source" and "sink" objects to make it easier to model cars arriving and leaving the facility. Notice also that the attendant must be notified when cars leave the car-wash machines so that it knows whether the machines are busy.

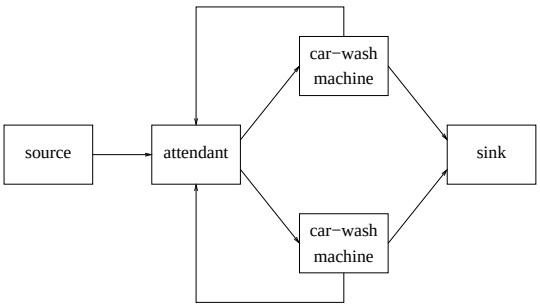


Figure 1: Discrete-event simulation of a car-wash facility. Arrows indicate flow of events.

For problems such as this, it often makes sense to base a parallel algorithm on defining a task for each entity; interaction between these tasks is then based on ordering constraints (e.g., "this calculation in task A must happen before that calculation in task B").

Indications

Use this pattern when:

- The most natural decomposition of your problem is into tasks whose interaction is organized by ordering constraints.

This pattern is particularly appropriate when:

- There are many fine-grained ordering constraints.
- The ordering constraints are irregular or dynamically generated.

Forces

- A good solution should make it simple to express the ordering constraints, which may be numerous and irregular and even arise dynamically. It should also make it possible for as many activities as possible to be performed concurrently.
- Ordering constraints can be expressed by encoding them into the program (e.g., via sequential composition) or using shared variables (e.g., semaphores or condition variables), but neither approach leads to solutions that are simple, capable of expressing complex constraints, and easy to understand even with a high degree of concurrency.

Solution

Overview

The best solution is based on expressing the ordering constraints as abstractions called "events", with each event having a task that generates it and a task that is to process it. Computation within each task consists of processing events.

Key Elements

- **Defining the tasks.** The basic structure of each task consists of receiving an event, processing it, and possibly generating events, as shown in Figure 2. The

```
initialize
while(not done)
{
  receive event
  process event
  send events
}
finalize
```

Figure 2: Basic structure of task in *EventBasedCoordination*.

  order in which tasks receive events must be consistent with the application's ordering constraints, as discussed below.

- **Representing event flow.** In order to allow communication and computation to overlap, one generally needs a form of asynchronous communication of events in which a task can create (send) an event and then continue without waiting for the recipient to receive it. In a message-passing environment, an event can be

represented by a message sent asynchronously from the task generating the event to the task that is to process it. In a shared-memory environment, a queue can be used to simulate message-passing. Since each such queue will be accessed by more than one task, it must be implemented in a way that allows safe concurrent access, as described in the *SharedQueue*[1] pattern. Other communication abstractions such as tuple spaces[2] can also be used effectively with this pattern.

- **Enforcing event ordering.** The enforcement of ordering constraints may make it necessary for a task to process events in a different order from the order in which they are sent, or to wait to process an event until some other event from a given task has been received, so it is usually necessary to be able to look ahead in the queue or message buffer and remove elements out of order. For example, consider the situation in Figure 3. Task 1 generates an event and sends it to task 2,

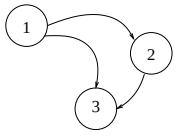


Figure 3: Event-based communication among three tasks. Task 2 generates its event in response to the event received from task 1. The two events sent to task 3 may arrive in either order.

which will process it, and also sends it to task 3, which is recording information about all events. Task 2 processes the event from task 1 and generates a new event, a copy of which is also sent to task 3. Suppose that the vagaries of the scheduling and underlying communication layer cause the event from task 2 to arrive before the event from task 1. Depending on what task 3 is doing with the events, this may or may not be problematic. If task 3 is simply tallying the number of events that occur, then there is no problem. If task 3 is writing a log entry that should reflect the order in which events are handled, however, simply processing events in the order in which they arrive would in this case produce an incorrect result. If task 3 is controlling a gate, and the event from task 1 results in opening the gate and the event from task 2 in closing the gate, then the out-of-order messages could cause significant problems, and task 3 should not process the first event until after the event from task 1 has arrived and been processed.

In discrete-event simulations, a similar problem can occur because of the semantics of the application domain. An event arrives at a station (task) along with a simulation time when it should be scheduled. An event can arrive at a station before other events with earlier times.

[1] A *SupportingStructures* pattern; see [MMS02].

[2] A tuple space is a conceptually shared repository for data containing objects called tuples that tasks use for communication in a distributed system. A small number of primitives are defined on the space and usually include primitives to insert tuples, remove tuples, and get a copy of a tuple. A pattern matching approach is used to select tuples of interest, and if a matching tuple is not in the space when requested, a task can wait until one arrives.

The first step is to determine whether, in a particular situation, out-of-order events can be a problem. There will be no problem if the "event" path is linear so that no out of order events will occur, or if the application semantics "don't care".

If out-of-order events may be a problem, then you can choose either an optimistic or pessimistic approach. An optimistic approach requires the ability to roll back the effects of events that are mistakenly executed (including the effects of any new events that have been created by the out of order execution). In the area of distributed simulation, this approach is called time warp [Jef85]. Optimistic approaches are usually not feasible if an event causes interaction with the outside world. Pessimistic approaches ensure that the events are always executed in order at the expense of increased latency and communication overhead. Pessimistic approaches do not execute events until it can be guaranteed "safe" to do so. In the figure, for example, task 3 cannot process an event from task 2 until it "knows" that no earlier event will arrive from task 1 and vice versa. Providing task 3 with that knowledge may require introducing null events that contain no information useful for anything except the event ordering and increased latency. Many implementations of pessimistic approaches are based on time stamps that are consistent with the causality in the system [Lam78].

Frameworks are available that take care of the details of event ordering in discrete-event simulation for both optimistic [RMC+98] and pessimistic approaches [CLL+99]. Middleware is available that deals event ordering problems in process groups caused by the communication system. An example is the Ensemble system developed at Cornell [vRBH+98].

