**Reengineering for Parallelism:**
An entry point into PLPP
(Pattern Language for Parallel Programming) for legacy applications†

†Berna L. Massingill¹*, Timothy G. Mattson², and Beverly A. Sanders³

¹Department of Computer Science, Trinity University, San Antonio, TX; bmassing@trinity.edu
²Microprocessor Technology Laboratory, Intel Corporation, DuPont WA; timothy.g.mattson@intel.com
³Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL; sanders@cise.ufl.edu

**SUMMARY**

Many parallel programs begin as legacy sequential code that is later reengineered to take advantage of parallel hardware. This paper presents a pattern called **Reengineering for Parallelism** to help with this task. The new pattern is intended to be used in conjunction with PLPP (Pattern Language for Parallel Programming), described in book form in **Patterns for Parallel Programming** (Mattson, Sanders, and Massingill; Addison-Wesley; 2004). PLPP contains a structured collection of patterns and embodies a methodology for developing parallel programs in which the programmer starts with a good understanding of the problem, works through a sequence of patterns, and finally ends up with the code. Most of the patterns in PLPP are also applicable when reengineering legacy code, but it is not always clear how to get started. **Reengineering for Parallelism** provides an alternate point of entry into PLPP and addresses particular issues that arise when dealing with legacy code.

**KEY WORDS:** Parallel programming, design patterns, pattern language, parallelization of legacy code

†This is a preprint of an article accepted for publication in **Concurrency and Computation: Practice and Experience** © 2006 John Wiley & Sons, Ltd.
*Correspondence to: Berna L. Massingill; Department of Computer Science, Trinity University; One Trinity Place; San Antonio, TX, 78212-7200; bmassing@trinity.edu
1. Introduction

Parallel hardware has become common. With the emergence of mainstream microprocessors containing multiple processor cores, parallel hardware is truly becoming ubiquitous. Parallel software, however, continues to be rare. Even with numerous programming environments and an extensive body of literature on parallel programming technology, most programmers do not write parallel software.

A key step in solving this problem and making parallel software as routine as parallel hardware is to develop a model for how parallel software is constructed. By writing this model down systematically, programmers new to parallel computing can more quickly learn how to write parallel software. More importantly, however, by making the model explicit, parallel computing researchers can systematically analyze, test, and debate the elements of this model.

Our model for how parallel software is created (PLPP — Pattern Language for Parallel Programming) uses the formalism of a design pattern language. Design patterns have proved popular in object-oriented programming as a way of capturing expert knowledge and making it available for more general use ([7], [2], and many others). A design pattern is a textual representation of a high-quality solution to a frequently occurring problem in some domain, written in a prescribed format chosen to make it easy for the reader to quickly understand both the problem and the proposed solution. For example, the patterns in PLPP follow the pattern format described in [21] and include some or all of the following sections.

- A name that provides a shorthand way of referring to the pattern.
- A short problem statement.
- A description of the context in which the problem occurs, which determines when this pattern is applicable.
- A set of forces — possibly conflicting goals and constraints that must be taken into account when developing a solution to the problem.
- A description of the proposed solution.
- One or more examples, including references where applicable to other work (“known uses”) that uses this pattern.
- A short discussion of related patterns.

Because each pattern has a name — ideally one that expresses the key idea in a few words — a collection of patterns provides a vocabulary with which to talk about these solutions.

A pattern language, however, is more than simply a catalog of patterns; it is a complex web of design patterns structured in such a way that complex systems can be designed using the patterns. At each decision point, the designer selects an appropriate pattern. Each pattern leads to other patterns, resulting in a final design in terms of a web of patterns. A pattern language thus embodies a design methodology and provides domain-specific advice to the application designer. (Notice that despite the overlapping terminology, however, a pattern language is not a programming language.) In the case of PLPP, the domain is parallel programming, and the pattern language embodies a development methodology for parallel programmers. Experienced parallel programmers likely work through a sequence of steps in their heads that closely resemble the patterns of PLPP; one contribution of PLPP is to make these steps explicit, so that they can be understood and followed by the less experienced. In addition, what is explicit can be discussed and debated, we hope leading to more effective approaches in the future.
In the methodology embodied by PLPP, a programmer starts with a good understanding of the problem and works through a sequence of patterns, finally ending up with a detailed design and possibly working 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 Reengineering for Parallelism pattern introduced in this paper extends PLPP by addressing this issue and providing an alternative entry point into PLPP. The format of this pattern matches the format used in our book [20]. The pattern includes a detailed example of its use.

