Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite

DANIELE DE SENSI, TIZIANO DE MATTEIS, MASSIMO TORQUATI, GABRIELE MENCAGLI, and MARCO DANELUTTO, University of Pisa

High-level parallel programming is an active research topic aimed at promoting parallel programming methodologies that provide the programmer with high-level abstractions to develop complex parallel software with reduced time to solution. *Pattern-based parallel programming* is based on a set of composable and customizable parallel patterns used as basic building blocks in parallel applications. In recent years, a considerable effort has been made in empowering this programming model with features able to overcome shortcomings of early approaches concerning flexibility and performance. In this article, we demonstrate that the approach is flexible and efficient enough by applying it on 12 out of 13 PARSEC applications. Our analysis, conducted on three different multicore architectures, demonstrates that pattern-based parallel programming has reached a good level of maturity, providing comparable results in terms of performance with respect to both other parallel programming methodologies based on pragma-based annotations (i.e., OpenMP and OmpSs) and native implementations (i.e., Pthreads). Regarding the programming effort, we also demonstrate a considerable reduction in lines of code and code churn compared to Pthreads and comparable results with respect to other existing implementations.

CCS Concepts: • **Computing methodologies → Parallel programming languages**;

Additional Key Words and Phrases: Parallel patterns, algorithmic skeletons, parsec, multicore programming, benchmarking

ACM Reference format:
Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. 2017. Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 14, 4, Article 33 (October 2017), 26 pages.
https://doi.org/10.1145/3132710

1 INTRODUCTION

Multicore systems have become commonplace. We can find chip multiprocessors (CMPs) in almost any device, from wearable smart sensors to high-end servers. Although the advent of CMPs has

This article is an extension of the conference paper: "P^{3}ARSEC: Towards Parallel Patterns Benchmarking" appeared in the Proceedings of the ACM Symposium on Applied Computing [21]. We extended this work by (i) modeling and implementing an additional seven applications by using parallel patterns (there were five in Danelutto et al. [21]), (ii) implementing four of these applications with another framework (SkePU), (iii) comparing our approach to the OmpSs programming model, (iv) adding other programmability metrics to our analysis, and (v) testing our work on two additional and completely different architectures.

This work was partially supported by the EU H2020-ICT-2014-1 project RePhrase (644235).

Authors' addresses: D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto; emails: {desensi, dematteis, torquati, mencagli, marcod}@di.unipi.it.

https://doi.org/10.1145/3132710

alleviated several problems of single-core processors (e.g., the so-called memory wall [59]), it has raised the issue of the *programmability wall* [16] that previously characterized the development of parallel software targeting traditional HPC platforms.

The standard approach to program CMPs relies on thread-level parallelism where data sharing is coordinated by synchronization primitives. This approach leaves a large degree of freedom to the programmer when coding applications, allowing low-level optimizations that may increase the performance but also the lines of code (LOC), thus reducing code portability and maintainability.

To face these issues, one approach consists of using high-level *parallel patterns* that the programmer composes and nests to build parallel applications. Each pattern applies a parallelism paradigm to solve recurrent problems [48, 49]. Examples of frameworks supporting this vision are Microsoft PPL [12], FastFlow [22], SkePU [30], and Delite [10]. The main drawback of this programming model is the potential lower flexibility offered to the application developer. In fact, parts of the application that could be parallelized might not exactly match any available pattern. For these reasons, some recent parallel frameworks, such as Intel TBB [49], adopt a sort of hybrid approach, where in addition to some predefined patterns (e.g., pipeline, parallel-for, reduce, scan), they offer support to the execution of generic graphs of tasks by respecting their precedence relations.

In addition to the programmability advantage, a focal aspect is to precisely assess which is the flexibility and performance gap observed when using high-level methodologies. An interesting work that tries to provide a first answer for task-based programming models is presented in Chasapis et al. [17]. The authors showed that the OmpSs porting of a significant subset of the PARSEC benchmark suite [6] does not provide substantial performance degradation with respect to native Pthreads implementations while reducing the programming effort measured in terms of LOCs. Our contribution with this work is to provide a similar and deeper analysis for pattern-based frameworks, showing that they are at least as expressive in exposing the parallel structure of the application as the task-based or pragma-based approaches. As far as we know, this is the first attempt to provide a thorough analysis of the pattern-based methodology. Our contributions can be summarized as follows:

- We show that 12 out of 13 PARSEC benchmarks can be modeled as composition of parallel patterns. Our analysis also shows that a relatively small number of parallel patterns are sufficient to model complex real-world applications, such as the ones in PARSEC.
- We implemented all pattern-based versions of the PARSEC benchmarks using the parallel patterns offered by the FastFlow framework [22]. A subset of these benchmarks have also been implemented in SkePU [32].
- We study the programming effort required to implement the parallel versions measured using two metrics: LOC and code churn [51]. Despite that a shorter code does not necessarily imply a simpler or better code, evaluating the programming effort in an objective way is a difficult task and no existing metric is universally accepted. We decided to use LOC and code churn since they are often used in several research works as proxy metrics to evaluate programmability [17, 51, 58]. The patterned implementations achieve an average LOC reduction of 26% (in both FastFlow and SkePU) compared to the native Pthreads implementation (and up to a maximum of 87% for some specific benchmarks). We also compared such metrics with those of the OpenMP and TBB implementations provided by PARSEC and with the OmpSs implementations described in Chasapis et al. [17].
- On three different multicore systems we compare performance of the pattern-based implementations to the one of native Pthreads implementations, with OpenMP and TBB versions provided by PARSEC (when available) and with OmpSs task-based parallelizations.

The FastFlow and SkePU implementations obtain an average gain of 14% and 7% with respect to Pthreads (up to a maximum of 42%) and comparable results with respect to the other implementations.

To create a new benchmark suite for pattern-based frameworks, the source code of all implemented benchmarks is made publicly available under the name P^{3}ARSEC—Parallel Patterns PARSEC.[1]

The rest of this article is organized as follows. Section 2 introduces the PARSEC benchmark suite and the pattern-based methodology. Section 3 presents the used patterns and the pattern-based implementation of the P^{3}ARSEC benchmarks. Section 4 provides details of the performance and programmability analysis presenting experimental results. Section 5 provides the related work, and Section 6 draws the conclusions of this work.

2 BACKGROUND

In this section we introduce the background of this work. First, we describe the PARSEC benchmark suite. Then, we will provide a brief review of recent methodologies and frameworks based on pattern-based parallel programming.

2.1 The PARSEC Benchmark Suite

PARSEC [6] (Princeton Application Repository for Shared-Memory Computers)[2] is a collection of various multithreaded programs with high system requirements that has been used in the past for stressing shared-memory architectures [54]. One of the most interesting aspects of this benchmark suite is that it covers a wide set of application domains, such as streaming, scientific computing, computer vision, and data compression. For this reason, the PARSEC suite has been recently used to assess the expressive power of emerging parallel programming frameworks [17].

2.1.1 Applications Taxonomy. PARSEC consists of 13 programs from different areas of computing. Each application is provided with several input sets for each benchmark. Three datasets, with different sizes, target the execution on simulators (i.e., *sim-small*, *sim-medium*, *sim-large*), whereas the *native* dataset is representative of a realistic execution scenario of the application.

From the parallel programming perspective, PARSEC applications are of great interest for testing frameworks because they have different memory access behaviors, data sharing patterns, amounts of parallelism, computational granularity, and synchronization frequency. Table 1 reports the official name of the benchmarks, their parallelism model, and the computational grain according to PARSEC documentation. Moreover, we show the official parallel versions released within the PARSEC suite that we use as reference implementations in this work.

Most of the applications belong to the data parallelism model, where the computation is performed on large data structures logically partitioned among multiple threads. Stream parallelism characterizes applications where a large sequence of data items are processed by a chain of threads that execute distinct computation phases on different items in parallel and in a pipeline fashion. The case of `Canneal` is an example of applications that do not straightforwardly follow any common parallelism paradigm (in the table, it is referred to as *unstructured*).

