

A Hybrid Approach to Parallel Pattern Discovery in C++

C. Brown, V. Janjic, A. Barwell, J. Thomson
School of Computer Science
University of St Andrews
St Andrews, UK
{cmb21, vj32, adb23, j.thomson}@st-andrews.ac.uk

J.D. Garcia-Sanchez, D. Del Rio Astorga
Computer Science and Engineering Dept.
Universidad Carlos III de Madrid
Madrid, Spain
{josedaniel.garcia, david.rio}@uc3m.es

R. Castañeda Lozano, M. Cole, B. Franke
School of Informatics
University of Edinburgh
Edinburgh, UK
{rcastae, mic, bfranke}@inf.ed.ac.uk

K. MacKenzie
IOHK
kenneth.mackenzie@iohk.io

Abstract—Parallel pattern libraries offer a strong combination of abstraction and performance. However, discovering places in sequential code where parallel patterns should be introduced is still highly non-trivial, often requiring expert manual analysis and profiling. We present a hybrid discovery technique to detect instances of parallel patterns in sequential code. This employs both static and dynamic trace-based analysis, together with hotspot detection. We evaluate our pattern discovery mechanism on a number of representative benchmarks. We evaluate the performance of the resulting parallelised benchmarks on a 24-core parallel machine.

I. INTRODUCTION AND BACKGROUND

With the massive increase of scale of parallelism and heterogeneity of modern parallel hardware, programming parallel systems has now become harder than ever. Models based on *parallel patterns* have gained lots of traction recently [1, 2, 4], with significant industry players releasing their own pattern libraries. Pattern libraries offer functions for common parallel structures such as *pipeline*, *map* and *reduce*, which can be instantiated into appropriate places in otherwise sequential application code. However, exactly *where* in the code to introduce patterns and *what* pattern to introduce is challenging. In this paper we present a novel hybrid composition of both *static* and *dynamic* analyses of sequential applications to detect places where parallel patterns should be introduced. Our study involves three complementary components: *profiling-based hotspot detection*, *static pattern detection* on hotspot loops, and *trace-based pattern detection* on hotspot loops. In this way, we are able to answer both the *where should parallelism be introduced* and *what type of parallelism should be introduced* questions for an existing sequential application. We evaluate our approach and demonstrate that we are able to identify, among numerous loops in the benchmarks, those that are both computationally intensive and conform to one of the standard parallel patterns. In this paper, we restrict ourselves

to two classical parallel patterns, which we believe to be the most widely known:

- The *map* pattern models a data parallel computation, where a single computational worker f is applied to a set of independent inputs x_1, \dots, x_n . Parallelism arises from applying the worker f to input elements in parallel.
- The *reduce* pattern models a reduction operation, where a binary associative function \oplus is applied pairwise to the elements of a set of inputs, x_1, \dots, x_n . The output of the reduce becomes, $x_1 \oplus x_2 \oplus x_3 \cdots \oplus x_n$. Parallelism arises by staging the reductions in parallel.

II. DETECTING PARALLELISM CANDIDATES IN C++

Our parallelism discovery methodology is shown in Figure 1. Here, the programmer starts with a program written in C++. The hotspot analysis over the sequential code duplicates the original sequential program, instruments it with profiling code, executes the instrumented version, and records the results for use by the tracing stage. For each hotspot loop candidate, we apply two contrasting pattern-detection mechanisms to determine whether the loop is an instance of a parallel pattern. In static pattern-detection, each parallelism candidate hotspot is passed through a static analysis, leveraging the existing Pattern Analyzer Tool PPAT [3]. PPAT takes advantage of the clang lib-tooling utilities to statically analyze the code at the Abstract Syntax Tree level. It traverses the AST and collects information about the data dependencies, analyses them and introduces annotation indicating the potential parallel patterns in the form of C++ attributes.

