Introduction to Parallel Design Patterns

Introduction

Many programmers know the notion of a *design pattern* from the famous "Gang of Four" book [Gamma et al, 1994], and it's natural to seek design patterns that especially apply to the unfamiliar technologies of parallel programming. A good design pattern captures the essence of a planning unit for code that a programmer thinks in terms of when solving a programming problem. A programmer must think at various levels when problem solving, from high levels that are near the domain of the problem being solved to low levels near the hardware that will run the completed application, and patterns capture reusable programming strategies at all levels.

Some in the parallel design patterns community have gone beyond merely assembling catalogs of patterns, and are actively developing a *methodology* for solving programming problems, whether or not those problems require parallel computation [Mattson, 2011]. This methodology seeks a structure that mirrors the types of models programmers use, and one solves programing problems in that methodology by traversing a web of connected patterns that reflect the problem-solving strategies programmers naturally use to create programs.

Example of the patterns methodology

A *pattern language* is a structured catalog of design patterns, together with a design methodology based on a web of connected patterns. We will now illustrate a pattern-based design methodology through the following example.

Using parallel patterns to solve a numerical integration problem

Suppose we intend to use numerical integration to estimate the constant pi in terms of the area of a semicircle. This technique is explained in the module "Computing Pi as an Area." For now, we will summarize that explanation by saying that we will approximate the area of the blue semicircle in Figure 1 by adding up the areas of rectangles (such as the green one in Figure 1) that overlap portions of that semicircle.

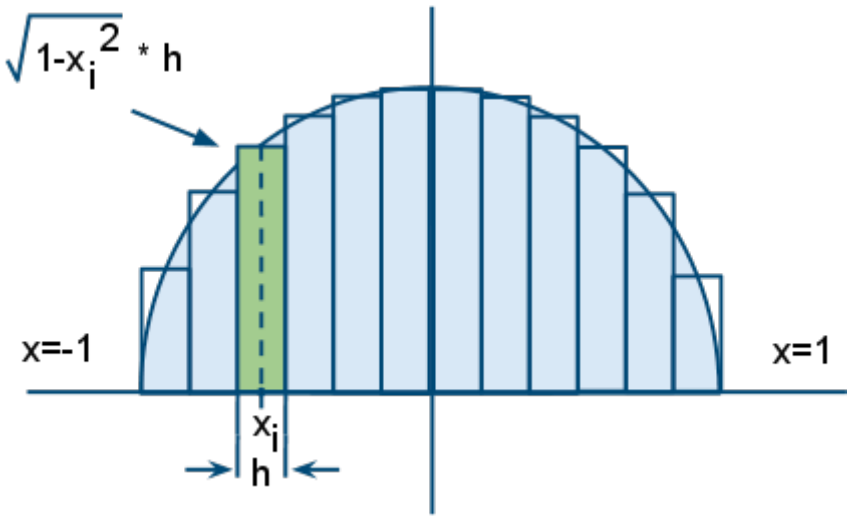

Figure 1: Illustration of numerical integration technique.

In order to estimate pi by this method, we plan to write a program to compute the sum of the areas of all the rectangles. For better accuracy, we will use very many rectangles instead of the small number shown in the figure.

Observe that the program will apply the same computational operations (yielding the area of a rectangle) to a large number of data values (such as loop indices). Note that those computational operations are independent of each other for different data values. Because of this independence, we can write parallel programs to speed up performance, using various parallel computing technologies.

That programming situation – applying the same independent computational operations to different data values – arises quite often, and indicates an opportunity for parallelism. Thus, it is an example of a recurring *parallel programming pattern*, i.e., a time-tested solution to a recurring problem in a well-defined parallel programming context. Most parallel programmers would refer to this pattern as *data parallelism.* If a programmer knows that and other patterns that signal opportunities for parallelism, he or she can apply them productively to write fast code that takes advantage of parallel platform resources such as multicore CPU design, general purpose GPU programming, and cloud cluster computing.

A number of authors have written about patterns for parallel programming for over a decade, and many such patterns have been described. The field continues to develop, and some comprehensive efforts have begun that seek both to cataloging patterns and to use them for designing computational problem solutions. We will give an overview of one of these systems later in this module, called the OPL system.

