PInT: Pattern Instrumentation Tool for Analyzing and Classifying HPC Applications

Fabian Schlebusch*, Yannik Müller*, Sandra Wienke*, Julian Miller*, Matthias S. Müller*
*IT Center, RWTH Aachen University
Chair for High Performance Computing, RWTH Aachen University
Aachen, Germany
Mail: (schlebusch|y.mueller|wienke|miller|mueller)@itc.rwth-aachen.de

***Abstract*—The relationship of application performance to its required development effort plays an important role in today's budget-oriented HPC environment. This effort-performance relationship is especially affected by the structure and characterization of an HPC application. We aim at a classification of HPC applications using (design) patterns for parallel programming. For an efficient analysis of parallel patterns and applicable pattern definitions, we introduce our tool *PInT* that is based on source code instrumentation and Clang LibTooling. Furthermore, we propose metrics to examine occurrences and compositions of patterns that can be automatically evaluated by PInT. In two case studies, we show the applicability and functionality of PInT.**

***Index Terms*—Parallel Pattern, Design Pattern, Pattern Analysis, Application Classification, LLVM, Clang LibTooling, Abstract Syntax Tree, Static Code Analysis, Source Code Instrumentation, Development Effort, High-Performance Computing**

I. Introduction

The ongoing progress of raising HPC performance to the next level evokes an increase in hardware and, hence, programming complexity. Furthermore, HPC centers are extensively faced with power and budget constraints so that software development costs also play an increasing role. As a consequence, HPC co-design activities have evolved, proxy apps as performance abstractions have been established, and numerous programming paradigms have been proposed such as RAJA [12], Kokkos [9] or Chapel [4], that encapsulate different layers of parallelism to ease programming across complex HPC systems.

We take one step back from these efforts and focus on the structure and characterization of HPC applications with emphasis on the relationship of performance and HPC development effort spent to achieve this performance. For that, we target a classification of HPC applications based on design patterns that is able to capture this effort-performance relationship. Once set, this classification gives information on desired features for programming paradigms, guides promising areas for follow-up research, or enables software cost estimations in HPC.

In this context, we focus on the analysis of patterns for parallel programming, i.e., solutions to recurring problems. Our basic assumption is that patterns can be used to describe the algorithm and data structures while the patterns' composition or hierarchy will reveal a *typical* behavior for a relevant subset of HPC applications. We further target the definition and evaluation of metrics that cover (a) typical occurrences and compositions of pattern (sub)sets with the aim to establish a (novel) pattern set that helps focusing on important code characterizations, and (b) development-related efforts such as classic metrics from software engineering.

With this work, we provide tool support to ease this pattern analysis and metrics evaluation. We develop the *P*attern *In*strumentation *T*ool *PInT*[1] that is based on static code analysis by using the LLVM/Clang tool environment. PInT extends the abstract syntax tree (AST) with pattern information provided by code instrumentation by the developer or analyst. From that, we automatically derive numerous metrics for an ample pattern analysis such as lines of codes, fan-in, fan-out, cyclomatic complexity or similarity measures. Furthermore, we provide an interface to easily include further metrics. We show PInT's application using two HPC case studies and evaluate the results produced by the tool.

The remainder of the paper is structured as follows: In Section II, we cover related work. Definitions for patterns in parallel programs are given in Section III. Our current set of metrics to analyze the occurrence and composition of patterns is defined in Section IV. In Section V, we describe our pattern instrumentation tool PInT in detail. Section VI deals with the analysis and evaluation of two case studies using the tool. Finally, we conclude in Section VII.

II. Related Work

Automatic program analysis is a common task in modern optimization compilers which utilizes a multitude of optimization techniques such as dependence, definition, data flow and inter-procedural analysis as described in, e.g., [1], [15], [25]. Moreover, kernel recognition techniques are commonly used for strength reduction [10], performance optimization [17] and automatic parallelization [2]. However, these techniques focus on the recognition of pre-defined sets of detectable patterns. With focus on manual instrumentation, this work enables the calculation of metrics based on user-defined and infinite sets of patterns.

Instrumentation is a common practice to analyze computer programs. In the area of HPC, instrumentation is most noticeably used for performance analysis of a program at runtime.

[1] https://github.com/rwth-hpc/pint

Additional instructions or probes are inserted into a program in order to observe the application's performance. This can happen at various stages [7]. Using source code instrumentation, the programmer can communicate events to a performance measurement tool by adding instrumentation calls to the source code at appropriate locations. For example, TAU [20] or likwid [22] are based on source code instrumentation. Instrumentation can also be applied at library level by wrapping libraries with a layer of instrumentation calls, at binary level by parsing and rewriting an executable or dynamically by using mutators which can insert instrumentation code into a running program. While these instrumentation techniques are typically used for runtime analysis, PInT utilizes static analysis of instrumented source code.

Another approach that uses automated analysis of MPI traces to detect parallel patterns has been researched by Preissl et al. [18]. The authors focus on communication patterns that reproduce a characteristic MPI trace. They define patterns as 'repeating communication structures in an MPI event graph' in contrast to our abstract high-level pattern definition. Their defined communication patterns can be compared with the ones produced by known high-level patterns such as broadcast or stencil operations. But not all (high-level) communication patterns, and rarely other patterns, reproduce such a characteristic trace pattern, therefore only a small subsets of the patterns we are interested in can be identified.

