del Rio Astorga, D., Dolz, M. F., Sánchez, L. M., García, J. D., Danelutto, M., & Torquati, M. (2018). Finding parallel patterns through static analysis in C++ applications. The International Journal of High Performance Computing Applications, 32(6), 779–788.

DOI: 10.1177/1094342017695639

Finding Parallel Patterns through Static Analysis in C++ Applications

David del Rio Astorga[1], Manuel F. Dolz[1], Luis Miguel Sanchez[1], J. Daniel García[1], Marco Danelutto[2] and Massimo Torquati[2]

[1] *Universidad Carlos III de Madrid, 28911–Leganés, Spain*
[2] *Department of Computer Science, University of Pisa, 56127–Pisa, Italy.*

Since free performance lunch of processors is over, parallelism has become the new trend in hardware and architecture design. However, parallel resources deployed in data centers are underused in many cases, given that sequential programming is still deeply rooted in current software development. To face this problem, new methodologies and techniques for parallel programming have been progressively developed. For instance, parallel frameworks offering programming patterns, allow expressing concurrency in applications in order to better exploit parallel hardware. Nevertheless, it remains a large portion of production software, coming from a broad range of scientific and industrial areas, that is still developed in sequential. Taking into account that these software modules contain thousands, or even millions, of lines of code, the effort needed to identify parallel regions is extremely high. To pave the way in this area, this paper presents Parallel Pattern Analyzer Tool (PPAT), a software component that aids discovering and annotating parallel patterns in source codes. This tool simplifies the transformation of sequential source code to parallel. Specifically, we provide support for identifying the map, farm and pipeline parallel patterns and evaluate the quality of the detection on a set of different C++ applications.
[1]

1 Introduction

Although most of the current computing hardware, such as multi-/many-core processors, GPUs or co-processors, has been envisioned for parallel computing, much of the prevailing production software is still sequential [20]. In other words, a large portion of the computing resources provided by modern architectures is basically underused. In order to exploit these resources, it becomes necessary to refactor sequential software into parallel. To tackle this issue, several solutions in the area, such as parallel programming frameworks, have been developed to efficiently exploit parallel computing architectures [18]. Indeed, there can be found multiple parallel programming frameworks that benefit from shared memory multi-core architectures, such as OpenMP, Cilk or Intel TBB; distributed platforms, such as MPI or Hadoop; and some others especially tailored for accelerators, as e.g., OpenCL and CUDA. Nevertheless, only a small portion of production software is using these frameworks.

[1]This is a post-print of an article published in The International Journal of High Performance Computing Applications. The final authenticated version is available online at: https://doi.org/10.1177/1094342017695639

Practical use cases in this sense are sequential data-intensive applications, broadly encountered in production scientific and industrial areas. Just a simple analysis of their codes would reveal that a vast majority of algorithms and methods could be refactored into parallel patterns [10]. A solution to parallelize these codes is to manually translate into parallel code, however this task results, in most cases, cumbersome and very complex for large applications. Another solution is to use refactoring tools, applications that advice developers or even semi-automatically transform sequential code into parallel [3]. Although source codes transformed using these techniques do not often get the best performance, they aid in reducing necessary refactoring time [11].

Unfortunately, refactoring tools found are still premature, not yet being fully adopted by development centers. In fact, many of them are human-supervised, being the developer the only responsible for providing specific sections of the code to be refactored. Although these tools relieve the burden of the source-to-source transformation, this process still remains semi-automatic. Key components for turning this process from semi- to full-automatic are parallel pattern detection tools. This fact motivates the goal of our paper: we present a tool capable of analyzing sequential C++ code statically in order to detect and annotate parallel patterns. This detection is performed using the compiler infrastructure and avoids costs related to profiling techniques. In general, this paper extends the work presented in [5] and makes the following contributions:

- We develop a Parallel Pattern Analyzer Tool (PPAT) to analyze, detect and annotate parallel patterns in C++ source codes.
- We implement a set of parallel pattern detection modules: `Pipeline`, `Farm` and `Map`.
- We perform an experimental evaluation of PPAT using a set of well-known sequential benchmark suites and a real video processing use-case.
- We demonstrate that, due to the modularity of PPAT, the tool can be easily extended to support other kind of parallel patterns.