Here are some further patterns that help us program the numerical integration problem effectively. In this introduction, we will not give full descriptions of the

patterns we cite; see the "Patterns Catalog" module or the patterns wiki [Parlab] for more complete descriptions.

Notes:

- This font convention indicates the name of an OPL pattern.
- OpenMP, TBB, and MPI are libraries or systems for carrying out parallel computations in programming languages such as C++.
- *Threads* are features for taking advantage of multi-core processing, and a *GPGPU (General Purpose Graphics Processing Unit)* is specialized "graphics card" hardware that can be programmed for parallel computations other than graphics.

- We must visit each data value for our data parallelism computational strategy. Using a simple loop to traverse all the values illustrates the Loop parallelism pattern. Since this approach appears so often in parallel computations, various parallel computing technologies for multi-core systems provide features to support this pattern explicitly, such as OpenMP's omp parallel for pragma and TBB's parallel_for template. Behind the scenes, these features manage threads and distribute work load for convenient, correct parallelization without requiring the programmer to program that parallelism.
- If a programmer does explicitly code with threads on a multi-core platform, the Fork-join pattern represents a typical way to manage those threads that works well with data parallelism. For this problem, we could divide the range of all rectangles up into subsets, start (fork) multiple threads that each add up the areas for one of those subsets, and wait for those threads to finish their work (join), at which point the main thread can add up the subtotals of areas. Threads packages provide support for these operations, e.g., a call (often called join()) for performing that wait operation.
- The Message passing pattern provides send and receive operations for communicating data values between nodes in a distributed system. If we plan to use a distributed platform for our numerical integration computation, a master node can send each computation node the parameters that node needs to sum the areas of a subset of all the rectangles. The master node can then perform receive operations for each of the computation nodes that block until those computation nodes each complete their summations and send their subtotals. This accomplishes the same sort of computational strategy as the Fork-join pattern, but on a distributed system. The standard MPI system implements send/receive and many other message-passing features.

- A GPU is capable of executing instructions on numerous individual data values simultaneously in instruction-by-instruction synchronization. Such features support the Strict data parallel and SIMD (single instruction multiple data) patterns for parallel programming.

Patterns by any other names…

The names of patterns do not mean the same things to all people. For example, some authors broaden the meaning of "data parallelism" to mean processing different data at the same time, and consider the term to apply to nearly all forms of parallel computing. Other authors provide narrower definitions than we described above. In fact, the OPL group reserve the name Data parallel for applying the same independent computational operations to different data values *that are elements of a data structure*, which doesn't necessarily apply to our numerical integration problem.

In spite of this lack of uniformity about definitions, learning about patterns has great value for a parallel programmer. They capture reusable units of strategy that experts use to solve parallel programming problems, and as such, they provide programmers new to parallelism with a way to acquire their own expertise. Also, parallel computing technologies increasingly provide direct support for one or more patterns, as indicated above. Thus, patterns can serve as guidelines for effectively and efficiently developing and implementing parallel programs.

An overview of OPL patterns

In this segment, we list some patterns selected from the OPL hierarchy of patterns that are especially common and/or relevant to examples in our modules. OPL stands for "Our Pattern Language." Here, "pattern language" does *not* mean a *programming* language: instead, as explained in [Mattson, 2011], "pattern language" is a historical term that means a catalog of patterns together with a methodology for applying those patterns to solve problems. For a more complete treatment, see [Kreutzer and Mattson, 2009], the Berkeley patterns wiki [Parlab], and the book *Patterns for Parallel Programming* [Mattson et al, 2004].

Levels of parallel design patterns

The OPL design patterns are organized into four levels, from high level (close to the application) to low level (close to the machine). We will introduce these levels here,

and list several of the most common OPL patterns in each level. See the Patterns Catalog module for description of the individual patterns.

1. **Structural and computational patterns (Application architecture level)**

   This highest level in our hierarchy of patterns concerns the architectural design of large software.

   Note that some application architecture patterns may not lead to parallelism; for example, the Finite state machine computational pattern often cannot be programmed effectively in parallel.

   Structural patterns.

   These patterns describe the overall organization of a software application, and how an application's computational patterns interact.

