Understanding Patterns:
Conceptual Tools for Design Pattern Analysis

by

Donna Kaminskyj Long
B.Sc., University of Victoria, 2010

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Understanding Patterns:
Conceptual Tools for Design Pattern Analysis

by

Donna Kaminskyj Long
B.Sc., University of Victoria, 2010

Supervisory Committee

Dr. Yvonne Coady, Co-Supervisor
(Department of Computer Science)

Dr. R. Nigel Horspool, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Yvonne Coady, Co-Supervisor
(Department of Computer Science)

Dr. R. Nigel Horspool, Co-Supervisor
(Department of Computer Science)

ABSTRACT

This thesis presents two separate and complementary tools for understanding and analyzing design patterns. The first tool, the High-Level Pattern Representation (HiLPR), exposes the fundamental characteristics hidden within a design pattern's solution. This tool combines the information in parallel patterns' solutions and forces, and integrates information that is critical for pattern implementation. The second tool, the Dynamic Pattern Categorization (DPC), works between all of the patterns in an entire pattern language, and groups patterns of similar characteristics to support analysis and selection. Possible categories are presented and discussed, and further work can combine the exposure of characteristics from HiLPR into categorization by the DPC. The evaluation of these tools highlights a hidden weakness of current design pattern languages and practices. The conclusions raised by this work suggest that there are methods that will support pattern language construction.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Yvonne Coady and Nigel Horspool for their help and support, and for pushing me to discover my own abilities;

Celina Gibbs for her mentoring and collaboration;

Liam Kiemele for his collaboration and always having a full coffee pot;

my friends for being there, and for being willing to support my procrastination with board games;

my family for celebrating the highs and supporting the lows; and finally,

my husband, Jeremy for always knowing what to do and say, even though this battle didn't involve dice.

Chapter 1

Introduction

Design patterns are a widely accepted approach to describing general solutions to frequently occurring problems in software [18]. A single design pattern is intended to provide a blueprint solution to a single problem and an identification of the implementation tradeoffs that will be encountered. That is, the structure of a portion of the program is provided but the developer is still required to make implementation decisions in terms of the tradeoffs presented in the pattern, coupled with application and architecture specific requirements.

Using patterns allows both programmers and developers, those who work on a piece of the program and those who consider the program in its entirety, to communicate more effectively, as they provide a common language to discuss programming design problems. Patterns describe reality; they are ideas that have been useful in one practical context that can be generalized to others. They allow reasoning about problems at a more abstract level, to describe similarities across different problems, and to reason about how some solutions can work together to solve even more complicated problems. There are many examples of analysis [1, 39] on the original Object-Oriented patterns [15, 16], including: relationships between patterns [48], and composition of patterns [47]. Less attention has been applied to parallel pattern languages such as "Our Pattern Language" [37, 35] (OPL). The lack of this kind of in-depth analysis is not surprising given that the OPL language is currently incomplete, but is nevertheless problematic. Analysis of OPL patterns will help to gauge their validity.

This thesis presents two main contributions: the High-Level Pattern Representation (HiLPR) and the Dynamic Pattern Categorization (DPC). Both of these contributions work with the Berkeley Parallel Pattern Language as a proof of concept, but they are not tied to that particular language and may be applied to others.

These contributions focus on different aspects of pattern languages, but have a common theme: supporting pattern analysis and selection. Both analysis and selection are particularly important to the health of pattern languages. Pattern analysis allows researchers to find similarities and relationships between patterns, which can either lead to more patterns, or a better understanding of how patterns work together. Pattern selection, on the other hand, is crucial for users. All design patterns, and all of the analysis on design patterns, is useless unless they are being used by developers. They may be used to transfer knowledge about widely-found problems, or to help with a particular problems' implementation, but either way, they must be used.

Patterns contain a lot of information, and while this is in general quite useful, it means that appropriate pattern selection can be difficult. Both the Dynamic Pattern Categorization and the High-Level Pattern Representation make fundamental pattern characteristics more visible, facilitating more informed pattern selection with less of a time investment in patterns that are inappropriate for the current developers' needs.

A review of the related work that this thesis builds upon is presented in Chapter 2. Then Chapter 3 describes an experiment where I traced a problem's implementation against the corresponding parallel design pattern, Sparse Linear Algebra. This experiment led to a visual representation of a design pattern's solution, which then directly led into the High-Level Pattern Representation.

Chapter 4 presents the motivation for and methodology of the High-Level Pattern Representation (HiLPR). It is followed by three applications of HiLPR to parallel design patterns: Sparse Linear Algebra, Pipeline, and Shared Queue, and discusses general conclusions that can be drawn from the application process.

Finally, Chapter 5 explores the Dynamic Pattern Categorization, a framework for organizing pattern languages based on intrinsic pattern characteristics, such as those exposed by HiLPR, and the proof of concept implementation using the Berkeley Parallel Patterns.

Chapter 2

Related Work

This chapter outlines both the context of the work presented in this thesis, and the factors that have influenced the contributions of this work. This chapter has been broken into three sections: *Design Patterns*, to give the historical context of the original Object-Oriented patterns, designed by a group of researchers who have been nicknamed the "Gang of Four" (GoF); *Parallel Design Patterns*, identified in Berkeley's Our Pattern Language (OPL), which the contributions use as proof-of-concept applications; and *Design Pattern Analysis*, which overviews the kinds of analysis previously applied to both Object-Oriented and Parallel Design Patterns.

2.1 Design Patterns

Groups of patterns are often presented as a unified catalogue, grouped categorically, with each pattern individually identifying relationships to other patterns. For example, the Gang of Four (GOF) patterns are grouped into *Creational*, *Structural* and *Behavioural* patterns, with each pattern including a section entitled *Related Patterns*. This organization provides a browseable set of patterns written by the four authors working closely together to create a consistent and uniform format across all patterns to ease use and application of patterns in real world development.

Design patterns are a widely accepted approach to describing a general solution in the context of a frequently occurring problem in software [18]. The applicability of a given pattern in multiple settings has made design patterns one of the most widely accepted abstractions in the software development design phase. Groups of patterns are often presented as a unified catalogue that are grouped categorically, with each

pattern individually identifying relationships to other patterns. For example, the Gang of Four (GoF) patterns are comprised as catalogue of object-oriented patterns grouped into the three categories of *Creational*, *Structural* and *Behavioural* sections, with each pattern including a section entitled *Related Patterns* with the intent of supporting selection and composition of patterns.

The original GoF patterns were organized in two distinct manners: their *Purpose*, which was broken into *Creational*, *Structural*, and *Behavioral* patterns; and their *Scope*, defined by *Class* and *Object* patterns. This structure can be seen in Figure 2.1. Another way of viewing the GoF patterns was provided by Zimmer, who analyzed the patterns based on how they interacted with each other [48]. Specifically, he observed three main relationships: *X uses Y*, *X is similar to Y*, and *X can be combined with Y*. He used these relationships to reason about how patterns may be composed, and to consider the implications of using certain types of patterns together. These relationships, and their implications, directly motivated the second contribution of this work.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Figure 2.1: The Object-Oriented Design Patterns, displayed in their original Gang of Four categories [16]

2.2 Parallel Design Patterns

The Berkeley Parallel Computing Lab [32] provides an over view of recent efforts within the parallel community to develop a pattern language [26, 24, 25] specifically to address parallel programming issues. This pattern language, initially called the *Pattern Language for Parallel Programming (PLPP)* [37] and more recently called

Our Pattern Language (OPL) [35] is still in the development stage, and began with five categories of patterns: *structural*, *computational*, *algorithm*, *implementation*, and *concurrent execution*. The patterns that have been developed have adopted a standardized format that includes: *problem*, *context*, *forces*, *solution*, *related patterns*, *etc.*

Since then, there have been additional families of domain specific patterns. The family that we focus on for this paper is parallel patterns and other high-level parallel strategies, which include: the "Pattern Language for Parallel Programs" (PLPP) [37], also known as "Our Pattern Language" (OPL) [35]; Microsoft Research's parallel programming concepts [7], which expand parallel strategies as broad definitions; and GoF patterns again, which have been modifed to support parallel architectures by unlocking their intrinsic concurrency based on their modularity [42].

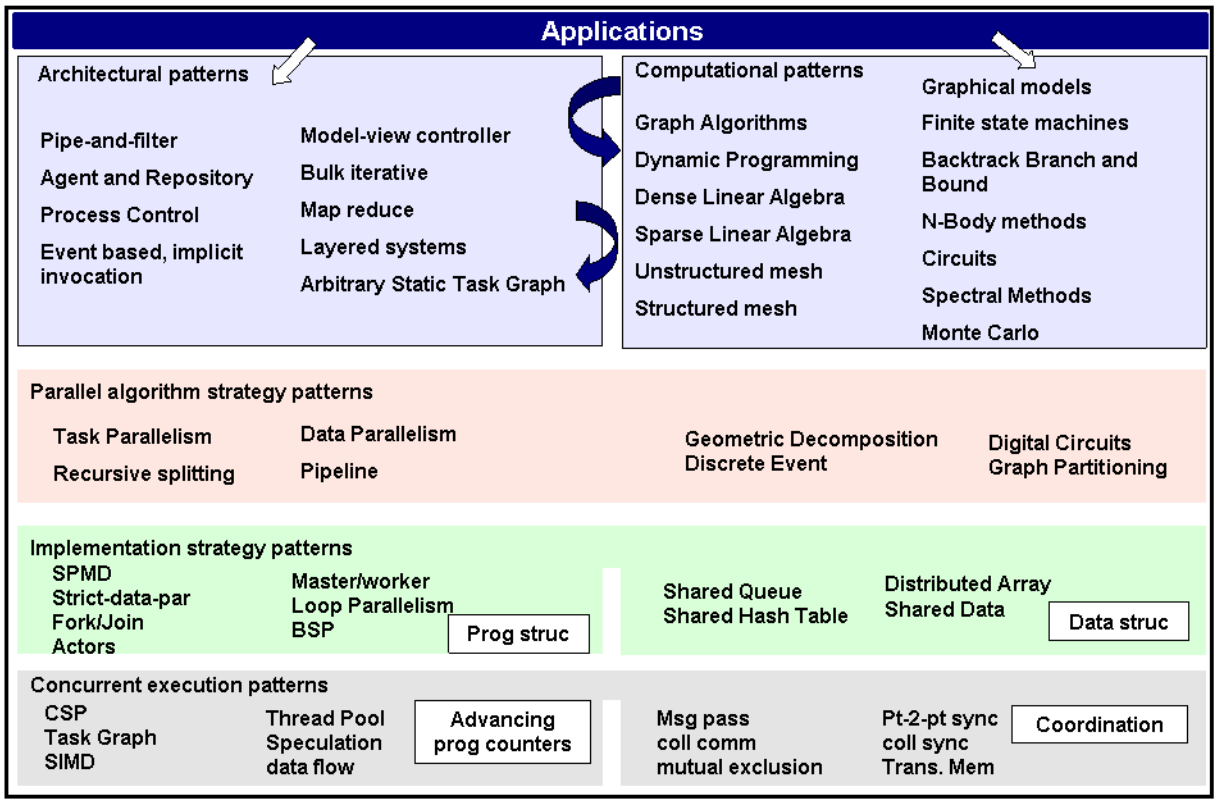

Figure 2.2: Berkeley's "Our Pattern Language" Categorization [35]. This figure shows the current OPL patterns by name, breaking them down into their categories: "Application Architectural", "Application Computational", "Parallel Algorithm Strategy", "Implementation Strategy", and "Concurrent Execution". We refer to this figure to show the relationship between the patterns that we have chosen to investigate.

There are stark differences in the organizational schemes used by the Gang of Four and the Berkeley's OPL. Where GoF's structure was explicitly divided along two difference axes, the OPL's structure is more implicit, described by detailed category names, as shown in Figure 2.2. Some of the OPL categories overlap, which is observed by different subheadings for select categories. Altogether, the OPL categories are: *Application Architectural*, *Application Computational*, *Parallel Algorithm Strategy*, *Implementation Strategy (Program Structure)*, *Implementation Strategy (Data Structure)*, *Concurrent Execution (Advancing Program Counters)*, and *Concurrent Execution (Coordination)* Patterns.

As previous described, at a basic level, each pattern has a: *problem*, *context*, *forces*, and *solution* section. The context serves to narrow the scope of the pattern and to draw the problem into the perspective of the solution. The forces identifies trade-offs in terms of the choices that a programmer must be aware of when implementing the solution. The forces section is closely tied to the solution, which provides a highly abstracted description of the implementation process. The solution does not provide code examples; its intent is to guide a programmer through the implementation decisions in terms of the tradeoffs that will be encountered as outlined in the forces section. A general synopsis of the SLA design pattern is provided in terms of this pattern outline.

2.3 Design Pattern Analysis

Pattern languages [2] also provide structure to lead a user through a collection of patterns. Though individual pattern languages have been successfully defined within smaller subdomains [9, 12], navigating a larger, disparate set of patterns written by less collaborative authors can be more challenging. The Berkeley Parallel Computing Lab [32] provides an overview of recent efforts within the parallel community to provide such a pattern language [26, 24, 25]. This pattern language, initially called *Our Pattern Language (OPL)*, began with a simple four-layered approach in which many of the individual design pattern write-ups are under development. The patterns developed so far have adopted a standardized format comprised of sections including: *problem*, *context*, *forces*, *solution*, *related patterns* and each pattern is assigned to one of the five categories: *structural*, *computational*, *algorithm*, *implementation*, and *concurrent execution*. While this format does provide an uniform outline across the patterns, the way in which each of the sections is written up can introduce variation

depending on the author and the research group they are involved with.

