A DESIGN PATTERN LANGUAGE FOR ENGINEERING (PARALLEL) SOFTWARE

Contributors

Kurt Keutzer
UC Berkeley

Tim Mattson
Intel Corporation

Abstract

The key to writing high-quality parallel software is to develop a robust software design. This applies not only to the overall architecture of the program, but also to the lower layers in the software system where the concurrency and how it is expressed in the final program is defined. Developing technology to systematically describe such designs and reuse them between software projects is the fundamental problem facing the development of software for tera-scale processors. The development of this technology is far more important than programming models and their supporting environments, since with a good design in hand, most any programming system can be used to actually generate the program's source code.

In this article, we develop our thesis about the central role played by the software architecture. We show how design patterns provide a technology to define the reusable design elements in software engineering. This leads us to the ongoing project centered at UC Berkeley's Parallel Computing Laboratory (Par Lab) to pull the essential set of design patterns for parallel software design into a Design Pattern Language. After describing our pattern language, we present a case study from the field of machine learning as a concrete example of how patterns are used in practice.

Index Words

Design Pattern Language
Software Architecture
Parallel Algorithm Design
Application Frameworks

The Software Engineering Crisis

The trend has been well established [1]: parallel processors will dominate most, if not every, niche of computing. Ideally, this transition would be driven by the needs of software. Scalable software would demand scalable hardware and that would drive CPUs to add cores. But software demands are not driving parallelism. The motivation for parallelism comes from the inability of integrated circuit designers to deliver steadily increasing frequency gains without pushing power dissipation to unsustainable levels. Thus, we have a dangerous mismatch: the semiconductor industry is banking its future on parallel microprocessors, while the software industry is still searching for an effective solution to the parallel programming problem.

The parallel programming problem is not new. It has been an active area of research for the last three decades, and we can learn a great deal from what has *not* worked in the past.

- *Automatic parallelism*. Compilers can speculate, prefetch data, and reorder instructions to balance the load among the components of a system. However, they cannot look at a serial algorithm and create a different algorithm better suited for parallel execution.
- *New languages*. Hundreds of new parallel languages and programming environments have been created over the last few decades. Many of them are excellent and provide high-level abstractions that simplify the expression of parallel algorithms. However, these languages have not dramatically grown the pool of parallel programmers. The fact is, in the one community with a long tradition of parallel computing (high-performance computing), the old standards of MPI [2] and OpenMP [3] continue to dominate. There is no reason to believe new languages will be any more successful as we move to more general-purpose programmers; i.e., it is not the quality of our programming models that is inhibiting the adoption of parallel programming.

"It is not the quality of our programming models that is inhibiting the adoption of parallel programming."

The central cause of the parallel programming problem is fundamental to the enterprise of programming itself. In other words, we believe that our challenges in programming parallel processors point to deeper challenges in programming software in general. We believe the only way to solve the programming problem in general is to first understand how to architect software. Thus, we feel that the way to solve the parallel programming problem is to first understand how to architect parallel software. Given a good software design grounded in solid architectural principles, a software engineer can produce high-quality and scalable software. Starting with an ill-suited sense of the architecture for a software system, however, almost always leads to failure. Therefore, it follows that the first step in addressing the parallel programming problem is to focus on software architecture. From that vantage point, we have a hope of choosing the right programming models and building the right software frameworks that will allow the general population of programmers to produce parallel software.

"Given a good software design grounded in solid architectural principles, a software engineer can produce high-quality and scalable software."

In this article, we describe our work on software architecture. We use the device of a pattern language to write our ideas down and put them into a systematic form that can be used by others. After we present our pattern language [4], we present a case study to show how these patterns can be used to understand software architecture.

Software Architecture and Design Patterns

Productive, efficient software follows from good software architecture. Hence, we need to better formalize how software is architected, and in order to do this we need a way to write down architectural ideas in a form that groups of programmers can study, debate, and come to consensus on. This systematic process has at its core the peer review process that has been instrumental in advancing scientific and engineering disciplines.

"Design patterns give names to solutions to recurring problems that experts in a problem-domain gradually learn and take for granted."

The prerequisite to this process is a systematic way to write down the design elements from which an architecture is defined. Fortunately, the software community has already reached consensus on how to write these elements down in the important work *Design Patterns* [5]. Our aim is to arrive at a set of patterns whose scope encompasses the entire enterprise of software development from architectural description to detailed implementation.

Design Patterns

Design patterns give names to solutions to recurring problems that experts in a problem-domain gradually learn and take for granted. It is the possession of this tool-bag of solutions, and the ability to easily apply these solutions, that precisely defines what it means to be an expert in a domain.