None of the above approaches can immediately be used for the instrumentation of parallel patterns in an application's source code. We require a tool which extracts structural information about a broad range of patterns, indicated through the instrumentation, at compile time. This requires that the tool can lex and parse the source code into a representation from which we can infer structural information. Furthermore, the tool should support the most commonly used languages in HPC, i.e., C/C++ and Fortran, with recent language features and extensions. We build upon an existing compiler front end implementation to parse the source code. Both, the ROSE compiler framework [24] and the Clang compiler [21] provide interfaces which enable tools to extract information from the compilers' internal representation, i.e., the syntax tree. PInT is based on Clang libTooling, considering the following points. While the ROSE compiler, in contrast to Clang, can parse Fortran code, Clang can understand more recent C++ and Open MP standards. Further, Clang libTooling has successfully been used for analysis of C++ programs in the past [8], [19]. Finally, LLVM and Clang development appears to be more active than the development of the ROSE framework. Hence, PInT will profit from ongoing development of Clang and will remain compatible with newer language standards.

III. Parallel Pattern Definitions

Since the PInT tool is designed to work with a large range of patterns, the following provides an overview of the patterns focused on in this work. Furthermore, the methodology used in defining and detecting those patterns is described. Hereby, the patterns are hierarchically grouped into high-level categories of algorithms, mid-level design patterns and low-level code patterns.

A. Algorithmic Classes

To investigate the relations between the general field of application of a software and its typical usage of patterns, we used the high-level categories provided by the 13 'dwarfs' [3]. This is especially useful, since dwarfs are commonly used in the design of benchmarks such as the Rodinia Benchmark suite [5]. Applications not designed by this methodology need to be classified manually. Typically, the purpose of the application as well as the implemented algorithm guide the classification.

B. Design Patterns

The patterns defined by Mattson et al. [13] focus on the design of algorithms. Those patterns are sorted in four design spaces. The 'Finding Concurrency' design space is the most abstract one. It contains patterns that describe where concurrency can be found, which way the code could be parallelized (data parallel, task parallel etc.) and which restrictions exist (groupings, orderings). In the 'Algorithm Structure' design space, patterns describe an algorithm that is adjusted for a type of concurrency described in the previous design space. The most common ones are task parallelism, and geometric decomposition for data parallel algorithms. The 'Supporting Structures' design space covers useful structures and techniques for parallel programming. These are program structures (SPMD, Fork/Join, Master/Worker, Loop Parallelism) on one hand and data structures (Shared Data, Shared Queue, Distributed Array) on the other hand. The last design space, 'Implementation Mechanism', contains some low level actions that every parallel program needs to deal with such as the unit of execution (UE) Management, Synchronization and Communication.

C. Code Patterns

The lowest level of pattern definitions in HPC are code patterns as given by McCool et al. [14]. They focus heavily on efficient implementations of algorithms to improve the scalability and maintainability of codes. They do not deal with higher level design choices described by Mattson and would fit mostly into the supporting program structures and partly into the algorithmic structures of Mattson.

D. Pattern Identification

This work focuses on the design of PInT. To verify its applicability, the design patterns of Mattson are chosen since they provide a balanced granularity and a wide code reach, i.e., they cover large parts of the implementation. The following discusses the methodology used to identify design patterns throughout this work. This includes the classification of a pattern based on Mattson's definitions as well as the identification of the code reach (begin and end) of the pattern.

First, the *occurrence* of a pattern is identified. We identified two main cases in deciding if a pattern has occurred:

- Multiple code sections (e.g., loops or some constructs in SPMD programs) can fulfill the same purpose but are separated by the developer for structural or practical reasons. In this case, they can likely be parallelized with the same pattern since they share the same algorithmic structure. Thus, those code sections are only counted for one pattern occurrence.
- Code sections might use the same pattern but are logically unrelated to each other. Then, it is useful to differentiate between multiple occurrences of the same pattern.

These cases lead towards the following definition of occurrence for this work:

An occurrence of a pattern is a logically connected code section where the pattern is used. One or more programming language constructs can be used to implement the pattern.

While this definition is partly vague, it allows for classification of a wide range of patterns as will be shown in the case studies in Section VI. We used a 3 step process to identify and count the patterns in the manual analysis and the instrumentation.

First, the pattern is classified based on the definitions and examples given by Mattson et al. [13]. The best-matching pattern is chosen based on a thorough analysis of the code. This can lead to ambiguity due to generic definitions of patterns, discrepancies between the structure of algorithms and their actual implementation and overlaps of patterns. In our experience, this can require the assessment of multiple HPC experts.

Second, the code reach of a pattern is identified. Mattson et al. [13] do not provide a definition of where the pattern starts, where it ends or if there are multiple instances of the same pattern. However, typical high-level frameworks like OpenMP utilize closely-confined language constructs which allow for a clear identification of the code reach of a pattern. For OpenMP, e.g., the parallel region often bounds the implemented pattern. Instead, lower-level frameworks like MPI can implement patterns such as data distribution and communication patterns which can span over large parts of the code. The affected parts of code of a pattern, thus, need to be connected through a manual code analysis. This significantly increases the complexity and effort in identifying and specifying the pattern.