This structure is beneficial in terms of grouping patterns by purpose and generality to support pattern selection, but navigation of these growing collections and understanding how they apply to source code is still a challenge. Alternative classifications, intended to reduce the number of fundamental design patterns to consider, have been combined with more systematic and concrete class libraries or families of patterns to make patterns both more accessible and traceable to code [1]. Other strategies such as *Design Pattern Rationale Graphs* [5], reconcile design with source to aid developers to make changes that are in keeping with an existing design. A graphical representation of both source and design patterns are linked by edges through an intermediate level, representing relationships between the source and patterns. Navigation is bidirectional between source and design by way of queries.

Current efforts within the parallel pattern community are also focusing on *methodological* patterns [23, 37] to further guide users through this framework, capturing the fine-grain relationships both within and between these proposed layers. The newest version of Berkeley's *Pattern Language for Parallel Programming (PLPP)* addresses this issue with a much more fine-grained, control-flow type of structure [32]. The intent is to guide a developer through pattern selection at the various levels of design. Patterns are grouped by design decisions like *choosing a high level structure*, *identifying key computation patterns* and *choosing a concurrent approach*. In addition, this newer version of the pattern language acknowledges the lower-level issues of efficiency that must be dealt with by the programmer. This approach narrows the scope for pattern selection.

Our previous case study investigating pattern tradeoffs in the pervasive domain proposed RIPPL [17] (Relationship Initiated Pervasive Pattern Language), a systematic methodology for the comparison of design patterns. This approach, grounded in the isolation of pattern tradeoffs as outlined within the *forces* sections of each pattern, demonstrated the comparison of implementation decisions across design patterns. While this preliminary work of RIPPL focused on a uniform representation of the forces sections of a set of patterns, the information from the other sections of the design patterns relevant to implementation specific decisions was not incorporated.

Like many other software artifacts, once the primary modularity of a design is chosen it is difficult to modularize all the key concerns associated with that design. That is, no matter what the dominant decomposition of the application is, there will be core concerns that do not fall cleanly into that modularity. It is this scattered na-

ture that adds to the complexity associated with understanding these concerns within a software artifact. Multi-dimensional separation of concerns [46] proposed a formal approach to modeling and implementing software artifacts with the separation of overlapping concerns across multiple dimensions [41]. Aspect-oriented programming [30] initially provided an approach to explicitly and modularly represent *crosscutting concerns* with linguistic mechanisms [29]. Both of these approaches looked to address the issues of complexity associated with a lack of modularity within the different phases of the software lifecycle. Further research in aspect-oriented software development considered its application to other artifacts in the software lifecycle including requirements [11] and design [19].

Chapter 3

Pattern Implementation and Evaluation

This chapter provides a motivating example for why design pattern analysis is critical to the health of pattern languages. It describes an experiment which attempts to analyze the usefulness and usability of the parallel design pattern *Sparse Linear Algebra*. The analysis of the strengths and weaknesses of this pattern motivated the work later done on the High-Level Pattern Representation.

This chapter explores the implementation of a parallel design pattern, and compares both processes—the design, and the development—to determine how similar they are. To test the pattern, I explore the Thirty-Metre Telescope Adaptive Optics problem, which at its core becomes a large sparse linear algebra system. I trace through this problem from both the design pattern and an unguided implemention of the solution to highlight the similarities and differences.

Core principles of software engineering have provided a foundation for the development of accepted practices and methodologies for creating quality software that is both understandable and maintainable. While these practices are part of the core education of today's software engineers they are not mainstream to scientists in the biological, engineering and physical sciences. These scientists are faced with both large scale computation problems and copious amounts of data. While the scale continues to grow, the underlying hardware resources are no longer growing in terms of processor speeds but instead are growing in terms of the number of processing units. The complexity surrounding programming for multiple processing units has become one of the key challenges of computer science, amplifying the need for support and

structure imposed by software engineering practices and methodologies.

In general, big science projects have unique requirements to be addressed by computer science, for example: vast amounts of data to be processed and computationally intensive algorithms with strict time constraints. These requirements usually include a desire to make use of commodity hardware for economical purposes. Large scale calculations such as linear algebra computation is common in areas of research such as adaptive optics technology used to correct wavefront errors on astronomical data collected from telescopes, distortion in communication systems and retinal imaging [10]. Algorithms associated with this problem domain lend themselves to parallel implementations but the size of the problem combined with real-time constraints make it challenging for developers to experiment with and consider a software solution.

Our work focuses on linear algebra systems. These systems can be divided into two major categories: *sparse* and *dense* [13], and with regards to the former, we leverage the SLA pattern [36] of the OPL. Sparse matrices, as indicated by their name, have a significant number of zero values, unlike the dense form which is highly populated with non-zero values. Significant work has been done to develop algorithms which take advantage of the properties of sparse matrices [14, 21] as they have many applications to real-world problems. The SLA pattern applies to many different domains, ranging from solving systems of linear equations to image processing and looks to help developers improve their solution in terms of storage, cost and stability.

Data level parallelism, that is, distributing data across multiple compute units and the simultaneous execution of tasks on this data, can be achieved on a *Single Instruction, Multiple Data (SIMD)* hardware model. SIMD, first used in supercomputers, is now applied in personal computers and commodity devices and uses a *vector processor approach*. In a vector processor, the same computation is performed on a set of values in a vector or array style data structure that aligns with the underlying hardware as opposed to a *scalar* approach, where the processor performs computation on a single value at a time. *Vectorization* is supported by existing languages through the use of *intrinsics*, which provides functionality that is handled by the compiler. SIMD and vectorization have been proven to be a significant performance increase in other domains [22], and have been applied in new high-performance GPUs, such as Intel's Larrabee [45].

GPUs are a specialized microprocessor originally intended to accelerate multi-dimensional graphics for use in game consoles, personal computers and hand held devices. The highly-parallel and low-cost nature of these graphics cards are now

making this architecture desirable for programs requiring the execution of identical tasks on large data sets [20]. In fact, the cost effective nature of GPUs is making software a viable solution over otherwise application-specific hardware. An example of a computationally heavy algorithm involving substantial matrix multiplication is the Three Dimensional Symmetrical Condensed Node Transmission Line Matrix (3D-SCN TLM) used to calculate electromagnetic fields. Leveraging GPUs, computation of a 3D-SCN TLM method has shown 120 times speed-up over a commercially available solver [43]. This speed-up not only required an efficient revamping of the algorithm to ensure a favourable memory locality for parallelization, but also the application of multiple aggressive optimization strategies that required intimate knowledge of the underlying architecture.

The parallel domain is supported by a rapidly growing number of linguistic mechanisms, ranging from libraries to full languages with underlying compiler support. No one linguistic mechanism has been deemed the clear winner in this space. That is, different mechanisms having varying tradeoffs including usability, underlying control and correspondence to underlying architecture. The Apple initiative, OpenCL [3], is a domain specific framework that is currently being developed to support efficient programming of data and task parallelization across a pool of multiple processing units that can be a mix of CPUs and GPUs. This framework, being standardized by the Khronos group [27], looks to provide abstractions of underlying hardware specifics through language mechanisms. Developers must implement the basic unit of code called a *compute kernel* which can be grouped to take advantage of data parallelization or alternatively leverage task parallelism. NVIDIA's CUDA [40] framework provides a similar form of linguistic support but is architecture specific and limited to programming GPUs. In this preliminary work we focus on the use of OpenCL for parallelization support.

This chapter investigates ways in which to make software engineering practices useful and accessible to the scientific programmer looking to optimize an application. Specifically, we begin by looking at the use of design patterns as a template to guide developers through design and implementation decisions. With the recent development of parallel specific patterns, I use an existing version of the *Sparse Linear Algebra* (SLA) pattern to investigate the ability to map the abstraction provided in the current form of design patterns through to implementation.

Design patterns describe general programming problems, which makes them widely applicable, but the solutions that they describe are broad, and try to take every pos-

sible issue into account. This can make patterns difficult to use, even in cases, such as this one, where pattern selection is not difficult. Working through the solution and finding the best implementation, can still be quite challenging, as the solution is hidden in with many decisions which do not apply to every problem or system.

I developed a visual representation of the pattern solution which draws out the implementation questions, making the choices a pattern solution requires more explicit. This representation does not modify the solution, it expresses it differently, and guides a developer through the implementation of their chosen pattern.

I applied this representation to the sparse linear algebra design pattern, and compared my representation to the original implementation. I determined that, even in this reduced form, there is still a lot of inconsistency between how a pattern expresses the solution, and how developers work on a problem, since the best representation to the solution was a static decision tree, which is rigid in structure and still does not fully support developer needs. Although this case study only includes sparse linear algebra, the visual representation can be extended to other design patterns.

Through an in depth analysis of the pattern and SLA implementation we evaluate the applicability of the pattern, how its design choices can be explicitly represented and propose ways in which to refine the pattern to enhance its accessability by scientific programmers.

3.1 Motivation

The motivation for this chapter was to determine whether the Sparse Linear Algebra parallel design pattern accurately reflected the implementation practices of software development. This motivation is deeper, drawing on a lack of recent evaluation for and on design patterns. There was a lot of work done on the Object-Oriented patterns to examine their relationships with each other, as well as further analysis of their strengths and weaknesses. This is not the case for parallel patterns. The patterns are being written, and their overall structure has been redesigned a couple times, but the patterns themselves have not been put up to the same intense scrutiny that the Object-Oriented patterns were.

This concern led to the experimental setup described later in this chapter, which compares an unprimed implementation against the Sparse Linear Algebra design pattern.

3.2 Methodology

This section is broken into two parts: a description of the Sparse Linear Algebra parallel design pattern, and an overview of the pattern implementation. The synthesis of these parts concludes this chapter.

3.2.1 Parallel Design Pattern

This section describes the Sparse Linear Algebra design pattern. Sparse matrices are interesting as there are a high number of optimization options that may be implemented. The following is an overview of the parallel design pattern

Problem

The problem, as described in the SLA design pattern, examines large-scale linear operations on matrices that contain a high number of zero entries. The pattern explores optimizations which will handle storage and performance issues associated with this problem.

Context

The context examines the benefits of the characteristics of a matrix that contains mostly zero-values. This situation is described as a common occurrence in some fields, "arising from the symmetry of the system or due to the fact that different subcomponents of the system are independent of each other. When the fraction of zeroes is significantly large,...there are benefits to explicitly [taking] these zeroes into account when solving [these] problems." [36].

Forces

The implementation trade-offs as enumerated within the forces section include:

1. Storage versus Cost: Whether intermediate results are better to keep or recompute

2. Portability versus Specificity: Whether hardware-specific software or portability is more important

3. Requirements versus Performance: Whether the data layout should follow the needs of those using the system or be optimized for performance

Solution

The solution of the SLA design pattern is broken into four subsections following a general introduction. The introduction familiarizes the reader with pre-written library tools; for many linear algebra problems, these are sufficient. The only choice that the developer must make in the majority of cases is that of a direct or iterative solver. Direct solvers are slow and reliable, as a result of their straightforward brute force computation of the linear equation. Iterative solvers are faster than direct solvers, though unreliable, as their solution is bounded by an error term. Iterative solvers are dependent on the specific properties of the matrices involved in computation for their performance—well behaved matrices can be solved much faster than randomly-sparse matrices.

In the case where high-performance implementations are important, the pattern organizes possible optimizations into the following categories:

1. **High-Level Optimization Approach**
   This section breaks optimizations into three areas of focus: "memory bandwidth improvement", "data-structure size reduction", and "instruction-throughput improvement". It is suggested that memory-bound computation leads the developer to focus on improving the data structure of cache management, since all available bandwidth is already being used.

2. **Sparse Matrix Data Structures**
   This section discusses multiple options for data structure, including the *Compressed Sparse Row (CSR)* and *register blocking*, which has two variations: "Block Coordinate" (BCOO) and "Block Compressed Sparse Row" (BCSR).

3. **Parallelism in SpMV** (Sparse-Matrix, Dense-Vector Multiplication)
   This section highlights issues created from utilizing parallel architectures, including load balancing and communication overheads. Graph-partitioning algorithms are discussed to help manage these problems.

4. **Cache and TLB Blocking**
   This section considers optimizations which exploit reusable results as a side-effect of computation.

3.2.2 Implementation

A basic matrix vector multiplication implementation was developed to investigate data layout and execution strategies. This was done using the relatively new programming language OpenCL [28, 4]. As previously mentioned, OpenCL uses *compute kernels* which can be compiled at runtime to execute on a specific platform or computational device such as a CPU or a GPU. We developed a basic kernel, which can be seen in Figure 3.1, and then created an optimized kernel, as can be seen in Figure 3.2 to analyze both the decisions that went into development and the forces present in optimization. We ran the code on two separate devices. The first was a Intel i7 processor which is a quad core system with eight logical threads. This provided a baseline for a standard computing environment. The second device used was an NVIDIA Geforce 5600. This provides an interesting comparison: GPUs, as opposed to CPUs, are optimized for floating point arithmetic and high levels of parallelism and instruction throughput.

