Reengineering for Parallelism: An Entry Point into PLPP (Pattern Language for Parallel Programming) for Legacy Applications [*]

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

Abstract

We have developed a pattern language for developing parallel application programs [MSM04]. This pattern language, which we call PLPP (Pattern Language for Parallel Programming), embodies a development methodology in which we develop a parallel application by starting with a good understanding of the problem and then working through a sequence of patterns, ending up with code. Often, however, people begin not with a problem to solve from scratch but a piece of legacy code they need to speed up by *parallelizing* it. Most of the patterns in PLPP are applicable to this situation, but it is not always clear how to get started. The pattern in this paper addresses this question and in essence provides an alternate point of entry into our pattern language.

1 Introduction

Over the course of several years we have developed a pattern language for developing parallel application programs. The patterns were developed in a series of PLoP papers [MMS99, MMS00, MMS01, MMS02, MMS03]; the complete language appears in [MSM04]. This pattern language, which we call PLPP (Pattern Language for Parallel Programming), embodies a development methodology in which we develop a parallel application by starting with a good understanding of the problem and then working through a sequence of patterns, ending up with code. Often, however, people begin not with a problem to solve from scratch but a piece of legacy code that they would like to speed up by *parallelizing* it. Most of the patterns in PLPP are applicable to this situation, but it is not always clear how to proceed. The pattern in this paper addresses

[*]Copyright © 2005, Berna L. Massingill. Permission is granted to copy for the PLoP 2005 conference. All other rights reserved.

[†]Department of Computer Science, Trinity University, San Antonio, TX; bmassing@trinity.edu.

[‡]Microprocessor Technology Laboratory, Intel Corporation, DuPont WA; timothy.g.mattson@intel.com.

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

this issue and essentially provides an alternate point of entry into the pattern language for programmers parallelizing an existing, or legacy, piece of software.

The remainder of this section describes the overall structure of our pattern language and lists the patterns that it comprises, with a brief description of each. Readers familiar with PLPP can skip directly to Section 2 (*Reengineering for Parallelism*).

1.1 Overall Organization

The pattern language is organized into four design spaces, described below. Programmers normally start at the top (in *Finding Concurrency*) and work down through the other design spaces in order until a detailed design for a parallel program is obtained.

1.2 The *Finding Concurrency* Design Space

This design space is concerned with structuring the problem to expose exploitable concurrency. The designer working at this level focuses on high-level algorithmic issues and reasons about the problem to expose potential concurrency. Patterns in this space include the following.

- Decomposition patterns, used to decompose the problem into pieces that can execute concurrently:
  - *Task Decomposition*: How can a problem be decomposed into tasks that can execute concurrently?
  - *Data Decomposition*: How can a problem's data be decomposed into units that can be operated on relatively independently?
- Dependency-analysis patterns, used to help group the tasks and analyze the dependencies among them:
  - *Group Tasks*: How can the tasks that make up a problem be grouped to simplify the job of managing dependencies?
  - *Order Tasks*: Given a way of decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?
  - *Data Sharing*: Given a data and task decomposition for a problem, how is data shared among the tasks?
- *Design Evaluation*: Is the decomposition and dependency analysis so far good enough to move on to the *Algorithm Structure* design space, or should the design be revisited?

1.3 The *Algorithm Structure* 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 working with the *Finding Concurrency* patterns. The *Algorithm Structure* patterns describe overall strategies for exploiting concurrency. Patterns in this space include the following.

- Patterns for applications where the focus is on organization by task:
  - *Task Parallelism*: How can an algorithm be organized around a collection of tasks that can execute concurrently?
  - *Divide and Conquer*: Suppose the problem is formulated using the sequential divide and conquer strategy. How can the potential concurrency be exploited?
- Patterns for applications where the focus is on organization by data decomposition:
  - *Geometric Decomposition*: How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable "chunks"?
  - *Recursive Data*: Suppose the problem involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing. How can operations on these data structures be performed in parallel?
- Patterns for applications where the focus is on organization by flow of data:
  - *Pipeline*: Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?
  - *Event-Based Coordination*: Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them, which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

1.4 The *Supporting Structures* Design Space

This design space represents an intermediate stage between the *Algorithm Structure* and *Implementation Mechanisms* design spaces: It is concerned with how the parallel algorithm is expressed in source code, with the focus on high-level program organization rather than low-level and very specific parallel programming constructs. Patterns in this space include the following.

- Patterns representing approaches to structuring programs:

  - *SPMD*: The interactions between the various units of execution (UEs) cause most of the problems when writing correct and efficient parallel programs. How can programmers structure their parallel programs to make these interactions more manageable and easier to integrate with the core computations?
  - *Master/Worker*: How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks among the units of execution?
  - *Loop Parallelism*: Given a serial program whose runtime is dominated by a set of computationally intensive loops, how can it be translated into a parallel program?
  - *Fork/Join*: In some programs, the number of concurrent tasks varies as the program executes, and the way these tasks are related prevents the use of simple control structures such as parallel loops. How can a parallel program be constructed around such complicated sets of dynamic tasks?

- Patterns representing commonly-used data structures:
  - *Shared Data*: How does one explicitly manage shared data inside a set of concurrent tasks?
  - *Shared Queue*: How can concurrently-executing units of execution (UEs) safely share a queue data structure?
  - *Distributed Array*: Arrays often need to be partitioned between multiple units of execution. How can we do this so as to obtain a program that is both readable and efficient?

