IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 1, 2018, accepted June 29, 2018, date of publication July 11, 2018, date of current version August 7, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2855064

Towards Automatic Parallelization of Stream Processing Applications

MANUEL F. DOLZ, DAVID DEL RIO ASTORGA, JAVIER FERNÁNDEZ, J. DANIEL GARCÍA, AND JESÚS CARRETERO

Department of Computer Science, Universidad Carlos III de Madrid, 28911 Leganés, Spain

Corresponding author: Manuel F. Dolz (mdolz@inf.uc3m.es)

This work was supported in part by the Spanish Ministerio de Economía y Competitividad through the Project Toward Unification of HPC and Big Data Paradigms under Grant TIN2016-79637-P and in part by the EU Project RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications under Grant ICT 644235.

ABSTRACT Parallelizing and optimizing codes for recent multi-/many-core processors have been recognized to be a complex task. For this reason, strategies to automatically transform sequential codes into parallel and discover optimization opportunities are crucial to relieve the burden to developers. In this paper, we present a compile-time framework to (semi) automatically find parallel patterns (Pipeline and Farm) and transform sequential streaming applications into parallel using GrPPI, a generic parallel pattern interface. This framework uses a novel pipeline stage-balancing technique which provides the code generator module with the necessary information to produce balanced pipelines. The evaluation, using a synthetic video benchmark and a real-world computer vision application, demonstrates that the presented framework is capable of producing parallel and optimized versions of the application. A comparison study under several thread-core oversubscribed conditions reveals that the framework can bring comparable performance results with respect to the Intel TBB programming framework.

INDEX TERMS Refactoring framework, automatic parallelization, load-balanced pipeline, parallel patterns.

I. INTRODUCTION

Whereas over the years high performance computing (HPC) facilities have enjoyed considerable enhancements —mostly due to the increases in the processors frequency—, the end of Moore's law in the middle of the past decade marked a milestone that shifted the microprocessor designs towards the multi-/many-core era [1]. This fact led scientific applications to cease their continuous improvements, being necessary to rewrite them in parallel in order to leverage resources provided by modern multi-/many-core processors and accelerators [2]. Since then, however, there remains a significant portion of legacy codes, coming from a broad range of scientific areas, running in sequential on parallel platforms. Although these applications are gradually parallelized, adapting and tuning them to operate on modern architectures is not straightforward.

For several years, the HPC community is making tremendous efforts in the design of frameworks and parallel programming models to relieve the burden on parallelizing applications. Numerous examples of programming models for multi-core architectures, such as OpenMP [3], OmpSs [4] or OpenACC [5], permit to exploit parallel hardware resources by merely annotating the code. Nevertheless, finding out which portions of the program are suitable to be parallelized is not elemental: programmers require additional expertise for writing applications in parallel, apart from the knowledge necessary in the application domain. As a result, applications are often rewritten using a daunting number of low-level concurrency primitives, such as specific threading management and communication mechanisms. This fact may prevent applications from scaling well and lead to communication overheads, load imbalance, poor data locality, deadlocks, data races, etc. Additionally, maintaining and porting applications to other parallel platforms can be an error-prone task which may require enormous efforts from a domain expert.

Among several approaches to address the parallel programming problem, we encounter automatic parallelization techniques and high-level parallel abstractions. On the one hand, automatic parallelization is the process of automatically converting a sequential program to a version that can directly run on parallel architectures. Indeed, parallel refactoring mechanisms have been widely explored since the rise of multi-core processors [6]. Unfortunately, tools found are still

premature and have not yet been fully adopted by developers. On the other hand, parallel patterns offer high-level APIs which abstract programmers from such complex concurrency mechanisms by providing standard interfaces to some recurrent parallel constructions encountered in data-intensive (e.g., MapReduce, Stencil) and stream processing (e.g., Pipeline, Farm) applications. Even though patterns alleviate the burden of parallel programming, developers still have to select and tune the most appropriate skeletons to provide efficient parallel versions of a given algorithm [7].

To mitigate these issues, in this paper we propose a complete framework which takes advantage of both approaches mentioned above to deal with the parallel programming challenge in the context of stream processing applications. This framework proceeds in the following phases: i) detection of stream pattern candidates in sequential codes; ii) rewrite found instances using a generic parallel pattern interface; and iii) optimization of Pipeline constructions to improve the throughput using a profile-guided approach. Concretely, in this paper we contribute with the following:

- We present a parallel pattern refactoring framework able to detect the Pipeline and Farm stream patterns in sequential C++ codes and rewrite candidates using GrPPI, a generic parallel pattern interface.
- We develop Pipeline balancing algorithms that calculate an optimal arrangement of Pipeline instances. These algorithms use an off-line profile-guided approach to optimize the architecture resource usage.
- We investigate the benefits of oversubscribed scenarios, i.e., increasing the number of threads above the available platform cores, to develop a novel strategy able to find the optimal concurrency degree.
- We analyze the presented Pipeline balancing algorithms and compare their time-to-solution and Pipelines execution time using the proposed framework.
- We evaluate the refactoring framework and the different strategies to find the optimal concurrency with varying configurations of the PiBa algorithms and GrPPI back ends. This evaluation is carried out using two stream processing applications, a synthetic video benchmark and a computer vision use case for detecting lane lines.

The rest of this document is organized as follows. Section II revisits some related works in the area. Section III defines the two major components used by the framework: the parallel pattern analyzer tool and the parallel pattern interface. Section IV describes the parallel pattern refactoring framework, as for the main contribution of this paper. Sections V and VI present the variants of the Pipeline balancing algorithms and strategy for finding an optimal concurrency degree. In Section VII, we evaluate the refactorings and the balancing algorithms under several configurations. Finally, Section VIII closes this paper with some concluding remarks and future works.

II. RELATED WORK

In the literature we find numerous efforts addressing both parallel pattern refactorization and optimization processes. We classify these efforts in the following categories: i) detection of parallel patterns; ii) refactorization tools for rewriting codes; iii) optimization of pattern-based parallel applications.

Detecting parallel patterns has been widely recognized to be a complex task [8]. This situation occurs because there exist no standard references to validate the patterns found in source codes. Also due to the wide variety of detection strategies that can be used at both compile- and run-time. For instance, Sean Rul *et al.* [9] instrument loops at LLVM-IR level and perform a run-time profiling analysis to decide whether a loop is a pipeline or not. Similarly, the tool by Li *et al.* [10] leverages dependency graphs to detect parallel patterns at run-time using an instrumented version of the codes. Other contributions, such as the work by Molitorisz *et al.* [11], detect potential parallel patterns at compile-time but require subsequent executions to find out data races and dependencies in the resulting codes. Alternatively, the tool FreshBreeze [12] uses static loop detection techniques to analyze dependencies and transform parallelizable loops using a task tree-structured model. In general, we observe that approaches based on static analysis are not as extended as those using dynamic strategies, basically because data dependency analyses are more complex to perform at compile-time than at run-time.

On the other hand, refactoring of applications has also been a large field of research in the last decades [13]. However many of the refactoring techniques for parallel programming presented in the literature are to some extent limited [14]. In a first inspection, we notice that many of these approaches are focused to specific structural rearrangements of the code (e.g., loop optimizations), which have been lately included in recent compiler optimizer modules using polyhedral or unimodular transformations [15], [16]. However, transformations applying parallel design patterns or algorithmic skeletons, have not been yet widely explored. In this line, we encounter works proposing pattern rewriting rules [7], commercial frameworks to introduce parallelism using structural refactoring steps [17], [18] and projects that aim to develop advanced refactoring frameworks [19]. Other works combine the refactoring and optimization techniques. For instance, Aldinucci and Danelutto [20] propose a method to minimize the service time of stream parallel pattern compositions by applying systematic rewritings. Also, some of the rewriting and tuning techniques in the literature have been embedded into skeleton-based programming frameworks [21].

As previously observed, while refactoring techniques already leverage optimization techniques at code rewriting, we also encounter efforts dealing exclusively with optimization of parallel skeletons [22]. Particularly, we find a considerable number of studies focused on load balancing techniques for pipeline constructions. For instance, the work

by Moreno *et al.* [23], [24] use mathematical models to replicate the slowest and collapse the fastest pipeline stages. However, it assumes that all of them are pure functions. Research on pipeline optimizations comparing task-parallelism and work-stealing with both stage replication/merging techniques can also be found in the state-of-the-art [25]. On the other hand, Kamruzzaman *et al.* [26] map all pipeline stages to a single thread and distribute loop iterations among worker threads, handling sequential stages using token-based synchronization. In contrast, Li *et al.* [27] present a collection of algorithms that can replicate pipeline stages to achieve optimal or near-optimal throughput but do not consider communication overheads.

In a nutshell, the contributions in this paper extend the current literature with a framework able to detect the Pipeline and Farm patterns and generate parallel codes which are optimized through a series of profiling steps. In our case, the parallelism is exploited by GrPPI, which offers a unified pattern interface to some programming frameworks (C++ threads, OpenMP or Intel TBB), while the pipeline balancing optimizations are performed using an off-line profile-guided approach.

III. PARALLELIZATION MECHANISMS

In this section, we overview the two major components used for building up the refactoring framework presented in this paper: the Parallel Pattern Analyzer Tool (PPAT) and the Generic and reusable Parallel Pattern Interface (GrPPI). Finally, we describe the parallel patterns supported by the proposed framework.

A. PARALLEL PATTERN ANALYZER TOOL

We leverage 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 [28]. This tool analyzes the Abstract Syntax Tree (AST) to collect relevant information and identify potential parallel patterns in C/C++ sequential codes. The parallel patterns encountered by the tool are annotated in the output source codes. Figure 1 depicts the general workflow of PPAT.

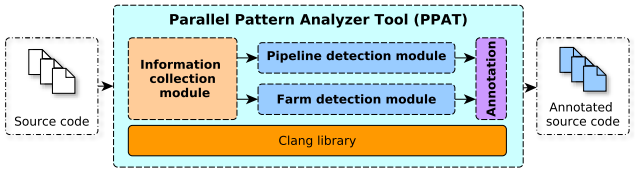


FIGURE 1. Parallel pattern analyzer tool workflow.

The significant advantages of PPAT are the following: i) it is entirely independent of the refactoring tool used; ii) it performs a static analysis of the code, avoiding profiling overheads; and iii) it guarantees that patterns detected comply with a series of requirements that ensure the correctness of the resulting parallel code. However, PPAT has some limitations when detecting Pipeline constructions at compile-time: it does not split their stages equally according to their computational load. Instead, it divides the Pipeline into the maximum number of stages, which may potentially lead to unbalanced stages. For this reason, in this paper, we propose a refactoring framework that leverages PPAT and is capable of generating more balanced and optimized Pipeline codes.

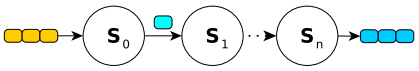
B. GENERIC AND REUSABLE PARALLEL PATTERN INTERFACE

As for the parallel pattern interface for our refactoring framework, we take advantage of GrPPI, a generic and reusable parallel pattern interface for C++ applications [29]. Specifically, GrPPI takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as a unified interface between the OpenMP, C++ threads and Intel TBB parallel programming models. Its design allows users to leverage the aforementioned execution frameworks in a single and compact interface, hiding away the complexity behind the use of concurrency mechanisms. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combine them to arrange more complex constructions. Thanks to this property, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relatively small efforts, having as a result portable codes that can be executed on multiple platforms.

