Chapter 8

Introduction to Design Patterns for Parallel Computing

Kurt Keutzer and Tim Mattson

1 Introduction

Our Pattern Language (OPL), described in the following paper A Design Pattern Language for Engineering (Parallel) Software, represents the confluence of three different directions of research. The first direction of research was aimed at analyzing the varieties of computing, with a goal of identifying key influences on computer architecture. This research led to the identification of "13 dwarfs" of computing, which in turn were became "13 computational patterns" of OPL (See Figure 1 of the following paper on OPL in this chapter). The second direction of research arose from generations of research on architecting large pieces of software. This research led to the identification of a series of "architectural styles" [6] that were incorporated into OPL as "structural patterns." These computational and structural patterns sit, side by side, at the top of OPL in Figure 1 of the following paper. The overriding vision of OPL, a unified pattern language for designing and implementing parallel software, came from Mattson's book *Patterns for Parallel Programming* [3]. A revised of version of Mattson's pattern languages constitutes the three lower levels of OPL.

2 Quick Introduction to Patterns

A design pattern is simply a generalizable solution to a recurring problem that occurs within a well-defined context. Patterns are written down in a highly structured format to capture the most essential elements of a problem's solution in such a way that a software designer can quickly find what he or she needs to solve a problem. The concept of design patterns originated in the work of Christopher Alexander. Alexander gathered his ideas in the book *A Pattern Language: Towns, Buildings, Construction* [2]. His goal was to identify and codify common solutions to recurrent problems ranging from the arrangement of towns and cities to the construction of a single building. Alexander's ideas on civil architecture are still debated, but the impact of his ideas on software design has been substantial.

Design patterns help software designers in a number of ways. First, they give a name to solutions to common problems in software design. Further, they define a palette of solution alternatives to these problems. This palette serves to educate novice programmers as to design alternatives, and it also serves to comfort more mature programmers that there is such a fixed set of alternatives. At a deeper level work by cognitive psychologists on the nature of programming has shown that patterns (which they called "plans") are at the core of the thought processes used by expert programmers [5]. Thus, a design pattern encodes expertise. In fact, we can view the difference between a novice and an expert in any field in terms of patterns; i.e., an expert has a larger library of design patterns to draw on.

Hence, in our own work to bring order and discipline to the practice of engineering parallel software, we have found design patterns to be a valuable tool. It is the connections between patterns that matter in a high-quality software design, so a catalog of isolated design patterns will not be sufficient. Instead, we need a web of interconnected patterns. A hierarchical layered construction of design patterns works together to define a language for patterns oriented-design. Following Alexander, we call this interconnected family of patterns a "Design Pattern Language."

3 Dwarfs, Motifs, and Computational Patterns

As discussed in the Introduction to this entire volume, efforts to define the 13 dwarfs predate the efforts on developing OPL, and OPL is not discussed at all in the original Berkeley View paper [1]. It was Tim Mattson who suggested that while the dwarfs did not indicate a definite micro-architecture or software implementation, they did provide a nice bridge between the computationally focused thinking of application developers and particular approaches to parallelizing computations that he had expressed in his pattern language for parallel programming [3]. Mattson made another important suggestion: these dwarfs could be understood as computational patterns. In other words, dwarfs were also generalizable solutions to recurrent problems of computation, and could form an integral part of a pattern language for parallel computing.

4 Architectural Styles, Design Patterns, and Computational Patterns

For some time OPL was envisioned as a set of computational patterns (the former dwarfs) sitting on top of Mattson's pattern language for parallel programming [3]. The result was still less than fully satisfying because few applications could be naturally decomposed into a single More generally, this generation of OPL didn't help at all with the important problem of how to compose computations. It was Koushik Sen who first suggested that a small set of techniques could be used to compose all the diverse sets of computations in parallel programs.

Approaches to composing software had received considerable attention in software engineering of sequential programs, and particularly in object-oriented programming. Initially this trend of research seemed unimportant and perhaps totally irrelevant for parallel software. Examining the software architectures developed in an object-oriented style gave little insight into how to structure an efficient parallel program. Over time, Kurt Keutzer began to champion the notion that while software architecture did not offer immediate insights into how to parallelize software, the need for software modularity was as great or greater in parallel software. Shaw and Garland had already codified a list of architectural styles [6] for composing software and creating software architectures. This list of architectural styles seemed immediately useful for parallel software as well; however, four structural patterns particularly useful for parallel computing were added - Map Reduce, Iterative Refinement, and Puppeteer, and Arbitrary Static Task Graph replaced the common Call-and-Return architecture of object-oriented programming.

5 Patterns for parallel programming and OPL

The structural and computational patterns are composed to define *software architecture*. A software architecture defines the components that make up an application, the communication among components, and the fundamental computations that occur inside components. The software architecture, however, has little to say about how the software architecture is mapped onto the hardware of a parallel computer. To address parallelism, we combined the computational and structural patterns with the lower level, parallel programming design pattern languages from [3]. These parallel programming patterns define a distinct pattern language for parallel programming (PLPP).

The original version of PLPP dates back to the late 1990's and early 2000's. At that time, data parallel algorithms had fallen into disrepute. The use of graphics processing units (GPUs) for general-purpose programming was considered more of a gimmick than a serious option for computing. Data parallel algorithms popularized by the older SIMD machines of the late 1980's to mid-1990's had failed spectacularly. The result is that the original version of PLPP deemphasized data parallel patterns and instead emphasized patterns relevant to cluster and shared-memory multiprocessor computers.

The work with the Par Lab, however, forced a reconsideration of PLPP. Projects such as the MRI image processing and Content Based Image Recognition applications described elsewhere in this book demonstrated the importance of

GPGPU programming and the data parallel algorithms behind them. Furthermore, the wider vector units on modern microprocessors showed the importance of data parallel algorithms mapped onto vector computing. The result is a complete redesign of PLPP. A book describing this new version of PLPP is being prepared. A preview of the new PLPP, however, can be found in the paper on OPL, A Design Pattern Language for Engineering (Parallel) Software, elsewhere in this chapter.

6 Patterns in Par Lab Applications and SEJITS

Patterns from OPL were used to guide the design of a number of applications described elsewhere in this book. A similar OPL-based methodology was used for the magnetic resonance image reconstruction application, the content-based image reconstruction application, the speech recognition application and a number of other variants of audio-processing described in the paper "PyCASP: A Pattern-Oriented Framework for Parallel Programming" found elsewhere in this chapter. In this methodology a high-level software architecture was proposed using structural and computational patterns, a parallel algorithm strategy was chosen, then an implementation strategy, and finally an execution strategy. In particular, the development of the high-level software architecture for content-based image retrieval is used as a model in the paper "A Design Pattern Language for Engineering (Parallel) Software," found elsewhere in this chapter. In truth, the design approach was not entirely top-down: often the sequence bounced back between explorations of high-level software architectures and final execution strategies with the parallel algorithm strategy following suit. In fact, it was the observation that after the software architecture and execution strategy were chosen, the parallel algorithm and implementation strategy seemed obvious and led to a pattern-oriented approach to SEJITS.

(Referring to Figure 1 in "A Design Pattern Language for Engineering (Parallel) Software" will help in following this discussion.) In a number of Par Lab applications we discovered that dense-linear algebra, structured grid, and other computational patterns offered ample opportunities for a data parallel algorithm strategy. With stream processing on a GPGPU processor as the execution target, an SPMD implementation strategy is a natural choice for exploiting the data parallelism. Thus we saw many applications follow a dense linear algebra, data parallelism, SPMD, stream processing flow through OPL. SEJITS was an ideal mechanism for building a single "stovepipe" flow to hardware rather than use a traditional layered compiler architecture with a number of intermediate representations. The idea of using SEJITS as an implementation medium for pattern-oriented frameworks is demonstrated in the paper "PyCASP: A Pattern-oriented Framework for Parallel Programming" later in this chapter and also in the paper "Copperhead: Compiling an Embedded Data Parallel Language" found elsewhere in this volume.

7 Evolving and Evangelizing OPL

The pattern language was evolved and evangelized using a number of mechanisms. Three Parallel Patterns Workshops were held from 2009 to 2011. Each workshop drew over thirty participants from around the world. Textual descriptions of the structural and computational patterns were developed in these workshops and also in three graduate seminars taught at Berkeley. A course entitled Engineering Parallel Software, using OPL's approach, was taught to undergraduate audiences three times. Average class size in each instance was twenty-five. In each course projects demonstrated that students could learn in a single semester how to architect code and efficiently implement it using the design patterns in OPL. More on educational efforts within Par Lab is discussed in the education chapter.

8 Summary and Future Work

It is our strong conviction that software architecture is the key for productively designing software targeted for either sequential or parallel targets. To address the current challenge of architecting and implementing parallel software we have developed Our Pattern Language (OPL). OPL uses a series of patterns to guide the architecting, parallelization, and implementation of software on parallel processor targets. We have demonstrated this approach on a number of applications described elsewhere in this book including MRI image reconstruction, content-based image retrieval, and speech recognition. We have also demonstrated this approach on applications in a number of other applications in computational finance, audio processing, and computer vision. We have then extended this work to pattern-oriented frameworks such as Copperhead and PyCASP described elsewhere in this book. OPL has proved robust, but many directions for future work remain. Many developers and researchers have asked for help in automatically identifying patterns in legacy code. Others have asked for help in annotating patterns with design constraints. Many domain experts don't think at the level of computational patterns (e.g., dense linear algebra) but at a level above that (e.g., convolution). We are developing pattern languages for particular application domains such as computer vision. OPL looks at program structure and computation but does not offer patterns to facilitate the refinement of communication from architecture to implementation. Arch Robison looked at the degree to which each pattern in OPL was supported in Intel's Threaded Building Blocks [4]. We hope that other programming environment developers will use OPL to evaluate their languages as well. There are many optimization approaches necessary to squeeze out the last percentages of performance in parallel software, but these repeated patterns of optimization are not currently captured in OPL. Finally, the utility of the 13 computational patterns in guiding micro-architecture is just beginning to be reconsidered.

Bibliography

[1] K. Asanović, R. Bodík, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

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

[3] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

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

[5] R. Rist. *Empirical Studies of Programmers*, chapter Plans in programming: definition, demonstration and development, pages 28–47. Norweed, NF, Ablex, 1986.

[6] M. Shaw and D. Garlan. Software architecture: perspectives on an emerging discipline, 1996.

A Design Pattern Language for Engineering (Parallel) Software[*]

Kurt Keutzer and Tim Mattson

Abstract

The key to writing high quality parallel software is to develop a robust software design. This applies 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. Technology to more systematically describe such designs and reuse them between software projects is the fundamental problem facing software for terascale processors. This 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 paper, we will develop our thesis regarding the central role played by software architecture for software. We will then 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 Par Lab to pull the essential set of design patterns for parallel software design into an integrated Design Pattern Language. After describing Our Pattern Language, we'll present a case study from the field of machine learning as a concrete example of how patterns are used in practice.