This design space also includes brief discussions of some additional supporting structures found in the literature, including SIMD (Single Instruction Multiple Data), MPMD (Multiple Program, Multiple Data), client server, declarative parallel programming languages, and problem solving environments.

1.5 The *Implementation Mechanisms* Design Space

This design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide descriptions of common mechanisms for process/thread management and interaction. The items in this design space are not presented as patterns since in many cases they map directly onto elements within particular parallel programming environments. We include them in our pattern language anyway, however, to provide a complete path from problem description to code. We discuss the following three categories.

- UE[1] management: Concurrent execution by its nature requires multiple entities that run at the same time. This means that programmers must manage the creation and destruction of processes and threads in a parallel program

[1]Units of execution — generic term for processes or threads.

- Synchronization: Synchronization is used to enforce a constraint on the order of events occurring in different UEs. The synchronization constructs described here include memory fences, barriers, and mutual exclusion.
- Communication: Concurrently executing threads or processes sometimes need to exchange information. When memory is not shared between them, this exchange occurs through explicit communication events. The major types of communication events are message passing and collective communication, though we briefly describe several other common communication mechanisms as well.

2 *Reengineering for Parallelism*

Problem

How can an existing application be parallelized using PLPP to improve performance by making use of parallel hardware?

Context

The lifetime of a complex application program is long compared to the lifespan of computer systems. Hence, over time software evolves as it is ported to new architectures and has new features grafted on to fit changing needs. The result is a large base of software that is complex, convoluted, and hard to change — so-called *legacy code*. In addition to the need to adapt legacy code to run on new architectures, there is often a demand for continuing increases in performance, either to solve the same problems in less time or to solve larger problems in the same time.

Previously the conventional-wisdom answer to improving the performance of existing code was to buy faster hardware; porting the code to take advantage of parallel hardware (*parallelizing* it) was viewed as something to be done only when the "buy a faster computer" solution could not deliver sufficient increases in performance. Recent advances in hardware, however, seem to be focusing not on making single processors faster but on making parallel hardware (SMP, multi-core, and clusters) mainstream. This makes it less and less likely that it will be possible to continue to improve the performance of existing applications simply by buying faster hardware, since "faster" in hardware is coming to mean "with more opportunities for parallelism."

One reason parallelizing existing code has traditionally been a last resort for improving performance is that it is difficult, time-consuming, and error-prone. There is extensive literature on parallel programming, but it tends to focus on the problem of developing applications that are intended from the beginning to execute on parallel hardware; PLPP [MSM04] is a collection of design patterns embodying a methodology for such development. Many of the issues affecting a parallel algorithm are the same whether one starts with a blank slate or with an existing program, however; this pattern addresses the special concerns introduced when the starting point is a complex existing application.

Forces

- An existing application, especially one significant enough to parallelize, typically has a user base. These users have expectations about the application's behavior; they may be unwilling to accept any deviation from current behavior, even when the differences are mathematically valid.

- Existing applications, especially those that best fit the label "legacy code", are often complex and difficult to understand. Programmers assigned to parallelize such code may not be able to invest the time to understand every detail; they need instead to be able to make progress without full knowledge of the code.

- Amdahl's law pushes programmers to avoid sequential bottlenecks at any cost, which in turn implies major changes to existing code, even possibly a wholesale restructuring of the program.

- The starting point of the project is working code that embodies significant programming work, bug fixes, and knowledge. Minimizing change to the existing code is desirable. It is rarely feasible to make sweeping rewrites of the entire application.

- Concurrency introduces new classes of errors that are hard to detect and make software difficult to validate.

Solution

Balancing the above forces requires dealing with two different issues simultaneously: *managing the process* of changing a legacy application, and *determining what changes to make* in order to exploit parallel hardware. The problem of managing the process has much in common with any project involving legacy code. Although much of the work on managing legacy applications using design patterns assumes the use of object-oriented programming, these ideas and techniques are applicable with any software development methodology. The problem of determining what changes to make has much in common with developing a parallel application from scratch. The pattern language in PLPP provides guidance.

This pattern provides guidance for (1) managing the process and (2) using the patterns in PLPP to determine what changes to make when parallelizing a legacy application. The overall approach is based on thinking of the parallelization process as *a series of result-preserving transformations*, with testing performed after each transformation. In the next sections, we describe first some things that need to be done before starting to modify the code, and then the process of modification (reengineering).

Preparation

There are a number of things that should be done before diving into the details of the code.

SURVEY THE LANDSCAPE

As preparation for the project, it is a good idea to get a first impression of the scope of the project by briefly assessing the existing application and the supporting artifacts, and identifying the available expertise. Some questions that might help are the following.

- Does the code seem to be well structured, or is it a BIG BALL OF MUD [FY00] (a.k.a. spaghetti code)?

- What libraries does it use? Are parallel version of these libraries available?

- What documentation is available? Does it seem to be up to date?

- Who are the local experts on the program? Are any of the original programmers available?
- Who are the main users of the program?
- What kind of testing was done to validate the program? Are the tests still available? Are they compatible with the current code?
- What are the new target architectures (a single multi-core machine, a cluster, etc.)?
- Does the serial application run on the new target architectures, or will work be needed to port it before starting to parallelize?
- Is the program numerically well behaved and insensitive to effects arising from the limited precision of floating-point arithmetic? What are the bounds of the possible error? Can they be calculated? Are they acceptable?