Independently from the static analysis, candidate hotspots are also passed through our new dynamic trace-based pattern detection mechanism (described in Section III). Finally, the user checks manually that the candidate detected patterns (from either analysis) are indeed applicable. A detected pattern might not be applicable for different reasons, for example if

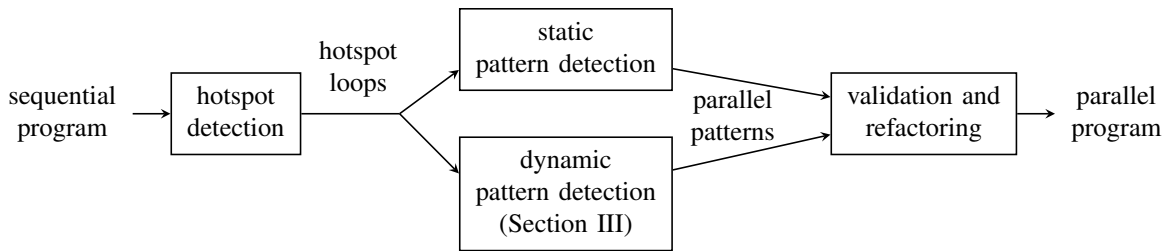


Fig. 1: Overview of our approach to parallel pattern discovery.

the input used for tracing does not uncover all dependencies across loop iterations.

As the fork in the diagram suggests (see Figure 1), in our current prototype we apply both static and dynamic approaches to all candidate loops. This recognizes that each technique could find instances of potential patterns undetected by the other. It also allows us to compare the performance of the two techniques directly. In a production system, we could decide to only apply the more computationally expensive dynamic method to candidates for which the static approach has not found an acceptable pattern.

III. DYNAMIC TRACE-BASED DETECTION OF PATTERNS IN HOT SPOTS

Our new dynamic *trace-and-match* method detects whether a hotspot loop corresponds to a parallel pattern. The method proceeds in two steps: first, it constructs a graph representation of the execution of the loop (Section III-A) by instrumenting and tracing the loop’s data flow. Then it applies pattern matching techniques to check whether the loop iterations correspond to a parallel pattern (Section III-B).

This paper explores the matching of *map* and *reduce* patterns. Generalizing the method to match more complex patterns such as *pipeline* and *stencil* is part of ongoing work.

A. Dynamic Dependency Graphs

The pattern detection method represents input loops as Dynamic Dependency Graphs (DDG). A DDG is a directed acyclic graph in which nodes correspond to executions of instructions (or groups of instructions), and arcs correspond to flow of data across them. The DDG is a natural representation of the essential computation of a program, as it abstracts away implementation aspects such as the choice of data structures or order of computation [9].

Our method constructs the DDG of a loop by instrumenting the loop at the compiler intermediate representation (IR) level and tracing its data flow as the loop is executed.

The size of DDGs grows quickly with the number of executed instructions and can make pattern matching prohibitive. To reduce this size, sets of nodes corresponding to loop body instructions from the same iteration are merged into single nodes. This simplification leads to DDGs containing two types of nodes: loop body nodes and single instruction nodes corresponding to iterator instructions such as induction variable increments.

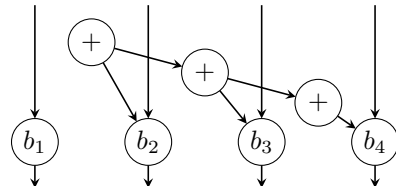
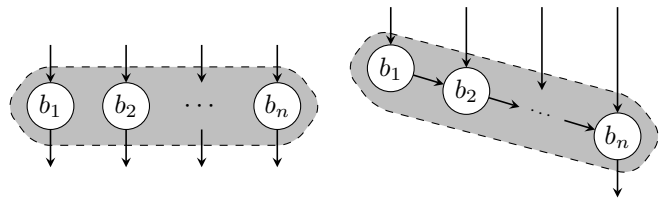


Fig. 2: DDG of a map style *for* loop iterated four times. Each b_i node corresponds to all loop body instructions in iteration i .