The next section gives a brief overview of PLPP. Section 3 contains the new pattern. Section 4 gives an overview of related work and concluding remarks.

2. Overview of PLPP

PLPP was developed over the course of several years. Early versions of many of the patterns were presented at PLoP workshops [13, 15, 16, 18, 19] and other venues [14, 17], culminating in the publication of a book [20] containing the entire pattern language.

This section describes the overall structure of PLPP and lists the patterns that it comprises, with a brief description of each. Space considerations make it impractical to include more detail in this paper; the full text of the patterns is given in [20] containing the entire pattern language. Readers familiar with PLPP can skip directly to Section 3 (Reengineering for Parallelism).

In the remainder of this paper, capitalized phrases in italics, such as This Name, refer to a pattern or group of patterns in PLPP.

2.1. Overall organization

The pattern language is organized into four design spaces, described below, corresponding to four broad phases of program design. 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. For each pattern, we give its problem statement (as described in Section 1) and a little about its key ideas.

2.2. 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.

Before starting to work with the patterns in this design space, the algorithm designer must first consider the problem to be solved and make sure the effort to create a parallel program will be justified: Is the problem sufficiently large, and are the results sufficiently significant, to justify expending effort to solve it faster? If so, the next step is to make sure the problem’s computations and data elements are well understood. Finally, the designer needs to understand which parts of the problem are most
computationally intensive, since it is on those parts of the problem that the effort to parallelize the problem should be focused.

Once this analysis is complete, the patterns in the Finding Concurrency design space can be used to start designing a parallel algorithm. The patterns in this design space can be organized into three groups as follows.

- **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?
    A good task decomposition balances the competing forces of flexibility, efficiency, and simplicity; for example, it decomposes the computation into tasks that are small enough to provide scheduling flexibility, but not so small that they are prohibitively expensive to manage.
  - **Data Decomposition**: How can a problem’s data be decomposed into units that can be operated on relatively independently?
    A good data decomposition balances the competing forces of flexibility, efficiency, and simplicity in much that same way as a good task decomposition does. In some cases the data decomposition arises naturally from the task decomposition; in others it should be thought through explicitly.

- **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?
    Tasks can be grouped in various ways — logically (based on problem decomposition) or in order to make it easy to satisfy ordering constraints.
  - **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?
    Groups of tasks can depend on each other in different ways (e.g., one group must complete before another can start), leading to different ordering constraints.
  - **Data Sharing**: Given a data and task decomposition for a problem, how is data shared among the tasks?
    While the range of possibilities for data sharing is almost endless, many cases fall into one of a few categories (e.g., read-only for all tasks, or read-write for one task and read-only for all others).

- **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?
  Many factors influence whether the design-in-progress is good enough to continue with — suitability for the target platform, flexibility, efficiency, and simplicity.

Nominally, the patterns are applied in the order listed above. In practice, however, it is often necessary to work back and forth between them, especially if Design Evaluation indicates that the design has major flaws.
2.3. 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. (Designers with more experience might even enter PLPP at this stage, bypassing explicit use of the Finding Concurrency patterns.) The Algorithm Structure patterns describe overall strategies for exploiting concurrency.

In other words, once potential concurrency has been identified, the next task is to refine the design by mapping this concurrency onto multiple units of execution (UEs) running on a parallel computer.

Of the countless ways to define an algorithm structure, most follow one of six basic patterns, which make up the Algorithm Structure design space. The key issue at this stage is to decide which pattern or patterns are most appropriate for the problem. In making this decision, various forces such as simplicity, portability, scalability, and efficiency may pull the design in different directions. The features of the target platform must also be taken into account.

There is usually a major organizing principle implied by the concurrency that helps choose a pattern. This usually falls into one of three categories:

- Organization by tasks:
  - **Task Parallelism**: How can an algorithm be organized around a collection of tasks that can execute concurrently?
    Key issues to be addressed are those related to managing the collection of tasks — how to define them, how to schedule them for execution on multiple units of execution, and how to and manage any data dependencies among them.
  - **Divide and Conquer**: Suppose the problem is formulated using the sequential divide and conquer strategy. How can the potential concurrency be exploited?
    Key issues to be addressed here are granularity of tasks, mapping of tasks to units of execution, and managing data dependencies.