Most of these issues are already familiar to programmers. Issues pertaining to the numerical properties of the algorithm, however, are much more subtle and require additional discussion.

Real numbers in a computer are approximated by floating-point numbers with a fixed number of bits. Results from a floating-point operation are rounded from the infinitely precise result to fit into a floating-point number. One consequence of this is that operations on floating-point values do not necessarily have the same properties as the corresponding operations on real numbers; in particular, floating-point addition is not associative. Thus, changing the order of a sequence of operations on floating-point values can change the final result because of differences caused by rounding of intermediate results. Parallelizing an algorithm may change the order of a sequence of operations, not only with respect to the serial algorithm, but even between different runs of the parallel program. Thus, it is common for parallel algorithms to generate results that are slightly different from the corresponding serial algorithms and different between different runs of the parallel algorithm.

In the majority of cases, any order of operations is equally valid. The variation in the result due to a different order of operations indicates the magnitude of the round-off error in the algorithm, and usually this round-off error is insignificant relative to the use of the results of the computation. To pick one order at random, such as the serial order, and call it "correct" is not justified. In some situations, however, the programmer has analyzed the mathematics of the algorithm in detail and used this analysis to define a specific order. If so, this order must be preserved by the parallel algorithm, limiting the speedup that can be obtained by parallelizing the algorithm. It might be a good idea to revisit the plan to parallelize the algorithm, as the anticipated performance improvements may be impossible to achieve. Finally, it sometimes happens that the round-off error is significant, and that this has not been noticed before by the users of the serial program. This indicates deficiencies in how the computation has been formulated and the mathematical constructs cast onto their representations with floating-point arithmetic. In this case, the serial algorithm cannot be parallelized and needs to be corrected.

Level of confidence in the original developers, comments in the code, and simple tests that reverse the order of iterations in loops can be used to determine whether numerical issues are likely to be a problem in a specific program. Comments and examination of the loops will usually reveal whether a specific order has been imposed that probably should be preserved.

A good source of additional information about these issues is [Gol91].

Define scope and get the users' buy-in

Establish targets in the following areas, validating them with the application's users and managing their expectations.

- Required precision of result: In the simplest view, a correct parallelization of an existing serial application gives output that is bit-for-bit identical to the output of the original code. However, in many cases there will be differences due to the limitations of floating-point arithmetic, as discussed earlier. Be prepared to educate users about such differences and work with them to determine the number of significant figures in the computational results, so that the amount of allowable variation can be specified.

- Input range: Determine the range of inputs and problem sizes for which the reengineered program is expected to work.

- Performance: Define performance goals. Sometimes this will be a hard constraint (e.g., an application to render 3D models in real time needs to generate a known number of frames per second). In other cases, it will be an expected level of scalability as additional processors are added.

- Feasibility: As soon as possible, do back-of-the-envelope calculations to get a rough idea of whether the users' expectations seem realistic. Will problems of the desired size fit into the memory of the parallel machine? What upper bound on the speedup does Amdahl's law give? Will that satisfy expectations? (The latter computation is easy once profiling of the serial computation on the new architecture has been done.)

Define a testing protocol

We need a concrete definition of the expected behavior of the application. We create one by constructing a test suite that can be used throughout the parallelization process. Test suite construction can be complicated [Mye79]: The test suite must exercise all portions of the program that will be encountered as the program is used. Furthermore, the test suite must explore the full range of input that may be encountered as the program is used. It is important to pay special attention to difficult or even mathematically pathological cases. Finally, it is important that both running the test suite and analyzing the output from the test suite be automated so it can be easily rerun after each step of transforming the program.

Identification of Hot Spots

The first step in reengineering the program for parallelism is to understand it well enough to determine its *hot spots* — that is, where it is spending most of its time. These will be the parts to try to parallelize first. Two approaches can help in finding hot spots:

- Read the code and try to understand the high-level algorithm — the overall structure of the computation rather than the details. Often large computations contain a clear sequence of phases. Once these are identified, it may be possible to determine where the program is likely to be spending most of its time. Also, identify the key data structures. Are there large arrays or collections of objects that appear to be central to the computation?

- Use profiling tools. Time up front learning how to use performance analysis tools in most cases pays off.

Parallelization

Once the program's hot spots have been identified, work can begin on parallelizing them. Focusing first on the hot spots that have the greatest influence on overall performance, use the patterns of PLPP to determine how to introduce parallelism based on the following sequence of steps. Notice that if the overall process of changing the code can be broken down into a sequence of small transformations, each followed by a validation/evaluation step, debugging is likely to be less painful.

Although we present our approach as a linear sequence of steps, first applied to one hot spot and then to the next, in practice it is a good idea, before committing to any changes, to look again at the whole program, or at least at the next few major hot spots, to see whether they have a similar structure. If so, there may be a common strategy that can deal with all of them.

DIG DEEPER INTO THE CODE

Dig deeper into the code implementing the hot spot chosen for attention. Identify the main data and control structures, and note how the data flows through the section of code being analyzed, including its boundaries. Often, in large and computationally intense programs, one finds that the main control structure is a loop iterating over elements of an array. Recursive control and data structures are also possible. In some programs, the structure may be complicated by optimizations, for example, loop blocking to improve cache behavior. It may be worthwhile to refactor the program to change the loops back to a simpler, non-optimized form before parallelization, both for clarity and to allow easier tuning of the parallelized program.