The matrix vector multiplication boils down to $Ax = b$ where A is the matrix, x is the vector and b is the resulting vector from the multiplication. We varied the size of A, x and b to see how the implementations would scale. Tests were done with x at 64 000, 640 000, and 2 640 000 floats, as can be seen in Table 3.1. A contained 12 times the floats in x for each test and b is the same size as x. This made for an extremely large and sparse matrix. These calculations require a significant amount of memory and are bounded by bandwidth as opposed to computation. Obtaining optimal performance depends on several factors involving memory use in terms of latency, bandwidth, access, alignment and cache size. It also depends on the level of parallelization possible, in terms of executing parallel threads and having high instruction throughput.

	Execution Time (ms)			
Floats in x	Basic CPU	Vectorized CPU	Basic GPU	Vectorized GPU
64000	936.01	295.93	6.03	1.77
640000	4450.69	2097.46	59.30	16.92
2640000	*	*	250.94	66.87

Table 3.1: Execution Times for Basic and Vectorized Implementations. The * indicates the test cases where the CPU was unable to execute, based on the large number of floating point values in the problem.

Since a sparse matrix contains mostly zeroes, we know that a large part of the

matrix is empty which makes an efficient representation that does not store these zero-values imperative. Ideally, the storage requirements are as small as possible, allowing for constant time access and aligning adjacent values to facilitate parallelization.

Next we consider the three key components in our implementation, data layout, parallel execution strategy, and optimizations.

Data Layout

The chosen representation was a *Compressed Sparse Row (CSR)* [6] matrix. This is a general strategy for layout which does not assume properties such as the matrix being diagonal or containing sets of dense regions. This also allows for a high level of parallelism. The CSR involves three arrays: one holds the column value for each element, the second is the value of the element and the third indexes the start of each row. The length of each row is implicitly defined and we can simply operate on one row until the start of the next. This data structure requires the equivalent space of list of lists; however, it is easier to work with.

This structure provides several benefits including alignment of data such that values are adjacent. When the hardware loads a value, it will also load the next set of values to be used, hiding memory latency and improving locality. Space is conserved because no zeroes are stored. This allows for a larger part of the array to be in cache. Each row can be operated on independently. This allows for easy parallelization and each executing thread can simply operate on a row in a straightforward manner. Each row in the matrix vector multiplication will only affect a single value in the result and therefore, as the computation is independent, we do not need to use locking mechanisms which may slow down execution.

Parallel Execution Strategies

There were two key considerations with respect to the execution of the algorithm:

1. resource utilization of the available processing elements
2. leveraging the capabilities of the hardware, including vectorization

To implementat the sparse matrix solver, an OpenCL compute kernel (Figure 3.1) was built to compute the dot product of each row of the matrix and the vector, in parallel. When computing on a large matrix, we will not be able to fit the entire

matrix in cache, this makes a high degree of parallelism important. If computation is stalled by having to retrieve a value from main memory we need to ensure that another computation is ready to execute. If working on a matrix that fits in cache, we will be able to use the parallelism to compute the result quickly. This provides a significant speedup and the execution is much faster than a linear or sequential implementation. OpenCL appears to closely match the hardware whether being executed on a CPU or GPU and takes advantage of the available executing threads.

```
__kernel void sparse2(__global int* cols,
__global float* vals, __global float* x,
__global float* b, __global int* index)
{
    int row = get_global_id(0);
    int start = index[row];
    int end = index[row+1];
    int i;

    b[row] = 0;
    for(i = start; i < end ; i++)
    {
        b[row] += vals[i]*x[cols[i]];
    }
}
```

Figure 3.1: OpenCL Sparse Matrix-Vector Multiplication Kernel

Vectorization can further provide a significant boost in performance, utilizing all of the computational resources of a single processing element. This works in parallel with the higher-level multi-core concurrency. In general, data is sent in bursts, the exact size of which depends on cache width, but we never receive a single piece of data from main memory, we always receive several. Vectorization allows us to explicitly load a set of data into *vector types* and then carry out mathematical operations on these vectors, providing more efficient memory access, as shown in Figure 3.2.

OpenCL will use available hardware to carry out vector operations as SIMD operations, greatly improving instruction throughput. In this case we used the `float4` datatype to improve performance. This can be expanded to further increase performance. By using the `float16` datatype we could perform 16 multiplication operations in a single instruction if the hardware supports it. On the other hand, if the hardware does not support such large SIMD operations, OpenCL will convert them to smaller

```
__kernel void vectorsparse2(__global int*cols,
__global float* vals, __global float* x,
__global float* b, __global int* index)
{
    int row = get_global_id(0);
    int start = index[row];
    int end = index[row+1];
    int i;
    float4 xcols, vvals, accum;
    accum.x = 0;
    accum.y = 0;
    accum.z = 0;
    accum.w = 0;

    b[row] = 0;
    for(i = start; i < end ; i += 4)
    {
          if(end - i >= 4)
          {
               xcols.x = x[cols[i]];
               xcols.y = x[cols[i+1]];
               xcols.w = x[cols[i+2]];
               xcols.z = x[cols[i+3]];
               vvals = vload4(0,&(vals[i]));
               accum += xcols*vvals;
          }
    }
    b[row] = accum.x+accum.y+accum.z+accum.w;
}
```

Figure 3.2: OpenCL Vectorized Kernel. This figure shows the additional coding complexity of vectorization as compared to Figure 3.1, but provides better performance as it greatly improves instruction throughput.

ones. In the worst case our instructions will be converted to sequential instructions and still execute across platforms.

In terms of the implementation, loops were unrolled by hand to allow several values to be worked on at once to take best advantage of OpenCL's optimizations, but this did not increase our instruction count or impact the elegance of our solution. It does help our system to take advantage of the available resources. Data can be loaded, computation performed, and data stored in parallel to make I/O less costly when handling large matrices.

Optimization Analysis

Here we consider the tradeoffs encountered in the optimization techniques employed in the implementation described above. There are four portions of the problem that can be further considered for optimization. They are described below:

1. **Cache Management Strategy**
   Making full use of cache is key to performance gains and in many cases matrices will not entirely fit. This motivates us to use the values in cache as much as possible before obtaining new values, and at the very least performing calculations while waiting for I/O. These targets can be achieved by ensuring a high level of parallelism and vectorization.

   Specifically, vectorization allows our implementation to load a set of contiguous values, do the work on those values and then store the data in memory. This ensures that the data loaded into cache is fully used and not simply occupying space. Parallel execution strategies, on the other hand, allow us to take advantage of what is currently in cache with parallel computation while we wait for the next set of values. Having many processing units allows us to keep execution running while waiting on I/O.

   Cache is limited, so issues storing intermediate values are possible, but were not encountered in this implementation. The fastest implementation had an additional twelve words, consisting of 32 bits, per thread of execution. This did not appear to cause a significant difference for any of our experiments for any array size. Having a slightly larger memory footprint in order to vectorize the code appeared to be extremely beneficial in terms of performance gains.

2. **Memory Access and Latency**

In terms of memory access there is one main optimization. Memory is accessed in contiguous blocks which our matrix representation allows. Latency associated with memory accesses can be introduced by data structures such as structs or objects which would have the relevant data interspersed. Having our data partitioned into three separate arrays allows us to load a set of values at the same time. The size of this set depends on hardware, but when loading one value, we also load the next several values used in the computation. This helps hide latency when accessing memory.

3. **Instruction Throughput**
   SIMD was used in our implementation to provide higher instruction throughput. OpenCL offers the programmer library mechanisms that perform SIMD operations. In the case of 16 element vectors, it is possible to do 16 multiplications with one instruction.

   The biggest benefit comes from loading, computing and storing values in these large chunks. Storing data contiguously allows for this to happen and is one of the reason we chose the Compressed Sparse Row (CSR) matrix representation. We load a significant portion of matrix in one operation. Unfortunately because of the compressed rows, we could not load the values in the vector x as efficiently. In the case of a diagonal matrix, SIMD could be leveraged even further to create an even more efficient implementation.

4. **Portability**
   OpenCL allows for the development of extremely portable code. It was originally developed on an Ubuntu 9.10 laptop using the ATI OpenCL implementation and executed on an quad-core CPU. The code was then recompiled in Visual Studio and ran on an NVIDIA GPU. This involved no changes to the core code. Using OpenCL allows us to maintain portability without sacrificing the other optimizations.

   The biggest gain is with the OpenCL vector types; we can program for a powerful machine which can take advantage of SIMD operations and vector hardware without losing the ability to execute the code on less powerful machines. In more advanced situations a program can obtain the system information and select or tune the compute kernel. OpenCL's ability to exploit memory locality is also extremely useful.

3.3 Application

This section describes the creation of the *Decisions Tree for Internal Pattern Implementation*. It synthesizes lessons learned from tracing through the pattern description and the implementation described in previously in this chapter.

The solution of the pattern describes a series of decisions that a programmer should consider when implementing a sparse matrix solver. However, we feel that the main branching points of this decision-tree are difficult to find in the text of the pattern, and that the actual implementation strategies are virtually camouflaged. After working through the solution, we propose explicitly creating this decision-tree to organize the information drawn from the textual representation of the optimizations and tradeoffs that span the *forces* and *solution* sections. This tree, depicted in Figure 3.3, serves to formalize the implementation decisions that a programmer must make. Our representation of the *solution* is intended to augment the exisiting pattern—it is not a sufficient tool on its own—but it provides programmers and scientists with accessible guidance through these implementation decisions.

Unless otherwise noted, we assume that movement through this structure flows from the *Sparse Matrix* root, in a downward direction along the edges. The initial decision point is based on *speed* versus *safety*. Following the safe path makes use of library implementations of *Direct Solvers*, four of which are shown here. A programmer requiring a more high-performance solution would follow the deeper path towards the *Iterative Solvers*, which provide subsequent optimization options.

There are three areas of focus discussed: "memory bandwidth improvement", "data-structure size reduction", and "instruction-throughput improvement". However, in our decision tree, we have only considered two of these three main branching points of optimization—the former, and the latter. Although "data-structure size reduction" is first introduced in the pattern in this section, it is considered to be a solution to the "memory bandwidth" problem, and not a main focus of further optimization. The *High-Level Optimization Approach* provides a simple test to determine which focus should be considered: the size of the matrix with regards to the size of cache. It is heavily suggested that only one of these "subtrees" is going to be important for achieving optimization in the code; we have modeled this by separating out the subtrees and by not expressing any sort of iterated development. So, as indicated by the pattern, at each node, the developer would pick a possible path to find their optimization. Upon reaching a leaf, this plan should be implemented. Reading the

decision-tree, one might assume that upon reaching such a leaf, that optimization is the *only* one that is appropriate. This directly mirrors a flaw in the pattern.

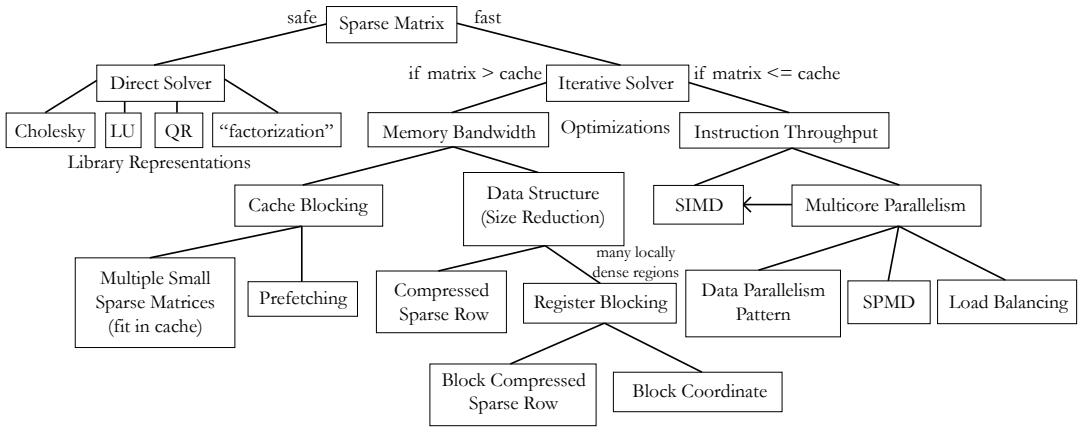

Figure 3.3: Sparse Linear Algebra Decision Tree [34]. This figure displays our previous attempt to reorganize the solution of Sparse Linear Algebra. It was a direct translation between the solution and a flowchart, and turned out far more complicated than we anticipated. This allowed us to consider that a direct translation was not useful, and provides a visual comparison for the structural additions our uniform representation makes (Figure 4.2).

Combining the Forces with the Solution

As the *context* of the pattern alludes: by reducing the storage to only the non-zero elements and taking advantage of the well-defined zero arithmetic in linear algebra problems, these sparse matrices become a hotbed for optimization.

The *forces* are not applicable to the needs of the developer. However, to reconcile the beneficial aspects that the abstracted form of the forces provide, we must tie them closer to the decisions we are suggesting that a developer make, so as to make explicit the deeper consequences of each decision.

Although subtle, the *solution* provided by the pattern takes each of these *forces* into account. Consider the decision-tree, and we will see each of the previously mentioned *forces*.

Storage versus Cost is realized at the "Iterative Solver" node. The decision leading towards "memory bandwidth" or "instruction throughput" is described in terms of storage. The more precious the cache space, the more we lean towards

"memory bandwidth" and the **cost** of recomputing values; while the smaller the matrix, the more we lean towards "instruction throughput", and the extra **storage** that this requires.

