Patterns for Parallel Application Programs*

Berna L. Massingill, University of Florida, blm@cise.ufl.edu
Timothy G. Mattson, Intel Corporation, timothy.g.mattson@intel.com
Beverly A. Sanders, University of Florida, sanders@cise.ufl.edu

Abstract

We are involved in an ongoing effort to design a pattern language for parallel application programs. The pattern language consists of a set of patterns that guide the programmer through the entire process of developing a parallel program, including 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. The current version of the pattern language can be seen at `http://www.cise.ufl.edu/research/~ParallelPatterns`.

In the current note, we present three selected patterns from our pattern language, selected from the set of patterns that are used after the problem has been analyzed to identify the exploitable concurrency. The EmbarrassinglyParallel pattern is used when the problem can be decomposed into a set of independent tasks. The SeparableDependencies pattern can be used when dependencies between tasks can be pulled outside the concurrent execution by replicating data prior to the concurrent execution and then combining the replicated data afterwards. The GeometricDecomposition pattern is used when the problem space can be decomposed into discrete subspaces and the problem solved by first exchanging information among subspaces and then concurrently computing solutions for the subspaces.

1 Introduction

Parallel hardware has been available for decades, and is becoming increasingly mainstream. Parallel software that fully exploits the hardware is much rarer, however, and mostly limited to the specialized area of supercomputing. We believe that part of the reason for this state of affairs is that most parallel programming environments, which focus on the implementation of concurrency rather than higher-level design issues, are simply too difficult for most programmers to risk using them.

We are currently involved in an ongoing effort to design 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 four design spaces. The FindingConcurrency design space includes high-level patterns that help find the concurrency in a problem and decompose it into a collection of tasks. The AlgorithmStructure design space contains patterns that help find an appropriate algorithm structure to exploit the concurrency that has been identified. The SupportingStructures design space includes patterns that describe useful abstract data types and other supporting structures, and the ImplementationMechanisms design space contains patterns that describe lower-level implementation issues. The latter two design spaces (slightly stretching the typical notion of a pattern) might even include reusable code libraries or frameworks. We use a pattern format for all four levels so that we can address a variety of issues in a unified way. The current, incomplete, version of the 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 replace hyperlinks with footnotes and citations to make it self-contained.)

In the current note, rather than describing the pattern language as a whole, we present the complete text of three selected patterns from the AlgorithmStructure design space. The chosen patterns are relatively mature, and significant enough to stand alone. The patterns in the AlgorithmStructure design space help the designer find an appropriate algorithm structure suitable for parallel implementation and are applicable after the concurrency in a problem has been identified. Thus, before attempting to apply these patterns the designer should have determined (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 dependencies exist among tasks. The following three patterns are given:

- The EmbarrassinglyParallel pattern is used when the problem can be decomposed into a set of independent tasks.
- The SeparableDependencies pattern can be used when dependencies between tasks can be pulled outside the concurrent execution by replicating data prior to the concurrent execution and then combining the replicated data afterwards.
- The GeometricDecomposition pattern is used when the problem space can be decomposed into discrete subspaces, this decomposition is used to drive the design of the parallel algorithm, and data must be exchanged between subspaces.

The concurrency in parallel programs introduces potentially nondeterministic behavior and the possibility of race conditions. Correctness concerns thus play a large role in parallel programming and are addressed by describing constraints on the problem and implementation. The goal is to provide rules that, if followed, will preclude concurrency errors. These constraints are typically described first in informal but precise language; in some cases this informal discussion is followed by a more formal and detailed discussion including references to supporting theory.

2 The EmbarrassinglyParallel Pattern

2.1 Pattern description

Intent:

- This pattern is used to describe concurrent execution by a collection of independent tasks. Parallel Algorithms that use this pattern are called *embarrassingly parallel* because once the tasks have been defined the potential concurrency is obvious.

Also Known As:

- Master-Worker.
- Task Queue.

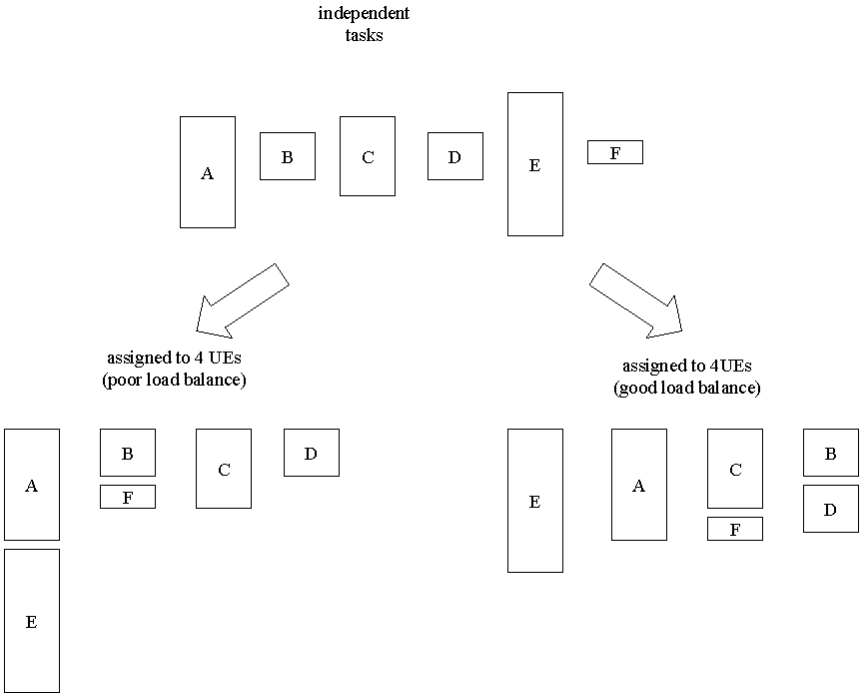
Motivation:

Consider an algorithm that can be decomposed into many independent tasks[1]. Such an algorithm, often called an “embarrassingly parallel” algorithm, contains obvious concurrency that is trivial to exploit once

[1] We use *task* to mean a sequence of operations that together make up some logical part of an algorithm or program. The FindingConcurrency patterns help the programmer decompose a problem into tasks that can execute concurrently; the AlgorithmStructure patterns show how to organize the tasks into a parallel program design.

these independent tasks have been defined, because of the independence of the tasks. Nevertheless, while the source of the concurrency is often obvious, taking advantage of it in a way that makes for efficient execution can be difficult.

The EmbarrassinglyParallel pattern shows how to organize such a collection of tasks so they execute efficiently. The challenge is to organize the computation so that all units of execution[2] finish their work at about the same time — that is, so that the computational load is balanced among processors. The following figure illustrates the problem.



This pattern automatically and dynamically balances the load as necessary. With this pattern, faster or less-loaded UEs automatically do more work. When the amount of work required for each task cannot be predicted ahead of time, this pattern produces a statistically optimal solution.

Examples of this pattern include the following:

- Vector addition (considering the addition of each pair of elements as a separate task).
- Ray-tracing codes such as the medical-imaging example described in the DecompositionStrategy pattern[3]. Here the computation associated with each "ray" becomes a separate task.
- Database searches in which the problem is to search for an item meeting specified criteria in a database that can be partitioned into subspaces that can be searched concurrently. Here the searches of the subspaces are the independent tasks.
- Branch-and-bound computations, in which the problem is solved by repeatedly removing a solution space from a list of such spaces, examining it, and either declaring it a solution, discarding it, or dividing it into smaller solution spaces that are then added to the list of spaces to examine. Such computations can be parallelized using this pattern by making each "examine and process a solution space" step a separate task.

[2] We use *unit of execution* (or UE) as a generic term for one of a collection of concurrently-executing entities (e.g., processes or threads).
[3] DecompositionStrategy is a pattern in our FindingConcurrency design space.

As these examples illustrate, this pattern allows for a fair amount of variation: The tasks can all be roughly equal in size, or they can vary in size. Also, for some problems (the database search, for example), it may be possible to solve the problem without executing all the tasks. Finally, for some problems (branch-and-bound computations, for example), new tasks may be created during execution of other tasks.

Observe that although frequently the source of the concurrency is obvious (hence the name of the pattern), this pattern also applies when the source of the concurrency requires some insight to discover; the distinguishing characteristic of problems using this pattern is the complete independence of the tasks.

More formally, the EmbarrassinglyParallel pattern is applicable when what we want to compute is a *solution(P)* such that

solution(P) =
f(subsolution(P, 0),
subsolution(P, 1), ...,
subsolution(P, N-1))

such that for *i* and *j* different, *subsolution(P, i)* does not depend on *subsolution(P, j)*. That is, the original problem can be decomposed into a number of *independent* subproblems such that we can solve the whole problem by solving all of the subproblems and then combining the results. We could code a sequential solution thus:

```
Problem P;
Solution subsolutions[N];
Solution solution;
for (i = 0; i < N; i++) {
    subsolutions[i] = compute_subsolution(P, i);
}
solution = compute_f(subsolutions);
```

If function `compute_subsolution` modifies only local variables, it is straightforward to show that the sequential composition implied by the `for` loop in the preceding program can be replaced by any combination of sequential and parallel composition without affecting the result. That is, we can partition the iterations of this loop among available UEs in whatever way we choose, so long as each is executed exactly once.