C. PIPELINE AND FARM PARALLEL PATTERNS

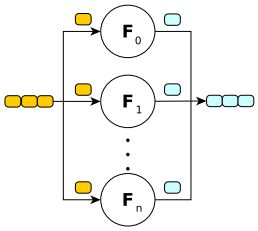
As an initial version of the refactoring framework, we consider only two stream-based parallel patterns: Pipeline and Farm [30]–[32]. The definitions for these two patterns are the following:

- Pipeline: This pattern processes the items appearing on the input stream in a linear sequence of stages. Each stage of this pattern processes data produced by the previous stage in the pipe and delivers results to the next one. Provided that the i-th stage in a n-staged Pipeline computes the function $f_i : \alpha \rightarrow \beta$, the Pipeline delivers the item x_i to the output stream applying the function $f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$. Assuming that the items appearing on the input stream are $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ the computation of stage f_j over the partial result relative to x_i can occur in parallel to the computation of f_{j+k} over the partial result relative to x_{i-k}.
  Figure 2a shows the Pipeline diagram and its GrPPI-related interface. This GrPPI interface receives the execution policy (`exec_policy`) and the functions (`stages`) related to the Pipeline stages by universal reference (`&&`). As can be seen, the C++ interface uses templates, making it more flexible and reusable for any data type. Note the use of C++11 variadic templates (`...`), allowing a Pipeline to have an arbitrary number of stages by receiving a collection of functions passed as arguments. It is important to remark that the C++ threads back end creates a thread per Pipeline




```
template <typename E, typename ... S>
void pipeline(E exec_policy, S && ... stages);
```

(a)




```
template <typename F>
void farm(int num_replicas, F && func);
```

(b)

FIGURE 2. Pipeline and Farm patterns diagrams with its GrPPI interface. (a) Pipeline pattern. (b) Farm pattern.

stage, while the Intel TBB back end leverages a task-parallelism paradigm with a pool of worker threads that continuously poll for work from a queue of ready tasks.

- Farm: This pattern computes in parallel the function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream. Thus, for each item x_i on the input stream the Farm pattern delivers an item to the output stream as $f(x_i)$. This pattern assumes that the function f is stateless, and can be applied to different stream items in parallel. In other words, the computation of $f(x_i)$ and $f(x_j)$ $\forall i, j : i \neq j$ may be computed in parallel by the different concurrent entities (replicas).
  Figure 2b depicts the Farm diagram along with its GrPPI interface. This interface receives the number of replicas (`num_replicas`) and the Farm function (`func`) that processes individual items coming from the input stream. It is important to highlight that the C++ threads back end creates a thread per Farm replica. Conversely, the Intel TBB back end creates a new task for processing each incoming item with the Farm function, executed by any of the worker threads in the pool.

In general, these patterns can be composed among them to produce more complex constructions. In this paper, the compositions supported between the Pipeline and Farm patterns are those in which the Pipeline stages can be parallelized individually using the Farm pattern, also known as parallel stages [31]. Thus, if a Pipeline stage corresponds with a pure function, this can be computed in parallel following a Farm construction. Throughout this paper, we denote the sequential stages of a Pipeline with "`s`", the Farm stages with "`f`" and the communication between two stages with the symbol "`|`". For instance, a Pipeline comprised of 4 stages, where the second and the third are Farm stages, is represented by "`(s|f|f|s)`".

IV. THE REFACTORING FRAMEWORK

In this section, we describe the Parallel Pattern Refactoring Framework, namely PPRF, as the main contribution of this paper. Figure 3 shows the general workflow of the framework. This workflow detects Pipeline and Farm candidates in sequential C++ codes using PPAT, balances and optimizes the pipelines found and generates parallel code using the GrPPI pattern interface. We describe next the three principal steps of PPRF to carry out these transformations.

Pattern detection and instrumentation. During the first stage, the Parallel Pattern Analyzer Tool (PPAT) receives the sequential C++ code to be analyzed. Internally, PPAT detects code constructions that match with any of the supported patterns (Pipeline and Farm). Afterward, the new refactoring module introduces the GrPPI pattern interfaces accordingly on those constructions. Additionally, it instruments the Pipeline stages for measuring their execution time.

Pipeline balancing and optimization. In this phase, the application is run to collect average execution times of the Pipeline stages. Using this execution time data, we feed our Pipeline Balancing Algorithm (PiBa) for generating its optimal configuration. This algorithm merges and replicates the Pipeline stages according to the available CPU cores. Additionally, PiBa is able to determine the optimal concurrency degree by refining the Pipeline arrangement iteratively until finding the configuration that delivers an optimal performance. Note that this framework uses profiling techniques to measure Pipeline stage execution time; therefore, the input data used during the profiling phase should be representative enough to perform the optimizations.

Parallel code generation. Finally, the refactoring module receives the Pipeline configuration and generates the parallel code version using the GrPPI interface. Specifically, the refactoring module is rerun to generate the final Pipeline arrangement according to the configuration determined by PiBa in the previous step.

Listing 1 depicts a worked example of the aforementioned PPRF steps. Listing 1a shows an excerpt of sequential code containing potential parallel patterns. Once PPAT has been executed, and parallel patterns have been found, the original code is transformed using the GrPPI interface and instrumented for measuring the execution time of the detected Pipeline stages (see Listing 1b). Note that this instrumentation is provided internally by the GrPPI execution model. With the Pipeline profile data, PiBa is able to determine an optimal configuration, merging the first and second Pipeline stages and using two threads for executing the third stage using a Farm construction. Finally, Listing 1c illustrates the transformed code taking into account the Pipeline arrangement stated by PiBa in the previous step. Thanks to this framework, sequential codes matching with Pipeline constructions can be automatically transformed into parallel and optimized for the target platform.

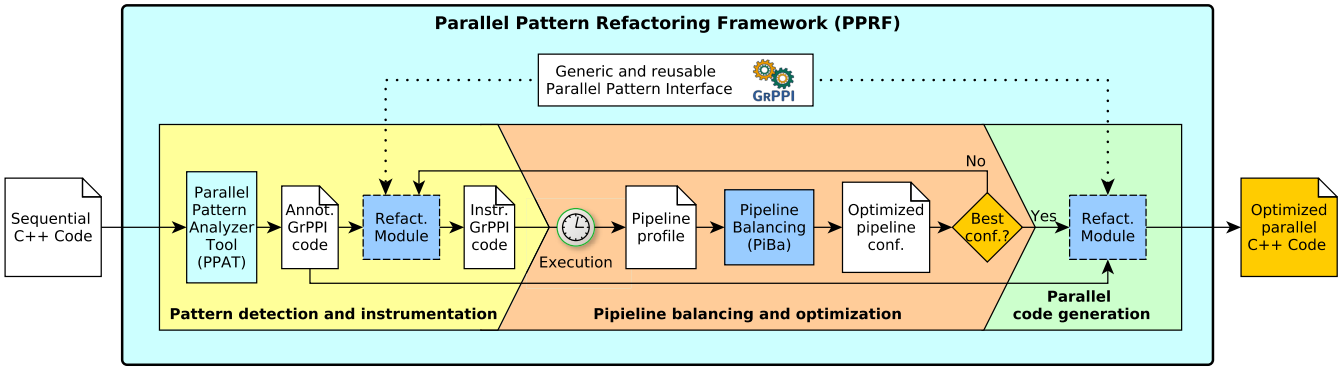


FIGURE 3. Workflow of the Parallel Pattern Refactoring Framework.

(a)

```
while ( i++ < MAX ){

  int it = doStuff(i);

  it = doStuff(it);

  it = pureFunction(it);

  cout << it << endl;

}
```

(b)

```
sequential_execution seq_exec{};
seq_exec.enable_instrumentation();
grppi::pipeline(seq_exec,
  [&]() -> optional<int> {
    return ( i++ < MAX ) ? i : {};
  },
  [](int it){ return doStuff(it); },
  [](int it){ return doStuff(it); },
  grppi::farm(1,
    [](int it){ return pureFunction(it); }
  ),
  [&](int it){ cout << it << endl; }
);
```

(c)

```
// GrPPI parallel pipeline + farm
// pipeline with 1 thread per stage
grppi::pipeline(parallel_execution_thr{},
  [&]() -> optional<int> {
    return ( i++ < MAX ) ? i : {};
  }, // Merged stages 1+2
  [](int it){ it = doStuff(it);
              return doStuff(it); },
  grppi::farm(2,
    [](int it){ return pureFunction(it); }
  ),
  [&](int it){ cout << it << endl; }
);
```

Listing 1. Example of PPRF code transformation. (a) Sequential code. (b) Instrumented code. (c) Optimized parallel code.

V. THE PIPELINE STAGE BALANCING ALGORITHM

In this section, we describe in detail the Pipeline Balancing Algorithm, namely PiBa, as part of the PPRF framework. Particularly, this algorithm tries to compute the optimal, or near optimal, arrangement of the Pipeline stages to optimize the use of all the available resources, i.e., CPU cores, for the target application.

We make first the following assumptions about the Pipeline construction. Consider a Pipeline $\mathcal{P}$ as the following list of stages

$$\mathcal{P} = (\lambda_1, \lambda_2, \ldots, \lambda_n),$$

where the i-th stage λ_i is the tuple (γ_i, t_i, r_i), being γ_i the function kind (pure or impure), t_i the execution time, and r_i the number of replicas or worker entities that execute the stage. Note that if the function related to a stage λ_i is pure, it can be executed in parallel by multiple replicas using a Farm pattern. Otherwise, if the function is impure, it can only be executed in series by one replica.

We next define the stage service time as the division of its execution time between the number of its assigned replicas (see Equation 1). We also define the pipeline service time as the maximum service time of its stages (see Equation 2). Note that the service time is the inverse of the throughput, i.e., units of time per processed item.

$$ST_{\lambda_i} = \frac{t_i}{r_i} \tag{1}$$

$$ST_{\mathcal{P}} = \max(ST_{\lambda_1}, ST_{\lambda_2}, \ldots, ST_{\lambda_n}) \tag{2}$$

Therefore, the goal of the PiBa algorithm is to find the Pipeline equivalent to the original one with the minimum service time using all the available CPU cores for the target application. For that, PiBa receives two input parameters: the Pipeline profile containing the execution time related to its stages, and the number of available CPU cores. Next, it leverages the following two techniques to find the optimal, or near optimal, Pipeline stage arrangement:

- Stage Replication: this technique increases the parallelism degree of those stages that follow the Farm pattern. Given the definition of this pattern, the degree of a Farm stage can be increased by introducing a new concurrent entity, i.e., $r_i = r_i + 1$. With this, the stage throughput is improved.
- Stage Merging: to reduce the total number of Pipeline stages, this technique combines two consecutive stages in single one, whose execution time is the sum of both original stages. It is important to remark that if the original stages are pure functions, the new merged stage is also pure and can be executed in parallel. In any other case, the resulting stage is impure. This technique is described in detail in Algorithm 1.

Using these techniques, we present in the following sections the three different alternatives implementing the PiBa algorithm using i) brute-force search; ii) heuristic search; and iii) an hybrid solution combining i and ii.

