HiLPR:

Pretty Pictures for Pretty Complicated (Parallel) Patterns

Donna Kaminskyj Long
University of Victoria
dkj@cs.uvic.ca

Celina Gibbs
University of Victoria
celinag@cs.uvic.ca

Nigel Horspool
University of Victoria
nigelh@cs.uvic.ca

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

ABSTRACT

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 paper proposes a new way to leverage visual cues in HiLPR, a proposed uniform representation for parallel patterns. We show the application of this approach to three design patterns: *Sparse Linear Algebra*, *Pipeline*, and *Shared Queue*. An evaluation of the combination of a pattern's *Forces* with its *Solution* within this representation indicates that this approach holds promise in terms of assisting developers in making better-informed decisions about pattern implementation.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Design

Keywords

Design Patterns, Parallel Design Patterns, Visual Representations

1. INTRODUCTION

Design patterns are a beneficial addition to our software engineering repertoire. Patterns allows us to communicate more effectively, as they provide us with 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 us to reason 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, 30] on the original Object-Oriented patterns [9, 10], including: relationships between patterns [36], and composition of patterns [35]. Less attention has been applied to parallel pattern languages such as "Our Pattern Language" [27, 25] (OPL). The lack of this kind of in-depth analysis is not surprising when we consider how new the language is, but is nevertheless problematic. Analysis of OPL patterns will help to gauge their validity, and is the main focus of this work.

This paper 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.

We propose HiLPR (High-Level Pattern Representation), a uniform representation for design patterns, which was created by tracing multiple implementation strategies used by developers. These strategies suggested a *software-hardware-optimization* strategy that we first developed through previous work [19]. This simple, consistent structure to each development strategy, one which we determine to be implicitly part of the implementation process. This simple structure allows HiLPR to build upon what is already present in the pattern—it does not force a representation where it does not belong. The uniform representation of HiLPR should not be considered a parallel programming pattern itself—it is a structural addition for the parallel design patterns.

After an overview of related work (Section 2), this paper discusses the problems associated with the lack of struc-

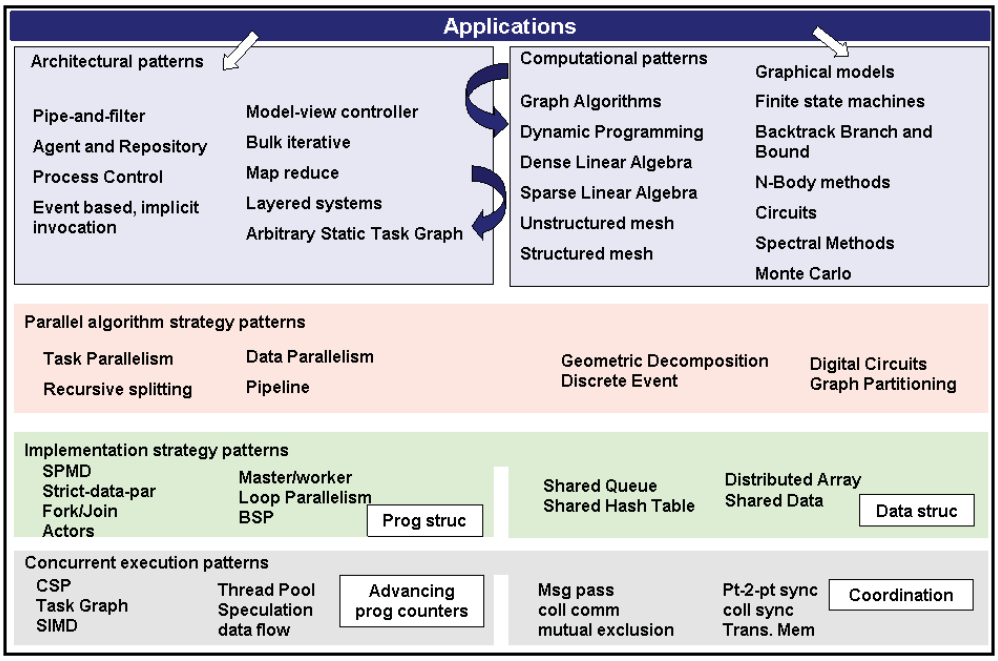

Figure 1: Berkeley's "Our Pattern Language" Categorization [25]. This figure shows the current OPL patterns by name, breaking them down into their categories: "Application Architectural", "Application Computational", "Parallel Algorithm Strategy", "Implementation Strategy", and "Concurrent Execution". We refer to this figure to show the relationship between the patterns that we have chosen to investigate.

tural support for reasoning about relationships within and between patterns (Section 3), proposes a uniform representation as a solution (Section 4), and discusses three different applications of the representation (Section 5). We then consider the lessons learned through this process and discuss additional implications for parallel patterns (Section 6), and suggest various avenues of future work (Section 7).

2. RELATED WORK

Design patterns are a widely accepted approach to describing a general solution to a frequently occurring problem in software [15]. 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.

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.

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 [6, 8], navigating a larger, disparate set of patterns written by many authors can be more challenging. The Berkeley Parallel Computing Lab [4] provides an overview of recent efforts within the parallel community to provide such a pattern language [20, 17, 18]. 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*, and *related patterns*; 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 variations 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* [3], 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 is 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.