This is the EmbarrassinglyParallel pattern in its simplest form — all the subproblems are defined before computation begins, and each subsolution is saved in a distinct variable (array element), so the computation of the subsolutions is completely independent. These computations of subsolutions then become the independent tasks of the pattern as described earlier.

There are also some variations on this basic theme:

- **Subsolutions accumulated in a shared data structure.** One such variation differs from the simple form in that it accumulates subsolutions in a shared data structure (a set, for example, or a running sum). Computation of subsolutions is no longer completely independent (since access to the shared data structure must be synchronized), but concurrency is still possible if the order in which subsolutions are added to the shared data structure does not affect the result.

- **Termination condition other than "all tasks complete".** In the simple form of the pattern, all tasks must be completed before the problem can be regarded as solved, so we can think of the parallel algorithm as having the termination condition "all tasks complete". For some problems, however, it may be possible to obtain an overall solution without solving all the subproblems. For example, if the whole problem consists of determining whether a large search space contains at least one item meeting given search criteria, and each subproblem consists of searching a subspace (where the union of the subspaces is the whole space), then the computation can stop as soon as any subspace is found to contain an item meeting the search criteria. As in the simple form of the pattern, each computation of a subsolution becomes a task, but now the termination condition is something other than "all tasks completed". This can also be made to work, although care must be

taken to either ensure that the desired termination condition will actually occur or to make provision for the case in which all tasks are completed without reaching the desired condition.

- **Not all subproblems known initially**. A final and more complicated variation differs in that not all subproblems are known initially; that is, some subproblems are generated during solution of other subproblems. Again, each computation of a subsolution becomes a task, but now new tasks can be created "on the fly". This imposes additional requirements on the part of the program that keeps track of the subproblems and which of them have been solved, but these requirements can be met without too much trouble, for example by using a thread-safe shared task queue. The trickier problem is ensuring that the desired termination condition ("all tasks completed" or something else) will eventually be met.

What all of these variations have in common, however, is that they meet the pattern's key restriction: It must be possible to solve the subproblems into which we partition the original problem *independently*. Also, if the subsolution results are to be collected into a shared data structure, it must be the case that the order in which subsolutions are placed in this data structure does not affect the result of the computation.

Applicability:

Use the EmbarrassinglyParallel pattern when:

- The problem consists of tasks that are known to be independent; that is, there are no data dependencies between tasks (aside from those described in "Subsolutions accumulated in a shared data structure" above).

This pattern can be particularly effective when:

- The startup cost for initiating a task is much less than the cost of the task itself.
- The number of tasks is much greater than the number of processors to be used in the parallel computation.
- The effort required for each task or the processing performance of the processors varies unpredictably. This unpredictability makes it very difficult to produce an optimal static work distribution.

Structure:

Implementations of this pattern include the following key elements:

- A mechanism to define a set of tasks and schedule their execution onto a set of UEs.
- A mechanism to detect completion of the tasks and terminate the computation.

Usage:

This pattern is typically used to provide high-level structure for an application; that is, the application is typically structured as an instance of this pattern. It can also be used in the context of a simple sequential control structure such as sequential composition, if-then-else, or a loop construct. An example is given in [Massingill99], where the program as a whole is a simple loop whose body contains an instance of this pattern.

Consequences:

The EmbarrassinglyParallel pattern has some powerful benefits, but also a significant restriction.

- Parallel programs that use this pattern are among the simplest of all parallel programs. If the independent tasks correspond to individual loop iterations and these iterations do not share data dependencies, parallelization can be easily implemented with a parallel loop directive.

- With some care on the part of the programmer, it is possible to implement programs with this pattern that automatically and dynamically adjust the load between units of execution. This makes the EmbarrassinglyParallel pattern popular for programs designed to run on parallel computers built from networks of workstations.

- This pattern is particularly valuable when the effort required for each task varies significantly and unpredictably. It also works particularly well on heterogeneous networks, since faster or less-loaded processors naturally take on more of the work.

- The downside, of course, is that the whole pattern breaks down when the tasks need to interact during their computation. This limits the number of applications where this pattern can be used.

Implementation:

There are many ways to implement this pattern. If all the tasks are of the same size, all are known *a priori*, and all must be completed (the simplest form of the pattern), the pattern can be implemented by simply dividing the tasks among units of execution using a parallel loop directive. Otherwise, it is common to collect the tasks into a queue (the *task queue*) shared among UEs. This task queue can then be implemented using the SharedQueue[4] pattern. The task queue, however, can also be represented by a simpler structure such as a shared counter.

Key elements.

Defining tasks and scheduling their execution.

A set of tasks is represented and scheduled for execution on multiple units of execution (UEs). Frequently the tasks correspond to iterations of a loop. In this case we implement this pattern by splitting the loop between multiple UEs. The key to making algorithms based on this pattern run well is to schedule their execution so the load is balanced between the UEs. The schedule can be:

- **Static.** In this case the distribution of iterations among the UEs is determined once, at the start of the computation. This might be an effective strategy when the tasks have a known amount of computation and the UEs are running on systems with a well-known and stable load. In other words, a static schedule works when you can statically determine how many iterations to assign to each UE in order to achieve a balanced load. Common options are to use a fixed interleaving of tasks between UEs, or a blocked distribution in which blocks of tasks are defined and distributed, one to each UE.

- **Dynamic.** Here the distribution of iterations varies between UEs as the computation proceeds. This strategy is used when the effort associated with each task is unpredictable or when the available load that can be supported by each UE is unknown and potentially changing. The most common approach used for dynamic load balancing is to define a task queue to be used by all the UEs; when a UE completes its current task and is therefore ready to process more work, it removes a task from the task queue. Faster UEs or those receiving lighter-weight tasks will go to the queue more often and automatically grab more tasks.

Implementation techniques include parallel loops and master-worker and SPMD[5] versions of a task-queue approach.

Parallel loop.

[4] SharedQueue is a pattern in our SupportingStructures design space.
[5] Single-program, multiple-data.

If the computation fits the simplest form of the pattern — all tasks the same size, all known *a priori*, and all required to be completed — they can be scheduled by simply setting up a parallel loop that divides them equally (or as equally as possible) among the available units of execution.

Master-Worker or SPMD.

If the computation does not fit the simplest form of the pattern, the most common implementation involves some form of a task queue. Frequently this is done using two types of processes, *master* and *worker*. There is only one master process; it manages the computation by:

- Setting up or otherwise managing the workers.
- Creating and managing a collection of tasks (the task queue).
- Consuming results.

There can be many worker processes; each contains some type of loop that repeatedly:

- Removes the task at the head of the queue.
- Carries out the indicated computation.
- Returns the result to the master.

Frequently the master and worker processes form an instance of the ForkJoin[6] pattern, with the master process forking off a number of workers and waiting for them to complete.

A common variation is to use an SPMD program with a global counter to implement the task queue. This form of the pattern does not require an explicit master.

Detecting completion and terminating.

Termination can be implemented in a number of ways.

If the program is structured using the ForkJoin pattern, the workers can continue until the termination condition is reached, checking for an empty task queue (if the termination condition is "all tasks completed") or for some other desired condition. As each worker detects the appropriate condition, it terminates; when all have terminated, the master continues with any final combining of results generated by the individual tasks.

Another approach is for the master or a worker to check for the desired termination condition and, when it is detected, create a "poison pill", a special task that tells all the other workers to terminate.

Correctness considerations.

The keys to exploiting available concurrency while maintaining program correctness (for the problem in its simplest form) are as follows.

- **Solve subproblems independently.** Computing the solution to one subproblem must not interfere with computing the solution to another subproblem. This can be guaranteed if the code that solves each subproblem does not modify any variables shared between units of execution (UEs).
- **Solve each subproblem exactly once.** This is almost trivially guaranteed if static scheduling is used (i.e., if the tasks are scheduled via a parallel loop). It is also easily guaranteed if the parallel algorithm is structured as follows:

[6] ForkJoin is a pattern in our SupportingStructures design space.

- A task queue is created as an instance of a thread-safe shared data structure such as SharedQueue, with one entry representing each task.

- A collection of UEs execute concurrently; each repeatedly removes a task from the queue and solves the corresponding subproblem.

- When the queue is empty and each UE finishes the task it is currently working on, all the subsolutions have been computed, and the algorithm can proceed to the next step, combining them. (This also means that if a UE finishes a task and finds the task queue empty, it knows that there is no more work for it to do, and it can take appropriate action — terminating if there is a master UE that will take care of any combining of subsolutions, for example.)