- **Avoiding deadlocks.** It is possible for systems using this pattern to deadlock at the application level — for some reason the system arrives in a state where no task can proceed without first receiving an event from another task that will never arrive. This can happen because of a programming error; in the case of a simulation, it can also be caused by problems in the model that is being simulated. In the latter case, the developer must rethink the solution.

  If pessimistic techniques are used to control the order in which events are processed, then deadlocks may occur when an event is available and actually could be processed, but is not processed because the event is not yet known to be safe. The deadlock can be broken by exchanging enough information that the event can be safely processed. This is a very significant problem as the overhead of dealing with deadlocks can cancel the benefits of parallelism and make the parallel algorithms slower than a sequential simulation. Approaches to dealing with this type of deadlock range from sending frequent enough "null messages" to avoid deadlocks altogether (at the cost of many extra messages) to using deadlock detection schemes to detect the presence of a deadlock and then resolving it (at the cost of possible significant idle time before the deadlock is detected and resolved.) The approach of choice will depend on the frequency of deadlock, and this will generally need to be determined empirically. A middle ground is to use timeouts instead of accurate deadlock detection, and is often the best approach.

- **Scheduling and processor allocation.** The most straightforward approach is to allocate one task per processing element and allow all the tasks to execute concurrently. If insufficient processing elements are available to do this, then multiple tasks may be allocated to each processor. This should be done so as to achieve a good load balance. Load balancing is a difficult problem in this pattern due to its potentially irregular structure and possible dynamic nature. Some middleware infrastructures that support this pattern support task migration so that the load can be balanced dynamically at runtime.

Correctness Considerations

- **Out-of-order events.** If out-of-order events can be a problem for the application, be sure your design includes a way to process them correctly, as discussed in **Enforcing event ordering** above.
- **Deadlock.** Be sure your design either avoids deadlocks or includes a mechanism for detecting and breaking them, as discussed in **Avoiding deadlock** above.

Efficiency Considerations

- **Load balance.** Try to allocate tasks to processing elements in a way that makes for good load balance, as discussed in **Scheduling and processor allocation** above.
- **Efficient communication of events.** Be sure the mechanism used to communicate events is as efficient as is feasible. In a shared-memory environment, this means making sure the mechanism does not have the potential to become a bottleneck. In a message-passing environment, there are several efficiency considerations, for example whether it makes sense to send many short messages between tasks or try to combine them. [YWC+95] and [WY95] describe some considerations and solutions.

Examples

Known Uses

- The CSWEB application described in [YWC+95]: This application simulates the voltage output of combinational digital circuits (i.e., circuits without feedback paths). The circuit is partitioned into subcircuits; associated with each are input signal ports and output voltage ports, which are connected to form a representation of the whole circuit. The simulation of each subcircuit proceeds in a timestepped fashion; at each time step, the subcircuit's behavior depends on its previous state and the values read at its input ports (which correspond to values at the corresponding output ports of other subcircuits at previous time steps). Simulation of these subcircuits can proceed concurrently, with ordering constraints imposed by the relationship between values generated for output ports and values read on input ports. The solution described in [YWC+95] fits *EventBased-*

Coordination, defining a task for each subcircuit and representing the ordering constraints as events.

- Other discrete-event simulation applications, such as the following:
  - The DPAT simulation used to analyze air traffic control systems [Wie01]: This is a successful simulation that uses optimistic techniques. It is implemented using the GTW (Georgia Tech Time Warp) System [DFP+94]. The paper ([Wie01]) describes application-specific tuning and several general techniques that allow the simulation to work well without excessive overhead for the optimistic synchronization.
  - The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) [Met]: This is another optimistic simulation engine that has been used for large scale war gaming exercises.
  - The Scalable Simulation Framework (SSF) [CLL+99]: This is a simulation framework with pessimistic synchronization. It has been used for large-scale modeling of the Internet.

Related Patterns

This pattern is similar to *PipelineProcessing*[3] in that both patterns apply to problems in which it is natural to decompose the computation into a collection of semi-independent entities. There are two key differences. First, in *PipelineProcessing* the interaction among entities is fairly regular, with all "stages" of the pipeline proceeding in a loosely synchronous way, whereas in *EventBasedCoordination* there is no such requirement, and the entities can interact in very irregular and asynchronous ways. Second, in *PipelineProcessing* the overall structure (number of tasks and their interaction) is usually fixed, whereas in *EventBasedCoordination* the problem structure can be more dynamic.

3 The *InseparableDependencies* Pattern

Problem

If the decomposition of your application into tasks generates data dependencies that cannot be managed with the techniques inherent in any of the other *AlgorithmStructure* patterns, how do you manage these dependencies?

Context

Inherent in each of the other *AlgorithmStructure* patterns is some technique for managing data dependencies among tasks, from the trivial to the complex: no dependencies and hence nothing to manage in *EmbarrassinglyParallel*[4]); dependencies that can be managed by replicating data in *SeparableDependencies*[5]; dependencies that

[3] Another *AlgorithmStructure* pattern; see [MMS02].
[4] Another *AlgorithmStructure* pattern; see [MMS99].
[5] Another *AlgorithmStructure* pattern; see [MMS99].

can be managed by alternating computation and nearest-neighbor communication in *GeometricDecomposition*[6]; and so forth. For many algorithms, these techniques are sufficient to manage all data dependencies among tasks. For some algorithms, however, there are data dependencies that cannot be managed with one of the other *AlgorithmStructure* patterns.