(a) *Map* pattern shape.

(b) *Reduce* pattern shape.

Fig. 3: Pattern shapes on DDGs.

Figure 2 shows an example DDG corresponding to an execution of four iterations of a typical map style *for* loop, where an induction variable is incremented in each iteration. The DDG includes addition nodes (+) that correspond to the increment of the induction variable and loop body nodes (b_i) that correspond to instructions executed in iteration i .

B. Matching of Parallel Patterns

The pattern detection method checks, for the simplified loop DDG, whether the loop body nodes match the shape of a *map* or *reduce* pattern. This check is formulated and performed as a graph pattern matching problem, where pattern matches are found by testing the satisfiability of pattern-specific constraints over the set of loop body nodes.

Figure 3 depicts the shape of *map* and *reduce* patterns on DDGs containing a set of loop body nodes $B = \{b_1, b_2, \dots, b_n\}$. Such a set matches a *map* pattern if and only if its nodes are disconnected:

$$b_j \text{ is not reachable from } b_i \quad \forall b_i, b_j \in B : b_i \neq b_j. \quad (1)$$

For example, the body nodes $\{b_1, b_2, b_3, b_4\}$ in Figure 2 match a *map* pattern, as none of them reach each other in the DDG.

A set of loop body nodes B matches a *reduce* pattern if and only if its nodes form a path where the outdegree of all but

the last node is equal to one:

$$\text{successors}(b_i) = \{b_{i+1}\} \quad \forall i : 1 \leq i < n. \quad (2)$$

These pattern definitions are solely based on data-flow structure, and do not take into account other criteria such as specific relations among different input data elements. This high-level, structural view simplifies and broadens the scope of pattern detection, for example allowing the detection of potential patterns across scattered data elements or different data collections. However, it also implies that our method can potentially propose invalid pattern matches that do not satisfy additional criteria besides matching the pattern’s data-flow structure. Examples are pattern matches containing non-commutative system calls and *reduce* matches where the loop body is not an associative reduction operator. Reducing the number of invalid pattern matches is part of future work.

IV. RESULTS

A. Parallel Pattern Detection Performance

This section studies the pattern detection capabilities of the static PPAT tool and the dynamic trace-and-match method described in Section III and identifies strengths, limitations, and opportunities based on the results.

Table I compares the *map* and *reduce* patterns detected by both approaches with those detected by a manual expert analysis for each of the loops selected by hotspot analysis. For the dynamic approach, we have traced each benchmark using small input data sets, to reduce the size of the resulting dependency graphs while executing a sufficient number of iterations of each of the targeted loops.

Table II summarizes the performance of each pattern detection method using the expert analysis results as the ground truth. The performance of each method is summarized by three measures: accuracy (percentage of loops classified correctly, whether containing patterns or not), precision (percentage of actual patterns among all detected patterns), and recall (percentage of detected patterns among all actual patterns).

The results show that, across the selected benchmarks, both pattern detection methods have a substantial overall accuracy (79%). As can be expected from their fundamental properties, the methods present complementary strengths: while the static pattern detector is sound (100% precision), the dynamic pattern detector trades some precision (−18%) for higher overall recall (+15%).

The precision results confirm the soundness of the static pattern detector, and show that the dynamic pattern detector has a lower yet substantial overall precision (82%), only affected by four loops where patterns are falsely detected. Loop 134 (*Ant Colony*) is a max reduction implemented with a conditional assignment. The dynamic detector abstracts away assignments that move but do not transform data, and falsely detects the resulting instructions as a *map* pattern. Loop 373 (*Black-Scholes*) contains non-commutative system calls (`fscanf`) which preclude parallelization in practice.