[1]The code of P^{3}ARSEC is publicly available at https://github.com/ParaGroup/p3arsec. Release `v1.0` is used in this article.
[2]In this article, we refer to PARSEC version 3.0: http://parsec.cs.princeton.edu/overview.htm.

Table 1. Classification and Characteristics of the PARSEC v3.0 Applications

Benchmark	Domain	Parallelism		Parallel Versions		
		Model	*Grain*	*Pthreads*	*OpenMP*	*Intel TBB*
`blackscholes`	Financial analysis	Data parallelism	Coarse	✓	✓	✓
`bodytrack`	Computer vision	Data parallelism	Medium	✓	✓	✓
`canneal`	Engineering	Unstructured	Fine	✓	✗	✗
`dedup`	Enterprise storage	Stream	Medium	✓	✗	✗
`facesim`	Animation	Data parallelism	Coarse	✓	✗	✗
`ferret`	Similarity search	Stream	Medium	✓	✗	✗
`fluidanimate`	Animation	Data parallelism	Fine	✓	✗	✓
`freqmine`	Data mining	Data parallelism	Medium	✗	✓	✗
`raytrace`	Computer vision	Data parallelism	Medium	✓	✗	✗
`streamcluster`	Data mining	Data parallelism	Medium	✓	✗	✓
`swaptions`	Financial	Data parallelism	Coarse	✓	✗	✓
`vips`	Media processing	Data parallelism	Coarse	✓	✗	✗
`x264`	Media processing	Stream	Coarse	✓	✗	✗

2.2 Parallel Pattern-Based Approaches

Parallel design patterns have been envisioned as a viable solution to improve the quality and efficiency of parallel software development while reducing the complexity of program parallelization and enhancing performance portability [48].

Parallel patterns are schemes of parallel computations that recur in many real-life algorithms and applications. Each of them usually has one or more well-known implementations of communication, synchronization, and computation models. The use of parallel patterns in the development of applications provides several advantages both concerning time to solution and the automatic or semiautomatic applicability of different optimization strategies (e.g., like the IBMones proposed in Chambers et al. [15], Gedik et al. [34], and Navarro et al. [53]). This last aspect is usually manually enforced in non–pattern-based parallel programming models such as MPI and Pthreads. Furthermore, some research works have recently proposed autonomic management strategies of nonfunctional concerns like performance and energy consumption for pattern-based approaches [23, 47] by using control knobs like concurrency throttling and dynamic voltage and frequency scaling (DVFS) [24].

Algorithmic skeletons [18] were developed independently of parallel patterns to support programmers with the provisioning of standard programming language constructs that model and implement common, parametric, and reusable parallel schemes. Algorithmic skeletons may be considered as a practice of implementation of parallel design patterns. Combinations of parallel design patterns and algorithmic skeletons are used in different parallel programming frameworks such as Microsoft PPL [12] and Intel TBB [49], as well as in niche pattern-based research frameworks such as SkePU [32], Muesli [31], FastFlow [22], SkeTO [29], SkelCL [55], Skandium [44], and OSL [43], just to mention a few of them. Other frameworks such as Google MapReduce [25] are instead built around a single powerful pattern.

To raise the level of abstraction in parallel software development for specific application domains, some research works proposed domain-specific languages (DSLs) built on top of

pattern-based frameworks [10, 41, 56]. Their main aim is to help the domain experts to easily prototype different parallel variants of their code and to introduce parallel runtime optimizations in a more selective way. A similar approach, which leverages the new C++1x features, consists of annotating the sequential code with C++ attributes to introduce parallel patterns in specific regions of code (usually compute-intensive kernels). Then, a source-to-source compiler is responsible for translating the annotated C++ code into a parallel code linked to the proper pattern-based runtime library [20, 37].

3 PARALLEL PATTERN-BASED PARSEC

Pattern-based frameworks provide a set of parallel patterns that solve recurrent problems in parallel programming. Some notable examples are *map*, *reduce*, *pipeline*, *farm*, *divide-and-conquer*, *stencil*, and *parallel-for*. In this section, we review the parallel patterns that we have used in the development of P[3]ARSEC. Then we provide some examples of patterned code written in FastFlow and SkePU to give an idea of the interface and the programming abstractions offered by some of the existing frameworks. Finally, we describe for each PARSEC application both the original parallel design and our pattern-based one. In some cases, we outline possible alternative compositions and optimizations of patterns.

3.1 A Small Catalog of Parallel Patterns

Several past works have described parallel patterns by providing a formal semantics that allows patterns to be composed and nested according to specific rules [2, 13]. Rewriting rules have been derived to transform a pattern expression into an equivalent one (i.e., a different pattern composition that preserves the computation correctness), possibly able to achieve better performance. Such analysis and formalism is outside the scope of this article. In this part, we recall the patterns that we have used in the implementation of P[3]ARSEC. To represent the patterns, we use a synthetic syntax that simplifies the description of alternative implementation schemes.

Sequential (`seq`). This pattern encapsulates a portion of the "business logic" code of the application that can be used in this way as a parameter of other more complex patterns. The implementation requires wrapping the code in a function $f : \alpha \rightarrow \beta$ with input and output (I/O) parameter types α and β, respectively. For each input $x : \alpha$, the pattern $(\texttt{seq}\ f) : \alpha \rightarrow \beta$ applies the function f on the input by producing the corresponding output $y : \beta$ such that $y = f(x)$. The pattern can also be applied when the input is a *stream* (i.e., a sequence possibly of unlimited length of items with the same type). Let α stream be a sequence $(x_1, x_2, \ldots,)$ where $x_i : \alpha$ for any i. The pattern $(\texttt{seq}\ f) : \alpha$ stream $\rightarrow \beta$ stream applies the function f to all items of the input stream, which are computed in their strict sequential order (i.e., x_i before x_j if and only if $i < j$).

Pipeline (`pipe`). The pattern works on an input stream of type α stream. It models a composition of functions $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$ where $f_i : \alpha_{i-1} \rightarrow \alpha_i$ for $i = 1, 2, \ldots, n$. The pipeline pattern is defined as $(\texttt{pipe}\ \Delta_1, \ldots, \Delta_n) : \alpha_0$ stream $\rightarrow \alpha_n$ stream. Each Δ_i is the i-th *stage*, which is a pattern instance having input type α_{i-1} stream and output type α_i stream. For each input item $x : \alpha_0$, the result out of the last pipeline stage is $y : \alpha_n$ such that $y = f_n(f_{n-1}(\ldots f_1(x)\ldots))$. The parallel semantics is such that stages process in parallel distinct items of the input stream, whereas the same item is processed in sequence by all the stages.

From an implementation viewpoint, a pipeline of sequential stages is implemented by concurrent activities (e.g., threads) passing items through cooperation mechanisms (e.g., via shared buffers).

Task-farm (`farm`). The pattern computes the function $f : \alpha \rightarrow \beta$ on an input stream α stream where the computations on distinct items are independent. The pattern is defined as $(\texttt{farm}\ \Delta) : \alpha$ stream $\rightarrow \beta$ stream where Δ is any pattern having input type α stream and output type β stream.

The semantics is such that all items $x_i : \alpha$ are processed and their output items $y_i : \beta$, where $y_i = f(x_i)$ computed. From the parallel semantics viewpoint, within the farm the pattern Δ is replicated $n \geq 1$ times (n is a nonfunctional parameter of the pattern called *parallelism degree*), and in general, the input items may be computed in parallel by the different instances of Δ.

In case of a farm of sequential pattern instances, the runtime system can be implemented by a pool of identical concurrent entities (*worker* threads) that execute the function f on their input items. In some cases, an active entity (the *emitter* thread in FastFlow [22]) can be designed to assign each input item to a worker, whereas in other systems, the workers directly pop items from a shared data structure. Output items can be collected and their order eventually restored by a dedicated entity (a *collector* thread) that produces the stream of results.