For example, consider the *phylogeny problem* from molecular biology, as described in [YWC+95]. Omitting most of the details, in this application the computation consists of examining all members of a power set and rejecting those that do not meet a consistency criterion. Different sets can be examined concurrently, so one can think of defining a task for each set and partitioning them among units of execution[7] (UEs). However, not all sets must necessarily be examined — if a set *S* is rejected, all supersets of *S* can also be rejected. Thus, it makes sense to keep track not only of which sets have not yet been examined (as one does for dynamic scheduling of tasks) but also which sets have been rejected. This problem could be readily solved using *EmbarrassinglyParallel* if it were not for the fact that all tasks need access to whatever data structure is used to keep track of the solution so far (which tracks which sets have been rejected), and they may need both read and write access. Replicating this data structure before beginning the parallel computation will not solve the problem because it changes during the computation, so *SeparableDependencies* is not a complete solution either. Partitioning the data structure and basing a solution on this data decomposition is likely to lead to poor load balance because the way in which the elements are rejected is unpredictable, so *GeometricDecomposition*[8] is not a good choice. The best solution, therefore, would probably be to start with *EmbarrassinglyParallel* and add something to explicit manage the data dependencies (access to the shared data structure).

Indications

Use this pattern when:

- The problem can be solved with one of the other *AlgorithmStructure* patterns, *except* that there are data dependencies among tasks that are not addressed by the selected pattern. (Generally, this will be the case when there is at least one data structure that must be accessed by multiple tasks in the course of the program's execution, and either at least one task must modify the data structure or there is no practical way to either share the data structure among tasks (e.g., the target platform does not provide shared memory) or give each task a copy (e.g., the data structure is too large to make copying it practical).)

Forces

- There is inherent lack of structure in the problem, and we would like to design abstractions that impose some structure.

[6] Another *AlgorithmStructure* pattern; see [MMS99].
[7] Generic term for a collection of concurrently-executing entities, usually either processes or threads.
[8] Another *AlgorithmStructure* pattern; see [MMS99].

- Well-chosen abstractions make for simplicity of understanding, but this may come at the expense of inadequate concurrency.
- Simple ways to manage the "inseparable dependencies" of this pattern are easier to implement and more likely to be correct, but they may reduce concurrency.
- More complex ways to manage the dependencies may allow more concurrency but be more difficult to implement and more prone to error.

Solution

Overview

As suggested in **Indications**, this pattern describes modifications to another *Algorithm-Structure* pattern (which we will call the *base pattern*) to explicitly manage data dependencies not addressed by the base pattern, such as the access to a shared data structure in the example in **Context**. We therefore do not discuss the overall structure of the solution (which will be that of the base pattern), only the additions necessary to manage the otherwise-problematical data dependencies. The overall approach is to start with a simple solution and try more complex solutions if necessary to obtain acceptable performance. For this pattern, therefore, **Key elements** lists the steps you should follow to achieve this goal.

Key Elements

- **Be sure you want this pattern.** The first step is to make really sure this is the right pattern to use; it may be worthwhile to revisit decisions made earlier in the design process (the decomposition into tasks, for example) to see whether different decisions might lead to a solution that fits one of the other *Algorithm-Structure* patterns. In particular, if the base pattern is *EmbarrassinglyParallel*, it may be worthwhile to consider again whether it is possible to instead use *SeparableDependencies*.
- **Define an abstract data type.** The best way to approach a solution is to view the shared data as an abstract data type (ADT) with a fixed set of (possibly complex) operations on the data. For example, if the shared data structure is a queue, these operations would include put (enqueue), take (dequeue), and possibly other operations such as a test for an empty queue or a test to see if a specified element is present. Each task will typically be performing a sequence of these operations. These operations should have the property that if they are executed serially (i.e., one at a time, without interference from other tasks), each operation will leave the data in a consistent state. (Another way to say this is that these operations are "atomic".)
- **Implement an appropriate concurrency-control protocol.** Once these operations are identified, the objective is to implement a concurrency-control protocol to ensure that these operations give the same results as if they were executed serially. There are several ways to do this; application designers should start with the

first method, which is the simplest, and then try the other more complex methods if it does not yield acceptable performance. Notice that these more complex methods can be combined if more than one is applicable.

– **One-at-a-time execution.** The easiest solution is to ensure that the operations are indeed executed serially.

  In a shared-memory environment, the most straightforward way to do this is to treat each operation as part of a single critical section and use a mutual-exclusion protocol to ensure that only one UE at a time is executing its critical section. This means that all of the operations on the data are mutually exclusive. Exactly how this is implemented will depend on the facilities of the target programming environment. Typical choices include mutex locks, synchronized blocks, and semaphores. If the programming language naturally supports the implementation of abstract data types, we recommend implementing each operation as a procedure or method, with the mutual-exclusion protocol implemented in the method itself.

  In a message-passing environment, the most straightforward way to ensure serial execution involves assigning the shared data structure to a processing element. Each operation should correspond to a message type; other processes request operations by sending messages to the process managing the data structure, which processes them serially.

  In either environment, this approach is usually not difficult to implement, but it may be overly conservative (i.e., it may disallow concurrent execution of operations that would be safe to execute simultaneously), and it may produce a bottleneck that negatively affects the performance of the program. If this is the case, the remaining approaches described in this section should be reviewed to see whether one of them can reduce or eliminate this bottleneck and give better performance.

– **Noninterfering sets of operations.** One approach to improving performance begins by analyzing the interference between the operations. We say that operation A *interferes with* operation B if A writes a variable that B reads. Notice that an operation may interfere with itself and that this would be a concern if more than one task executes the same operation (e.g., more than one task executes "take" operations on a shared queue). You may find, for example, that the operations fall into two disjoint sets, where the operations in different sets do not interfere with each other. In this case, you can increase the amount of concurrency by treating each of the sets as different critical sections. That is, within each set operations execute one a time, but operations in different sets can proceed concurrently.

– **Readers/writers.** If there is no obvious way to partition the operations into disjoint sets, consider the type of interference. You may find that some of the operations modify the data, but others only read it. For example, you might have operation A that both reads and writes the data and operation B that only reads the data (i.e., A interferes with itself and with B, but B does not interfere with itself). Thus, if one task is performing operation A, no other task should be able to execute either A or B, but any number of tasks

should be able to execute B concurrently. If this is the case, it may be worthwhile to implement a readers/writers protocol that will allow this to happen.