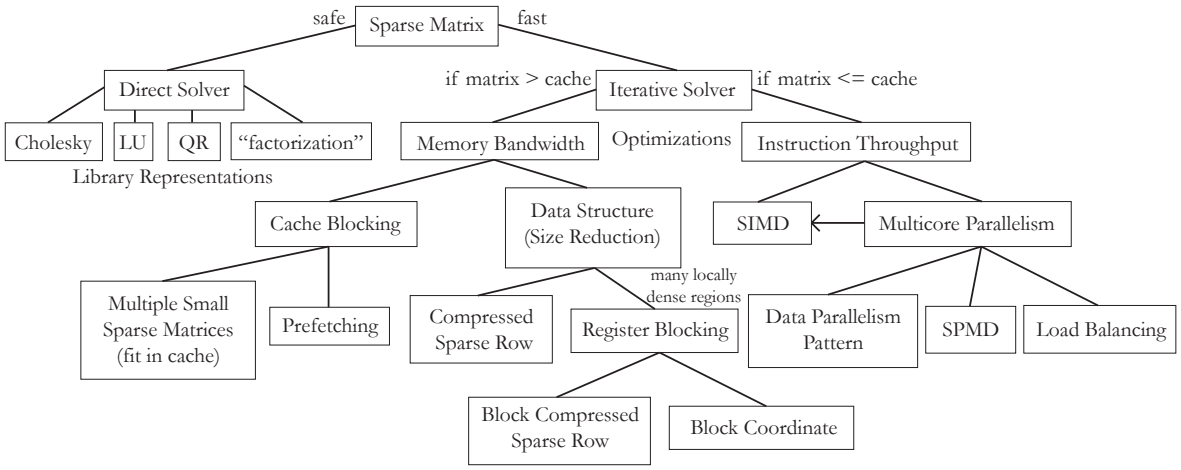

Figure 2: Sparse Linear Algebra Decision Tree [19]. This figure displays our previous attempt to reorganize the solution of Sparse Linear Algebra. It was a direct translation between the solution and a flowchart, and turned out far more complicated than we anticipated. This allowed us to consider that a direct translation was not useful, and provides a visual comparison for the structural additions our uniform representation makes (Figure 4).

Current efforts within the parallel pattern community are also focusing on *methodological* patterns [16, 28] 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 [4]. The intent is to guide developers 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 investigated pattern tradeoffs in the pervasive domain proposed RIPPL [11] (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.

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 [19]. In this study, the pattern under investigation (*Sparse Lienar Algebra* [26]) did not naturally align with the sequence in which the developer had to make design decisions. To aid developers using the pattern, we proposed a refinement: including, as part of the *Solution*, a visual representation of its content which highlighted critical decision points (Figure 2). We believe this proposed format makes the decision points within a pattern more explicit and provides 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.

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 nature that adds to the complexity associated with understanding these concerns within a software artifact. Multi-dimensional separation of concerns [34] proposed a formal approach to modeling and implementing software artifacts with the separation of overlapping concerns across multiple dimensions [31]. Aspect-oriented programming [22] initially provided an approach to explicitly and modularly represent *crosscutting concerns* with linguistic mechanisms [21]. 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 [7] and design [13].

The added complexity associated with parallel-specific software development is amplified by the dynamic impact of large scale design decisions and fine-grained implementation choices. Though current Systems Development Life Cycle (SDLC) models support an iterative and structured approach to software development, they currently do not explicitly take into account the inherent complexities of the dynamic consequences associated with parallel development.

We believe our proposed extension of existing software development models to introduce iterative and systematic workflows structured around key causal relationships of parallel applications [12] should be represented explicitly within design artifacts. That is, the static representation of a design pattern should incorporate a view of the points at which dynamic results will feed back into and force changes to the design and ultimately the implementation.

3. PROBLEM

The problem posed in this paper 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*.

3.1 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.

3.2 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. PROPOSED SOLUTION: HiLPR

Our previous work showed that, to implement a parallel pattern, several design decisions needed to be made. The order in which these decisions we made impacts the resulting complexity of the solution, and as such, our proposed addition to the parallel patterns is designed to guide programmers through these decisions in an order which both supports their previous decisions and increases the complexity of the current one as little as possible.

HiLPR, the concrete application which addresses the problem in Section 3, 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 [23], and a tutorial of the *Sparse Linear Algebra* problem using OpenCL [5]. 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.

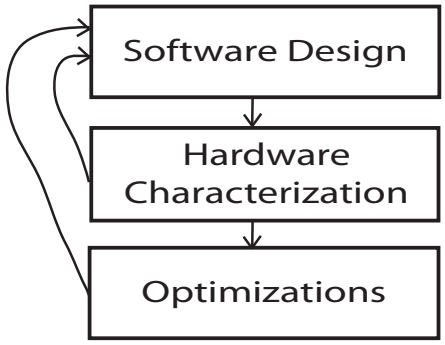

Figure 3: HiLPR, as an abstract uniform representation. This figure shows the abstract structure we suggest governs the solutions of the OPL parallel patterns, broken into three different stages. The arrows suggest the relationships and transitions between the stages for software development purposes.

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 paper 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 the problem of a single pattern with a flowchart that contains pertinent information from all the sections of the pattern. We ap-

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

Table 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.

ply our process to *Sparse Linear Algebra* [26] (Section 5.1), *Pipeline* [24] (Section 5.2), and *Shared Queue* [29] (Section 5.3), and show that either the *Forces* or the *Solution* separately do not contain all of the details necessary to make well-informed decisions about pattern implementation. This process is similar to the iterative development model of software engineering [33], and can be considered localized to the abstraction of parallel programming patterns.

4.1 Uniform Representation

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 3.

4.1.1 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.

4.1.2 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 [33], in constrast to the current sequential process seemingly espoused by the patterns.

4.1.3 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 5.3), or that a necessary optimization for performance is impossible, such as if SIMD isn't available in the *Sparse Linear Algebra* example (Section 5.1), 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.