IDENTIFY EXPLOITABLE CONCURRENCY USING *Finding Concurrency*

Review the patterns in the *Finding Concurrency* design space to identify exploitable concurrency — first the decomposition patterns, then the patterns for analyzing dependencies. Although one is more constrained when parallelizing legacy code than when starting with a blank slate, the same approach described in the patterns is applicable. Sometimes it is easier to find exploitable concurrency by thinking in terms of the data (i.e., to first define a data decomposition, and then infer from that a task decomposition). Other times it is easer to focus on the control structure itself (i.e., to first define a task decomposition, and then infer from that a data decomposition). These dual decompositions are discussed in *Task Decomposition* and *Data Decomposition*. A common example is a computation that loops over elements of an array — the tasks correspond to a subset of the loop iterations, while the array is decomposed into groups of elements, each consisting of elements handled by one task. In a divide-and-conquer algorithm, it is often effective to associate tasks with the recursive calls. Once the major decomposition strategy had been found, identify the dependencies between tasks and how they can be managed. *Data Sharing* can be especially helpful here. Look very closely at how the complexity of managing the dependencies might change if the decomposition were changed in some way.

Move back and forth between the decomposition and dependency-analysis patterns until the result seems consistent and effective. Then look at *Design Evaluation* to make sure that all the issues it describes have been addressed.

CHOOSE AN OVERALL STRATEGY USING *Algorithm Structure*

Next, use the results of the analysis from the *Finding Concurrency* design space to choose an *Algorithm Structure* pattern. This is done in much the same way regardless of whether one is writing a new parallel program or parallelizing a legacy application, that is, either by using the decision tree presented in [MSM04] or by simply skimming the descriptions of the patterns looking for ones that fit. The *Algorithm Structure* patterns most used when parallelizing legacy code are *Task Parallelism* (when the primary decomposition principle is a set of tasks), *Geometric Decomposition* (when the primary decomposition is the division of data into chunks), and *Divide and Conquer* (when the control structure is recursive). The other *Algorithm Structure* patterns (*Pipeline*, *Event-Based Coordination*, and *Recursive Data*) are not commonly used when parallelizing legacy code, since they tend to require large scale restructuring that goes well beyond changes tolerated in most projects to parallelize legacy applications.

Notice also that it is possible that more than one *Algorithm Structure* pattern can be exploited, either combined hierarchically or in sequence.

IMPLEMENT THE STRATEGY USING *Supporting Structures*

Next, implement the chosen strategy using one or more of the *Supporting Structures* patterns, keeping mind the following.

- When choosing a program-structuring pattern, the set of target platforms must be taken into account. If the target platforms present the programmer with a single address space, programs typically use multithreading with a shared-memory API such as OpenMP. This is particularly well suited to the *Loop Parallelism* pattern. If the target platforms include ones with distributed memory, a shared-nothing, multi-process approach based on the *SPMD* pattern is more common.

Note that even when the system provides a single address space, it may still be preferable to use a distributed memory approach and *SPMD* for the program. This happens most commonly when the program's control structure is complicated and not dominated by loops. This approach is also used when the program's data access patterns are complicated and unintended sharing of data (and hence race conditions) is a serious concern.

- *SPMD* is the most broadly applicable of the program-structuring patterns. It often, however, requires more-sweeping program changes than, say, *Loop Parallelism*, and it may not be obvious how to organize these changes into a sequence of small transformations. This is particularly the case if the target platform does not support shared memory, thereby requiring all shared data structures to be explicitly distributed among UEs. A key aspect of parallelizing existing code using *SPMD* is to decide which data should be replicated (so each UE has its own copy) and which should be distributed among UEs. Once a decision has been made that a particular data structure should be distributed, eventually every computation that involves this data structure must be modified (typically using an *owner-computes* rule). Modifying all of these computations at once, however, does not fit well with the overall strategy of making changes in small steps, testing as one goes. To work around this, begin by replicating the data structure and having all UEs perform all the calculations concurrently, and then parallelize one computation at a time. Notice that if a parallelized calculation (in which each UE operates only on its own data) is followed by a serial calculation (in which all UEs perform the calculation on replicated data), the serial calculation must be preceded by communication operations to copy the results of the parallel calculation to other UEs. Once all computations have been parallelized, the data structure can be converted into its final distributed form.

Evaluate and debug

Evaluate how well the changed code meets the correctness and performance targets. Debug as needed.

Repeat as needed

Repeat the preceding steps (identify concurrency, make changes to exploit it, evaluate the result) for the program's other hot spots.

Evaluate the final result in terms of the original correctness and performance targets. At this point, more is known about the application, so it is worthwhile to review the targets themselves. A point made in several of the patterns of PLPP is that design is an iterative process, in which backtracking and reworking may be needed to achieve a good result. This applies to the process of parallelizing existing code as much as to the development of a new application. Many of the relevant issues are addressed in *Design Evaluation*.

Example

As an example of applying the patterns of PLPP to existing code, we consider parallelizing an electromagnetics application that uses the finite-difference time-domain (FDTD) technique to model transient electromagnetic scattering and interactions with objects of arbitrary shape and composition. The application is described in more detail in [KL93]; an earlier experiment with parallelizing it is described in [Mas99].