1 The Software Engineering Crisis

The trend has been well established [3]: 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 CPU's to add cores; however, this is not the case. The motivation for parallelism comes from the inability to deliver steadily increasing processor 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 [10].

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 and OpenMP 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.

[*]This is an updated version of the paper "A Design Pattern Language for Engineering (Parallel) Software", K. Keutzer and T. Mattson, Intel Technology Journal, Vol. 13, No. 4, pp. 6-19, 2009.

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 on 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 almost always leads to failure - no matter how much software tuning is added later. 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.

In this paper, we describe our work on software architecture. Towards this end 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 [1], we present a case study to show how these patterns can be used to design and comprehend software architecture.

2 Software Architecture and Design Patterns

Productive, efficient software follows from good software architecture. Hence, we need to develop a theory of 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.

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 write these elements down: they are known as design patterns [8] and find their inspiration in early work regarding design patterns for civil architecture [2].

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

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 execute close to their theoretically allowed peak performance. Applications of this pattern typically use standard building 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.

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 [1] there would be sample code, examples, references, invariants and other information needed to guide a software developer interested in dense linear algebra problems.

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, midlevel 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 of patterns that guide a designer from the beginning of a design problem to its successful realization ([2, 11]).

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, debate, and come to consensus on a common set of design patterns is a seminal intellectual challenge of our field

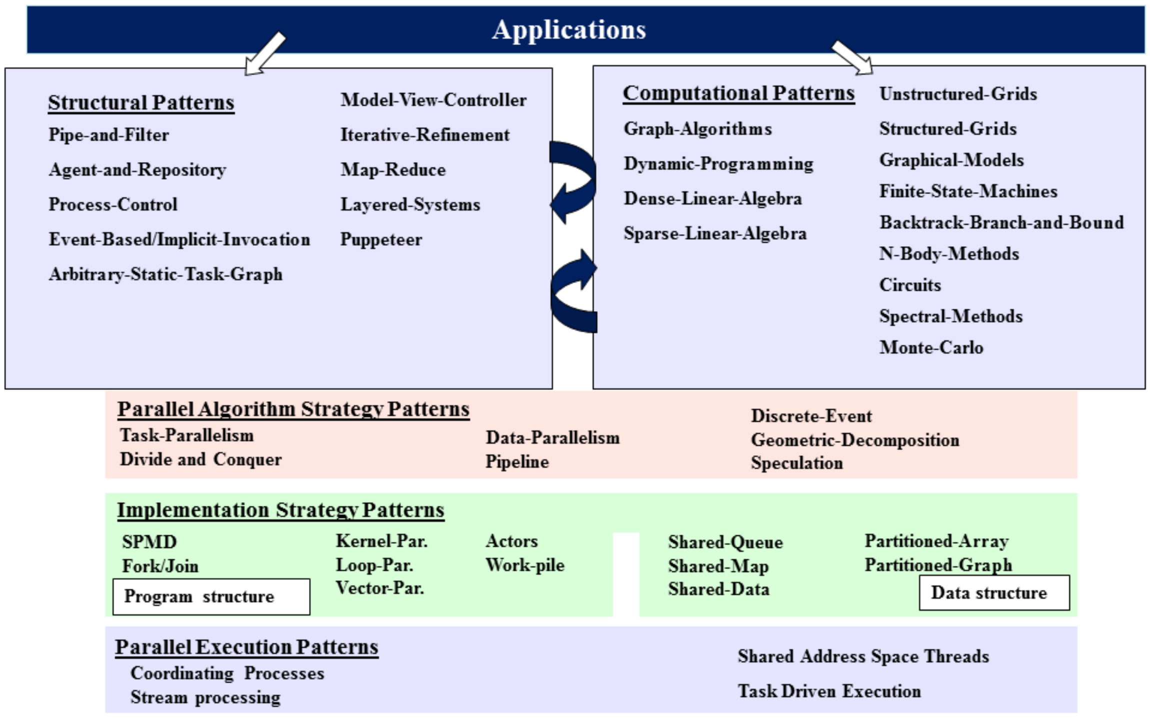

Figure 1: The structure of OPL and the five categories of design patterns.

3 Our Pattern Language

A software architecture describes 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 is modular and lets the designer localize problems and define design elements that can be reused from one problem to another.

The goal of 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 [9, 14], to the basic computational patterns (also known as computational dwarfs or motifs) for each stage of the problem [4, 3], to the low level details of the parallel algorithm [11]. 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 bounce around between layers with the designer working at whichever layer is most productive at a given time (so called "opportunistic refinement" [12]). 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.

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.

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

1. 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 [9]. Informally, these patterns correspond to the "boxes and arrows" an architect draws to describe the overall organization of the computation and communication 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 dwarfs or motifs made famous in [4], 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) 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 software design.

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.

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 iterations can safely execute concurrently and in any order. Ideally, this leads to a single source code tree that generates a serial program (using a serial compiler) or a parallel program (using compilers that understand the parallel loop constructs).

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 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.

The 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.

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 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 inside the runtime systems associated with different parallel programming environments. Application programmers use these patterns when optimizing the mapping of their algorithm onto a particular platform. The Stream Processing pattern is a good example of a parallel execution pattern.

Patterns in these three lower layers are tightly coupled. For example, a problem using the "divide and conquer" algorithm strategy is likely to utilize a fork-join implementation strategy which is commonly supported at the execution level with a "shared address space threads" pattern. These connections between patterns are a key point in the text of the patterns.

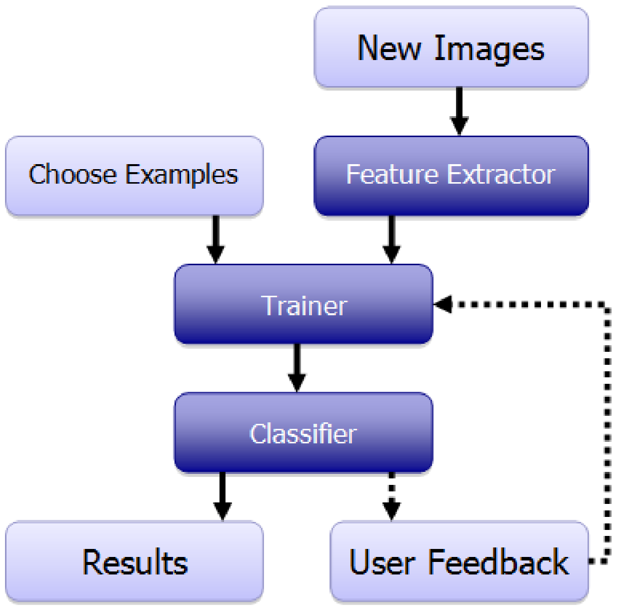

Figure 2: The CBIR application framework.

Parallel Execution Pattern: Stream Processing

Solution: A large collection of kernel-instances are streamed through the processing elements of a platform. The runtime system uses the fact that (1) the number of kernel-instances is much greater than the number of processing elements to hide memory access latencies and (2) many of the kernel instances are running the same sequence of operations (a convergent control flow) which can be mapped onto specialized SIMD processing elements for greater performance.

There is a large intellectual history leading up to OPL. The structural patterns of Category 1 are largely taken from the work of Garlan and Shaw on architectural styles [9, 14]. That these architectural styles could also be viewed as design patterns was quickly recognized by Buschmann [5]. To Garlan and Shaw's architectural styles we added two structural patterns that have their roots in parallel computing: Map Reduce, influenced by [6] and Iterative Refinement, influenced by Valiant's bulk-synchronous pattern [15]. The computation patterns of Category 2 were first presented as "dwarfs" in [4] and their role as computational patterns was only identified later [3]. 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 off earlier and more traditional patterns for parallel algorithms [11]. Mattson's work was somewhat inspired by Gamma's success in using design patterns for object-oriented programming [8]. Of course all work on design patterns has its roots in Alexander's ground-breaking work identifying design patterns in civil architecture [2].

4 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 new section we will describe a problem and its parallelization using patterns from OPL. In doing so we will 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 facilitate software architecture, we use a content-based image retrieval (CBIR) application as an example. From this example we will 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 which are connected by pipes (data communication channels). Data flows through the succession of filters which do not share state and only take input from their input pipe(s). The filters perform the appropriate computation on that 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 using patterns in a hierarchical fashion. Since each of the filters of CBIR are complex computations they can be further decomposed. In the following discussion we consider 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 in many classification tasks such as image recognition, bioinformatics, and text processing. The structure and computations in the SVM classifier are described 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 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 MapReduce (see the MapReduce sidebar).

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. Concurrency in the map phase is straightforward to exploit 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.

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 hierarchically decompose the application 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 will give additional detail on the SVM classifier filter.

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.

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 MapReduce 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 MapReduce structure in terms of a Concurrent Algorithmic Strategy. The natural choices for Algorithmic Strategies are the data parallelism and geometric decomposition patterns. Using data parallelism we can compute the kernel value of each dot product in parallel (see the data parallelism side bar). Alternatively, using geometric decomposition (see the geometric decomposition side bar) we can divide the dot products into regular chunks of data, apply the dot products locally on each

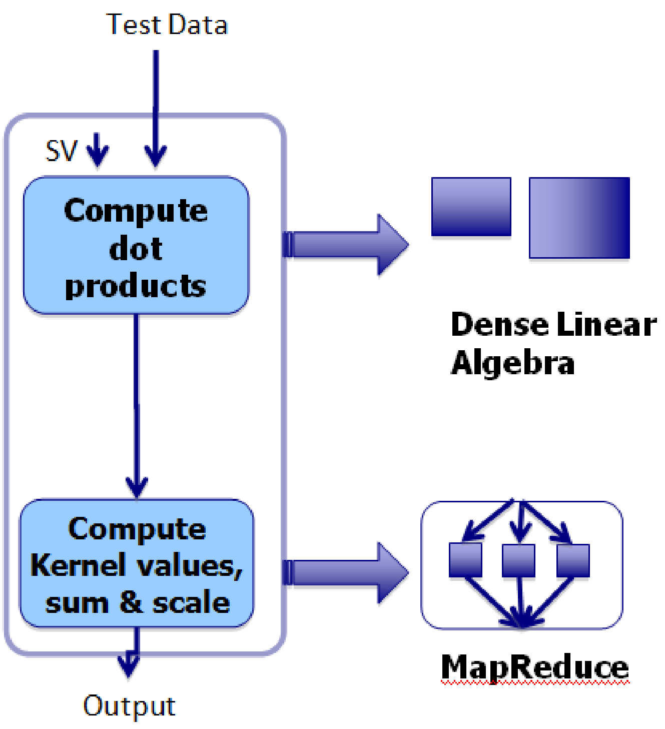

Figure 3: Architecture of the SVM classifier filter.