A. THE BRUTE-FORCE SEARCH

Regarding the first alternative, we present the naive version of the PiBa algorithm based on brute-force search. As can be

Algorithm 1 Stage Merging Technique

```
Function mergeStages(stageA, stageB, mergedStage)
    if stageA.kind = Pure & stageB.kind = Pure then
        mergedStage.kind ← Pure
        mergedStage.time ← stageA.time + stageB.time
        mergedStage.replicas ←
            stageA.replicas + stageB.replicas − 1
    else
        mergedStage.kind ← Impure
        mergedStage.time ← stageA.time + stageB.time
        mergedStage.replicas ← 1
    end
```

Algorithm 2 PiBa Brute-Force

```
Function PiBaBruteForce(
        pipeline[],          // Stages of the pipeline
        numCores)            // Number of cores

    for i ← 2 to min(numCores, pipeline[].numStages) do
        // Generate all the possible
        combinations with i stages
        pipeMerged[] ← genMerges(pipeline[], i)
        for j ← 0 to numCores − i do
            // Generate all the possible
            pipelines with j replicas
            pipeCombs[] ← genReplicas(pipeMerged[], j)
        end
    end
    pipeline[] ← minServiceTimePipeline(pipeCombs[])
```

Algorithm 3 PiBa Heuristic

```
Function pibaHeuristic(
        pipeline[],          // Stages of the pipeline
        numThreads)          // Number of threads

    Function getServiceTime(stage)
        return getTime(stage)/getReplicas(stage)

    while true do
        maxFarm ← getMaxFarmStage(pipeline[])
        maxSeq ← getMaxSeqStage(pipeline[])
        sumPipe[] ←
        mergeConsecutiveStages(pipeline[])
        minMerge ← getMinStage(sumPipe[])
        if countFarms(pipeline[]) > 0 &
          countReplicas(pipeline[]) < numThreads then
            maxFarm.replicas ← maxFarm.replicas + 1
        else if countReplicas(pipeline[]) > numThreads
        then
            applyMerging(pipeline[], minMerge)
        else if countFarms(pipeline[]) > 0
              & getServiceTime(maxFarm) >
                getServiceTime(maxSeq)
              & getServiceTime(maxFarm) >
                getServiceTime(minMerge) then
            maxFarm.replicas ← maxFarm.replicas + 1
            applyMerging(pipeline[], minMerge)
        else
            break
        end
    end
```

seen in Algorithm 2, this procedure uses the aforementioned stage merging technique (`genMerges` function) in order to compute all possible merging combinations for different number of stages, i.e., from 2 to the minimum between the number of cores and the number of stages. Then, for each of these combinations, the function `genReplicas` leverages the stage replication technique to calculate all feasible replications of the Farm stages. It is important to remark that these replications are made until the number of replicas equals the cores. Finally, the algorithm returns the Pipeline configuration with the minimum service time.

However, generating all possible merging combinations of stages and replicas has a non-negligible computational cost of $\Omega(n^3)$ and $O(c^n)$ for the best and worst cases, respectively. In this case, n stands for the combination of the number of stages and threads, while c represents a constant. Precisely, the best case corresponds with a full sequential Pipeline, as the function `generateReplicas` is not used. On the contrary, the worst case is related to a Pipeline whose stages are all Farm constructions.

B. THE HEURISTIC APPROACH

As noted in the previous section, the computational cost of the brute-force search is prohibitive, therefore it is necessary to design an heuristic so as to provide a solution within a reasonable time frame. Algorithm 3 presents the heuristic search of PiBa. This is implemented as an iterative procedure, where the pipeline service time is improved in each step. First, the heuristic calculates: i) the slowest sequential and Farm stages (`getMaxSeqStage` and `getMaxFarmStage`, respectively); and ii) the merged stage of two consecutive stages with the minimum service time (`mergeConsecutiveStages`[1] and `getMinStage`), using the stage merging technique. Afterwards, depending on the current Pipeline state, the heuristic performs iteratively one of the subsequent actions in the following order:

- **A1** If there are Farm stages and the total number of current replicas is less than the number of cores, the Farm stage with the maximum service time is granted with an additional replica.[2]
- **A2** If the total number of replicas is greater than the number of cores, the merge of two consecutive stages with the minimum service time is incorporated in the resulting Pipeline.
- **A3** If the slowest stage is a Farm and it is slower than the merge of two consecutive stages with the minimum service time, the algorithm adds a replica to the slowest Farm stage and incorporates the merged stage in the resulting Pipeline.
- **A4** If none of the previous conditions are met, the procedure finishes, as it cannot reduce the Pipelineservice time anymore.

[1]Note that the function `mergeConsecutiveStages` considers all possible mergings among consecutive **Pipeline** stages on a given iteration.

[2]The purpose of this action is to use all available CPU cores for the target application, though in extreme cases the **Pipeline** throughput improvement might be marginal.

TABLE 1. PiBA working example.

Iteration	Pipeline	Max. Stage	Replicas	Min. Merge	Action
1	$(s_1(3,1), f_2(5,1), f_3(7,1), s_4(2,1), f_5(1,1), s_6(1,1))$	$f_3(7,1)$	6	$(f_5 + s_6) \rightarrow s_{5,6}(2,1)$	**A2**
2	$(s_1(3,1), f_2(5,1), f_3(7,1), s_4(2,1), s_{5,6}(2,1))$	$f_3(7,1)$	5	$(s_4 + s_{5,6}) \rightarrow s_{4,5,6}(4,1)$	**A2**
3	$(s_1(3,1), f_2(5,1), f_3(7,1), s_{4,5,6}(4,1))$	$f_3(7,1)$	4	$(f_2 + f_3) \rightarrow f_{2,3}(12,2)$	**A3**
4	$(s_1(3,1), f_{2,3}(12,2), s_{4,5,6}(4,1))$	$f_{2,3}(12,2)$	4	$(s_1 + f_{2,3})(15,1)$	**A4**

To illustrate the workings of the PiBa heuristic, Table 1 shows an example of Pipeline that is steadily improved using this procedure. We start from a Pipeline comprised of three sequential and three Farm stages which are intended to run on four cores. Note that $s_i(t_i, r_i)$ stands for sequential stages and $f_i(t_i, r_i)$ for Farm stages, executed by r_i replicas. In the first iteration, the PiBa heuristic performs the action **A2**, since the total number of replicas is greater than the number of cores. Hence, f_5 and s_6 are merged into the single stage $s_{5,6}$. In the second iteration, action **A2** is taken again so as to merge s_4 and $s_{5,6}$ in $s_{4,5,6}$. In the third iteration, actions **A1** and **A2** are discarded, given that the number of replicas is equal to the number of cores. Therefore, since the slowest stage corresponds to a Farm and its service time is greater ($t_3 = 7$) than the fastest merged stage ($t_{2,3} = \frac{12}{2} = 6$), action **A3** merges f_2 and f_3. Finally, as the Pipelineservice time cannot be reduced anymore and the Pipeline replicas equal to the number of cores, the procedure finishes its execution. The computational complexity of the heuristic PiBa algorithm is $\Theta(n^2)$ for all cases.

C. THE HYBRID APPROACH

The hybrid approach combines the benefits from both brute-force and heuristic search approaches. This variant leverages the heuristic search presented in Algorithm 3, reducing the number of replicas until reaching the total number of available cores. From that point on, it continues improving the Pipeline service time using the brute-force search, as presented in Algorithm 2. Thereby, it reduces the computational best-case cost of the brute-force algorithm to $\Omega(n^2)$, providing more accurate service time optimizations than the single heuristic search approach. However, the worst-case cost is still $O(c^n)$, as the brute-force search.

VI. FINDING THE OPTIMAL CONCURRENCY DEGREE

According to the previous section, the PiBa algorithm transforms the Pipeline stages arrangement to have the same number of replicas as CPU cores used by the target application. This algorithm also balances, as much as possible, the stages workload. However, in real scenarios, the resulting Pipelines cannot be perfectly balanced in most of the cases. These situations cause bottlenecks due to imbalanced stages, which leads to underused resources (cores). A way to improve the resource usage, and thus the Pipeline performance, is to increase the number of replicas (threads) above the total number of cores. Hence, the additional threads can leverage the partially idle resources, overlapping threads contention with useful computation. In this context, we refer to oversubscription when an application uses more threads than available CPU cores and relies on the OS scheduler to keep them all busy.

To motivate this issue, we have implemented a synthetic benchmark consisting of two Pipeline collections, using CPU- and memory-intensive stages, respectively. These collections were comprised of 1,500 randomly-generated Pipelines constituted by a number of stages ranging from 4 to 12 and the percentage of Farm stages varying between 30 % and 90 %. Afterward, these Pipelines were processed using the PiBa algorithm to adjust the number of stages/replicas in the range of 4 to 24 threads. Next, we executed these Pipelines on an 8-core platform to find out their optimal concurrency degree. Figure 4 shows the performance attained by six representative CPU-/memory-intensive Pipelines for different number of threads.[3] Note that the lines extending vertically from the points represent the confidence intervals as a variability metric over 10 Pipeline runs. As can be observed, all Pipelines (P1–P6) improve their speedup up to the number of cores. Nevertheless, while some of them stop improving after this point, others continue boosting their performance beyond that. These results confirm our previous impressions where a concurrency degree higher than the total number of cores may improve the Pipeline performance in some cases.

A. ITERATIVE SEARCH

A naive search approach to obtain the optimal concurrency degree is to execute multiple times the PiBa algorithm for a different number of final replicas (instead of using the number of cores) and check which Pipeline arrangement delivers the best performance. However, this method is very time-consuming as the framework has to execute each time all feasible Pipeline combinations.

To optimize this naive search approach, we rely on the benchmark results that show, in many cases, strictly increasing speedup curves followed by monotonic decreasing ones. Note that this occurs in more than 90 % of the synthetic Pipelines. With this assertion, we can refine the previous approach with an iterative variant that applies PiBa to optimize the Pipeline configurations using a different number of threads, starting from the number of cores until the performance stops improving. Note that, to deal with the inherent variability of oversubscribed scenarios, we execute each

[3]Note that the CPU-intensive stages were mimicked by means of performing an arbitrary number of floating-point operations, while the memory-intensive compute was emulated accessing consecutively to 2 million elements in a floating-point array.

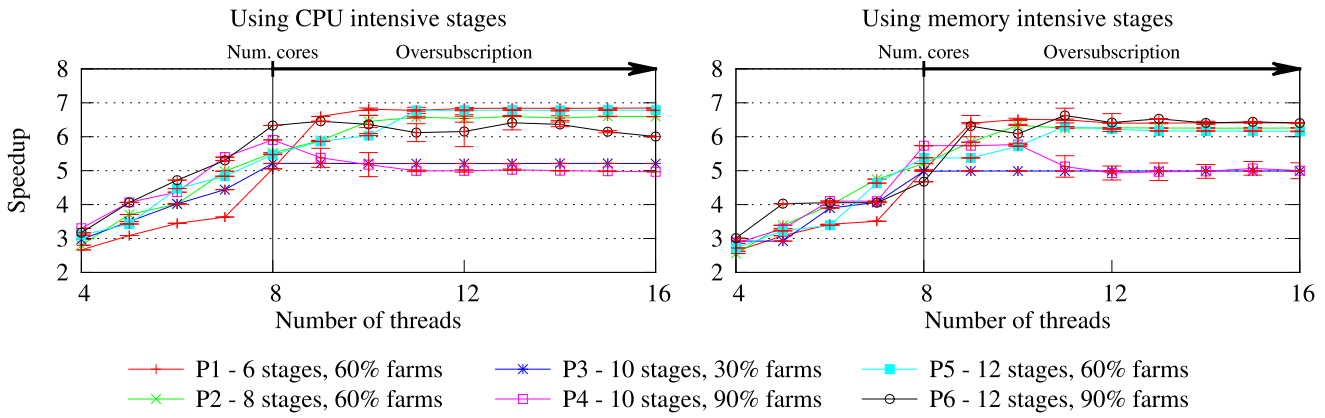


FIGURE 4. Analysis of the maximum concurrency degree using representative pipelines.

configuration multiple times to calculate averaged reliable values. With the averaged speedups in hand, this iterative search algorithm checks, in each iteration, if the speedup of a configuration B using $n + 1$ threads is higher than A using only n threads. If this is true, it checks for the next configuration using one more thread. This search stops when the configuration B using $n+1$ threads delivers worse or equal performance than A using n threads.