Third, patterns get accumulated based on our definitions of occurrences. For example, the data inside a loop parallelization pattern is often accessed via distributed arrays. It is possible to pack all the data into one structure and use one array of structures with the distributed array pattern. However, for performance reasons, it can be beneficial to use structures of arrays instead. Then, each of the arrays in the structure uses the distributed array pattern. Logically, however, they all belong together since they fulfill the same purpose as the array of structures implementation. Therefore, we propose to count such data structures as one occurrence of the distributed array pattern. An example for that is shown in Listing 1. The shown code section from the SRAD (Rodinia) calculates a diffusion coefficient. In the parallel loop multiple arrays use the distributed array pattern. Four of them represent four different directions in the same data structure and the last one, is the diffusion coefficient itself. Since the data is structured in a way that allows every entry to be computed concurrently it is the easiest solution to use the same structure for the resulting diffusion coefficient and reuse the indices. Thus, all usages of the distributed array pattern have a common cause and should be counted only once.

```
#pragma omp parallel for \
    shared(dN, dS, dW, dE, c, rows, cols,...)\
    private(i, j, k, Jc,...)
for (int i = 0 ; i < rows ; i++) {
    for (int j = 0; j < cols; j++) {
        k = i * cols + j;
        Jc = J[k];
        dN[k] = J[iN[i] * cols + j] - Jc;
        dS[k] = J[iS[i] * cols + j] - Jc;
        dW[k] = J[i * cols + jW[j]] - Jc;
        dE[k] = J[i * cols + jE[j]] - Jc;
        [...]
        c[k] = 1.0 / (1.0+den) ;
        [...]
    }
}
```

Listing 1. Excerpt of Rodinia SRAD where multiple patterns are logically connected and, thus, count towards a single occurrence of such pattern.

In general we used the following rules to guide identification and counting occurrences of parallel patterns:

- Finding concurrency patterns are mutually exclusive and do typically cover the whole application, i.e., the pattern occurs once per application.
- Algorithm structure patterns are closely related to a finding concurrency pattern. Therefore, a 1 to 1 relation between those is expected.
- Supporting structures can occur multiple times. However, we only count data structures once per surrounding program structure.

The definition of occurrences as well as the position of those are independent from our tool itself. For the development of the tool it is only necessary to agree on some basic concept such as a pattern can have multiple occurrences and the lines of code affected by these are not always continuous. Adaptations can always be made without affecting the functionality of the tool. They rather have an impact on the quality and expressiveness of the gathered results.

IV. Metrics Definitions

A central feature of PInT is the automatic calculation of pre-implemented metrics from an instrumented source code. However, no concise metrics have been defined in the literature for the analysis of parallel patterns in a source code. Therefore, we propose new metrics and adapt well-known metrics from other fields such as software engineering. In the following, we distinguish between metrics and similarity measures. A metric gives one or multiple concise measures about the analyzed source code. We use the terms statistic and metric exchangeably. Metrics are calculated in order to compare two source codes. For example, we define a metric for the number

of code lines spent for the implementation of a specific pattern. Then, two source codes can be compared with regard to the number of lines used to implement this specific parallel pattern.

When a pattern is implemented in the source code, we call this a pattern occurrence (cf. Section III). Within occurrences of a pattern, often other patterns are employed. We call this *composition* or nesting of patterns. In the most general case, similarity measures quantify the similarity between two comparable mathematical objects, e.g., sets or words. We apply the concept of similarity measures to pattern compositions. Consider the example from Listing 2 and let the function `someCode` contain an occurrence of the `Barrier` pattern. Then, e.g., (`LoopParallelism`, `Barrier`) is a pattern composition. Sequences of composed patterns (or *pattern sequences* for short) are tuples containing subsets of the composed patterns of a composition, while keeping the order of the composition. Considering the earlier example composition, (`LoopParallelism`, `Barrier`) or (`Barrier`) are pattern sequences. A PInT similarity measure yields a measure of similarity for two pattern sequences. For example, a similarity measure can be defined as follows: if two sequences have the same length, they have a similarity of 1. If they have different length, we define the similarity as 0. Then, the similarity of (`LoopParallelism`, `Barrier`) and (`Barrier`) is 0.

The defined metrics and similarity measures can be arbitrarily complex. We propose four metrics and one similarity measure, which we implement in PInT. Further metrics and similarity measures will be defined and implemented in the future.

A. Metrics and Statistics

Two basic statistics are the *pattern count* and *line count* statistics. The pattern count statistic indicates, how often a pattern occurs in the source code. Patterns which occur frequently are more relevant for optimization. As mentioned in Section III, occurrences of patterns comprise one or multiple logically connected code sections. The statistic includes the number of code sections for each pattern such that patterns which occur in a highly distributed manner can be identified. The line count statistic measures the number of code lines over which the pattern is spread. This metric is a rough indicator for the effort necessary to implement a specific pattern. It should also be seen as a reference to whether the chosen granularity of the pattern definition is suitable.

We define a metric following the *cyclomatic complexity* [23] (or McCabe metric) used in software engineering. In software engineering, cyclomatic complexity is a metric for the number of independent paths through a module's source code. It describes how hard a module is to understand, test or modify. The metric is computed using the control flow graph and is defined as $v(G) = e - n + 2$ where e is the number of edges and n the number of nodes in the control flow graph. PInT calculates the cyclomatic complexity from the pattern graph instead of the control flow graph (cf. Section V-B for an explanation of the pattern graph), using the same formula. This will give an insight into the complexity of the composition of parallel patterns in the analyzed source code. For calculation of the metric, PInT refers to a simplification of the pattern graph as shown in Figure 1 (based on the pattern graph from Figure 3). In this representation, the graph's nodes are the pattern code sections which occur in the source code. An edge from node A to node B indicates that the pattern code section B is contained in the code section A, either directly or indirectly through a call to a function containing code section B. For example, the code section `SubMRec` is indirectly composed with `MainParLoop` through a call to `subMethod`. We calculate the cyclomatic complexity of the graph from Figure 1 as $v(G) = 3 - 3 + 2 = 2$. McCabe proposed limiting the cyclomatic complexity to 10. We will have to find a suitable limitation of the cyclomatic complexity for parallel patterns empirically through consideration of compositions of parallel patterns in real codes.