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 to geometric decomposition solution, we choose the data parallelism pattern for implementing the map reduce computation.

The use of the data parallelism algorithmic strategy pattern to parallelize the MapReduce 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 Pattern: Kernel Parallelism

Solution: an index space is defined onto which the data structures within a problem are mapped. The computations of the problem are organized into kernels. A sequence of kernels is submitted for execution each of which executes concurrently at each point in the index space.

To implement the data parallelism strategy from the MapReduce pseudo-code, we need to find the best Implementation Strategy Pattern. Looking at the patterns in OPL, both the kernel-parallel and loop-parallel patterns are applicable.

Whether we choose either kernel-parallel or loop-parallel patterns in the implementation layer, we can use the stream-processing pattern for realizing the execution. For example, we can apply stream-processing on line 2 in Figure 4 for calculating the kernel value of each dot product in parallel. The same concept can be used on line 7 in Figure 4 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 Figure 4, we can use the barrier construct described within the collective synchronization pattern for achieving this goal.

In summary, the computation of the SVM classifier can be viewed as a composition of the pipe-and-filter, dense linear algebra, and MapReduce patterns. To parallelize the MapReduce computation, we used the data parallelism pattern. To implement the data parallelism Algorithmic Strategy, both the kernel-parallel and loop-parallel patterns

```
function ComputeMapReduce( DataArray, Result) {
  for i ← 1 to n {
    LocalValue[i] ← ComputeKernelValue(DataArray[i]);
  }
  Barrier();
  for reduceLevel ← 1 to MaxReduceLevel {
    for i ← 1 to n {
      if (NeedReduce(i, reduceLevel) ) {
        offset ← ComputeOffset(i, reduceLevel);
        LocalValue[i] ← Reduce(LocalValue[i],
          LocalValue[i+offset]);
      }
    }
    Barrier();
  }
}
```

Figure 4: Pseudo-code of the map reduce computation.

are applicable. We choose the kernel-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 using the loop-parallel pattern as well. To map the kernel-parallel pattern onto a platform for execution, we chose stream-processing-pattern. While we didn't 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 barrier. 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.

5 Other Patterns

OPL is not complete. Currently OPL is restricted to those parts of the design process associated with architecting and implementing applications targeting 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 Threaded Building Blocks (TBB) [13].

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

- Finding concurrency patterns [11]: 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 in 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.

- Parallel programming "best practices" patterns or "optimization patterns": This describes a broad range of patterns we are actively mining as we examine the detailed work in creating highly-efficient parallel implementations. 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.

6 Summary, Conclusions and Future Work

We believe that software architecture is the key to writing successful software. In particular, we believe that the key to addressing the new challenge of programming multicore and manycore 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, OPL, that spans all the way from the initial software architecture of an application down to the lowest level details of software implementation.

OPL is a "work in progress". We have defined the layers in OPL, listed the patterns at each layer, and written text for all of the patterns. Details are available online [1]. On the one hand, much work remains to be done. On the other hand, we do 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. We will continue to mine patterns from existing parallel software to identify patterns that may be missing from our language. Nevertheless, over the last four years only minor taxonomic changes have occurred in OPL. This shows that while OPL is not fully complete, it is useful.

Complementing the efforts to mine existing parallel applications for patterns is the process of architecting new applications 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 manycore architectures helped identify the importance of a family of data-layout patterns.

OPL is an ambitious project. Its scope 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.

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 system. 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.

Bibliography

[1] `http://parlab.eecs.berkeley.edu/wiki/patterns/patterns`.

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

[3] 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. *Commun. ACM*, 52(10):56–67, Oct. 2009.

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

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerladand, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, 1996.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

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

[9] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, CMU Software Engineering Institute Technical Report, Pittsburgh, PA, USA, 1994.

[10] W. Hwu, K. Keutzer, and T. G. Mattson. The concurrency challenge. *IEEE Des. Test*, 25(4):312–320, July 2008.

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

[12] M. Petre. *Psychology of Programming*, chapter Expert Programmers and Programming Languages, page 103. Academic Press, 1990.

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

[14] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[15] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

Appendix A Design Pattern Descriptions

In this appendix, we will describe the contents of each category of patterns within OPL. For each category of patterns, we will define the goal of the patterns within that category, the artifacts from the design process produced with this category of patterns, the activities associated with these patterns, and finally the patterns themselves.

A.1 Structural Patterns

Goal: These patterns define the overall structure for a program.
Output: The overall organization of a program; often represented as an informal picture of a program's high level design. These are the "boxes and arcs" a software architect would write on a whiteboard in describing their design of an application.
Activities: The basic program structure is identified from among the structural patterns. Then the architect examines the "boxes" of the program structure to identify computational kernels.

- **Pipe-and-filter:** These problems are characterized by data flowing through modular phases of computation. The solution constructs the program as filters (computational elements) connected by pipes (data communication channels). Alternatively, they can be viewed as a graph with computations as vertices and communication along edges. Data flows through the succession of stateless filters, taking input only from its input pipe(s), transforming that data, and passing the output to the next filter via its output pipe

- **Agent and Repository:** These problems are naturally organized as a collection of data elements that are modified at irregular times by a flexible set of distinct operations. The solution is to structure the computation in terms of a single centrally-managed data repository, a collection of autonomous agents that operate upon the data, and a manager that schedules the agents' access to the repository and enforces consistency.

- **Process control:** Many problems are naturally modeled as a process that either must be continuously controlled; or must be monitored until completion. The solution is to define the program analogously to a physical process control pipeline: sensors sense the current state of the process to be controlled; controllers determine which actuators are to be affected; actuators actuate the process. This process control may be continuous and unending (e.g. heater and thermostat), or it may have some specific termination point (e.g. production on assembly line).

- **Event-based implicit invocation:** Some problems are modeled as a series of processes or tasks which respond to events in a medium by issuing their own events into that medium. The structure of these processes is highly flexible and dynamic as processes may know nothing about the origin of the events, their orientation in the medium, or the identity of processes that receive events they issue. The solution is to represent the program as a collection of agents that execute asynchronously: listening for events in the medium, responding to events, and issuing events for other agents into the same medium. The architecture enforces a high level abstraction so invocation of an event for an agent is implicit; i.e. not hardwired to a specific controlling agent.

- **Model-view-controller:** Some problems are naturally described in terms of an internal data model, a variety of ways of viewing the data in the model, and a series of user controls that either change the state of the data in the model or select different views of the model. While conceptually simple, such systems become complicated if users can directly change the formatting of the data in the model or view-renderers come to rely on particular formatting of data in the model. The solution is to segregate the software into three modular components: a central data model which contains the persistent state of the program; a controller that manages updates of the state; and one or more agents that export views of the model. In this solution the user cannot modify either the data model or the view except through public interfaces of the model and view respectively. Similarly the view renderer can only access data through a public interface and cannot rely on internals of the data model.

- **Iterative refinement:** Some problems may be viewed as the application of a set of operations over and over to a system until a predefined goal is realized or constraint is met. The number of applications of the operation in question may not be predefined, and the number of iterations through the loop may not be able to be statically determined. The solution to these problems is to wrap a flexible iterative framework around the operation that operates as follows: the iterative computation is performed; the results are checked against a termination condition; depending on the results of the check, the computation completes or proceeds to the next iteration.

- **Map reduce:** For an important class of problems the same function may be applied to many independent data sets and the final result is some sort of summary or aggregation of the results of that application. While there are a variety of ways to structure such computations, the problem is to find the one that best exploits the computational efficiency latent in this structure. The solution is to define a program structured as two distinct phases. In phase one a single function is mapped onto independent sets of data. In phase two the results of mapping that function on the sets of data are reduced. The reduction may be a summary computation, or merely a data reduction.

- **Layered systems:** Sophisticated software systems naturally evolve over time by building more complex operations on top of simple ones. The problem is that if each successive layer comes to rely on the implementation details of each lower layer then such systems soon become ossified as they are unable to easily evolve. The solution is to structure the program as multiple layers in such a way that enforces a separation of concerns. This separation should ensure that: (1) only adjacent layers interact and (2) interacting layers are only concerned with the interfaces presented by other layers. Such a system is able to evolve much more freely.

- **Puppeteer:** Some problems require a collection of agents to interact in potentially complex and dynamic ways. While the agents are likely to exchange some data and some reformatting is required, the interactions primarily involve the coordination of the agents and not the creation of persistent shared data. The solution is to introduce a manager to coordinate the interaction of the agents, i.e. a puppeteer, to centralize the control over a set of agents and to manage the interfaces between the agents.

- **Arbitrary static task graph:** Sometimes it's simply not clear how to use any of the other structural patterns in OPL, but still the software system must be architected. In this case, the last resort is to decompose the system into independent tasks whose pattern of interaction is an arbitrary graph. Since this must be expressed as a fixed software structure, the structure of the graph is static and does not change once the computation is established.

A.2 Computational Patterns

Goal: These patterns define the computations carried out by the components that make up a program.
Output: Definitions of the types of computations that will be carried out. In some cases, specific library routines will be defined.
Activities: The key computational kernels are matched with computational patterns. Then the architect examines how the identified computational patterns should be implemented. This may lead to another iteration through structural patterns, or a move downward in the hierarchy to algorithmic strategy patterns.

- **Backtrack, branch and bound:** Many problems are naturally expressed as either the search over a space of variables to find an assignment of values to the variables that resolves a Yes/No question (a decision procedure) or assigns values to the variables that gives a maximal or minimal value to a cost function over the variables, respecting some set of constraints. The challenge is to organize the search such that solutions to the problem, if they exist, are found, and the search is performed as computationally efficiently as possible. The solution strategy for these problems is to impose an organization on the space to be searched that allows for sub-spaces that do not contain solutions to be pruned as early as possible.

- **Circuits:** Some problems are best described as Boolean operations on individual Boolean values or vectors (bit-vectors) of Boolean values. The most direct solution is to represent the computation as a combinational circuit and, if persistent state is required in the computation, to describe the computation as a sequential circuit: that is, a mixture of combinational circuits and memory elements (such as flip-flops).

- **Dynamic programming:** Some search problems have the additional characteristic that the solution to a problem of size N can always be assembled out of solutions to problems of size $\leq N - 1$. The solution in this case is to exploit this property to efficiently explore the search space by finding solutions incrementally and not looking for solutions to larger problems until the solutions to relevant sub-problems are found.

- **Dense linear algebra:** A large class of problems expressed as linear operations applied to matrices and vectors for which most elements are non-zero. 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 execute close to their theoretically allowed peak performance. Applications of this pattern typically use standard building defined in terms of the dimensions of the dense arrays with vectors (BLAS level 1), matrix-vector.