Master-worker (`master-worker`). This pattern works on a *collection* (α collection) of type α—that is, a set of data items $\{x_1, x_2, \ldots, x_n\}$ of the same type $x_i : \alpha$ for any i. There is an intrinsic difference between a stream and a collection. Although in a collection all data items are available to be processed at the same time, in a stream the items are not all immediately available, but they become ready to be processed spaced by a certain and possibly unknown time interval. The pattern is defined as (`master-worker` Δ, p) : α collection $\rightarrow$ α collection, where Δ is any pattern working on an input type α and producing a result of the same type, whereas p is a Boolean predicate. The semantics is that the `master-worker` terminates when the predicate is `false`. Different items can be computed in parallel within the `master-worker`.

A `master-worker` of sequential pattern instances consists of a pool of concurrent workers that perform the computation on the input items delivered by a *master* entity. The master also receives the items back from the workers and, if the predicate p is `true`, reschedules some items.

Map (`map`). The pattern is defined as (`map` f) : α collection $\rightarrow$ β collection and computes a function $f : \alpha \rightarrow \beta$ over all items of an input collection whose elements have type α. The output produced is a collection of items of type β where each $y_i : \beta$ is $y_i = f(x_i)$. The precondition is that all items of the input collection are independent and can be computed in parallel.

The runtime of the `map` pattern is similar to the one described for the `farm` pattern. The difference lies in the fact that since we work with a collection, the assignment of items to the worker entities can be performed either statically or dynamically. Depending on the framework, an active entity can be designed to assign input items to the workers according to a given policy.

Map+reduction (`map+reduce`). This is defined as (`map+reduce` f, $\oplus$) : α collection $\rightarrow$ β, where $f : \alpha \rightarrow \beta$ and $\oplus : \beta \times \beta \rightarrow \beta$. The semantics is such that the function f is applied on all the items x_i of the input collection (map phase). Then the final result of the pattern $y : \beta$ is obtained by composing all items y_i of the output collection result of the map phase by using the operator $\oplus$ (i.e., $y = y_1 \oplus y_2 \oplus \ldots \oplus y_n$).

A typical implementation is the same as the map where the reduction phase can be executed serially, once all output items have been produced, or in parallel according to a tree topology by exploiting additional properties on the operator $\oplus$ (i.e., if it is associative and commutative).

Composition (`comp`). This pattern is the composition of two pattern instances that work either on single items, on streams, or on collections. In case of collections, the composition is (`comp` Δ_1, Δ_2) : α collection $\rightarrow$ γ collection, where Δ_1 is any pattern (e.g., `map` or `master-worker`) working on input α collection and that produces an output β collection, whereas Δ_2 is a pattern working with input type β collection and transforming it into a type γ collection. The semantics is that the first pattern is executed, and when its execution has finished (i.e., all items in the input collection have been computed), the second pattern can be started by processing the collection produced by the first pattern. In case of streams, the composition semantics is applied on an item-by-item basis—that is, each item in the input stream is processed first by Δ_1 and then by Δ_2 before starting to compute the next item.

The runtime system of a pattern-based framework must ensure that the two patterns within the `comp` instance are executed serially. In the case of collections, a barrier can be added after the call to the first pattern and before starting the second one.

Iterator (`iterator`). In its basic form, this pattern iterates a pattern Δ working on a single input item (`seq` or `comp`) or on a collection of items (`map`, `master-worker`). In case of collections, the pattern is defined as (`iterator` Δ, p) : α collection $\rightarrow$ α collection, where p is a Boolean predicate. The inner pattern Δ is iterated until the predicate is `true`.

At the implementation level, the runtime executes the pattern for a certain number of times determined statically or at runtime. At the end of each iteration there is an implicit barrier, since the output collection computed at iteration $i - 1$ may be used as input for the iteration i.

3.2 Examples of Parallel Pattern-Based Code

Over the past 20 years, many parallel programming models and frameworks based on parallel patterns and algorithmic skeletons have been proposed. In Gonzalez-Velez and Leyton [35], a review of several of them can be found. Some of these frameworks, such as P^{3}L [5], ASSIST [57], and SAC [36], provide a new language used to introduce pattern abstractions already in the early phases of the software development process. More recent approaches like FastFlow [22], SkePU [32], GrPPI [26], SPar [37], and PACXX [38], rely on new features of modern C++ language. Patterns are introduced by instantiating class objects at any place in the code or by using suitable C++11 attributes as in SPar. In this work, we decided to use FastFlow and SkePU.

FastFlow [22] is a C++11 header-only template library that allows the programmer to build directed graphs of streaming computations. It provides the application programmer with a variety of ready-to-use stream- and data-parallel patterns than may be freely composed and customized to implement complex parallel applications. The patterns provided are *pipeline*, *farm*, *map*, *map+reduce*, *master-worker*, *feedback-loop*, and *sequential*. Patterns are implemented with threads that communicate by using nonblocking lock-free synchronization, enabling efficient processing in high-throughput streaming scenarios [3]. Parallel patterns can be used by instantiating proper objects from the FastFlow classes. The framework has been originally designed to target shared memory multi/many cores with two main goals in mind: performance and programmability.

SkePU [30] provides a multibackend framework for heterogeneous parallel systems. The framework is composed by a source-to-source compiler and a runtime library. Data-parallel patterns are implemented as C++ objects whose instances with their I/O arguments are called *skeletons*. As the SkePU compiler recognizes a C++ construct that represents a data-parallel skeleton, it can rewrite the source code and generate backend-specific versions of the user functions to execute the skeleton on the selected backend. SkePU version 2.0 provides backends for sequential C++, multicore OpenMP, GPU with CUDA, and OpenCL. The following patterns are provided: *Map*, *Reduce*, *MapReduce*, *MapOverlap*, and *Scan*.

The `iterator` pattern is not natively provided by these frameworks. However, by knowing that the pattern is iterated, we can still exploit this design information to optimize the code, such as by keeping the threads alive between two successive iterations of the pattern instead of destroying and creating them at each iteration.

Overall, we decided to use these frameworks because both of them are well-known and currently maintained projects. In addition to this, FastFlow offers all required patterns (i.e., stream- and data-parallel ones), whereas SkePU represents a valid alternative for data-parallel skeletons. Implementations of the PARSEC benchmarks with other pattern-based frameworks are left as possible future work.

As examples of parallel pattern–based code, in the following we present the implementation of two PARSEC benchmarks: *(i)* `Ferret`, which is a stream-parallel benchmark, and *(ii)* `Swaptions`,

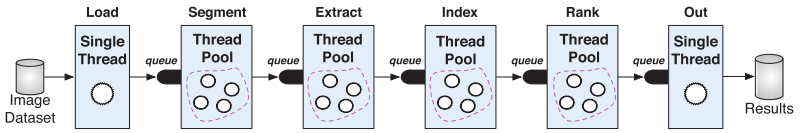

Fig. 1. General scheme of the Ferret pipeline.

```
1  // fist stage
2  struct Load: ff_node_t<long,load_data> {
3      load_data *svc(long*) { <business logic code> };
4  } In;
5  // second stage
6  struct Segment:ff_node_t<load_data,seg_data> {
7      seg_data *svc(load_data *in) { <business-logic code> };
8  };
9  // third stage
10 struct Extract:ff_node_t<seg_data,extr_data> {
11     extr_data *svc(seq_data *in) { <business-logic code> };
12 };
13 // fourth stage
14 struct Index:ff_node_t<extr_data,vec_query_data> {
15     vec_query_data *svc(extr_data *in) { <business-logic code> };
16 };
17 // fifth stage
18 struct Rank:ff_node_t<vec_query_data,rank_data> {
19     rank_data *svc(vec_query_data *in) { <business-logic code>};
20 };
21 // sixth stage
22 struct Output:ff_node_t<rank_data> {
23     void *svc(rank_data *in) { <busines-logic code> };
24 } Out;
25 // creating the pipeline with farm workers
26 ff_Pipe<> pipe(In,
27                make_Farm<Segment,n>(),
28                make_Farm<Extract,n>(),
29                make_Farm<Index,n>(),
30                make_Farm<Rank,n>(), Out);
31 // pipeline execution
32 pipe.run_and_wait_end();
```

Listing 1. FastFlow implementation of the `ferret` benchmark.

which is a data-parallel benchmark. The first one is implemented using the FastFlow streaming patterns, whereas `Swaptions` has been implemented with the SkePU `map` pattern.