5. APPLYING HiLPR TO PATTERNS

This section applies HiLPR to three different patterns: *Sparse Linear Algebra* [26] (Section 5.1), *Pipeline* [24] (Section 5.2), and *Shared Queue* [29] (Section 5.3). 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. Blue and green colouring indicates that the information is directly taken from the pattern's *Solution* and *Forces* respectively. Black and red colouring indicates that the information is our addition: black indicates that the addition relates to HiLPR, while red shows additions that we have made that do not fall under HiLPR's structure. With this colouring, notice how the pieces from the *Solution* and *Forces* are organized—intertwined—in these diagrams, 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 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*.

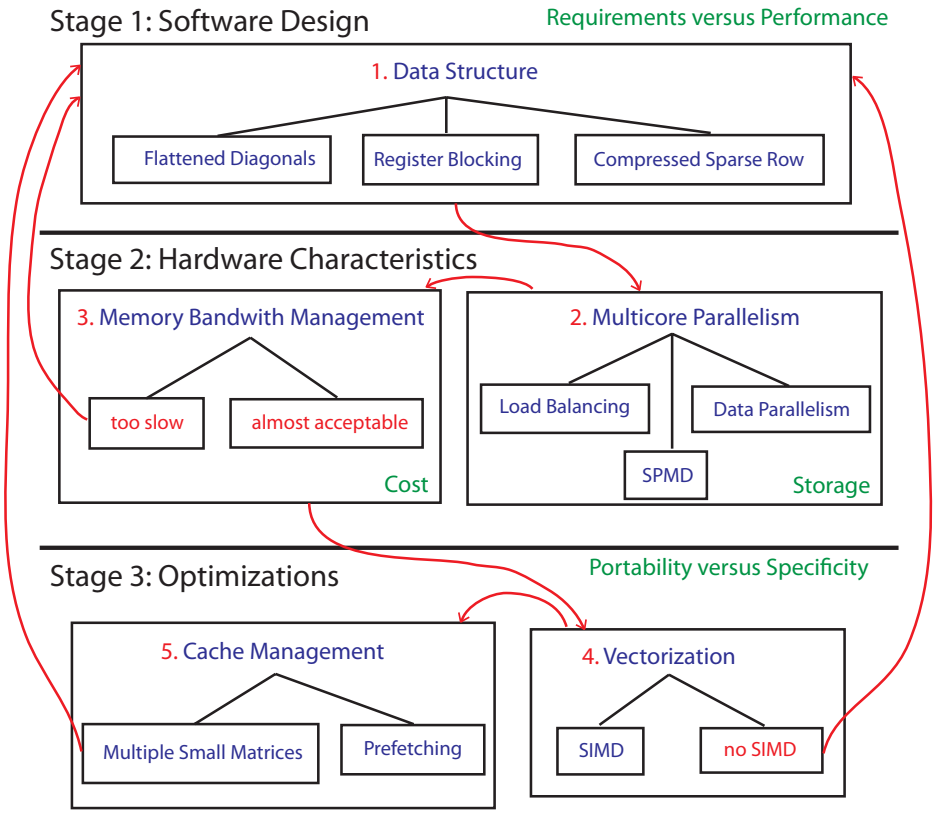

Figure 4: Sparse Linear Algebra. This figure displays how the Sparse Linear Algebra parallel design pattern's solution breaks down into HiLPR's structure. Each stage is populated with steps from the pattern's solution, and each step contains specific decision points that are suggested by the pattern. 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.

We have chosen each of these patterns based on their relationship to our previous work on the Adaptive Optics problem for the thirty-metre telescope [14]. *Sparse Linear Algebra* is the focus and inspiration for this uniform representation, and is directly related to the Adaptive Optics problem. *Pipeline*, another option to solve this problem, makes use of *Shared Queue* and so we chose to study it as well.

Table 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.

Each case study is then broken into three sections for discussion—the *Software Design*, the *Hardware Characteristics*, and the *Optimizations*—mirroring HiLPR's abstract uniform representation, to address how the pattern fits within our structure.

5.1 Sparse Linear Algebra

The proposed visual representation of the *Sparse Linear Algebra* Design Pattern is shown in Figure 4. The external structure was determined by HiLPR, with the internal structure of the solution guided by our previous work on this pattern [19]. This representation 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; they are weaker because they are not explicitly tied to implementation decisions.

5.1.1 *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.

This stage requires the developer to consider the "Requirements versus Performance" theme, which is the third force listed in the pattern description. Our representation demonstrates that this force influences the first decisions that a developer must make, as the chosen data structure impacts the program's performance.

5.1.2 *Hardware Characteristics*

The next two steps of the solution both fall into the *Hardware Characteristics* stage of the uniform representation. These are interrelated problems, although they are expressed sequentially in Figure 4 and Table 2, as "Multicore Paral-

lelism" impacts the choices made for "Memory Bandwidth Management".

The theme in this stage is one of "Storage verus Cost", the first force in the pattern description. This decision is essentially broken down into the use of two distinct resources: cache space and bandwidth. "Multicore Parallelism" considers how cache space may be used to reduce redundant calculations and communication overhead. Conversely, "Memory Bandwidth Management" considers reducing the burden on the cache by recomputing intermediate results and communicating them across processing elements.

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.

5.1.3 Optimizations

In the final stage, *Optimization*, we consider the theme of "Portability versus Specificity", the second force listed in the pattern. We consider both vectorization and cache management as Specific optimization decisions that impact the Portability of the software. "Vectorization" is a crucial optimization—the ability to do simultaneous computation on multiple sets of data is at the heart of high-performance parallelism. If vectorization is unavailable for any reason, the design must be seriously reconsidered, and HiLPR suggests returning to the *Software Design* stage to do so.