- **Sparse Linear Algebra:** This includes a large class of problems expressed in terms of linear operations over sparse matrices (i.e. matrices for which it is advantages to explicitly take into account the fact that many elements are zero). Solutions are diverse and include a wide range of direct and iterative methods.

- **Finite state machine:** Some problems have the character that a machine needs to be constructed to control or arbitrate a piece of real or virtual machinery. Other problems have the character that an input string needs to be scanned for syntactic correctness. Both problems can be solved by creating a finite-state machine that monitors the sequence of input for correctness and may, optionally, produce intermediate output.

- **Graph algorithms:** A broad range of problems are naturally represented as actions on graphs of vertices and edges. Solutions to this class of problems involve building the representation of the problem as a graph, and applying the appropriate graph traversal or partitioning algorithm that results in the desired computation.

- **Graphical models:** Many problems are naturally represented as graphs of random variables, where the edges represent correlations between variables. Typical problems include inferring probability distributions over a set of hidden states, given observations on a set of observed states observed states, or estimating the most likely state of a set of hidden states, given observations. To address this broad class of problems is an equally broad set of solutions known as graphical models.

- **Monte Carlo:** Monte Carlo approaches use random sampling to understand properties of large sets of points. Sampling the set of points produces a useful approximation to the correct result.

- **N-body:** Problems in which the properties of each member of a system depends on the state of every other member of the system. For modest sized systems, computing each interaction explicitly for every point is feasible (a naïve $O(N^2)$ solution). In most cases, however, the arrangement of the members of the system in space is used to define an approximation scheme that produces an approximate solution for a complexity less than the naïve solution.

- **Spectral methods:** These problems involve systems that are defined in terms of more than one representation. For example, a periodic sequence in time can be represented as a set of discrete points in time or as a linear combination of frequency components. This pattern addresses problems where changing the representation of a system can convert a difficult problem into a straightforward algebraic problem. The solutions depend on an efficient mechanism to carry out the transformation such as a fast Fourier transform.

- **Structured mesh:** These problems represent a system in terms of a discrete sampling of points in a system that is naturally defined by a mesh. For a structured mesh, the points are tied to the geometry of the domain by a regular process. Solutions to these problems are computed for each point based on computations over neighborhoods of points (explicit methods) or as solutions to linear systems of equations (implicit methods)
- **Unstructured mesh:** Some problems that are based on meshes utilize meshes that are not tightly coupled to the geometry of the underlying problems. In other words, these meshes are irregular relative to the problem geometry. The solutions are similar to those for the structured mesh (i.e. explicit or implicit) but in the sparse case, the computations require gather and scatter operations over sparse data.

A.3 Parallel Algorithm Strategy Patterns

Goal: These patterns describe the high level strategies used when creating the parallel algorithms used to implement the computational patterns.
Output: Definition of the algorithms and choice of concurrency to be exploited.
Activities: Once the pattern for a key computation is identified, there may be a variety of different ways to perform that computation. At this step the architect chooses which particular algorithm, or family of algorithms, will be used to implement this computation. Also, this is the stage where the opportunities for concurrency, which are latent in the computation, are identified. Trade-offs among different algorithms and strategies will be examined in attempt to identify the best match to the computation at hand.

- **Task parallelism:** These problems are characterized in terms of a collection of activities or tasks. The solution is to schedule the tasks for execution in a way that keeps the work balanced between the processing elements of the parallel computer and manages any dependencies between tasks so the correct answer is produced regardless of the details of how the tasks execute. This pattern includes the well known embarrassingly parallel pattern (no dependencies).
- **Divide and Conquer:** A problem is solved by splitting it into a number of smaller subproblems, solving them independently in parallel, and merging the subsolutions into a solution for the whole problem. The subproblems can be solved directly, or they can in turn be solved using the same divide-and-conquer approach, leading to an overall recursive program structure.
- **Data parallelism:** Some problems are best understood as parallel operations on the elements of a data structure. When the operations are for the most part uniformly applied to these elements, an effective solution is to treat the problem as a single stream of instructions applied to each element. This pattern can be extended to a wider range of problems by defining an index space and then aligning both the parallel operations and the data structures around each point in the index space (see the "kernel parallel" pattern).
- **Pipeline:** For these problems consist of a stream of data elements and a serial sequence of transformations to apply to these elements. On initial inspection, there appears to be little opportunity for concurrency. If the processing for each data element, however, can be carried out concurrently with that for the other data elements, the problem can be solved in parallel by setting up a series of fixed coarse-grained tasks (stages) with data flowing between them in an assembly-line like manner. The solution starts out serial as the first data element is handled, but with additional elements moving into the pipeline, concurrency grows up to the number of stages in the pipeline (the so-called depth of the pipeline)
- **Discrete event:** Some problems are defined in terms of a loosely connected sequence of tasks that interact at unpredictable moments. The solution is to setup an event handler infrastructure of some type and then launch a collection of tasks whose interaction is handled through the event handler. The handler is an intermediary between tasks, and in many cases the tasks do not need to know the source or destination for the events. This pattern is often used for GUI design and discrete event simulations.
- **Geometric decomposition:** 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.

- **Speculation:** The problem contains a potentially large number of tasks that can usually run concurrently; however, for a subset of the tasks unpredictable dependencies emerge and these make it impossible to safely let the full set of tasks run concurrently. An effective solution may be to just run the tasks independently, that is speculate that concurrent execution will be committed, and then clean up after the fact any cases where concurrent execution was incorrect. Two essential element of this solution are: 1) to have an easily identifiable safety check to determine whether a computation can be committed and 2) the ability to rollback and re-compute the cases where the speculation was not correct.

A.4 Implementation Strategy Patterns

Goal: These patterns focus on how a software architecture is implemented in software. They describe how threads or processes execute code within a program; i.e. they are intimately connected with how an algorithm design is implemented in source code. These patterns fall into two sets: program structure patterns and data structure patterns.
Output: Pseudo-code defining how a parallel algorithm will be realized in software.
Activities: This is the stage where the broad opportunities for concurrency identified by the parallel algorithmic strategy patterns are mapped onto particular software constructs for implementing that concurrency. Advantages and disadvantages of different software constructs will be weighed.

- Program structure
  - Single-Program Multiple Data (SPMD): Keeping track of multiple streams of instructions can be very difficult for a programmer. If each instruction stream comes from independent source code, the software can quickly become unmanageable. There are a number of solutions to this problem. One is to have a single program (SP) that is used for all of the streams of instructions. A process/thread ID (or rank) is defined for each instance of the program and this can be used to index into multiple data sets (MD) or branch into different sub-sets of instructions. Most instances of this pattern use the Bulk Synchronous Programming model (BSP) described in [15]. A problem using the BSP model is decomposed into a sequence of "super-steps" each of which consists of a computation phase followed by a communication phase.
  - Fork/join: The problem is defined in terms of a set of functions or tasks that execute within a shared address space. The solution is to logically create threads (fork), carry out concurrent computations, and then terminate them after possibly combining results from the computations (join). This pattern is also used with fine grained "threads" or "tasks" that are forked and then placed into queues from which they are processed by a collection of opaque worker threads (e.g. Cilk [7]).
  - Kernel Parallel: This pattern is a generalization of the data parallel pattern. An index space is defined onto which the data structures within a problem are mapped. The computations of the problem are organized into kernels. A sequence of kernels is submitted for execution each of which executes concurrently at each point in the index space. In essence, this is a fine-grained expression of the SPMD pattern tuned to the needs of data parallel algorithms.
  - Loop-level parallelism: The problem is expressed in terms of a modest number of compute intensive loops. The loop iterations can be transformed so they can safely execute independently. The solution is to transform the loops as needed to support safe concurrent execution, and then replace the serial compute intensive loops with parallel loop constructs (such as the "for worksharing construct" in OpenMP). A common goal of these solutions is to create a single program that executes in serial using serial compilers or in parallel using compilers that understand the parallel loop construct.
  - Vector parallel: For a subset of data parallel algorithms, a sequence of operations are applied to the elements of a vector. These operations are strictly data parallel by which we mean the processing elements involved in the computation all execute the same instructions "in lock-step" to blocks of a vector (though with masks you can turn off operations on specific lanes of a vector unit). This can be extended to a wider range of algorithms through data parallel collective operations such as a prefix scan. The vector-parallel pattern is often used to provide a finer grained "processor-level" parallelism in combination with a larger scale parallel pattern applied across the nodes of a parallel platform.

- Actors: An important class of object oriented programs represents the state of the computation in terms of a set of persistent objects. These objects encapsulate the state of the computation and include the fundamental operations to solve the problem as methods for the objects. In these cases, an effective solution to the concurrency problem is to make these persistent objects distinct software agents (the actors) that interact over distinct channels (message passing).
- Work-pile: A common problem in parallel programming is how to balance the computational load among a set of processing elements within a parallel computer. For task parallel programs with no communication between tasks (or infrequence but well-structured, anonymous communication) an effective solution with "automatic dynamic load balancing" is to: (1) place the tasks (the "work") into a data structure (a "work-pile" such as a queue) and then (2) have a set of "workers" grab a task, do the work, and store the results. This continues until all the tasks have been computed. When the work-pile and the results of the computations are managed by a distinct process, this is the well-known "master-worker" pattern.

- Data Structure Patterns
  - Shared queue: Some problems generate streams of results that must be handled in some predefined order. It can be very difficult to safely put items into the stream or pull them off the stream when concurrently executing tasks are involved. The solution is to define a shared queue where the safe management of the queue is built into the operations upon the queue.
  - Shared Map: A wide range of problems are organized around data structures composed of a set of key-value pairs. These data structures are called maps or in some programming languages, dictionaries. As with the partitioned array pattern, the problem is the indexing required to transform a global key into a local key for a particular member of the set of processes or threads involved with a parallel computation. The solution is to place the indexing operations inside a method associated with a map abstract data type to insulate this complexity from the application program and support reuse between related programs.
  - Shared data: Programmers should always try to represent data shared between threads or processes as shared data types with a well-defined API to hide the complexity of safe concurrent access to the data. In some cases, however, this just is not practical. The solution is to put data into a shared address space and then define synchronization protocols to protect that data.
  - Partitioned array: The array is a critical data structure in many problems. Operating on components of the array concurrently (for example, using the geometric decomposition pattern) is an effective way to solve these problems in parallel. Concurrent computations may be straightforward to define, but defining how the array is decomposed among a collection of processes or threads can be very difficult. In particular, solutions can require complex book-keeping to map indices between global indices in the original problem domain and local indices visible to a particular thread or process. The solution is to define a partitioned array and fold the complicated index algebra into access methods on the partitioned array data type. The programmer still needs to handle potentially complex index algebra, but it's localized to one place and can possibly be reused across programs that use similar array data types.
  - Partitioned Graph: A graph is typically a single monolithic structure with edges indicating relations among vertices. The problem is how to organize concurrent computation on this single structure in such a way that computations on many parts of the graph can be done concurrently. The solution is to find a strategy for partitioning the graph such that synchronization is minimized and the workload is balanced.

