HiLPR:
The High-Level Pattern Representation with Pretty Pictures

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

Celina Gibbs
University of Victoria
celinag@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. Using 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, 35] on the original Object-Oriented patterns [14, 15], including; relationships between patterns [44], and composition of patterns [43]. Less attention has been applied to parallel pattern languages such as “Our Pattern Language” [32, 30] (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.

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 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 [24]. We show how HiLPR fits with the information presented by these patterns. Each pattern is chosen from a different section of the OPL’s categorization, lending strength to our analysis. We describe not only the benefits of a uniform representation, but the benefits of this particular uniform representation, and the intriguing relationships we can find between patterns in this form. We discuss various analyses
that HiLPR allows, analyses that were difficult to attempt without the additional structural support the proposed representation supplies.

After an overview of related work (Section 2), this paper discusses the problems associated with the lack of structural 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 evaluate the results of our case studies (Section 6), discuss the greater implications for parallel patterns (Section 7), and suggest various avenues of future work (Section 8).

2. RELATED WORK

Design patterns are a widely accepted approach to describing a general solution to a frequently occurring problem in software [20]. 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 browsable 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 [3] also provide structure to lead a user through a collection of patterns. Though individual pattern languages have been successfully defined within smaller subdomains [9, 13], navigating a larger, disparate set of patterns written by less collaborative authors can be more challenging. The Berkeley Parallel Computing Lab [5] provides an overview of recent efforts within the parallel community to provide such a pattern language [25, 22, 23]. 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. Alterna-
Our previous case study investigating pattern tradeoffs in the pervasive domain proposed RIPPL [16] (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 [24]. In this study, the pattern under investigation (Sparse Linear Algebra [31]) 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 soft-
Multi-dimensional separation of concerns [42] proposed a formal approach to modeling and implementing software artifacts with the separation of overlapping concerns across multiple dimensions [36]. Aspect-oriented programming [27] initially provided an approach to explicitly and modularly represent crosscutting concerns with linguistic mechanisms [26]. 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 [12] and design [18].

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 [17] 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, described in detail below, involves the way individual sections of a pattern are written. The second issue pertains to the challenge of understanding how to use all of the details split across the sections of Problem, Context, Forces and Solution of a given pattern. This problem is structural—its solution does not require a separate pattern, and any solution for this problem is not going to be a programming pattern.

3.1 Internal Structure

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, 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, we argue that there is an implicit relationship across these sections which is necessary to consider when implementing a pattern, 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.

To compound this problem, not all patterns are written by the same author and while there may be uniformity in terms of what the sections are that make up a pattern, how those sections are written is by no means constant. 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 diversity makes pattern use challenging for a developer trying to collect relevant implementation information across the pattern sections.

3.2 External Structure

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 need to move back and forth between the Forces and the Solution. In software engineering, the Waterfall method [38] is taught as a starting point and leveraged to explain to students the benefits of an iterative method. 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 itself.

4. PROPOSED SOLUTION: HiLPR

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 [28], and a tutorial of the Sparse Linear Algebra problem using OpenCL [7]. 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 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 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 a patterns’ problem with a flowchart that contains pertinent information from all the sections of the pattern. We apply our process to Sparse Linear Algebra [31] (Section 5.1), Pipeline [29] (Section 5.2), and Shared Queue [34] (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 [38],
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 Characterization, 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 Characterization 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 Characterization

Hardware Characterization 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 [38], 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 [31] (Section 5.1), Pipeline [29] (Section 5.2), and Shared Queue [34] (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. Text in black is HiLPR’s structure, blue is from the Solution, green from the Forces, and red are our structuring additions to the pattern. 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 (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.

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 [19]. 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.
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 [24]. 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. Further elaboration can be found in Section 7.3.1.

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 Parallelism” impacts the choices made for “Memory Bandwidth Management”.

The theme in this stage is one of “Storage versus 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 optimization 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.

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.
Stage 1: Software Design

- Requirements versus Performance
- Portability versus Specificity

1. Data Structure
   - Flattened Diagonals
   - Register Blocking
   - Compressed Sparse Row

Stage 2: Hardware Characteristics

- Memory Bandwidth Management
- Multicore Parallelism
  - too slow
  - almost acceptable
- Vectorization
  - Multiple Small Matrices
  - Prefetching
- Cache Management

Stage 3: Optimizations

- Performance versus Portability

- SPMD
- Load Balancing
- Data Parallelism
- SIMD
- no SIMD

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.