- Organization by data decomposition:
  - **Geometric Decomposition**: How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”?
    The focus here is on how to manage a concurrent update (or repeated updates) of the data structure’s chunks. Key issues are how to partition the data, how to make sure that each task has access to all the data it needs for the update (which usually involves communication or synchronization among tasks), and how to map the data and computation to units of execution.
  - **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?
    Sometimes it is possible to restructure the operation in a way that exposes more concurrency than might be apparent at first (the pattern gives examples), usually in a way that involves repeatedly operating on all elements of the data structure concurrently.

- 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?

The key idea of this pattern is best expressed in terms of an assembly line, with the workers corresponding to pipeline stages and the items being assembled corresponding to sets of data.

– **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?

The key idea of this pattern is to express the data flow in terms of “events”, with each event having a task that generates it and a task that is to process it.

The most effective parallel algorithm design may make use of multiple algorithm structures (combined hierarchically, compositionally, or in sequence). For example, it often happens that the very top level of the design is a sequential composition of one or more Algorithm Structure patterns. Other designs may be organized hierarchically, with one pattern used to organize the interaction of the major task groups and other patterns used to organize tasks within the groups — for example, an instance of Pipeline in which individual stages are instances of Task Parallelism.

### 2.4. The Supporting Structures design space

This design space represents an intermediate stage between the Algorithm Structure and Implementation Mechanisms design spaces. The Finding Concurrency and Algorithm Structure design spaces focus on algorithm expression. At some point, however, algorithms must be translated into programs. The patterns in the Supporting Structures design space address that phase of the parallel program design process. We call these patterns Supporting Structures since they describe software constructions or "structures" that support the expression of parallel algorithms. The patterns fall into two groups:

- **Program-structuring patterns:**
  - **SPMD** (Single Program, Multiple Data): 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?

    With this pattern, the program is structured in terms of number of concurrently-executing entities, all executing the same source code but each having its own data and a unique identifier. Different entities may take different paths through the source code depending on their data and identifiers.

    MPI programs are usually examples of this pattern.

  - **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?

    With this pattern, the program is structured in terms of two logical elements, a master and one or more workers, with the master organizing the computation and partitioning work...
among workers. The master may be implicit (e.g., when workers share access to a “bag of tasks”).

- **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? With this pattern, the program is structured in terms of splitting iterations of each computationally intensive loop among units of execution. Key considerations include how to partition the iterations and assign them to UEs and how to manage data dependencies. OpenMP programs are usually examples of this pattern.

- **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?

  With this pattern, the program structure can vary; the issue to address is how to map tasks to units of execution, either indirectly (using a task pool, e.g.) or directly.

- **Patterns representing data structures:**

  - **Shared Data**: How does one explicitly manage shared data inside a set of concurrent tasks? This pattern presents a stepwise approach to managing shared data based on first defining an abstract data type and then determining a way to implement operations on this ADT in the simplest way that gives acceptable performance.

  - **Shared Queue**: How can concurrently-executing units of execution (UEs) safely share a queue data structure?

    This pattern is an example of applying the **Shared Data** pattern. Ideally parallel programs needing a shared queue would have access to a library class or functions; this pattern presents an implementation from scratch to illustrate the concepts.

  - **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?

    The idea behind this pattern is simple to state at a high level (split the array into contiguous subarrays and distribute them among UEs) but can be complicated to implement; the pattern discusses the ideas and some approaches to managing the complications.

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.

### 2.5. 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. The other three design spaces focus on designing algorithms and the high-level constructs used to organize parallel programs. This design space, in contrast, considers a program’s source code and the low-level operations used to write parallel programs. It is presented without the formalism of patterns because most of these implementation mechanisms are included within the major parallel programming environments. A complete and detailed discussion of these building blocks for parallel programs is beyond the scope of PLPP. Most parallel programmers,
however, use only a modest core subset of the full range of mechanisms. The Implementation Mechanisms design space identifies these core mechanisms and presents for each one a high-level description and then an investigation of how the mechanism maps onto three representative programming environments (OpenMP, MPI, and Java). These mechanisms fall into three categories:

- **UE† 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.

- **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.

3. **The Reengineering for Parallelism pattern**

This section contains the text of the new pattern; it follows the pattern format used for PLPP [20].

---

**Reengineering for Parallelism**

---

**Problem**

How can an existing application be reengineered to improve performance by making use of parallel hardware, and how can PLPP be used in the process?