The force **Portability versus Specificity** is tied to the choice following from the "Memory Bandwidth" node. When the developer is able to consider "Cache Blocking", they are accessing the **specific** hardware architecture of their machine. If this is not available for **portability** requirements, the only option left is to modify the "Data Structure".

The choice of **Requirements versus Performance** is the first one a developer will make, rooted at the "Sparse Matrix" node. Where **performance** is key, a less rigid "Iterative Solver" may be used—on the other hand, stringent **requirement** will likely make that path difficult to optimize, which leads a developer towards a "Direct Solver" solution.

Now that we have refined the original SLA design pattern, we evaluate our refinement based on how well our decision-tree mirrors the process taken by the programmer. To do so, we trace the design log of the implementation, and compare it with our organizational analysis of the solution. Finally, we will explore the differences between both processes.

3.4 Discussion

This section provides an evaluation of the SLA pattern as it currently exists with respect to our implementation experiences. We found that some of the forces were either not helpful, or made somewhat irrelevant by our choice of tools. Primarily, *portability versus specificity* and *storage vs costs* were not defining factors in our implementation. We have considered possible reasons for this weakness and have determined that the most likely explanation is that the pattern expresses the *forces* in terms of absolutes, as in: "you can have either portability or specificity", whereas the implementation neatly captured both.

First we will examine the implementation with regards to the stated **forces** in the pattern, keeping in mind that our evaluation is coming from the point-of-view of our language, OpenCL, and its features. Then we will examine additional factors that were either not mentioned in the pattern or could have been expanded upon.

The pattern has three forces: *storage versus cost*, *portability versus specificity*, and *requirements versus performance*. In addition to the forces that we have previously

considered, we also look at other factors where the SLA pattern could have assisted the design process further: *matrix representation*, *implementation assumptions*, *implementing parallelism*, and *iterative development*. Finally, this section concludes with a discussion of the difference between the decision tree and the implementation by tracing the design log through the decision tree in Figure 3.5.

Storage vs Cost

The time it takes to recompute a value is insignificant compared to the time required to load one from memory, therefore, we found that a small number of intermediate values did not hinder execution—especially on large matrices. Where the implementation consists of operations on thousands of floats, having an additional twelve floats in memory to hold intermediate values is insignificant.

It is also worth noting that matrix-vector or matrix-matrix multiplication does not require a large of amount of intermediate values. $Ax = b$ is a relatively simple equation and the output of each dot product of Ax can immediately be stored in b. In this case storage becomes a non-issue. In the case of extremely large matrices, a small number of intermediate values will not influence performance, so either way, this tradeoff is non-existent.

Portability vs Specificity

We found that using OpenCL allowed us to—for the most part—avoid the trade off between portability and specificity.

First and foremost, OpenCL runs in parallel across each row of the data and takes advantage of data locality to ensure the cache is used efficiently and code executes as quickly as possible. This is a basic function of OpenCL, and will work on any system without changing the code.

Secondly, the largest speed-ups were attained through vectorization and SIMD instructions. Generally implementing these would involve using hardware specific *intrinsics*; therefore, this would provide increased performance at the cost of reduced portability. The code using intrinsics would have to be written for each platform. OpenCL, on the other hand, provides several vector types and functions which are implemented across all OpenCL platforms. More importantly, if the hardware does not support the vectorized instruction, OpenCL will convert them to supported instructions. The main benefit is that we can program assuming a machine which

supports vectorization and SIMD instructions, and OpenCL will have it match reality. In the best case we have increased performance, otherwise we are no worse off.

We can write vectorized code and have it run successfully on a CPU and GPU, the latter of which could significantly take advantage of the vector specific hardware. The code did not have to be altered for either implementation, since using a level of abstraction allowed us to produce efficient code that was highly portable.

Requirements vs Performance

The key to this trade-off is to recognize compromises that may be introduced to requirements when performance is taken into account. In particular, with scientific and engineering applications, precision and accuracy often dominate non-functional requirements—such as performance.

Matrix Representation

The representation of the matrix was a huge factor in design, this can change depending on the type of sparse matrix present. We found that the compressed sparse row representation worked best in our situation, but the other representations could have been useful had our matrix been different. We feel as though these choices should not only be mentioned in the pattern, as they greatly affect implementation, but that the pattern could have had a more detailed organization of the tradeoffs between the various representations. A good representation should require a minimal amount of space while providing optimal functionality. This should be done by keeping data values contiguous and have constant access times. The optimal form of a matrix's representation may be up to domain experts, but there are significant improvements that can be made beforehand. It is also worth noting that permutations can change a matrix significantly and allow the use of a more efficient representation.

Implementation Assumptions

When starting with a linear algebra project with a focus on performance and scalability, certain assumptions are made almost immediately. We want to use every optimization tool available, which means using everything that the hardware can give us. In the case of OpenCL, we can design the system to take advantage of various optimizations, even if the current hardware does not support them. We immediately

utilized multicore parallelism, and from that, assumed that the compute kernel would be load balanced, to achieve our performance goals.

Implementing Parallelism

Parallelizing a matrix depends heavily on our underlying system, but there are some general methods of parallelizing matrix-vector or matrix-matrix multiplication that could be considered in most cases. We can also take advantage of different types of parallelization at different levels. Ideally, we can have a thread execute accross each row of the matrix in parallel and we can use SIMD instructions to increase our instruction throughput. We can avoid race conditions by having each thread only writing to distinct sets of values. We can use a high level of parallelism to continue to do work while we wait on inevitable cache misses and we can use vectorization to ensure that the data we bring in from memory is used effectively.

Iterative Development

During implementation, one optimization generally affected the others. For instance the desire to vectorize the algorithm required that the matrix be represented with contiguous values. In the case where a developer would choose a tradeoff, they may be able to choose both equally or optimize for one and then go back and optimize for the other. As the choices affect each other, it is important for a developer to attempt to optimize in each direction possible in order to obtain optimal performance.

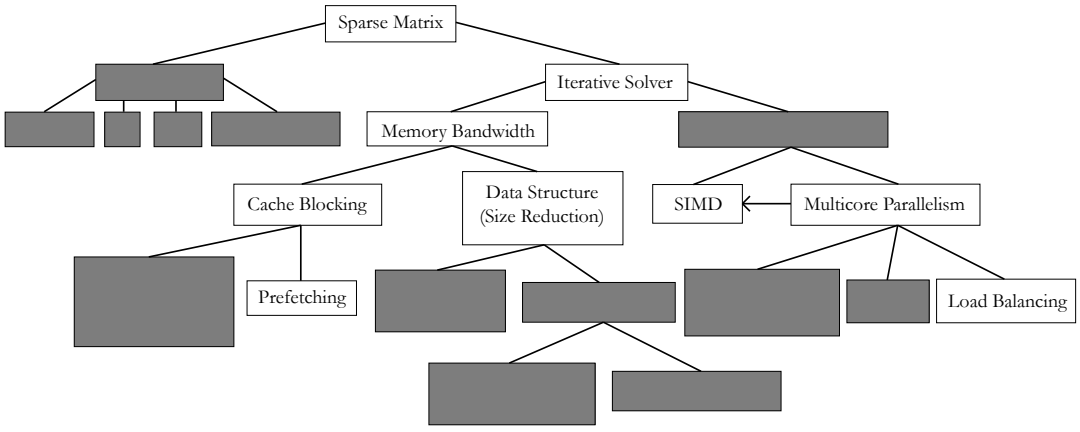

Figure 3.4: Reduced Decision Tree

Decision Tree Trace

Considering Figure 3.5, from our starting point (labeled *Start*), the grey arrows lead through the assumptions made by our implementation and the black arrows move sequentially through the design decisions that we made.

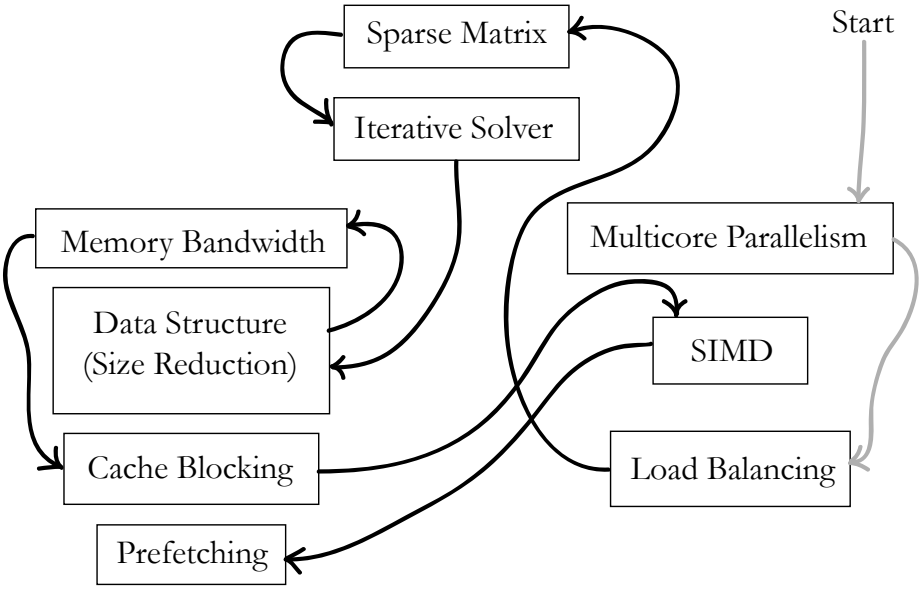

Figure 3.5: Flowchart Trace. This figure shows the trace of the implementation log through the visual representation of the decision tree. The optimizations have been grouped into two columns, which, in the decision tree, are supposed to be mutually exclusive. Furthermore, the stacking implies an order, where decisions on top are to be made first. The path starts grey, showing features that were provided by the OpenCL language. This trace shows that the ordering assumed by the pattern is not heeded or even necessary.

The two assumptions, "Multicore Parallelism" and "Load Balancing" are based on the choice of OpenCL as our language. OpenCL has features that neatly manage parallelism and load balancing, so although they are not a large component of the kernels described in Figures 3.1 and 3.2, we expect them to be present in the implementation.

From this point, we move to the sequential design decisions. The first and second, "Sparse Matrix" and "Iterative Solver" respectively, can nearly be considered assumptions, which is why we separated them from the two columns of optimizations. The reason that we did not connect them with our assumptions is that we went into this problem cold, and at the beginning, did not know whether we were going to need

to optimize for a sparse matrix, or not. From there, we chose the iterative solver to facilitate the experiments that we had described with Table 3.1, where we required a fast implementation.

After this point, the design log does not take the tree structure into account when choosing optimizations; in particular, note how the arrows move between the subtrees ("Cache" → "SIMD" → "Prefetching") and how they also move from child nodes to parent nodes ("Data Structure (Size Reduction)" → "Memory Bandwidth"). In section 5.1.1, we noted that not expressing iterative development was a failing towards the programmer; this is visually represented in Figure 3.5 by the number of choices that do not follow a singular path in the tree—namely, all of them. The pattern gave us this tree structure, but as we have shown here, our implementation—while following some decision points—does not strictly adhere to the structure of the tree.

Differences

One of the very interesting traces of the design log is that implementation is very "bottom-up"—that is, decisions are made from a low-level perspective. Nowhere is the most direct choice of the flowchart (whether matrix $>$ cache or matrix $\leq$ cache) considered, nor is the specific optimization examined before looking at what the pattern considers to be possibilities to help allieviate any roadblocks. Furthermore, the implementation picks multiple optimizations, from both sides of the "matrix $>$ cache" and "matrix $\leq$ cache" subtrees, making it very clear that those sorts of set decisions are not as clear-cut—although it is true that a large matrix will have memory bandwidth issues, for peak performance, instruction throughput *must* be considered as well.

Chapter 4

High-Level Pattern Representation

This chapter presents the first main contribution of this thesis: the High-Level Pattern Representation. It explores the remaining issues with the Chapter 3's visual representation. The previous pattern analysis led to a representation which provided a more explicit organizational structure to the pattern solution, but remained rigid in its structure, and failed to support developers. Furthermore, the visual representation loses important information from the *Forces* section of the pattern, as some of the more important decisions made in implementation of a parallel design pattern are described there, and the visual representation only includes information found in the *Solution* of the pattern.

Our preliminary investigation of the issues surrounding design pattern use, as applied to real world scientific applications, revealed that patterns do not necessarily reflect the actual design decisions that are being made by developers creating optimal solutions [34]. In this study, the pattern under investigation (*Sparse Lienar Algebra* [36]) did not naturally align with the sequence in which the developer had to make design decisions. To aid developers using the pattern, I proposed a refinement: including, as part of the *Solution*, a visual representation of its content which highlighted critical decision points (Figure 3.3). I believed that this proposed format made the decision points within a pattern more explicit and provided developers with a consolidated view of the implementation choices highlighted in the design pattern. While this preliminary study only considered a single pattern it provided a starting point for the consolidation of the implementation choices scattered across design pattern sections.

This chapter further identifies a problem facing the pattern community, one that manifests itself in many different forms: a lack of structural support which would

reveal critical relationships within and between patterns. There is a natural variation across pattern languages, with each language catering to the specific concerns of its discipline. These concerns are reflected in the structure of the pattern, where different languages may have vastly different structural designs. Pattern languages are not static. There will be future variation within languages, where structures require a *Solution*, but have no uniform description of what a solution entails. This sort of diversity, particularly in a domain with subtle interactions between software, hardware, and optimizations, can amplify complexity. It makes it difficult not only to use patterns, but to analyze them, work with them, and reason about them relative to each other.

