Regular contribution

Parallel programming with a pattern language

Berna L. Massingill\textsuperscript{1,*}, Timothy G. Mattson\textsuperscript{2}, Beverly A. Sanders\textsuperscript{1}

\textsuperscript{1}Department of Computer and Information Sciences, University of Florida, FL 32611, USA; E-mail: sanders@cise.ufl.edu
\textsuperscript{2}Parallel Algorithms Laboratory, Intel Corporation, Dupont WA 98327, USA; E-mail: timothy.g.mattson@intel.com

Published online: 8 February 2001 – © Springer-Verlag 2001

Abstract. A design pattern is a description of a high-quality solution to a frequently occurring problem in some domain. A pattern language is a collection of design patterns that are carefully organized to embody a design methodology. A designer is led through the pattern language, at each step choosing an appropriate pattern, until the final design is obtained in terms of a web of patterns. This paper describes a pattern language for parallel application programs aimed at lowering the barrier to parallel programming by guiding a programmer through the entire process of developing a parallel program. We describe the pattern language, present two example patterns, and sketch a case study illustrating the design process using the pattern language.

Key words: Design patterns – Parallel programming

1 Introduction

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

A design pattern describes, in a prescribed format, a high-quality solution to a frequently occurring problem in some domain. The format is chosen to make it easy for the reader to quickly understand both the problem and the proposed solution. Because the pattern has a name, a collection of patterns provides a vocabulary with which to talk about these solutions.

A structured collection of design patterns is called a pattern language. A pattern language is more than just a catalog of patterns: the structure of the pattern language is chosen to lead the user through the collection of patterns 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. (In spite of the overlapping terminology, a pattern language is not a programming language.)

This paper describes a pattern language for parallel application programs. The current state of the pattern language can be viewed at \url{http://www.cise.ufl.edu/research/ParallelPatterns}. The pattern language is extensively hyperlinked, allowing the programmer to work through it by following links. The goal of the pattern language is to lower the barrier to parallel programming by guiding a programmer through the entire process of developing a parallel program. The main target audience is experienced programmers who may lack experience with parallel programming. The programmer brings to the process a good understanding of the actual problem to be solved and then works through the pattern language, eventually obtaining a detailed parallel design or perhaps working code.
In this paper, we first give an overview of the organization of the pattern language. We then present the text of two of the patterns, followed by a simple case study illustrating the design process using the pattern language. We close with brief descriptions of related approaches.

2 Organization of the pattern language

The pattern language is organized into four design spaces — FindingConcurrency, AlgorithmStructure, SupportingStructures, and ImplementationMechanisms — which form a linear hierarchy, with FindingConcurrency at the top and ImplementationMechanisms at the bottom.

2.1 The FindingConcurrency 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. There are three major design patterns in this space:

- **DecompositionStrategy.** This pattern addresses the question of how to decompose the problem into parts that can execute simultaneously. It and related patterns (TaskDecomposition and DataDecomposition) discuss the two major strategies for decomposing problems — task-based decomposition and data-based decomposition — and help the programmer select an appropriate strategy based on one or the other or a combination of both.

- **DependencyAnalysis.** Once the programmer has identified the entities into which the problem is to be decomposed, this pattern helps him or her understand how they depend on each other. These dependencies include both ordering constraints and data dependencies; this pattern and related patterns (GroupTasks, OrderTasks, and DataSharing) help the programmer analyze these dependencies in detail.

- **DesignEvaluation.** This pattern is a consolidation pattern; it is used to evaluate the results of the other patterns in this design space and prepare the programmer for the next design space, the AlgorithmStructure design space.

Figure 1 illustrates the relationships among patterns in this design space. In the figure, arrows indicate the direction in which the programmer moves between patterns; double-headed arrows indicate that it may be necessary to move back and forth between patterns as the analysis proceeds.

2.2 The AlgorithmStructure design space

This design space is concerned with structuring the algorithm to take advantage of potential concurrency. That is, the designer working at this level reasons about how to use the concurrency exposed in the previous level. Patterns in this space describe overall strategies for exploiting concurrency. Selected patterns in this space have been described in [17].

Patterns in this design space can be divided into the following three groups, plus the ChooseStructure pattern, which addresses the question of how to use the analysis performed by using the FindingConcurrency patterns to select an appropriate pattern from those in this space. A key part of the ChooseStructure pattern is the figure included here as Fig. 2; it illustrates the decisions involved.

2.2.1 “Organize by ordering” patterns

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

- **PipelineProcessing.** The problem is decomposed into ordered groups of tasks connected by data dependencies.

- **AsynchronousComposition.** The problem is decomposed into groups of tasks that interact through asynchronous events.

2.2.2 “Organize by tasks” patterns

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

- **EmbarrassinglyParallel.** The problem is decomposed into a set of independent tasks. Most algorithms based on task queues and random sampling are instances of this pattern.

- **SeparableDependencies.** The parallelism is expressed by splitting up tasks among units of execution (threads or processes). Any dependencies between tasks can be pulled outside the concurrent execution by replicating the data prior to the concurrent execution and then combining the replicated data after the concurrent execution. This pattern applies when variables involved in data dependencies are written but not subsequently read during the concurrent execution.

- **ProtectedDependencies.** The parallelism is expressed by splitting up tasks among units of execution. In this case, however, variables involved in data dependencies are both read and written during the concurrent execution; thus, they cannot be pulled outside the concurrent execution but must be managed during the concurrent execution of the tasks.