To illustrate the workings of this iterative search, Figure 5a depicts the case of the Pipeline P2 using CPU-intensive stages from Figure 4. In this scenario, the approach starts iterating from 8 cores on until the 4th iteration, given that in each step the configuration using $n + 1$ threads delivers higher speedups than using n threads. On the 5th iteration, i.e., with 12 threads, the performance stops improving. Then, the algorithm determines 11 threads as for the optimal concurrency degree. As a result, the Pipeline is oversubscribed with 3 threads, which overlap contention of the first 8 with useful computations.

B. GREEDY ITERATIVE SEARCH

The iterative search approach reduces the Pipeline configurations that have to be tested using a trial and error method. However, this variant can be very slow when the optimal concurrency degree is far ahead of the number of cores. To improve this algorithm, we propose a technique to obtain an estimation of the ideal degree. We start from the fact that the Pipelines are not perfectly balanced, and therefore, available resources are underused due to their inherent congestion. With this assertion, we estimate the Pipeline resource usage rate with

$$usage\_rate = \frac{eff\_usage}{real\_usage} = \frac{n\_items \cdot \sum_{i=0}^{n\_stages} t_i}{n\_cores \cdot exec\_time}, \quad (3)$$

where eff_usage and $real\_usage$ are estimations of the effective and real usage of the platform resources, respectively. In the equation, the effective usage is computed as the sequential execution time of the Pipeline stages multiplied by the number of items processed in a given test. Alternatively, the real usage is calculated as the execution time of the previous test multiplied by the total computational resources (cores). Using this technique, we propose a heuristic that pursues a greedy iterative approach and estimates in advance the ideal concurrency degree, instead of testing all configurations. This procedure, shown in Algorithm 4, iteratively applies the PiBa heuristic of Algorithm 3 with a number of threads that depends on the usage rate. This number is calculated on-the-fly dividing the current threads by the usage rate obtained with Equation 3 and rounding the result up. The procedure stops iterating when the usage rate using additional threads does not improve any more.

Algorithm 4 PiBa Heuristic With Oversubscription

```
Function pibaOversubs(
        pipeline[],          // Stages of the pipeline
        numCores,            // Number of cores
        numItems)            // Number of items

    pibaPipeline[] ← pipeline[]
    pibaHeuristic(pibaPipeline[], numCores)
    execTimeCur[] ← exec(pipeline[], numItems)
    numThreads ← numCores
    repeat
        usageRate ← usageRate(pipeline[], numItems,
        numCores, execTimeCur[])
        execTimeOld[] ← execTimeCur[]
        numThreads ← ceil(numThreads / usageRate)
        pibaHeuristic(pibaPipeline[], numThreads)
        execTimeCur[] ← exec(pipeline[], numItems)
    until stopCondition(execTimeOld[], execTimeCur[])
```

To demonstrate the benefits of this algorithm over the iterative version, Figure 5b shows the steps taken for the same Pipeline P2. As can be seen, three steps are required to find an optimal concurrency degree. In the first iteration (using 8 threads), the resource usage rate is roughly 69 %; thus the algorithm determines that 3 additional threads are required. With it, during the second iteration (using 11 threads), the usage rate increases to 82.5 %. Hence, the algorithm determines that 13 threads are needed to exploit idle resources. After the execution with 13 threads, the procedure stops, as it

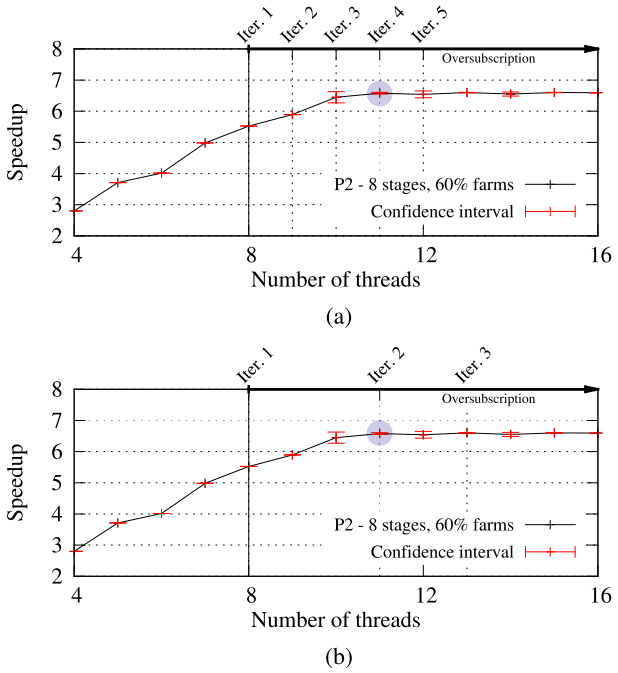


FIGURE 5. Automatic approaches for finding the optimal concurrency degree of PiBa Pipelines. (a) Iterative search. (b) Greedy iterative search.

detects that the usage rate does not improve. In this case, the optimal concurrency degree determined by the greedy iterative search is 11, i.e., coming to the same conclusion as the iterative algorithm but with fewer iterations.

VII. EVALUATION

In this section, we perform an experimental evaluation of the main components of the PPRF framework proposed in this paper: the Pipeline stage balancing algorithm and the strategy for finding the optimal concurrency degree. To carry out this evaluation, we use the following hardware and software components:

- Target platform: The evaluation has been carried out on a server platform equipped with 2× Intel Xeon Ivy Bridge E5-2630 v3 with a total of 16 cores running at 2.40 GHz, 20 MB of L3 cache and 256 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.5 LTS with kernel 3.13.0-85.
- Software: To evaluate the performance of PPRF framework, we leveraged the C++ threads, and Intel TBB GrPPI back ends. We used the Clang compiler from the LLVM compiler infrastructure v3.7.0 for finding and refactoring the code within the PPAT tool. Finally, the C++ compiler used to assemble the refactored GrPPI codes is GNU GCC v5.0.
- Benchmarks: To analyze the proposed balancing PiBa algorithms, we used the following three benchmarks:

  PIPE-BENCH: A benchmark collection of 1,500 randomly-generated Pipelines, as described in Section VI, with a number of stages ranging between 4 and 12. These Pipelines were combined with both sequential and parallel stages, where the degree of the parallel (Farm) stages varied between 30 % and 90 %. To generate the Pipelines source codes we used a Python script to emit the Pipeline and Farm GrPPI patterns leveraging the C++ back end. It is important to remark that the workload type for the stages were CPU-bound computations by means of performing double-precision operations.

  VIDEO-APP: A sequential video stream-processing synthetic benchmark composed of two types of filters (Gaussian Blur and Sobel operators) in order to detect edges appearing in the video frames.[4] Specifically, the application core works in a Pipeline fashion, in which the first and last stages read and write the video frames, respectively. Consequently, the intermediate stages apply the aforementioned filters in different ways. Regarding the workload type, the Gaussian Blur filter only performs arithmetic operations, while the Sobel operator also executes square root operations, both using double-precision numbers.

  LANE-DETECTION: A real-world computer vision application for detecting road lane lines in autonomous driving systems. This application is composed of a Pipeline in charge of processing individual video frames using a series of filter algorithms, such as the Canny edge detector and the Hough transform [33]. This application is vital to steer vehicles, as lanes represent a constant reference to the road. Indeed, identifying lane lines on the road is one of the most fundamental vision tasks required by autonomous cruise controls, lane change assist, lane centering, etc.

The evaluation methodology of this section consists of the following parts. First, we analyze the presented Pipeline balancing algorithms and compare their time-to-solution and the execution time of the Pipelines in Pipe-Bench using the PPRF framework. Next, we evaluate the different strategies to find the optimal concurrency degree in terms of speedup gains, number of steps taken and accuracy. Afterward, we test PPRF with Video-App, in order to transform the sequential code into parallel and evaluate different configurations of the PiBa algorithm and GrPPI back ends. Additionally, we complement the study using Video-App with a fine-grained analysis of different Pipeline configurations via execution traces. Finally, we employ Lane-Detection to demonstrate the benefits of PPRF in a real-world application.

A. ANALYSIS OF THE PIPELINE STAGE BALANCING ALGORITHM

To evaluate the different versions of the PiBa algorithm, we leveraged the collection of 1,500 synthetic Pipelines of Pipe-Bench. Afterward, we balanced its Pipelines using the three variants of PiBa, i.e., brute-force, heuristic and hybrid

[4]This benchmark has been inspired by an OpenCV edge detection example from http://docs.opencv.org/3.1.0/d3/d63/edge_8cpp-example.html.

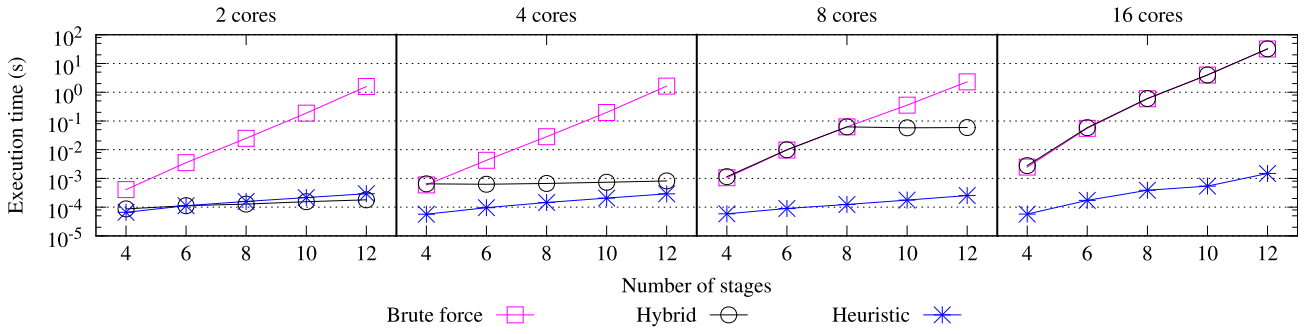


FIGURE 6. Time-to-solution of the basic PiBa algorithms.