Users of parallel patterns need to carefully consider many subtle aspects of software design. In particular, implicit relationships with hardware realities coupled with aggressive strategies for optimization are daunting in this domain. This chapter proposes a new way to leverage visual cues in the High-Level Pattern Representation (HiLPR), a proposed uniform representation for parallel patterns.

HiLPR provides internal structure to the pattern, like our previously proposed visual representation, but also reorganizes design decisions into broad categories that better match iterative implementation practices. These categories break down along software, hardware, and optimization decisions, and also include information found in the *Forces* section of the pattern.

4.1 Motivation

The problem posed in this chapter stems from a combination of two issues that make pattern use challenging to follow through to implementation. The first issue focuses on the internal structure of a pattern, and involves the way individual sections of a pattern are written. The second issue focuses on the external decomposition of a pattern, and pertains to the challenge of understanding how to use all of the the details which are split across the pattern sections: *Problem*, *Context*, *Forces* and *Solution*.

Internal: Lack of Uniformity

Patterns, in their definition, are a static representation of a solution, with each of the sections describing a specific issue related to the implementation. For example,

in Berkeley's *Our Pattern Language* (OPL), the *Context* provides a narrowing of the *Problem*, the *Forces* section is intended to identify the tradeoffs a developer will encounter whereas the *Solution* section provides a guide to the core implementation steps. While this is a logical decomposition of a pattern, there is an implicit relationship across these sections which is necessary to consider during implementation, and which also helps to develop an appreciation for the content and complexity of the solution. Specifically, the *Solution* section, by definition, is separate from explicit consideration of the tradeoffs presented in the *Forces* section as a developer moves through an implementation of the pattern.

Patterns need to be consistent, otherwise, the benefits of gathering the information are lost when a user must learn the idiosyncrasies of each writer. Since not all patterns are written by the same author, there may be uniformity in the section headings, but how those sections are written and organized may be very different. Some *Solution* sections are written with explicit steps to follow for an implementation while others are not. Some *Forces* sections are broken down into *universal* and *implementation* subsections while again, others are not. This issue can make pattern use challenging for a developer, as implementation information is scattered across the sections of a pattern.

External: Decomposition into Sections

The decomposition of pattern structure can make it challenging to use all of the information provided by the pattern. Patterns are presented in a way that lends themselves to be read in a linear fashion, section by section. This structuring can make using patterns difficult. To get the best information out of the current structure, a user would read the *Forces* and the *Solution* sections concurrently. In software engineering, the Waterfall method is taught as a starting point and leveraged to explain to students the benefits of an iterative method. However, the current structure does not capture what we believe to be a naturally iterative approach between related issues in different sections, or even within the *Solution* section itself.

4.2 Methodology

We propose HiLPR (High-Level Pattern Representation), a uniform representation for design patterns developed by tracing multiple implementation strategies used by

developers. The general structure that remained consistent through these strategies was found in multiple patterns, showing itself to be implicitly part of the solution. The simplicity of our structure is one of its main benefits. HiLPR builds upon what is already present in the pattern—it does not force a representation that does not belong. The uniform representation of HiLPR is a structural addition for the parallel design patterns—it should not be considered a parallel programming pattern itself, as it does not solve a *programming* problem. Our addition is based upon previous work that suggests a simplified *software-hardware-optimization* strategy [34].

HiLPR, the concrete application which addresses the problem presented in the Motivation, was determined by tracing through two separate implementation approaches to the problem: a design log which tracked the programmer's thoughts as the solution was implemented and problems were overcome [31], and a tutorial of the *Sparse Linear Algebra* problem using OpenCL [8]. Both discussions of the *Sparse Linear Algebra* problem have the same basic structure for managing iterative solutions: determining the software design, managing the hardware characteristics, and optimizing for performance. We have taken these three basic steps as a guide to how programmers implement this particular solution, and examined other parallel patterns to see whether the same basic structure holds.

Our initial research into *Sparse Linear Algebra* was grounded in the *implementation forces* of the pattern, tying each force to a decision point in a tree style representation of the pattern's solution. This result was our first consideration of consolidating the important decisions found both in the *Forces* and *Solution* sections of the pattern. This chapter extends that work by proposing a uniform structure to represent the information provided across the sections of a pattern in a localized and explicit form. Our structure is not a new addition to the parallel pattern language, nor is it a pattern itself. It is an organizational process that solves an *organizational* problem, and further ties the application of patterns into an agile application development lifecycle model.

With our new overall structure to parallel pattern solutions, we visually represent the process of solving a patterns' problem with a flowchart that contains pertinent information from all the sections of the pattern. We suggest a uniform structure that captures the three major stages of solving parallel problems: *Software Design*, *Hardware Characteristics*, and *Optimizations*—this structure, HiLPR, is shown in Figure 4.1.

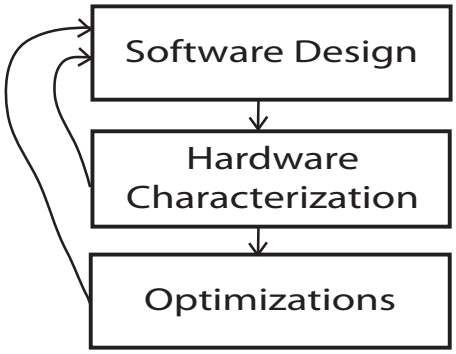

Figure 4.1: HiLPR, as an abstract uniform representation. This figure shows the abstract structure we suggest governs the solutions of the OPL parallel patterns, broken into three different stages. The arrows suggest the relationships and transitions between the stages for software development purposes.

Software Design

Software Design is the first stage of problem-solving for parallel patterns. The decisions that fall into this stage are primarily those of design and organization. This is the stage where a plan is crafted, one which considers the software constraints and design requirements of the problem. It is difficult to fully assess hardware characteristics and optimizations without having an intermediate design to evaluate against. This structure is designed to guard against premature optimization, which can take a great deal of time and effort before being shown to be completely separate from the problem being solved.

Both the *Hardware Characteristics* and *Optimizations* stages can lead back to the *Software Design* stage, as difficulties that are encountered at those stages can require modifications to the original design. Furthermore, any changes to a program's structure should also be reflected in the design to help ensure consistency across all the stages of software development.

Hardware Characteristics

Hardware Characteristics is the second stage of problem-solving, prompting developers to consider the underlying hardware upon which the solution will be implemented. It is a crucial stage for high-performance computing, as good designs that do not mesh well with the hardware structure can lead to inferior performance compared to a less polished design that does. It is likely that the design process will move through both

this stage and the *Software Design* stage multiple times, becoming more refined with each iteration. This is consistent with other software design methodologies such as the iterative design model [44], in constrast to the current sequential process seemingly espoused by the patterns.

Optimizations

The final stage is *Optimizations*. Typically this stage will include different ways of managing hardware to draw out peak performance. These can include universal optimizations, such as cache management, which can be generalized among multiple patterns, or implementation-specific optimizations that are localized to the current problem. These considerations are part of the last stage of development as they depend most on choices made in the previous stages.

In the case where the optimization changes the structure of the solution, such as multiple queues for the *Shared Queue* example (Section 4.3.3), or that a necessary optimization for performance is impossible, such as if SIMD isn't available in the *Sparse Linear Algebra* example (Section 4.3.1), then we suggest returning to the *Software Design* stage to incorporate this information into the design. We do not suggest simply returning to the *Hardware Characteristics* in these cases; major changes to the structure of the solution should be reflected in all stages of the design process.

4.3 Application

The following sections apply HiLPR to three different patterns: *Sparse Linear Algebra* [36] (Section 4.3.1), *Pipeline* [33] (Section 4.3.2), and *Shared Queue* [38] (Section 4.3.3). With these examples, we show that developers who focus on either the *Forces* or *Solution* separately gain an incomplete picture of the pattern which can only be remedied by taking them together. This property requires a way to combine the information in the *Forces* and *Solution* sections without losing the semantics of their differences. Our uniform representation provides that structure, highlighting the relationship between the information in each section.

In this section, we colour these images to visually represent where the data is coming from. Text in black is HiLPR's structure, blue is from the *Solution*, green from the *Forces*, and red shows our structuring additions to the pattern. Notice how the pieces from the *Solution* and *Forces* are organized—intertwined—in these diagrams,

Pattern	**Sparse Linear Algebra**	**Pipeline**	**Shared Queue**
OPL Category	Application Computational	Parallel Algorithm	Implementation Strategy
Software Design	1. Data Structure	1. Define Stages 2. Structure Computation	1. Define ADT
Hardware Characteristics	2. Multicore Parallelism 3. Memory Bandwidth	3. Represent Dataflow	2. Concurrency Protocol
Optimizations	4. Vectorization 5. Cache Management	4. Handle Errors 5. Processor Allocation	3. Shared Queues

Table 4.1: Overview of Case Studies. This table provides an overview of the results of this section. It shows, for each pattern discussed, the OPL category the pattern comes from. It also displays each stage of our abstract representation, showing how each step of the pattern solution breaks down between the stages. Notice how the Software, Hardware, and Optimization issues differ between each pattern, and the similarities between decisions in the same stage.

even though they are kept completely separate in the pattern. We have chosen to express this information using colour, as it gives the best visual (and maybe even visceral) description of the problem. After each individual case study, we provide a table breaking down the information expressed in the flowchart to describe its origin—*Forces* or *Solution*.

The following sections describe three applications of the High-Level Pattern Representation to parallel design patterns. The patterns are *Sparse Linear Algebra* in Section 4.3.1, *Pipeline* in Section 4.3.2, and *Shared Queue* in Section 4.3.3. Each pattern was chosen to be from a different section of Berkeley's pattern categorization.

Table 4.1 provides a reference for the following Case Studies. Each row in this table contains the pattern name, and provides the original categorization of that pattern in the OPL, as well as the numbered steps of the solution that we have assigned to each stage of our uniform representation.

4.3.1 Sparse Linear Algebra

The proposed visual representation of the *Sparse Linear Algebra* Design Pattern is shown in Figure 4.2. The external structure was determined by HiLPR, with the internal structure of the solution guided by our previous work on this pattern [34]. This representation, unlike those following, separates out the forces as "themes" for each of the uniform stages instead of explicitly making them part of the decision process. This is due to the weaker forces of *Sparse Linear Algebra* pattern, which are not explicitly tied to implementation decisions.

Software Design

The first step in solving a *Sparse Linear Algebra* problem is to decide upon the data structure that will be used. This step is crucial to this problem, as all future hardware decisions and optimizations are based directly upon the representation of the data.

1. **Data Structure**

    - This step describes a software decision with great impact on future hardware choices.
    - The complexity of the solution is rooted in this step, and is directly tied to the Requirements versus Performance theme, as described in the *Forces*.
    - **Requirements** come in two general types: the nature of the sparse matrix, and the constraints of the other program components.
    - **Performance** is the major goal of this stage. The data structure has a strong impact on the possible optimizations which may be applied.

Hardware Characteristics

The next two steps of the solution fall into the *Hardware Characteristics* stage of the uniform representation. The problems that they represent are interrelated, even though they are presented sequentially in Figure 4.2 and Table 4.2.

2. **Multicore Parallelism**

    - This step describe the choices that must be made to balance performance gains from considering multicore options.

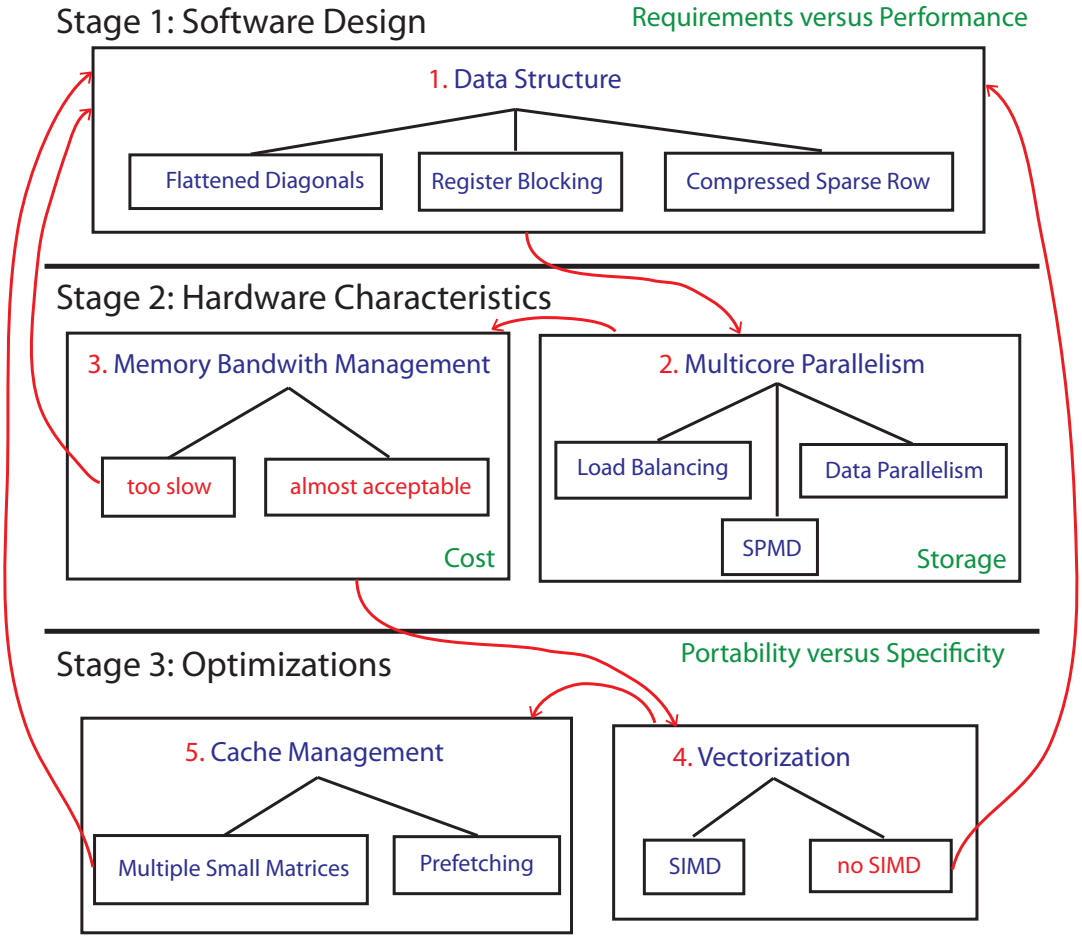

Figure 4.2: Sparse Linear Algebra. This figure displays how the Sparse Linear Algebra parallel design pattern's solution breaks down into HiLPR's structure. Each stage is populated with steps from the pattern's solution, and each step contains specific decision points that are suggested by the pattern. Our additions are in red. They show the movement between the steps and the stages, as well as some 'intrinsic' decisions that the pattern suggests but does not make explicit.