A.5 Parallel Execution Patterns

Goal: These patterns describe how a parallel algorithm is organized into software elements that execute on real hardware and interact tightly with a specific programming model.
Output: Should produce particular approaches to exploit the hardware capabilities for parallelism so that we can execute programs efficiently.
Activities: This is the stage where the previously identified software constructs are matched up with the actual execution capability of the underlying hardware. At this point the performance of the underlying hardware mechanisms may be known and the advantages and disadvantages of different mappings to hardware can be precisely measured.

- **Coordinating Processes:** The problem is expressed in terms of a set of long-lived (i.e. coarse-grained) tasks operating concurrently on their own streams of data. The solution is to construct the parallel program as sequential processes that execute independently and coordinate their execution through discrete communication events.
- **Stream processing:** For a large class of data parallel applications, the parallel algorithm can be viewed as a large collection of kernel-instances streaming through the processing elements of a platform. The runtime system uses the fact that (1) the number of kernel-instances is much greater than the number of processing elements to hide memory access latencies and (2) many of the kernel instances are running the same sequence of operations (a convergent control flow) which can be mapped onto specialized SIMD processing elements for greater performance.
- **Shared Address Space Threads:** Programs designed for shared memory computers usually organize computations as a dynamic set of threads that share an address space. To support these computations, a runtime system provides utilities to create threads (e.g. fork), destroy threads (e.g. join), synchronize threads to safely constrain updates to memory, and reuse threads (thread-pools).
- **Task Driven Execution:** Parallel algorithms generate tasks. In some algorithms, the runtime system supporting a programming environment is best organized directly around these tasks. The tasks are defined by the application, accumulated by the runtime system, and then scheduled for execution. The scheduler can be driven by the availability of data needed by a task (data-flow) or tasks ready to execute can be accumulated into some variety of queue and executed as resources become available. The key is to define low overhead schedulers that balance the load between processing elements.

PyCASP: A Pattern-oriented Framework for Parallel Programming

Ekaterina Gonina and Gerald Friedland and Kurt Keutzer

Abstract

In this paper we describe PyCASP: a pattern-oriented framework for multimedia content analysis applications. PyCASP aims to give domain experts in multimedia content analysis a high productivity environment for designing and implementing applications on parallel hardware. To achieve this, PyCASP takes the form of a pattern-oriented framework. A pattern-oriented framework is a software environment in which all software customization is in harmony with a software architecture. Software architecture, in turn, is defined in terms of a composition of design patterns. Using PyCASP a domain expert may express a complete application, such as meeting diarization, music recommendation or video event detection in Python code, and PyCASP will compile that code for efficient execution on a variety of parallel targets – ranging from desktop CPUs and GPUs to a cluster. We show that using PyCASP, content analysis applications can be captured in 50-400 lines of Python code and achieve up to 115× faster-than-real-time performance on Nvidia GPUs (in the case of speaker diarization) and 15.5× speedup on 16-node cluster (in the case of video event detection). PyCASP is currently used by domain researchers at the International Computer Science Institute (ICSI) to perform acoustic model training and analysis of video content. It is also used at Intel Corporation to explore power optimization techniques and the capabilities of Intel-based platforms with real multimedia applications.

1 Introduction

We believe that in many application areas, the software programming population is bifurcating into domain experts interested in productive programming environments and efficiency programmers who build highly tuned environments to support domain experts. The demands of achieving domain expertise in contemporary computer science research areas such as computer vision, multimedia analysis, or machine learning are sufficiently great that even computer scientists may not find the time to develop expert skills in optimizing software for the details of a particular platform. This lack of expertise is even more evident in areas outside of computer science such as computational finance or computational biology. Domain experts in all the aforementioned areas are likely to do much or all of their application development in high productivity programming environments such as MATLAB [30], R [32] or Python [34]. These domain experts may have limited programming skills and so the question arises as to how to best support them and allow them to utilize parallel hardware.

2 Alternative Solutions

A variety of mechanisms have been proposed and employed for improving the productivity of software development and improving the efficiency of the resulting code. These include libraries, frameworks and domain specific languages (DSLs).

2.1 Libraries

Libraries, such as OpenCV [21] for computer vision and the BLAS [6] for linear algebra, define standard interfaces to particular computations with well defined results. The user of the library focuses on the interface and the results, with little or no need to understand the internals of the library. Hence, libraries have proven to be an effective solution to the problem of how to expose both productivity and efficiency to domain specialist programmers.

Libraries also have a variety of recurrent challenges. One is final code efficiency. Nearly every problem has different input distributions. If the input distribution is not fully general, a hand-tuned solution for a particular distribution may produce much more efficient code. Also, libraries with aggressive compiler optimization and auto-tuning such as ATLAS [39] and OSKI [38] can be used to enhance both the portability and efficiency of library code by tuning the library code to a particular platform. While the clarity of the interface of libraries is a strength, the brittleness and narrowness of the interface can also be a significant limitation. If a library comes close to providing the software solution required, then programmers will invest some effort into contorting their problem in such a way that a library solution can be employed; however, if the effort becomes too large then the library solution will be scrapped and programmers will craft their own custom solution.

2.2 Frameworks

Application frameworks, assist programmers by providing guidance on how to design applications as a whole to allow for most efficient and scalable implementation. There are a variety of ways of approaching the design of frameworks, such as [16], [18], and [25]. Following Johnson [25], frameworks are envisioned as a kind of domain-specific architecture and a practical way to express reusable design. In Johnson's approach a framework consists of the design, guided by design patterns and a set of components - the frameworks computational building blocks. The design guides the use and composition of components. Programmers can develop their applications using the framework that guides them in the composition of the various components of the framework in a pre-defined way. Some examples of successful frameworks are Cactus [20] and CHARMS [22].

While frameworks present a promising solution for creating reusable tools, their adoption is limited due to framework complexity, low-level interfaces, and poor efficiency of generated code. To bring both productivity and efficiency to this approach, several projects have focused on using both a high-level language and a low-level implementation for computational components. Specifically, several frameworks provide a Python-level interface to computations written in highly-optimized low-level code. Such frameworks include Numpy [2] and Gnumpy [37] for performing dense matrix operations on multicore CPUs and GPUs respectively. Other frameworks allow for more customization and use of more complex computations such as Theano [4] and Copperhead [8]. All of these frameworks allow for both programmer productivity and application efficiency by abstracting away the complex low-level implementations of compute-intensive components from the application code. These approaches span general domains of dense linear algebra and operations on dense vectors. Frameworks like Numpy, Theano and Copperhead provide a more flexible mechanism for achieving efficiency in applications as well as code reuse. However, due to their general-purpose design, these approaches do not know the application domain within which they will be used and thus cannot implement available domain-specific optimizations. Thus, with the gain in generality, these approaches can lose in efficiency.

2.3 Domain-specific Languages

Domain specific languages (DSLs) attempt to offer more expressive power than library solutions or frameworks. For example HTML5 and XML [29] are DSLs for web-page development, Verilog and VHDL [31, 3] are DSLs for hardware description and SQL [11] is a DSL for database querying. They aspire to provide as much flexibility as standard programming languages but with higher productivity and better domain "feel" By restricting a language to target a particular application domain, domain-specific algorithm optimizations are available to the language designer. At the same time, the DSL provides familiar primitives for the domain experts to use.

The recurrent challenges of domain specific languages is that it is hard for them to gain customer confidence as they are new. It takes quite a bit of DSL designer and community effort to adopt these languages as the default approach. Moreover, in the absence of strong library support their productivity gains may evaporate as users quickly build new parts of their system but find themselves re-implementing standard elements due to missing libraries. DSLs, due to their design, present a restricted set of tools to the programmer, making it difficult to extend and modify existing functionality. Thus, the domain expert is "locked-in" to the DSL environment and the existing functionalities.

Libraries, frameworks, DSLs, and other mechanisms for providing programming environments are discussed in the particular context of parallel programming in [23].

3 Our Approach: Pattern-Oriented Frameworks

A challenge for the designers of software development tools has been to find a "sweet spot" that restricts programmer's freedom in a way that eases program composition and verification, but provides enough capabilities to allow applications to be fully and easily described.

Taking inspiration from Ruby on Rails [35] and its use of the model-view-controller pattern [26], our approach to this challenge is to build *pattern-oriented frameworks* that allow user customizability only insofar as it is harmonious with the initial architecture. For example, in Ruby on Rails all user customization must be in harmony with the initial mode-view-controller software architecture. While it may seem quite restrictive to constrain all programs to follow this single architecture, experience has shown that Ruby on Rails is very useful for a variety of problems in web programming.

We aim to apply this approach to the problem of multimedia content analysis; however, we immediately face the question: what is the appropriate software architecture to support this range of applications? Or, noting that a software architecture is simply a hierarchical composition of patterns, we may phrase the question as: what is a minimal set of structural and application patterns that will allow for easy and efficient application development of applications for multimedia domain? The goal of minimality is not motivated by some mathematical purism; it is motivated by the desire to minimize what users must learn. The goals of easy and efficient development are self-evident. Finally, by embedding the framework into an existing high-level language (Python), we allow the domain programmer to program in an environment they are familiar with as well as to integrate other existing tools that are out there without having to modify the framework.

4 Mining Patterns in Multimedia Content Analysis

We explored the question of a necessary and sufficient set of patterns to define a framework for multimedia content analysis applications by examining, or pattern mining, a large set of representative multimedia content analysis applications. In this section, we describe the application domain and then dive into the pattern-mining approach for the framework design.

4.1 Multimedia Content Analysis Applications

Multimedia content analysis applications implement algorithms that extract information from multimedia data such as audio and video. With hundreds of videos and images being uploaded to the web every minute [28], there is a high demand for scalable solutions to large-scale multimedia content analysis. For example, appealing multimedia content analysis applications include automatic video and audio transcription and search, recommendation of new content, and tools for geo-location and privacy analyses. Accurate and robust applications require training on hundreds of thousands of learning examples requiring hours or days of processing time. In addition, such applications require real-time processing when integrated into interactive environments such as home entertainment systems and mobile applications, which require fast, low-latency, portable solutions to multimedia processing. With such intensive application demands this domain presents a lucrative target for an application framework that allows for automatic parallelization and scaling on a variety of parallel hardware.