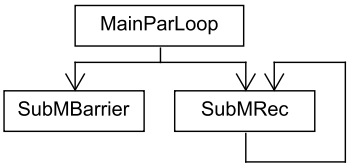

Fig. 1. The modified pattern graph used for calculation of the cyclomatic complexity and illustration of FI/FO. The full pattern graph can be found in Figure 3.

Another metric from software engineering is *fan-out* [11]. The fan-out of a procedure is defined as the number of outgoing information flows plus the number of data structures updated. We define new metrics for parallel patterns and refer to them as fan-in and fan-out (or FI/FO for short), referring to the software engineering metric. The FI of a pattern is the number of all unique pattern occurrences which are one layer above any occurrence of the given pattern in a composition.

$$FI(P) = |\{p \in occurences \mid \exists\, c = (\ldots,\ p,\ P, \ldots) \\ a\ composition\ of\ pattern\ occurrences\}| \\ for\ pattern\ occurence\ P.$$

FO is defined like FI, but the order of p and P is switched in the composition. A graph-based definition can be given to make this measure more conceivable. Consider the graph from Figure 1, used for the definition of the cyclomatic complexity. Then, the fan-in of a pattern is defined as the number of unique pattern occurrences which have an outgoing edge toward an occurrence of the pattern for which we calculate FI. FO can be defined analogously. In the example, the pattern `Recursion` has FI of two. Only the node `SubMRec` has to be considered because it corresponds to the only code section where the `Recursion` pattern is implemented. The graph node has incoming edges from the `MainParLoop` and `SubMRec` nodes. Notice that if `subMethod` was called from within another, different code section of the `MainParLoop` occurrence, the FI of `Recursion` would remain unchanged because only *unique* pattern occurrences count, according to both the compositional and graph definitions. The FO of

`Recursion` is one because `SubMRec` has a self-loop. High FI indicates that a pattern is employed within many different occurrences of other patterns. Similarly, high FO hints that within occurrences of this pattern, many occurrences of other patterns are used. In summary, high FI and FO are indicators that a pattern is composed with different pattern occurrences.

B. Similarity Measures

Many different similarity measures exist for specific purposes. In PInT, a similarity measure based on the *Jaccard index* [6] is implemented. Further similarity measures will follow later in the development of the tool. The Jaccard index is a measure for similarity of two sets and is used in a variety of fields from computer vision to biology. The index is defined as the size of the intersection divided by the size of the union of two sets. We use the index to output a similarity measure for two pattern compositions. The compositions are ordered n-tuples, however we can define a set which contains all patterns occurring in a pattern composition.

$$Set(c) = \{p \in Patterns \mid p\, occurs\, in\, c\}.$$

Using this transformation, we can define the Jaccard index for two compositions:

$$J(c_1, c_2) = \frac{|\, Set(c_1) \cap Set(c_2)\,|}{|\, Set(c_1) \cup Set(c_2)\,|}.$$

Notice that the Jaccard index does not take the ordering of the two compositions into account. It is instead a measure for how similar two compositions are in terms of what patterns occur in these compositions. This Jaccard-based similarity measure is useful in order to identify compositions of patterns which consists of mainly the same patterns, although possibly in different ordering. If there are many compositions with almost identical patterns, the pattern definitions should possibly be reviewed.

V. Pattern Instrumentation Tool Based on Clang Compiler Front End

PInT enables users to analyze instrumented code regions where parallel patterns are implemented in a source code. Code regions are instrumented by framing them with special function calls provided by a PInT header file in the source code. The beginning of a pattern region is indicated by a `Pattern_Begin` call, where a string literal containing design space, pattern name and an identifier is passed as a parameter (cf. Listing 2, line 1). A pattern region is closed by the user with a `Pattern_End` call (cf. Listing 2, line 5). A string literal containing the region identifier is passed to the function as parameter. PInT currently uses pattern definitions by Mattson et al. (cf. Section III) but is designed to be extendable to a large range of pattern definitions.

PInT relies on functionality from the Clang compiler front end to lex and parse the source code input with instrumented pattern code regions and perform static analysis on the code. The compiler front end constructs the AST from the source code, from which PInT then extracts the instrumentation

```
Pattern_Begin("SupportingStructure                    1
    LoopParallelism MainParLoop");
#pragma omp for                                       2
for (int i = 0; i < MAX_ITERATIONS; i++)              3
    subMethod();                                      4
Pattern_End("MainParLoop");                           5
```

Listing 2. Example instrumentation of an OpenMP parallel for loop in a C++ code. For later examples, we will assume that `subMethod` contains two more implementations of patterns: one implements the `Barrier` pattern, the other `Recursion`.

```
./PInT ../path-to-code/example.cpp --extra-
    args=-fopenmp
```

Listing 3. Applying PInT to a source code and passing additional parameters.

information. This information would be preferably given by comments or pragmas in the code. However, when parsing the source code and constructing the AST, Clang does not keep the comment nodes in the place where they occurred in the code. Function calls remain in the AST at the exact position they appear in the source code. Hence, the current PInT implementation relies on function calls for instrumentation.

Launching PInT on a source code is syntactically very similar to a call to the Clang compiler (cf. example call in Listing 3). One or more source files are passed as parameters, in the example "`example.cpp`". Arguments added after "`--extra-args=`" are directly passed to the Clang front end. Further, PInT searches for a compilation database (`compilation_database.json`) in the source directory. The compilation database contains information on the compiler flags used to compile the source files when building the application. It can be automatically generated by build systems such as Cmake or hand-written. If no compilation database is found, all compiler flags have to be passed using "`--extra-args=`".