The final optimization suggested by the pattern guides developers to consider the structure of the cache. One of the possible considerations for this step is to determine whether the matrix can be decomposed into pieces, each of which will be able to fit into the cache. If this is the case, there is potential for optimzation by returning to the *Software Design* Stage with the aim of incorporating this information into the choice of the "Data Structure".

5.1.4 Summary

Finally, we provide a summary of the *Sparse Linear Algebra* case study, breaking the information from our image down into Table 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.

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

Table 2: Summary of Sparse Linear Algebra

5.2 Pipeline

The visual representation of the *Pipeline* parallel design pattern, shown in Figure 5, 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 [19].

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.

5.2.1 Software Design

The *Software Design* stage is dominated by one decision: whether the *Pipeline* should, in general, have few or many stages. Although defining the stages is a part of the solution, the discussion on the length of the pipeline is found in the *Forces* section, as the only universal force described by the pattern. This force specifically discusses the tradeoffs with regards to the characteristics of the resulting pipeline: deep pipelines have better throughput while short pipelines reduce latency.

The second step of the solution, "Structuring the Computation", provides two main choices, both of which are from the solution. The first is to use the SPMD (Single Program, Multiple Data) pattern where each stage of the pipeline would be considered a case inside a switch statement; the second is the modular approach, using object-oriented frameworks as the pipeline, with each stage mapped to an object. We have added a third option that has not yet been considered: allowing each stage to be a parallel program. This suggestion occurs in the pattern in the next design stage. At that point, it is too late to consider as the decision will have already been made, which is why it has been moved it forward in the uniform representation.

5.2.2 Hardware Characteristics

The second stage of the representation, *Hardware Characteristics*, contains two of the steps of the *Pipeline* solution. This first step in this stage is to "Represent the Dataflow" of the pipeline, which, through the forces that are realized in this section, refers to hardware management. The first decision to be made is presented in the *Forces* section of the pattern, and is the second of the two implementation forces. The decision focuses on what sort of hardware the solution will be implemented upon.

Next, we can consider another of the pattern's *Forces*, whether the solution will contain multiprocessors on one node or whether it will use several nodes on one cluster. Choosing to use general-purpose hardware, likely for portability requirements, will limit the developers choices.

After these decisions, HiLPR redirects us to the solution to cpmsoder how data is going to move between the stages—a difficult decision without having some understanding of the underlying hardware. The choices that the pattern suggests are message passing using MPI (Message Passing Interface),

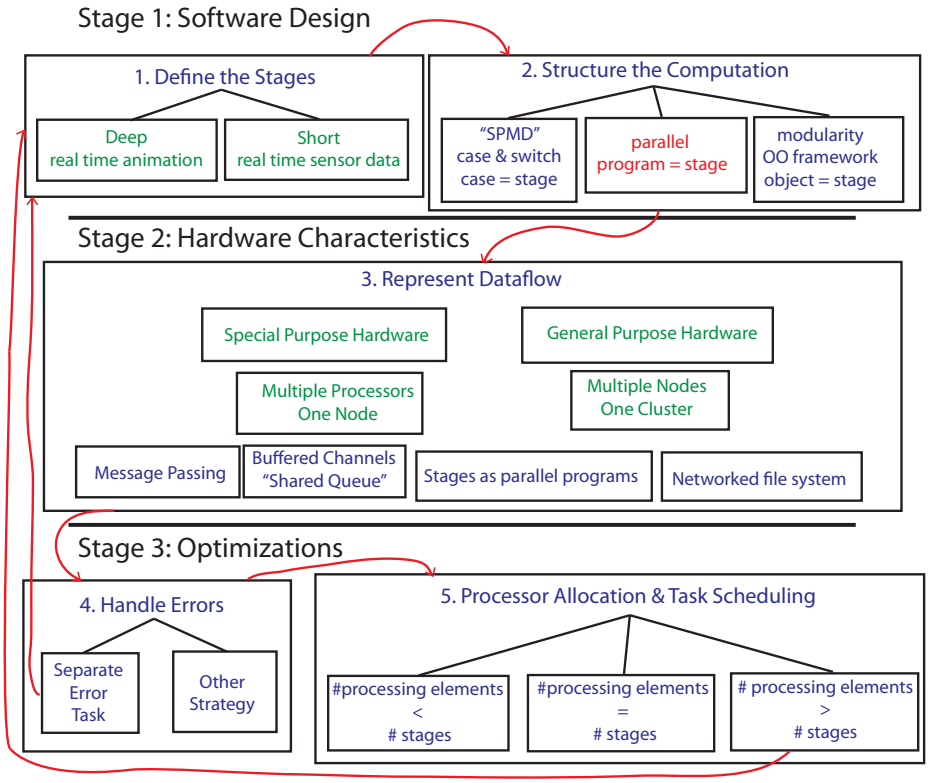

Figure 5: Pipeline. This figure displays the Pipeline parallel design pattern's solution, divided between HiLPR's stages. Each step within a stage, like in Sparse Linear Algebra, contains different choices suggested by the pattern. Step 3, "Represent Dataflow", has a larger number of choices than any 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.

buffered channels in a *Shared Queue* structure, implementing each stage as parallel programs, or using a networked file system to manage the underlying data.