4.2 Patterns in Multimedia Content Analysis

To define a necessary and sufficient set of patterns to create a framework for multimedia applications we looked at prior efforts in pattern-oriented frameworks for automatic speech recognition [13, 27]. In this work, a pattern-oriented framework was designed around a software architecture typical in the Automatic Speech Recognition (ASR) domain. Specifically, the framework implements the Viterbi algorithm for performing speech inference using Hidden Markov Models (HMMs) for modeling words and Gaussian Mixture Models (GMMs) for modeling acoustic information of speech. The framework implemented a simple ASR system using the GMM and HMM computations and composed

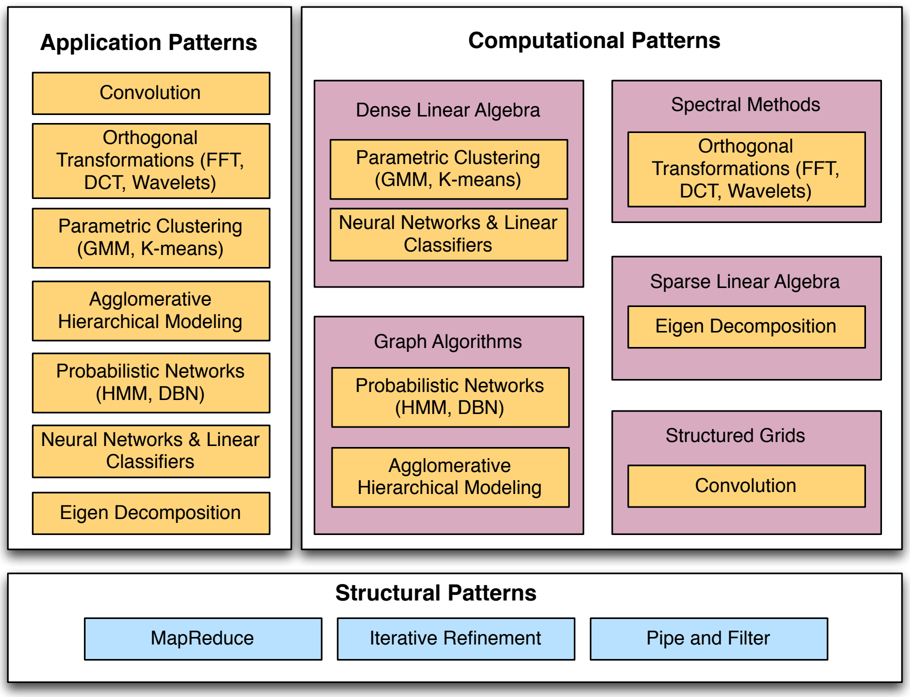
Application Patterns
Convolution
Orthogonal Transformations (FFT, DCT, Wavelets)
Parametric Clustering (GMM, K-means)
Agglomerative Hierarchical Modeling
Probabilistic Networks (HMM, DBN)
Neural Networks & Linear Classifiers
Eigen Decomposition
Computational Patterns
Dense Linear Algebra
Parametric Clustering (GMM, K-means)
Neural Networks & Linear Classifiers
Spectral Methods
Orthogonal Transformations (FFT, DCT, Wavelets)
Sparse Linear Algebra
Eigen Decomposition
Graph Algorithms
Probabilistic Networks (HMM, DBN)
Agglomerative Hierarchical Modeling
Structured Grids
Convolution
Structural Patterns
MapReduce
Iterative Refinement
Pipe and Filter

Figure 1: Application, computational and structural patterns for design of PyCASP.

them using the Pipe-and-Filter, Iterator, and Map-Reduce patterns. The framework provided extension points to customize the system for a specific application by changing the format and computation of the observation (acoustic model) probability, pruning strategy and the output format. This framework seemed like a good start for providing domain experts with tools to create ASR applications that automatically scaled to parallel processors. However, the fixed software architecture was clearly not flexible enough to support the broader range of multimedia applications that we envision.

Instead, we set out to create a modular framework that consists of a set of customizable *components* that can be more flexibly *composed* together using structural patterns. In order to create such a framework that will comprehensively span a variety of multimedia applications, we first extensively studied applications in the multimedia content analysis domain by implementing applications on a variety of parallel platforms [15, 14, 40, 19, 1, 17] as well as discussing the applications with domain experts. By mining the applications we discovered a set of application patterns common to multimedia content analysis, shown in the left panel of Figure 1.

After identifying the application patterns in our application domain, we were able to distill them into five (out of thirteen total [26]) distinct computational patterns (right panel of Figure 1) in order to reason about the underlying computation of each application pattern. After pattern mining and reasoning through the computations in our domain, we focus on mining the applications to identify the set of structural patterns (also known as architectural styles [36]) that they use. A structural pattern defines a way in which application patterns are composed into a software architecture [26]. We identified three (out of ten total) structural patterns that are sufficient to describe software architectures of applications in the multimedia content analysis domain, shown at the bottom of Figure 1.

Having identified the core application, computational, and structural patterns, we embedded our framework in Python to allow us to develop a productive, modular software environment that is concise and comprehensive enough to allow for development of a variety of applications in the domain. By mining the primary application and structural patterns present in the multimedia content analysis applications, we provide the basic computational building blocks and mechanisms for their composition that are familiar to the application developers. This pattern mining effort has its own value and may be useful to others who wish to follow related approaches to providing frameworks for an application domain. We also support customization of the application logic as well as automatic tuning to particular hardware platforms using the Selective Embedded Just-in-Time Specialization (SEJITS) approach [9].

5 PyCASP: A Pattern-Oriented Framework for Multimedia Applications

We call our framework PyCASP (Python-based Content Analysis using SPecialization), a pattern-oriented framework for multimedia applications.PyCASP is designed for application developers and researchers with limited knowledge of parallel hardware. The goal of PyCASP is to present the domain expert with the productivity of a high-level language but the performance and portability to a variety of modern parallel system more commonly associated with a low level "efficiency" language.

To aide in productivity, PyCASP uses a pattern-oriented approach described in [26] with the goal of creating a comprehensive, modular software environment for application writers. Using this approach, we mined a variety of multimedia applications to identify a set of core patterns in the domain. This allows PyCASP to have a tractable scope and yet to be applicable to a wide variety of multimedia content analysis applications.

We use the Selective Embedded JIT Specialization (SEJITS) approach [9] to give efficiency and portability to applications written using PyCASP. SEJITS separates the concerns of productivity programmers from those of efficiency programmers to enable automatic parallelization. PyCASP consists of a set of SEJITS-based components that implement the application patterns as well as a high-level mechanisms for their composition using structural patterns (Pipe-and Filter, Iterator and MapReduce).

To illustrate our approach, we describe a set of multimedia analysis applications, their software architecture, and their implementation using PyCASP's Gaussian Mixture Model (GMM) parametric clustering and Support Vector Machine (SVM) linear classification components. Programmers have the ability to customize their code when using these two components. When using the GMM component, they can select which platform to run the computation on (NVIDIA GPU or Intel CPU), whether to autotune the code(select the best parallel code variant), set the number of iterations of the GMM training and initial set of parameter values. When using the SVM components, the programmers can select the kernel function, convergence threshold and point selection heuristic (see [10] for details of the algorithm) as well as the GPU to run the computation on.

Using PyCASP, applications can be prototyped in Python code and automatically scale to parallel platforms and achieve high performance. Quantifying productivity is a difficult task, but from our previous work with more traditional software development techniques the same application written in C/C++ would require at least 10-60$\times$ lines of code and 7-30$\times$ time to develop the application.

6 Architecting Multimedia Applications

6.1 Speaker Verification

A speaker verification system automatically determines if a piece of input audio belongs to a particular (target) speaker. The implementation of the speaker verification system is based on the GMM-SVM model described in [7]. After extracting the Mel-Frequency Cepstrum Coefficients (MFCC) features from audio files, a Universal Background Model (UBM) (represented by a GMM) is trained on example audio files from many speakers. Then, the UBM is adapted to the target speaker (Speaker 1), (i.e. the GMM is trained on target speaker audio, initialized with the UBM parameters) and speech from the "intruder" speaker(s) (Speaker 2). Finally, an SVM is trained to distinguish the target speaker audio from other audio using the adapted GMM means. During the classification phase, given a new audio file, the UBM is adapted to the extracted features and the SVM is used to classify the adapted model.

Figure 2 shows the software architecture of the application. At the top level, MFCC, GMM and SVM components are composed using a Pipe and Filter pattern. The MFCC component consist of several signal processing stages linked in a Pipe and Filter architecture. The GMM and SVM components internally are defined as an Iterator pattern which contains a Pipe-and-Filter pattern with each filter using the MapReduce pattern inside of which the computations are described by the Dense Linear Algebra computational pattern.

To customize this system, the MFCC component can be replaced with another audio feature extraction component, for example PLP. GMM component can be replaced with another acoustic modeling component, for example a Neural Network and the SVM component can be replaced with another classifier, for example Linear Regression. These customizations are done in harmony with the initial software architecture and thus provide flexibility to the application writer while maintaining correctness.

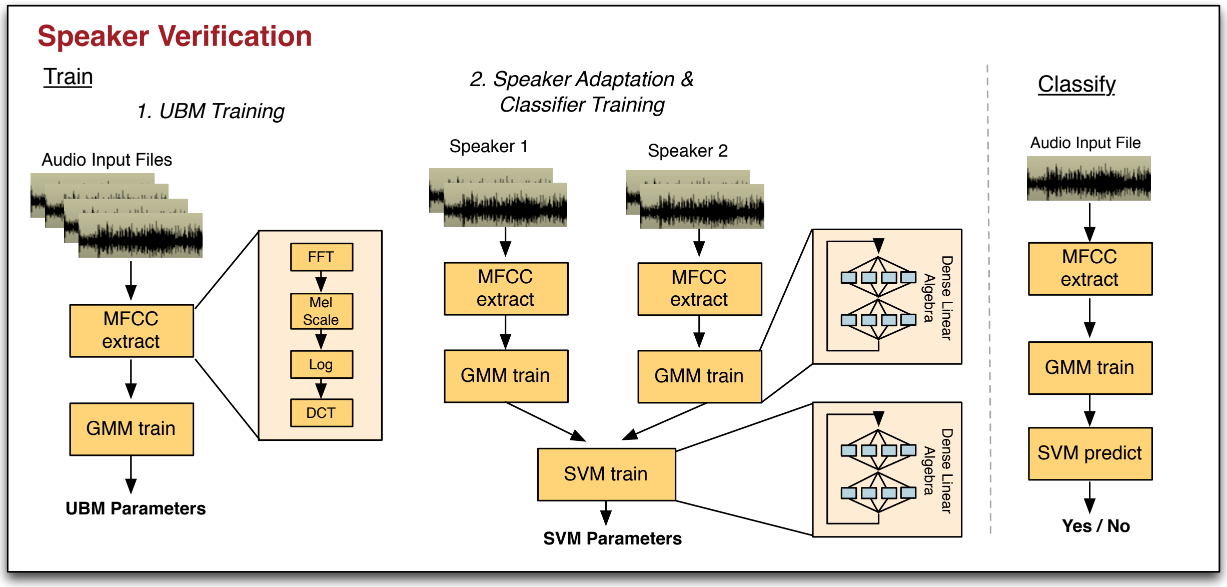

Figure 2: Software Architecture of the Speaker Verification System.