In the serial program, the object and surrounding space are represented by a 3D grid of computational cells. An initial excitation is specified; the rest of the computation consists of a time-stepped simulation of the electric and magnetic fields over the grid. At each time step, the program first calculates the electric field in each cell based on the magnetic fields in the cell and in neighboring cells, and then similarly calculates the magnetic fields based on the electric fields. Fig. 1 shows pseudocode for the algorithm.

```
// global variables

    // grid dimensions
    Int const NX, NY, NZ

    // properties of grid (representation of object and space)
    Array of Properties :: prop (NX, NY, NZ)

    // x, y, z components of electric fields
    Array of Real :: elec_x (NX, NY, NZ)
    Array of Real :: elec_y (NX, NY, NZ)
    Array of Real :: elec_z (NX, NY, NZ)

    // x, y, z components of magnetic fields
    Array of Real :: mag_x (NX, NY, NZ)
    Array of Real :: mag_y (NX, NY, NZ)
    Array of Real :: mag_z (NX, NY, NZ)

    // other variables (more details later)

// code

    initialize()

    loop over time steps

        update_elec_fields()
        adjust_elec_field_boundaries()
        update_mag_fields()
        output_selected_values()

    end loop
```

Figure 1: Simplified pseudocode for electromagnetics application.

Preparation

SURVEY THE LANDSCAPE

We start by working through the list of questions posed in the Solution section above.

- Does the code seem to be well structured, or is it a BIG BALL OF MUD (a.k.a. spaghetti code)?

  The serial program is written in FORTRAN 77. It is reasonably well structured, with a clean separation into subroutines, but nearly all data is global (i.e., kept in `COMMON` blocks), which makes it more difficult to identify which subroutines affect which variables.

- What libraries does it use? Are parallel version of these libraries available?

  It uses no outside libraries.

- What documentation is available? Does it seem to be up to date?

  The program contains fairly extensive comments, which appear to be consistent with the code.

- Who are the local experts on the program? Are any of the original programmers available?

  At the time the program was first parallelized (as described in [Mas99]), a local expert familiar with the code was available.

- What kind of testing was done to validate the program? Are the tests still available?

  The program's input is hard-wired into the code itself, so the program is a self-contained test. Comments in the code indicate how to in effect vary the input by changing the code, but we do not have access to domain experts who could help with this, beyond very simple changes such as varying the number of time steps and the number of cells in the grid. Reportedly, the output of the program was compared with an analytic solution of the problem and found to be correct.

- What are the new target architectures (a single multi-core machine, a cluster, etc.)?

  The target platform is a small cluster of workstations; the target parallel programming environment is MPI.

- Does the serial application run on the new target architectures, or will work be needed to port it before starting to parallelize?

  Yes, the serial code runs without problems on a single workstation.

- Is the program numerically well behaved and insensitive to effects arising from the limited precision of floating-point arithmetic?

  The most common cause of differences in output due to rearranged computations (as described in the Solution section above) is a reduction operation involving an operation that is not quite associative. This program does not appear to involve any such calculations, so it is reasonable to hope for bit-for-bit equivalence of output.

DEFINE SCOPE AND GET THE USERS' BUY-IN

In some sense the serial code is not so much an application with an established user base as a proof of concept for the algorithm it embodies. Parallelizing will similarly be a proof of concept; the goal will be bit-for-bit equivalent output and better performance. Again, working through the list of topics described in the Solution section above:

- Required precision of result:

  As discussed earlier, we believe that none of the calculations will be subject to reordering that could change the results, so we believe it is reasonable to aim for output that is bit-for-bit equivalent to that of the serial program.

- Input range:

  This is not really an issue; aside from varying the number of time steps and/or grid cells, the input is always the same (what is hard-coded in the program).

- Performance:

  We set a modest goal here: We want the parallelized algorithm to be not significantly slower than the original code on a single workstation and to show reasonable speedups as we increase the number of processors.

- Feasibility:

  Since the performance goal is modest, it does not seem to make sense to spend much time considering feasibility, beyond an observation that the calculations seem to mostly involve the kind of operations on large grids that are generally amenable to parallelization.

DEFINE A TESTING PROTOCOL

The testing protocol is simple: Run the program and compare its output to that of the original code.

Identification of Hot Spots

This program is simple enough that its hot spots can be identified via a basic understanding of the computation: Most of the work of the computation consists of the alternating updates of the electric and magnetic fields, so those will be the target of parallelization efforts.

Parallelization

Following the steps outlined earlier:

DIG DEEPER INTO THE CODE

The code making up the program's hot spots is straightforward — nested loops over the arrays that represent the electric and magnetic fields in the grid. Most of the rest of the code is similarly organized around loops over grid-based arrays. The program's variables fall into three categories:

- Grid-based arrays; that is, arrays with one element per grid cell. These arrays include the `elec_*`, `mag_*`, and `props` arrays of Fig. 1.
- Grid-related arrays; that is, arrays whose sizes are related to the grid size, but which do not fit into the previous group. This category includes work arrays used in the `adjust_elec_field_boundaries()` step; these are 3D arrays, where two of the dimensions are the same as the grid and the third dimension is constant. (So, one might think of each of these arrays as being the right size and shape to match up with one of the faces of the grid.)
- Non-grid-related variables; that is, miscellaneous constants and small arrays (with dimensions unrelated to the grid size).

Digging further into the code, we can make the following observations.