**Context**

The lifespan of complex application programs is long compared to that 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 hardware, 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 approach for improving the performance of legacy applications was to buy faster hardware; porting the code to take advantage of parallel hardware (parallelizing the program) was viewed as something to be done only as a last resort. Current trends in hardware, however, focus less on making single processors faster and more on utilizing parallel architecture (SMP, multi-core, and clusters) to deliver greater performance. This means that the performance of legacy applications can no longer be increased simply by buying faster processors; with performance coming from parallel hardware, software engineers will have no choice but to parallelize their legacy applications if they need them to run faster.

Creating parallel software, however, is difficult, time-consuming, and error-prone. It is especially challenging for legacy applications. There is extensive literature on parallel programming, but most of it focuses on developing new applications intended from the beginning to execute on parallel hardware. The pattern language PLPP [20] is a good example of this approach. It embodies a design methodology for creating parallel software based on a web of design patterns that take programmers step by step from an understanding of their problem to a complete design for a parallel program. As with much of the literature in parallel programming, PLPP focuses on creating new parallel applications from scratch.

The issues in designing a parallel program are largely the same for legacy and new applications, so the patterns in PLPP are relevant for the programmer working to parallelize a legacy application. The overall methodology, however, is slightly different. In essence, the programmer enters PLPP from a slightly different starting point and must take pains to ensure that the new parallel application produces results in conformance with the original version of the legacy application. This pattern addresses these special concerns and provides a methodology for transforming a legacy application into a new parallel application using PLPP.

**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.
- Having a single code that works on both sequential and parallel systems reduces future maintenance effort, at a possible cost in added complexity.
- 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, for which the pattern language PLPP provides guidance.

This pattern provides guidance for (1) managing the parallelization 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. While these transformations ideally are small and localized to specific “hot spots”, in some cases (for example, in distributed-memory programs where large data structures must be redesigned as distributed data structures) the transformations may be complex and global in scope. Even when this is the case, however, the overall procedure described in this pattern is applicable; the steps to follow for each change are the same whatever its size and scope; and the emphasis on making changes in as disciplined a way as possible, and testing each change using a predefined testing protocol, is if anything more important for larger changes.

The next sections describe preparatory steps to be completed before considering any modifications to code, and then the process of modification (reengineering) itself.

Preparation

Several tasks should be completed before exploring the details of the program’s source 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 [5] (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. In this case 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 is incorrect, and prior to parallelization, a new, numerically correct algorithm must be found.

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 [8].

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. A concrete definition of the expected behavior of the application is needed. This is done by constructing a test suite to be used throughout the parallelization process. Test suite construction can be complicated [22]: 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.

In some cases, transformation will require major restructuring of the program or introduction of new distributed data structures. This is particularly the case when the target platform is a distributed-memory system using message passing. Even in this case, when working with a legacy application, it is important to implement the changes as a series of discrete transformation that can be tested at each step.

Dig deeper into the code. Dig deeper into the code for 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. Use 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 approach described in PLPP is still applicable.

In every case the programmer must decompose the problem in two dimensions, a task decomposition and a data decomposition, but frequently one decomposition will dominate the other. 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 easier to focus on the control structure itself (i.e., to first define a task decomposition, and then infer from that a data decomposition). For example, in 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. The Finding Concurrency patterns address these dual decompositions.

Once the major decomposition strategy has 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

‡These patterns are described briefly in Section 2 and presented in full in [20].
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 [20] 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 in mind the following.

- When choosing a program-structuring pattern, the full range 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 set of intended target platforms includes distributed-memory computers such as clusters, it is more common to use a message-passing API and the SPMD pattern. For a target platform with a shared address space but nonuniform memory access times (a NUMA or nonuniform memory access computer), SPMD may be preferable, even when the programmer intends to use OpenMP. The SPMD pattern forces the programmer to make decisions up front about how data is decomposed among the units of execution. This often results in a more optimal utilization of a complex memory hierarchy.

- The control structures in a program often dictate which Supporting Structures patterns to use. For programs dominated by well defined loops, Loop Parallelism is appropriate. But for programs with more complicated control structures, the more general SPMD pattern is best. SPMD, however, requires more-sweeping program changes than Loop Parallelism. Many of the data structures in an SPMD program must be distributed, replicated, or even completely redesigned. This makes using SPMD in legacy applications with the approach outlined here considerably more challenging. But for legacy applications, it is essential to be disciplined and execute the changes required by the SPMD pattern as a series of distinct transformations that can be independently validated.