For example, consider the *Dense-Linear-Algebra* pattern. Experts in fields that make heavy use of linear algebra have worked out a family of solutions to these problems. These solutions have a common set of design elements that can be captured in a *Dense-Linear-Algebra* design pattern. We summarize the pattern in the sidebar, but it is important to know that in the full text to the pattern [4] there would be sample code, examples, references, invariants, and other information needed to guide a software developer interested in dense linear algebra problems.

Computational Pattern: Dense-Linear-Algebra
Solution: A computation is organized as a sequence of arithmetic expressions acting on dense arrays of data. The operations and data access patterns are well defined mathematically so data can be pre-fetched and CPUs can execute close to their theoretically allowed peak performance. Applications of this pattern typically use standard building blocks defined in terms of the dimensions of the dense arrays with vectors (BLAS level 1), matrix-vector (BLAS level 2), and matrix-matrix (BLAS level 3) operations.

The *Dense-Linear-Algebra* pattern is just one of the many patterns a software architect might use when designing an algorithm. A full design includes high-level patterns that describe how an application is organized, mid-level patterns about specific classes of computations, and low-level patterns describing specific execution strategies. We can take this full range of patterns and organize them into a single integrated pattern language — a web of interlocking patterns that guide a designer from the beginning of a design problem to its successful realization [6, 7].

To represent the domain of software engineering in terms of a single pattern language is a daunting undertaking. Fortunately, based on our studies of successful application software, we believe software architectures can be built up from a manageable number of design patterns. These patterns define the building blocks of all software engineering and are fundamental to the practice of architecting parallel software. Hence, an effort to propose, argue about, and finally agree on what constitutes this set of patterns is the seminal intellectual challenge of our field.

"A full design includes high-level patterns that describe how an application is organized, mid-level patterns about specific classes of computations, and low-level patterns describing specific execution strategies."

Our Pattern Language

Software architecture defines the components that make up a software system, the roles played by those components, and how they interact. Good software architecture makes design choices explicit, and the critical issues addressed by a solution clear. A software architecture is hierarchical rather than monolithic. It lets the designer localize problems and define design elements that can be reused in other architectures.

"It is important to appreciate that many design problems do not lend themselves to a top-down or bottom-up analysis."

The goal of Our Pattern Language (OPL) is to encompass the complete architecture of an application from the structural patterns (also known as architectural styles) that define the overall organization of an application [8, 9] to the basic computational patterns (also known as computational motifs) for each stage of the problem [10, 1], to the low-level details of the parallel algorithm [7]. With such a broad scope, organizing our design patterns into a coherent pattern language was extremely challenging.

Our approach is to use a layered hierarchy of patterns. Each level in the hierarchy addresses a portion of the design problem. While a designer may in some cases work through the layers of our hierarchy in order, it is important to appreciate that many design problems do not lend themselves to a top-down or bottom-up analysis. In many cases, the pathway through our patterns will be to bounce around between layers with the designer working at whichever layer is most productive at a given time (so called, opportunistic refinement). In other words, while we use a fixed layered approach to organize our patterns into OPL, we expect designers will work though the pattern language in many different ways. This flexibility is an essential feature of design pattern languages.

As shown in Figure 1, we organize OPL into five major categories of patterns. Categories 1 and 2 sit at the same level of the hierarchy and cooperate to create one layer of the software architecture.

1. Structural patterns: Structural patterns describe the overall organization of the application and the way the computational elements that make up the application interact. These patterns are closely related to the architectural styles discussed in [8]. Informally, these patterns correspond to the "boxes and arrows" an architect draws to describe the overall organization of an application. An example of a structural pattern is *Pipe-and-Filter*, described in the sidebar.
2. Computational patterns: These patterns describe the classes of computations that make up the application. They are essentially the thirteen motifs made famous in [10] but described more precisely as patterns rather than simply computational families. These patterns can be viewed as defining the "computations occurring in the boxes" defined by the structural patterns. A good example is the *Dense-Linear-Algebra* pattern described in an earlier sidebar. Note that some of these patterns (such as *Graph-Algorithms* or *N-Body-Methods*) define complicated design problems in their own right and serve as entry points into smaller design pattern languages focused on a specific class of computations. This is yet another example of the hierarchical nature of the software design problem.

Structural Pattern: Pipe-and-Filter
Solution: Structure an application as a fixed sequence of filters that take input data from preceding filters, carry out computations on that data, and then pass the output to the next filter. The filters are side-effect free; i.e., the result of their action is only to transform input data into output data. Concurrency emerges as multiple blocks of data move through the Pipe-and-Filter system so that multiple filters are active at one time.