- `initialize()` consists of several subroutines that initialize all the variables (grid-based and otherwise). There is some error checking built in (not important for the current hard-coded input, since it presumably is error-free, but useful if the program were changed to embody different input); some of the subroutines also print informational data (such as the number of time steps).
- `update_elec_fields()` consists of three nearly-identical subroutines, one for each component of the field. Each subroutine updates one component of the field, looping over all cells in the grid and computing a new value in that cell based on its previous value, the values of other arrays in the same cell and nearby cells, and its position in the grid. For example, the pseudocode in Fig. 2 illustrates the calculation for `elec_x`.
- `update_mag_fields()` consists of three nearly-identical subroutines, one for each component of the field. Each subroutine updates one component of the field, looping over all cells in the grid and computing a new value in that cell based on its previous value and the values of other arrays in the same cell and nearby cells; the overall structure is similar to the subroutines that make up `update_elec_fields()`, but the calculations are simpler. For example, the pseudocode in Fig. 3 illustrates the calculation for `mag_x`.
- `adjust_elec_field_boundaries()` consists of six nearly-identical subroutines, two for each component of the field. These subroutines make use of 12 additional arrays that fit the description of grid-related (but not grid-based) variables given earlier; for example, the ones used in updating `elec_x` have the form shown in Fig. 4.

```
loop over k in 1 .. NZ
loop over j in 1 .. NY
loop over i in 1 .. NX
    elec_x(i, j, k) = complicated_function(
        elec_x(i, j, k),
        mag_y(i, j, k), mag_y(i, j, k-1),
        mag_z(i, j, k), mag_z(i, j-1, k),
        props(i, j, k),
        i, j, k,
        .... ) // miscellaneous non-grid-related variables
end loop
end loop
end loop
```

Figure 2: Simplified pseudocode for update of `elec_x`.

```
loop over k in 1 .. NZ
loop over j in 1 .. NY
loop over i in 1 .. NX
    mag_x(i, j, k) = not_so_complicated_function(
        mag_x(i, j, k),
        elec_y(i, j, k), elec_y(i, j, k+1),
        elec_z(i, j, k), elec_z(i, j+1, k),
        .... ) // miscellaneous non-grid-related variables
end loop
end loop
end loop
```

Figure 3: Simplified pseudocode for update of `mag_x`.

.

```
Array of Real :: workx_1 (NX-1, 4, NZ-1)
Array of Real :: workx_2 (NX-1, 4, NZ-1)
Array of Real :: workx_3 (NX-1, NY-1, 4)
Array of Real :: workx_4 (NX-1, NY-1, 4)
```

Figure 4: Simplified example of grid-related (but not grid-based) work arrays.

Each subroutine updates one component of the field (e.g., `elec_x`), looping over all cells on the boundary of the grid and computing a new value for `elec_x` based its old value, the value of `elec_x` in nearby cells, and the values of `workx_*` in nearby cells.

- `output_selected_values()` consists of a single subroutine. The first time this subroutine is called, it initializes arrays representing sampling points — indices of selected cells for which output is desired. On that and subsequent calls, it computes and prints values for the specified sampling points (based on the values of the electric and magnetic fields in the selected cells and nearby cells). The pseudocode in Fig. 5 outlines the calculation.

```
//  global variables
    Int const NPOINTS

    Array of Coordinates :: sample_points (NPOINTS)
    Array of Real        :: values (NPOINTS)

//  code
    if first time step
        initialize sample_points
    end if

    loop over m in NPOINTS
        values(m) = collect_data(sample_points(m))
    end loop

    write values to output file
```

Figure 5: Simplified pseudocode for `output_selected_values`. `collect_data()` obtains data from global variables, including electric and magnetic fields, using the coordinates in `sample_points(m)`.

IDENTIFY EXPLOITABLE CONCURRENCY USING *Finding Concurrency*

Working through the patterns in *Finding Concurrency*, it seems fairly clear that this problem is best approached by decomposing its major data structures, namely the arrays representing the grid-based variables (electric and magnetic fields and grid properties). Since most of the calculations for a particular cell involve values in the same and nearby cells, a decomposition in contiguous subarrays seems to make sense. We can similar decompose the work arrays used in updating boundary points. We can then infer task decompositions for grid-based calculations (the ones involved in updating the electric and magnetic fields); for each such calculation, there will be one task for each subarray, consisting of updating the elements in that subarray.

Referring again to the previous discussion of how the variables in the program fall into three categories, we observe that the decomposition so far tells us something about what to do with the variables we described as grid-based and grid-related, but it tells us nothing about what to do with the variables not related to the grid (constants and small arrays), so we must continue.

With regard to how data is shared among tasks, we can make a few observations: Most of the non-grid-related variables are set once, at the beginning of the program, and not changed thereafter, except for the ones used to collect data for output. In the terminology of *Data Sharing*: The grid-based and grid-related variables consist of effectively-local data (elements in the interior of one of the contiguous subarrays) and multiple-read/single-write data (elements on the boundary of one of the subarrays), and the decomposition strategy we have so far seems reasonable. The non-grid-related variables are either read-only data or accumulate data, which suggests that an appropriate strategy for dealing with these variables is to replicate them (one copy per UE) and periodically recombine the ones that represent accumulate data.

CHOOSE AN OVERALL STRATEGY USING *Algorithm Structure*

The applicable *Algorithm Structure* pattern, given this analysis, is *Geometric Decomposition*, which is based on partitioning grid(s) into regular contiguous subgrids (local sections) and distributing them among processes. This partitioning and distribution is described in more detail in *Distributed Array* (a *Supporting Structures* pattern).