- **Correctly save subsolutions.** This is trivial if each subsolution is saved in a distinct variable, since there is then no possibility that the saving of one subsolution will affect subsolutions computed and saved by other tasks.

- **Correctly combine subsolutions.** This can be guaranteed by ensuring that the code to combine subsolutions does not begin execution until all subsolutions have been computed as discussed above.

The variations mentioned earlier impose additional requirements:

- **Subsolutions accumulated in a shared data structure.** If the subsolutions are to be collected into a shared data structure, then the implementation must guarantee that concurrent access does not damage the shared data structure. This can be ensured by implementing the shared data structure as an instance of a "thread-safe" pattern.

- **Termination condition other than "all tasks complete".** Then the implementation must guarantee that each subsolution is computed at most once (easily done by using a task queue as described earlier) and that the computation detects the desired termination condition and terminates when it is found. This is more difficult but still possible.

- **Not all subproblems known initially.** Then the implementation must guarantee that each subsolution is computed exactly once, or at most once (depending on the desired termination condition.) Also, the program designer must ensure that the desired termination detection will eventually be reached. For example, if the termination condition is "all tasks completed", then the pool generated must be finite, and each individual task must terminate. Again, a task queue as described earlier solves some of the problems; it will be safe for worker UEs to add as well as remove elements. Detecting termination of the computation is more difficult, however. It is not necessarily the case that when a "worker" finishes a task and finds the task queue empty that there is no more work to do — another worker could generate a new task. One must therefore ensure that the task queue is empty *and all* workers are finished. Further, in systems based on asynchronous message passing, one must also ensure that there are no messages in transit that could, on their arrival, create a new task. There are many known algorithms that solve this problem. One that is useful in this context is described in [Dijkstra80]. Here tasks conceptually form a tree, where the root is the master task, and the children of a task are the tasks it generated. When a task and all its children have terminated, it notifies its parent that it has terminated. When all the children of the root have terminated, the computation has terminated. This of course requires children to keep track of their parents and to notify them when they are finished. Parents must also keep track of the number of active children (the number created minus the number that have terminated). Additional algorithms for termination detection are described in [Bertsekas89].

Efficiency considerations.

- If all tasks are roughly the same length and their number is known *a priori*, static scheduling (usually performed using a parallel loop directive) is likely to be more efficient than dynamic scheduling.

- If a task queue is used, put the longer tasks at the beginning of the queue if possible. This ensures that there will be work to overlap with their computation.

Examples:

Vector addition.

Consider a simple vector addition, say $C = A + B$. As discussed earlier, we can consider each element addition ($C_i = A_i + B_i$) as a separate task and parallelize this computation in the form of a parallel loop:

- See the section "Vector Addition" in the examples document (Section 2.2 of this paper).

Varying-length tasks.

Consider a problem consisting of N independent tasks. Assume we can map each task onto a sequence of simple integers ranging from 0 to $N-1$. Further assume that the effort required by each task varies considerably and is unpredictable. Several implementations are possible, including:

- A master-worker implementation using a task queue. See the section "Varying-Length Tasks, Master-Worker Implementation" in the examples document (Section 2.2 of this paper).

- An SPMD implementation using a task queue. See the section "Varying-Length Tasks, SPMD Implementation" in the examples document (Section 2.2 of this paper).

Optimization.

See [Massingill99] for an extended example using this pattern.

Known Uses:

There are many application areas in which this pattern is useful. Many ray-tracing codes use some form of partitioning with individual tasks corresponding to scan lines in the final image [Bjornson91a]. Applications coded with the Linda coordination language are another rich source of examples of this pattern [Bjornson91b].

Parallel computational chemistry applications also make heavy use of this pattern. In the quantum chemistry code GAMESS, the loops over two electron integrals are parallelized with the TCGMSG task queue mechanism mentioned earlier. An early version of the Distance Geometry code, DGEOM, was parallelized with the Master-Worker form of the EmbarrassinglyParallel pattern. These examples are discussed in [Mattson95].

Related Patterns:

The SeparableDependencies[7] pattern is closely related to the EmbarrassinglyParallel pattern. To see this relation, think of the SeparableDependencies pattern in terms of a three-phase approach to the parallel algorithm. In the first phase, dependencies are pulled outside a set of tasks, usually by replicating shared data and converting it into task-local data. In the second phase, the tasks are run concurrently as completely independent tasks. In the final phase, the task-local data is recombined (reduced) back into the original shared data structure.

[7] Described in Section 3.

The middle phase of the SeparableDependencies pattern is an instance of the EmbarrassinglyParallel pattern. That is, you can think of the SeparableDependencies pattern as a technique for converting problems into embarrassingly parallel problems. This technique can be used in certain cases with most of the other patterns in our pattern language. The key is that the dependencies can be pulled outside of the concurrent execution of tasks. If this isolation can be done, then the execution of the tasks can be handled with the EmbarrassinglyParallel pattern.

Many instances of the GeometricDecomposition[8] pattern (for example, "mesh computations" in which new values are computed for each point in a grid based on data from nearby points) can be similarly viewed as two-phase computations, where the first phase consists of exchanging boundary information among UEs and the second phase is an instance of the EmbarrassinglyParallel pattern in which each UE computes new values for the points it "owns".

It is also worthwhile to note that some problems in which the concurrency is based on a geometric data decomposition are, despite the name, not instances of the GeometricDecomposition pattern but instances of EmbarrassinglyParallel. An example is a variant of the vector addition example presented earlier, in which the vector is partitioned into "chunks", with computation for each "chunk" treated as a separate task.

2.2 Supporting examples

Vector Addition:

The following code uses an OpenMP parallel loop directive to perform vector addition.

```
!$OMP PARALLEL DO
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END PARALLEL DO
```

Varying-Length Tasks, Master-Worker Implementation:

The following code uses a task queue and master-worker approach to solving the stated problem. We implement the task queue as an instance of the SharedQueue pattern.

The master process, shown below, initializes the task queue, representing each task by an integer. It then uses the ForkJoin pattern to create the worker processes or threads and wait for them to complete. When they have completed, it consumes the results.

```
#define Ntasks 500       /* Number of tasks    */
#define Nworkers 5       /* Number of workers   */

SharedQueue task_queue;     /* task queue      */
Results Global_results[Ntasks]; /* array to hold results */

void master()
{
    void Worker();

    // Create and initialize shared data structures
    task_queue = new SharedQueue();
    for (int i = 0; i < N; i++)
        enqueue(&task_queue, i);

    // Create Nworkers threads executing function Worker()
    ForkJoin (Nworkers, Worker);

    Consume_the_results (Ntasks);
}
```

[8] Described in Section 4.

The worker process, shown below, loops until the task queue is empty. Every time through the loop, it grabs the next task, does the indicated work (storing the results into a global results array). When the task queue is empty, the worker terminates.

```
void Worker()
{
   int i;
   Result res;

   While (!empty(task_queue) {
       i = dequeue(task_queue);
       res = do_lots_of_work(i);
       Global_results[i] = res;
   }
}
```

Note that we ensure safe access to the key shared variable (the task queue) by implementing it using patterns from the SupportingStructures space. Note also that the overall organization of the master process is an instance of the ForkJoin pattern.

Varying-Length Tasks, SPMD Implementation:

As an example of implementing this pattern without a master process, consider the following sample code using the TCGMSG message-passing library (described in [Harrison91]). The library has a function called `NEXTVAL` that implements a global counter. An SPMD program could use this construct to create a task-queue program as shown below.

```
While (itask = NEXTVAL() < Number_of_tasks){
    do_lots_of_work(itask);
}
```

3 The SeparableDependencies Pattern

3.1 Pattern description

Intent:

This pattern is used for task-based decompositions in which the dependencies between tasks can be eliminated as follows: Necessary global data is replicated and (partial) results are stored in local data structures. Global results are then obtained by reducing (combining) results from the individual tasks.

Also Known As:

- Replicated data, loop splitting [Mattson96].

Motivation:

In general, task-based algorithms present two distinct challenges to the software designer: allocating the tasks among the processors so the computational load is evenly distributed; and managing the dependencies between tasks so that if multiple tasks update the same data structure, these updates do not interfere with each other.

This pattern represents an important class of problems in which these two issues can be separated. In these problems, dependencies can be pulled outside the set of concurrent tasks, allowing the tasks to proceed independently.