This paper is structured as follows: Section 2 reviews the state-of-the-art about existing parallel refactoring and parallel pattern detection tools. Section 3 describes the parallel patterns supported in the detection process. Section 4 explains in detail the parallel pattern analyzer tool along with the `Pipeline`, `Farm` and `Map` detection modules. Section 5 addresses the experimental analysis and evaluation of the tool. Finally, Section 6 enumerates some concluding remarks and future works.

2 Related work

We find several research works in the state-of-the-art that address the detection of potential parallel codes and refactoring processes. However, the detection task is not simple and the tools developed to identify parallel patterns in sequential codes are strongly tied to the programming language requiring profiling techniques. For example, the approach developed by Sean Rul et al. [17] leverages LLVM to instrument loops in the sequential code and performs an LLVM-IR profiling analysis to decide whether a loop is a pipeline or not. After that, it transforms the code to produce a parallel source code. However, this tool presents some shortcomings: it needs to execute the target application several times and profile it. Also, it is tied to the C programming language. Our approach addresses these limitations by means of performing a static analysis without requiring any previous execution or profiling techniques, and supports both C/C++ programming languages.

Other contributions, such as the work by Molitorisz et al. [12], detect statically potential parallel patterns, nevertheless they do not check for dependencies, so the correctness of the resulting parallel application cannot be guaranteed. Instead, they need a subsequent execution to discover potential data races and dependencies. Our work addresses these issues by checking memory accesses at compile-time. Likewise, PoCC [13], a flexible source-to-source compiler using polyhedral compilation, is able to detect and parallelize loops, however it does not take into account high-level parallel patterns. On the other hand, we also find tools that detect parallel patterns using only profiling techniques. For example, DiscoPoP leverages dependency graphs in order to detect parallel patterns [8]. Nevertheless, this tool has an important drawback: the profiling techniques have a non-negligible execution time and memory usage. A similar approach, presented by Tournavitis et al. [21], detects and transforms sequential code into parallel introducing parallel pipeline patterns. Alternatively, FreshBreeze [7], a dataflow-based execution and programming model, leverages static loop detection techniques that analyze dependencies and transform parallelizable loops using a task tree-structured memory model. It is important to remark that, approaches based on static analysis are not very extended in the area, since analyzing data dependencies becomes much more complex at compile time.

Orthogonally, some works take advantage of functional languages. For instance, István Bozó et al. [2] develop a tool that detects parallel patterns in applications written in Erlang. Compared to other languages, Erlang features make the detection process much simpler. Nonetheless, the tool requires profiling techniques in order to decide which pattern suits best for a concrete problem.

3 Parallel patterns

This section provides a brief overview about the parallel patterns supported by the PPAT tool, i.e., `Pipeline`, `Farm` and `Map` patterns [9].

3.1 `Pipeline` parallel pattern

The `Pipeline` stream parallel pattern consists of a chain of processing entities arranged in the way that the output of each entity is the input of the next one. Considering a pipeline of n stages, the i-th stage computes the function $f_i : \alpha \rightarrow \beta$. Then, for each item x appearing in the input stream, the functions of the pipeline stages $(f_1, \ldots, f_n)$ are applied consecutively to produce elements in the output stream, i.e., $f_n(f_{n\ \ 1}(\ldots f_1(x)\ldots))$. The main requirement of this pattern is that the functions $f_1, \ldots, f_n$ related to the stages must be pure. That is, *i)* the function must always generate the same result given the same input argument, and *ii)* the function result must not depend on any hidden information or global state that may change during the execution. The parallelization approach of the pipeline pattern can be performed at stage-level. Assuming that the items of the input stream are $\ldots, x_{i+1}, x_i, x_{i\ \ 1}, \ldots$, the computation of the stage f_j over the partial result of x_i, happens in parallel with the computation of f_{j+k} over the partial result of $x_{i\ \ k}$. Furthermore, in order to remove possible bottlenecks in certain pipeline stages, it is possible to parallelize a stage by introducing a `Farm` pattern.