The calculations that make up the program's hot spots (the grid updates) are a classic example of *Geometric Decomposition*; the other calculations (initializing, adjusting boundary values, and writing output) are also fairly typical of this pattern. The details of the parallelization are somewhat involved, but the underlying ideas are straightforward:

- The strategy is based on decomposing the 3D grid (consisting of what we have described as grid-based and grid-related variables) into contiguous subgrids, one per UE. Each element of a grid-based or grid-related array corresponds to one cell in the 3D grid; from this idea follows the strategy for distributing these arrays among UEs. Non-grid-related variables are replicated, with each UE getting its own copy.

- Updates to variables that have been distributed will be split up among UEs using an *owner-computes* strategy: Wherever there is a loop over all cells in the grid, rewrite the loop to address only cells in the local section; if the loop is over selected cells, rewrite so that each UE operates on the selected cells that are part of its local section. If these updates need data owned by another UE (as will happen with updates of some values in cells near the boundary of a subgrid), synchronization or communication is needed to make this data (from other UEs) available without race conditions.

- Updates to variables that have been replicated can either be performed simultaneously by all UEs or using some type of reduction (e.g., checking for errors by having each UE compute a local value for number of errors and then taking the sum or maximum of all these local values). The exception to this general principle is the program's output files; it is simplest (and will work for this program) to delegate all operations on output files to a single UE.

- Each cell in the grid has both global coordinates (with respect to the original grid) and coordinates in the decomposition (a combination of UE ID and local coordinates). In the serial program, these are identical, so coordinates can be used both for locating a cell's variables and also for purposes such as computing a distance from a fixed point. In the parallel program, we will need to use local coordinates for the locating variables but global coordinates for computing distances and similar calculations.

Most of these ideas are discussed in more detail in *Geometric Decomposition* and *Distributed Array*.

IMPLEMENT THE STRATEGY USING *Supporting Structures*

Since the target platform is one without shared memory, the obvious choice of program structure is *SPMD*. *Geometric Decomposition* and *Distributed Array* map well onto this program structure; it is mostly a matter of getting the many details right.

As a first step, we review the program code again, asking the following questions:

- For each variable, is it grid-based, grid-related, or non-grid-related? This will tell us whether it will be duplicated or replicated.

- For each of the major loops in the program, what variables does it use as input and what as output?

  If it uses grid-based or grid-related variables as output, we need to split up the computation based on the owner-computes strategy. If it uses such variables as input, and some values used for one cell come from other cells, we will need to insert code to communicate these values, preferably before starting the loop. A library function that performs a standard exchange of boundary functions will probably be helpful.

  If it uses other variables, very often the calculation can simply be replicated (i.e., performed in all UEs). Some calculations, however, need to either be limited to a single UE (e.g., writing to output files) or require some type of reduction (e.g., checking for errors).

- For each use of cell coordinates, are the coordinates used only to find elements of related arrays, or are they being used as global coordinates (e.g., in computing a distance)?

Once all of this has been analyzed, the actual work of parallelizing the program can begin. A few examples may help give a sense of how the strategy plays out.

First, some of the global variables need to be changed to reflect the strategy of distributing data. Fig. 6 sketches some of the needed changes.

Next, consider parallelizing the code that updates one component of the electric field. Pseudocode for the serial version was shown in Fig. 2; pseudocode sketching a parallelization is shown in Fig. 7.

Finally, consider parallelizing the code that collects and writes out the output data. Pseudocode for the serial version was shown in Fig. 5; pseudocode sketching a parallelization is shown in Fig. 8.

```
// grid dimensions
Int const NX, NY, NZ
// dimensions of local section (excluding ghost boundary)
Int const NXlocal, NYlocal, NZlocal
// low/high indices for local section (including ghost boundary)
Int const IXLO, IXHI IYLO, IYHI, IZLO, IZHI

// properties of grid (representation of object and space)
Array of Properties :: prop(IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)

// x, y, z components of electric fields
Array of Real :: elec_x (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
Array of Real :: elec_y (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
Array of Real :: elec_z (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)

// x, y, z components of magnetic fields
Array of Real :: mag_x (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
Array of Real :: mag_y (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
Array of Real :: mag_z (IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
```

Figure 6: Revised global variables.

```
loop over k in 1 .. NZlocal
loop over j in 1 .. NYlocal
loop over i in 1 .. NXlocal
    elec_x(i, j, k) = complicated_function(
        elec_x(i, j, k),
        mag_y(i, j, k), mag_y(i, j, k-1),
        mag_z(i, j, k), mag_z(i, j-1, k),
        props(i, j, k),
        local_to_global_x(i),
        local_to_global_y(j),
        local_to_global_z(k),
        .... ) // miscellaneous non-grid-related variables
end loop
end loop
end loop
```

Figure 7: Parallelization of pseudocode in Fig. 2 (update of `elec_x`). Fig. 6 shows changes to variables. Notice the changes to loop limits and also the use of functions `local_to_global_x`, etc., to transform local coordinates to global coordinates.

```
//  global variables
    Int const NPOINTS

    Array of Coordinates :: sample_points (NPOINTS)
    Array of Real        :: values (NPOINTS)

//  local variable
    Array of Real        :: values_local (NPOINTS)

    if first time step
        initialize sample_points
    end if

    loop over m in NPOINTS
        if (in_local_section(sample_points(m))
            values_local(m) = collect_data(sample_points(m))
        else
            values_local(m) = 0
        endif
    end loop

    reduce(values_local, values, NPOINTS, SUM)

    if output process
        write values to output file
    endif
```

Figure 8: Parallelization of pseudocode in Fig. 5 (`output_selected_values()`. Notice that all UEs replicate the initialization, each collects data for sampling points in its local section, and the results of these collections are reduced into an array that can be printed by the single UE responsible for writing to output files.