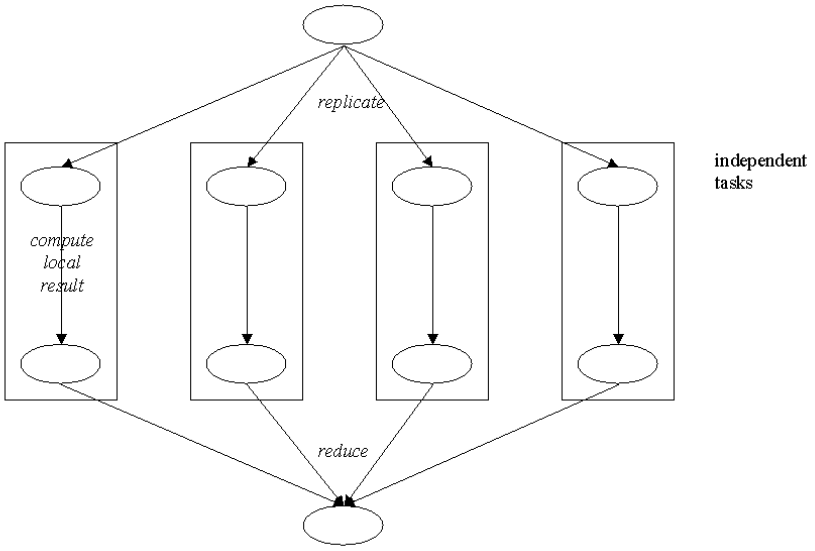
Consider an example, the classic N-body problem: A system contains N bodies that move in space, each exerting distance-dependent forces on each of the other N-1 bodies. The problem is to calculate the motion of the bodies. For each instant in time, each body has a position, a velocity, and a force vector. For each

time instant, a sequential solution calculates the force vector incident on each body by summing the force contributions from each of the other N-1 bodies and then computing the new position and velocity of each particle using its force vector.

One way to parallelize the problem is to decompose the computation performed at each time instant into tasks such that each task is responsible for computing the position and velocity of a subset of the bodies along with the contribution of the forces from that subset on the rest of the system. These tasks are not independent, since each task needs to read the locations of the other bodies, and each task must update the force vectors of all the other bodies.

This problem has two features that can be exploited. First, during the calculation for a particular time instant, the location of each body is first read by the other tasks and then modified by only a single task; it is not read by any other task after it has been written. Therefore, dependencies between tasks involving the location data can be eliminated by replicating this data in all the tasks. Second, since the force vectors are the sums of forces due to each body, they can be computed in two stages as follows: Each task can compute a partial sum, placing the result in a task-local variable. Once all the tasks have computed their partial sums, these partial sums can then be summed (reduced) to give the desired local force vectors. As a result, all the dependencies between the tasks during the concurrent execution have been "pulled out" of the concurrent part of the computation.

The techniques described in the EmbarrassinglyParallel[9] pattern can be applied to the now-independent tasks. Some dependencies between tasks can be eliminated by replacing global data structures with copies local to each UE. (In a shared-memory environment, it is possible for all tasks that do not modify the global data structure to share a single copy.) Others may be eliminated by writing results to a local object and then, in a logically separate step, reducing (merging) the local objects into a single object. In essence, the dependencies between tasks are eliminated from the concurrent part of the computation, thus making the construction of a parallel algorithm much simpler. The following figure illustrates the central idea of this pattern.



Applicability:

This pattern can be used when:

- The problem is represented as a collection of concurrent tasks.

[9] Described in Section 2.

- Dependencies between the tasks satisfy the restriction that, for any shared object, one of the following holds:

  - Only one task modifies the object, and other tasks need only its initial value. The object can thus be replicated. (Observe that in a shared-memory environment all tasks that do not modify the object can share a single copy of the object.)

  - The object's final value can be formulated as

    $result = combine(v_0, v_1, ..., v_{M-1})$

    where the computation of v_i is computed independently by task i, possibly after replicating global data. The object can thus be treated as a "reduction variable" (using the term "reduction" somewhat more broadly than usual), with each task computing a local partial result and these partial results being subsequently combined into a final global result.

The pattern is especially effective when:

- $result = (v_0 \cdot v_1 \cdot ... \cdot v_{M-1})$ where $\cdot$ is an associative operator (i.e., the result can be computed as a reduction, using the term in its usual sense). This fact can be exploited to improve performance in the implementation of the combining (reduction) step, since the order in which the operations are performed does not matter. The pattern is even more effective when the reduction operator $\cdot$ is both associative and commutative.

Structure:

Implementations of this pattern include the following key elements:

- A mechanism to define a set of tasks and schedule their execution onto a set of units of execution (UEs).

- Definition and update of a local data structure, possibly containing replicated data.

- Combination (reduction) of the many local objects into a single object.

Usage:

When this pattern is used in a parallel algorithm, it usually drives the top-level organization of the parallel algorithm. It frequently appears, as in the N-body example, as the body of a compute-intensive loop within a program. Often, but not necessarily, the reduction step uses one of a small set of standard commutative and associative reduction operators (+, *, logical **and**, logical **or**, bitwise **and**, bitwise **or**, min, max).

Consequences:

This pattern shows up in many forms.

- In most cases, the pattern makes the processing of the concurrent tasks very similar to the embarrassingly parallel case. Hence, the automatic dynamic load balancing that is the hallmark of a good embarrassingly parallel algorithm applies to SeparableDependencies algorithms as well.

- Although the concurrent tasks are independent, the reduction step itself involves intertask communication and synchronization. This may be very expensive, especially when the number of UEs is large. Thus, the challenge in using this pattern is in ensuring that the cost of the reduction step does not overwhelm the computation done by the tasks. Properties of the reduction operator, such as associativity and commutativity, can be exploited to obtain more efficient implementations of the reduction step. Fortunately, because of the importance of this pattern, most programming environments supply high-quality implementations of reduction using the standard reduction operators. Some also allow user-defined reduction operators.

- In distributed-memory environments, objects that are read or modified by multiple tasks are replicated in each task. If the size of the object is very large, this can result in very large memory requirements.

- When using this pattern with floating-point merge operations, it is important to remember that floating-point arithmetic is not associative, and thus the final results can depend on the details of the implementation of the reduction operation. With a well-behaved algorithm, the variation in the final results due to changes in the order of the merge operations is not significant. Usually, if this variation is significant, the underlying algorithm is not adequate for the job at hand. The exception to this rule is the case in which a careful numerical analysis has defined a preferred order for the reduction operation. In this case, the programmer may need to use a more expensive order-preserving reduction operation.

Implementation:

Before describing the implementation of this pattern, we remark that it is useful to remember that what we are essentially doing is isolating the dependencies so the concurrency becomes embarrassingly parallel.

Key elements.

Defining the tasks and scheduling their execution.

A set of tasks is represented and scheduled for execution on multiple units of execution (UEs). Usually, the tasks correspond to iterations of a loop. In this case we implement this pattern by splitting the loop between multiple UEs. The key to making algorithms based on this pattern run well is to schedule their execution so the load is balanced between the UEs. The approaches used for this scheduling are the same as those described in the Implementation section of the EmbarrassinglyParallel pattern.

Defining and updating a local data structure.

As discussed previously, the tasks cooperatively update one or more objects, and this pattern is not applicable if values written to such an object by one task are subsequently read by another. Once these objects have been identified, local copies must be created and initialized. It is simplest to think of each task as having its own copy of these shared objects, with their values initialized to the objects' initial values during task initialization and updated as the task performs its work. Often, the local update will in fact be a "local reduction". For example, a task in the N-body problem may handle several bodies; within the task, the local force vector updates would in fact be a reduction of the force contributions for the subset of the bodies handled by that task.

In practice it may be possible to reduce the number of physical copies required, particularly in shared-memory environments, since tasks that do not update a particular object can share a single copy of the object. In a shared-memory environment, an object that is read by multiple tasks but updated by only one need only be duplicated, with one copy maintaining its initial value (to be used by all tasks that do not modify the object) and one copy being updated by the single task that modifies the object.

Also, in practice it is usually enough to have one local copy per unit of execution (UE) rather than one local copy per task.

Combining (reducing) local objects into a single object.

The reduction step occurs after all the tasks have finished updating their local copies of the data structure. The first step is thus to determine that all partial results have been computed. In its most general form, this is almost the same as detecting termination in the EmbarrassinglyParallel pattern. The difference is that the tasks should have finished the update of the global data structure but may not have actually terminated, since they may still need to participate in the reduction operation.

The reduction step itself performs the calculation indicated earlier (*result = combine(v_0 ,v_1,...,v_{M-1})*). In most cases the *combine* function can be expressed in the form *(v_0 · v_1 · ... · v_{M-1})*, where · is a binary operator. Computations of this form are sufficiently common in parallel algorithms that they are discussed separately in the Reduction[10] pattern; many programming environments also supply high-quality implementations of this pattern.

Examples:

Matrix-vector multiplication.