5.2.3 *Optimizations*

The first optimization for a *Pipeline* is deciding how to handle errors. Since the problems that the *Pipeline* parallel design pattern solves tend to be distributed programs, error handling is more complicated than for a single, self-contained program. The solution of the pattern considers this as being important, and suggests having some sort of parallel pipeline stage to handle errors that only executes if any of the other stages send error notifications. We suggest returning to the "Define Stages" step if an error handling stage is used to consider the implications that a new stage has on the overall structure of the solution.

The final step of the solution is to consider how to allocate the processors and tasks between the various stages. In this case, the pattern breaks the problem into all three cases: fewer processing elements than stages, the same number of processing elements as stages, and more processing elements than stages. The middle case is considered simple, the first case complex, with suggestions on how to distribute between the processing elements. The last case allows for greater realizations of concurrency, suggesting that the stage-definition could be further optimized, represented in our flowchart as the arrow leading back to the first stage.

5.2.4 *Summary*

Finally, we provide a summary of the *Pipeline* case study, breaking the information from our image down into Table 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**
1. Define Stages	Deep versus Short
2. Structure Computation	
3. Dataflow Managing Data	Special or General Hardware Multiple Processors or Nodes
4. Handle Errors	
5. Processor Allocation, Task Scheduling	

Table 3: Summary of Pipeline

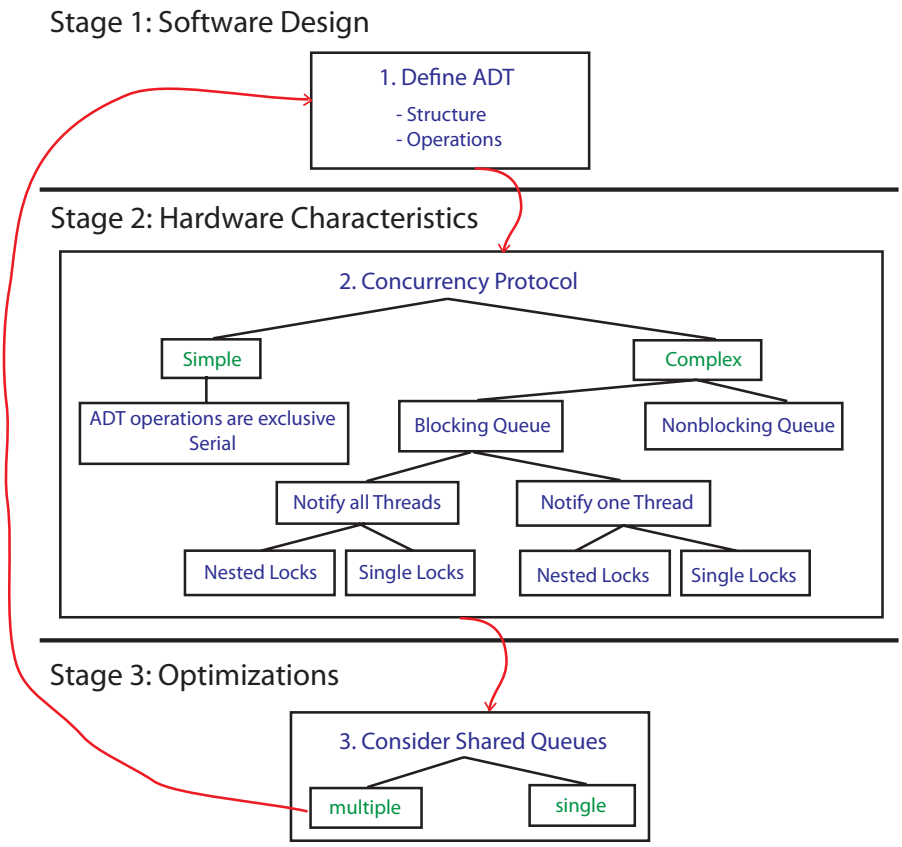

Figure 6: Shared Queue. This figure displays the parallel design pattern Shared Queue as it fits into HiLPR. Shared Queue is tightly connected to its forces, each of which change the internal structure of the steps. A complex queue is much more complicated than a simple one, shown by the number of additional 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.

5.3 Shared Queue

HiLPR's representation of *Shared Queue* is shown in Figure 6. *Shared Queue*, like *Pipeline*, has a solution section that is broken up into three different steps, each of which correspond to one of our stages.

5.3.1 Software Design

The first stage, *Software Design*, aligns with the first step of the solution, which is to define the behaviour of the abstract data type for the *Shared Queue*. The solution takes the reader through the two main decisions to consider, both the structure of the queue and the operations that may be performed on it.

5.3.2 Hardware Characteristics

The second stage, *Hardware Characteristics*, handles the second step of the pattern—the "Concurrency Protocol" that will manage the parallel structure of the solution. The first choice that must be made is suggested in the *Solution*, but more fully outlined in the first of the *Forces*: whether the concurrency should be implemented with simple structures, or complex ones. A simple protocol gives the benefit of being less error prone, while a complicated protocol will allow for much greater fine-tuning and optimizations, with the added risks that complexity entails. If it is decided that the simple path should be followed, the speed of execution is not likely a major issue, and the bulk of the pattern will be unnecessary to the developer. On the other hand, if the developer wishes to manage the complexity for the greater performance benefits, there are a series of decisions which much be made that are now found in the *Solution*.

The first decision for a complex "Concurrency Protocol" is whether the queue should be blocking or non-blocking. A non-blocking queue is a simple structure that does not require additional insight to use. A blocking queue, however, increases the complexity again, which causes a set of different questions that need to be considered. If a queue blocks, it means that the threads that are trying to access it are waiting for some sort of notification. A programmer must decided whether all of the threads need to be notified when access is restored or only those threads that are waiting. The second decision is whether to use nested locks to manage queue access or not.

