International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2018, Vol. 32(6) 779–788

DOI: 10.1177/1094342017695639
journals.sagepub.com/home/hpc
SAGE

Finding parallel patterns through static analysis in C++ applications

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

Abstract

Since the 'free lunch' of processor performance 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 address 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 to better exploit parallel hardware. Nevertheless, a large portion of production software, from a broad range of scientific and industrial areas, is still developed sequentially. Considering that these software modules contain thousands, or even millions, of lines of code, an extremely large amount of effort is needed to identify parallel regions. To pave the way in this area, this paper presents Parallel Pattern Analyzer Tool, a software component that aids the discovery and annotation of parallel patterns in source codes. This tool simplifies the transformation of sequential source code to parallel. Specifically, we provide support for identifying `Map`, `Farm`, and `Pipeline` parallel patterns and evaluate the quality of the detection for a set of different C++ applications.

Keywords

Parallel patterns, C++11 attributes, source code analysis tools

1 Introduction

Although most current computing hardware, such as multicore or many-core processors, graphics processing units, or coprocessors, has been envisioned for parallel computing, much of the prevailing production software is still sequential (Sutter, 2012). In other words, a large portion of the computing resources provided by modern architectures is underused. To exploit these resources, it becomes necessary to refactor sequential software into parallel. To tackle this issue, several solutions, such as parallel programming frameworks, have been developed to efficiently exploit parallel computing architectures efficiently (Sanchez et al., 2013). Indeed, there can be found multiple parallel programming frameworks that benefit from shared memory multicore 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.

Practical use cases in this sense are sequential data-intensive applications, broadly encountered in production, scientific, and industrial areas. A simple analysis of their codes would reveal that a vast majority of algorithms and methods could be refactored into parallel patterns (McCool et al., 2012). One solution to parallelize these codes is to translate them manually into parallel code; however, in most cases, this task is cumbersome and very complex for large applications. Another solution is to use refactoring tools, applications that can be used to advise developers or even semi-automatically transform sequential code into parallel (Brown et al., 2013). Although source codes transformed using these techniques do not often deliver the best performance, they aid in reducing necessary refactoring time (Meade et al., 2011).

Unfortunately, refactoring tools currently found are still premature, not yet being fully adopted by development centers. In fact, many of them are

[1]Department of Computer Science, University Carlos III of Madrid, Spain
[2]Department of Computer Science, University of Pisa, Italy

Corresponding author:
David del Rio Astorga, Department of Computer Science, University Carlos III of Madrid, 28911–Leganés, Spain.
Email: drio@pa.uc3m.es

human-supervised, with the developer 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 by del Rio Astorga et al. (2016) 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, owing to the modularity of PPAT, the tool can be easily extended to support other kinds of parallel pattern.

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

2 Related work

We find several research works 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 Rul et al. (2010) leverages the LLVM compiler 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 performing a static analysis without requiring any previous execution or profiling techniques, and supports both C and C++ programming languages.

Other contributions, such as the work of Molitorisz et al. (2015), 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 require a subsequent execution to discover potential data races and dependencies. Our work addresses these issues by checking memory accesses at compile time. Likewise, PoCC (Pouchet et al., 2009), 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. Conversely, we also find tools that detect parallel patterns using only profiling techniques. For example, DiscoPoP leverages dependency graphs to detect parallel patterns (Li et al., 2015b). 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 and Franke (2010), detects and transforms sequential code into parallel, introducing parallel `Pipeline` patterns. Alternatively, FreshBreeze (Li et al., 2015a), 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 note that approaches based on static analysis are not well extended in this area, since analyzing data dependencies becomes much more complex at compile time.

Some works take advantage of functional languages. For instance, Bozó et al. (2014) have developed a tool that detects parallel patterns in applications written in Erlang. Compared with other languages, Erlang features make the detection process much simpler. Nonetheless, this tool requires profiling techniques to decide which pattern is best suited for a concrete problem.

3 Parallel patterns

This section provides a brief overview of the parallel patterns supported by the PPAT tool, i.e. the `Pipeline`, `Farm`, and `Map` patterns (Mattson et al., 2004).

3.1 `Pipeline` *parallel pattern*

The `Pipeline` stream parallel pattern consists of a chain of processing entities arranged such that the output of each entity is the input of the next one. Considering a `Pipeline` of n stages, the ith 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

Table 1. RePhrase attributes.

RePhrase attribute	Description
`rph::pipeline`	Identifies a `Pipeline` pattern.
`rph::stream`	Identifies the data streams used across stages of a `rph::pipeline`.
`rph::stage`	Identifies a code section as a `Pipeline` stage.
`rph::plid`	Is associated with rph::stage and includes the `Pipeline` ID.
`rph::farm`	Specifies the `Farm` pattern.
`rph::map`	Determines the `Map` pattern.
`rph::in`	References the input variables of a pattern.
`rph::out`	References the pattern output variables.