3.2 `Farm` parallel pattern

The `Farm` stream parallel pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream. Thus, for each item x_i in the stream, an item $f(x_i)$ will be delivered onto the output stream. Computations relative to different stream items are completely independent and can be processed in parallel without side effects, i.e., the function f must be pure. The parallel implementation of the farm pattern uses a set of entities $\{W_1, W_2, \ldots, W_N\}$ namely *workers* that compute the function f in parallel on different input tasks. However, computation of $f(x_i)$ can only start when x_i is available in the input stream. Therefore, assuming items appear in the input stream with an interarrival time T_a and the computation of f takes T_f, then at most $\frac{T_f}{T_a}$ computations will take place in parallel, at any time. This amount of computations potentially happening in parallel may be limited by the parallelism degree of the pattern.

3.3 `Map` parallel pattern

The `Map` data parallel pattern computes the function $f : \alpha \rightarrow \beta$ over all the data items of the input data collection, where the input and output elements have type α and β, respectively. The output result is a collection of data items $y_1, y_2, \ldots, y_N$ where $y_i = f(x_i)$ for each $i = 1, 2, \ldots, N$, and x_i is the corresponding element of the input collection. The only requirement of the `Map` pattern is that the function f must be a pure function. Since each data item in the input collection is independent from the other items, all the elements can be computed in parallel. However, the maximum number of data items that can be computed in parallel depends on the parallelism degree of the pattern itself. Although both `Farm` and `Map` pattern seem to be similar, the difference lies in the fact that the `Farm` patterns works on a stream input data, while the `Map` patterns receives a data collection of a fixed number of items that are partitioned among the available computing resources. However, the parallelization approach of the `Map` pattern is quite similar to that for the `Farm` pattern implementation.

4 Parallel pattern detection

In the following sections, we describe the main contribution of this paper, a tool that is able to detect and annotate parallel patterns in sequential C/C++ source codes.

4.1 RePhrase attributes for parallel patterns

In order to annotate parallel patterns using custom C++11 attributes [6], we have extended the set of attributes defined for the projects Repara [15] and RePhrase [16]. Table 1 describes the attributes used for annotating the `Pipeline`, `Farm` and `Map` parallel patterns.

Thanks to these attributes, a refactorization tool would have enough information to transform annotated code regions into parallel. Also, they allow to split the detection and transformation processes, so different tools can be used in these stages. Note that this work only covers the detection phase, but leaves the refactorization as part of the future work.

Table 1: RePhrase attributes.

RePhrase Attribute	Description
`rph::pipeline`	It identifies a pipeline pattern.
`rph::stream`	This attribute identifies the data streams used across stages of a `rph::pipeline`.
`rph::stage`	It identifies a code section as a pipeline stage.
`rph::plid`	It is associated to `rph::stage` and includes the pipeline ID.
`rph::farm`	This attribute specifies the farm pattern.
`rph::map`	It determines the map pattern.
`rph::in`	This attribute references the input variables of a pattern.
`rph::out`	It references the pattern output variables.

4.2 Parallel Pattern Analyzer Tool

In this section, we describe the Parallel Pattern Analyzer Tool (PPAT). This tool takes advantage of the Clang library to generate the Abstract Syntax Tree (AST). Then, it walks through it in order to collect relevant information about the source code and identify parallel patterns.