Concurrent Algorithm Strategy Pattern: Data-Parallelism
Solution: An algorithm is organized as operations applied concurrently to the elements of a set of data structures. The concurrency is in the data. This pattern can be generalized by defining an index space. The data structures within a problem are aligned to this index space and concurrency is introduced by applying a stream of operations for each point in the index space.

In OPL, the top two categories, the structural and computational patterns, are placed side by side with connecting arrows. This shows the tight coupling between these patterns and the iterative nature of how a designer works with them. In other words, a designer thinks about his or her problem, chooses a structural pattern, and then considers the computational patterns required to solve the problem. The selection of computational patterns may suggest a different overall structure for the architecture and may force a reconsideration of the appropriate structural patterns. This process, moving between structural and computational patterns, continues until the designer settles on a high-level design for the problem.

Structural and computational patterns are used in both serial and parallel programs. Ideally, the designer working at this level, even for a parallel program, will not need to focus on parallel computing issues. For the remaining layers of the pattern language, parallel programming is a primary concern.

Implementation Strategy Pattern: Loop-Parallel
Solution: An algorithm is implemented as loops (or nested loops) that execute in parallel. The challenge is to transform the loops so that iterations can safely execute concurrently and in any order. Ideally, this leads to a single source code tree that generates a serial program (by using a serial compiler) or a parallel program (by using compilers that understand the parallel loop constructs).

Parallel programming is the art of using concurrency in a problem to make the problem run to completion in less time. We divide the parallel design process into the following three layers.

3. Concurrent algorithm strategies: These patterns define high-level strategies to exploit concurrency in a computation for execution on a parallel computer. They address the different ways concurrency is naturally expressed within a problem by providing well-known techniques to exploit that concurrency. A good example of an algorithm strategy pattern is the *Data-Parallelism* pattern.
4. Implementation strategies: These are the structures that are realized in source code to support (a) how the program itself is organized and (b) common data structures specific to parallel programming. The *Loop-Parallel* pattern is a well-known example of an implementation strategy pattern.
5. Parallel execution patterns: These are the approaches used to support the execution of a parallel algorithm. This includes (a) strategies that advance a program counter and (b) basic building blocks to support the coordination of concurrent tasks. The single instruction multiple data (SIMD) pattern is a good example of a parallel execution pattern.

Parallel Execution Pattern: SIMD
Solution: An implementation of a strictly data parallel algorithm is mapped onto a platform that executes a single sequence of operations applied uniformly to a collection of data elements. The instructions execute in lockstep by a set of processing elements but on their own streams of data. SIMD programs use specialized data structure, data alignment operations, and collective operations to extend this pattern to a wider range of data parallel problems.

Patterns in these three lower layers are tightly coupled. For example, software designs using the *Recursive-Splitting* algorithm strategy often utilize a *Fork/Join* implementation strategy pattern which is typically supported at the execution level with the *thread-pool* pattern. These connections between patterns are a key point in the text of the patterns.

OPL draws from a long history of research on software design. The structural patterns of Category 1 are largely taken from the work of Garlan and Shaw on architectural styles [8, 9]. That these architectural styles could also be viewed as design patterns was quickly recognized by Buschmann [11]. We added two structural patterns that have their roots in parallel computing to Garlan and Shaw's architectural styles: *Map-Reduce*, influenced by [12] and *Iterative-Refinement*, influenced by Valiant's bulk-synchronous-processing pattern [13]. The computation patterns of Category 2 were first presented as "dwarfs" in [10] and their role as computational patterns was only identified later [1]. The identification of these computational patterns in turn owes a debt to Phil Colella's unpublished work on the "Seven Dwarfs of Parallel Computing." The lower three categories within OPL build on earlier and more traditional patterns for parallel algorithms by Mattson, Sanders, and Massingill [7]. This work was somewhat inspired by Gamma's success in using design patterns for object-oriented programming [5]. Of course all work on design patterns has its roots in Alexander's ground-breaking work identifying design patterns in civil architecture [6].

"All work on design patterns has its roots in Alexander's ground-breaking work identifying design patterns in civil architecture."