   - *Structural patterns:* Event-based, implicit invocation; Map-reduce; Model-view-control; Iterative refinement; Real-time process control; Pipe and filter

   Computational patterns. These patterns describe the essential classes of computations that make up an application

   - *Computational patterns:* Dynamic programming; Finite state machine; Monte Carlo methods; Sparse linear algebra; Dense linear algebra; N-body methods; Spectral methods

2. **Parallel algorithm strategy patterns**

   These patterns define high-level strategies to exploit the concurrency in a computation for execution on a parallel computer.

   - Task parallel; Recursive splitting; Pipeline; Geometric decomposition; Data parallel

3. **Implementation strategy patterns**

   This category consists of two types of pattern.

Program structure patterns. These patterns focus on organizing a program.

- *Program structure patterns:* SPMD; Strict data-parallel; Fork-join; Master-worker; Loop parallel; Actors; BSP; Task queue

Data structure patterns. These patterns describe common data structures specific to parallel programming.

- *Data structure patterns:* Shared array; Distributed array; Shared queue; Shared map; Shared data

4. **Concurrent execution patterns**

Two types of patterns make up this lowest level in the patterns hierarchy

Advancing program counters patterns. These patterns concern the timing relationships for executing instructions among different threads or processes.

- *Advancing program counters:* MIMD; SIMD; Thread pool; Data flow

Coordination patterns. These patterns provide mechanisms for processes or threads to correctly access the data they need (cf. interprocess communication).

- *Coordination:* Message passing; Collective communication; Mutual exclusion; Transactional memory

Readings for further study

- **Early works on patterns in computing**
  - [Gamma et al, 1994] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison Wesley, 1994
    The "Gang of Four" book popularized patterns for computing, focusing on patterns for object-oriented programming
  - [Lea, 1999] Concurrent Programming in Java™: Design Principles and Pattern (2nd Edition), Doug Lea, Prentice Hall, 1999.

Doug Lea's work led to the highly regarded java.util.concurrency package

- **OPL resources**

  - [Kreutzer and Mattson, 2009] A design pattern language for engineering (parallel) software, K. Kreutzer and T. G. Mattson, Intel Technology Journal, Vol. 13, Issue 4, pp. 6-19, 2009.
    Articulation of OPL goals and philosophy

  - [Mattson et al, 2004] Patterns for Parallel Programming, Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill. Addison Wesley, Design Patterns series, 2004.
    Provides a catalog of parallel programming patterns and goes on to present a design methodology for using those patterns to develop parallel programs. Contributes lower levels of OPL

  - [Parlab] Berkeley ParLab patterns repository: http://parlab.eecs.berkeley.edu/wiki/patterns
    Wiki presenting all patterns of OPL, organized hierarchically for problem solving

  - [Mattson, 2011] A historical overview of design patterns for parallel programming, by Timothy G. Mattson, white paper, 2011. https://docs.google.com/document/d/1EBoBRw7gUAH_hdEiH9hp5V6_YpnyIUs-qEDHiITpNHY/edit?hl=en_US
    Background on the design patterns movement.

- **Other recent/ongoing works on patterns**

  - [UIUC] UIUC patterns links: http://www.cs.uiuc.edu/homes/snir/PPP/

  - [ParaPLoP] http://www.upcrc.illinois.edu/workshops/paraplop11/index.html

  - [McCool] Structured Parallel Programming with Deterministic Patterns http://www.usenix.org/event/hotpar10/tech/full_papers/McCool.pdf

  - [Campbell et al, 2010] Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore

Architectures (Patterns & Practices), Colin Campbell, Ralph Johnson, Ade Miller, Stephen Toub. Microsoft Press, 2010

- [Campbell and Miller, 2011] A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns & Practices), Colin Campbell, Ade Miller. Microsoft Press, 2011

Acknowledgment

Materials and several conversations with Tim Mattson, Intel Corporation have been indispensible for this project. Thanks also to Jay Petersen '13, St. Olaf College; Libby Shoop, Macalester College; Matt Wolf, Georgia Tech and Oak Ridge National Labs; Michael Wrinn, Intel Corporation; Michael Heroux, Sandia Labs and St. John's University.

Funding provided by Intel Corporation, the National Science Foundation (DUE-0942190), and St. Olaf College.