As sketched in Figure 1, `Ferret` can be modeled as a single pipeline pattern of six stages where the first and last one are intrinsically sequential, whereas the other four stages are internally concurrent. Listing 1 shows the FastFlow parallel code. The business logic code of each pipeline stage is encapsulated in a sequential FastFlow node (`ff_node_t`) by implementing the `svc` method (a *pure* virtual method of the `ff_node_t` class). Then each node is added to the `ff_Pipe` pattern, respecting the pipeline order (lines 26 through 30). The four middle stages are further parallelized using the `farm` pattern created with the utility function `make_Farm`, which creates *n* replicas of the sequential `ff_node_t` passed as a template parameter.

Listing 2 shows the code of the SkePU version of `Swaptions`, which has been parallelized with a single `map` pattern. In this case, I/O SkePU smart data containers need to be created from the existing data structures (lines 5 and 6). Then the `map` object is created by providing the map function that encapsulates the business logic code for the computation of the single element of the input data collection (line 8). If needed, a specific backend runtime and a parallelism degree can be selected for

```
1  // map function working on the single item of the input collection
2  MapOutput mapFunction(skepu2::Index1D index, parm elem) { <business-logic code>};
3
4  // preparing the input and output data structures
5  skepu2::Vector<parm> swaptions_sk(swaptions, nSwaptions, false);
6  skepu2::Vector<MapOutput> output_sk(nSwaptions);
7  // creating the map object by providing the function to compute
8  auto map = skepu2::Map<1>(mapFunction);
9  // setting up the OpenMP backend and the number of threads to use
10 auto spec = skepu2::BackendSpec{skepu2::Backend::Type::OpenMP};
11 spec.setCPUThreads(nThreads);
12 map.setBackend(spec);
13 // map execution invocation
14 map(output_sk, swaptions_sk);
```

Listing 2. SkePU implementation of the Swaptions benchmark.

the map pattern (lines 10 through 12). Finally, the data-parallel computation is executed, inserting in the output data collection the computed results (line 14).

3.3 P^{3}ARSEC Parallel Implementations

Starting from the Pthreads implementations available in the PARSEC suite, we designed and implemented a parallel version of each application by composing and nesting the patterns described in Section 3.1. To provide an immediate view of the patterned scheme, we use the syntax introduced in Section 3.1. Although this description exactly matches the structure of the implementation in most of the applications, for other complex benchmarks the executed patterns depend on conditions evaluated at runtime. In those cases, the description has been simplified by focusing on the most important computational kernels. The exact structure can be found in the P^{3}ARSEC source code.

Furthermore, some of the PARSEC applications have a quite complex structure and semantics that often exploits lock-based synchronizations. To be conservative in the porting of such applications, in some cases we maintained the lock primitives that cannot be easily eliminated in the sequential portions of code passed as input parameter to the patterns instantiation.

Blackscholes. This application belongs to the Intel RMS benchmark suite [28] (Recognition, Mining, and Synthesis). It performs pricing for a portfolio of European options by numerically solving the Black-Scholes partial differential equations [7]. The Pthreads implementation divides the portfolio into work units, one for each available thread. Then each thread calculates the prices for the options in its work unit. This algorithm is iterated multiple times to obtain the final estimation of the portfolio. This benchmark is an iterative data-parallel computation. We model it as an iterator pattern where the internal pattern is a map whose input is the collection of items composing the portfolio. The pattern scheme is therefore

```
iterator(map).
```

Bodytrack. This application is aimed at tracking the body pose of a human subject by analyzing videos collected by multiple cameras. A *frame* contains one *image* from each camera. Bodytrack has basically two phases that are executed for each frame. In the first phase, three kernels are executed for each image. After this phase, two additional kernels are applied a number of times on the frame. Before applying a kernel, we need to ensure that the previous kernel is terminated. Accordingly, we can exploit parallelism only within each kernel.

The Pthreads version is implemented by using a thread pool, which can execute different kernels. The execution starts in the main thread and, for each frame, when a kernel needs to be executed, the main thread sends a command to the pool with an identifier corresponding to the

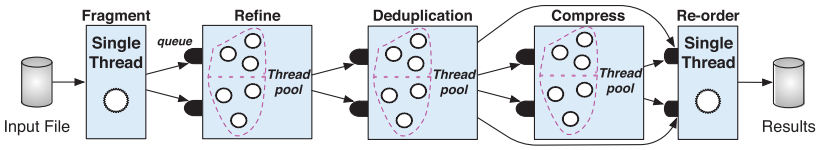

Fig. 2. General scheme of the Dedup pipeline.

kernel type. The threads in the pool will then start to process chunks of the frame with the specified kernel. To keep the load balanced, the chunks are not statically partitioned. Each thread, after the processing of the current chunk, accesses a shared variable (using a lock) to get the identifier of the next chunk and updates that variable.

In our pattern-based implementation, we remove the thread pool, parallelizing each kernel as a `map`. During the execution, every time a kernel is found the corresponding `map` is executed. Within the pattern runtime, load balancing is achieved by using a dynamic scheduling policy without any synchronization among the workers of the map. The structure of the benchmark is the following:

```
iterator(iterator(map1; map2; map3); iterator(map4; map5)).
```

To simplify the notation, we use the "`;`" symbol to represent the `comp` pattern. As an example, the syntax `map1;map2;map3` is a shortcut to write `comp(map1,comp(map2,map3))`. Furthermore, it is possible that between the composition of two patterns, some piece of plain sequential code is executed after the completion of the first pattern and before starting the second one. In the sequel, the presence of sequential code regions between the composition of two parallel patterns will be considered implicit with the "`;`" symbol.

`Canneal`. The application minimizes the routing cost of a chip design. The algorithm applies random swaps between nodes and evaluates the cost of the new configuration. If the new configuration increases the routing cost, the algorithm performs a rollback step by swapping the elements back. Although the evaluation of the elements to be swapped can be performed in parallel, swaps are executed atomically through a CAS instruction (compare-and-swap). After each iteration, a convergence condition is checked and eventually the benchmark is terminated. The workload is memory intensive because the resulting memory accesses are irregular and not easily cacheable.

The Pthreads version follows an unstructured interaction model among threads that execute atomic instructions on shared data structures. At the end of each iteration, a barrier is executed and each thread checks the termination condition.

We model this application as a single `master-worker` pattern, where the workers are sequential pattern instances executing the swaps, the evaluation, and eventually the rollback actions. At the end of each iteration, the workers notify the master, which in turn (i) implements the barrier between two iterations by waiting all notifications by the workers, (ii) evaluates the termination condition, and (iii) (re)starts the workers' computation if the condition is false.

`Dedup`. Dedup is a streaming application that compresses a data stream with a combination of global and local compression phases called *deduplication*.

The Pthreads version implements a pipeline with five stages, where each middle stage is implemented with a thread pool (the first and last stages are single threaded). To lower the contention on communication channels, cooperation between two consecutive stages is implemented using multiple queues of fixed size. Each queue is assigned to a subset of threads in the same pool. Figure 2 shows a representation of the `Dedup` pipeline. Interestingly, results out of the third stage may be transmitted directly to the last stage, bypassing the fourth stage. Furthermore, the second stage can generate more output items per input item.

The first stage (*Fragment*) reads the data stream from the disk and then partitions the data at fixed positions, and then it produces a stream of data chunks in output. Each chunk can be processed independently from the other chunks. The second stage (*Refine*) further partitions the input chunk into smaller fine-grain chunks generating a nested stream. The third stage (*Deduplication*) checks if the chunk has already been compressed in the past by accessing a hash table. If so, the chunk is marked as *duplicate*. The fourth stage (*Compress*) compresses all chunks that are not marked as *duplicate* and updates the corresponding table entries. To ensure correctness in the access to the table performed by the *Deduplication* and the *Compress* stages, each bucket in the hash table is protected with a *lock*. Finally, the *Re-order* stage writes the final compressed output data into the output file. If the input chunk was marked as *duplicate*, it stores a "reference" to the corresponding chunk. This stage reorders the data chunks as they arrive to match the original order of the uncompressed data, and it represents the main bottleneck of the `Dedup` pipeline, both due to data reordering and to I/O.