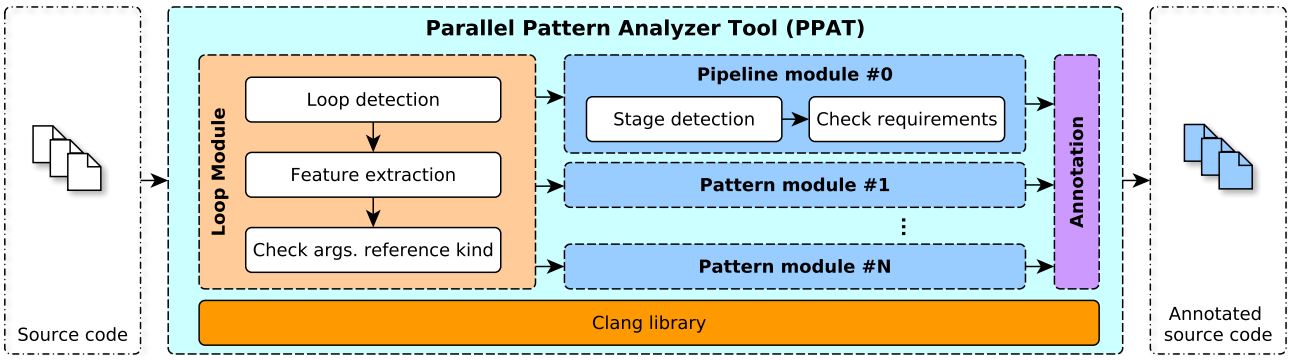

Figure 1: Workflow diagram of PPAT.

Figure 1 depicts the general workflow diagram of PPAT. First, the tool receives the sequential source code files that should be analyzed. Next, the following steps are executed:

1. *Loop detection.* This step detects potential loops that can be transformed into parallel patterns. Basically, it iterates the AST, extracts loop-related subtrees and gathers information of different AST nodes (e.g. variables, function calls, conditional statements). On the other hand, it collects information about the functions implemented in the source code.
2. *Feature extraction.* This step leverages the structures collected in the previous step in order to extract specific features about variable declarations, references, function calls, inner loops, memory accesses, operations, etc. Next, for each statement encountered, it stores information about location on the original code, variable and functions name, reference kind (Write or Read) and global storage references.
3. *Check arguments reference kind.* The last step checks whether the kinds of variable references passed as arguments in functions can be determined or not. In some cases, it is not possible to know statically if the kind of arguments passed by reference are read or written. When this occurs, the tool performs the following actions:

   (a) If the function code is available or the function is implemented in the user code, it is possible to check the set of arguments and assign the right variable kind (Write or Read). If an argument is not modified, it is considered as Read. In contrast, if there exist write accesses to the variable, Write is assigned as the variable kind. Alternatively, if there is a read-after-write (RAW) dependency on a variable, the kind is set to Write/Read, since the argument can generate potential feedbacks among iterations.

   (b) On the contrary, if the function cannot be accessed, it is not possible to check the actual variable kind. Thus, the tool takes a conservative decision: it sets the arguments kinds always to Write/Read. Despite of this, it inserts the function name and parameter kinds into a dictionary file in order to improve the

detection process in future analyses. Afterwards, the user can eventually modify this dictionary to set the right parameter kinds for these specific functions.

Next, marked loops are passed to the different pattern analyzer modules. Finally, the parallel patterns found are forwarded to the annotation module responsible for inserting the above-described RePhrase attributes in the corresponding loops. In the the following sections we describe the parallel pattern analyzer modules that are currently supported by the PPAT.

4.3 `Pipeline` detection module

In this section we detail the internal workings of the `Pipeline` detection module. As defined in previous section, this pattern defines a code that can be split into stages and run in parallel by different threads, so that the output of a stage is the input of the next one. The requirements for a `Pipeline` to be detected are the following:

- *No global variables can be modified*. In other words, there should not exist instructions that write on global variables.
- *No feedback*. This requirement controls that no feedback exists among iterations of the loop. To do so, it checks if there are no variables written before they are read, i.e., there are no RAW dependencies.
- *Multiple stages*. The last requirement checks whether the potential pipeline can be split in, at least, two stages. Otherwise, the loop cannot be treated as a parallel pipeline and PPAT discards it right away.
  The current strategy to split a loop into stages is to create a new stage each time a function call or an inner loop is found in the main loop. Afterwards, for each stage encountered, PPAT checks whether the stage is fed with, at least, a previous stage output. If this is not the case, the complete stage is merged with the previous one until all stages comply with the requirement. Note that this strategy assumes that each stage has a substantial amount of work, however if function calls or nested loops inside a stage have negligible workloads, the tool may identify a `Pipeline` with unbalanced stages. In the future, we plan to improve this strategy by adopting more advanced techniques capable of assessing the computational load of the `Pipeline` stages. Finally, PPAT checks for the presence of other parallel pattern in the stages codes. If so, the corresponding stages are annotated as well.

4.4 `Farm` detection module

This pattern defines a loop that can be run in parallel by different threads over a data stream. In this case, the code analyzed should be equivalent to a pure function, i.e., there should not exist data dependencies producing potential side effects. This requirement is controlled using the following constraints:

- *No RAW dependencies*. There should not exist RAW dependencies of variables used within iterations of the loop.
- *No global variables are modified*. There should not exist instructions that modify global variables in the loop.
- *No break statements*. There should not exist break statements (i.e., continue, break or return) in the loop, as they cannot be parallelized. However, they may be placed in inner scopes of the main loop.

4.5 `Map` detection module

This section describes the implementation of the `Map` detection module within PPAT. Basically, the `Map` pattern represents a parallel code executing a pure function that is responsible for generating the output elements. Note that in this case the total number of input elements is known in advance. To ensure these requirements, the `Map` pattern adds the following two constrains over the requirements of the `Farm` pattern:

- *Known number of input elements*: The input data must be declared and allocated before the definition of the analyzed loop.
- *At least one output*: According to the previous `Map` definition, the pure function of the `Map` pattern processes an input to produce an output. So, the set of outputs for a given loop should not be empty.

5 Evaluation

In this section, we perform an experimental evaluation of PPAT using a series of sequential scientific benchmarks in order to analyze how many loops can be transformed into `Pipeline`, `Farm` or `Map` parallel patterns. To do so, we use the following hardware and software components:

Table 2: Results for the benchmark suites. P, F and M stand for the number of `Pipeline`, `Farm` and `Map` patterns detected, respectively.

		PPAT			Manual		
Test	**Loops**	**P**	**F**	**M**	**P**	**F**	**M**
b+tree	80	3	7	7	2	7	7
particlefilter	44	1	8	8	1	10	10
bfs	7	0	1	1	0	2	1
nw	12	0	6	6	0	6	6
cfd	78	16	12	12	15	13	13
lavaMD	10	0	1	1	0	2	2
heartwall	54	1	4	2	0	4	3
nn	2	0	0	0	0	0	0
backprop	28	0	2	2	0	5	5

(a) `Rodinia` benchmark.

		PPAT			Manual		
Test	**Loops**	**P**	**F**	**M**	**P**	**F**	**M**
IS	16	1	8	8	0	9	9
LU	187	1	37	37	1	81	81
FT	41	0	7	7	3	20	20
EP	8	1	2	2	0	3	3
MG	80	1	26	26	1	44	44
UA	321	3	116	116	2	171	170
DC	30	2	5	5	1	7	7
SP	250	1	51	51	1	103	103
BT	181	1	46	46	1	78	78

(b) `NAS` benchmark.

- *Target platform*. The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57.
- *Software*. The PPAT tool has been compiled using the Clang compiler from the LLVM compiler infrastructure v3.7.0 and the proposed attributes. For refactoring the codes, we leveraged the OpenMP and TBB [14] parallel programming frameworks.
- *Benchmarks*. To evaluate PPAT, we used the sequential versions of the two scientific benchmark suites: `Rodinia` [19] and `NAS` Parallel Benchmarks (NPB) [1]. We also leverage a processing video application taken from the FastFlow framework [4] as a real use case.