A. Extracting Information from the AST Using LibTooling

LibTooling is a library to support writing standalone Clang tools. We use LibTooling to implement PInT. Clang tools can run front end actions on the code, e.g., operate on the AST or performing syntax checks. The PInT functionality that extracts the necessary information from the source code is implemented as a `FrontEndAction` and executed using `ClangTool::run()`. To not lose information about the patterns, Clang may not inline the instrumentation functions. Therefore, we pass the `--no-inlining` flag to the front end action using an `ArgumentAdjuster`. Clang LibTooling provides visitor classes for AST traversal such as `RecursiveASTVisitor`. We adapt the visitor's behavior by deriving from the Clang class and by overriding the methods `VisitFunctionDecl` and `VisitCallExpr`.

If the PInT visitor reaches a call expression, a distinction between pattern instrumentation calls and calls to a sub-method is made based on the function name. The string literal passed as parameter for instrumentation calls is further processed

using Clang `ASTMatcher` classes and C++ regular expression functionality provided by the standard library. Passing the pattern information as a string parameter has two advantages: first, the implementation to extract this information from the AST using AST matchers is convenient. Second, if changes to the input format are required later, e.g., due to changes in the pattern definition, the regular expression can easily be adapted to these changes. If a new pattern code region is opened, a reference to the `PatternCodeRegion` object (cf. Section V-B for explanations on the internal representation) is stored on top of a stack. When the code region is closed later on, the tool checks by the unique string identifier if the code regions are closed in the opposite order they have been opened, i.e., in accordance to the stack structure. In the visitor functions, additional information, e.g., information required for the calculation of metrics (cf. Sections V-C and IV), can be extracted using the `SourceManager` class.

When calls to sub-methods appear in the source code, PInT has to consider the function body of the callee. The source code of the callee (or any function called by the callee) can contain patterns and, therefore, has to be considered when the structure is analyzed. The AST, however, is cut off at this place: a `CallExpr` object indicates that another method is called but information about the callee's function body is not revealed at this point. PInT requests the callee's declaration from the `CallExpr` object to obtain information about the callee's body. We calculate a hash value from this function declaration using Clang's hash function for AST nodes which is stable across different runs of the compiler. Then, a list of known function declarations is searched for this hash value to link call expressions to function declarations (and function declarations to function bodies). A reference to the function declaration is kept in the place of the call expression.

Upon visiting a function declaration during AST traversal, PInT uses the hash function to calculate the unique hash for this declaration. Again, the hash value is searched for in a list of function declarations. This step links function declarations to function definitions, i.e., function bodies. Together with the link between call expressions and function declarations, we can therefore close the link between call expressions and function bodies which is necessary for static analysis of the pattern composition across function bodies and even compilation units.

When traversing through a function body, a reference to the body's function declaration is kept by the visitor object. If function calls and patterns occur in a function body, they can then be correctly registered as the function's descendants. When the visitor reaches a function call, it calculates the hash value in the same way. If an entry with this hash exists, the surrounding pattern is registered as a parent of the callee. If the entry is not contained in a pattern code region, the caller is registered as parent of the callee. Vice versa, the callee is registered as child of the calling method or the surrounding pattern. If no entry with this hash value exists, a new entry is created and the links are registered with this new entry.

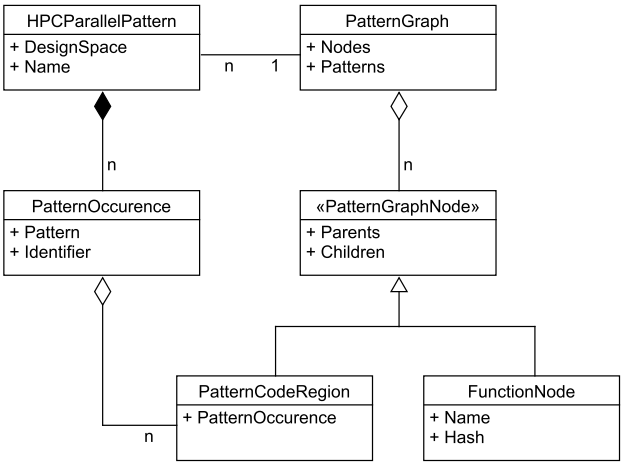

Fig. 2. PInT's internal data structure visualized using an UML class diagram. The pattern graph holds information about all nodes and patterns. A (unique) pattern can occur one or multiple times. Each occurrence consists of code regions where this pattern is implemented. These code regions — together with the functions — are the nodes of the pattern graph.

B. Internal Representation

The information extracted from the Clang AST is converted into PInT's own internal representation during the AST traversal. A UML diagram of the data structure is shown in Figure 2. Later stages of the tool, i.e., statistics and metrics, can identify the nesting and composition structure of the patterns in the analyzed source code without having to extract the information for each metric separately.

PInT's representation of the pattern nesting structure is graph-based. An example for a pattern graph can be found in Figure 3. Nodes of the pattern graph inherit from the abstract `PatternGraphNode` class. Namely, these nodes represent (1) the code regions where a pattern is implemented (`PatternCodeRegion`) and (2) function nodes (`FunctionNode`). Graph nodes manage lists of parents, i.e., their surrounding patterns or calling functions. References to children are kept analogously. The internal representation is extensible with little effort. New types of nodes have to inherit from `PatternGraphNode` and can implement their own functionality.