- **Reducing the size of the critical section.** Another approach to improving performance begins with analyzing the implementations of the operations in more detail. You may find that only part of the operation involves actions that interfere with other operations. If so, you can reduce the size of the critical section to that smaller part. Notice that this sort of optimization is very easy to get wrong, so it should be attempted only if it will give significant performance improvements over simpler approaches.
- **Application-specific optimizations.** Another approach is to consider application-specific optimizations. For example, you might notice that put and take in a queue do not interfere if the queue has at least two elements. It is therefore possible to implement concurrency-control protocols that would allow put and take to execute concurrently when safe to do so and serially otherwise. Notice that this sort of optimization, like reducing the size of the critical section, is very easy to get wrong, so it should only be attempted if it seems likely to yield a significant performance improvement (and you are sure you know what you are doing!).
- **Application-specific semantic relaxation.** Yet another approach is to consider partially replicating shared data (the "software caching" described in [YWC+95]) and perhaps even allowing the copies to be inconsistent if this can be done without affecting the results of the computation. For example, a distributed-memory solution to the phylogeny problem described earlier might give each UE its own copy of the set of sets already rejected and allow these copies to be out of synch; tasks may do extra work (in rejecting a set that has already been rejected by a task assigned to a different UE), but this extra work will not affect the result of the computation, and it may overall be more efficient than the communication cost of keeping all copies in synch.

Correctness Considerations

- **Correctness considerations of base pattern.** Review the correctness considerations for the base pattern.
- **Access to shared data.** Ensure that access to shared data is "safe" — in the terms of the preceding discussion, tasks can only execute operations concurrently if the operations do not interfere with each other.
- **Memory synchronization.** Make sure memory is synchronized as required: Caching and compiler optimizations may result in unexpected behavior of shared variables. For example, a stale value of a variable may be read from a cache or register instead of the newest value written by another task, or the latest value from another task may not have been flushed to memory and thus is not visible to other tasks. Unfortunately, techniques to avoid such problems are very platform-specific. In OpenMP the flush directive can be used to synchronize memory

explicitly; it is implicitly invoked by several other directives. In Java, memory is implicitly synchronized when entering and leaving a synchronized block.

Efficiency Considerations

- **Efficiency considerations of base pattern.** Review the efficiency considerations for the base pattern.
- **Task scheduling.** Consider whether the explicitly-managed data dependencies addressed by this pattern affect task scheduling. A key goal in deciding how to schedule tasks is good load balance; in addition to the considerations described in the base pattern, the application designer should also take into account that tasks may be suspended waiting for access to shared data. It may therefore make sense to try to assign tasks in way that minimizes such waiting, or to assign multiple tasks to each UE in the hope that there will always be one task per UE that is not waiting for access to shared data.
- **Efficient access to shared data.** Make sure the mechanism used to ensure safe access to shared data does not unnecessarily restrict concurrency, as described in the preceding section.

Examples

Known Uses

- The phylogeny problem described in **Context** above: The solution described in [YWC+95] fits *EmbarrassinglyParallel* plus explicit management of the required shared data structure using replication and periodic updates to reestablish consistency among copies.
- The Gröbner basis program [YWC+95]: Omitting most of the details, in this application the computation consists of using pairs of polynomials to generate new polynomials, which are then compared against a master set of polynomials, and those that are not linear combinations of elements of the master set are added to it and used to generate new pairs. Different pairs can be processed concurrently, so one can think of defining a task for each pair and partitioning them among UEs. The solution described in [YWC+95] fits *EmbarrassinglyParallel* (with a "task queue" consisting of pairs of polynomials) plus explicit management of the master set using an application-specific protocol the authors call "software caching".

Related Patterns

As noted, this pattern really represents a way of extending or modifying one of the other *AlgorithmStructure* patterns. If the base pattern is *EmbarrassinglyParallel*, this pattern can be thought of as a "pattern of last resort" for task-based decompositions — a pattern that tells you how to manage data dependencies when none of the other task-based patterns will work. Other anticipated uses are to provide safe access to a

shared data structure not described as part of the base pattern (for example, the shared task queue sometimes used to implement *EmbarrassinglyParallel*, or the neighboring sections of a distributed-among-tasks data structure in *GeometricDecomposition*).

4 The *RecursiveData* Pattern

Problem

Your problem involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing. How can you perform operations on these data structures in parallel?

Context

Many problems with recursive data structures naturally use the divide and conquer strategy described in *DivideAndConquer*[9]. In other situations, we often find that most operations on recursive data structures such as lists and trees seem to have little exploitable concurrency, since one must follow the links that make up the data structure sequentially. Surprisingly, however, it is sometimes possible to reshape such algorithms in a way that a program can operate concurrently on all elements of the data structure.

For example, suppose we have a forest of rooted directed trees (defined by specifying, for each node, its immediate ancestor, with a root node's ancestor being itself) and want to compute, for each node in the forest, the root of the tree containing that node. To do this in a sequential program, we would probably trace depth-first through each tree from its root to its leaf nodes; as we visit each node, we have the needed information about the corresponding root. Total running time of such a program for a forest of N nodes would be $O(N)$.

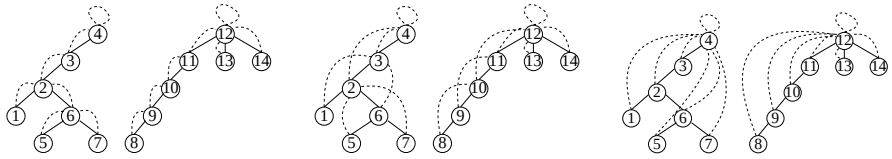
A parallel version can be created as described in [JáJá92]: Let each node's "successor" initially be defined to be its parent. For each step, we calculate for every node its "successor's successor". For a node whose successor is the desired result (the root of the tree to which the node belongs), the successor and the successor's successor are the same (since a root's ancestor is itself), so we must continue the "for each step" process until it converges, i.e., until the values produced by one step are the same as the values produced by the preceding step. Figure 4 shows an example requiring three steps to converge.