Our evaluation methodology is based on a comparison between a manual inspection and an automatic one, using PPAT, of the loops appearing in the benchmark codes. To conduct a double-blind study, the manual inspection is performed before the automatic one, so that the manual results are not biased by those from PPAT. For each benchmark, we collect the number of loops and parallel patterns detected. Then, we discuss the results collected during the manual inspection with those obtained by PPAT in order to demonstrate the quality of the pattern detection process.

Moreover, we transform the sequential code of the `Rodinia` benchmark tests using the PPAT annotations, and then, compare the performance of the PPAT parallel versions with respect to the parallel ones provided by the benchmarks suites. Finally, we test PPAT on a real use case in order to evaluate the quality of the pattern detection. The results obtained for this test are contrasted with the FastFlow parallel version.

5.1 Results for the benchmarks suites

As mentioned, the two benchmark suites used to evaluate PPAT are `Rodinia` and `NAS`. Note that we only employ the sequential versions of these benchmarks to detect potential parallel patterns. Table 2 presents the results obtained by PPAT and manual inspection for both `Rodinia` and `NAS` benchmarks. As can be seen, the number of patterns detected manually and through PPAT are perfectly matching. Therefore, we observe that the pattern detection quality of PPAT is close to that performed by a human expert.

Focusing on the differences between manual and automatic detection, as can be seen in Table 2, the human expert is able to detect more `Farm` patterns than the tool for some of the tests. These differences mainly occur when the tool is not able to guarantee the parallel correctness of the pattern when shared variables are used. Listing 1 shows an example of this situation in a non-annotated `Farm`-like pattern. In this case, PPAT detects that the variable `new_dx` and iterators `j` and `k` are shared and, therefore, the tool cannot ensure that the code corresponds with a parallel pattern. However, PPAT lets the user know that, if these variables had been declared as local, the code would have corresponded with a parallel pattern. Listing 2 shows a version of the code in which we have used shared variables on purpose to demonstrate how the `Farm` and `Map` patterns would have been introduced. We observed in the annotated loops that this situation happens in many cases for the `NAS` benchmark tests, as the loop iterators are declared right before the loop sentences.

In order to analyze the benefits of PPAT on the patterns detected, we have implemented parallel versions of the `Rodinia` tests following parallelization suggestions given by PPAT. Both `Farm` and `Map` patterns were implemented using OpenMP, while the `Pipeline` pattern was introduced using the corresponding Intel TBB construction using "serial in-order" stages. Note that we have transformed all parallel loops suggested by PPAT, even the nested ones. Figure 2 shows performance results of the sequential, PPAT parallel and OpenMP versions for the `Rodinia` benchmark suite. In all cases, the tests were executed with the default input parameters, using 24 threads to fully populate the multi-core machine. For brevity, we only highlight some interesting cases. For the `lavaMD` test, we

Listing 1: Non-annotated **backprop** snippet.

```
363
364
365 for (j = 1; j <= ndelta; j++)
366
367
368  for (k = 0; k <= nly; k++) {
369   new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM *
          oldw[k][j]));
370   w[k][j] += new_dw;
371   oldw[k][j] = new_dw;
372  }
```

Listing 2: Annotated **backprop** snippet.

```
[[rph::map, rph::farm,
   rph::in(nly,delta,ly,oldw,w), rph::out(w,oldw)]]
for (int j = 1; j <= ndelta; j++)
 [[rph::map, rph::farm,
    rph::in(delta,ly,oldw,w,j), rph::out(w,oldw)]]
 for (int k = 0; k <= nly; k++) {
  float new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM *
        oldw[k][j]));
  w[k][j] += new_dw;
  oldw[k][j] = new_dw;
 }
```

note that PPAT is not able to annotate the main application hotspot (or loop) as a parallel pattern. This is mainly because some of its instructions, operating on sparse datasets, rely on indirect memory accesses in the form of `A[B[i]]`. In these cases, PPAT is not yet able to detect, at compile time, such potential data dependencies among loop iterations. Regarding the `heartwall` test, the PPAT version detects more parallel loops, or patterns, than those parallelized originally in the OpenMP version.