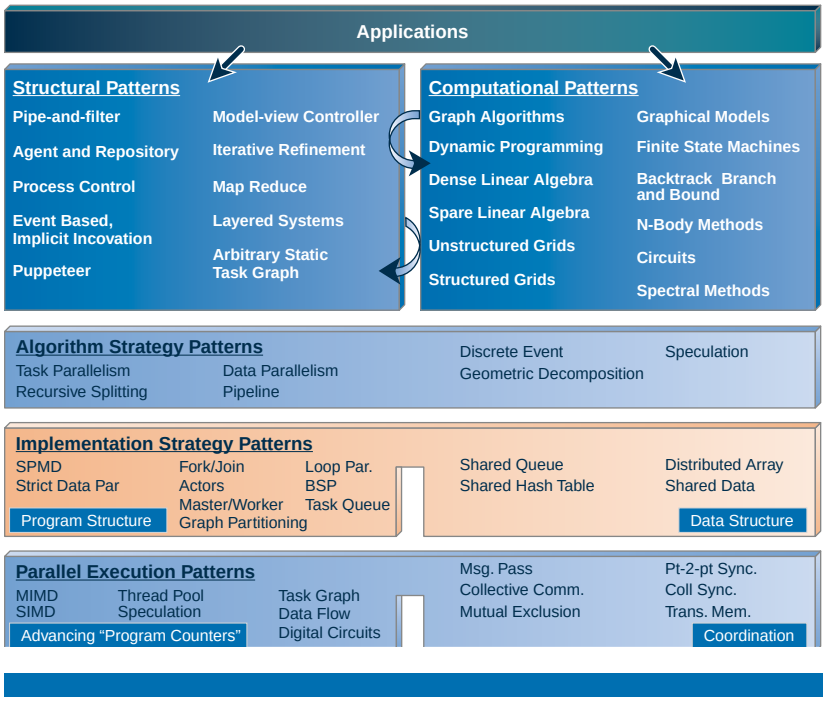

Figure 1: The Structure of OPL and the Five Categories of Design Patterns. Details About Each of the Patterns can be Found in [4].
Source: UC Berkeley ParLab, 2009

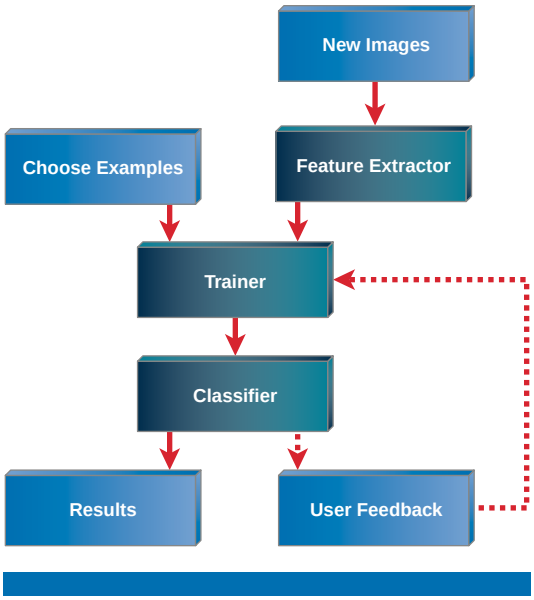

Figure 2: The CBIR Application Framework
Source: UC Berkeley ParLab, 2009

Case Study: Content-based Image Retrieval

Experience has shown that an easy way to understand patterns and how they are used is to follow an example. In this section we describe a problem and its parallelization by using patterns from OPL. In doing so we describe a subset of the patterns and give some indication of the way we make transitions between layers in the pattern language.

In particular, to understand how OPL can help software architecture, we use a content-based image retrieval (CBIR) application as an example. From this example (drawn from [14]), we show how structural and computational patterns can be used to describe the CBIR application and how the lower-layer patterns can be used to parallelize an exemplar component of the CBIR application.

In Figure 2 we see the major elements of our CBIR application as well as the data flow. The key elements of the application are the feature extractor, the trainer, and the classifier components. Given a set of new images the feature extractor will collect features of the images. Given the features of the new images, chosen examples, and some classified new images from user feedback, the trainer will train the parameters necessary for the classifier. Given the parameters from the trainer, the classifier will classify the new images based on their features. The user can classify some of the resulting images and give feedback to the trainer repeatedly in order to increase the accuracy of the classifier. This top-level organization of CBIR is best represented by the *Pipe-and-Filter* structural pattern. The feature-extractor, trainer, and classifier are filters or computational elements that are connected by pipes (data communication channels). Data flows through the succession of filters that do not share state and only take input from their input pipe(s). The filters perform the appropriate computation on those data and pass the output to the next filter(s) via its output pipe. The choice of *Pipe-and-Filter* pattern to describe the top-level structure of CBIR is not unusual. Many applications are naturally described by *Pipe-and-Filter* at the top level.