Notice that at each step we can operate on all N nodes in the tree concurrently and that the algorithm converges in at most $logN$ steps.

An interesting aspect of this restructuring is that the new algorithm involves substantially more total work than the original sequential one ($O(NlogN)$ versus $O(N)$), but the restructured algorithm contains potential concurrency that if fully exploited reduces total running time to $O(logN)$ (versus $O(N)$).

Most strategies and algorithms based on this pattern similarly trade off an increase in total work for a potential decrease in execution time. Notice also that the exploitable

[9] Another *AlgorithmStructure* pattern; see [MMS02].



(a) Step 1 (b) Step 2 (c) Step 3

Figure 4: Finding roots in a forest. Solid lines represent the original parent-child relationships among nodes; dashed lines point from nodes to their successors.

concurrency can be extremely fine-grained (as in the example above), making this pattern an impractical choice for many current programming environments. If the pattern is not effective in practice, however, it may still serve as an inspiration for lateral thinking about how to parallelize problems that at first glance appear to be inherently sequential.

The technique is sometimes referred to as pointer jumping or recursive doubling.

Indications

Use this pattern when:

- The original problem involves a recursive data structure.
- The problem does not fit *DivideAndConquer*.

This pattern can be particularly effective when:

- The target platform supports efficient execution of very fine-grained concurrency.

Forces

- The strategies represented by this pattern offer significant potential speedups.
- Applying these strategies requires significant cleverness on the part of the designer.
- The resulting concurrency may be too fine-grained to be practical on some platforms.

Solution

In contrast to the high degree of cleverness involved in the restructuring of the algorithm (usually in terms of a simultaneous update of all elements of the key data structure), the implementation of the pattern is usually fairly straightforward.

Key Elements

- **Data decomposition.** You need a way of assigning each element of the recursive data structure to a unit of execution (UE)[10]. Typically this pattern will be implemented on a platform that allows unit of execution and hence each element to be assigned to a different processing element (for example, a SIMD machine such as one of the early Connection Machines). If there are not enough processing elements to do this, it is also possible to assign more than one element to each UE. Ideally, this is done in a way that permits unit-time access to each element of the data structure.
- **Structure.** The basic structure of one of these algorithms is a sequential composition in the form of a loop, in which each iteration can be described as "perform this operation simultaneously on all (or selected) elements of the recursive data structure". Typical operations include "replace each element's successor with its successor's successor" (as in the example in the **Context** section) and "replace a value held at this element with the sum of the current value and the value of the predecessor's element."
- **Synchronization.** Algorithms that fit this pattern are described in terms of *simultaneously* updating all elements of the data structure. Some target platforms (e.g., SIMD) may make this trivial to accomplish since instructions are executed in a lockstep fashion at each processing element and each data element will generally be assigned to a separate processing element.

  If the target platform is less synchronized, it will generally be necessary to introduce the synchronization explicitly. For example, if the operation performed during a loop iteration contains the assignment

  ```
  next[k] = next[next[k]]
  ```

  then you must ensure that `next[k]` is not updated before other UEs that need its value for their computation have received it. One common technique is to introduce a new variable, say `next2`, at each element. Even-numbered iterations then read `next` but update `next2`, while odd-numbered iterations read `next2` and update `next`. The necessary synchronization is accomplished by placing a barrier between each successive pair of iterations.

  If there are fewer processing elements than data elements, you must decide whether to assign each data element to a UE and assign multiple UEs to each processing element (thereby simulating some of the parallelism) or whether to assign multiple data elements to each UE and process them serially. The latter is less straightforward but may be more efficient.

Correctness Considerations

- **Initial design.** Be careful that in rethinking the problem to expose concurrency you do not introduce errors.

[10] Generic term for one of a collection of concurrently-executing entities, usually either processes or threads.

- **Simultaneous updates.** If the redesigned algorithm involves simultaneous updates of all elements of the data structure, be sure such updates are performed correctly. On a SIMD platform with one processing element per data element, this will happen automatically; on other platforms, it may be necessary to use barrier synchronization or data-copying.

Efficiency Considerations

- **Granularity.** Be sure the target platform allows efficient execution of programs with whatever granularity of concurrency is required (sometimes very fine-grained for this pattern). For example, if frequent barrier synchronization is needed, the cost of these barrier operations could overwhelm the speedup gained by updating all parts of the data structure in parallel.

Examples

Partial Sums of a Linked List

This example is adopted from Hillis and Steele [HS86]. Each element in a linked list contains a value x. The problem is to compute the prefix sums of all the elements in the list. In other words, after the computation is complete, the first element will contain x_0, the second will contain $x_0 + x_1$, the third $x_0 + x_1 + x_2$, etc.

Figure 5 shows pseudocode for the basic algorithm. Figure 6 shows the evolution

```
for all k in parallel
{
    temp[k] = next[k];
    while temp[k] != null
    {
        x[temp[k]] = x[k] + x[temp[k]];
        temp[k] = temp[temp[k]];
    }
}
```

Figure 5: Pseudocode for finding partial sums of a list.

of the computation where x_i is the initial value of the $(i-1)$th element in the list.

This example can be generalized by replacing addition with any associative operator and is sometime known as a prefix scan. It can be used in a variety of situations, including solving various types of recurrence relations.