For example, once a decision has been made that a particular data structure should be distributed in an SPMD program, eventually every computation that involves this data structure will need to be modified. This modification can be implemented incrementally in a legacy application by replicating the data structure as an intermediate step and having each UE carry out the associated computations redundantly. Then the program’s hot spots can be parallelized one by one, replacing the redundant computation on replicated data with computation the new distributed data. Eventually, all the redundant computation will be replaced, yielding the final parallel program.
Evaluate and debug. After each transformation, no matter how trivial, run the test suite and verify that the program meets the correctness criteria. Debugging will be much easier since only the last transformation in the sequence must be studied. Also pay attention to how the performance changes as larger portions of the program run in parallel. It is not unusual for hot spots earlier ruled out as insignificant to become crucial performance bottlenecks as the parallel fraction of the program increases.

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 algorithm as well as the performance targets. As discussed in PLPP, 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 the Design Evaluation pattern.

It is important to note that in some cases the changes required in the legacy application will be too extensive to support the incremental approach described in this pattern. Such applications can only be migrated to a parallel machine by extensively if not totally rewriting the program. Programmers parallelizing a legacy application must be very careful about deviating from the procedures described in this pattern. It is in the nature of legacy applications that knowledge about the internals of the code is incomplete. In some cases, the domain experts responsible for large portions of the code are no longer available. It is therefore worth considerable additional effort to use the incremental approach described in this pattern whenever practical to do so.

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 [10]; an earlier experiment with parallelizing it is described in [12].

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.

Preparation

Survey the landscape. We start by working through the list of questions posed in the Solution section above, omitting those that are not relevant for this particular example.

- Does the code seem to be well structured, or is it a BIG BALL OF MUD (a.k.a. spaghetti code)?
Figure 1. Simplified pseudocode for electromagnetics application.

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 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.
- What kind of testing was done to validate the program? Are the tests still available?
  Reportedly, the output of the program was compared with an analytic solution of the problem and found to be correct. The original test is available.
- 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:
  The program is a single self-contained test of its algorithm. Number of grid cells and number of time steps can be varied at compile time. The parallelized program should give the same results as the sequential code for the default values of these variables and also for whatever increased values are needed to demonstrate performance improvement.

- 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. 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.

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 the Data Sharing pattern: 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. The simplest approach is to perform all such communication before starting the loop. More sophisticated approaches in which computation and communication overlap may also be possible. In this initial analysis, we focus on the first approach because of its simplicity, but with a mental note that if it gives inadequate performance it may be desirable to try something more sophisticated. A library function or functions to perform a standard exchange of boundary values 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 should 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. Notice that with a 3D array there are several decomposition options: The array can be decomposed along any one of the dimension, or two, or all three. It is likely that the last option will give the best performance, since it requires less total data to be exchanged. As suggested in **Geometric Decomposition**, we will provide a mechanism for varying this at compile time (defining dimensions of a “processor grid” over which the arrays will be distributed) to allow easy experimentation with the different options. 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.

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.
// 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. Each process has a local section of the array, surrounded by a ghost boundary. The array bounds are expressed in terms of constants IXLO, etc., that specify the size of the local block of the array plus the ghost boundary. In this example, the stencil used in the finite difference operators requires a ghost boundary of a fixed size of 1. Hence IXLO = 0, IXHI = NXlocal+1, and so forth.

- 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 may be 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
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). The code is almost identical to the original serial code. Only three changes have been made: (1) the arrays are now local sections of the global arrays, as shown in Fig. 6; (2) the loop limits have been changed to run over the size of the local sections (excluding ghost boundaries of fixed size 1, as required by the stencil operation); and (3) the global coordinates required by complicated_function are obtained from the local coordinates using functions local_to_global_x, etc.

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 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 decreases in execution time as the number of processes increases.\textsuperscript{8} Further, as expected, decomposing the array along all three dimensions gives better performance than decomposing it along only one or two dimensions.

\textsuperscript{8}The code was timed on a cluster of 16 PCs with 3Ghz Pentium 4 processors, connected by a 100Mbit network and running Linux. Speedups (defined as execution time of the sequential program divided by execution time of the parallel program, using the best-performing decomposition of data for that number of processors) ranged from 1.49 for 2 processors to 4.5 for 16 processors.