In our approach we architect software by using patterns in a hierarchical fashion. Each filter within the CBIR application contains a complex set of computations. We can parallelize these filters using patterns from OPL. Consider, for example, the classifier filter. There are many approaches to classification, but in our CBIR application we use a support-vector machine (SVM) classifier. SVM is widely used for classification in image recognition, bioinformatics, and text processing. The SVM classifier evaluates the function:

$$\hat{z} = \mathrm{sgn}\left\{b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z)\right\}$$

where x_i is the i^{th} support vector, z is the query vector, Φ is the kernel function, α_i is the weight, y_i in {-1, 1} is the label attached to support vector x_i, b is a parameter, and sgn is the sign function. In order to evaluate the function quickly, we identified that the kernel functions are operating on the products and norms of x_i and z. We can compute the products between a set of query

"Many applications are naturally described by Pipe-and-Filter *at the top level."*

vectors and the support vectors by a BLAS level-3 operation with higher throughput. Therefore, we compute the products and norms first, use the results for computing the kernel values, and sum up the weighted kernel values. We architect the SVM classifier as shown in Figure 3. The basic structure of the classifier filter is itself a simple *Pipe-and-Filter* structure with two filters: the first filter takes the test data and the support vectors needed to calculate the dot products between the test data and each support vector. This dot product computation is naturally performed by using the *Dense-Linear-Algebra* computational pattern. The second filter takes the resulting dot products, and the following steps are to compute the kernel values, sum up all the kernel values, and scale the final results if necessary. The structural pattern associated with these computations is *Map-Reduce* (see the *Map-Reduce* sidebar).

In a similar way the feature-extractor and trainer filters of the CBIR application can be decomposed. With that elaboration we would consider the "high-level" architecture of the CBIR application complete. In general, to construct a high-level architecture of an application, we decompose the application hierarchically by using the structural and computational patterns of OPL.

Constructing the high-level architecture of an application is essential, and this effort improves not just the software viability but also eases communication regarding the organization of the software. However, there is still much work to be done before we have a working software application. To perform this work we move from the top layers of OPL (structural and computational patterns) down into lower layers (concurrent algorithmic strategy patterns etc.). To illustrate this process we provide additional detail on the SVM classifier filter.

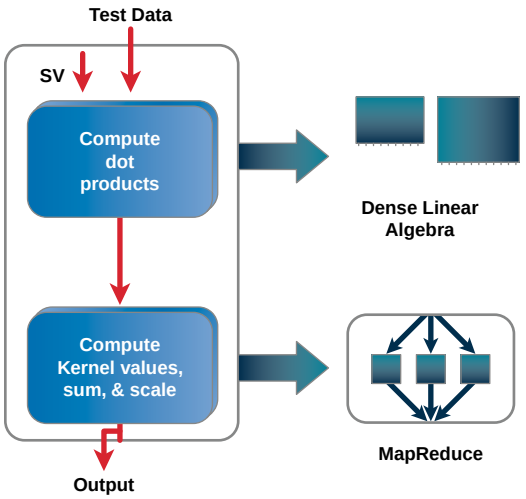

Figure 3: The Major Computations of the SVM Classifier
Source: UC Berkeley ParLab, 2009

Concurrent Algorithmic Strategy Patterns

After identifying the structural patterns and the computational patterns in the SVM classifier, we need to find appropriate strategies to parallelize the computation. In the *Map-Reduce* pattern the same computation is *mapped* to different non-overlapping partitions of the state set. The results of these computations are then gathered, or *reduced*. If we are interested in arriving at a parallel implementation of this computation, then we define the *Map-Reduce* structure in terms of a Concurrent Algorithmic Strategy. The natural choices for Algorithmic Strategies are the *Data-Parallelism* and *Geometric-Decomposition* patterns. By using the *Data-Parallelism* pattern we can compute the kernel value of each dot product in parallel (see the *Data-Parallelism* sidebar). Alternatively, by using the *Geometric-Decomposition* pattern (see the *Geometric-Decomposition* sidebar) we can divide the dot products into regular chunks of data, apply the dot products locally on each chunk, and then apply a global reduce to compute the summation over all chunks for the final results. We are interested in designs that can utilize large numbers of cores. Since the solution based on the *Data-Parallelism* pattern exposes more concurrent tasks (due to the large numbers of dot products) compared to the more coarse-grained geometric decomposition solution, we choose the *Data-Parallelism* pattern for implementing the map reduce computation.