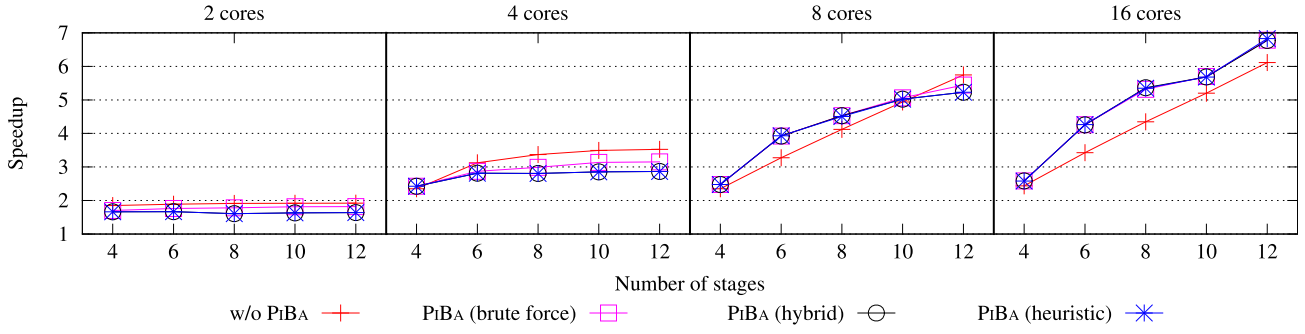


FIGURE 7. Speedup of the of Pipelines in Pipe-Bench balanced with the basic PiBa algorithm variants.

searches, and configured them to run on platforms having from 2 to 16 cores. Figure 6 depicts the averaged time-to-solution of the three PiBa variants for the different number of Pipeline stages and cores. As can be seen, the execution time of PiBa algorithms increases in general with the number of stages and cores. However, the growth rate of the brute-force is larger than for the heuristic version, which confirms the exponential algorithm complexity stated in Section V-A. A final observation is that the time-to-solution of the hybrid version is equal to the brute-force search for a number of stages lower or equal than the total cores.

Our next study analyzes the execution times of the same collection of Pipelines after balancing them using the three different PiBa versions. To emulate platforms with a different number of cores, we leveraged the Linux utility `taskset` for restricting the cores that can be used by the Pipeline threads.[5] As can be observed in Figure 7, the Pipelines processed using the heuristic and hybrid PiBa variants attain a similar performance in all cases. On the contrary, the brute-force search provides better speedup only when the number of cores is much lower than the stage count. Note that the observed speedup slowdown of the Pipelines balanced with PiBa is caused by the fact that the presented algorithms use as many threads as available cores. In contrast, the unbalanced Pipelines, always use at least as many threads as stages, leading to oversubscribed scenarios and, in some cases, to higher speedups with respect to the balanced Pipelines.

All in all, it can be concluded that the PiBa heuristic variant can provide acceptable well-balanced Pipelines in reasonable time frames, while the brute-force and hybrid searches are able to provide slightly better stage arrangements at the expense of prolonged time-to-solutions in the worst cases. Therefore, in the subsequent experiments of this paper, we select the heuristic variant as the default balancing PiBa algorithm.

B. ANALYSIS OF THE OPTIMAL CONCURRENCY DEGREE SEARCH ALGORITHMS

In this section, we analyze the extended PiBa algorithms proposed for finding the optimal concurrency degree with the basic PiBa variant. First, we compare the basic procedure with the two new strategies, the iterative and greedy iterative searches, able to obtain the ideal number of threads above the total number of cores. Next, we examine the performance of the Pipelines in Pipe-Bench on oversubscribed scenarios.

Figure 8 compares these algorithms regarding i) speedup, with respect to the sequential execution of the Pipeline; ii) the number of iterations needed until finding the optimal concurrency degree; and iii) the accuracy rate with regard to the best solution obtained with the brute-force approach. Note that these metrics were averaged over the 1,500 Pipelines comprised by the tested benchmark and using only eight cores. Focusing on the speedup results, it can be clearly seen

[5]The `taskset` utility is used to set or retrieve the CPU affinity of a running process in Linux. http://linuxcommand.org/man_pages/taskset1.html

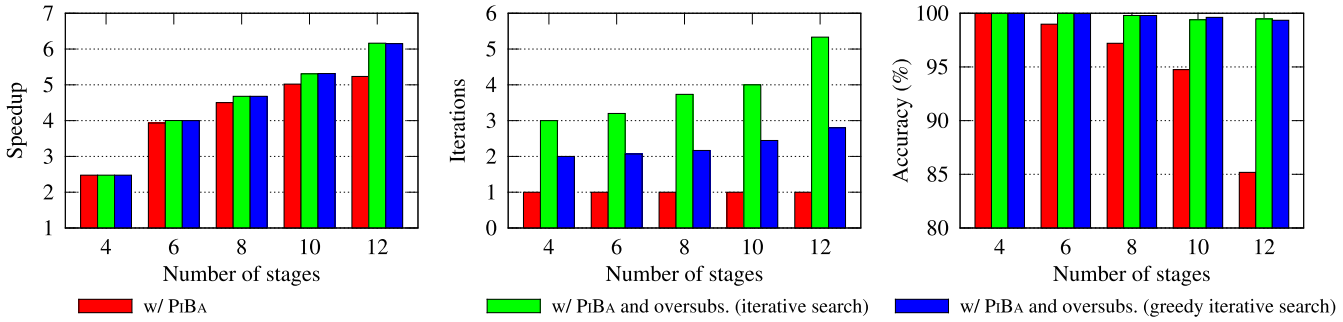


FIGURE 8. Comparative of the speedup, number of iterations and accuracy of the basic PiBa w.r.t the extended versions for finding the optimal concurrency degree.

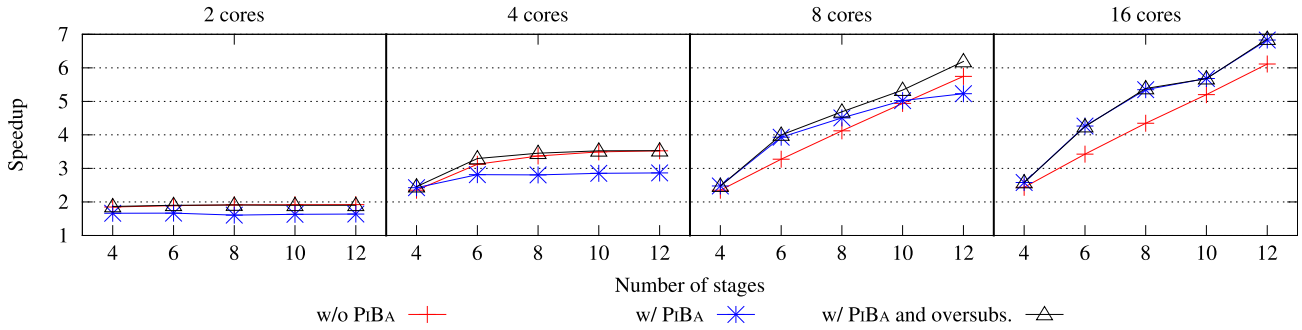


FIGURE 9. Speedup of the of Pipelines in Pipe-Bench balanced with the extended PiBa algorithms for oversubscribed scenarios.

that the extended PiBa algorithms for oversubscription are able to deliver better performance figures than using only the basic PiBa variant. This is mainly because the inherent threads contention is overlapped with useful computations of the exceeding threads. A detailed inspection of the executions revealed that this contention was caused by the threads suspended on an internal blocking queue when no items could be (de)queued.

Looking at the number of iterations taken by both PiBa algorithms including the oversubscription strategies, we observe that the iterative search requires more iterations with respect to the greedy approach. It is remarkable that this difference increases with the number of Pipeline stages since these Pipelines usually deliver better performance when employing oversubscribed threads. Finally, focusing on the accuracy, we notice that Pipelines comprising several stages attain lower accuracy than using only the basic PiBa algorithm. This is given by the fact the PiBa heuristic working alone does not always lead to optimal Pipeline arrangements when they contain many stages. For instance, from 10 stages on, the known error might be higher than 5 %. In contrast, with the extended PiBa algorithms, the accuracy is sustained regardless of the number of Pipeline stages. This demonstrates that the inherent flaws of the PiBa heuristic algorithm can be bypassed by increasing the concurrency degree using the extended PiBa algorithms.

In the light of the results, while both extended PiBa versions deliver similar performance figures, the iterative search requires more iterations than the greedy approach. Therefore, the experiments carried out hereafter are only performed using the greedy iterative search for finding the optimal concurrency degree.

As a complementary study, Figure 9 analyzes the speedups attained by the Pipelines in Pipe-Bench with a different number of cores when i) PiBa is not applied; ii) the basic PiBa algorithm is used; and iii) the extended PiBa algorithm along with the iterative greedy search are leveraged. As observed, when the number of stages is higher than the total cores, the fact of not balancing the Pipelines leads to oversubscribed scenarios, providing better performance than if the basic PiBa algorithm shortens the Pipelines. On the other hand, if the available cores are greater than the number of Pipeline stages, the balanced versions exploit better the resources. Looking at the results of the extended PiBa algorithm, the speedups are always higher or equal to the best case. The reason is that the oversubscribed threads in these Pipelines can effectively overlap the potential bottlenecks generated by the basic PiBa algorithm.

C. EVALUATION OF VIDEO-APP

To evaluate the PPRF framework along with the Pipeline balancing algorithms, we leveraged Video-App, a synthetic video stream-processing application able to detect edges appearing on the incoming frames. First, we fed the PPAT module with the sequential code to obtain an annotated version of the code with the potential parallel patterns detected.

(a)

```
for (;;) {
  Mat fr;

  if (!cap.read(fr)) break;

  Gaussian_pure(fr, ... );

  Sobel_pure(fr, ... );

  Gaussian_impure(fr, ... );
   ...
  Sobel_pure(fr, ... );

  Gaussian_pure(fr, ... );

  Sobel_impure(fr, ... );

  imshow("edges",fr);
}
```

(b)

```
sequential_execution seq_exeq;
seq_exec.enable_instrumentation();
grppi::pipeline(seq_exec,
  [] () -> optional<Mat> {
    Mat fr;
    if (!cap.read(fr)) return {};
    else return fr;
  },
  grppi::farm(1,
    [](Mat fr) {
      Gaussian_pure(fr, ... );
      return fr;
    }
  ),
  ...
  [](Mat fr) {
    Sobel_impure(fr, ... );
    return fr;
  },
  [](Mat fr) { imshow("edges",fr); }
}
```

(c)

```
grppi::pipeline(parallel_execution_thr{},
  [] () -> optional<Mat> {
    Mat fr;
    if (!cap.read(fr)) return {};
    else { Gaussian_pure(fr, ... );
           return fr; }
  },
  grppi::farm(2,
    [](Mat fr) {
      Sobel_pure(fr, ... );
      return fr;
    }
  ),
  ...
  [](Mat fr) {
    Gaussian_pure(fr, ... );
    Sobel_impure(fr, ... );
    return fr;
  },
  [](Mat fr) { imshow("edges",fr); }
}
```

Listing 2. **Example of annotated Pipeline from VIDEO-APP. (a) Sequential code. (b) Instrumented parallel code. (c) Optimized parallel code.**

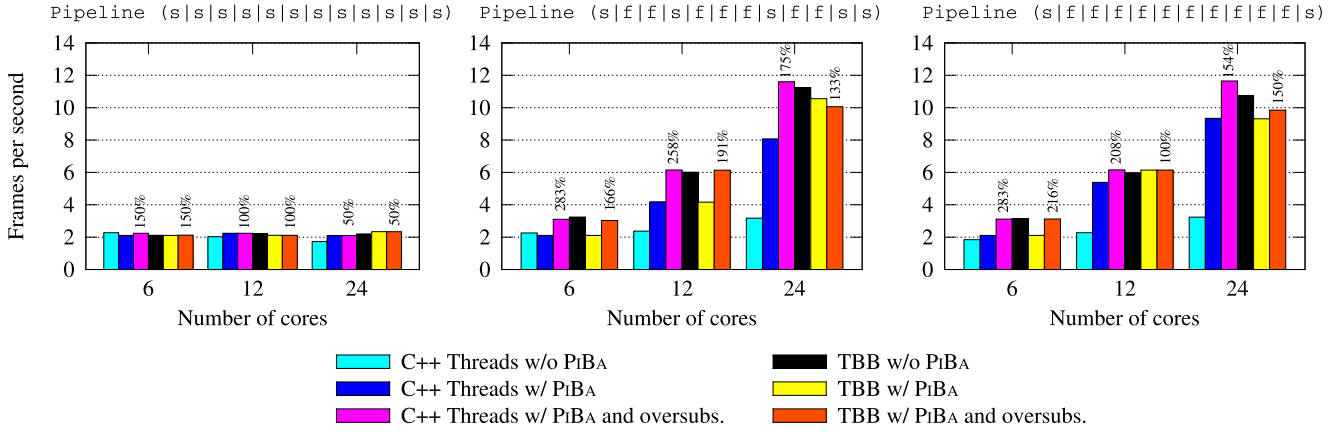


FIGURE 10. **Frames per second obtained by the different Pipeline variants of VIDEO-APP using C++ threads and Intel TBB GRPPI back ends and the basic and extended PIBA algorithms.**

It is important to remark that the main parallel pattern detected is a **Pipeline** where the image filtering stages correspond to **Farms** constructions. (Note that in this paper we do not explicitly evaluate the quality of the pattern detection performed by PPAT, as this was already analyzed in [28].) Next, we used the PPRF refactoring module to generate the parallel version of the code using the C++ threads, and the Intel TBB GrPPI back ends. Finally, we leveraged the basic and extended PiBa algorithms to improve the performance of Video-App.