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 might 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 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, 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 to 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 that items appear in the input stream with an interarrival time T_a and that the computation of f takes T_f, then at most T_f/T_a computations will take place in parallel, at any time. This amount of computations potentially happening in parallel might 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 are of types 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 of 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 the `Farm` and `Map` patterns seem to be similar, the difference lies in the fact that the `Farm` pattern works on a stream input data, while the `Map` pattern 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 of 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*

To annotate parallel patterns using custom C++11 attributes (ISO/IEC, 2011), we have extended the set of attributes defined for the projects Repara (Repara, 2016) and RePhrase (RePhrase, 2016). 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, the attributes enable the detection and transformation processes to be split, so that 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.

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.

Then, it walks through the Abstract Syntax Tree to collect relevant information about the source code and identify parallel patterns.

Figure 1 depicts the general work flow of PPAT. First, the tool receives the sequential source code files to 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 Abstract Syntax Tree, extracts loop-related subtrees and gathers information of different Abstract Syntax Tree nodes (e.g. variables, function calls, conditional statements). In addition, 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 the 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 dependency on a variable, the kind is set to write/read, since the argument can generate potential feedback 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 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 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 the 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 as follows.

- *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 that there are no variables written before they are read, i.e. that there are no read-after-write dependencies.
- *Multiple stages*. The last requirement checks whether the potential `Pipeline` can be split into 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 stage 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 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.

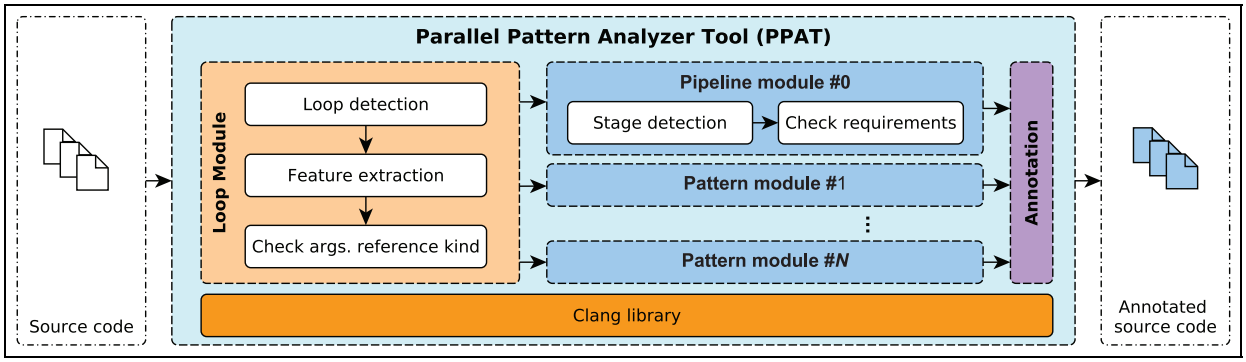

Figure 1. Workflow of Parallel Pattern Analyzer Tool.

- *No read-after-write dependencies*. There should not exist read-after-write 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, these 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. 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 constraints 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. Thus, 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.

- *Target platform*. The evaluation was carried out on a server platform comprising two 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 operating system is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57.
- *Software*. The PPAT tool was compiled using the Clang compiler from the LLVM compiler infrastructure v3.7.0 and the proposed attributes. To refactor the codes, we leveraged the OpenMP and TBB Reinders (2007) parallel programming frameworks.
- *Benchmarks*. To evaluate PPAT, we used the sequential versions of the two scientific benchmark suites: `Rodinia` (Shuai et al., 2009) and `NAS` Parallel Benchmarks (Bailey et al., 1991). We also leverage a processing video application taken from the FastFlow framework (Danelutto and Torquati, 2015) 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 obtained 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 compare the performance of the PPAT parallel versions with that of the parallel versions provided by the benchmark 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 benchmark suites

As mentioned, the two benchmark suites used to evaluate PPAT are `Rodinia` and `NAS`. Note that we only

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

(a) `Rodinia` benchmark.

Test	Loops	Parallel Pattern Analyzer Tool			Manual		
		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

(b) `NAS` benchmark.