Structural Pattern: *Map-Reduce*
Solution: A solution is structured in two phases: (1) a map phase where items from an "input data set" are mapped onto a "generated data set" and (2) a reduction phase where the generated data set is reduced or otherwise summarized to generate the final result. It is easy to exploit concurrency in the map phase, since the map functions are applied independently for each item in the input data set. The reduction phase, however, requires synchronization to safely combine partial solutions into the final result.

Algorithm Strategy Pattern:
Geometric-Decomposition
Solution: An algorithm is organized by (1) dividing the key data structures within a problem into regular chunks, and (2) updating each chunk in parallel. Typically, communication occurs at chunk boundaries so an algorithm breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The size of the chunks is dictated by the properties of the memory hierarchy to maximize reuse of data from local memory/cache.

The use of the *Data-Parallelism* algorithmic strategy pattern to parallelize the Map-Reduce computation is shown in the pseudo code of the kernel value calculation and the summation. These computations can be summarized as shown in Figure 4. Line 1 to line 4 is the computation of the kernel value on each dot product, which is the map phase. Line 5 to line 13 is the summation over all kernel values, which is the reduce phase. Function NeedReduce checks whether element "i" is a candidate for the reduction operation. If so, the ComputeOffset function calculates the offset between element "i" and another element. Finally, the Reduce function conducts the reduction operation on element "i" and "i+offset".

Implementation Strategy Patterns

To implement the data parallelism strategy from the Map-Reduce pseudo-code, we need to find the best Implementation Strategy Pattern. Looking at the patterns in OPL, both the *Strict-Data-Parallel* and *Loop-Parallel* patterns are applicable.

Whether we choose the *Strict-data-parallel* or *Loop-parallel* patterns in the implementation layer, we can use the *SIMD* pattern for realizing the execution. For example, we can apply SIMD on line 2 in Code Listing 1 for calculating the kernel value of each dot product in parallel. The same concept can be used on line 7 in Code Listing 1 for conducting the checking procedure in parallel. Moreover, in order to synchronize the computations on different processing elements on line 4 and line 12 in Code Listing 1, we can use the barrier construct described within the *Collective-Synchronization* pattern for achieving this goal.

Implementation Strategy Pattern:
Strict-Data-Parallel
Solution: Implement a data parallel algorithm as a single stream of instructions applied concurrently to the elements of a data set. Updates to each element are either independent, or they involve well-defined collective operations such as reductions or prefix scans.

```
function ComputeMapReduce( DotProdAndNorm, Result) {
1  for i ← 1 to n {
2    LocalValue[i] ←
                    ComputeKernelValue(DotProdAndNorm[i]);
3        }
4        Barrier();
5        for reduceLevel ← 1 to MaxReduceLevel {
6        for i ← 1 to n {
7          if (NeedReduce(i, reduceLevel) ) {
8                  offset ← ComputeOffset(i, reduceLevel);
9                  LocalValue[i] ← Reduce(LocalValue[i],
                       LocalValue[i+offset]);
10                }
11         }
12         Barrier();
13 }
14}
```

Code Listing 1: Pseudo Code of the Map Reduce Computation
Source: Intel Corporation, 2009

In summary, the computation of the SVM classifier can be viewed as a composition of the *Pipe-and-Filter*, *Dense-Linear-Algebra*, and *Map-Reduce* patterns. To parallelize the *Map-Reduce* computation, we used the *Data-Parallelism* pattern. To implement the *Data-Parallelism* Algorithmic Strategy, both the *Strict-Data-Parallel* and *Loop-Parallel* patterns are applicable. We choose the *Strict-Data-Parallel* pattern, since it seemed a more natural choice given the fact we wanted to expose large amounts of concurrency for use on many-core chips with large numbers of cores. It is important to appreciate, however, that this is a matter of style, and a quality design could have been produced by using the *Loop-Parallel* pattern as well. To map the *Strict-Data-Parallel* pattern onto a platform for execution, we chose a *SIMD* pattern. While we did not show the details of all the patterns used, along the way we used the *Shared-Data* pattern to define the synchronization protocols for the reduction and the *Collective-Synchronization* pattern to describe the barrier construct. It is common that these functions (reduction and barrier) are provided as part of a parallel programming environment; hence, while a programmer needs to be aware of these constructs and what they provide, it is rare that they will need to explore their implementation in any detail.

"We choose the Strict-Data-Parallel pattern …. however, that is a matter of style … a quality design could have been produced using the Loop-Parallelism pattern as well."