Known Uses

- Data-parallel algorithms: This pattern is an example of a style of programming sometimes referred to as *data parallel* and used for SIMD platforms such as the Connection Machine. These platforms support the fine-grained concurrency required for the pattern and handle synchronization automatically since every computation step occurs in lockstep on all the processors. Hillis and Steele [HS86]

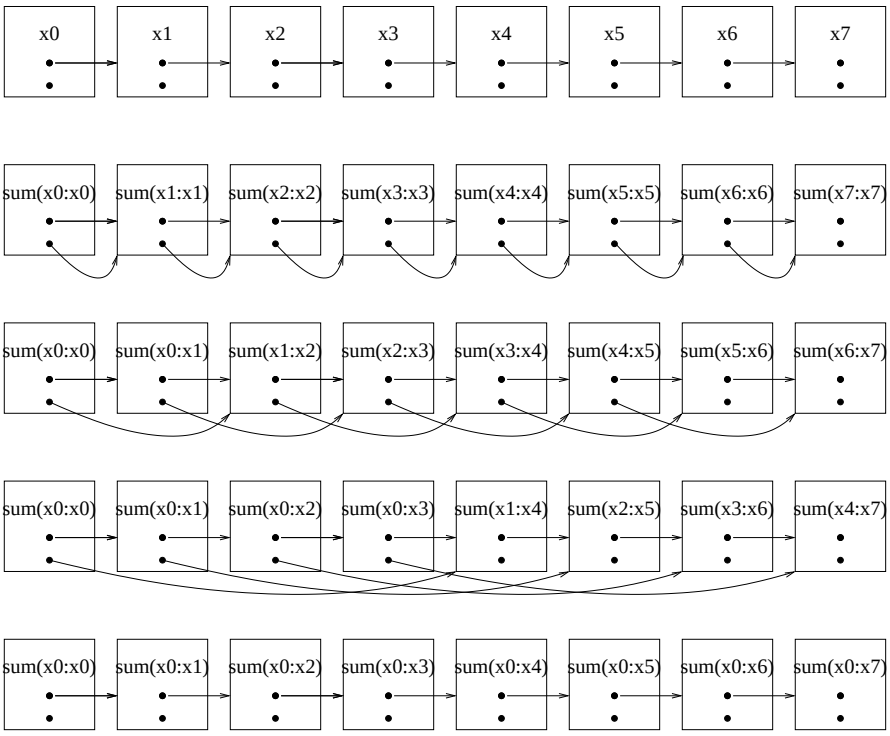


Figure 6: Steps in finding partial sums of a list: Straight arrows represent links between elements; curved arrows indicate additions.

describe several interesting applications of this pattern, including finding the end of a linked list, computing all partial sums of a linked list, region labelling in two dimensional images, and parsing.

- Combinatorial optimization: In combinatorial optimization, problems involving traversing all nodes in a graph or tree can often be solved handled with this pattern by first finding an ordering on the nodes to create a list. Euler tours and ear decomposition [EG88] are well-known techniques to compute this ordering.

Related Patterns

With respect to the actual concurrency, this pattern is very much like *GeometricDecomposition*[11], a difference being that in this pattern the data structure containing the elements to be operated on concurrently is recursive (at least conceptually). What makes it different is the emphasis on fundamental rethinking to expose unexpected fine-grained concurrency.

Acknowledgments

We thank Intel Corporation, the National Science Foundation (grant #9704697), and the Air Force Office of Scientific Research (grant #4514209-12) for their financial support. We also thank Doug Lea, Alan O'Callaghan, Bob Hanmer, and Steve MacDonald, who acted as shepherds for our previous PLoP papers ([MMS99], [MMS00], [MMS01], and [MMS02]), and Aamod Sane, who shepherded this paper.

References

[CLL+99] James Cowie, Hongbo Liu, Jason Liu, David Nicol, and Andy Agielski. Towards realistic million-node internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing*, 1999. See also `http://www.ssfnet.org`.

[DFP+94] S. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, 1994.

[EG88] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Reviews in Computer Science*, 3:233–283, 1988.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[JáJá92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[11] Another *AlgorithmStructure* pattern; see [MMS99].

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.

[Met] Metron, Inc. SPEEDES (synchronous parallel environment for emulation and discrete-event simulation). `http://www.speedes.com`.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1), 1986.

[MMS99] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS00] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for finding concurrency for parallel application programs. In *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP 2000)*, August 2000. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS01] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. More patterns for parallel application programs. In *Proceedings of the Eighth Pattern Languages of Programs Workshop (PLoP 2001)*, 2001. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS02] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Some algorithm structure and support patterns for parallel application programs. In *Proceedings of the Ninth Pattern Languages of Programs Workshop (PLoP 2002)*, 2002. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[RMC+98] R. Radhakrishnan, D. E. Martin, M. Chetlur, D. Madhava Rao, and P. A. Wilsey. An object-oriented Time Warp simulation kernel. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98) LNCS 1505*, 1998. Also available as `http://www.ececs.uc.edu/~paw/lab/papers/warped/iscope98.ps.gz`. See also `http://www.ece.uc.edu/~paw/warped`.

[vRBH+98] Robbert van Renesse, Keneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 28(9):963–979, August 1998. See also `http:www.cs.cornell.edu/Info/Projects/Ensemble`.

[Wie01] Frederick Wieland. Practical parallel simulation applied to aviation modeling. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 109–116, 2001.

[WY95] C.-P. Wen and K. Yelick. Portable runtime support for asynchronous simulation. In *International Conference on Parallel Processing*, August 1995.

[YWC+95] K. Yelick, C.-P. Wen, S. Chakrabarti, E. Deprit, J. Jones, and A. Krishnamurthy. Portable parallel irregular applications. In *Workshop on Parallel Symbolic Languages and Systems*, October 1995. To appear in *Lecture Notes in Computer Science*.

A Overview of the Pattern Language

As mentioned in Section 1, our pattern language is organized into three *design spaces*, each corresponding to one major phase in the design-and-development process. In this appendix we describe these design spaces and sketch an example of using the pattern language to develop an application.

A.1 The *FindingConcurrency* Design Space

The patterns in this space (presented in [MMS00]) are used early in the design process, once the problem and its key computations and data structures are understood. They help programmers understand how to expose the exploitable concurrency in their problems. More specifically, these patterns help the programmer

- Identify the entities into which the problem will be decomposed.
- Determine how the entities depend on each other.
- Construct a coordination framework to manage the parallel execution of the entities.