- The *Force* which impacts this step is **Storage**, balanced against the **Cost** required in the next step.
- **Storage** requires users to consider the tradeoffs of memory storage requirements in main memory and the cache. This step suggests computing intermediate values within a processing element to reduce the memory bandwidth requirements.

3. **Memory Bandwidth Management**

   - This step manages the tradeoffs created from the last step, namely, the communication overhead of moving to parallel architectures.
   - The *Force* which impact this step is **Cost** of communication, balancing the computational speed of adding additional processing elements.
   - **Cost** suggests reducing redundant calculation and storage requirements by doing shared computation in one element and communicating intermediate results to the other.

Since the mathematical processes that underly *Sparse Linear Algebra* are well defined, it is difficult to speed up a program considerably by changing the algorithm. Therefore, if the program is still running too slowly, the uniform representation guides the programmer back to the *Software Design* Stage.

Optimizations

In the final stage, *Optimization*, we consider the theme of "Portability versus Specificity", the second force listed in the pattern.

4. **Vectorization**

   - This problem is strongly tied to computational concerns, making Single-Instruction Multiple-Data computation a crucial optimization for performance. The ability to do simultaneous computation on multiple sets of data is at the heart of high-performance parallelism.
   - Should this optimization not be possible, it is highly recommended to reexamine the implementation, starting with the data structure, to be able to access it.

5. **Cache Management**

    - This step suggests managing data in such a way to take every advantage of memory and cache.
    - There are multiple types of cache management which may apply: changing the structure of the data so that certain decompositions fit fully in the cache, and managing the computation and communication such that they can be better overlapped.

The interaction of the Optimization steps with the *Forces* are the same for both Vectorization and Cache.

- The maximum speed of the solution is dependent on fully utilizing hardware, and is directly tied to the Specificity versus Portability theme, as described in the *Forces*.
- **Specificity** ties a program to a particular set of platform design, as SIMD instructions and cache considerations must, which may harm **Portability** requirements innate to the program and its interaction with other components in a larger system.

Summary

Finally, we provide a summary of the *Sparse Linear Algebra* case study, breaking the information from our image down into Table 4.2. This table lists each step of the solution of the pattern, dividing the information between the forces and solution sections. We use double horizontal lines in the table to partition the stages of our representation. Note that the *Forces* for *Sparse Linear Algebra* are not in the same order as discussed in the pattern. By tying them to the decision points where they are relevant, we order them chronologically with regard to the overall solution.

4.3.2 Pipeline

The visual representation of the *Pipeline* parallel design pattern, shown in Figure 4.3, highlights interesting differences between the organization of its solution compared to *Sparse Linear Algebra*. *Sparse Linear Algebra* is easily organized into HiLPR at a high level, where the specific steps that make up the pattern are harder to find in its solution [34].

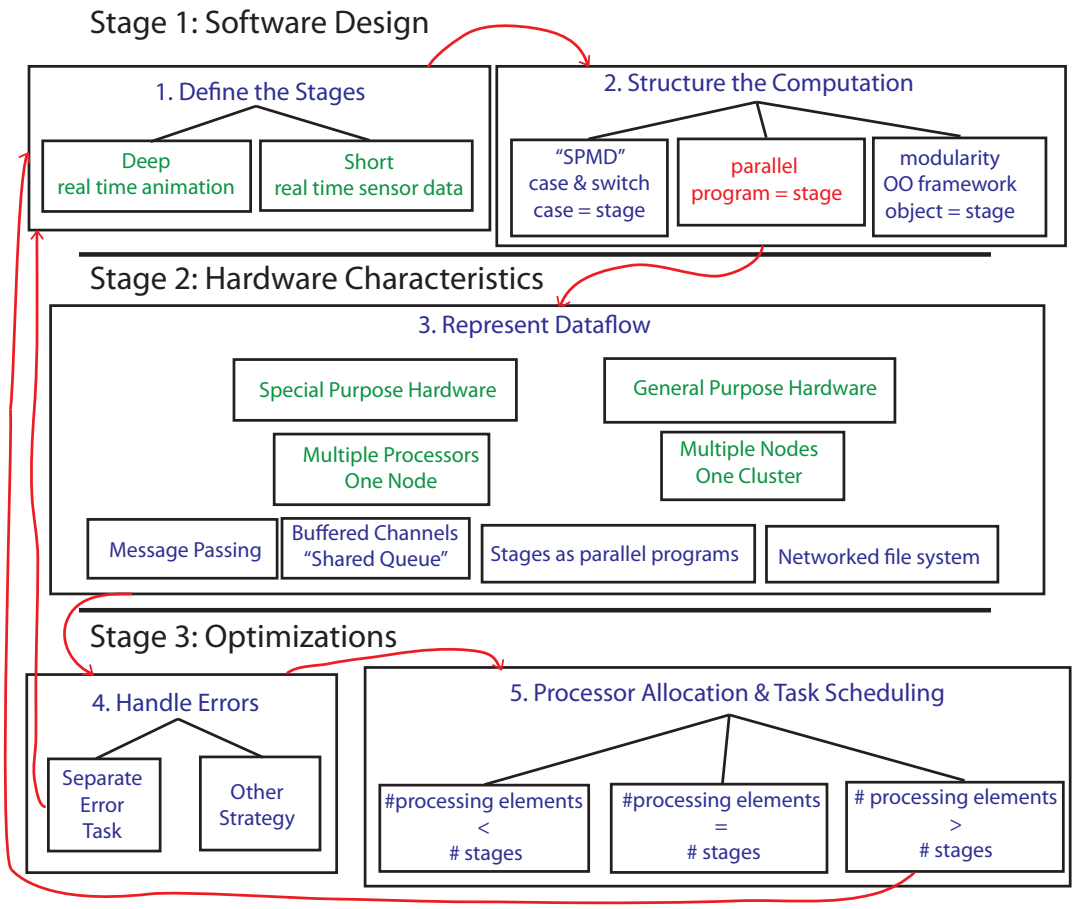

Figure 4.3: Pipeline. This figure displays the Pipeline parallel design pattern's solution, divided between HiLPR's stages. Each step within a stage, like in Sparse Linear Algebra, contains different choices suggested by the pattern. Step 3, "Represent Dataflow", has a larger number of choices than any of the other steps, and therefore is shown differently, with each horizontal line representing a choice even though there are no connecting lines between them.

	Solution	Forces
Software	1. Data Structure	Requirements v. Performance
Hardware	2. Multicore Parallelism	Storage
	3. Memory Bandwidth	Cost
Optimization	4. Vectorization	Portability v. Specificity
	5. Cache Management	

Table 4.2: Summary of Sparse Linear Algebra by HiLPR steps and Pattern Forces

Pipeline already has an internal organization in its solution. These steps conform to the stages of the visual representation: the first two, "Define the Stages" and "Structure the Computation" are software questions that fit into the *Software Design* Stage; the next, "Represent Dataflow" is a hardware question that fits into the *Hardware Characteristics* Stage; and the final two steps, "Handle Errors" and "Processor Allocation & Task Scheduling" are *Optimizations* that, while not easily applied to other patterns, as they specifically discuss the pipeline stuctures and the organization, place them into the final stage.

Software Design

The *Software Design* stage is dominated by one decision: whether the *Pipeline* should, in general, have few or many stages.

1. **Define the Stages**

   - This stage requires to user to decide the length of the pipeline.
   - This step describes a section of the solution, determining the length of the pipeline, which requires the pattern's *Universal Force* to fully understand.
   - A **Deep** pipeline optimizes for throughput, as used for real time animation, while a **Short** pipeline reduces latency, as used for real time sensor data.

2. **Structure the Computation**

   - This step provides two main choices from the *Solution* of the pattern: SPMD (Single Program, Multiple Data) stages, or a modular approach with object-oriented frameworks.

- The pattern later offers a third choice we have reorganized into this step: allowing each stage to be a parallel program.
- This step does not consider any of the pattern's *Forces*.

Hardware Characteristics

The second stage of the representation, *Hardware Characteristics*, contains two of the steps of the *Pipeline* solution.

3. **Represent Dataflow**

    - This step is closely tied to the *Forces* section of the pattern, revealing the hardware requirements of each decision.
    - The decisions within this step are sequential:
        1. Hardware Selection: Implementing the pattern on special-purpose hardware provides more options for the remaining choices.
        2. Processors and Nodes: The system architecture determines which optimizations are possible.
        3. Communication: Dependent on the previous choices, data communication may be very different.

Optimizations

Neither of the Optimizations in this stage are tied to the pattern forces, revealing how specific they are to the hardware considerations. Each of the Optimizations can modify previously made choices, requiring the user to consider how decisions in the Software and Hardware stage may have been changed.

4. **Handle Errors**

    - Error handling becomes more complex when the pipeline is spread across multiple programs or nodes.
    - Should the complexity require a separate task, consideration of how the task interacts with the rest of the pipeline is required.

5. **Processor Allocation & Task Scheduling**

   - This step describes how stages may be divided between processing elements.
   - There are three cases, ordered by complexity:
     1. Complex: Fewer processing elements than stages
     2. Simple: Equal numbers of processing elements as stages
     3. Ideal: More processing elements than stages, which suggests redefining the pipeline to take advantage of more options for concurrency.

Summary

Finally, we provide a summary of the *Pipeline* case study, breaking the information from our image down into Table 4.3. This table lists each step of the solution of the pattern, dividing the information between the forces and solution sections. We use double horizontal lines in the table to partition the stages of our representation. Although each step does not have a corresponding force, those that do are incomplete without them. Unlike *Sparse Linear Algebra*, where the forces outlined the "themes" of each stage, the *Pipeline* forces contain information crucial to implementing the solution.

	Solution	**Forces**
Software	1. Define Stages	Deep or Short
	2. Structure Computation	
Hardware	3. Managing Dataflow	Special or General Hardware Multiple Processors or Nodes
Optimization	4. Handle Errors	
	5. Processor and Task Allocation	

Table 4.3: Summary of Pipeline by HiLPR steps and Pattern Forces

4.3.3 Shared Queue

HiLPR's representation of *Shared Queue* is shown in Figure 4.4. *Shared Queue*, like *Pipeline*, has a solution section that is broken up into three different steps, each of