benchmark	loop ID	exec.	map pattern			reduce pattern		
			static	dynam.	expert	static	dynam.	expert
Ant Colony	258	99%	○	○	○	○	●	○
	259	84%	○	●	●	○	○	○
	120	84%	○	○	○	○	○	○
	123	51%	○	○	●	●	○	●
	211	15%	●	●	●	○	○	○
	90	14%	●	●	●	○	○	○
	212	12%	●	●	●	○	○	○
	134	8%	○	●	○	○	○	●
Black-Scholes	285	40%	○	●	●	○	○	○
	290	24%	●	●	●	○	○	○
	373	3%	○	●	○	○	○	○
Convolution	158	100%	●	●	●	○	○	○
	112	99%	●	●	●	○	○	○
	111	99%	●	●	●	○	○	○
	119	94%	○	○	●	●	●	●
	120	45%	○	○	●	●	●	●
Mandelbrot	60	96%	●	●	●	○	○	○
	62	96%	●	●	●	○	○	○
	33	90%	○	○	○	○	○	○
Transfil	54	99%	○	○	●	○	○	○
	233	90%	○	●	●	○	○	○
	78	60%	○	○	●	○	○	○
	94	60%	○	○	○	○	○	○
	157	59%	○	○	○	○	○	○
	183	45%	○	●	●	○	○	○
	90	45%	○	●	●	○	○	○
	120	39%	○	○	○	○	●	○
	217	5%	○	○	●	●	●	●
464	2%	●	●	●	○	○	○	

TABLE I: Patterns detected by the static and dynamic approaches and by expert analysis in the loops selected by hotspot analysis. The loops within each benchmark are sorted in descending order of execution time (**exec.**). Filled circles (●) indicate that the corresponding pattern is detected, empty circles (○) indicate the opposite.

pattern	accuracy		precision		recall	
	static	dynamic	static	dynamic	static	dynamic
<i>map</i>	62%	72%	100%	88%	48%	71%
<i>reduce</i>	97%	86%	100%	60%	80%	60%
total	79%	79%	100%	82%	54%	69%

TABLE II: Pattern detection performance.

Loops 258 (*Ant Colony*) and 120 (*Transfil*) are structured as *reduce* patterns but their body is not associative.

The static pattern detector misses a total of twelve *map* and *reduce* patterns, due to its conservativeness in the face of potential side-effects, pointer aliasing, and unknown information at compile time. The dynamic pattern detector misses eight actual patterns. In Loops 123 (*Ant Colony*), 119 and 120 (*Convolution*), and 217 (*Transfil*), the Dynamic Dependency Graph simplification described in Section III-A precludes detecting *partial patterns* that only cover some of the instructions in the loop body. In Loop 134 (*Ant Colony*), the dynamic approach misses the *reduce* pattern due to the assignment instruction limitation described above. Finally, in Loops 54 and 78 (*Transfil*), the dynamic approach fails to

find *map* patterns due to loop-carried dependencies induced by output buffering, pseudo-random number generation, and intermediate result caching. While these dependencies could be in principle removed by a manual expert, the transformations required are beyond the scope of our detection methods.

Stencil pattern case study. Regarding the Convolution benchmark using the static pattern analyzer, in addition to the map patterns detected by the dynamic and static approaches, it also detects an instance of a stencil pattern in Loop 112. Listing 1 shows an excerpt of the loop annotated by the tool. As observed, the static analyzer detected the loop as a stencil pattern since each output item can be transformed independently and depend on multiple input data items. Furthermore, the tool also annotated the input/output parameters of the pattern in order to ease the code transformation by a developer.