Other Patterns

OPL is not complete. Currently OPL is restricted to those parts of the design process associated with architecting and implementing applications that target parallel processors. There are countless additional patterns that software development teams utilize. Probably the best known example is the set of design patterns used in object-oriented design [8]. We made no attempt to include these in OPL. An interesting framework that supports common patterns in parallel object-oriented design is Thread Building Blocks (TBB) [15].

OPL focuses on patterns that are ultimately expressed in software. These patterns do not, however, address methodological patterns that experienced parallel programmers use when designing or optimizing parallel software. The following are some examples of important classes of methodological patterns.

- *Finding Concurrency patterns* [7]. These patterns capture the process that experienced parallel programmers use when exploiting the concurrency available in a problem. While these patterns were developed before our set of Computational patterns was identified, they appear to be useful when moving from the Computational patterns category of our hierarchy to the Parallel Algorithmic Strategy category. For example, applying these patterns would help to indicate when geometric decomposition is chosen over data parallelism as a dense linear algebra problem moves toward implementation.

"OPL focuses on patterns that are ultimately expressed in software."

- *Parallel Programming "Best Practices" patterns*. This describes a broad range of patterns we are actively mining as we examine the detailed work in creating highly-efficient parallel implementations. Thus, these patterns appear to be useful when moving from the Implementation Strategy patterns to the Concurrent Execution patterns. For example, we are finding common patterns associated with optimizing software to maximize data locality.

"We can define a systematic methodology for software architecture in terms of design patterns *and a* pattern language. *"*

There is a growing community of programmers and researchers involved in the creation of OPL. The current status of OPL, including the most recent updates of patterns, can be found at: ***http://parlab.eecs.berkeley.edu/wiki/patterns/patterns***. This website also has links to details on our shorter monthly patterns workshop as well as our longer, two-day, formal patterns workshop. We welcome your participation.

Summary, Conclusions, and Future Work

We believe that the key to addressing the challenge of writing software is to architect the software. In particular, we believe that the key to addressing the new challenge of programming multi-core and many-core processors is to carefully architect the parallel software. We can define a systematic methodology for software architecture in terms of *design patterns* and a *pattern language*. Toward this end we have taken on the ambitious project of creating a comprehensive pattern language that stretches all the way from the initial software architecture of an application down to the lowest-level details of software implementation.

"We also need to continue mining patterns from existing parallel software to identify patterns that may be missing from our language."

OPL is a work in progress. We have defined the layers in OPL, listed the patterns at each layer, and written text for many of the patterns. Details are available online [4]. On the one hand, much work remains to be done. On the other hand, we feel confident that our structural patterns capture the critical ways of composing software, and our computational patterns capture the key underlying computations. Similarly, as we move down through the pattern language, we feel that the patterns at each layer do a good job of addressing most of the key problems for which they are intended. The current state of the textual descriptions of the patterns in OPL is somewhat nascent. We need to finish writing the text for some of the patterns and have them carefully reviewed by experts in parallel applications programming. We also need to continue mining patterns from existing parallel software to identify patterns that may be missing from our language. Nevertheless, last year's effort spent in mining five applications netted (only) three new patterns for OPL. This shows that while OPL is not fully complete, it is not, with the caveats described earlier, dramatically deficient.

Complementing the efforts to mine existing parallel applications for patterns is the process of architecting new applications by using OPL. We are currently using OPL to architect and implement a number of applications in areas such as machine learning, computer vision, computational finance, health, physical modeling, and games. During this process we are watching carefully to identify

where OPL helps us and where OPL does not offer patterns to guide the kind of design decisions we must make. For example, mapping a number of computer-vision applications to new generations of many-core architectures helped identify the importance of a family of data layout patterns.

The scope of the OPL project is ambitious. It stretches across the full range of activities in architecting a complex application. It has been suggested that we have taken on too large of a task; that it is not possible to define the complete software design process in terms of a single design pattern language. However, after many years of hard work, nobody has been able to solve the parallel programming problem with specialized parallel programming languages or tools that automate the parallel programming process. We believe a different approach is required, one that emphasizes how people think about algorithms and design software. This is precisely the approach supported by design patterns, and based on our results so far, we believe that patterns and a pattern language may indeed be the key to finally resolving the parallel programming problem.

"Mapping a number of computer-vision applications to new generations of many-core architectures helped identify the importance of a family of data layout patterns."

While this claim may seem grandiose, we have an even greater aim for our work. We believe that our efforts to identify the core computational and structural patterns for parallel programming has led us to begin to identify the core computational elements (computational patterns, analogous to atoms) and means of assembling them (structural patterns, analogous to molecular bonding) of *all* electronic systems. If this is true, then these patterns not only serve as a means to assist software design but can be used to architect a curriculum for a true discipline of computer science.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. "A View of the Parallel Computing Landscape." *Communications of the ACM, volume 51*, pages 56-67, 2009.