5.3.3 Optimizations

The final stage, *Optimizations* looks at the final step in the

pattern—"Considering Shared Queues". The pattern suggests that performance bottlenecks may be avoided if multiple queues are used. Multiple queues, however, are not always possible depending on the system. Should the user wish to use multiple queues, we suggest that they return back to the first stage, since this may change many of the other decisions that had been previously made—like previous *Optimizations* in *Sparse Linear Algebra* and *Pipeline*, we cannot simply begin the design process assuming that there will be multiple queues. Without the single queue implementation to compare against, we are unable to properly gauge the performance benefits of using the more complicated structure.

5.3.4 Summary

Finally, we provide a summary of the *Shared Queue* case study, breaking the information from our image down into Table 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**
1. Define ADT	Deep versus Short
2. Concurrency Protocol Queue Behaviour Thread Behaviour Locking Behaviour	Simple versus Complex
3. Shared Queues	Single versus Multiple

Table 4: Summary of Shared Queue

6. LESSONS LEARNED & DISCUSSION

This section discusses the evaluation of HiLPR, including a comparison to our previous work, the benefits of adoption, and the scalability benefits.

6.1 Comparison to Previous Work

To evaluate HiLPR, we compare it to previous work in visualizing pattern solutions [19], and discuss how we solve the issues presented by those visualizations.

The previous work expressed concern when the implementation chose multiple optimizations where it appeared that the *Sparse Linear Algebra* design pattern only suggested that one was necessary. Furthermore, the chosen optimizations fell on both sides of the *matrix* > *cache* and the *matrix* ≤ *cache* divide proposed by their flowchart (Figure 2), implying that the decision process was not as stringent as it appeared. With HiLPR, the iterative process manages these concerns. HiLPR specifies multiple forms of optimization, and as such, does not invite that same sort of cognitive dissonance.

Another concern that had been expressed was that the proof of concept implementation followed a significantly different path than what was suggested by the pattern. This issue has been solved through the iterative method expressed in the visual representation of HiLPR. Further confirmation of our process comes from its successful application to additional parallel patterns.

For comparison, consider the previous flowchart (Figure 2). Although both this one and the current flowchart (Figure 4) for the *Sparse Linear Algebra* design pattern seem quite different, they are actually similar in structure. For example, the decision point titled "Memory Bandwidth" previously had two choices: reducing the size of the data structure and managing access to the cache. In our current work, although "Memory Bandwidth" is no longer one of the first steps, the two choices that follow are still the same. The first returns to the previous stage, where data structure decisions are considered, while the second continues on to the final stage, where cache management is one of the suggested optimizations.

6.2 Benefits of a Uniform Representation

The current organization of the patterns (*Problem*, *Context*, *Forces*, *Solution*) is a good method of grouping like-information, but makes it difficult to use the pattern to actually solve a programming design problem. HiLPR regroups information into a sequential series of algorithmic steps that are designed to help guide a programmer through an efficient solution-process. This is an addition to the pattern language, not a redesign, and certainly not an additional pattern itself. Both representations have their purpose.

Futhermore, some people find that diagrams can help them gain an intuitive feel of a process better than reading a text. HiLPR provides that benefit, *without* taking away from those who are already comfortable with patterns as they are.

We anticipate that HiLPR is applicable to many patterns in the OPL, as we have applied it to a range of the OPL's categories (shown in Table 1). However, we feel that we can also glean more information about patterns that do not neatly fall into HiLPR's structure, as we will discuss in Section **??**, which describes a pattern that does not fit the representation.

6.3 Composition

With HiLPR, we can consider methods of analyzing patterns and pattern composition. We discuss how the visual representation can allow us to reason about the level of abstraction in a pattern, and then specifically consider a case based off of the patterns previously discussed in this paper.

6.3.1 Hierarchy of Abstraction

HiLPR allows us to consider the categorization of patterns such as *Sparse Linear Algebra* and *Pipeline* with relation to each other. *Sparse Linear Algebra* is currently classified as an "Application Computational" pattern, while *Pipeline* is an "Algorithm Strategy" pattern. However, when we compare representations made from the patterns in the same manner, we see that the type of discussion is very different.

Consider: *Sparse Linear Algebra's* flowchart contains a detailed set of discussions and optimizations for solving the problem of solving a sparse matrix system. As a pattern, *Sparse Linear Algebra cannot* be used for anything other than solve Sparse Matrices—it is that specific. *Pipeline*, on the other hand, solves the problem of a much more general flow of data through different stages.

6.3.2 Sparse Linear Algebra within Pipeline

Although there may be instances of *Pipeline* that are low-level, in the same vein as *Sparse Linear Algebra*—we are

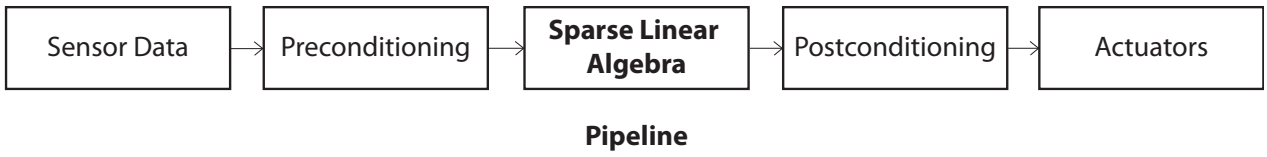

Figure 7: Composition of Pipeline/Sparse Linear Algebra in an Adaptive Optics context. This figure shows a concrete example where one pattern can be placed into the structure of another pattern, becoming a piece of its solution. This example of composition now allows us to consider the relationship between the two patterns.

also capable of applying the pattern to the problem shown in Figure 7. This figure contains an instance of *Pipeline* that traces the dataflow from collection by sensors, preparing it for manipulation, solving the resulting sparse matrix, preparing that data for use, and modifying the actuators based on the data—the process required by the adaptive optics problem [14]. Here we have an instance of a *Pipeline* containing an instance of *Sparse Linear Algebra*.