Given that both filters Gaussian Blur and Sobel are pure functions by nature, we have slightly modified Video-App in order to include impure versions of these filters. This way, the **Pipeline** stages can be detected as sequential or potential **Farms**, depending on the filter version encountered. Thus, for the subsequent experiments, we have developed three different versions of the Video-App **Pipeline**, composed by 10 intermediate filtering operations, using both pure and impure versions. The **Pipeline** stages of these three Video-App variants have been arranged in the following way: i) fully sequential `(s|s|s|s|s|s|s|s|s|s|s|s)`; ii) combined `(s|f|f|s|f|f|f|s|f|f|s|s)`; and iii) fully parallel `(s|f|f|f|f|f|f|f|f|f|f|s)`. As a mere example of the steps taken by PPRF, Listing 2a shows the combined **Pipeline** variant of the Video-App original code. Next, Listing 2b shows the result after executing PPAT and generating the instrumented code. Finally, Listing 2c shows an optimized version of the Video-App parallel code according to the arrangement proposed by PiBa using the profile execution data.

1) PERFORMANCE EVALUATION

Given the foregoing, in this section, we assess the proposed basic and extended PiBa algorithms using the three variants of the Video-App **Pipeline**. These experiments have been run on 6, 12 and 24 cores using the C++ threads and Intel TBB GrPPI back ends. Figure 10 depicts the frames per second delivered by each of the Video-App versions. Focusing on the fully sequential **Pipeline** (left-hand side plot), we observe that the fact that all stages are sequential prevents PiBa from

replicating stages, and thus, the impact of the potential bottlenecks cannot be reduced. Therefore, the performance either using any of the PiBa algorithms or not is limited by the slowest stage.

Looking at the Pipeline comprised of sequential and parallel Farm stages (center and right-hand side plots), we notice a completely different behavior. Our first observation is that using the basic PiBa algorithm and C++ threads back end, the application can benefit from the stage merging and replication techniques. Basically, when two stages executed by individual threads are merged, a resource (core) becomes available and can be exploited by an additional replica (thread) in the slowest Farm stage. The PiBa benefits are more pronounced when increasing the number of cores compared with the non-balanced Pipelines. On the other hand, the results with the extended PiBa algorithm even outperform the throughput delivered by those processed with the basic PiBa version. Note that the percentages above the bars indicate the oversubscription degrees calculated by the greedy search approach. These improvements are achieved by using more threads than available cores. Therefore additional replicas can accelerate Pipeline bottlenecks by overlapping contention of the fastest stages with useful computations.

On the contrary, setting TBB as the GrPPI back end, the use of the basic PiBa algorithm negatively affects the performance, as TBB does not allow to establish the number of replicas in Pipeline stages. Indeed, TBB uses an entirely different approach for executing stages: it leverages task-parallelism with a pool of worker threads that continuously poll for work from a ready task queue [25]. Therefore, for each item in the Pipeline, a new task is created and picked by a worker thread as soon as its dependencies have been resolved. In this sense, the fact that PiBa uses stage merging leads to longer but fewer Pipeline stages and results in larger congestion. In some cases, we observe that using 24 cores, extended PiBa with the C++ threads GrPPI back end attains better performance results than using TBB. All in all, we can conclude that the extended PiBa algorithm is the recommended option when using the C++ threads GrPPI back end, while for TBB the recommended solution is not to balance the Pipelines.

2) FINE-GRAINED ANALYSIS

In this section, we complement the study with a fine-grained inspection of different Pipeline configurations to analyze their internal behavior. This study has been carried out using the C++ threads GrPPI back end instrumented with the Extrae library [34] to obtain execution traces. Afterward, the traces are visualized using the Paraver tool [35]. We only focus on the combined Pipeline comprising both sequential and parallel stages and using 12 cores. Figure 11 depicts a task trace and the number of simultaneously active threads during the execution of Video-App without having used any of the PiBa algorithms. Similarly, the left-hand side plot in Figure 14 represents the time percentage spent by the threads for the different states. Note that the colors in the trace and plot represent three different states: i) in-stage stands for the effective computation of the stages; and ii) enqueue and dequeue represent blocking states due to communications between stages via queues. As can be seen, the stages 1, 5 and 8 correspond to the slowest stages (bottlenecks), which dictate the total execution time. Also, the number of simultaneously active threads is always lower than 6, i.e., half of the available cores. Correspondingly, only half of the threads are simultaneously performing useful computations during the execution.

With the previous results, it can be observed that if the Pipeline were balanced, the resources could be better exploited, and thus, the execution time could also be improved. Given that, we make use of the basic PiBa algorithm to provide a better-balanced stage arrangement. As shown in Figure 12, the Video-App Pipeline processed by the basic PiBa algorithm generates a trace where the stages (threads) show much lower contention times. This is by the fact that the workload among stages is better balanced than if PiBa is not applied. Focusing on the right-hand side plot in Figure 14, we observe that PiBa has assigned additional replicas to those bottlenecks detected in the previous experiment (stages 1, 5 and 8). We also observe that the simultaneous active threads are, in the beginning, close to 12. As the execution progresses, and as soon as the stages complete processing the items, the number of active threads slightly decays until the end. In this case, thanks to the basic PiBa algorithm the execution time has been reduced by a factor of 30 %.

Focusing again on the trace in Figure 12, it can be seen that some of the resources are underused. This reveals an opportunity to further improve the Video-App execution time. In consequence, we leverage the extended PiBa algorithm to exploit better available resources. Figure 13 depicts the execution task trace where the ideal concurrency degree was 253 %, i.e., 31 threads running on 12 cores, according to the results in Figure 10. A first inspection of the resulting trace reveals a higher contention ratio due to queue communications. However, as the contention is shared among the fastest stages, it allows exceeding threads to exploit resources freed by such contentions (see Figure 15). On the other hand, the active threads are mostly sustained above 12 during the entire execution. Note that having more active threads than available cores results in suspended threads waiting for CPU time. In the end, the extended PiBa algorithm leads to better performance than using only the basic variant. In this concrete case, the execution time has been reduced by a factor of 60 % with respect to the non-balanced Pipeline.

D. EVALUATION OF LANE-DETECTION

In this section, we evaluate the proposed PPRF framework using a sequential real-world computer vision application able to detect road lane lines in autonomous driving systems. This stream-processing application processes individ-

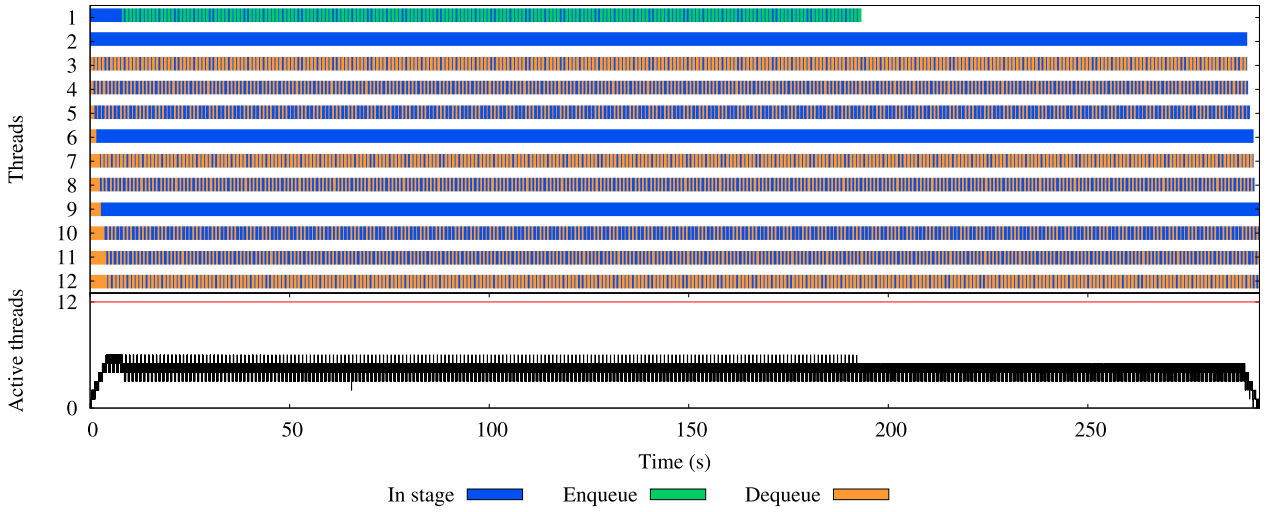


FIGURE 11. Task trace and number of active threads during the execution of the application without PIBA.

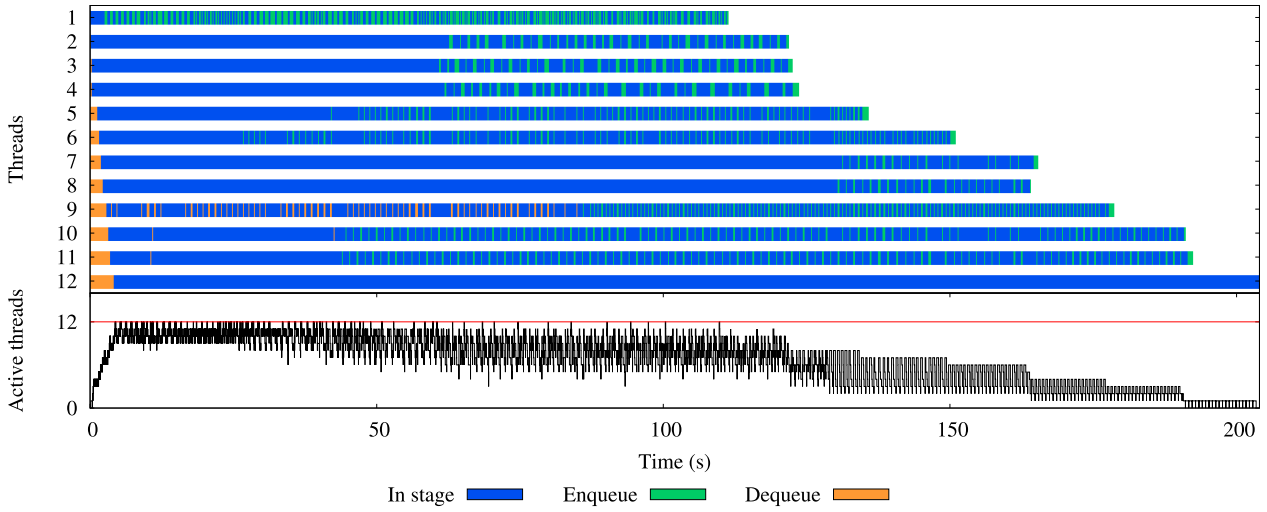


FIGURE 12. Task trace and number of active threads during the execution of the application with the basic PIBA algorithm.