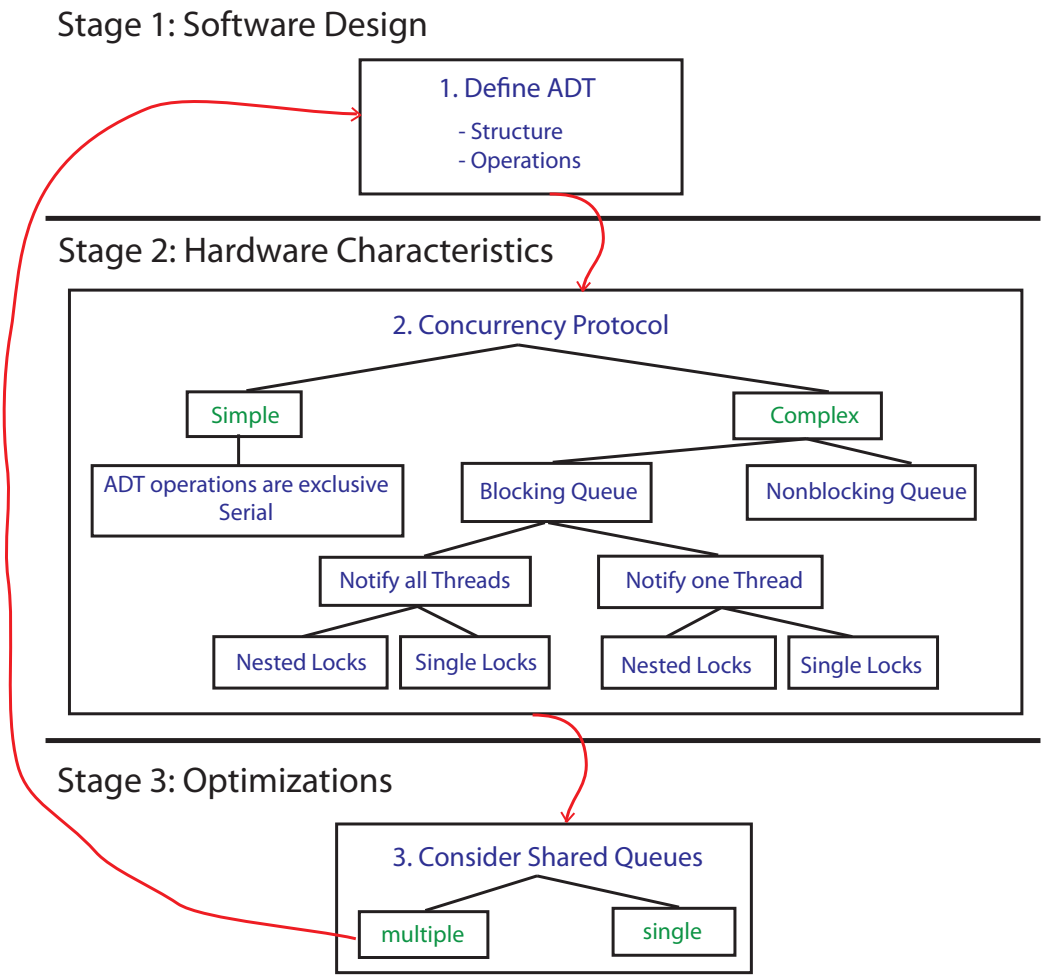

Figure 4.4: Shared Queue. This figure displays the parallel design pattern Shared Queue as it fits into HiLPR. Shared Queue is tightly connected to its forces, each of which change the internal structure of the steps. A complex queue is much more complicated than a simple one, shown by the number of additional decisions that must be made. A Shared Queue that uses multiple queues also begins the process again—which may appear redundant, but is actually necessary so that the modification for multiple queues have a baseline of one queue to be compared against.

which correspond to one of our stages.

Software Design

The first stage, *Software Design* focuses on defining the operations which are necessary for the queue to function.

1. **Define ADT**

    - This step requires users to define the structure and operations that will be used with the queue.
    - This stage does not interact with any of the pattern *Forces*.

Hardware Characteristics

This stage outlines the protocols which will manage the parallel structure of the queue's solution.

2. **Concurrency Protocol**

    - This step is controlled by the Simple versus Complex choice outlined in the *Forces*.
    - A **Simple** queue has an easy solution: allow ADT operations to be sequential and serial, while being less error prone. This makes the rest of the pattern unnecessary for the user.
    - A **Complex** queue allows the developer to user greater fine-tuning of queue optimizations. However, it requires a greater understanding of the problem, the underlying system, and requires considering the following steps:
        1. Blocking versus Non-Blocking: Blocking queues are more complex, as they require additional thread synchronization. If a Non-Blocking queue is possible, the remaining steps may be skipped.
        2. Notification: In considering synchronization, this choice considers whether all threads need to be notified when a lock is released, or only those threads which are waiting.
        3. Lock Structure: In either case, the user will need to decide whether they need nested locks or if single locks are sufficient to manage queue access.

Optimizations

The only optimization that the pattern considers greatly increases the complexity of the solution. Without an intial single-queue implementation, it is impossible to tell if the additional complexity adds any benefit.

3. **Consider Shared Queues**

    - Multiple queues may mitigate performance bottlenecks.
    - The problem and hardware may not support multiple queues.
    - If they are chosen, consider how this impacts the initial design of the system.

Summary

Finally, we provide a summary of the *Shared Queue* case study, breaking the information from our image down into Table 4.4. This table lists each step of the solution of the pattern, dividing the information between the forces and solution sections. We use double horizontal lines in the table to partition the stages of our representation. Notice the second step of the solution—should the "simple" side of the force be followed, much of the complexity disappears.

	Solution	**Forces**
Software	1. Define ADT	Deep versus Short
Hardware	2. Concurrency Protocol	Simple versus Complex
Optimization	3. Shared Queues	Single versus Multiple

Table 4.4: Summary of Shared Queue by HiLPR steps and Pattern Forces

4.4 Discussion

This section discusses the application and benefits of the High-Level Pattern Representation. It focuses on the scalability required and given by application, as on a small scale issues such as the composition of patterns are trivial, but any sort of growth—like an increase in the number of patterns—and these issues become exceedingly difficult to manage. We evaluate scalability issues for both parallel patterns

and HiLPR. We discuss the challenges of applying the visual representation, both to current patterns and those not yet written. We then further elaborate on the benefits that the representation provides to future patterns and to the interaction between pattern languages.

Each section of the Discussion considers the scalability challenges of a different aspect of HiLPR and design patterns.

...applying HiLPR to the OPL

Applying the representation to patterns that are already written is easy—requiring less work than a knowledgeable programmer would need to write a pattern in the first place. Our process requires that we go through each pattern, read and understand the content and concept, then apply this organizational structure to it. This is not unreasonable. Consider: patterns writing requires that a skilled and knowledgable programmer go through each problem, understand and write the content and concept, and apply the pattern *Problem*, *Context*, *Forces*, *Solution* structure to it. The similarities between the processes can be leveraged to not only make the application of the representation easy, but to help write new patterns.

...that HiLPR gives to new patterns

The process of the uniform representation focuses on implementing a *Solution* based on the *Forces* and *Context* of the pattern. While a pattern is being written, HiLPR's structure can be generated alongside as part of the writing process, forming the visual representation with little additional work.

HiLPR provides another benefit for authors of new patterns. Patterns contain a lot of information, and while writing them, it can be difficult to know where exactly to start. Guided by the visual structure, writers can leverage the structure as a common starting point.

...that HiLPR gives to the OPL

As the language of parallel patterns grows, it becomes more difficult to reason about concepts such as composition, as the number of possible combinations grows exponentially with each new pattern. HiLPR gives us a new vocabulary to help compare two patterns, in the explicit description of each step and each stage. The visual

representation includes clues that will be crucial as more patterns are added. Additionally, comparing multiple patterns becomes easier, since the salient structure of the uniform representation is the same across patterns and the internal steps that a pattern follows are abstracted to be a guide for implementation, without all of the details.

...that we see between pattern languages

We can consider the differences that HiLPR could find between pattern languages. For example, although the OPL patterns need to consider the underlying hardware as a crucial step for their parallel computation, we would not find the same result with the Gang of Four's Object-Oriented patterns [16]. However, the other stages of the representation (*Software Design* and *Optimizations*) apply. This allows us to consider the structural differences between different pattern languages. Other pattern languages may require additional stages to fully explain their processes, which would allow comparisons between the structure of those languages and the OPL.

Chapter 5

Dynamic Pattern Categorization

The second contribution in this thesis is the Dynamic Pattern Categorization, which is a methodology for categorizing design patterns. This methodology groups patterns based on innate characteristics. I first discuss the motivation that lead to the creation of this methodology, and then describe the DPC itself.

This chapter explores the original Object-Oriented design patterns, to determine characteristics that may have contributed to their success. A characteristic of the pattern language which has so far been overlooked is its organizational structure. The structure of the Gang of Four (GoF) patterns has two elegant properties: first, the characteristics that make up the categorization are broad, which allows the patterns to be neatly partitioned along their *Purpose* and their *Scope*; and second, these categories are independent of one another.

This idea can change how we think about, and use, design patterns. By taking what works well in practice as a guideline, we increase our chance of success with our organization scheme. And we are not alone in being concerned about the state of design patterns: Ralph Johnson, one of the members of GoF, has stated that "[w]e are getting a lot more patterns than we did with design patterns and that's one of the things I worry—that's going to be overwhelming to people." [23]. There were 23 GoF patterns. One of the parallel pattern languages is already approaching 50 patterns—and that's just *one* of the parallel pattern languages.

I propose a methodology in this chapter titled the Dynamic Pattern Categorization (DPC). I use what I have learned from the Gang of Four categorizations to describe this methodology so that it may be applied to many different pattern languages. Based on my understanding of the Gang of Four pattern organization, the Dynamic Pattern Categorization groups patterns based on their innate characteristics. These

groupings are *dynamic*, in that they are only applied at the needs of the user. This makes my methodology readily extensible by individuals who may have different needs from the pattern language.

To prove the benefits of this methodology, I propose three different groupings for the OPL parallel patterns. I describe how these groupings were anchored each group in different sections of the pattern, so each group has unique characteristics and remains independent of one another. I also show how these groupings may be used together to partition the language in multiple ways.

To evaluate this methodology, I compare the entire set of three groups against the current OPL categorization. I show how each categorization scheme highlights different aspects of the patterns.

5.1 Motivation

Design patterns are useful tools. Unfortunately, they can be difficult to use properly, especially for novices. This issue is exacerbated by the fact that the concepts contained in patterns are both abstract and complex, as such, selecting the appropriate pattern for a problem is difficult and requires human time. This is not a task that a computer is capable of handling for us. However, we propose that there is a general structure that can be applied to all patterns to help mitigate the task of selecting appropriate patterns. For this, we have returned to the original work by a group of researchers known as the Gang of Four (GoF). GoF's OO patterns are still used today [42], so we believe that there was some aspect of their structure that was successful. We determine what we believe may be the key properties of their success, and apply them to another category of patterns as a test.

There are many reasons why the system GoF has created is successful. Understanding them allows us to find the parts that are applicable to more recent patterns. Consider Figure 2.1. *Purpose* and *Scope* are basic categories that are also quite elegant – their meaning is clear, yet they are also broad enough to be effective. Broad categories are important: a categorization scheme falls apart if too many items that it categorizes have no place. We show what GoF has accomplished in Figure 5.1. The two axes that GoF has chosen neatly define the space of possible OO patterns. *Purpose* and *Scope* are inclusive, by virtue of being so broad.

This simple categorization system has kept people returning to GoF's patterns, even as a strategy to handle newer parallel architectures [42]. Another strategy to

simplify parallel programming is the creation of broad design categories, which are not written in the typical pattern format [7]. We use the OPL as an concrete example of how the lessons learned from GoF can be applied to another family of patterns. Our method of characterization is not tied to the OPL patterns, instead, they are a proof of concept that GoF's lessons are easily extendable. We do not claim to be working with the full set of OPL patterns, merely those with enough information to actually be categorized on each of our new axes.

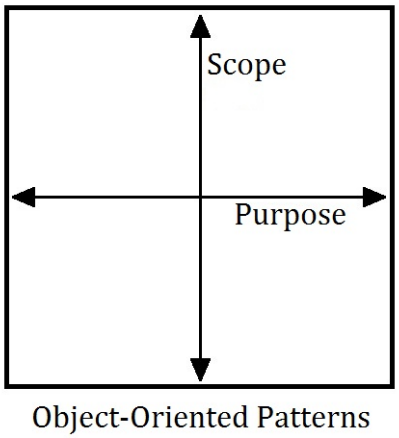

Figure 5.1: A revisualization of the Gang of Four's categorization scheme. This image shows how each axis of the categorization fully describes all possible patterns, while remaining independent from each other.

5.2 Methodology

I propose the Dynamic Pattern Categorization (DPC), a method to organize patterns into independent *groupings* which span the entire space of possible patterns in a pattern language. A grouping is simply a set of categories into which the patterns in a language can be organized. Groupings have three main properties, listed below:

1. **Dynamic**
   Groupings may be applied at the user's discretion. The addition or removal of a grouping should not effect the other groupings which have been applied. This allows users to only discriminate on pattern properties important to them.

2. **Complete**
   A single grouping completely describes the space of possible patterns. All pat-

terns should fit into a grouping category, and not be left out. Too many exceptions means that the categories are not properly described, and creates confusion when applying the grouping.

3. **Independent**
   A pattern's position in one grouping does not affect its position in another. This property is important to maintain the first property, and to allow many different users to create and share groupings without harming the strength of the method.

To use the Dynamic Pattern Categorization, a user would select the groupings that they are most interested in, and see how the patterns are organized in those particular categories. To illustrate, we suggest three different groupings for Berkeley's OPL in the following chapter, as a proof-of-concept implementation of the DPC.

5.3 Application

I have identified three major focal points of information in current OPL patterns. These points correspond with the sections *Problem/Context*, *Forces*, and *Solution*. We use these sections as a guide to form our categories by considering the tension at each focal point. The category descriptors that we have chosen to fit these sections are *Data-Task*, *Implementation-Execution*, and *Synchronous-Asynchronous Behavior*.

Each of our categories is designed to have any possible parallel pattern fit naturally in one quadrant. Our actual breakdown of patterns, in a visualization chosen to closely match GoF, can be seen together in Tables 5.1 and 5.2. The columns indicate whether a pattern is Data or Task, the rows Implementation or Execution, while membership in either table is determined by Synchronous or Asynchronous Behavior.

Problem & Context

The tension found in the *Problem/Context* sections of the pattern mirrors the two broad types of problems that can occur in parallel systems—a large space of data, or a complicated set of tasks. These are broad problems, and are not mutually exclusive. However, since each pattern focuses on a specific problem, reading these related sections makes this category clear. Consider the first line of the *Branch and Bound* pattern: "We have an extremely large space which we need to search...".