The `Dedup` benchmark can be modeled using different nestings of `pipe` and `farm` patterns. The composition is possible even though some of the stages keep an internal state that is accessed concurrently. Such state is lock protected using the same schema used in the native Pthreads implementation. The first solution is the one with a structure closest to the original Pthreads implementation. We model the application as a pipeline, where the first stage and the last stage are `seq` patterns, whereas the three middle stages are instances of the `farm` pattern. We implement the bypassing mechanism between the *Deduplication* stage and the *Compress* stage by adding a flag to each data element. The flag is set if the data element must be transmitted directly to the last stage. In that case, the *Compress* stage only forwards the element to the final stage without any further processing. The synthetic scheme of this parallelization is the following:

```
1. pipe(seq1, farm(seq2), farm(seq3), farm(seq4), seq5).
```

By using well-known rules about `farm` and `pipe` pattern compositions that preserve the semantics [2], we can also provide an alternative implementation described as follows:

```
2. pipe(seq1, farm(pipe(seq2, seq3, seq4)), seq5).
```

As we can see, all middle stages can be replicated within a `farm` pattern (i.e., each farm worker is a nested pipeline of three sequential stages). Alternatively, we can execute the stages of the inner pipeline sequentially by replacing the `pipe` with a `comp`, thus obtaining

```
3. pipe(seq1, farm(seq2; seq3; seq4), seq5).
```

Finally, it is possible to derive a fourth version that exploits a specialization of the `farm` pattern available in some frameworks (e.g., FastFlow). The `ofarm` pattern is a `farm` that preserves I/O ordering. When available, the use of this pattern allows lightening of the computational burden to the last stage (denoted by `seq5'`), which now will just write the already ordered results on disk:

```
4. pipe(seq1, ofarm(seq2; seq3; seq4), seq5').
```

Section 4 will show a comparison among these different versions.

`Facesim`. Facesim is an Intel RMS application simulating the motion of human faces. It applies the iterative Newton-Raphson algorithm over a sparse matrix. At every timestep, different kernels are executed on a mesh (some kernels are executed multiple times within a single timestep).

The Pthreads version uses a thread pool that, at every timestep, executes different kernels on the mesh. Every time a kernel is found during the execution, it is executed by the thread pool, where each thread works on a statically assigned portion of the mesh.

In our pattern-based design, each kernel is parallelized with a map pattern. We report only a synthetic view of the overall structure of the application, as there are 19 different map kernels, some of them repeated multiple times at a single timestep. We focus on the seven most time-consuming kernels (the remaining 12 map kernels are parallel operations on arrays invoked multiple times during the execution):

```
iterator(map1; map2; map1; map3; map4; map2; iterator(map5; map6; map7);
                                    map1; map4; map2).
```

Ferret. Ferret is based on a toolkit used for content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes [45]. The toolkit is configured for image similarity search.

The Pthreads parallel implementation decomposes the application into six pipeline stages. The first and last stages are single threaded, whereas the other stages are configured with a thread pool each. Communication channels between pools are implemented using queues of fixed size. The Ferret pipeline does not have bypassing links as in Dedup (see Figure 1).

We model the application as a pipe pattern. Differently from Dedup, all stages access only private data. The four middle stages are instances of the farm pattern, whereas the first and last stage, in charge of I/O operations, are seq instances. As for Dedup, we identified three possible nested schemes of patterns:

```
1. pipe(seq1, farm(seq2), farm(seq3), farm(seq4), farm(seq5), seq6) 2.
pipe(seq1, farm(pipe(seq2, seq3, seq4, seq5)), seq6) 3. pipe(seq1, farm(seq2;
                                    seq3; seq4; seq5), seq6).
```

Moreover, seq1 is actually composed by two phases: seq1.1, which iterates over the files in the input folder, and seq1.2, which for each file in the folder loads the image contained in the file in the main memory. Since seq1.2 can be performed in parallel over different files, we can move it inside the farm.[3] This leads to the following patterns' composition:

```
4. pipe(seq1.1, farm(seq1.2; seq2; seq3; seq4; seq5), seq6).
```

Additionally in this case, in Section 4 we will show a comparison among such patterned schemes.

Fluidanimate. Fluidanimate is another Intel RMS benchmark that uses an extension of the smoothed particle hydrodynamics method to simulate an incompressible fluid. At every timestep, the application executes nine kernels to compute the position of the fluid particles at the next timestep. As in other benchmarks, the sequence of kernels is sequential, and parallelism can be safely exploited within each kernel region.

In the Pthreads implementation, the 3D space is statically divided among the threads. Each thread applies each kernel on its space partition. A barrier is executed by all threads between two successive kernels.

In our pattern-based implementation, we design each kernel as a map pattern. Since the kernel sequence is iterated a number of times (one for each timestep), the overall structure can be represented as follows:

```
iterator(map1; map2; ... ; map9).
```

Freqmine. Freqmine is a data mining program that finds the most frequent items within a transactional dataset. It is based on the frequent pattern tree data structure and executes the frequent pattern growth algorithm [39]. This data mining application uses a compact tree data structure to

[3]The same technique can be applied to the other two alternative pattern compositions.

store information about frequent patterns of the transaction database. Seven kernels are identified in the application, where the last kernel is executed multiple times.

The Pthreads parallelization is not present in PARSEC, whereas the standard version is an OpenMP one. Each kernel is parallelized using the OpenMP 2.0 parallel-for construct.

In our version, each kernel corresponds to a `map` pattern—the last one iterated a number of times:

```
map1; map2; ... ; map6; iterator(map7).
```

Raytrace. This application consists of a graphical render aimed at generating animated 3D scenes by using a hierarchical grid raytracing algorithm. A kernel is executed at each frame.

In the Pthreads version, the kernel is parallelized by partitioning the 3D scene among the threads. The work is dynamically partitioned, and similarly to the `Bodytrack` Pthreads implementation, once a thread finishes to process a partition, it gets another one to keep the load balanced.

The application can be modeled as a `map` iterated a fixed number of times. Differently from `Blackscholes`, the computation is extremely unbalanced and a good dynamic scheduling of the partitions is of great importance. Furthermore, the computational weight of each `map` iteration is low while the number of iterations is high. The patterned scheme can be expressed as

```
iterator(map).
```

`Streamcluster`. Streamcluster is an application that solves the online clustering problem over incoming streaming data. The program consists of a sequence of loops whose iterations can be executed in parallel. Different loops are executed sequentially by using barriers, and they are interleaved by serial regions of code whose length impacts the overall speedup.

The computational kernel consists of two phases. The first iterates a composition of a `map+reduce` and a number of `map` instances (which in turn are iterated multiples times). The second phase, working on different data, repeats the same steps exactly one time. The simplified patterned structure can be expressed as follows:

```
// Phase 1: iterator(map+reduce; map1; iterator(map2; map3; map4);
iterator(map5; map6; map7)); // Phase 2 map+reduce; map1; iterator(map2; map3;
map4); iterator(map5; map6; map7).
```

`Swaptions`. This application is based on the Heath-Jarrow-Morton (HJM) method [40] to price a portfolio of financial options.

The Pthreads parallel version divides the data structures of the program into blocks equal to the number of threads and assigns one block to each thread. The threads are in charge of applying the method on the options within their partition.

This benchmark has a simple structure that can be modeled as a single `map` pattern, where the input is a collection of items representing the swaptions portfolio.

`Vips`. Vips is based on the VASARI Image Processing System [46] and includes basic image processing kernels such as affine transformations and convolutions. This benchmark is a domain-specific runtime system that can be used for image manipulation.

In the Pthreads version, the user specifies a function to get the next partition. Each thread executes a loop, where at each iteration (i) it gets a new partition of the image by calling the function specified by the user, (ii) the partition is processed by using another function specified by the user, and (iii) the end of the processing on the current partition is notified to the main thread by using a POSIX semaphore. The main thread calls a user-defined function at each notification.