Listing 1: Annotated stencil pattern in Convolution benchmark

```

1  [[rph::stencil, rph::in(vstep,dim,hstep,mask_index
   ,mask_dim,index,in_image,mask), rph::out(
   out_image)]]
2  for(int x = 0; x < dim; x++)
3  [rph::map, rph::in(x,dim,hstep,mask_index,
   mask_dim,index,in_image,mask), rph::out(
   out_image)]]
4  for(int y = 0; y < dim; y++) {
5  left = (x < vstep) ? 0 : (x - vstep);
6  right = ((x + vstep - 1) >= dim) ? dim - 1 : (x
   + vstep - 1);
7  top = (y < hstep) ? 0 : (y - hstep);
8  bottom = ((y + hstep - 1) >= dim) ? dim - 1 : (y
   + hstep - 1);
9  sumFX = 0;
10
11  [[rph::reduce, rph::reduce(sumFX)]]
12  for(int j = left; j <= right; j++)
13  [[rph::reduce, rph::reduce(sumFX)]]
14  for(int k = top; k <= bottom; k++) {
15  mask_index = (k - (y - hstep)) * mask_dim + (j
   - (x - vstep));
16  index = k * dim + j;
17  sumFX += ((float)in_image[index] * mask[
   mask_index]);
18  }
19
20  sumFX += 0.5f;
21
22  out_image[y*dim + x] = (unsigned short) sumFX;
23  }

```

B. Parallel Pattern Runtime Performance

For each benchmark, we have taken the loop reported as having the highest execution time by the hotspot analysis and also reported as a *map* pattern by at least one of the pattern detectors. Our goal is to verify whether the execution time of the identified loops translates into highly effective speedups.

All of our execution experiments are conducted on a server with Intel Xeon E5-2690 CPU with 28 cores, running at 2.6 GHz with 256 GB of RAM, with Scientific Linux 6.2.

The performance results for the Mandelbrot, Convolution, Ant Colony, and Black-Scholes benchmarks are shown in Figure 4. These benchmarks show significant speedups, with

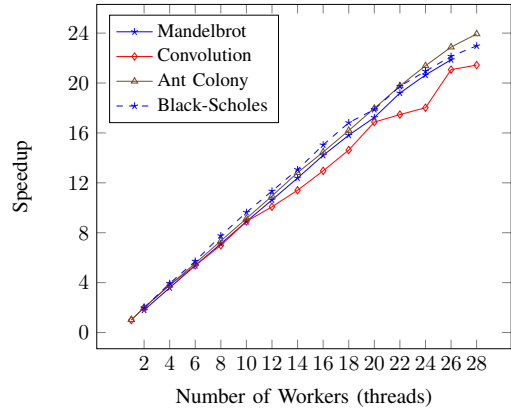


Fig. 4: Speedup results for Mandelbrot, Convolution, Ant Colony, and Black-Scholes where the hottest *map* pattern is parallelized.

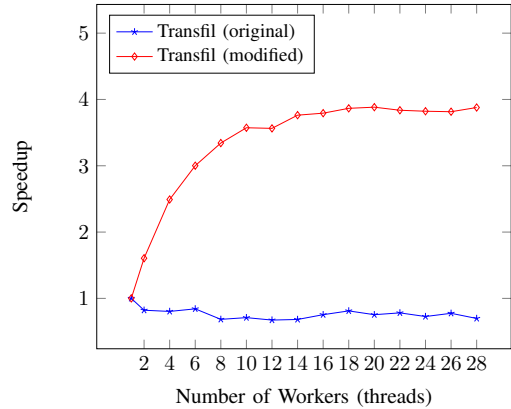


Fig. 5: Speedup results for Transfil (original and modified) where the hottest *map* pattern (Loop 233) is parallelized.

Mandelbrot (Loop 60), Ant Colony (Loop 259) and Black-Scholes (Loop 285) all reaching speedups of around 22 on 28 cores. Convolution starts to plateau at around 18 cores, with a speedup of 16, but steadily increases up to a speedup of around 21 on 28 cores.

The performance results for the Transfil benchmark (where Loop 233 is parallelized) are shown in Figure 5 (*Transfil original*). As can be seen, Transfil differs radically from the other benchmarks in that it does not show any speedup, even though Loop 233 takes 90% of its total execution time. The key bottleneck in this loop is the intensive use of a single random number generator that imposes a serialization among the different threads accessing it. Detecting such performance bottlenecks is currently outside the scope of our discovery technique.