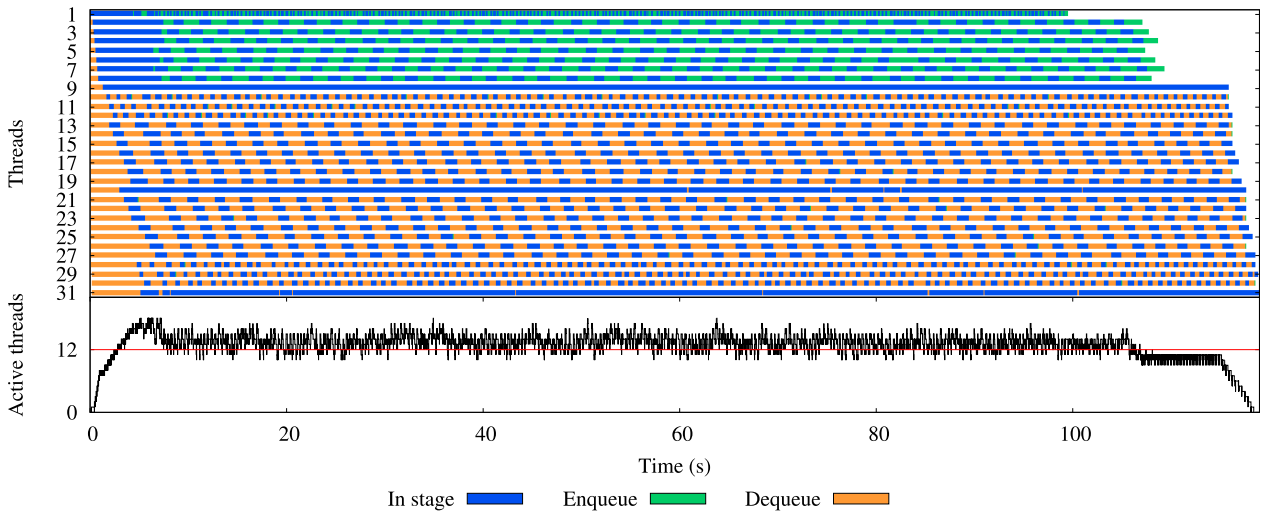


FIGURE 13. Task trace and number of active threads during the execution of the application with the extended PIBA algorithm.

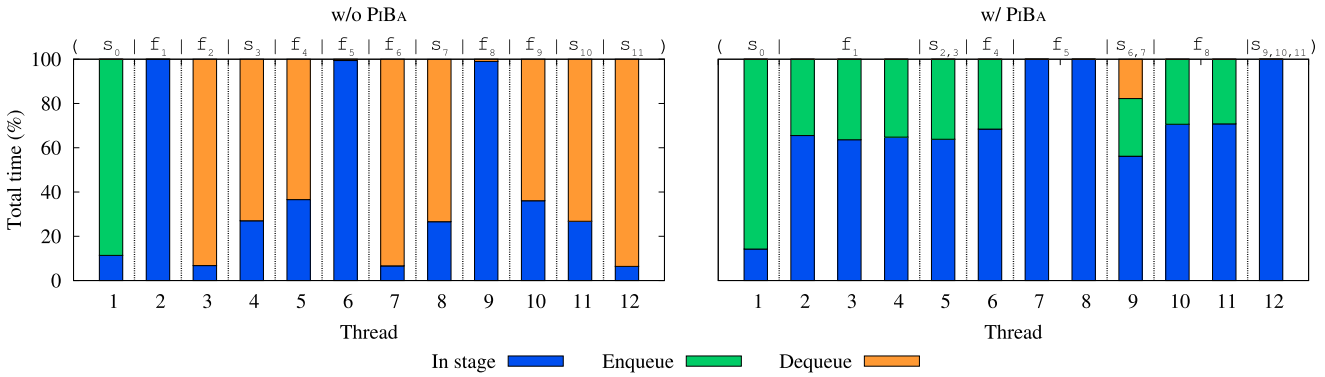


FIGURE 14. In-state time percentage per thread/stage of the VIDEO-APP Pipeline w/o and w/ the basic PIBA algorithm.

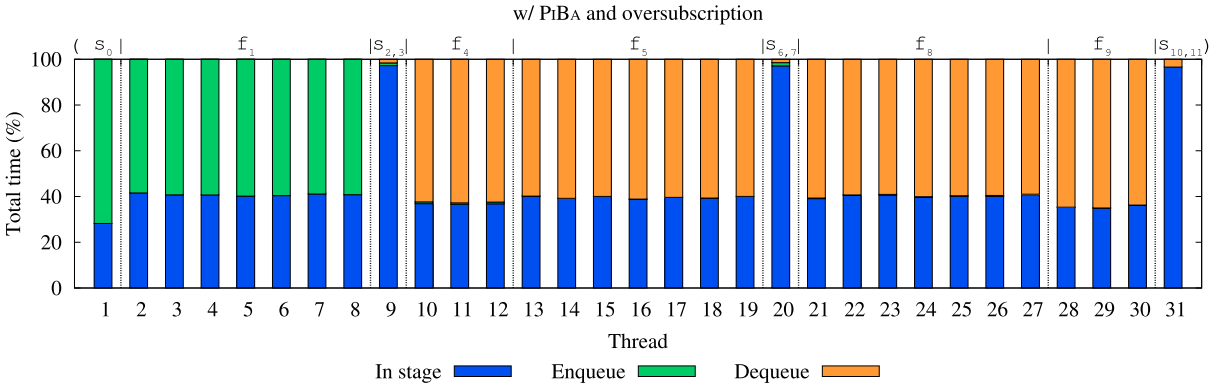


FIGURE 15. In-state time percentage per thread/stage of the VIDEO-APP Pipeline using the extended PIBA algorithm.

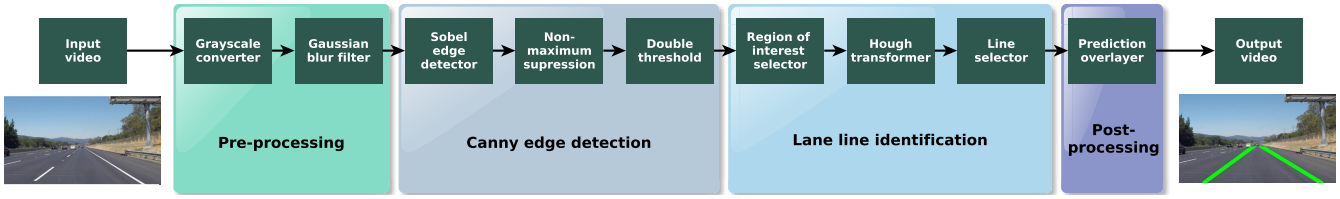


FIGURE 16. LANE-DETECTION application workflow.

ual video frames using a series of filter algorithms, such as the Canny edge detector and the Hough transform [33]. Figure 16 depicts the application workflow through an 11-staged Pipeline, where the first and last stages are processed in series, and the intermediate ones can be executed in parallel using the Farm pattern.

In a first step, we use PPRF to introduce parallel patterns in the application source code. In this case, the PPAT module detects a Pipeline with the following structure `(s|f|f|f|f|f|f|f|f|s)`, i.e., where all intermediate stages can process individual video frames in parallel. Afterward, we employ the refactoring module along with both basic and extended PiBa algorithms to evaluate the application performance using the C++ threads, and the Intel TBB GrPPI back ends. Figure 17 shows the speedup obtained by Lane-Detection when using both back ends and PiBa versions running on 6, 12 and 24 cores. As can be seen for the C++ back end, the efficiency obtained when no PiBa algorithm is on average is 22 %, while for TBB is roughly 80 %. These contrasting results are given due to the different nature of both GrPPI back ends: C++ threads back end maps Pipeline stages onto threads, while TBB handles the processing of a stage on a given item as a task which is executed by one of the worker threads in its internal scheduler pool. On the other hand, when the basic PiBa algorithm is leveraged to balance the application Pipeline, the speedup obtained by the C++ threads back end is much closer to that delivered by TBB. Although the PiBa helped in balancing the Pipeline, there remain bottleneck stages which cause congestions in the faster ones. This fact leads the C++ threads back end to slightly reduced speedups compared to TBB. Finally, although the extended PiBa algorithm does not avoid these bottlenecks, the exceeding threads help in overlapping the contention with useful computations.

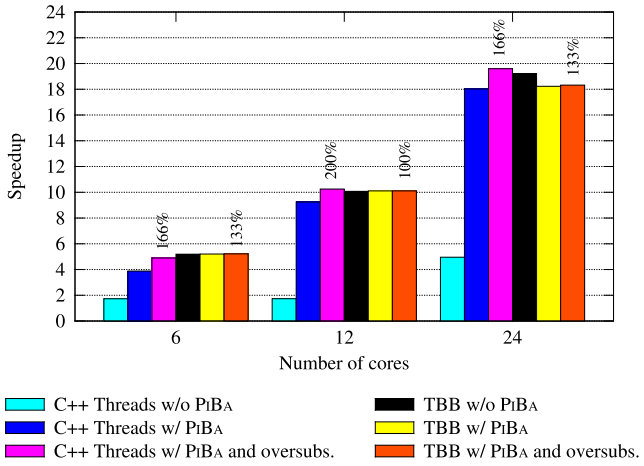


FIGURE 17. Speedup obtained by LANE-DETECTION using C++ threads and Intel TBB GRPPI back ends and the basic and extended PIBA algorithms.

Specifically, the C++ threads back end achieves comparable performance figures with respect to the TBB back end. Further, in some cases (e.g., 24 cores), the oversubscription used by the extended PiBa algorithm can even outperform the TBB performance.

In general, throughout this study, we conclude that the extended PiBa algorithm a more suitable option for balancing Pipelines, as it takes advantage of the inherent contention that this pattern may cause. The benefits of this algorithm can be summarized as follows. Firstly, the extended PiBa includes by design the basic variant, and thus, when the optimal Pipeline arrangement requires less or equal replicas than the number of cores, both approaches arrive at the same solution. Secondly, in some cases, a concurrency degree higher than the total cores might be the preferred option.

VIII. CONCLUSIONS

In this work, we have presented PPRF, a refactoring framework which proceeds in the following steps. First, it detects stream parallel patterns (Pipeline and Farm) in sequential codes thanks to PPAT tool [28]. Next, it rewrites the code related to the patterns found using GrPPI, a generic parallel pattern interface [29]. In the last step, it optimizes Pipeline patterns composed with Farm stages using PiBa, a novel profile-guided approach to improve their throughput. Finally, we extended the PiBa algorithm to improve the Pipeline resource usage by increasing the number of replicas above the total cores (oversubscription).

Throughout the analysis of the PiBa algorithm variants, we have concluded that heuristic version can provide reasonably well-balanced Pipelines in a reasonable time frame, while the brute-force and hybrid searches are able to provide slightly better stage arrangements at the expense of protracted time-to-solutions. We also have validated that both PiBa algorithms for oversubscribed scenarios deliver similar performance, although the iterative variant requires more iterations than the greedy approach.