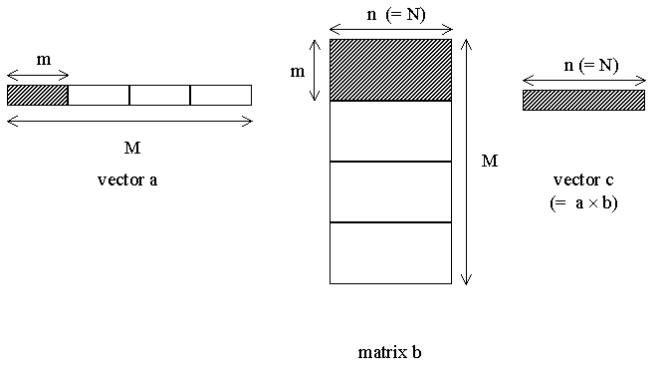
This example, taken from the MPI reference manual [Snir96], uses this pattern to compute the product of a vector and a matrix. A simple sequential program to accomplish this result is as follows:

```
SUBROUTINE BLAS2(M,N,a,b,c)
REAL a(M), b(M,N) !input vector and matrix
REAL c(N)   !result
INTEGER M,N,i,j

DO j = 1,N
    c(j) = 0.0
    DO i = 1,M
        c(j) = c(j) + a(i)*b(i,j)
    END DO
END DO

RETURN
```

Each element of the result vector is a reduction of M partial sums, so the SeparableDependencies pattern applies — we can calculate each element by computing and then combining partial sums, with the decomposition based on partitioning the input vector and matrix as shown in the following figure (shaded areas indicate data for one task).



To calculate the matrix-vector product, each UE computes a local sum (the product of its section of vector `a` and its section of matrix `b`); final values for the elements of product vector `c` are obtained by summing these local sums.

- See the section "Matrix-Vector Multiplication, MPI Implementation" in the examples document (Section 3.2 of this paper) for an implementation using MPI.

Numerical Integration.

This example performs numerical integration using the trapezoid rule. This example is almost trivial, but it effectively addresses most of the key issues raised by this pattern. The goal here is to compute pi by

[10] Reduction is a pattern in our SupportingStructures design space.

integrating the function $1/4x^2$ over the interval from 0 to 1. But what is important here is not the mathematics but the pattern, which consists of computing a sum of individual contributions and which is parallelized by computing a number of partial sums and then combining them to get the global result.

First, here's a simple sequential version of a program that solves this problem:

```
#include <stdio.h>
static long num_steps = 100000;
double step;
void main ()
{
    int i;
    double step, x, pi, sum = 0.0;

    step = 1.0/(double)(num_steps);

    for( i = 1; i < num_steps; i++){
        x = (i-0.5) * step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = sum * step;
    printf(" pi is %f \n",pi);
}
```

We present two parallel implementations.

- See the section "Numerical Integration, OpenMP Implementation" in the examples document (Section 3.2 of this paper) for an implementation using OpenMP.

- See the section "Numerical Integration, MPI Implementation" in the examples document (Section 3.2 of this paper) for an implementation using MPI.

Known Uses:

This pattern has been used extensively in computational chemistry applications. The parallel computational chemistry book [Mattson95] includes several chapters that discuss the use of this pattern. In particular:

- In chapter 2, Windus, Schmidt, and Gordon use this pattern to parallelize the Fock matrix computation in the GAMESS ab initio quantum chemistry program.

- In chapter 8, Baldridge uses this pattern for the parallel update of Fock matrix elements in the MOPAC semi-empirical quantum chemistry code.

- In chapter 9, Plimpton and Hendrickson use this pattern in their replicated-data algorithm for molecular dynamics simulations. The algorithm looks quite similar to the simple example we present in this pattern, except that the summations occur into an array rather than into a single scalar.

- In chapter 10, Mattson and Ravishanker describe the use of this pattern in molecular dynamics for modest parallelism on workstation clusters. They describe an owner-computes filter to handle the scheduling of the tasks onto UEs.

Data parallel algorithms make heavy use of the SeparableDependencies pattern. While traditionally intended for SIMD computers, data parallel algorithms can be implemented on other architectures as well. Several data parallel algorithms, including some that would be suitable for reduction with associative operators, sorting algorithms, and algorithms on linked lists are described in [Hillis86].

Related Patterns:

This pattern converts into the EmbarrassinglyParallel pattern when the dependencies are removed.

If the dependencies involve reads as well as writes, the pattern is transformed into the ProtectedDependencies[11] pattern.

3.2 Supporting examples

Matrix-Vector Multiplication, MPI Implementation:

The following code (to be executed, SPMD-style, by each UE) accomplishes the desired result using MPI. Code to distribute the matrix among UEs and initiate the computation is not shown.

```
SUBROUTINE PAR_BLAS2(m,n,a,b,c,comm)
REAL a(m), b(m,n)  !local slice of array and matrix
REAL c(n)      !result
REAL sum(n)     !local vector for partial results
INTEGER m,n,comm,i,j,ierr

!calculate local sum
DO j = 1,n
    sum(j) = 0.0
    DO i = 1,m
        sum(j) = sum(j) + a(i)*b(i,j)
    END DO
END DO

!calculate global sum, result or reduction placed in c at all nodes
CALL MPI_ALLREDUCE(sum,c,n,MPI_REAL,MPI_SUM,comm,ierr)

RETURN
```

The `MPI_ALLREDUCE` primitive performs the necessary synchronization and communication to compute the desired result behind the scenes.

Numerical Integration, OpenMP Implementation:

With OpenMP, we use fork-join parallelism to break up the loop in the program for execution on multiple threads. In this case, the number of threads is set to `NUM_THREADS` with a call to `omp_set_num_threads()`. We then parallelize the program using a single "parallel for" pragma. The reduction is handled using the standard reduction clause from OpenMP.

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 2
static long num_steps = 100000;
double step;
void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    omp_set_num_threads (NUM_THREADS);

    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf(" pi is %f \n",pi);
}
```

In this case we used the default schedule, which is implementation dependent. The performance of this simple example is not sensitive to the particular schedule used. If the computation within a loop iteration

[11] ProtectedDependencies is a pattern in our AlgorithmStructure design space.

varied unpredictably, we would want to use a dynamic schedule, which would be selected by using the schedule (dynamic) clause on the omp pragma.

Numerical Integration, MPI Implementation:

We can also implement this algorithm as an SPMD program using MPI. The approach used here is very common. One copy of the program is run on each UE. The program initializes MPI and then queries the system to find a processor ID (`MPI_Comm_rank`) and the number of UEs (`MPI_Comm_size`). The loop is scheduled in an interleaved manner by running the loop iterations from "id" to the number of integration steps (`num_steps`) with an increment equal to the number of UEs.

```
#include <stdio.h>
#include "mpi.h"
static long num_steps = 100000;
double step;
void main (int argc, char** argv)
{
    int i, id, num_procs;
    double x, pi, sum = 0.0;

    MPI_Init(&argc, &argv);
    step = 1.0/(double) num_steps;

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    for (i=(id+1);i<= num_steps; i+=num_procs){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    MPI_Reduce(&sum, &pi, 1, MPI_FLOAT, MPI_SUM, 0 MPI_COMM_WORLD);
    if(id == 0){
        pi *= step;
        printf(" pi is %f \n",pi);
    }
}
```

The reduction is handled by an MPI library routine, `MPI_Reduce()`. The form used here combines the local results into a single value on the node of rank 0; this node then handles the single output operation.

4 The GeometricDecomposition Pattern

4.1 Pattern description

Intent:

This pattern is used when (1) the concurrency is based on parallel updates of chunks of a decomposed data structure, and (2) the update of each chunk requires data from other chunks.

Also Known As:

- Domain decomposition.
- Coarse-grained data parallelism.

Motivation:

There are many important problems that are best understood as a sequence of operations on a core data structure. There may be other work in the computation, but if you understand how the core data structures are updated, you have an effective understanding of the full computation. For these types of problems, it is

often the case that the best way to represent the concurrency is in terms of decompositions of these core data structures.

The way these data structures are built is fundamental to the algorithm. If the data structure is recursive, any analysis of the concurrency must take this recursion into account. For arrays and other linear data structures, however, we can often reduce the problem to potentially concurrent components by decomposing the data structure into contiguous substructures, in a manner analogous to dividing a geometric region into subregions — hence the name GeometricDecomposition. For arrays, this decomposition is along one or more dimensions, and the resulting subarrays are usually called blocks. We will use the term "chunks" for the substructures or subregions, to allow for the possibility of more general data structures such as graphs. Each element of the global data structure (each element of the array, for example) then corresponds to exactly one element of the distributed data structure, identified by a unique combination of chunk ID and local position.

This decomposition of data into chunks then implies a decomposition of the update operation into tasks, where each task represents the update of one chunk, and the tasks execute concurrently. We consider two basic forms of update: (1) an update defined in terms of individual elements of the data structure (i.e., one that computes new values for each *point*) and (2) an update defined in terms of chunks (i.e., one that computes new values for each *chunk*).

If the computations are strictly local, i.e., all required information is within the chunk, the concurrency is "embarrassingly parallel" and the EmbarrassinglyParallel[12] pattern should be used. In many cases, however, the update requires information from points in other chunks (frequently from what we can call "neighboring chunks" — chunks containing data that was nearby in the original global data structure). In these cases, information must be shared between chunks in order to complete the update.

Motivating examples.

Before going further, it may help to briefly present two motivating examples: a mesh-computation program to solve a differential equation and a matrix-multiplication algorithm.

- **Mesh-computation program.** The first example illustrates the first class of problems represented by this pattern, those in which the computation is "by points". The problem is to solve a 1D differential equation representing heat diffusion:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

  The approach used is to discretize the problem space (representing U by a 1-dimensional array and computing values for a sequence of discrete time steps). We will output values for each time step as they are computed, so we need only save values for U for two time steps; we will call these arrays `uk` (U at the timestep k) and `ukp1` (U at timestep $k+1$). At each time step, we then need to compute for each point in array ukp1 the following:

```
ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
```

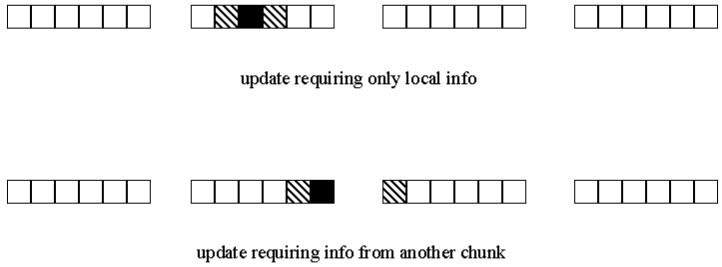
  Variables `dt` and `dx` represent the intervals between discrete time steps and between discrete points respectively. (We will not discuss the derivation of the above formula; it is not relevant to the parallelization aspects of the problem and so is outside the scope of our pattern language.)

  Observe that what is being computed above is a new value for variable `ukp1` at each point, based on data at that point and its left and right neighbors — an example of the "update defined in terms of individual elements of the data structure".

  We can create a parallel algorithm for this problem by decomposing the arrays `uk` and `ukp1` into contiguous subarrays (the chunks described earlier) and operating on these chunks concurrently (one task per chunk). We then have a situation in which some elements can be updated using only

[12] Described in Section 2.

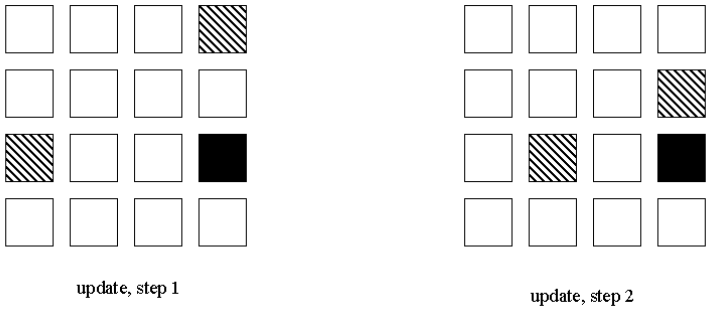
data from within the chunk, while others require data from neighboring chunks, as illustrated by the following figure (solid boxes indicate the element being updated, shaded boxes the elements containing needed data).



- **Matrix-multiplication program.** The second example, taken from [Fox88], illustrates the second class of problems, those in which the computation is "by chunks". The problem is to multiply two square matrices (i.e., compute $C = A \cdot B$), and the approach is to decompose the matrices into square blocks and operate on blocks rather than on individual elements. If we denote the (i,j)-th block of C by C^{ij}, then we can compute a value for this block in a manner analogous to the way in which we compute values for new elements in the standard definition of matrix multiplication:

$$C^{ij} = \sum_k A^{ik} \cdot B^{kj}$$

We can readily compute this using a loop like that used to compute each element in the standard matrix multiplication; at each step we compute the matrix product $A^{ik} \cdot B^{kj}$ and add it to the running matrix sum. This gives us a computation in the form described by our pattern -- one in which the algorithm is based on decomposing the data structure into chunks (square blocks here) and operating on those chunks concurrently. If we decompose all three matrices into square blocks (with each task "owning" corresponding blocks of A, B, and C), the following figure illustrates the updates at two representative steps (again solid boxes indicate the "chunk" being updated, and shaded boxes indicate the chunks containing data needed for the update).



These two examples illustrate the two basic categories of algorithms addressed by the GeometricDecomposition pattern. In both cases, the data structures (two 1D arrays in the first example, three 2D matrices in the second) are decomposed into contiguous subarrays as illustrated.

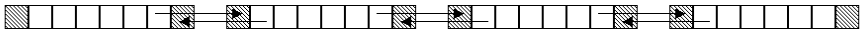
More about data structures.

Before turning our attention to how we can schedule the tasks implied by our data decomposition, we comment on one further characteristic of algorithms using this pattern: For various reasons including algorithmic simplicity and program efficiency (particularly for distributed memory), it is often useful to define for each chunk a data structure that provides space for both the chunk's data and for duplicates of

whatever non-local data is required to update the data within the chunk. For example, if the data structure is an array and the update is a grid operation (in which values at each point are updated using values from nearby points), it is common to surround the data structure for the block with a "ghost boundary" to contain duplicates of data at the boundaries of neighboring blocks.

Thus, each element of the global data structure can correspond to more than one element of the distributed data structure, but these multiple elements consist of one "primary" copy (that will be updated directly) associated with an "owner chunk" and zero or more "shadow" copies (that will be updated with values computed as part of the computation on the owner chunk).

In the case of our mesh-computation example above, each of the subarrays would be extended by one cell on each side. These extra cells would be used as shadow copies of the cells on the boundaries of the chunks. The following figure illustrates this scheme: The shaded cells are the shadow copies (with arrows pointing from their corresponding primary copies).



Updating the data structures.

Updating the data structure is then done by executing the corresponding tasks (each responsible for the update of one chunk of the data structures) concurrently. Recalling that the update for each chunk requires data from other chunks ("non-local data"), we see that at some point each task must obtain data from other tasks. This involves an exchange of information among tasks, which is often (but not always) carried out before beginning the computation of new values.

Such an exchange can be expensive: For a distributed-memory implementation, it may require not only message-passing but also packing the shared information into a message; for a shared-memory implementation it may require synchronization (with the consequent overhead) to ensure that the information is shared correctly. Sophisticated algorithms may schedule this exchange in a way that permits overlap of computation and communication.

Balancing the exchange of information between chunks and the update computation within each chunk is potentially difficult. The goal is to structure the concurrent algorithm so that the computation time dominates the overall running time of the program. Much of what is discussed in this pattern is driven by the need to effectively reach that goal.

Mapping the data decomposition to UEs.

The final step in designing a parallel algorithm for a problem that fits this pattern is deciding how to map the collection of tasks (each corresponding to the update of one chunk) to units of execution (UEs). Each UE can then be said to "own" a collection of chunks and the data they contain.

Observe that we thus have a two-tiered scheme for distributing data among UEs: partitioning the data into chunks, and then assigning these chunks to UEs. This scheme is flexible enough to represent a variety of popular schemes for distributing data among UEs:

- Block distributions (of which row distributions and column distributions are simply extreme cases) are readily represented by assigning one chunk to each UE.

- Cyclic and block-cyclic distributions are represented by making the individual chunks small (perhaps as small as a single row or column, or even a single element) and assigning more than one chunk to each UE, with the mapping of chunks to UEs done in a cyclic fashion. Cyclic distributions can be effective in situations where simpler block distributions would lead to poor load balance.

Applicability:

Use the GeometricDecomposition pattern when:

- Your problem requires updates of a large non-recursive data structure.
- The data structure (region) can be decomposed into chunks (subregions), such that updating each chunk can be done primarily with data from the same chunk.
- The amount of computation required for the update within each chunk is large enough to compensate for the cost of obtaining any data required from other chunks.

In many cases in which the GeometricDecomposition decomposition pattern is applicable, the data to share between UEs is restricted to the boundaries of the chunks associated with a UE. In other words, the amount of information to share between UEs scales with the "surface area" of the chunks. Since the computation scales with the number of points within a chunk, it scales as the "volume" of the regions. This "surface-to-volume effect" gives these algorithms attractive scalable execution behavior. By increasing the size of the problem for a fixed number of UEs, this surface-to-volume effect leads to favorable scaling behavior, making this class of algorithms very attractive for parallel computing.

More formally, this pattern is applicable when the computation involves:

- A potentially-large non-recursive data structure that either (1) consists of a collection of points (a "region") and a set of variables, such that each point is represented by a set of values for the variables (a simple example being a multi-dimensional array), or (2) can be decomposed into substructures (e.g., in the case of an array, partitioned into blocks) in a way that is algorithmically useful (e.g., the desired computation can be expressed in terms of operations on the blocks).
- A sequence of update operations on that data structure, where an update operation takes one of two forms: (1) assigning, for every point in the region, new values for one or more of the variables, or (2) assigning, for every substructure, new values for its variables. The update operation must be one that can be parallelized effectively, as described subsequently.

To see which operations can be parallelized effectively, we need to look at what data is read and written during the course of the update operation, focusing on two questions:

- *Locality considerations*: Is the data needed to update point *p* (or chunk *c*) local to *p* (*c*), or does it correspond to some other point (chunk)? The less data from other points (chunks) required for the update, the more likely it is that the parallelization will be effective.