After this study, we make the following observations: *i)* PPAT obtains close performance figures with respect to the OpenMP implementations, as the PPAT annotations correspond with the OpenMP original pragmas in most cases; and *ii)* PPAT annotated versions may add slight overheads, as they contain initialization loops that were annotated to run in parallel, while in the original versions were vectorized by the compiler optimizations.

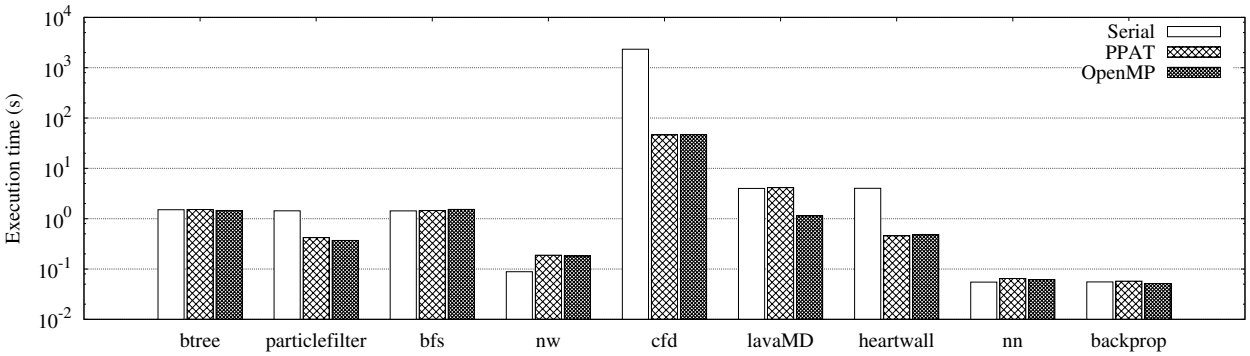

Figure 2: Execution time of sequential, transformed PPAT code and OpenMP versions of `Rodinia` benchmark.

5.2 Results of the Fastflow use case

Finally, we test PPAT using a video processing use case taken from the FastFlow framework. Basically, this application processes a stream of video frames captured by a camera and applies to each of them two different image processing filters: Gaussian blur and Sobel filters. Listing 3 shows the results of the analysis performed by PPAT. As can be seen, the tool is able to detect a `Pipeline` comprised of 4 stages that operate in the following way. The first stage detected captures the input video frames and forwards them to the next stage. The second and third stages are responsible for applying, in series, the two image processing filters Gaussian blur and Sobel, respectively. The last stage takes care of delivering the frames processed to the user. Another observation is that PPAT is able to determine as well that the filtering stages can be parallelized using `Farm` patterns individually. Therefore, multiple threads can execute the filters over the frames concurrently without side effects, as these operations have been determined to be pure functions.

In this particular case, we note that the parallel patterns detected are exactly the same as those used in the implementation of the original parallel version. Therefore, we believe that PPAT will be able to minimize the development costs of parallel C/C++ applications by means of detecting regions that can be represented as parallel patterns.

6 Conclusions and future works

In this work, we have presented the Parallel Pattern Analyzer Tool (PPAT), a tool that allows analyzing, detecting and annotating parallel patterns on sequential C/C++ codes using static analysis techniques. The experimental evaluation demonstrates that PPAT is able to obtain similar performance results as the “handmade” parallel versions of the benchmark suites tested. Therefore, reducing the human effort in transforming sequential codes into parallel.