Test	Loops	Parallel Pattern Analyzer Tool			Manual		
		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

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;
 }
```

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 matches perfectly. 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 shown 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 to 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

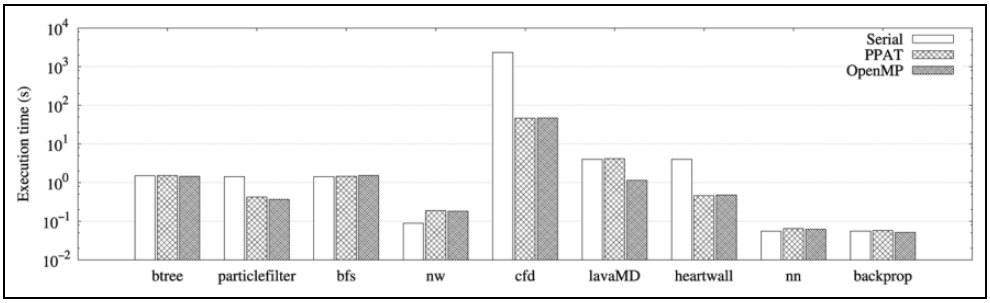

Figure 2. Execution time of sequential, transformed Parallel Pattern Analyzer Tool (PPAT) code, and OpenMP versions of Rodinia benchmark.

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;}
 }
}
```

cases for the NAS benchmark tests, as the loop iterators are declared immediately before the loop sentences.

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 the 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 populate the multicore machine fully. For brevity, we only highlight some interesting cases. For the lavaMD test, we note that PPAT is unable 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 good performance figures with respect to the OpenMP implementations, as the PPAT annotations correspond to 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, these were vectorized by the compiler optimizations.

5.2 Results of the FastFlow use case

Finally, we tested 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 two different image processing filters to each of them: 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 comprising four 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 also able to determine 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 or C++ applications by detecting regions that can be represented as parallel patterns.

6 Conclusions and future work

In this work, we have presented the Parallel Pattern Analyzer Tool (PPAT), a tool that allows analysis, detection, and annotation of parallel patterns on sequential C or C++ codes using static analysis techniques. The experimental evaluation demonstrates that PPAT is able to obtain similar performance results to "handmade" parallel versions of the benchmark suites tested, therefore, reducing the amount of human effort expended in transforming sequential codes into parallel.

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 the parallel patterns detected comply with a series of requirements that ensure the correctness of the parallelization.

As part of our 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, for nested patterns, the decision system should also be able to decide which of them are candidates to be introduced. Finally, we aim to develop a source-to-source refactoring tool, such that on receiving the annotated source code as an input, PPAT will be able to translate the code into different parallel programming frameworks, e.g. TBB, FastFlow, and the future parallel Standard Template Library from C++17.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: 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".

References

Bailey DH, Barszcz E, Barton JT, et al. (1991) The NAS parallel benchmarks. *International Journal of Supercomputer Applications* 5(3): 63–73.

Bozó I, Fordós V, Horvath Z, et al. (2014) Discovering parallel pattern candidates in Erlang. In: *Proceedings of the thirteenth ACM SIGPLAN workshop on Erlang*, Gothenburg, Sweden, 1–3 September 2014, pp.13–23. New York: ACM.

Brown C, Hammond K, Danelutto M, et al. (2013) Paraphrasing: Generating parallel programs using refactoring. In: Beckert B, Damiani F, de Boer F, et al (eds) *Formal Methods for Components and Objects*. Berlin: Springer, pp.237–256.

Danelutto M and Torquati M (2015) Structured parallel programming with "core" FastFlow. In: Zsók V, Horváth Z and Csató L (eds) *Central European Functional Programming School*. Cham: Springer, pp.29–75.

del Rio Astorga D, Dolz MF, Sánchez LM, et al. (2016) 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, 12–16 March 2016, pp.11–19. New York: ACM.

ISO/IEC 14882:20111 (2011) Information technology—Programming languages—C++.

Li X, Dennis JB, Gao GR, et al. (2015a) FreshBreeze: A data flow approach for meeting DDDAS challenges. *Procedia Computer Science* 51: 2573–2582.

Li Z, Atre R, Ul-Huda Z, et al. (2015b) DiscoPoP: A profiling tool to identify parallelization opportunities. In: Niethammer C, Gracia J, Knüpfer A, et al. (eds) *Tools for high performance computing 2014*, Cham: Springer International Publishing, pp.37–54.

Mattson TG, Sanders BA and Massingill BL (2004) *Patterns for Parallel Programming*. Boston, MA: Addison-Wesley.

McCool M, Reinders J and Robison AD (2012) *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco: Morgan Kaufmann.

Meade A, Buckley J and Collins JJ (2011) 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*, Szeged, Hungary, 5–6 September 2011, pp.1–5. New York: ACM.

Molitorisz K, Müller T and Tichy WF (2015) 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*, San Francisco, CA, 7–8 February 2015, pp.153–163. New York: ACM.