Regarding the evaluation of the framework using the Video-App and Lane-Detection applications, we have demonstrated that PPRF is able to parallelize codes through parallel patterns automatically. During these experiments, we have proved that the extended PiBa algorithms can be good choices for the C++ threads GrPPI back end, while for TBB is better to leave the Pipeline as is. Using performance analysis tools, we have confirmed our initial hypothesis where a concurrency degree higher than the total cores might deliver better performance. In general, oversubscribed threads in these scenarios can overlap the potential bottlenecks with useful computations. However, the presented PiBa algorithms leverage profiling techniques; therefore, the input data used during the profiling phase should be representative enough to perform the optimizations. Also, if the application has an irregular input workload, the PiBa algorithms may not find the optimal Pipeline arrangement.

As future work, we plan to extend the tool for refactoring other parallel patterns in PPRF, such as Stencil or MapReduce. Furthermore, we plan to supply the framework with a decision system to select the most suitable GrPPI back end for the target architecture and to apply, when necessary, the extended PiBa algorithm for oversubscribed scenarios. An ultimate goal is to complement the balancing algorithms to target heterogeneous and distributed platforms, where the Pipeline stages can be executed in distinct physical processors.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *IEEE Solid-State Circuits Soc. Newslett.*, vol. 11, no. 3, pp. 33–35, Sep. 2006, doi: 10.1109/N-SSC.2006.4785860.

[2] V. G. Vaidya, P. Agrawal, A. Athavale, A. Sane, S. Sah, and P. Ranadive, "Increasing parallelism on multicore processors using induced parallelism," in *Proc. 2nd Int. Conf. Softw. Technol. Eng. (ICSTE)*, vol. 1, Oct. 2010, pp. V1-5–V1-8, doi: 10.1109/ICSTE.2010.5608971.

[3] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming* (Scientific and Engineering Computation). Cambridge, MA, USA: MIT Press, 2007.

[4] A. Duran *et al.*, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011, doi: 10.1142/S0129626411000151.

[5] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—First experiences with real-world applications," in *Proc. 18th Int. Conf. Parallel Process. (Euro-Par)*, Berlin, Germany: Springer-Verlag, 2012, pp. 859–870, doi: 10.1007/978-3-642-32820-6_85.

[6] S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques* (Synthesis Lectures on Computer Architecture). San Rafael, CA, USA: Morgan & Claypool, 2012, doi: 10.2200/S00340ED1V01Y201201CAC019.

[7] V. Janjic *et al.*, "RPL: A domain-specific language for designing and implementing parallel C++ applications," in *Proc. 24th Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process. (PDP)*, Feb. 2016, pp. 288–295.

[8] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," *Int. J. Softw. Eng.*, vol. 7, no. 3, pp. 73–90, Dec. 2016. [Online]. Available: http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJSE-163

[9] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, no. 9, pp. 531–551, 2010, doi: 10.1016/j.parco.2010.05.006.

[10] Z. Li, A. Jannesari, and F. Wolf, "Discovery of potential parallelism in sequential programs," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 1004–1013, doi: 10.1109/ICPP.2013.119.

[11] K. Molitorisz, T. Müller, and W. F. Tichy, "Patty: A pattern-based parallelization tool for the multicore age," in *Proc. 6th Int. Workshop Program. Models Appl. Multicores Manycores (PMAM)*, New York, NY, USA, 2015, pp. 153–163, doi: 10.1145/2712386.2712392.

[12] X. Li *et al.*, "FreshBreeze: A data flow approach for meeting DDDAS challenges," *Procedia Comput. Sci.*, vol. 51, pp. 2573–2582, Jun. 2015, doi: 10.1016/j.procs.2015.05.365.

[13] T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Softw.*, vol. 32, no. 6, pp. 44–51, Nov./Dec. 2015, doi: 10.1109/MS.2015.105.

[14] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, H. Schöner, and T. Breddin, "Paraphrasing: Generating parallel programs using refactoring," in *Formal Methods for Components and Objects*. Berlin, Germany: Springer, 2013, pp. 237–256.

[15] T. Grosser, A. Groesslinger, and C. Lengauer, "POLLY—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, p. 1250010, 2012, doi: 10.1142/S0129626412500107.

[16] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST generation is more than scanning polyhedra," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 4, Jul. 2015, Art. no. 12, doi: 10.1145/2743016.

[17] C. Brown, H. Li, and S. Thompson, "An expression processor: A case study in refactoring Haskell programs," in *Trends in Functional Programming*. Berlin, Germany: Springer, 2011, pp. 31–49, doi: 10.1007/978-3-642-22941-1_3.

[18] University St Andrews. (May 2017). *ParaFormance Technologies Advanced Tools for Supporting Multicore Software Development*. [Online]. Available: https://www.paraformance.com/

[19] K. Hammond *et al.*, "The ParaPhrase project: Parallel patterns for adaptive heterogeneous multicore systems," in *Proc. Int. Symp. Formal Methods Compon. Objects*, Torino, Italy, in Lecture Notes in Computer Science), vol. 7542, B. Beckert, F. Damiani, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Germany: Springer, 2013, pp. 218–236. [Online]. Available: http://calvados.di.unipi.it/storage/paper_files/2013_fmco11_paraphrase.pdf, doi: 10.1007/978-3-642-35887-6_12.

[20] M. Aldinucci and M. Danelutto, "Stream parallel skeleton optimization," in *Proc. 11th IASTED Int. Conf. Parallel Distrib. Comput. Syst. (IASTED/ACTA)*, Cambridge, MA, USA: MIT Press, 1999, pp. 955–962. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.9607

[21] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Proc. 1st Int. Conf. Scalable Inf. Syst. (InfoScale)*, New York, NY, USA, 2006, Art. no. 13, doi: 10.1145/1146847.1146860.

[22] K. J. Brown *et al.*, "Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, New York, NY, USA, 2016, pp. 194–205, doi: 10.1145/2854038.2854042.

[23] A. Moreno, E. Cesar, A. Guevara, J. Sorribes, and T. Margalef, "Load balancing in homogeneous pipeline based applications," *Parallel Comput.*, vol. 38, no. 3, pp. 125–139, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001566

[24] A. Moreno, A. Sikora, E. César, J. Sorribes, and T. Margalef, "HeDPM: Load balancing of linear pipeline applications on heterogeneous systems," *J. Supercomput.*, vol. 73, no. 9, pp. 3738–3760, Sep. 2017, doi: 10.1007/s11227-017-1971-4.

[25] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2009, pp. 281–290.

[26] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Load-balanced pipeline parallelism," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, New York, NY, USA, 2013, Art. no. 14, doi: 10.1145/2503210.2503295.

[27] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Adding data parallelism to streaming pipelines for throughput optimization," in *Proc. 20th Annu. Int. Conf. High Perform. Comput. (HiPC)*, Bangalore, India, Dec. 2013, pp. 20–29, doi: 10.1109/HiPC.2013.6799119.

[28] D. del Rio Astorga *et al.*, "Finding parallel patterns through static analysis in C++ applications," *Int. J. High Perform. Comput. Appl.*, to be published, doi: 10.1177/1094342017695639.

[29] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "A generic parallel pattern interface for stream and data processing," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 24, p. e4175, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4175

[30] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Reading, MA, USA: Addison-Wesley, 2004.

[31] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[32] S. Pelagatti, "Task and data parallelism in P3L," in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. Berlin, Germany: Springer-Verlag, 2003, pp. 155–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=778641.778657

[33] S. Yenikaya, G. Yenikaya, and E. Düven, "Keeping the vehicle on the road: A survey on on-road lane detection systems," *ACM Comput. Surv.*, vol. 46, no. 1, Jul. 2013, Art. no. 2, doi: 10.1145/2522968.2522970.

[34] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed performance analysis using coarse grain sampling," in *Euro-Par—Parallel Processing Workshops*. Berlin, Germany: Springer, 2010, pp. 185–198, doi: 10.1007/978-3-642-14122-5_23.

[35] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A tool to visualize and analyze parallel code," in *Proc. 187th World Occam Transputer User Group Tech. Meeting Transputer Occam Develop. (WoTUG)*, P. Nixon, Ed. Manchester, U.K.: IOS Press, Apr. 1995, pp. 17–33.

MANUEL F. DOLZ received the B.Sc. degree in computer science from the Universitat Jaume I de Castelló, Spain, in 2008, the M.Sc. degree in parallel and distributed computing from the Polytechnic University of Valencia, Spain, in 2010, and the Ph.D. degree from the Universitat Jaume I de Castelló in 2014. From 2013 to 2015, he was a Post-Doctoral Research Assistant with the Scientific Computing Group, University of Hamburg, Germany, responsible for the Exa2Green EU-Project. He is currently a Post-Doctoral Research Assistant with the Computer Architectures, Communications and Systems Research Group, Universidad Carlos III de Madrid, Spain, responsible for the RePhrase EU-Project. His main research interests are energy efficiency and programming models in the high performance computing domain.

DAVID DEL RIO ASTORGA received the B.Sc. degree in computer science and the M.Sc. degree in informatics engineering from the Universidad Carlos III de Madrid, Spain, in 2013 and 2015, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Universidad Carlos III de Madrid. His current research interests are programming models in the high-performance computing domain.

JAVIER FERNÁNDEZ received the Ph.D. degree in computer science from the Universidad Carlos III de Madrid, Spain, in 2004, with a thesis focused on O.S. support for multimedia QoS. He has been a Visiting Researcher with the EPCC Supercomputing Center, Edinburgh, U.K., in 2009, and with HLRS supercomputing center, Stuttgart, Germany, in 2011. He is currently an Associate Professor at the Computer Science and Engineering Department, Universidad Carlos III de Madrid. He has participated in five publicly funded European research projects, 18 publicly funded national and regional research projects, and 14 technology transfer contracts, several with major companies in the aerospace field such as EADS and GMV or in the railways field such as RENFE or ADIF. He has published 22 papers on journals with impact factor and over 40 papers on international conferences and workshops. His main research topics are related to high-performance computing but other topics of interest included developing, simulating, and maintaining industrial systems focusing on real time and embedded systems.

J. DANIEL GARCÍA 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 been a Visiting Researcher with the University of Modena, Italy, and a Visiting Faculty with Texas A&M University. Since 2008, he has been the Spanish Head of delegation in the ISO C++ Standards Committee where he actively participated in the development of the C++11, C++14, and C++17 standards. He is currently an Associate Professor with the Computer Science and Engineering Department, Universidad Carlos III de Madrid, Spain. He has participated in 14 technology transfer contracts and 20 publicly funded research projects, as well as many others before joining academia. He has published over 25 international journals and 45 conference papers and has edited several journal special issues. His current research interests focus on programming models for applications improvement. In particular, his aim is improving both the performance of applications (faster applications) and maintainability (easier to modify).

JESÚS CARRETERO has been a Full Professor of computer architecture and technology with the Universidad Carlos III de Madrid, Spain, since 2000. He is currently an Action Chair of the IC1305 COST Action Network for Sustainable Ultrascale Computing Systems, and he is also currently involved in the EU project ICT RePhrase: Refactoring Parallel Heterogeneous Resource Aware Applications. His research activity is centered on high-performance computing systems, large-scale distributed systems, and real-time systems. He is a member of the ACM and a Senior Member of the IEEE Computer Society.