6.2 Speaker Diarization

A speaker diarization system segments an audio recording into speaker-homogeneous regions, addressing the question "who spoke when" We implement one popular diarization method - the Bayesian Information Criterion (BIC) with GMMs trained with MFCC features [1]. This method simultaneously segments the audio and constructs speaker models for each speaker.

Figure 3 shows the software architecture of the application. At the top level, the application consists of an Iterator pattern over a Pipe and Filter of GMM train, GMM evaluation and Segment stages. The MFCC component and GMM training components are the same as in the speaker verification application. GMM evaluation is a Pipe and Filter of MapReduce stages implementing Dense Linear Algebra computations. The Segment stage is a simple Python loop that assigns chunks of audio to particular speakers based on GMM likelihoods from the GMM eval stage. The full implementation of the system was described in [19] and was well-received by the speech recognition community.

To customize this system, the MFCC component can be replaced with another audio feature extraction component, for example PLP. GMM component can be replaced with another acoustic modeling component, for example a Neural Network. These customizations are done in harmony with the initial software architecture and thus provide flexibility to the application writer while maintaining correctness.

For an illustration of the speaker diarization implementation in Python using PyCASP, please see Appendix A

6.3 Music Recommendation

Our music recommendation system returns a set of songs that are most similar to a given song or artist. The system is based on the UBM adaptation approach described in [12]. Similar to the speaker verification system, we first train a UBM on a random subset of all the songs in the database (in our case we are using the Million Song Dataset [5]). After a query comes in, we adapt the UBM on the features that matched the query and use a distance computation to compute the set of closest songs based on the adapted UBM means.

Figure 4 shows the software architecture of the application. At the top level, GMM training and Distance Computation components are composed into a Pipe and Filter architecture. GMM training is an Iterator over a Pipe and Filter of MapReduce stages. The Distance Computation component is a matrix-matrix multiply operation, an instance of Dense Linear Algebra computational pattern.

To customize this system, the GMM component can be replaced with another audio modeling component, for example a Neural Network and the distance computation can be customized to use different normalization metrics. These customizations are done in harmony with the initial software architecture and thus provide flexibility to the

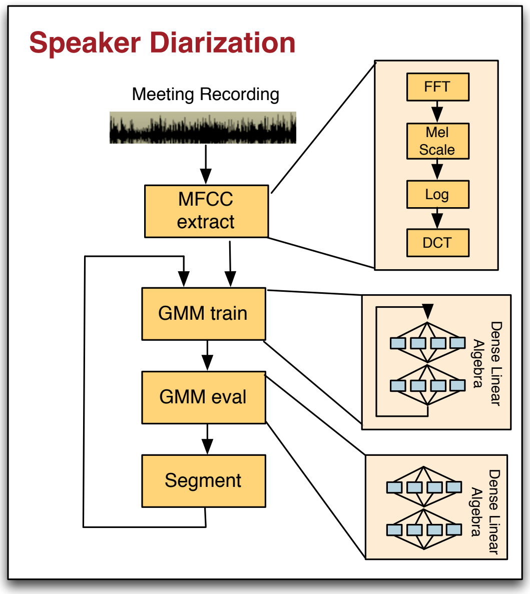

Figure 3: Software Architecture of the Speaker Diarization System.

application writer while maintaining correctness.

6.4 Video Event Detection

A video event detection system aims to identify a set of audio events in a set of videos [17]. The event detection system is generalized from speaker diarization for indexing audio contents. As in the speaker diarization system, we use GMMs to represent the audio concepts. Each video is diarized in parallel using the MapReduce pattern allowing the application to run on a cluster. A K-means method is then used to cluster the audio event vectors found in the videos that were generated from all of the low level acoustic concepts, resulting in clusters that represent audio

Figure 5 shows the software architecture of the system. At the top level the application uses the MapReduce pattern with the Map phase executing diarization on each video and the Reduce phase calling the K-means clustering to come up with the total list of audio events across all videos.

To customize this system, the GMM component can be replaced with another audio modeling component, for example a Neural Network and the K-means component can be replaced with another clustering component, for example Nearest Neighbors. These customizations are done in harmony with the initial software architecture and thus provide flexibility to the application writer while maintaining correctness.

7 Results

Table 1 shows the approximate number of lines of Python code used to implement the four example applications and performance remarks for each. Since we don't have an equivalent serial implementation for all applications, and as performance metrics for each application differ, we present performance remarks to illustrate the efficiency of each application implemented using PyCASP. For example, the speaker diarization application can diarize a meeting $71 - 115\times$ faster-than-real-time[1] on an NVIDIA GTX480 GPU [19]. The video event detection system was run on a cluster of 16 GPUs using the diarization engine and our MapReduce composition allowed for $15.5\times$ speedup compared to sequentially processing the videos on one GPU-equipped machine.

[1](i.e. 1 second of audio is diarized in $1/71$ st to $1/115$th of a second)

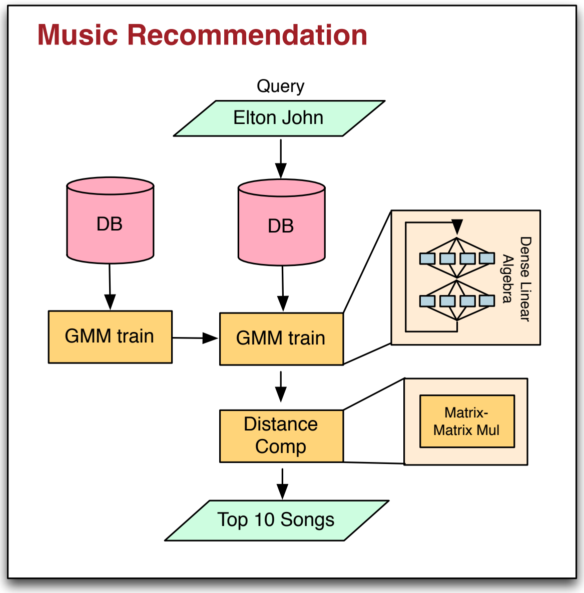

Figure 4: Software Architecture of the Music Recommendation System.

Table 1: Number of lines of Python code and performance remarks for the example applications. Speaker diarization and music recommendation performance is on NVIDIA GTX480 GPU. Video event detection system speedup is relative to sequentially processing videos on one GPU-equipped machine.

	Speaker Verification	**Speaker Diarization**	**Music Recommendation**	**Video Event Detection**
LOC	260	100	600	120
Performance remarks	Train: 13 min speech in 2.4 sec Classify: 5 min speech in 0.3 sec	$71 - 115\times$ Faster-than-real-time	Under 1 second recommendation time for all queries	$15.5\times$ speedup on 16 nodes

8 Domain Experts Experience Using PyCASP

PyCASP is currently used by domain researchers at the International Computer Science Institute (ICSI) to perform acoustic model training and analysis of video content. It is also used at Intel Corporation to demonstrate the hardware capabilities of Intel hardware and energy optimization on real multimedia applications. The ability to prototype a state-of-the-art multimedia application from start to end in Python and at the same time leverage the underlying hardware parallelism has been seen as a tremendous advantage of PyCASP.

9 Limitations of PyCASP

There are some drawbacks of using pattern-oriented application-domain-specific frameworks as developer tools to aide in the spread of parallel programming. First, this approach inherently requires thorough domain analysis and domain engineering. Second, since frameworks are a lot more complex than libraries, developers need a longer time to be trained to use them and require better documentation. Third, SEJITS-based frameworks such as PyCASP are harder and more costly to develop since they require both domain and parallel performance expertise and it is often harder to find engineers that have a deep enough understanding in both areas. Finally, a significant amount of effort

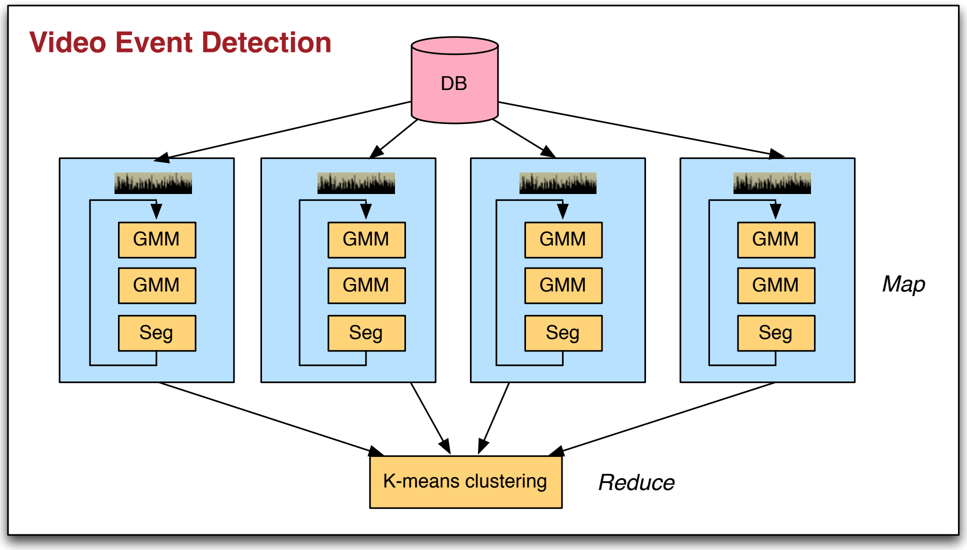

Figure 5: Software Architecture of the Video Event Detection System.

needs to go into developing the framework itself, such that it comprehensively covers the application domain.

10 Conclusions and Future Work

A general goal of the Par Lab has been to understand how design patterns can be used to productively write correct and efficient software. The most advanced result of this direction of research has been to develop pattern-oriented frameworks: frameworks in which all user customization must be in harmony with the original software architecture, which is in turn defined by patterns. To pursue this goal we mined patterns from our earlier work in Automatic Speech Recognition and used those to define a pattern-oriented framework for multimedia content analysis. In particular, we found that we could define the architecture underlying our pattern-oriented framework for multimedia content analysis with only three structural patterns. To implement this framework we chose Python because we found it to be a flexible and popular programming environment. To define the implementation flow we used a pattern-direct approach in combination with SEJITS. Using PyCASP, we found that content analysis applications can be captured in 50-400 lines of Python code and achieve up to 115$\times$ faster-than-real-time performance on Nvidia GPUs (in the case of speaker diarization) and 15.5$\times$ speedup on 16-node cluster (in the case of video event detection). Moreover, we observed that multimedia domain experts from the International Computer Science Institute, as well as programmers from Intel, were able to learn how to use PyCASP quickly and easily.