Listing 3: Example of annotated loop from the video use case.

```
[[rph::pipeline(0) , rph::stream(cap,frames,frame,frame1)]]
for(;;)  {
 class cv::Mat frame1, frame;
 [[rph::stage(0), rph::plid(0), rph::in(cap), rph::out(frame1,frame,cap)]]{
  if(cap.read(frame) == false) break;
 }
 [[rph::stage(1), rph::plid(0), rph::farm, rph::in(frames,filter1,frame,frame1),
   rph::out(frames,frame1,frame)]]{
  frames++;
  if(filter1) {
   cv::GaussianBlur(frame, frame1, cv::Size(0, 0), 3);
   cv::addWeighted(frame, 1.5, frame1, -0.5, 0, frame);
  }
 }
 [[rph::stage(2), rph::plid(0), rph::farm, rph::in(filter2,frame), rph::out(frame)]]{
  if(filter2) Sobel(frame,frame,-1,1,0,3);
 }
 [[rph::stage(3), rph::plid(0), rph::in(outvideo,frame)]]{
  if(outvideo) { imshow("edges", frame); if(waitKey(30) >= 0) break;}
 }
}
```

As we have seen, this tool differs from others in three main aspects: *i)* PPAT is completely independent of the refactoring tool used, since it identifies parallel patterns; *ii)* PPAT performs a static analysis of the code, avoiding the use of profiling techniques and becoming much faster than other approaches; and *iii)* PPAT guarantees that parallel patterns detected comply with a series of requirements that ensure the correctness of the parallelization.

As part of the future work, we plan to extend PPAT and include more modules that support other parallel patterns, e.g., divide and conquer, reduce and stencil. Furthermore, we intend to endow PPAT with a decision system that selects the most suitable parallel pattern for a given code, from the performance point of view. Similarly, in case of nested patterns, the decision system should also be able to decide which of them are candidate to be introduced. Finally, we aim to develop a source-to-source refactoring tool such that, receiving as an input the annotated source code from PPAT will be able to translate the code into different parallel programming frameworks, e.g., TBB, FastFlow and the future parallel STL from C++17.

[2]

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[2] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM.

[3] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. Paraphrasing: Generating parallel programs using refactoring. In Bernhard Beckert, Ferruccio Damiani, FrankS. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 237–256. Springer Berlin Heidelberg, 2013.

[4] Marco Danelutto and Massimo Torquati. Structured Parallel Programming with "core" FastFlow. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.

[5] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sánchez, and J. Daniel García. Discovering pipeline parallel patterns in sequential legacy C++ codes. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP, Barcelona, Spain*, pages 11–19, 2016.

[6] ISO/IEC. Information technology – Programming languages – C++. International Standard ISO/IEC 14882:20111, ISO/IEC, Geneva, Switzerland, August 2011.

[2]This work was partially supported by the EU Projects ICT 644235 "RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the FP7 609666 "Repara: Reengineering and Enabling Performance And poweR of Applications".

[7] Xiaoming Li, Jack B. Dennis, Guang R. Gao, Willie Lim, Haitao Wei, Chao Yang, and Robert Pavel. FreshBreeze: A Data Flow Approach for Meeting DDDAS Challenges. *Procedia Computer Science*, 51(Complete):2573–2582, 2015.

[8] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*, chapter 3, pages 37–54. Springer International Publishing, August 2015.

[9] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. 2004.

[10] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[11] Anne Meade, Jim Buckley, and J. J. Collins. Challenges of evolving sequential to parallel code: An exploratory review. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 1–5, New York, NY, USA, 2011. ACM.

[12] Korbinian Molitorisz, Tobias Müller, and Walter F. Tichy. Patty: A pattern-based parallelization tool for the multicore age. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 153–163, New York, NY, USA, 2015. ACM.

[13] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[14] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[15] *EU Project "Reengineering and Enabling Performance And poweR of Applications"*, 2016. `http://repara-project.eu/`.

[16] *EU Project "RePhraseRefactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach"*, 2016. `http://rephrase.weebly.com/`.

[17] 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, 2010.

[18] L. M. Sanchez, J. Fernandez, R. Sotomayor, S. Escolar, and J. D. Garcia. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing*, 31(3):139–161, 2013.

[19] C. Shuai, M. Boyer, M. Jiayuan, D. Tarjan, J. W. Sheaffer, L. Sang-Ha, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[20] H. Sutter. Welcome to the Jungle. `http://herbsutter.com/welcome-to-the-jungle/`, 2012. [Last access 20th Oct 2016].

[21] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.