OVERVIEW OF A PATTERN LANGUAGE FOR PARALLEL PROGRAMMING[1]

Beverly A. Sanders[2], Timothy. G. Mattson[3], and Berna L. Massingill[4]

Introduction

A *design pattern* describes a good solution to a recurring problem. A pattern has a name and includes, at a minimum, the problem and its context, the forces or tradeoffs that must be addressed by the solution, and a proven solution to the problem. Generally, patterns are presented in a prescribed format (there are several formats in common use) to make it easier for the reader to quickly identify appropriate patterns. A *pattern language* is an organized collection of patterns that deal with problems in some domain. It is called a language, because the names of the patterns provide a vocabulary for talking about the domain. The patterns in a language may be related in various ways, including hierarchically or compositionally, and the structure of the language helps the designer determine how to apply the patterns. Pattern languages are important because they document and communicate expertise so that it can be utilized by others.

In this note, we give a brief overview of a pattern language for parallel programming[5]. Most large computational problems contain exploitable concurrency. This means that a program to solve the problem can be structured so that different parts of the problem can be solved simultaneously on multiple processors, allowing the problem to be solved in less time and/or enabling bigger problems to be solved than with a single processor. The difficulty is finding this exploitable concurrency and then constructing a program to efficiently exploit it. Our pattern language is designed to help with the entire process of constructing a parallel program, from the high-level design, to obtaining working code.

The pattern language is organized into four design spaces, *Finding Concurrency*, *Algorithm Structure*, *Supporting Structures*, and *Implementation Mechanisms* as shown in the figure.

Before starting to work with the patterns language, the algorithm designer should 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 the results sufficiently significant, to justify expending effort to solve it faster? If so, the next step is to make sure the key features and data elements within the problem 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

---

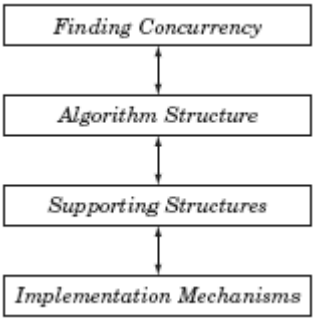
[1]Abstract of invited talkat 4th Congress on Systems Engineering, Veracruz, Mexico. August 2004.

[2]Department of Computer and Information Sciences, University of Florida, Gainesville, FL 32611-6120. sanders@cise.ufl.edu

[3] Intel Corporation. timothy.g.mattson@intel.com

[4] Department of Computer Science, Trinity University, San Antonia, TX. bmassing@cs.trinity.edu

[5] A complete description of the pattern language is given in Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley. 2004. Also, see www.cise.ufl.edu/research/ParallelPatterns



patterns in the *Finding Concurrency* design space can be used to identify and analyze the exploitable concurrency in the problem. The patterns in the *Algorithm Structure* design space describe strategies for mapping the concurrency onto processes or threads. The *Supporting Structures* design space contains two groups of patterns: those that address ways to organize code, and those that represent commonly used shared data structures. The *Implementation Mechanisms* design space is concerned with how the patterns of the higher-level spaces map onto the facilities of a particular parallel programming environment. In the rest of this note, we will briefly describe these design spaces in more detail.

The Finding Concurrency Design Space

The patterns in this design space are used in the high-level analysis of the problem to find exploitable concurrency. They are organized into three groups.

- **Decomposition Patterns** are used to decompose the problem into pieces that can execute concurrently. The *Task Decomposition* pattern addresses how to decompose a problem into tasks[6], while the *Data Decomposition* pattern deals with decomposing a problem's data into chunks that can be operated on relatively independently.
- **Dependency Analysis Patterns** are used to group the tasks together and analyze dependencies among them. The *Group Tasks* pattern considers how to group the tasks together to make handling dependencies easier. The *Order Tasks* pattern deals with ordering constraints on the task groups. The *Data Sharing* pattern helps the designer analyze the way that data is shared between tasks.
- ***Design Evaluation*** is the final pattern in the *Finding Concurrency* space. This pattern leads the designer through an analysis of the design so far. If the design is good enough, it is time to move to the *Algorithm Structure* design space. Otherwise, the designer will need to revisit some of the decisions already made. A goal of this pattern is to help the designer identify problems early in the design process where they are easier to correct.

The Algorithm Structure Design Space

After analyzing the concurrency in a problem, the next task is to refine the design and move it closer to a program by mapping the concurrency onto multiple processes or threads. Of the countless ways to define an algorithm structure, most follow one of the six basic design patterns that make up the Algorithm Structure design space. Many parallel algorithm designs make use of multiple algorithm structures combined hierarchically, compositionally, or in sequence. One tries to identify a part of the algorithm where there is a major organizing principle implied by the concurrency. This major organizing principle usually falls into one of the three categories listed below and it can be used to help select an appropriate algorithm structure pattern.