Copyright © 2006 John Wiley & Sons, Ltd.
Concurrency Computat.: Pract. Exper. 2006; 00:0–0
Prepared using cpeauth.cls
// 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.

Repeat as needed. Although the parallel code met the original performance goals, given that these were modest, it makes sense to consider whether improvements are feasible. Performance testing was done on a cluster of workstations connected by a relatively slow network; timing the parts of the code that are affected by the slow network (the exchanges of boundary values and the reduction in output_selected_values()), it became apparent that each process was spending a significant fraction of its time performing exchanges of boundary values. Tuning the library routine performing the exchanges (by experimenting with different variants of MPI message-passing) proved to be of limited benefit.

Thus, at this point it makes sense to consider whether any of the patterns mention performance optimizations that were previously rejected in favor of simplicity of coding but which now should be considered again. One such optimization discussed in Geometric Decomposition and mentioned above is overlapping computation and communication. Many examples of overlapping computation
REENGINEERING FOR PARALLELISM: AN ENTRY POINT INTO PLPP

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.

and communication involve a single distributed array; in these examples, the overlap is produced by refactoring the update operation to first initiate the exchange of boundary values, then compute interior values, then wait for the exchange to complete, and finally compute boundary values. The application here involves updates of multiple distributed arrays (three for the components of the electric field and three for the components of the magnetic field), however, so we can get some overlap (and thus performance benefit) with substantially less refactoring of the application code by simply initiating a boundary exchange for each distributed array as soon as its values have been computed and then waiting for it to complete just before the next use of the array. This does require refactoring of the library routine for exchanging boundary values (splitting it into a routine to initiate the exchange and one to wait for it to complete). When the application and library code was revised in this way, the refactored code again met the correctness goals and showed better performance for larger numbers of processors.

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

\[\text{Related Patterns}\]

\[\begin{align*}
// \text{original code} \\
&\text{loop over } i \text{ in } 1 \ldots N_\text{X}-1 \\
&\quad \text{var1}(i) = f(i) \\
&\quad \text{var2}(i+1) = g(i) \\
&\text{end loop} \\

// \text{modified code} \\
&\text{loop over } i \text{ in } 1 \ldots N_\text{X}-1 \\
&\quad \text{var1}(i) = f(i) \\
&\text{end loop} \\
&\text{loop over } i \text{ in } 2 \ldots N_\text{X} \\
&\quad \text{var2}(i) = g(i-1) \\
&\text{end loop}
\end{align*}\]
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 [3, 5]. Other work dealing with refactoring legacy programs [6, 4, 9] 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 [20], and also in [2, 23, 11].

4. Conclusions and related work

Patterns and pattern languages have been developed for many domains, ranging from the first pattern language developed by Christopher Alexander for city planning and architecture [1] to the now classic “Gang of Four” book [7] containing patterns for reusable object-oriented software. There is even a pattern language about writing pattern languages [21]. Patterns specifically related to parallel programming and dealing with legacy applications were cited in the Related Patterns section of the Reengineering for Parallelism pattern (in Section 3) itself.

The goal of a pattern language is not to present entirely novel solutions to problems, but to disseminate good solutions known to experts and to provide a vocabulary for talking about these solutions. Pattern languages have proved valuable for bringing order to object-oriented programming and establishing common practice and vocabulary across the community. The impact of the Gang of Four book [7] on object-oriented programming can be easily seen by perusing, for example, the documentation of the Java API java.sun.com/j2se/1.5.0/docs/api/, where references to the GoF patterns are frequent.

For parallel programming, PLPP is a pattern language that provides a model of how programmers reason about parallel programming. There is great value in making this model explicit, both to help parallel programmers work more effectively, and also so that parallel programming researchers can debate, test, and work with us to refine this model and the pattern language that describes it. PLPP was developed over the course of many years with the help of valuable feedback from others at the PLoP conferences and other workshops. Indeed, comments from readers of [20] inspired us to develop the pattern presented here, which extends PLPP to the case of legacy applications. For too long, parallel programming research has focused on new languages or alternative APIs with far too little attention paid to the software design process as applied to parallel programming. Pattern languages have proved valuable for bringing order to object-oriented programming and establishing common practice across the community. PLPP is an attempt to do the same for parallel programming, and the pattern given in this paper is a next step in its evolution.

ACKNOWLEDGEMENTS

We gratefully acknowledge the help of Brian Foote, who made many helpful suggestions about the PLoP paper on which this paper was based.

REFERENCES


7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