Pattern occurrences are concrete instances of one specific pattern. `MainParLoop`, e.g., is an occurrence of the `LoopParallelism` pattern in the earlier code example (cf. Listing 2). In this case, the pattern occurrence consists of a single, coherent code region. However, a pattern occurrence can be distributed in the source code, e.g., in SPMD programs. Then, the occurrence comprises multiple code regions. Every `PatternCodeRegion` object belongs to a `PatternOccurrence` object in PInT The tool detects that multiple code regions belong to a single pattern occurrence by comparing the identifiers. In PInT, every pattern occurrence holds a reference to its pattern object. `HPCParallelPattern` objects contain information about the design space and name of a pattern.

Function nodes are mainly kept as connecting elements

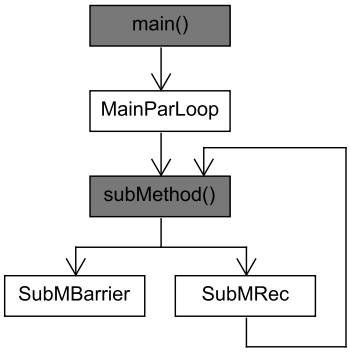

Fig. 3. Pattern graph generated for the example from Listing 2. We assume that the sub-method contains more patterns and is recursive. Function nodes are colored gray, pattern code regions are colored white.

between two function bodies, and compilation units in particular (cf. Section V-A for implementation details). Consider `subMethod` in the example from Listing 2. All patterns and function calls occurring as descendants of the callee body are also descendants of the surrounding pattern or function. Thus, if `subMethod` has a code region which implements the `Barrier` pattern (cf. Figure 3), the `Barrier` pattern code section `SubMBarrier` is connected to `MainParLoop` indirectly via the `subMethod` function node. As explained in Section V-A, PInT can detect such cross-function or cross-compilation unit compositions of patterns with the help of function nodes. Notice that the existence of such function nodes can lead to cycles in the pattern graph.

The `PatternGraph` singleton class contains references to all `PatternGraphNode` objects. We chose to implement this class as singleton since a pattern graph corresponds to exactly one execution of the Clang front end action. Hence, only one pattern graph can exist per each execution of PInT. The main advantage of a singleton implementation is that we avoid laborious passing of references to the pattern graph between the front end action and possible user-implementations like statistics and other metrics (cf. Section V-C for the implementation of metrics).

C. Automatic Statistics and Similarity Measures

From the internal graph structure, statistics and metrics (cf. Section IV for definitions) can be deduced automatically. PInT comes with a set of pre-implemented statistics and metrics. The functionality used in these metrics is made publicly available for other implementations within helper functions which are grouped in namespaces. This API will be further extended over time, with more statistic implementations sharing commonly useful functionality. Currently, changes to the source code are required to register a statistic. In the future, we want to find a way where no changes to PInT's source code by the user will be necessary. PInT provides an abstract class `HPCPatternStatistic` from which statistic classes must inherit. This way, we ensure that all statistics implement the following set of methods:

- `Calculate`: this method is called by PInT after the pattern graph was built. This method has to calculate the statistic and prepare the data for printing using the `Print` function.
- `Print`: PInT calls this method when all statistics are calculated. The print function has to give a textual or graphical representation of statistic.
- `CSVExport`: Similar to `Print`, this function is called after all statistics have been calculated. This function has to implement a simple CSV export of the statistic.

Similarity measures are implemented as a more specific type of PInT statistics (cf. Section IV for the difference between statistics and similarity measures). Similarity measures must not only inherit from the statistic class, but also from the abstract `SimilarityMeasure` class. This class provides a generic implementation to extract sequences of patterns from the pattern graph which can be re-used by user-implemented similarity measures. Beginning from a list of given pattern, the implementation searches the pattern graph in child or parent direction, depending on a parameter option, for sequences of patterns with a specified length. Every two pattern sequences form a `SimilarityPair`. For every pair, a similarity measure is calculated.

VI. Case Studies

To evaluate PInT's functionality and usability, we carried out pattern analysis of two case studies. Ongoing and future work deals with pattern analysis of more HPC applications and will reveal insights into classification approaches of HPC codes.

A. Methodology

To investigate a large variety of patterns, we chose two case studies with vastly different complexities: First, we investigate a clearly structured code with a wide range of computational structures mapped to OpenMP parallelism. Second, we examine a large MPI code that presents further challenges in pattern identification and the calculation of the PInT metrics. For both case studies, we manually analyze pattern occurrences in the source codes following the pattern definitions from Section III. We use PInT's source code instrumentation functionality to annotate those patterns in the codes. We manually evaluate the pattern count metric and a sketch of the nesting structure and compare the results to the statistics calculated by PInT. Furthermore, we show results for the more complex metrics implemented in PInT. An overview of all collected metrics can be found in Table I.

B. Benchmarks

1) Breadth-first Search: The first case study analyzed code from the Rodinia benchmark suite [5] for OpenMP. Rodinia contains four applications and five kernels that cover the computational behavior defined by the 13 dwarfs. The codes from Rodinia are short (often < 500 LOC) and readable such that the manual analysis results could be collected in reasonable time. In this paper we will further discuss the

TABLE I
OVERVIEW OF THE OBTAINED METRICS OF BOTH CASE STUDIES.