These patterns collaborate closely with the *AlgorithmStructure* patterns, and one of their main functions is to help the programmer select an appropriate pattern in the *AlgorithmStructure* design space. Experienced designers might know how to do this immediately, in which case they could move directly to the patterns in the *AlgorithmStructure* design space.

The patterns in this design space are organized as illustrated in Figure 7. The main pathway through the patterns proceeds through three major patterns:

- *DecompositionStrategy*: This pattern helps the programmer decide whether the problem should be decomposed based on a data decomposition, a task decomposition, or a combination of the two.
- *DependencyAnalysis*: Once the entities into which the problem will be decomposed have been identified, this pattern helps the programmer understand how they depend on each other.
- *DesignEvaluation*: This pattern is a consolidation pattern. It is used to evaluate the results of the other patterns in this design space and prepare the programmer for the next design space, the *AlgorithmStructure* design space.

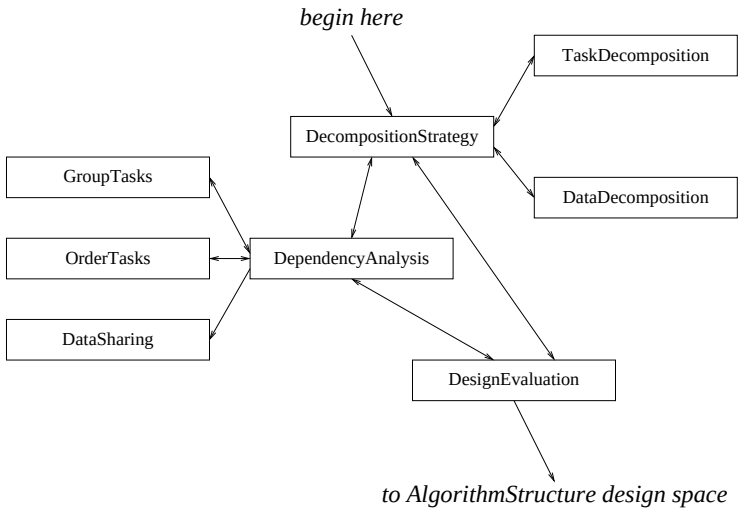


Figure 7: Organization of the *FindingConcurrency* design space.

Branching off from the *DecompositionStrategy* and *DependencyAnalysis* patterns are groups of patterns that help with problem decomposition and dependency analysis. We use double-headed arrows for most of the pathways in the figure to indicate that one may need to move back and forth between the patterns repeatedly as the analysis proceeds. For example, in a dependency analysis, the programmer may group the tasks one way and then determine how this grouping affects the data that must be shared between the groups. This sharing may imply a different way to group the tasks, leading the programmer to revisit the tasks grouping. In general, one can expect working through these patterns to be an iterative process. As part of this process it is usually necessary to at least consider the platform or platforms on which the final program is to run. The need for portability encourages postponing decisions about target platform as much as possible. There are times, however, when delaying consideration of platform-dependent issues can result in choosing a poor algorithm.

Example analysis

As an example of the analysis performed by the patterns in this design space, consider the following problem taken from the field of medical imaging. (This example is presented in more detail in [MMS00].) We can decompose this problem in two ways — in terms of tasks and in terms of data.

An important diagnostic tool is to give a patient a radioactive substance and then watch how that substance propagates through the body by looking at the distribution of emitted radiation. Unfortunately, the images are of low resolution, due in part to the scattering of the radiation as it passes through the body. It is also difficult to reason from the absolute radiation intensities, since different pathways through the body attenuate

the radiation differently.

To solve this problem, medical imaging specialists build models of how radiation propagates through the body and use these models to correct the images. A common approach is to build a Monte Carlo model. Randomly selected points within the body are assumed to emit radiation (usually a gamma ray), and the trajectory of each ray is followed. As a particle (ray) passes through the body, it is attenuated by the different organs it traverses, continuing until the particle leaves the body and hits a camera model, thereby defining a full trajectory. To create a statistically significant simulation, thousands if not millions of trajectories are followed.

The problem can be parallelized in two ways. Since each trajectory is independent, it would be possible to parallelize the application by associating each trajectory with a task. Another approach would be to partition the body into sections and assign different sections to different processing elements.

As in many ray-tracing codes, there are no dependencies between trajectories, making the task-based decomposition the natural choice. By eliminating the need to manage dependencies, the task-based algorithm also gives the programmer plenty of flexibility later in the design process, when how to schedule the work on different processing elements becomes important.

The data decomposition, however, is much more effective at managing memory utilization. This is frequently the case with a data decomposition as compared to a task decomposition. Since memory is decomposed, data-decomposition algorithms also tend to be more scalable. These issues are important and point to the need to at least consider the types of platforms that will be supported by the final program. The need for portability drives one to make decisions about target platforms as late as possible. There are times, however, when delaying consideration of platform-dependent issues can lead one to choose a poor algorithm.

A.2 The *AlgorithmStructure* Design Space

This design space is concerned with structuring the algorithm to take advantage of potential concurrency. That is, the designer working at this level reasons about how to use the concurrency exposed in the previous level. Patterns in this space describe overall strategies for exploiting concurrency.

Patterns in this design space can be divided into three groups as shown in Figure 8, plus *ChooseStructure* [MMS02], which addresses the question of how to use the analysis performed by using the *FindingConcurrency* patterns to select an appropriate pattern from those in this space.

"Organize by ordering" patterns

These patterns are used when the *ordering of groups of tasks* is the major organizing principle for the parallel algorithm. This group has two members, reflecting two ways task groups can be ordered. One choice represents "regular" orderings that do not change during the algorithm; the other represents "irregular" orderings that are more dynamic and unpredictable.