Furthermore, while we can imagine *Sparse Linear Algebra* being a stage of *Pipeline* (Figure 7) while the other way around does not make any sense, as a general pattern has difficulty playing a role within a specific one.

This propety even appears in their visual representations. *Pipeline* considers how to manage nebulous "stages", any one of which may be another, more specific pattern. *Sparse Linear Algebra*, though, is very direct. We cannot fit *Pipeline* into *Sparse Linear Algebra's* flowchart, since the data does not flow—the computation is well defined and not organized for that structure.

6.4 Scalability

This section is titled scalability, 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.

6.4.1 ...of 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.

6.4.2 ...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. Doing both processes concurrently 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.

6.4.3 ...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. As we will describe in Section 6.3.2, HiLPR gives us a new vocabulary to help compare two patterns. 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.

6.4.4 ...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 [10]. However, the other stages of the representation (*Software Design* and *Optimizations*) apply. This allows us 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.

7. FUTURE WORK

We suggest that HiLPR allows for further exploration of how we categorize and compose patterns. This work is based on the relationships that can exist between patterns, which give us clues to how they fit together in composition.

7.1 Pattern Categorizations

The way that patterns are organized into their categories changes how we think about them with relation to each other.We must consider categorization carefully—based on current OPL categorizations, we would not suggest placing

an instance of *Sparse Linear Algebra* within an implementation of *Pipeline*, but we have shown in Section 6.3.2 that it is a reasonable decision. This section explores levels of abstraction as they apply to parallel patterns.

7.1.1 Level of Abstraction in Patterns

HiLPR allows us to more easily determine the level of abstraction implicit to a parallel design pattern. This is an important step for understanding pattern composition, as patterns on the same level can compose in different ways then patterns between levels. Take *Three Layer Cake* [32] (Figure 8) as an example: Message Passing is a highly abstract method of managing parallelism and SIMD (Single Instruction, Multiple Data) is a low-level implementation method. Three Layer Cake suggests that SIMD, being so low level, should not be creating Fork-Join or Messages to pass, and Fork-Join, in the middle, should also not be spawning Messages. This structure is a heirarchy, from the most abstract to the least. If we can leverage this sort of heirarchy as clues to composition, and our representation as a clue to the position a pattern holds in the heirarchy, we have a structure that can define certain sorts of composition easily, in a way that was previously difficult.

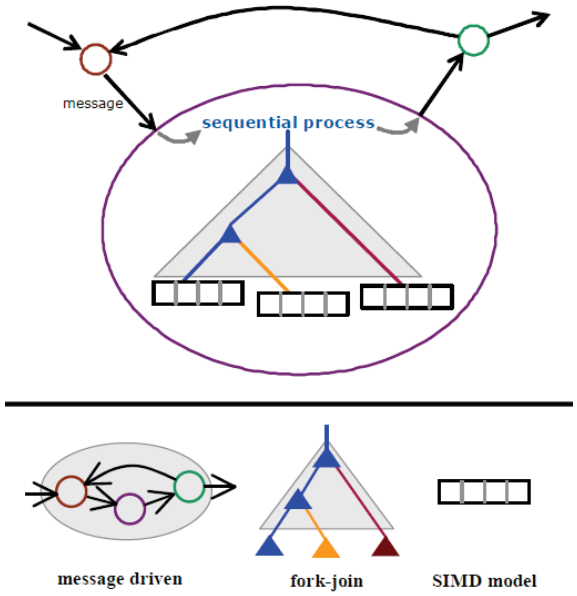

Figure 8: Three Layer Cake [32]. This pattern is built of three other parallel design patterns. Each individual pattern is shown below the composition. The outer most layer is *Message Passing*, inside the message is *Fork-Join*, which eventually gives way to *SIMD*.

7.2 Relationships between patterns

We can further consider pattern composition and abstraction through another means—by examining the various relationships between patterns. This parallels work done on the original GOF patterns.

7.2.1 GOF Relationships

Zimmer's [36] "relationships between design patterns" suggested the following relationships to help analyze pattern selection and composition:

- X uses Y
- X is similar to Y
- X can be combined with Y

We consider these relationships an interesting idea that can also be applied to parallel patterns to help consider the question of composition. The following section suggests an additional relationship that wil make this set more complete with regards to parallel patterns.

7.2.2 Uniform Relationships

Based on HiLPR, we can add an additional relationship for parallel patterns that is more explicit than those Zimmer suggested for Object-Oriented patterns. This relationship is "X includes Y". This relationship is different from X uses Y, for in that relationship, X requires Y to function. In this relationship, X does not require Y—X is more abstract than Y, and one of the pieces of X could be another pattern—in this case, Y. We show an example of this with *Pipeline* and *Sparse Linear Algebra* (in this case, *Pipeline* includes *Sparse Linear Algebra*) in Figure 7. This relationship is also different from one where X can be combined with Y, which gives an equal participation to both patterns as patterns that typically work together.

In our proposed relationship, X has no particular ties to Y—consider again *Three Layer Cake* [32]. We can say that Message Passing includes Fork-Join—this does not mean that the only message that can be passed is a Fork-Join, only that Fork-Join is one of many that could be composed into a message.