---

[6] A task is a sequence of instructions that operate together as a group and corresponds to some logical part of an algorithm or program.

- **Organize by tasks**. The *Task Parallelism* pattern is used when the tasks are linear. The pattern helps the designer handle map tasks to processes or threads and deal with dependencies between the tasks. An important special case of *Task Parallelism* is the situation where the tasks are completely independent. Such problems are called *embarrassingly parallel.* The *Divide and Conquer* pattern applies when the tasks are recursive, for example, when tasks correspond to the function calls in a sequential divide-and-conquer algorithm.
- **Organize by data decomposition**. The *Geometric Decomposition* pattern is likely to be relevant when the problem data is decomposed into discrete chunks, where the solution for each chunk can be solved using data from only a few other chunks. The pattern helps the user organize the computation so that the data is efficiently communicated when and where it is needed. The *Recursive Data* pattern is typically the best choice when the problem involves following links through a recursive data structure.
- **Organize by flow of data**. These patterns apply when the major organizing principle is how the flow of data imposes an ordering on the groups of tasks. When the ordering is regular, one-way, and static, the *Pipeline* pattern is probably a good choice. When the flow of data is irregular or dynamic, the *Event-Based Coordination* pattern should be considered. It gets its name because we can think of the arrival of data as an event.

The Supporting Structures Design Space

This design space represents an intermediate stage between the *Algorithm Structure* and *Implementation Mechanisms* design spaces. These patterns fall into two groups

- **Program structuring** patterns help organize the program code. These include the following patterns. In the *SPMD* (single program multiple data) pattern, each process or thread executes the same code using process or thread identifiers for control decisions. *Master/Worker* allows the load to be dynamically balanced among the threads or processes by putting a "master" in charge of tasks that are allocated to "workers" to be executed. Often, this is implemented by associating the master with a task queue from which workers retrieve a new task to execute whenever they are free. To give one example of how the patterns in different design spaces are related, *Master/Worker* is frequently used in the implementation of instances of *Task Parallelism*. It, in turn may use an instance of *Shared Queue*. The *Loop Parallelism* pattern describes how to translate a serial program whose runtime is dominated by a set of computationally intensive loops into a parallel program by distributing the loop iterations to multiple processes or threads. The *Fork/Join* pattern deals with the situation where the number of concurrent tasks varies as the program executes and the way the tasks are related prevents the use of simpler control structures such as parallel loops.
- **Shared data structures** are common in parallel programs and several patterns in the language provide advise for dealing with these. The *Distributed Array* pattern deals with partitioning the large arrays that are common in scientific computations. The *Shared Queue* pattern discusses a data structure that is often used in the implementation of instances of the *Master/Worker*, *Pipeline*, and *Event-Coordination* patterns. The *Shared Data* pattern provides solutions for the general case of shared data.

Implementation Mechanism Design Space

Up to this point, we have focused on designing algorithms and the high-level constructs used to organize parallel programs. With this design space, we shift gears and consider a program's source code and the low-level operations used to write parallel programs. Most of the implementation mechanisms are included within the major parallel programming environments. Hence, rather than continuing to use the formalism of patterns, we provide a high-level description of each implementation mechanism and then an investigation of how the mechanism maps onto three target programming environments: OpenMP, MPI, and Java. A complete and detailed discussion of these parallel programming ``building blocks" would fill a large book. Fortunately, most parallel programmers use only a modest core subset of these mechanisms. These core implementation mechanisms fall into three categories:

- **Management of Processes and Threads**, and in particular, their creation and destruction, is necessary since concurrent execution, by its nature requires multiple entities that run at the same time.
- **Synchronization** is used to enforce constraints on the order of events occurring in different processes or threads. Important synchronization constructs include memory fences, barriers, and mutual exclusion.
- **Communication** allows executing processes or threads to exchange information. When memory is not shared between them, this exchange occurs through an explicit communication event. The major types of communication events are message passing, and collective communication. The latter involves communication between a group of threads or processes, often all that are participating in a computation.

Summary

The pattern language for parallel programming described here provides several benefits by providing a catalog of good solutions to important problems, an expanded vocabulary, and a methodology for the design of parallel programs. We hope to lower the barrier to parallel programming by providing guidance through the entire process of developing a parallel program. In the longer term, we hope that this pattern language can provide a basis for both a disciplined approach to the qualitative evaluation of different programming models and the development of parallel programming tools.