	Data	**Task**
Implementation	Branch and Bound Dense Linear Algebra Graphical Models Monte Carlo Methods Sparse Linear Algebra Loop Parallelism	
Execution	Geometric Decomposition Parallel Sorting Structured Grids Unstructured Grids	Map Reduce Recursive Splitting (Task Queue)

Table 5.1: DPC: Synchronous Behavior

	Data	**Task**
Implementation	Pipeline Shared Queue	Iterative Refinement Task Parallelism Discrete Event Speculation Task-Queue Implementation Task Graph
Execution	Agent and Repository Dynamic Programming Finite State Machine Graph Algorithms Graph Partitioning N-Body Methods Pipe-and-Filter	Event-Based Implicit Invocation Model-View-Controller Real Time Process Control Recursive Splitting (Fork-Join)

Table 5.2: DPC: Asynchronous Behavior

Compare to the first line of the *Recursive Splitting* pattern: "Consider the problem where an algorithm can be expressed as the composition of a series of tasks...". With just one example from each quadrant, the Data-Task axis has become well-defined.

Forces

The tension found in the *Forces* section of the pattern balances between benefits realized at compile-time, and those realized at run-time. In a large system where a pattern is only a small part of the whole, this knowledge can help to reason about how those parts work in concert. Consider a force found in the Execution pattern *Geometric Decomposition*—"Load-Balance". This is a clear run-time issue. Compare to the Implementation pattern *Discrete Event* which discusses the trade-off: "Ordering constraints implied by the data dependencies can be expressed by encoding them into the program...". Considering "Load Balance" against "Hardcoded Dependencies" illuminates the stark difference between Implementation and Execution patterns.

Solution

The tension found in the *Solution* section of the pattern is directly influenced by the implementation and subsequent execution of the program. This category makes it easy to remove a certain class undesirable patterns from consideration. For example, a distributed system is not going to work efficiently if it is designed with a central authority. We do not want to *select* the Asynchronous category in this case, but we do wish to *exclude* the Synchronous category, since they are likely to include behavior that our system cannot handle. Consider the Synchronous *Map Reduce* pattern, that requires a manager task to handle many interchangeable workers, and compare to the Asynchronous *Model-View-Controller* pattern, whose solution begins: "Divide the application into three interacting subsystems...", where each has a distinct duty. The difference between this one-to-many relationship in the tasks and multiple one-to-one relationships underline the ways each solution manages its behavior.

5.4 Discussion

This section describes two methods of evaluation for the Dynamic Pattern Categorization, and concludes with a discussion of the possible combination of the DPC with

HiLPR, the other tool presented in this thesis. The first method of evaluation compares the proof-of-concept implementations against the stated properties required for correct groupings, while the second method of evaluation compares how the parallel design patterns break down between both the proposed groupings and the current OPL categories.

Evaluation by Grouping Properties

I compare my proposed groupings against the properties which were described in Chapter 5, to show how these groupings maintain the necessary properties to be included in a DPC. In order, a grouping must be:

1. **Dynamic**
   This property applies to the Dynamic Pattern Categorization, so each individual grouping intrinsically maintains it. None of these groupings are required to be applied.

2. **Complete**
   Each grouping fully describes the space of the parallel pattern language, as it was in August of 2010.

3. **Independent**
   These groupings were chosen to be related to different sections of the parallel patterns: *Problem & Context*, *Forces*, and *Solution*. This fact supports their independence, for each pattern section describes a different aspect of the problem that it solves.

The independence of these groupings is further supported by the empty quadrant *Synchronous Task Implementation*. If these three groupings were intended to be a static representation of the parallel pattern language, I would expect each quadrant to contain patterns. Since the groupings are independent, it is possible that a subset of them will appear to have holes, as this one does. The holes are meaningful—in this particular set of properties, no current pattern falls into this category. This highlights two different avenues of exploration: are there patterns which belong that have not yet been discovered, or perhaps there is something about the categories which preclude a pattern from being a member in this particular set. The one thing that we cannot say about the hole is that any of the groupings that created it are weak or invalid. Each grouping, by itself, splits relatively evenly into its describing characteristics.

Evaluation by Berkeley's OPL

The Dynamic Pattern Categorization is designed to highlight different properties of the parallel patterns, properties that are not readily exposed by the current OPL categorizations. To show this, after creating the three independent groupings described above, I mapped the results of each grouping against the OPL categories. The results are shown in Table 5.3.

	Application Structural	Application Computational	Parallel Algorithm	Implementation Strategy	Concurrent Execution	Total
Data	7	7	2	3	0	19
Task	5	0	5	1	1	12
Implementation	6	0	4	3	1	14
Execution	6	7	3	1	0	17
Synchronous	6	3	2	1	0	12
Asynchronous	6	4	5	3	1	19
Total	12	7	7	4	1	31

Table 5.3: This table shows the breakdown of the patterns in each grouping against the categories used by Berkeley's OPL. It shows that, in general, the information presented by the groupings is different than the OPL categories, as the patterns break down relatively evenly between each side of a single grouping.

As can be seen in Table 5.3, most of the groupings are showing different information than the OPL categories, as the patterns break down relatively evenly into the groupings.

There are two main exceptions that we can determine, both *Data-Task* and *Implementation-Execution* in the Application Computational category. Out of the seven Application Computational patterns, all of them fell into *Data* and *Execution*.

The other issue is in Concurrent Execution. When I was first organizing the patterns into this grouping, there was only one complete pattern from this category. This makes it incredibly difficult to have an even distribution into both halves of a grouping.

I did not evaluate the set of three groupings against the OPL categories—because each grouping, as fitting with the DPC properties, is independent. It would be interesting to break down multiple grouping applications to see where the patterns fit, but it does not directly evaluate the individual groupings themselves.

Combination with HiLPR

The final point of dicussion for the Dynamic Pattern Categorization is the combination of this categorization method with the information exposed by the High-Level Pattern Representation. There is great potential for the two contributions presented in this thesis to work together. The first, HiLPR, exposes internal pattern characteristics to make pattern implementation and analysis easier. The second, the DPC, uses pattern characteristics to categorize, supporting analysis and selection.

These contributions are complementary. One exposes information, the other can use that information to group patterns on characteristics that can then be applied at the user's discretion, making the information collected by HiLPR more accessible to developers. A clear candidate for a possible grouping directly based on HiLPR's abstract structure is *Software-Hardware*. This grouping would express where the bulk of the pattern decisions were being made: in either the software or the hardware stage. The information exposed in this grouping would allow developers to determine whether the pattern was more abstract, with choices being made in the software stage, or tied tightly to the implementation platform, with choices being tied to the hardware stage. For example, Sparse Linear Algebra and Shared Queue would be hardware patterns, while Pipeline would be a software pattern, since the behaviour of the Pipeline changes more based on implementation decisions, not the hardware which supports it.

This combination of HiLPR information with DPC structure further supports design pattern analysis, and can lead to more information about pattern relationships.

Chapter 6

Conclusion

The two contributions presented in this thesis were the High-Level Pattern Representation and the Dynamic Pattern Categorization. Together, they provide new ways to work with and analyze parallel design patterns. The Dynamic Pattern Categorization works between the patterns in a language to group those that share similar characteristics, while the High-Level Pattern Representation works within a particular pattern to further expose those characteristics and highlight crucial implementation decisions. The internal representation supports the external categorization—the DPC can use HiLPR to determine which characteristics may be focal points for groupings.

HiLPR can be used to help evaluate patterns, since it highlights the main internal structure of the solution. This is not enough, though, since HiLPR itself is not well-defined inside the stages. Comparisons with HiLPR can happen, but are difficult as there is not a uniform internal structure, which would be necessary to further support comparison between patterns and design pattern analysis. Analysis is critical. While more patterns are being written, the evaluation and analysis of the patterns written or the few tools that are being created are not keeping pace. HiLPR is a first step, and can be a framework for considering inter-pattern and pattern-pattern analysis. But the next step should be a dialogue for what is important in evaluation, and a concerted effort to actually developing a methodology for pattern evaluation.

I can not tell you how good these tools I have created (the DPC and HiLPR) are, compared to other tools, since there is no way to compare them. I have developed ad hoc methods for myself to support what I think, but they are indeed *ad hoc* and are not easily extensible to other pattern comparison and analysis tools. This lack of evaluation support is the greatest weakness facing the pattern community, and is the most important direction HiLPR should push researchers in the future.

Bibliography

[1] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 134–143. ACM Press, 1998.

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[3] Apple. Opencl. `http://www.apple.com/server/macosx/snowleopard/`, 2008.

[4] Apple. OpenCL. http://www.apple.com/server/macosx/snowleopard/, 2008.

[5] Elisa L.A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design pattern rationale graphs: Linking design to source. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 352–362, 2003.

[6] Susan Blackford. Compressed row storage, 2000. http://web.eecs.utk.edu/ dongarra/etemplates/node373.html.

[7] Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. O'Reilly, 2010.

[8] Brian Catanzaro. Opencl optimization case study: Diagonal sparse matrix vector multiplication, 2010. `http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study.aspx`.

[9] Nicholas Chen, Rajesh Kumar Karmani, Amin Shali, Bor-Yiing Su, and Ralph Johnson. Collective communication patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[10] Max Claire. Introduction to adaptive optics and its history. American Astronomical Society 197th Meeting, 2001.

[11] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[12] Dwight Deugo, Michael Weiss, and Elizabeth Kendall. Reusable patterns for agent coordination. In *Omicini, A., Coordination of Internet Agents*, pages 347–368. Springer, 2001.

[13] Jack J. Dongarra and Victor Eijkhout. Numerical linear algebra algorithms and software. *Journal of Computational and Applied Mathematics*, 123(1-2):489 – 514, 2000.

[14] Salvatore Filippone and Michele Colajanni. Psblas: a library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Softw.*, 26:527–550, December 2000.

[15] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP' 93 – Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin / Heidelberg, 1993.

[16] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[17] Celina Gibbs and Yvonne Coady. Joining forces: A rippl effect? a constraint-oriented perspective on a pervasive pattern language. *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World*, 0:214–219, 2009.

[18] Hillside Group. Home of the design patterns library and host of the plop conferences. `http://hillside.net/`, 2010.

[19] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37:161–173, November 2002.

[20] Mark Harris. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[21] Leslie Hogben. *Handbook of Linear Algebra*. (Discrete Mathematics and Its Applications). Chapman & Hall/CRC, 1st edition, November 2006.

[22] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Ralph Johnson. Using a pattern language to design a system. http://www.cincomsmalltalk.com/userblogs/ralph, July 2009.

[24] Ralph Johnson, Kurt Keutzer, and Tim Mattson. Mechanisms that separate algorithms from implementations for parallel patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2009.

[25] Ralph Johnson, Kurt Keutzer, and Tim Mattson. Workshop on parallel programming patterns (paraplop), March 2010.

[26] Kurt Keutzer and Tim Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal: Addressing the Challenges of Terascale Computing*, 13(4), 2010.

[27] Khronos. Opencl. http://www.khronos.org/opencl, 2011.

[28] Khronos. OpenCL. http://www.khronos.org/opencl/, 2011.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer-Verlag, 2001.

[30] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. SpringerVerlag, 1997.

[31] Liam Kiemele. Design logbook: Implementation of the sparse linear algebra problem, 2011. `http://www.liamkiemele.com/dbook/DesignLogbook.html`.

[32] Berkeley Parallel Computing Lab. A pattern language for parallel programming ver1.0. `http://parlab.eecs.berkeley.edu/wiki/patterns/pattern1\_0`, 2010.

[33] Yunsup Lee. Pipeline pattern, 2009. `http://parlab.eecs.berkeley.edu/wiki/_media/patterns/pipeline-v1.pdf`.

[34] Donna Kaminskyj Long, Liam Kiemele, Celina Gibbs, Andrew Brownsword, and Yvonne Coady. Mind the gap!: Bridging the dichotomy of design and implementation. In *Proceeding of the 4th international workshop on Software engineering for computational science and engineering*, SECSE '11, pages 46–55, New York, NY, USA, 2011. ACM.

[35] Tim Mattson and Kurt Keutzer. Our pattern language (opl). In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2009.

[36] Tim Mattson and Mark Murphy. Sparse linear algebra pattern, 2009. `http://parlab.eecs.berkeley.edu/wiki/patterns/sparse_linear_algebra`.

[37] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[38] Timothy Mattson and Youngmin Yi. Shared queue pattern, 2008. `http://parlab.eecs.berkeley.edu/wiki/_media/patterns/sharedqueue.pdf`.

[39] James Noble, Robert Biddie, and Ewan Tempero. Metaphor and metonymy in object-oriented design patterns. In *Proceedings of Australian Computer Science Conference (ACSC). Australian Computer Society*, pages 187–195, 2002.

[40] NVIDIA. Cuda zone. `www.nvidia.com/object/cuda\_home.html`, 2010.

[41] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44:43–50, October 2001.

[42] Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 790–805. ACM, October 2010.

[43] Filippo V. Rossi. Graphics hardware accelerated transmission line matrix procedures. Master's thesis, Department of Electrical and Computer Engineering, University of Victoria, August 2010.

[44] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[45] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, August 2008.

[46] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. *Proceedings of International Conference on Software Engineering (ICSE)*, 0:107, 1999.

[47] John Vlissides. Composite design patterns (they aren't what you think), June 1998. `http://www.research.ibm.com/designpatterns/pubs/ph-jun98.pdf`.

[48] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1994.