Although it may look like a data-parallel computation, `Vips` can also be modeled as a stream-parallel computation. Indeed, since the function to get the next image partition is specified by the

user, we cannot access the entire image at once and decide how to partition it. For this reason, we model this benchmark as a `pipe`, where the first stage is a farm where each worker retrieves a partition and processes it by using the functions provided by the user. The last stage of the pipeline is sequential and calls the progress function specified by the user. The structure is expressed as follows:

```
pipe(farm(seq1), seq2).
```

X264. This application has been considered stream parallel, although it has a complex structure and interaction among its stages. In Dios et al. [27], the authors have presented this application as a wavefront algorithm instead of a stream-parallel one. In P^{3}ARSEC, we do not implement this application, as, due to its complexity, is not possible to easily separate the parallelism management from the functional code. Moreover, in addition to requiring domain-specific knowledge, this application cannot be easily expressed by using the available patterns.

4 EXPERIMENTS

The new suite P^{3}ARSEC is provided as an extension of the original PARSEC suite and can be executed with the same tools used to run the native suite (e.g., the `parsecmgmt` tool). All benchmarks have been implemented in FastFlow [22], whereas some data-parallel applications also have a SkePU [32] implementation. We verified the correctness of all implemented benchmarks with the corresponding original sequential and Pthreads implementations.

In the analysis, we focus both on the programming effort and the performance achieved. The comparison is made with the parallel versions already available in PARSEC (Table 1) and with the task-based implementations written in OmpSs and presented in Chasapis et al. [17].[4] Their work covers most of the PARSEC applications, except `Raytrace` and `Vips`. For `x264`, the authors provided an implementation that maps one to one the Pthreads version (i.e., thread creations are replaced with task spawns and thread joining with task waiting). Results in terms of performance and code complexity are the same as the Pthreads version and are not reported in the remaining part of this section. Furthermore, the authors declared a performance improvement compared to Pthreads up to 42% in `Bodytrack` and `Dedup`. By studying their implementations, we found that this advantage is mainly due to some optimizations and code rewriting that changed the PARSEC sequential semantics (i.e., their output is different from the one produced by the original sequential and Pthreads versions). To be more precise, in their implementation of `Bodytrack`, consecutive frames are processed in parallel, whereas according to the algorithm semantics, the parallelism can be exploited inside a frame but not between frames, as the computation of a frame depends on the result of the previous one. This produces an output that is different from the original one. In `Dedup`, the output produced by the OmpSs version is not deterministic, and the `Dedup` decompressor (provided with the original PARSEC benchmark) is not able to decompress it. Since we want to strictly preserve the original semantics of the applications, we do not consider these implementations. In our evaluation, we decided to not modify the original reference implementations (Pthreads, OpenMP, and TBB) since the purpose of this work is not to optimize the PARSEC benchmarks but to show that they can be parallelized using parallel patterns obtaining similar performance figures with lower LOC and lower code churn.

In the following, we first evaluate the programming effort and then the performance results.

[4]We would like to thank the authors for making their source code publicly available at https://pm.bsc.es/gitlab/benchmarks/parsec-ompss. (At the time of writing this article, commit `ea319e57` was the most recent one.)

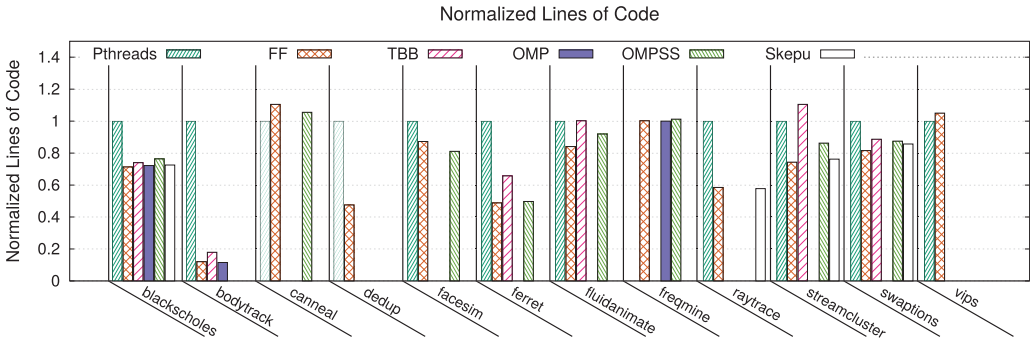

Fig. 3. LOC of the different parallel implementations, normalized between 0 and 1 with respect to the Pthreads version (the lower the better).

4.1 Programming Effort

To analyze the programming effort required to parallelize each benchmark with different parallel programming approaches, we use LOC and code churn as metrics. Evaluating the programming effort in an objective way is a difficult task, and no universally accepted metrics exist. We decided to use the LOC and code churn metrics since they are often used as proxy metrics to evaluate programmability [17, 51, 58].

Lines of code. This metric is commonly used in software engineering to measure code and programming complexity [58]. For each benchmark, we considered only the source files required to implement the parallelization or those modified during the parallelization (the other files are the same in all versions). These files include the definition of data structures used for thread communications, synchronization mechanisms, and the files containing calls to the different parallel programming frameworks. To have a fair comparison, these files have been normalized by formatting them according to a fixed programming style (brackets on the same line of the statement, single-line `if`, etc.). After that, we removed empty lines, comments, and sections of code that are not executed due to inactive macros. The measures have been normalized with respect to the Pthreads version (i.e., Pthreads is always 1, and a value greater than 1 means more LOC and lower than 1 means fewer LOC).

Code churn. A useful metric to estimate software complexity is the *code churn* [51, 52], defined as the number of lines modified and added with respect to a previous version. In our case, we consider the code churn of each parallel version with respect to the original sequential code. Starting from the sequential code, two different parallel implementations may have a similar number of code lines. However, if an implementation needs to modify and introduce a higher number of lines, this means that the effort required is likely higher than the one needed to implement the other versions. This metric is computed on the files normalized with the same process described for LOC.[5]

Discussion. We analyze the two metrics over all benchmarks and over all parallel versions. The results are shown in Figures 3 and 4. Note that when a bar is missing in the plot, it means that the implementation with the corresponding framework is not available.

[5]For reproducibility of results, we provide the script used to compute the metrics in the P[3]ARSEC repository under the `scripts/` folder.

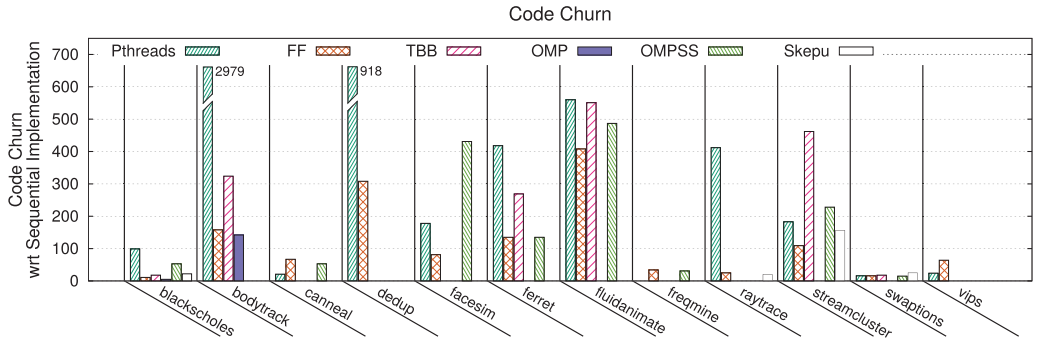

Fig. 4. Code churn (i.e., number of modified and added LOC) of the different parallel implementations with respect to the original sequential implementation (the lower the better).

On `Freqmine` and `Swaptions`, there are no particular differences between the implementations. In `Canneal`, FastFlow and OmpSs versions have a slightly higher code churn, as Pthreads code is very similar to the sequential one (the same functional part is executed by n threads).