Table 2: Summary of Sparse Linear Algebra

<table>
<thead>
<tr>
<th>Solution</th>
<th>Forces</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data Structure</td>
<td>Requirements v. Performance</td>
</tr>
<tr>
<td>2. Multicore Parallelism</td>
<td>Storage</td>
</tr>
<tr>
<td>3. Memory Bandwidth</td>
<td>Cost</td>
</tr>
<tr>
<td>4. Vectorization</td>
<td>Portability v. Specificity</td>
</tr>
</tbody>
</table>

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 [24].

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 structures 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 Computa-
Stage 1: Software Design

1. Define the Stages
   - Deep real time animation
   - Short real time sensor data

Stage 2: Hardware Characteristics

3. Represent Dataflow
   - Special Purpose Hardware
   - General Purpose Hardware
     - Multiple Processors
     - Multiple Nodes
     - One Node
     - One Cluster
     - Message Passing
     - Buffered Channels
     - "Shared Queue"
     - Stages as parallel programs
     - Networked file system

Stage 3: Optimizations

4. Handle Errors
   - Separate Error Task
   - Other Strategy

5. Processor Allocation & Task Scheduling
   - #processing elements < # stages
   - #processing elements = # stages
   - #processing elements > # stages

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.

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 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 cpmsoer 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), 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 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.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Forces</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Define Stages</td>
<td>Deep versus Short</td>
</tr>
<tr>
<td>2. Structure</td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td></td>
</tr>
<tr>
<td>3. Dataflow</td>
<td>Special or General Hardware</td>
</tr>
<tr>
<td>Managing Data</td>
<td>Multiple Processors or Nodes</td>
</tr>
<tr>
<td>4. Handle Errors</td>
<td></td>
</tr>
<tr>
<td>5. Processor Allocation,</td>
<td></td>
</tr>
<tr>
<td>Task Scheduling</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Summary of Pipeline

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 decide 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.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Forces</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Define ADT</td>
<td>Deep versus Short</td>
</tr>
<tr>
<td>2. Concurrency Protocol</td>
<td>Simple versus Complex</td>
</tr>
<tr>
<td>Queue Behaviour</td>
<td></td>
</tr>
<tr>
<td>Thread Behaviour</td>
<td></td>
</tr>
<tr>
<td>Locking Behaviour</td>
<td></td>
</tr>
<tr>
<td>3. Shared Queues</td>
<td>Single versus Multiple</td>
</tr>
</tbody>
</table>

Table 4: Summary of Shared Queue
6. EVALUATION
This section discusses the evaluation of HiLPR, including a comparison to our previous work, the benefits of adoption, and the scalability benefits. Section 7 addresses further insights we have gleaned from this representation.

6.1 Comparison to Previous Work
To evaluate HiLPR, we compare it to previous work in visualizing pattern solutions [24], 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 $\text{matrix} > \text{cache}$ and the $\text{matrix} \leq \text{cache}$ 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.

Furthermore, 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 nearly fall into HiLPR’s structure, as we will discuss in Section 7.1, which describes a pattern that does not fit the representation.

6.3 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.

6.3.1 ...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 knowledgeable 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.

6.3.2 ...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.

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.

6.3.3 ...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 7.2.2, 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.

6.3.4 ...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 [15]. 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. DISCUSSION
The following sections discuss the conclusions that we can draw about patterns, based on their visual representations. This allows us to consider interesting data from meta-patterns, to composition, to the analysis of pattern forces. This sort of analysis has been done on the GOF patterns, but has yet to have been applied to the OPL.

7.1 Meta-Patterns
With HiLPR, we can consider other facts about patterns that we have not previously had a method of comparing. While applying its representation to patterns associated with the Adaptive Optics problem, we found one that would not fit. It was Task Parallelism, and we suggest that it is actually a different type of pattern altogether.

Interestingly enough, even reading through the Task Parallelism pattern puts it at a different level than one such as Sparse Linear Algebra. Sparse Linear Algebra focuses on hardware- and software-specific optimizations, while Task Parallelism references other parallel patterns. We consider Task Parallelism through both its Context and Solution.

7.1.1 Task Parallelism: Context
Task Parallelism has a broadly-defined problem that has no specific context other than that given to it by other parallel patterns. In particular, Task Parallelism lists:

||
|---|---|
|Cache Management|\|
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.

1. **Agent and Repository** [8], to discuss collections of independent tasks;
2. **Graph Algorithm** [40], to discuss tasks managing the vertices and edges of a graph traversal;
3. **Monte Carlo Methods** [10], to discuss tasks as separate experiments;
4. **Dynamic Programming** [11], to discuss tasks as independent sub-problems;
5. and **Backtrack Branch and Bound** [39], to discuss tasks as parallel evaluation processes.