7.3 Tool Support for Patterns

Using HiLPR with parallel design patterns, tool support for patterns becomes easier to manage as the visual representation provides a clear structure for developers to work with. This section considers the purpose that these tools may serve, based on the properties of each stage of the uniform representation. A summary can be seen in Table 5. Inclusion in this table does not mean that these tools do not exist, or that they must be built entirely from scratch. What we describe are three pieces of an integrated environment.

Stage	Tool Properties
Software Design	Integrated Design Analysis
Hardware Characteristics	System Characterization
Optimizations	Automated Platform Dependencies

Table 5: Properties of Pattern Tool Support. This table shows a summary of HiLPR's stages and the properties that tool support could provide. Each property is organized by the stage where it will be most useful.

The first stage, *Software Design*, could have tool support that fufills much the same purpose as the overall structure of

our uniform representation: keeping information organized and up-to-date. A tool for this stage would assist managing the other tools, and could preliminarily check design changes against the other stages to determine what areas of the program are most affected. These areas could then be examined in further detail.

The second stage, *Hardware Characteristics*, could be integrated with the underlying hardware, providing an easier manner in which to check the design against the implementation system. This tool would be responsible for ensuring that the design meshes well with the hardware, and would interrupt the *Software Design* tool should changes in the previous stage become unmanageable.

The final stage, *Optimizations*, can be broken into two general types of queries. The first are design decisions: for example, a *Shared Queue* using multiple queues. These optimizations are difficult to provide tool support for. On the other hand, the other type of optimization is ripe for tool enhancement. These are platform-specific optimizations, precisely fine-tuned to the program's characteristics, that do not vastly modify the structure of the program itself. These optimizations are time-consuming and elaborate, but they also have well-defined metrics for success. These are the sorts of optimizations that tool support could help to automate.

8. CONCLUSIONS

This paper addresses the problem of diversity between different patterns in the OPL, and the needs of programmers versus the structure of the patterns. We propose HiLPR, a visual and uniform representation to apply to the OPL patterns which will address both of these issues. HiLPR breaks the solution to a pattern into three stages: *Software Design*, *Hardware Characterization*, and *Optimization*. We show how the visual representation can be applied to multiple patterns in different categories in the OPL, defend the validity of this process based on a comparison with previous work and an analysis of our result, and describe the benefits of having this particular uniformity across these patterns.

We discuss further benefits beyond implementation, which will make it easier to write and compare new patterns for relationships and composition. We show how patterns that do not fit HiLPR can still gain some benefit, and are in and of themselves important results that suggest they may be meta-patterns. We also discuss how this method allows us to consider the content of the pattern, and whether some of the *Forces* are strong guides for implementation.

Finally, we recommend interesting avenues of research that build upon work analyzing the Gang of Four patterns, work that we have been unable to extend to the OPL patterns until now. We further leverage the visual representation to discuss the effectiveness of the current OPL pattern categorizations, and use it as a framework to consider the strengths and weaknesses of the current organizational strategy.

9. ACKNOWLEDGMENTS

The authors would like to thank Sebastian Günther, Cristoph Fehling, Pedro Monterio, Mehdi Mirakhorli, and Ralph Johnson for their time and reviews at the PLoP Writer's Workshop. The authors would also like to thank Mary Curtin for her help shepherding this paper. Support for this research has been provided by MITACS in a partnership with IBM.

10. REFERENCES

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

[3] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. volume 0, pages 352–362, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[4] Berkeley Parallel Computing Lab. A pattern language for parallel programming ver1.0. `http://parlab.eecs.berkeley.edu/wiki/patterns/pattern1_0`, 2010.

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

[6] N. Chen, R. K. Karmani, A. Shali, B.-Y. Su, and R. Johnson. Collective Communication Patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[7] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[8] D. Deugo, M. Weiss, and E. Kendall. Reusable patterns for agent coordination. In *Omicini, A., Coordination of Internet Agents*, pages 347–368. Springer, 2001.

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

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[11] C. Gibbs and Y. 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.

[12] C. Gibbs and Y. Coady. Parallelization and the Application Programmer: Random Self-Oscillation or Old Faithful? Reno, NV, USA, 2010.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37:161–173, November 2002.

[14] G. Herriot et al. NFIRAOS: TMT's facility adaptive optics system. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7736 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, July 2010.

[15] Hillside Group. Home of the Design Patterns Library and Host of the PLoP Conferences. http://hillside.net/, 2010.

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

[17] R. Johnson, K. Keutzer, and T. Mattson. Mechanisms that separate algorithms from implementations for parallel patterns (paraplop). March 2009.

[18] R. Johnson, K. Keutzer, and T. Mattson. Workshop on parallel programming patterns (paraplop). March 2010.

[19] D. Kaminskyj Long, L. Kiemele, C. Gibbs, A. Brownsword, and Y. 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.

[20] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal: Addressing the Challenges of Tera-scale Computing*, 13(4), 2010.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer-Verlag, 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

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

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

[25] T. Mattson and K. Keutzer. Our pattern language (opl). In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2009.

[26] T. Mattson and M. Murphy. Sparse linear algebra pattern, 2009. `http://parlab.eecs.berkeley.edu/wiki/patterns/sparse_linear_algebra`.

[27] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[28] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

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

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

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

[32] A. D. Robison and R. E. Johnson. Three layer cake for shared-memory programming (paraplop), 2010. `http://www.upcrc.illinois.edu/workshops/paraplop10/papers/paraplop10_submission_8.pdf`.

[33] 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.

[34] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. volume 0, page 107, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

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

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