Concerning `Blackscholes`, `Bodytrack`, `Facesim`, and `Raytrace`, the Pthreads implementation has an higher LOC and code churn because of thread pools implementations in the different benchmarks, which for `Blackscholes` is simply a wrapping of Pthreads calls to simplify threads management. In `Blackscholes`, all other implementations are equivalent, with OmpSs having a slightly higher code churn. The TBB implementation of `Bodytrack` has around double the code churn of FastFlow and OpenMP. This happens because our FastFlow implementation widely exploits C++ *lambda expressions* that simplify code development. However, the TBB version available in the PARSEC suite does not exploit this C++ feature, forcing the programmer of the TBB version to move and rewrite code, which would not have been necessary if lambda were used. As discussed earlier, we did not change the code of the reference applications since this is not the purpose of this work. In `Facesim`, despite that the LOC of the FastFlow version is slightly higher than OmpSs, the code churn of OmpSs is much higher, as the FastFlow version modified only a minimal part of the sequential code.

In the FastFlow versions of `Dedup` and `Ferret`, we implemented different pattern compositions. We show the metrics of the version characterized by the best performance (discussed in Section 4.2). The other alternative FastFlow versions have similar measures. For both `Dedup` and `Ferret`, the FastFlow code has significantly lower LOC and code churn, as the Pthreads version also needs to implement the threading support and all data structures required to let the threads communicate and synchronize with each other. Furthermore, in `Dedup`, the advantage is even more significant since we were able to remove all code lines related to data reordering, which in our case is implicit in the `ofarm` pattern (pattern composition number 4.). The TBB code of `Ferret` is slightly longer, and more lines have been modified.

The Pthreads version of `Fluidanimate` has a higher LOC and code churn because of a hand-written synchronization primitive (a spin-wait barrier) that has been implemented and used to separate the different parallel kernels. This is not needed in FastFlow, as this is implicit at the end of the `map` pattern. The OmpSs implementation has a higher code churn as well due to a routine that is used to create a data structure used to enforce nontrivial dependencies between the parallel tasks. The TBB code has a higher code churn due to the specific parallelization design.

In `Streamcluster`, the pattern-based implementations (FastFlow and SkePU) have the lowest LOC and code churn. These metrics are higher for Pthreads since also in this case a spin-wait

Table 2. Multicore Machines Used in the Performance Evaluation

Name	Description	Configuration
Intel Xeon	Dual-socket NUMA machine with two Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40GHz featuring 24 cores (12 per socket). Each hyperthreaded core has 32KB private L1, 256KB private L2, and 30MB of L3 shared with the cores on the same socket. The machine has 64GB of DDR3 RAM.	Linux 3.14.49 x86_64 shipped with CentOS 7.1. Available compiler gcc version 4.8.5.
Intel Xeon Phi	Machine with the Intel Xeon Phi model 7210 (codename Knights Landing, KNL). The KNL is equipped with 32 tiles (each with two cores) working at 1.3GHz, interconnected by an on-chip mesh network. Each core (4-way hyperthreading) has 32KB L1d private cache and a L2 cache of 1MB shared with the sibling core on the same tile. The machine is configured with 96GB of DDR4 RAM with 16GB of high-speed on-package MCDRAM configured in cache mode.	Linux 3.10.0 x86_64 shipped with Centos 7.2. Available compiler gcc version 4.8.5.
IBM Power 8	Dual-socket IBM server 8247-42L with two Power 8 processors each with 10 cores (total 20 cores) working at 3.69GHz. Each core (8-way SMT) has private L1d and L2 caches of 64KB and 512KB, a shared on-chip L3 cache of 8MB per core. The machine has 64GB of RAM.	Linux 4.4.0-47 ppc64 shipped with Ubuntu 16.04. Available compiler gcc version 5.4.0.

barrier implementation is provided. OmpSs has a higher code churn as well due to the rewriting of some processing routines but also to the introduction of additional parallelizations of some sections with respect to Pthreads and FastFlow implementations.

In `Vips`, the FastFlow version has a slightly higher LOC and code churn (approximately 20 lines). This happens because this benchmark is a framework that can be customized with code specified by its users. It has been designed to be parallelized with Pthreads and has some stringent constraints and assumptions on the code provided by its users. However, being able to design a different parallelization by only modifying a few tens of LOC while still preserving the same design and semantics is an important result.

4.2 Performance Evaluation

In the following, we describe the performance results achieved on three different multicore architectures. They are described in Table 2.

Experimental settings. In all three architectures, we used FastFlow version `2.1`, SkePU version 2, and OmpSs version `16.06.3`. The source codes of all parallel versions have been compiled with the `-O3` flag.[6] For the evaluation, we used the PARSEC *native* input set to obtain results representative of real-world program executions. The `parsecmgmt` tool has been used for launching the original PARSEC benchmarks and the FastFlow and SkePU implementations. For the OmpSs implementations, we used the scripts released by the authors. In the parallel versions of the benchmarks, we need to specify the concurrency degree n to use, which, with exception of `Dedup` and `Ferret`, corresponds to the number of threads executed. We used different values for n, ranging from 1 to the number of threads contexts available in the used architecture (i.e., 48 in the Intel

[6]Other benchmark-specific flags are those specified by default in the `Makefiles` distributed by PARSEC.

Xeon server, 256 in the Intel Xeon Phi, and 160 in the IBM Power 8 server). The only exception to this rule is `Swaptions`, which cannot be executed with more than 128 threads due to limitations in the input set provided. The `Canneal`, `Raytrace`, and `Vips` benchmarks cannot be compiled on the IBM Power architecture due to architecture-specific assembler instructions used in the original implementations.

Discussion. The time measured is the one spent in the so-called region of interest (ROI), which includes all parts sensitive to the parallelization. This approach is commonly adopted when comparing different parallelizations of the same application [17]. Each program has been run multiple times, and the average results are shown (the standard deviation is always negligible, and it is not shown for readability reasons). All benchmarks have been executed with the original parameters provided by PARSEC. The results are shown in Figure 5, where the best execution times of the benchmarks for each version, obtained by varying the *n* parameter, have been normalized to the PARSEC reference implementation (i.e., OpenMP for `Freqmine`, Pthreads for all other benchmarks). Accordingly, values lower than 1 represent cases with execution time lower than the one of the reference PARSEC implementation. For completeness, Table 4 (shown later) reports the values of the best execution times of the various versions on the different architectures. Detailed performance results are available on the GitHub repository.[7]

Small differences and discrepancies in the results (between different versions of the same benchmark and/or between different architectures) are reasonably due to differences in the compiler and the architecture, and by the intrinsic differences and optimizations in the runtime of the frameworks used. Concerning architecture differences, the IBM Power 8 implements an eight-way simultaneous multithreading (SMT), whereas the Intel Xeon Phi and the Intel Xeon server implement four-way and two-way hyperthreading, respectively. The OmpSs implementations of `Blackscholes`, `Canneal`, and `Fluidanimate` executed on the Intel Xeon Phi give poor performance results. This is due to the fact that currently the OmpSs runtime has not been optimized for this new kind of platform. In the sequel, we will discuss the most remarkable differences among the analyzed versions.

For `Dedup`, we show in Figure 5 only the best FastFlow version (the one with scheme `pipe(seq1, ofarm(seq2, seq3, seq4), seq5')`). This version is significantly faster than the Pthreads one, as it removes all logic related to data reordering from the `seq5` stage, leaving only the writing of the data on disk. The improvement is less evident in the Intel Xeon Phi architecture since the writing part of the `seq5` stage is slower with respect to the other architectures due to the much lower clock, thus reducing the impact of this optimization. As shown in Table 3, the performance of this patterned version is 26% higher than the one of the other patterned versions that are more similar to the original Pthreads implementation. This is an interesting case where pattern composition allows the programmer to prototype alternative versions that are more efficient than the initial one by changing just few LOC (less than 10).

Although different versions could also be implemented with other programming models, this would require expressing again from scratch all communications and the data dependencies between different parts of the parallel application. This is an error-prone task and could significantly increase the code length. However, in the pattern-based model, dependencies and communications are implicitly coded in the pattern.

In `Facesim`, both the FastFlow and the OmpSs versions outperform the Pthreads parallelization (up to 40% faster). This is mainly due to implementation choices adopted in the different versions. In Pthreads, when a parallel kernel is found during the execution, one abstraction of

[7] Under the `results_TACO` folder.