Future work for PyCASP includes implementing more components as well as creating a set of customizable components that use the SEJITS abstract syntax tree transformation tools to lower Python code to low-level efficiency code. With abstract syntax tree transformation mechanisms of SEJITS, PyCASP's components can allow for much broader customization, further expanding the set of applications that can be implemented using PyCASP. In addition, future work includes further exploration of the composition mechanisms of PyCASP using the structural patterns as well as implementing components and composition mechanisms for large-scale mutlimedia data analysis applications. Development of a successful and broadly used framework is a significant endeavor that requires a community of interested and highly motivated developers. Only time will tell whether PyCASP is able to support such an effort; however, we believe that the principles demonstrated in PyCASP – pattern-oriented frameworks, embedding in a high-level language, and SEJITS for efficient code generation - have general utility.

11 Acknowledgements

Our thanks to Timothy Mattson for giving thoughtful suggestions and detailed comments on this chapter. Our thanks to Henry Cook for initial work on the implementation of the GMM specializer and to Shoaib Kamil and Armando Fox for providing the Asp infrastructure and insightful discussions during this work. Also our thanks to Penporn Koanantakool, Michael Driscoll and Evangelos Georganas for their work on the MapReduce specializer used in the video event detection application and Eric Battenberg for providing insights on the music recommendation application.

Bibliography

[1] X. Anguera, S. Bozonnet, N. W. D. Evans, C. Fredouille, G. Friedland, and O. Vinyals. Speaker diarization : A review of recent research. *IEEE Transactions On Acoustics Speech and Language Processing" (TASLP), special issue on "New Frontiers in Rich Transcription"*, 20:356–370, 2012.

[2] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition, 1999.

[3] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral.

[5] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[7] W. Campbell, D. Sturim, D. Reynolds, and A. Solomonoff. Svm based speaker verification using a gmm supervector kernel and nap variability compensation. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, page I, may 2006.

[8] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[9] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

[10] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 104–111, New York, NY, USA, 2008. ACM.

[11] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[12] C. Charbuillet, D. Tardieu, and G. Peeters. Gmm supervector for content based music similarity. In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx 2011)*, 2011.

[13] J. Chong. *Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2010.

[14] J. Chong, G. Friedland, A. Janin, N. Morgan, and C. Oei. Opportunities and challenges of parallelizing speech recognition. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[15] J. Chong, E. Gonina, Y. Yi, and K. Keutzer. A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit. In *10th Annual Conference of the International Speech Communication Association (InterSpeech)*, September 2009.

[16] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Professional, 1998.

[17] B. Elizalde, G. Friedland, H. Lei, and A. Divakaran. There is no data like less data: Percepts for video concept detection on consumer-produced media. In *1st ACM Workshop on Audio and Multimedia Methods for Large-Scale Video Analysis, ACM Multimedia*, Nara, Japan, November 2012.

[18] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[19] E. Gonina, G. Friedland, H. Cook, and K. Keutzer. Fast speaker diarization using a high-level scripting language. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 553 –558, dec. 2011.

[20] T. Goodale, G. Allen, G. Lanfermann, J. MassŮ, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *In High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference*, pages 26–28. Springer.

[21] V. W. Gregory. *Programmers tool chest: the OpenCV library*. Dr. Dobbs Journal, Nov 2000.

[22] E. Grinspun, P. Krysl, and P. SchrŽder. Charms: A simple framework for adaptive simulation. In *ACM Transactions on Graphics*, pages 281–290. ACM Press, 2002.

[23] W.-m. Hwu, K. Keutzer, and T. G. Mattson. The concurrency challenge. *IEEE Des. Test*, 25(4):312–320, July 2008.

[24] D. Imseng and G. Friedland. Robust speaker diarization for short speech recordings. In *Proceedings of the IEEE workshop on Automatic Speech Recognition and Understanding*, pages 432–437, 12 2009.

[25] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, Oct. 1997.

[26] K. Keutzer and T. G. Mattson. A design pattern language for engineering (parallel) software. In *Intel Technology Journal*, volume 4, 2010.

[27] D. Kolossa, J. Chong, S. Zeiler, and K. Keutzer. Efficient manycore chmm speech recognition for audiovisual and multistream data. In T. Kobayashi, K. Hirose, and S. Nakamura, editors, *INTERSPEECH*, pages 2698–2701. ISCA, 2010.

[28] A. Kosner. Youtube turns seven today, now uploads 72 hours of video per minute, May 2012.

[29] E. Maler, J. Paoli, C. M. Sperberg-McQueen, F. Yergeau, and T. Bray. Extensible markup language (XML) 1.0 (third edition). first edition of a recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[30] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[31] S. Palnitkar. *Verilog&#174; hdl: a guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[32] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[33] D. Reynolds and P. Torres-Carrasquillo. Approaches and applications of audio diarization. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 5, pages v/953 – v/956 Vol. 5, march 2005.

[34] G. Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[35] Ruby On Rails. *Ruby on Rails Manual*, March 2013. Version 3.2.13.

[36] M. Shaw and D. Garlan. Software architecture: perspectives on an emerging discipline. 1996.

[37] T. Tieleman. Gnumpy: an easy way to use GPU boards in Python. Technical Report UTML TR 2010-002, University of Toronto, Department of Computer Science, 2010.

[38] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[39] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[40] K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y. Chen, W. Sung, and K. Keutzer. Parallel scalability in speech recognition: Inference engine in large vocabulary continuous speech recognition. In *IEEE Signal Processing Magazine*, number 6, pages 124–135, November 2009.

Appendix A Speaker Diarization Code Example

As mentioned in Section 6.2, our speaker diarization method combines the speech segmentation and segment clustering tasks into a single stage using agglomerative hierarchical clustering, a process by which many simple candidate models are iteratively merged into more complex, accurate models.

The diarization is based on 19-dimensional, Gaussianized, Mel-Frequency Cepstral Coefficients (MFCCs). We use a frame period of 10 ms with an analysis window of 30 ms in the feature extraction. In the segmentation and clustering stage of speaker diarization, an initial segmentation is generated by uniformly partitioning the audio track into K segments of the same length. K is chosen to be much larger than the assumed number of speakers in the audio track. For meeting recordings of about 30 minute length, previous work [24] experimentally determined $K = 16$ to be a good value.

The procedure for diarization is takes the following steps (more details can be found in [1]):

1. **Initialize**
   Train a set of GMMs, one per initial segment, using the expectation-maximization (EM) algorithm.

2. **Re-segment**
   Re-segment the audio track using majority vote over the GMMs' likelihoods.

3. **Re-train**
   Retrain the GMMs on the new segmentation.

4. **Agglomerate**
   Select the most similar GMMs and merge them. At each iteration, the algorithm checks all possible pairs of GMMs, looking to obtain an improvement in BIC scores by merging the pair and re-training it on the pair's combined audio segments. The GMM clusters of the pair with the largest improvement in BIC scores are permanently merged. The algorithm then repeats from the re-segmentation step until there are no remaining pairs whose merging would lead to an improved BIC score.

The result of the algorithm consists of a segmentation of the audio track with n segment subsets and with one GMM for each subset, where n is assumed to be the number of speakers.

This system was proven to be highly effective, but the computational burden was such that the processing took about real-time.

Using PyCASP, the implementation of our diarization system is implemented in about 100 lines of Python code. For readability, we capture the application in 50 lines of Python code, shown in Figure 6 by omitting some function internals. The components that are executed on the parallel platform (either GPU or multi-core CPU) highlighted in light-gray.

Based on the algorithm description above, we now step through the Python code:

1. **Initialize**
   First we import the GMM specializer of PyCASP (line 1) and uniformly initialize a list of K GMMs (in our case 16 5-component GMMs) on line 5. After creating the list of GMMs, we perform initial training on equal subsets of feature vectors (lines 6-9). The training computation is executed on the parallel platform. Next, we implement the agglomerative clustering loop based on the Bayesian Information Criterion (BIC) score [33] (line 12-13).

2. **Re-segment**
   In each iteration of the agglomeration, we re-segment the feature vectors into subsets using majority vote segmentation (lines 16-18). We use the parallel platform to compute the log-likelihoods (`gmm.score()` method), which calls the E-step of the GMM training algorithm.

3. **Re-train**
   After re-segmentation we re-train the Gaussian Mixtures on the parallel platform on the corresponding subsets of frames (lines 22-23).

4. **Agglomerate**
   After re-training, we decide which GMMs to merge by first computing the unscented-transform based KL-divergence of all GMMs (line 31). We then compute the BIC score of the top k pairs of GMMs (in our case $k = 3$) by re-training merged GMMs on the GPU (lines 33-37) and keeping track of the highest BIC score. Finally we merge two GMMs with the highest BIC score (lines 43-45) and repeat the iteration until no more GMMs can be merged.

All data structure allocation and transfers are handled by PyCASP. The above implementation of speaker diarization written in Python achieves 50-250× *faster than real-time* performance by executing the computationally intensive components on an NVIDIA GTX480 GPU [19]

```
import numpy as np
from gmm import *

def diarize(self, M, D, K, data):

 gmm_list = new_gmm_list(M,D,K)
 per_cluster = N/K
 init = uniform_init(gmm_list, data, per_cluster, N)
 for gmm, data in init:
   gmm.train(data)

 # Perform hierarchical agglomeration
 best_BIC_score = 1.0
 while (best_BIC_score > 0 and len(gmm_list) > 1):

   # Resegment data based on likelihood scoring
   L = gmm_list[0].score(data)
   for gmm in gmm_list[1:]:
     L = np.column_stack((L, gmm.score(data) ))
   most_likely = L.argmax()
   split_data = split_obs_data_L(most_likely, data)

   for gmm, data in split_data:
     gmm.train(data)

   # Score all pairs of GMMs using BIC
   best_merged_gmm = None
   best_BIC_score = 0.0
   m_pair = None

   #find most likely merge candidates using KL
   gmm_pairs = get_top_K_GMMs(gmm_list, 3)

   for pair in gmm_pairs:
       gmm1, d1 = pair[0] #get gmm1 and its data
       gmm2, d2 = pair[1] # get gmm2 and its data
       new_gmm, score =
           compute_BIC(gmm1, gmm2, concat((d1, d2)))
       if score > best_BIC_score:
         best_merged_gmm = new_gmm
         m_pair = (gmm1, gmm2)
         best_BIC_score = score

   # Merge the winning candidate pair
   if best_BIC_score > 0.0:
     merge_gmms(gmm_list, m_pair[0], m_pair[1])
```

Figure 6: Speaker diarization in Python. Components that are executed on the parallel platform are highlighted in light-gray.