Pouchet LN, Bondhugula U, Bastoul C, et al. (2009) Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report.

Reinders J (2007) *Intel Threading Building Blocks-Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, CA: O'Reilly.

REPARA (2016) Reengineering and Enabling Performance And poweR of Applications. Available at: http://repara-project.eu/1 st October 2016.

REPHRASE (2016) RePhrase: Refactoring parallel heterogeneous resource-aware applications—a software engineering approach. Available at: http://rephrase-eu.weebly.com/ 1 st October 2016.

Rul S, Vandierendonck H and Bosschere KD (2010) A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing* 36(9): 531–551.

Sanchez LM, Fernandez J, Sotomayor R, et al. (2013) A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing* 31(3): 139–161.

Shuai C, Boyer M, Jiayuan M, et al. (2009) Rodinia: A benchmark suite for heterogeneous computing. In: *IEEE international symposium on workload characterization*, Austin, TX, 4–6 October 2009, pp.44–54. Piscataway, NJ: IEEE.

Sutter H (2012) Welcome to the jungle. Available at: http://herbsutter.com/welcome-to-the-jungle/ (accessed 20 October 2016).

Tournavitis G and Franke B (2010) 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*, Vienna, Austria, 11–15 September 2010, pp.377–388. New York: ACM.

Author biographies

David del Rio Astorga received his BSc degree in computer science from University Carlos III of Madrid, Spain, in 2013, and the MSc degree in informatics engineering from the same university in 2015. He is currently working toward the PhD degree in the Department of Computer Science at the University Carlos III of Madrid (Spain). His current research interests are programming models in the high performance computing domain.

Manuel F Dolz received his BSc degree in computer science from Universitat Jaume I de Castell, Spain, in 2008, and the MSc degree in parallel and distributed computing from the Polytechnic University of Valencia, Spain, in 2010. He obtained his PhD degree from Universitat Jaume I in 2014. Between 2013 and 2015, he worked as a postdoctoral research assistant at the Scientific Computing group in the University of Hamburg, Germany, which was responsible for the Exa2Green EU-project. Currently, he works as a postdoctoral research assistant at the Computer Architectures, Communications and Systems Research group in the University Carlos III of Madrid (Spain), which is responsible for the RePhrase EU-project. His main research interests are energy efficiency and programming models in the high performance computing domain.

Luis Miguel Sánchez received his PhD in computer science from University Carlos III of Madrid in 2009. He has published more than 30 journal and conference papers, mainly related with high performance computing, distributed or parallel file systems, and heterogeneous computing. He has participated in the European FP7 project REPARA as workpackage leader, researching and developing partitioning tools.

J Daniel García is an associate professor at the Computer Science and Engineering Department at University Carlos III of Madrid, Spain. He has worked in industry for major companies in Spain and Germany, including Telefonica, British Telecom, ING Bank, and Siemens, having the opportunity to participate in large-scale international projects. He has published more than 50 international journal and conference papers and has edited several journal special issues. He has participated in 10 technology transfer contracts and 12 publicly funded research projects, as well as many others before joining academia. He has been visiting researcher at University of Modena, Italy, and visiting faculty at University of Texas, A& M. Since 2008 he has been the Spanish Head of Delegation in the ISO C++ Standards Committee, where he actively participates in the development of the C++11 and C++14 standards. His current research interests focus on programming models for applications improvement. In particular, his aim is to improve both the performance of applications (to develop faster applications) and their maintainability (making them easier to modify).

Marco Danelutto is a professor at the Department of Computer Science in the University of Pisa. His main research interests lie in the field of parallel programming models, in particular in the area of parallel design patterns and algorithmic skeletons. He is author of more than 150 papers appearing in refereed international journals and conferences. He has been and is currently responsible of the University of Pisa research unit in different EU-funded projects (CoreGRID, GRIDcomp, ParaPhrase, REPARA, RePhrase) and a member of a number of different program committees of international conferences in the the field of parallel and distributed computing. He is currently responsible for the EuroMicro PDP conference series and has been a member of the EuroPar steering committee. He is currently responsible for the Master's degree in

Computer Science and Networking, a joint degree between the University of Pisa and the Scuola Superiore Sant'Anna.

Massimo Torquati is an assistant professor at the Computer Science Department of the University of Pisa, Italy. He has published more than 60 peer-reviewed papers in conference proceedings and journals, mostly in the fields of parallel and distributed programming and runtime systems for high performance computing. His current research interests are pattern-based parallel programming models, high performance data stream processing, concurrent lock-free data structures, and autonomic management in parallel systems. Currently, he is the main developer of the FastFlow parallel programming framework.