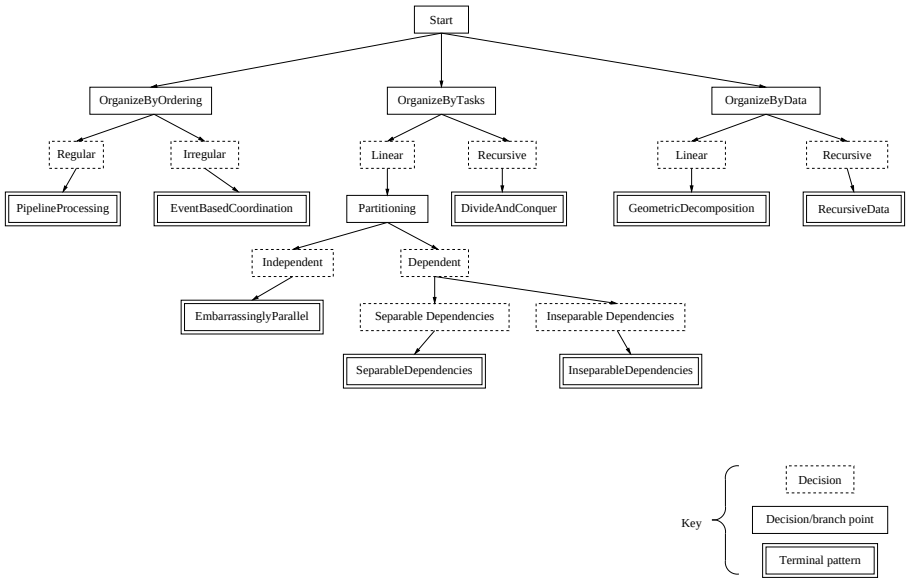


Figure 8: Organization of the *AlgorithmStructure* design space.

- *PipelineProcessing* [MMS02]: The problem is decomposed into ordered groups of tasks connected by data dependencies.
- *EventBasedCoordination* (Section 2 of this paper): The problem is decomposed into groups of tasks that interact through asynchronous events.

"Organize by tasks" patterns

These patterns are those for which the tasks themselves are the best organizing principle. There are many ways to work with such "task-parallel" problems, making this the largest pattern group.

- *EmbarrassinglyParallel* [MMS99]: The problem is decomposed into a set of independent tasks. Most algorithms based on task queues and random sampling are instances of this pattern.
- *SeparableDependencies* [MMS99]): The parallelism is expressed by splitting up tasks among units of execution (threads or processes). Any dependencies between tasks can be pulled outside the concurrent execution by replicating data prior to the concurrent execution and then combining the replicated data after the concurrent execution. This pattern applies when variables involved in data dependencies are written but not subsequently read during the concurrent execution.

- *InseparableDependencies* (Section 3 of this paper): The parallelism is usually expressed by splitting up tasks among units of execution. In this case, however, variables involved in data dependencies are both read and written during the concurrent execution in a way that is not addressed by any of the other *AlgorithmStructure* patterns and therefore must be explicitly managed as the tasks execute.
- *DivideAndConquer* [MMS02]: The problem is solved by recursively dividing it into subproblems, solving each subproblem independently, and then recombining the subsolutions into a solution to the original problem.

"Organize by data" patterns

These patterns are those for which the decomposition of the data is the major organizing principle in understanding the concurrency. There are two patterns in this group, differing in how the decomposition is structured (linearly in each dimension or recursively).

- *GeometricDecomposition* [MMS99]: The problem space is decomposed into discrete subspaces; the problem is then solved by computing solutions for the subspaces, with the solution for each subspace typically requiring data from a small number of other subspaces. Many instances of this pattern can be found in scientific computing, where it is useful in parallelizing grid-based computations, for example.
- *RecursiveData* (Section 4 of this paper): The problem involves an operation on a recursive data structure that appears inherently sequential but can be rethought in a way that exposes exploitable concurrency.

A.3 The *SupportingStructures* Design Space

The patterns of the *AlgorithmStructure* design space capture recurring solutions to the problem of turning problems into parallel algorithms. But these patterns in turn contain recurring solutions to the problem of mapping high-level parallel algorithms into programs using a particular parallel language or library. Those solutions are what the patterns in the *SupportingStructures* design space capture. Patterns in this space fall into three main groups: patterns for structuring concurrent execution, patterns that represent commonly-occurring computations, and patterns that represent commonly-occurring data structures.

Patterns for structuring concurrent execution

Patterns in this group describe ways of structuring concurrent execution, with particular attention to whether processes or threads are created statically or dynamically and whether concurrently-executing processes or threads all perform the same work. These patterns include the following:

- *SPMD* (single program, multiple data) [MMS02]: The computation consists of *N* processes or threads running concurrently. All *N* processes or threads execute the same program code, but each operates on its own set of data. A key feature of the program code is a parameter that differentiates among the copies.
- *ForkJoin* [MMS02]: A main process or thread forks off some number of other processes or threads that then continue in parallel to accomplish some portion of the overall work before rejoining the main process or thread.

Patterns representing computational structures

Patterns in this group describe commonly-used computational structures; they include the following:

- *Reduction* [MMS02]: A number of concurrently-executing processes or threads cooperate to perform a reduction operation, in which a collection of data items is reduced to a single item by repeatedly combining them pairwise with a binary operator.
- *MasterWorker*: A master process or thread sets up a pool of worker processes or threads and a task queue. The workers execute concurrently, with each worker repeatedly removing a task from the task queue and processing it, until all tasks have been processed or some other termination condition has been reached. In some implementations no explicit master is present.

Patterns representing data structures

Patterns in this group describe commonly-used shared data structures; they include the following:

- *SharedQueue* [MMS02]: This pattern represents a "thread-safe" implementation of the familiar queue abstract data type (ADT), that is, an implementation of the queue ADT that maintains the correct semantics even when used by concurrently-executing processes or threads.
- *SharedCounter*: This pattern, like the previous one, represents a "thread-safe" implementation of a familiar abstract data type, in this case a counter with an integer value and increment and decrement operations.
- *DistributedArray*: This pattern represents a class of data structures often found in parallel scientific computing, namely arrays of one or more dimensions that have been decomposed into subarrays and distributed among processes or threads.