		Breadth-first Search				LAMMPS			
		Occurrences	LOC	Fan-In	Fan-Out	Occurrences	LOC	Fan-In	Fan-Out
Finding Concurrency		Data Decomposition	25	1	1	Data Decomposition	87	0	1
Dependency Analysis	Group Task	1	60	0	1	0	—	—	—
	Order Task	1	58	1	1	0	—	—	—
	Data Sharing (explicit)	0	—	—	—	0	—	—	—
Algorithm Structure		Geometric Decomposition	23	1	1	Geometric Decomposition	85	1	2
Supporting Structures (Program)	SPMD	0	—	—	—	4	215	1	10
	Master/ Worker	0	—	—	—	0	—	—	—
	Loop Parallelism	1	21	1	1	0	—	—	—
	Fork/ Join	0	—	—	—	0	—	—	—
Supporting Structures (Data)	Shared Data	1	1	1	0	0	—	—	—
	Shared Queue	0	—	—	—	0	—	—	—
	Distributed Array	0	—	—	—	0	—	—	—
Implementation Mechanisms	UE Management	1	19	1	2	0	—	—	—
	Synchronization	1	0	1	0	4	4	1	0
	Communication	0	—	—	—	12	18	1	0
Cyclomatic Complexity		Cyclomatic Complexity: 2				Cyclomatic Complexity: 2			

analysis of the breadth-first search (BFS) from Rodinia. It features a parallel implementation of the well known graph algorithm, that traverses all the connected components in a graph. Large graphs involving millions of vertices are common in scientific and engineering applications.

2) LAMMPS: Secondly, we analyze the core part of the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) coming from the SPEC MPI benchmark suite [16]. LAMMPS is a general molecular dynamics code developed at the Sandia National Laboratory. The full LAMMPS code covers 200.000 lines of code and has a much more complex computational flow than any Rodinia benchmark. The manual analysis did therefore only cover the parts where the most important patterns are expected. We analyzed the routines that set up the N-Body data decomposition and the part where Atoms get reassigned to their new nodes, thus causing lots of data exchanges.

C. Manual Analysis

The manual analysis of the Rodinia benchmark suite resulted in annotated versions of the original codes and a sheet tracking the number of occurrences of each pattern found per code. The BFS code, which was analyzed with PInt as well, follows a geometric data decomposition and utilizes one group task and one order task pattern at the beginning of the core computation loop. In this loop each of the following patterns is used once: loop parallelization, shared data, distributed array, UE management, synchronization. From the raw data gathered from all Rodinia codes, we can already conclude simple results. For example, data parallelism has been used in 95% of the codes, and 100% used loop parallelism which is to be expected from OpenMP codes. More interestingly we could also find an occurrence of the SPMD pattern, whereas Fork/Join or Master/Worker pattern were never used. We could infer some simple metrics such as 78% of parallel loops used the distributed array pattern. This was only possible due to the definition we introduced earlier that a parallel loop has 1 or 0 distributed array patterns inside and the observation that distributed arrays have never been used outside of parallel loops. However, in general, our manual analysis did not yield any information about the nesting of occurrences. Therefore, creating further metrics of this kind would enforce a manual reevaluation of the annotated code.

When analyzing the LAMMPS code, we observed multiple uses of SPMD and distributed arrays. The nesting is hard to track down because the call tree structure is complicated and may depend on the input. The whole LAMMPS code was written with a geometric decomposition in mind. We focused our analysis on the core parallelization and communication part. This part uses a lot of SPMD patterns and well designed data structures that support the data decomposition pattern.

D. PInT Results and Metrics

1) Pattern Counts and Nesting: PInT automatically generates a CSV output that contains metrics also tracked in the manual analysis as well as further statistics (see Section IV). The counts were identical to our manually-generated results. PInT, in contrast to our manual approach, was able to construct

the whole nesting structure for BFS and for LAMMPS. When implementing new metrics, these raw nesting information can be used for calculation. Implementing new metrics does never cause the code, that should be analyzed, to be revisited. Once it is instrumented, the code can be reused for future analysis even for metrics that have not yet been thought of previously.

2) FI/FO: The FI/FO metric reflects the hierarchical approach from Mattson's design spaces. In the LAMMPS code, the data decomposition is on top of the pattern graph with no FI and 1 FO, followed by its concretization, the geometric decomposition with 1 FI and 2 FOs. The SPMD Pattern is a mid level pattern which uses multiple sub patterns resulting in 10 FOs. Communication and Synchronization are on the lowest level and have therefore 0 FOs. The results could also indicate finer levels in the hierarchy, program structure pattern have more FOs than data structure patterns, however the code base analyzed may be too small to draw such conclusions already.

3) Cyclomatic Complexity: The cyclomatic complexity was 2 for both, Rodinia BFS and LAMMPS. The pattern graph for Rodinia BFS comprises 7 nodes and 7 edges. For LAMMPS, it is 22 nodes and 22 edges. Contrary to our expectations, the cyclomatic complexity was not higher for the more complex nesting structure of LAMMPS. Hence, we will have to revisit this metric and possibly make adaptations.

4) Jaccard Index: PInT also computes the Jaccard index as defined in Section IV-B. Here, we focus on the corresponding results of the LAMMPS code since it provides interesting and complex compositions of patterns. Our Jaccard index reveals that the sequence of patterns (`Synchronization`, `SPMD`, `GeometricDecomposition`) has a similarity of 0.75 (i.e., high similarity) with the sequence (`Synchronization`, `SPMD`, `GeometricDecomposition`, `DataDecomposition`). Similarly, the sequence (`Communication`, `SPMD`, `GeometricDecomposition`) presents a Jaccard index of 0.75 with respect to (`Communication`, `SPMD`, `GeometricDecomposition`, `DataDecomposition`). As expected, when we compare sequences containing the pattern `Communication` on the one hand to sequences with `Synchronization` on the other hand, the similarity decreases. For example, PInT calculates a similarity of 0.4 for the sequences (`Communication`, `SPMD`, `GeometricDecomposition`) and (`Synchronization`, `SPMD`, `GeometricDecomposition`, `DataDecomposition`). While these are exemplary results, we are not yet able to derive any general conclusions on frequently-occurring pattern compositions from these studies. This will require pattern analysis of numerous HPC applications which is part of our ongoing work.