Although **Task Parallelism** has a context that discusses managing large numbers of tasks, it focuses more on other patterns than on a specific problem itself. We propose that this is because the problem is too broad and too general to be distilled into a single pattern, and instead **Task Parallelism** helps designers find other patterns that are more specific to their problems.

### 7.1.2 Task Parallelism: Solution

We cannot find a structure to the **Task Parallelism** pattern that gives clues to an implementation method, as even in the solution section, **Task Parallelism** references the problems of other patterns such as **Monte Carlo**, **Graph Algorithms**, and **Recursive Splitting** [41]. Based on the results of attempting to apply HiLPR to **Task Parallelism**, we consider **Task Parallelism** as an ‘organizational’ or ‘meta’ pattern that discusses broad patterns and solutions among the other parallel patterns.

### 7.2 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.

#### 7.2.1 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.

#### 7.2.2 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 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 [19]. Here we have an instance of a **Pipeline** containing an instance of a solution to **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 property 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.

### 7.3 Forces

We discuss the validity of forces in patterns as highlighted by HiLPR. We explore **Sparse Linear Algebra**, and discuss what it means when the forces are not an explicit part of a pattern’s visual representation.

#### 7.3.1 Sparse Linear Algebra

HiLPR allows us to consider the validity of some of the forces suggested in the patterns that we have analyzed, specifically...
Sparse Linear Algebra. Compare Figure 4 and Figure 5. The first is the visual representation of Sparse Linear Algebra, the second is the representation of Pipeline. Note how the Forces interact with the Solution—in Pipeline, the Forces are clear decision points that must be directly considered.

However, Sparse Linear Algebra does not show this interplay of Forces and Solution. The Forces in the pattern are more general, and as such do not become actual decisions to be considered. Indeed, in our previous work [24], our implementation completely ignored the forces, finding them frustratingly vague and without the discussion necessary to help the programmer choose which way is the correct one for their particular implementation. This is shown in the current Sparse Linear Algebra flowchart, where we suggest that the forces are themes for each stage rather than actual decisions.

8. 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.

8.1 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 7.2.2 that it is a reasonable decision. This section continues to explore meta-patterns and abstraction as they apply to categorizations.

8.1.1 Task Parallelism and Meta-Patterns
Now that we have a starting point for considering the job and the purpose of Task Parallelism, we can compare this result with recent work completed by Microsoft Research, applying parallel patterns to their current frameworks [6]. In their work, they describe what we consider meta-patterns, superstructures that could contain the more specific patterns outlined in the OPL [30]. One of their patterns is analogous to Task Parallelism. Since it is like a super-pattern in that context, there are parallels that we can draw to the OPL’s version of Task Parallelism that allow us to consider how the OPL treats patterns such as Task and Data Parallelism—ones that are more broad than others.

8.1.2 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 [37] (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 hierarchy, from the most abstract to the least. If we can leverage this sort of hierarchy as clues to composition, and our representation as a clue to the position a pattern holds in the hierarchy, we have a structure that can define certain sorts of composition easily, in a way that was previously difficult.

![Three Layer Cake](image)

Figure 8: Three Layer Cake [37]. 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.

8.2 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.

8.2.1 GOF Relationships
Zimmer’s [44] “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 will make this set more complete with regards to parallel patterns.
8.2.2 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. 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.

8.3 Pattern Composition

The previous sections mention pattern composition as a benefit to HiLPR—that we will be able to reason more effectively about using multiple patterns together when we can look at them all in the same way. We suggest this with the following reasoning: Three Layer Cake considers composition of patterns based on the idea of a level of abstraction creating an implicit hierarchy in the patterns; the visual representation susues out that same abstraction hierarchy by putting patterns into the same form. As Figure 8 shows, this hierarchy can be used to discuss intelligent composition of patterns.

8.4 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.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Tool Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Design</td>
<td>Integrated Design Analysis</td>
</tr>
<tr>
<td>Hardware Characteristics</td>
<td>System Characterization</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Automated Platform Dependencies</td>
</tr>
</tbody>
</table>

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 fulfills 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 rife for tool enhancement. These are platform-specific optimizations, precise fine-tuning to the program’s characteristics, that do not vastly modify the structure of the program itself. These optimizations are time-consuming and fussy, but they also have well-defined metrics for success. These are the sorts of optimizations that tool support could help to automate.

9. 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.

10. ACKNOWLEDGMENTS

The authors would like to thank Mary Curtin for shepherd ing this paper.
11. REFERENCES


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