[2] B. Chapman, G. Jost, and R van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT press, Cambridge, Massachusetts, 2008.

[3] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. 2nd edition, MIT Press, Cambridge, Massachusetts, 1999.

[4] ***http://parlab.eecs.berkeley.edu/wiki/patterns/patterns***

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[6] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, New York, 1977.

[7] T. G. Mattson, B. A. Sanders, B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, Massachusetts, 2004.

[8] D. Garlan and M. Shaw. "An introduction to software architecture." *Carnegie Mellon University Software Engineering Institute Report* CMU SEI-94-TR-21, Pittsburg, Pennsylvania, 1994.

[9] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[10] K. Asanovic, et al. "The landscape of parallel computing research: A view from Berkeley." *EECS Department, University of California, Berkeley*, Technical Report UCB/EECS-2006-183, 2006.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, Hoboken, New Jersey, 1996.

[12] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *In Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implementation*. San Francisco, CA, December 2004.

[13] L. G. Valiant, "A Bridging Model for Parallel Computation." *Communication of the ACM*, volume 33, pages 103-111, 1990.

[14] Catanzaro, B., B. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer. "Efficient, High-Quality Image Contour Detection." *IEEE International Conference on Computer Vision* (ICCV09), pages 2381-2388, Kyoto Japan, 2009.

[15] J. Reinders. *Intel Threaded Building Blocks*. O'Reilly Press, Sebastopol, California, 2007.

Acknowledgments

The evolution of OPL has been strongly influenced by the collaborative environment provided by Berkeley's Par Lab. The development of the language has been positively impacted by students and visitors in two years of graduate seminars focused on OPL: Hugo Andrade, Chris Batten, Eric Battenberg, Hovig Bayandorian, Dai Bui, Bryan Catanzaro, Jike Chong, Enylton Coelho, Katya Gonina, Yunsup Lee, Mark Murphy, Heidi Pan, Kaushik Ravindran, Sayak Ray, Erich Strohmaier, Bor-yiing Su, Narayanan Sundaram, Guogiang Wang, and Youngmin Yi. The development of OPL has also received a boost from Par Lab faculty — particularly Krste Asanovic, Jim Demmel, and David Patterson. Monthly pattern workshops in 2009 also helped to shape the language. Special thanks to veteran workshop moderator Ralph Johnson as well as to Jeff Anderson-Lee, Joel Jones, Terry Ligocki, Sam Williams, and members of the Silicon Valley Patterns Group.

Authors' Biographies

Kurt Keutzer. After receiving his Ph.D. degree in Computer Science from Indiana University in 1984, Kurt joined AT&T Bell Laboratories where he was a Member of Technical Staff in the last of the golden era of Bell Labs Research. In 1991 he joined Synopsys, Inc. where he served in a number of roles culminating in his position as a Chief Technical Officer and Senior Vice-President of Research. Kurt left Synopsys in January 1998 to become Professor of Electrical Engineering and Computer Science at the University of California at Berkeley. At Berkeley he worked with Richard Newton to initiate the MARCO-funded Gigascale Silicon Research Center and was Associate Director of the Center from 1998 until 2002. He is currently a principal investigator in Berkeley's Universal Parallel Computing Research Center.

Kurt has researched a wide number of areas related to both the design and programming of integrated circuits, and his research efforts have led to four best-paper awards. He has published over 100 refereed publications and co-authored six books, his latest being *Closing the Power Gap Between ASIC and Custom*. Kurt was made a Fellow of the IEEE in 1996.

Tim Mattson. Tim received a Ph.D. degree for his work on quantum molecular scattering theory from UC Santa Cruz in 1985. Since then he has held a number of commercial and academic positions working on the application of parallel computers to mathematics libraries, exploration geophysics, computational chemistry, molecular biology, and bioinformatics.

Dr. Mattson joined Intel in 1993. Among his many roles he was applications manager for the ASCI Red Computer (the world's first TeraFLOP computer), helped create OpenMP, founded the Open Cluster Group, led the applications team for the first TeraFLOP CPU (the 80-core tera-scale processor), launched Intel's programs in computing for the Life Sciences, and helped create OpenCL.

Currently, Dr. Mattson is a Principal Engineer in Intel's Visual Applications Research Laboratory. He conducts research on performance modeling and how different programming models map onto many-core processors. Design patterns play a key role in this work and help keep the focus on technologies that help the general programmer solve real parallel programming problems.

Copyright