These examples, as noted, should give a sense of how the strategy plays out. There are many details to consider, far too many to list, but some observations may be useful:

- Sometimes it can help to rearrange some of the original computation before beginning to parallelize. In the code for this program, there are many instances of loops in which different statements in the loop body require slightly different parallelization strategies. It can be helpful to first partition the loop into two loops and treat them individually; Fig. 9 sketches an example.

```
//  original code
    loop over i in 1 .. NX-1
        var1(i) = f(i)
        var2(i+1) = g(i)
    end loop

//  modified code
    loop over i in 1 .. NX-1
        var1(i) = f(i)
    end loop
    loop over i in 2 .. NX
        var2(i) = g(i-1)
    end loop
```

Figure 9: Example of loop transformation useful before beginning parallelization. `var1` and `var2` are arrays that will be distributed. In the original form, the loop bounds do not match which elements of `var2` are being changed, which will make it more complicated to apply the owner-computes strategy. Converting to two loops avoids this problem.

- Other rearrangements may also help. For example, each of the subroutines that make up what is called `adjust_elec_field_boundaries()` in the pseudocode consists of two phases, one that updates one of the components of the electric field and another that uses the updated component to update work arrays. Based on an analysis of exactly which elements are used as input and output, it is apparent that between these two phases there needs to be an exchange of boundary values on the updated component. It turns out to be simpler and more efficient to rearrange the code so that we first do the first phase in each of the six subroutines, then do all the exchanges of boundary values, and then do the second phase in each subroutine.

- A well-chosen set of utility routines can be a great help. For this program, helpful routines include a routine to exchange boundary values between UEs containing neighboring subgrids, transformations between local and global indices, and routines to determine whether a given range of global indices overlaps the local section. Such routines might already be available as a result of parallelizing other programs using a similar strategy.

- While a strategy of incremental parallelism is very attractive in general, there are difficulties in applying this strategy to *SPMD* programs, as described earlier (in the Solution section). However, with this program it is possible to at least make the rearrangements discussed above (loop rearrangements, restructuring of `adjust_elec_field_boundaries()`) and confirm that they preserve results before starting the changes related to decomposing and distributing arrays. It is also easy and somewhat helpful to take the first step into the MPI world (the target platform for this program) by transforming the program into one in which all processes execute the original code (plus performing the usual MPI setup and shutdown), except that only one opens the output file and writes results.

EVALUATE AND DEBUG

The parallelized code (produced using the analysis and techniques described in this paper, plus the code developed for [Mas99]) meets the correctness targets: Output on a cluster of workstations is bit-for-bit identical to the output of the serial program executed on one workstation. Performance meets the modest goals we set: Execution time of the parallel program on a single workstation is only slightly more than that of the original code (reflecting a small amount of added overhead), and for nontrivial problem sizes the parallel code shows continuing decreases in execution time as the number of processes increases.

Related Patterns

Parallelizing a legacy serial application requires solving many of the same problems, both social and technical, as working with any legacy code, plus additional issues specific to parallelization. Pattern languages that address some of the social and technical issues of working with legacy code that are likely to be useful for parallel programmers are [DDN02, FY00]. Other work dealing with refactoring legacy programs [Fow99, Fea04, Ker04] employs the same overall approach as advocated in this pattern: define tests and then perform small modifications, testing after each change. The focus, however, is on improving the structure of object-oriented programs, and the individual patterns are at a fairly low level.

Help with the parallelization itself can be found, as mentioned earlier, in [MSM04], and also in [BMR+96, SSRB00, Lea00].

Acknowledgments

We gratefully acknowledge the help of our shepherd for this paper, Brian Foote.

References

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Son Ltd, 1996.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 1st edition, 2002.

[Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, 2004.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition, 1999. With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts.

[FY00] Brian Foote and Joseph Yoder. Big ball of mud. In Neal Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, March 1991.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Signature Series. Addison-Wesley Professional, 2004.

[KL93] K. S. Kunz and R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*. CRC Press, 1993.

[Lea00] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2nd edition, 2000.

[Mas99] Berna L. Massingill. Experiments with program parallelization using archetypes and stepwise refinement. *Parallel Processing Letters*, 9(4):487–488, 1999. Also in *Parallel and Distributed Processing (Lecture Notes in Computer Science, vol. 1388)*, 1998, edited by José Rolim, and in *Proceedings of the Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98 / IPPS'98)*. Extended version available as UF CISE TR 98-012 (`ftp://ftp.cise.ufl.edu/cis/tech-reports/tr98/tr98-012.ps`).

[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)*, August 1999.

[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.

[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)*, September 2001.

[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)*, September 2002.

[MMS03] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Additional patterns for parallel application programs. In *Proceedings of the Tenth Pattern Languages of Programs Workshop (PLoP 2003)*, September 2003.

[MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Son Ltd, 2000.