E. Tool Evaluation

Summarizing PInT's functionality compared to manual pattern analysis, we achieved the same results with PInT as with manual analysis, at least in all cases where the latter was possible. In addition, PInT successfully computed metrics that were not achievable with manual analysis. PInT metrics are calculated at the cost of adding instrumentation calls at the appropriate locations in the source code. We emphasize that the annotations of patterns within the code must be carried out in either the manual and tool-supported analysis. Thus, additional effort by the analyst to use PInT's source code instrumentation can be neglected. Furthermore, PInT provides a very flexible way to deal with modified or new pattern definitions (needed for future work). We just introduce new pattern definitions to the tool's source code and re-apply it to the annotated source code. Thirdly, PInT is advantageous over manual pattern analysis in terms of automatic metrics analysis. Metrics and similarity measures can be computed without re-analyzing the code manually. Especially, novel metrics can be added and calculated by PInT without touching the application's source code. Lastly, the definitions used to implement the metrics are independent from PInT itself, thus, the tool can be adjusted for many use cases with different demands.

VII. Conclusion

In the context of this work, we focus on pattern analysis of applications in HPC and develop the pattern instrumentation tool PInT to support this research. With that, we work towards a pattern-based classification of HPC codes that covers the relationship of code performance to development effort needed to reach this performance.

Basis of our pattern analysis is the definition of parallel patterns and their identification in real-world application. Here, we investigate the design patterns by Mattson et al. [13], but emphasize flexibility for future (re-)definitions of pattern sets. Moreover, we define a set of metrics and similarity measures to analyze and compare parallel patterns and their structures. They can serve as an indicator for the complexity and effort needed to implement a specific parallel pattern such as lines of code, fan-in/fan-out or the cyclomatic complexity. Further metrics such as the pattern count or similarity measures can be used to concentrate further research directions on pattern sets that revealed to be important in real codes.

To evaluate code patterns and compute metrics automatically, we provide PInT which is based on source code instrumentation and static code analysis (of patterns). PInT uses the Clang LibTooling framework to integrate pattern information into the AST. Internally, we setup a pattern graph structure that represents the composition of patterns within the code. From that, we can derive and compute previously-defined metrics automatically and provide an interface to add new metrics with low effort. We further show that PInT can be applied to two HPC applications, i.e., BFS from the Rodinia benchmark suite and LAMMPS from the SPEC MPI benchmark suite and the obtained results are consistent with a manual code analysis. Moreover, PInT can collect more profound information by leveraging static code analysis than manual analysis without requiring additional effort in the identification of the patterns.

In the future, we will investigate further real-world HPC applications to derive general conclusions on the occurrence and

composition of patterns. This will lead research in classifying HPC applications with appropriate pattern sets and capturing their effort-performance relationship. Additionally, we will implement more metrics and extend the metrics API. While the current metrics can indicate the required development effort of parallel codes, we target at tracking real development time during pattern implementation in the future. For that, we vision to extend PInT's static code analysis to include tracing capabilities during development time. To make the instrumentation process more intuitive for users, we will consider changing the instrumentation API to a macro-based API. Finally, we will continue to improve PInT's user-friendliness and stability for broad application in the HPC community. Therefore, PInT is also publicly available on Github.

REFERENCES

[1] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[2] Manuel Arenaz, Juan Touriño, and Ramon Doallo. Xark: An extensible framework for automatic recognition of computational kernels. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):32, 2008.

[3] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] Bradford L. Chamberlain. *Programming Models for Parallel Computing*, chapter Chapel. The MIT Press, 2015.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[6] Alan H Cheetham and Joseph E Hazel. Binary (presence-absence) similarity coefficients. *Journal of Paleontology*, pages 1130–1136, 1969.

[7] Jack Dongarra, Allen D Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. In *International Conference on Computational Science*, pages 53–62. Springer, 2003.

[8] Edward B Duffy, Brian A Malloy, and Stephen Schaub. Exploiting the clang ast for analysis of c++ applications. In *Proceedings of the 52nd annual ACM southeast conference*, 2014.

[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[10] Michael P Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):85–122, 1995.

[11] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, (5):510–518, 1981.

[12] R. D. Hornung and J. A. Keasler. The RAJA Poratability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

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

[14] Michael McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[15] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[16] Matthias S Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. SPEC MPI2007-an application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[17] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM, 1995.

[18] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting patterns in mpi communication traces. *37th International Conference on Parallel Processing*, 37:230–237, 9 2008.

[19] Stephen Schaub and Brian A Malloy. Comprehensive analysis of c++ applications using the libclang api. *International Society of Computers and Their Applications (ISCA)*, 2014.

[20] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[21] The Clang Team. Clang 8 documentation: libtooling. https://clang.llvm.org/docs/LibTooling.html, accessed September 3, 2018.

[22] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[23] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[24] ROSE Wikibooks. Rose compiler framework/rose tools. https://en.wikibooks.org/wiki/ROSE_Compiler_Framework/ROSE_tools, accessed September 3, 2018.

[25] Michael Joseph Wolfe and Michael Wolfe. *High performance compilers for parallel computing*, volume 102. Addison-Wesley Reading, 1996.