- *Timing considerations*: Is the data needed to update point *p* (or chunk *c*) available at the beginning of the update operation, or will it be generated in the course of the update? Conversely, does the update for point *p* (chunk *c*) affect data that must be read during the update of other points (chunks)? If all the needed data is present at the beginning of the update operation, and if none of this data is modified during the course of the update, parallelization is easier and more likely to be efficient.

With respect to the latter question, the simplest situation is that the set of variables modified during the update is disjoint from the set of variables whose values must be read during the update (as is the case in the two motivating examples presented earlier). In this situation, it is not difficult to see that we can perform the updates in any order we choose, including concurrently; this is the basis for the correctness of this parallelization scheme. It is easy to see that in this situation we can obtain correct results by separating each task into two phases, a communication phase (in which tasks exchange data such that after the exchange each task has copies of any non-local data it will need) and a computation phase.

Structure:

Implementations of this pattern include the following key elements:

- A way of partitioning the global data structure into substructures or "chunks" (decomposition).
- A way of ensuring that each task has access to all the data it needs to perform the update operation for its chunk, including data in chunks corresponding to other tasks.
- A definition of the update operation, whether by points or by chunks.
- A way of assigning the chunks among UEs (distribution) — i.e., a way of scheduling the corresponding tasks.

Usage:

This pattern can be used to provide the high-level structure for an application (that is, the application is structured as an instance of this pattern). More typically, an application is structured as a sequential composition of instances of this pattern (and possibly other patterns such as EmbarrassinglyParallel[13] and SeparableDependencies[14]), as in the examples in the Examples section.

Consequences:

- The programmer must explicitly manage load balancing. When the data structures are uniform and the processing elements[15] (PEs) of the parallel system are homogeneous, this load balancing can be done once at the beginning of the computation. If the data structure is non-uniform and changing, the programmer will have to explicitly change the decomposition as the computation progresses in order to balance the load between PEs.
- The sizes of the chunks, or the mapping of chunks to UEs, must be such that the time needed to update the chunks owned by each UE is much greater than the time needed to exchange information among UEs. This requirement tends to favor using large chunks. Large chunks, however, make it more difficult to achieve good load balance. These competing forces can make it difficult to select an optimum size for a chunk.
- How the chunks are mapped onto UEs can also have a major impact on the efficiency of these problems. For example, consider a linear algebra problem in which elements of the matrix are successively eliminated as the computation proceeds. Early in the computation, all of the rows and columns of the matrix have numerous elements to work on, and decompositions based on assigning full rows or columns to UEs are effective. Later in the computation, however, rows or columns become sparse, the work per row or column becomes uneven, and the computational load becomes poorly balanced between UEs. The solution is to decompose the problem into many more chunks than there are UEs and to scatter them among the UEs (e.g., with a cyclic or block-cyclic distribution). Then as a chunk becomes sparse, there are other non-sparse chunks for any given UE to work on, and the load remains well balanced.

[13] Described in Section 2.
[14] Described in Section 3.
[15] We use *processing element* as a generic term to reference a hardware element in a parallel computer that executes a stream of instructions; equivalent in most contexts to "processor".

Implementation:

Key elements.

Data decomposition.

Implementing the data-decomposition aspect of the pattern typically requires modifications in how the data structure is represented by the program; choosing a good representation can simplify the program. For example, if the data structure is an array and the update operation is a simple mesh calculation (in which each point is updated using data from neighboring points), and the implementation needs to execute in a distributed-memory environment, then typically each chunk is represented by an array big enough to hold the chunk plus shadow copies of array elements owned by neighboring chunks. (The elements intended to hold these shadow copies form a so-called "ghost boundary" around the local data, as shown in the figure under "More about data structures" earlier.)

The exchange operation.

A key factor in using this pattern correctly is ensuring that non-local data required for the update operation is obtained before it is needed. There are many ways to do this, and the choice of method can greatly affect program performance. If all the data needed is present before the beginning of the update operation, the simplest approach is to perform the entire exchange before beginning the update, storing the required non-local data in a local data structure designed for that purpose (for example, the ghost boundary in a mesh computation). This approach is relatively straightforward to implement using either copying or message-passing.

More sophisticated approaches, in which the exchange and update operations are intertwined, are possible but more difficult to implement. Such approaches are necessary if some data needed for the update is not initially available, and may improve performance in other cases as well.

Overlapping computation and computation can be a straightforward addition to the basic pattern. For example, in our standard example of a finite difference computation, the exchange of ghost cells can be started, the update of the interior region can be computed, and then the boundary layer (the values that depend on the ghost cells) can be updated. In many cases, there will be no advantage to this division of labor, but on systems that let communication and computation occur in parallel, the saving can be significant. This is such a common feature of parallel algorithms that standard communication APIs (such as MPI) include whole classes of message-passing routines to support this type of overlap.

The update operation.

If the required exchange of information has been performed before beginning the update operation, the update itself is usually straightforward to implement — it is essentially identical to the analogous update in an equivalent sequential program (i.e., a sequential program to solve the same problem), particularly if good choices have been made about how to represent non-local data. For an update defined in terms of points, each task must update all the points in the corresponding chunk, point by point; for an update defined in terms of chunks, each task must update its chunk.

Data distribution / task scheduling.

In the simplest case, each task can be assigned to a separate UE; then all tasks can execute concurrently, and the intertask coordination needed to implement the exchange operation is straightforward.

If multiple tasks are assigned to each UE, some care must be taken to avoid deadlock. An approach that will work in some situations is for each UE to cycle among its tasks, switching from one task to the next when it encounters a "blocking" coordination event. Another, perhaps simpler, approach is to redefine the tasks in such a way that they can be mapped one-to-one onto UEs, with each of these redefined tasks being responsible for the update of all the chunks assigned to that UE.

Correctness considerations.

- The primary issue in ensuring program correctness is making sure the program has the correct values for any non-local data before using it. In simple instances of this pattern (in which all non-local data is available at the start of the computation), this is easily guaranteed by structuring the program as a sequential composition of two phases, an exchange-information phase in which data is copied among UEs and a local-computation phase in which all the UEs compute new values for their chunks using the just-copied data.

- Optimizations such as avoiding copying (for shared memory or when each UE has multiple chunks) and overlapping communication with computation can improve efficiency but make it more difficult to be confident that indeed the correct values for non-local data are being used.

Efficiency considerations.

- Decisions about how to decompose and distribute the data affect efficiency not only by influencing the ratio of computation to coordination (which should be as large as possible) but also by influencing load balance. Large chunks can make it easier to achieve a high ratio of computation to coordination but can make it more difficult to achieve good load balance.

- Efficiency will likely be improved if communication can be overlapped with computation, although this likely requires a more sophisticated and hence more potentially error-prone program.

- Efficiency can also be greatly affected by the details of the implementation of the exchange operation, so it is helpful to seek out an optimized implementation, which for many applications is likely to be found in one of the collective communication routines of a coordination library (MPI, for example).

Examples:

Mesh computation.

The problem is as described in the *Motivation* section.

First, here's a simple sequential version of a program (some details omitted) that solves this problem:

```
        real uk(1:NX), ukp1(1:NX)
        dx = 1.0/NX
        dt = 0.5*dx*dx
C-------initialization of uk, ukp1 omitted
        do k=1,NSTEPS
            do i=2,NX-1
                ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
            enddo
            do i=2,NX-1
                uk(i)=ukp1(i)
            enddo
            print_step(k, uk)
        enddo
    end
```

This program combines a top-level sequential control structure (the time-step loop) with two array-update operations; the first can be parallelized using the GeometricDecomposition pattern, and the second can be parallelized using the EmbarrassinglyParallel[16] pattern. We present two parallel implementations:

- See the section "Mesh Computation, OpenMP Implementation" in the examples document (Section 4.2 of this paper) for an implementation using OpenMP.

[16] Described in Section 2.

- See the section “Mesh Computation, MPI Implementation” in the examples document (Section 4.2 of this paper) for an implementation using MPI.

Matrix multiplication.

The problem is as described in the *Motivation* section.

First, consider a simple sequential program to compute the desired result, based on decomposing the `N` by `N` matrix into `NB*NB` square blocks. To keep the notation relatively simple (though not legal Fortran), we use the notation `block(i,j,X)` to denote, on either side of an assignment statement, the (i,j)-th block of matrix `X`.

```
        real A(N,N), B(N,N), C(N,N)
C-------loop over all blocks
        do i = 1, NB
        do j = 1, NB
C-----------compute product for block (i,j) of C
            block(i,j,C) = 0.0
            do k = 1, NB
                block(i,j,C) = block(i,j,C)
     $              + matrix_multiply(block(i,k,A), block(k,j,B))
            end do
        end do
        end do
        end
```

We first observe that we could rearrange the loops as follows without affecting the result of the computation:

```
        real A(N,N), B(N,N), C(N,N)
        do i = 1, NB
        do j = 1, NB
          block(i,j,C) = 0.0
        end do
C-------loop over number of elements in sum being computed for each block
        do k = 1, NB
C-----------loop over all blocks
            do i = 1, NB
            do j = 1, NB
C-----------compute increment for block(i,j) of C
                block(i,j,C) = block(i,j,C)
  $               + matrix_multiply(block(i,k,A), block(k,j,B))
            end do
        end do
        end do
    end
```

We first observe that here again we have a program that combines a high-level sequential structure (the loop over k) with an instance of the GeometricDecomposition pattern (the nested loops over i and j).

Thus, we can produce a parallel version of this program for a shared-memory environment by parallelizing the inner nested loops (over i and j), for example with OpenMP loop directives as in the mesh-computation example.

Producing a parallel version of this program for a distributed-memory environment is somewhat trickier. The obvious approach is to try an SPMD-style program with one process for each of the `NB*NB` blocks. We could then proceed to write code for each process as follows:

```
        real A(N/NB,N/NB), B(N/NB,N/NB), C(N/NB,N/NB)
C-------buffers for holding non-local blocks of A, B
        real A_buffer(N/NB,N/NB), B_buffer(N/NB,N/NB)
        integer i, j
        C = 0.0
C-------initialize i, j to be this block’s coordinates (not shown)
C-------loop over number of elements in sum being computed for each block
        do k = 1, NB
C-----------obtain needed non-local data:
C-----------  block(i,k) of A
```

```
            if (j. eq. k) broadcast_over_row(A)
            receive(A_buffer)
C-----------  block(k,j) of B
            if (i .eq. k) broadcast_over_column(B)
            receive(B_buffer)
C-----------compute increment for C
            C = C + matrix_multiply(A,B)
        end do
        end
```

We presuppose the existence of a library routine `broadcast_over_row()` that, called from the process corresponding to block (i,j), broadcasts to processes corresponding to blocks (i, y), and an analogous routine `broadcast_over_column()`(broadcasting from the process for block (i,j) to processes for blocks (x,j)).

A cleverer approach, in which the blocks of `A` and `B` circulate among processes, arriving at each process just in time to be used, is given in [Fox88]. We refer the readers to [Fox88] for details.

Known Uses:

Most problems involving the solution of differential equations use the geometric decomposition pattern. A finite-differencing scheme directly maps onto this pattern.

Another class of problems that use this pattern comes from computational linear algebra. The parallel routines in the ScaLAPACK library are for the most part based on this pattern.

These two classes of problems cover a major portion of all parallel applications.

Related Patterns:

If the update required for each "chunk" can be done without data from other chunks, then this pattern reduces to the EmbarrassinglyParallel pattern. (As an example of such a computation, consider computing a 2-dimensional FFT by first applying a 1-dimensional FFT to each row of the matrix and then applying a 1-dimensional FFT to each column. Although the decomposition may appear data-based (by rows / by columns), in fact the computation consists of two instances of the EmbarrassinglyParallel pattern.)

If the data structure to be distributed is recursive in nature, then rather than this pattern the application designer should use the DivideAndConquer[17] or BalancedTree[18] pattern.

4.2 Supporting examples

Mesh Computation Example, OpenMP Implementation:

Generating a parallel version using OpenMP is fairly straightforward. Since OpenMP is a shared-memory environment, there is no need to explicitly partition and distribute the two key arrays (`uk` and `ukp1`); computation of new values is implicitly distributed among UEs by the two parallel loop directives, which also implicitly provide the required synchronization (all threads must finish updating `ukp1` before any threads start updating `uk`, and vice versa. The `SCHEDULE(STATIC)` clauses distribute the iterations of the parallel loops among available threads.

```
        real uk(1:NX), ukp1(1:NX)
        dx = 1.0/NX
        dt = 0.5*dx*dx
C-------initialization of uk, ukp1 omitted
        do k=1,NSTEPS
C$OMP       PARALLEL DO SCHEDULE(STATIC)
            do i=2,NX-1
                ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
```

[17] DivideAndConquer is a pattern in our AlgorithmStructure design space.
[18] BalancedTree is a pattern in our AlgorithmStructure design space.

```
            enddo
C$OMP       END PARALLEL DO
C$OMP       PARALLEL DO SCHEDULE(STATIC)
            do i=2,NX-1
                uk(i)=ukp1(i)
            enddo
C$OMP       END PARALLEL DO

            print_step(k, uk)
        enddo
        end
```

Mesh Computation Example, MPI Implementation:

An MPI-based program for this example uses the data distribution described in the Motivation section of the main document (Section 4.1 of this paper). The program is an SPMD computation in which each process's code strongly resembles the sequential code for the program, except that the loop to compute values for variable `ukp1` is preceded by message-passing operations in which the values of `uk` in the leftmost and rightmost cells of the subarray are sent to the processes owning the left and right neighbors of the subarray, and corresponding values are received. The following shows the code to be executed by each process; some details are omitted for the sake of simplicity. `NP` denotes the number of processes.

```
        real uk(0:(NX/NP)+1), ukp1(0:(NX/NP)+1)
        dx = 1.0/NX
        dt = 0.5*dx*dx
C-------process IDs for this process and its neighbors
        integer my_id, left_nbr, right_nbr, istat(MPI_STATUS_SIZE)
C-------initialization of uk, ukp1 omitted
C-------initialization of my_id, left_nbr, right_nbr omitted
C-------compute loop bounds (must skip ends of original array)
        istart=1
        if (my_id .eq. 0) istart=2
        iend=NX/NP
        if (my_id .eq. NP-1) iend=(NX/NP)-1
C-------main timestep loop
        do k=1,NSTEPS
C-----------exchange boundary information
            if (my_id .ne. 0) then
                call MPI_SEND(uk(1),1,MPI_INTEGER,left_nbr,0,
   $                MPI_COMM_WORLD,ierr)
            endif
            if (my_id .ne. NP-1) then
                call MPI_SEND(uk(NX/NP),1,MPI_INTEGER,right_nbr,0,
   $                MPI_COMM_WORLD,ierr)
            endif
            if (my_id .ne. 0) then
                call MPI_RECV(uk(0),1,MPI_INTEGER,left_nbr,0,
   $                MPI_COMM_WORLD,istat,ierr)
            endif
            if (my_id .ne. NP-1) then
                call MPI_RECV(uk((NX/NP)+1),1,MPI_INTEGER,right_nbr,0,
   $                MPI_COMM_WORLD,istat,ierr)
            endif
C-----------compute new values for this chunk of ukp1
            do i=istart, iend
                ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
            enddo
C-----------copy to uk for next iteration and print
            do i=istart, iend
                uk(i)=ukp1(i)
            enddo
            print_step_distrib(k, uk, my_id)
        enddo
        end
```

Acknowledgments

We gratefully acknowledge financial support from Intel Corporation and the NSF (grant ASC-9704697). We also thank Doug Lea for his feedback as part of the PLoP shepherding process.

References

[Bertsekas89] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation Numerical Methods.* Prentice-Hall, 1989.

[Bjornson91a] R. Bjornson, C. Kolb, and A. Sherman, "Ray Tracing with Network Linda", SIAM News, volume 24, number 1, January 1991.

[Bjornson91b] R. Bjornson, N. Carriero, T.G. Mattson, D. Kaminsky, and A. Sherman, "Experience with Linda", Yale University Computer Science Department, Technical Report RR-866, August, 1991.

[Dijkstra80] E. Dijkstra and C. S. Scholten. “Termination detection for diffusing computations”. Information Processing Letters, 11:1, August (1980).

[Fox88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[Harrison91] R. J. Harrison, “Portable Tools and Applications for Parallel Computers,” Int. J. Quantum Chem., vol. 40*, pp. 847-863, 1991*.

[Hillis86] W.D. Hillis and G.L Steele, Jr. "Data Parallel Algorithms" Comm. ACM, Vol 29 No 12 pp 1170-1183.

[Massingill99] B. L. Massingill, T. G. Mattson, and B. A. Sanders. “A Pattern Language for Parallel Application Programming”. Technical Report CISE TR 99-009, University of Florida, 1999. Available via ftp://ftp.cise.ufl.edu/cis/tech-reports/tr99/tr99-009.

[Mattern87] F. Mattern. Algorithms for distributed termination detection. Distributed Computing. 2,3 (1987).

[Mattson95] T. G. Mattson (editor), Parallel Computing in Computational Chemistry, ACS Symposium Series 592, American Chemical Society, 1995.

[Mattson96] T.G. Mattson, "Scientific Computation", in A. Zomaya (ed.) Parallel and Distributed Computing Handbook, McGraw Hill, 1996.

[Snir96] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J Dongarra, MPI: The Complete Reference, MIT Press 1996.