The effect of using a single random number generator can be noticed by comparing the speedup of the original Transfil benchmark (*Transfil original* in Figure 5) with the speedup of a modified version (*Transfil modified* in the same figure) where the most frequent random generation calls are replaced by numeric constants. In the latter case, reasonable speedup

is obtained, suggesting that each thread should use its own random number generator to obtain an effective parallelization.

V. RELATED WORK

There is a significant amount of previous work on detecting parallelism by tracing data dependencies dynamically [5, 6, 7, 8, 10, 11]. Among these approaches, those by Rul et al. [10] and Huda et al. [6] are most closely related to our trace-and-match method from Section III. Both approaches aim at detecting *map* and *pipeline* patterns (the latter also targets *reduce* patterns) on different program representations based on profiled data- and control-flow information. Both Rul et al. [10] and Huda et al. [6] use graphs as the main abstraction in their representations, but, unlike our approach, they aggregate data and control-flow information about different executions of the same program region into single nodes. While this yields a more compact representation, we believe that conserving specific information about different executions of a program region is needed to detect patterns that act differently depending on input data and other dynamic properties, which we aim to explore in future work. Rul et al. [10] track dependencies at the data structure level, while Huda et al. [6] use *read-compute-write* units. In contrast, our method admits varying levels of granularity based on static program structure. We plan to explore adapting the granularity during pattern detection to balance the trade-off between scalability and detection performance. As discussed in Section IV-A, a finer granularity would increase significantly the scope of detectable patterns.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new hybrid-approach for parallel pattern discovery in C++. This novel discovery approach employs three kinds of analysis to find instances of parallel patterns, including *dynamic hotspot detection* that profiles the execution with monotonic runtime timing, *static analysis* using the PPAT tool, and *dynamic trace-based methods*, which analyse the execution data flow to determine instances of patterns. Driven by candidates suggested by hotspot analysis, both pattern-detection techniques reported instances of parallel patterns in each of the benchmarks reported. We confirmed that instances in the source code could indeed be transformed into instances of parallel patterns with significant speedups of up to 22 on a 28-core machine. In the future, we plan to extend our hybrid pattern discovery technique further, in order to support additional patterns, such as divide and conquer and to other languages as well. We also plan to evaluate our hybrid discovery technique on larger use-cases. Finally, we intend to refine the technique to take into account different types of scalability bottlenecks.

ACKNOWLEDGEMENTS

This work was supported by the EU Horizon 2020 project, TeamPlay, grant number 779882, and UK EPSRC Discovery, grant number EP/P020631/1.

REFERENCES

- [1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons, Ltd, 2017.
- [2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI’04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [3] David del Rio Astorga, Manuel F Dolz, Luis Miguel Snchez, J Daniel Garca, Marco Danelutto, and Massimo Torquati. Finding parallel patterns through static analysis in C++ applications. *The International Journal of High Performance Computing Applications*, 32(6):779–788, 2018.
- [4] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP ’10, pages 5–14, New York, NY, USA, 2010. ACM.
- [5] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. Data-flow/dependence profiling for structured transformations. In *Principles and Practice of Parallel Programming*, pages 173–185, New York, NY, USA, 2019. ACM.
- [6] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. Automatic parallel pattern detection in the algorithm structure design space. In *International Parallel and Distributed Processing Symposium*, pages 43–52, Washington, DC, USA, May 2016. IEEE.
- [7] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 437–448, Washington, DC, USA, 2012. IEEE.
- [8] Zhen Li, Ali Jannesari, and Felix Wolf. Discovery of potential parallelism in sequential programs. In *Parallel Processing*, pages 1004–1013, Washington, DC, USA, 2013. IEEE.
- [9] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
- [10] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531–551, September 2010.
- [11] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Parallel Architectures and Compilation Techniques*, PACT ’10, pages 377–388, New York, NY, USA, 2010. ACM.