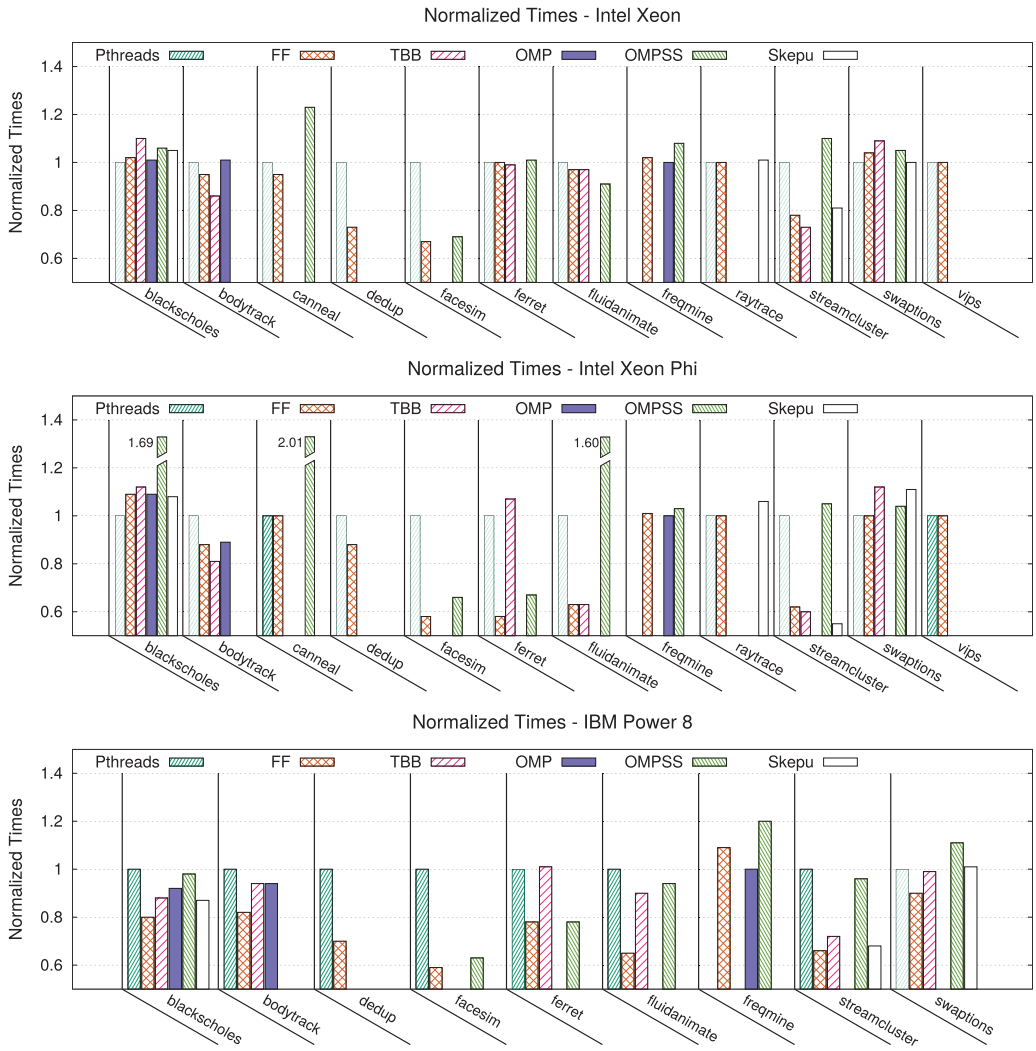

Fig. 5. Best execution times normalized with respect to the PARSEC reference (i.e., OpenMP for `Freqmine`, Pthreads for the remaining benchmarks).

Table 3. Best Speedups of Different Parallel Patterns for `Dedup` and `Ferret`

Arch.	Bench	1.	2.	3.	4.
Intel Xeon	`Dedup`	9.23	7.36	8.74	**9.26**
	`Ferret`	25.44	24.48	25.89	**25.89**
Intel Xeon Phi	`Dedup`	6.22	6.54	6.32	**6.6**
	`Ferret`	51.13	52.9	55.69	**92.6**
IBM Power 8	`Dedup`	10.79	12.07	12.61	**13.59**
	`Ferret`	25.53	23.79	25.32	**35.2**

Note: Numbers refer to the different pattern compositions described in Section 3.

Table 4. Best Execution Times (seconds)

A	Version	BS	BD	CN	DD	FC	FR	FL	FQ	RT	SC	SW	VP
Intel Xeon Server	Seq	135.2	126.9	82.9	28.6	350.5	368.1	286.9	431.9	145.6	417.3	240.1	97.6
	Pthreads	4.6	16.1	7.8	4.2	47.8	14.3	27.1	-	6.1	42.4	9.7	4.4
	FastFlow	4.7	15.3	7.4	3.1	32.2	14.2	26.3	34.1	6.1	33.1	10.1	4.4
	TBB	5.1	13.8	-	-	-	14.1	26.3	-	-	31.1	10.6	-
	OpenMP	4.6	16.3	-	-	-	-	-	33.3	-	-	-	-
	OmpSs	4.9	-	9.6	-	32.7	14.4	24.6	35.9	-	46.7	10.2	-
	SkePU	4.8	-	-	-	-	-	-	-	6.1	34.5	9.7	-
Intel Xeon Phi	Seq	997.7	704.5	311.4	120.0	1653.5	2102.2	1403.7	1835.8	981.8	1251.5	1416.4	691.1
	Pthreads	7.6	66.4	10.2	20.7	137.6	39.4	63.2	-	13.2	79.6	16.8	10.7
	FastFlow	8.3	58.6	10.1	18.1	80.3	22.7	40.0	102.8	13.2	49.0	16.9	10.7
	TBB	8.5	53.7	-	-	-	40.6	39.7	-	-	48.1	18.9	-
	OpenMP	8.3	59.1	-	-	-	-	-	101.7	-	-	-	-
	OmpSs	12.9	-	20.4	-	90.5	26.3	101.4	105.1	-	83.5	17.5	-
	SkePU	8.3	-	-	-	-	-	-	-	14.0	43.9	18.7	-
IBM Power 8	Seq	167.1	146.3	-	45.4	376.3	228.2	344.3	541.6	-	545.0	284.6	-
	Pthreads	5.3	17.2	-	4.7	58.1	8.3	35.0	-	-	100.7	10.1	-
	FastFlow	4.2	14.2	-	3.3	34.5	6.5	22.9	45.5	-	66.7	9.1	-
	TBB	4.7	16.2	-	-	-	8.4	31.3	-	-	72.7	10.0	-
	OpenMP	4.9	16.2	-	-	-	-	-	41.7	-	-	-	-
	OmpSs	5.2	-	-	-	36.4	6.5	32.8	50.1	-	96.3	11.2	-
	SkePU	4.6	-	-	-	-	-	-	-	-	68.8	10.2	-

Note: For the parallel versions, they are obtained by varying the concurrency degree. BS (`blackscholes`), BD (`bodytrack`), CN (`canneal`), DD (dedup), FC (`facesim`), FR (`ferret`), FL (`fluidanimate`), FQ (`freqmine`), RT (`raytrace`), SC (`streamcluster`), SW (`swaptions`), VP (`vips`). Concerning the missing data, in the OmpSs implementation of the benchmark suite, `Raytrace` and `Vips` are not available. Moreover, the output produced by `Dedup` and `Bodytrack` is different from the one produced by the original PARSEC implementation, and therefore their related results are not shown. Finally, `Canneal`, `Raytrace`, and `Vips` benchmarks cannot be compiled on the IBM Power architecture due to some architecture-specific assembler instructions.

a mesh partition is inserted for each thread in a shared queue, accessed by all threads and protected by `locks`. Instead, in the other two implementations, a partition is statically assigned to each thread without any need to access any shared data structure, thus achieving better speedup. Indeed, as shown in Figure 6, although the different versions are equivalent with low concurrency levels, the Pthreads version starts to perform poorly when more threads are used due to the high contention on this shared queue. A parallelization strategy similar to that of FastFlow could probably be used in the other programming models as well. However, as described earlier, we decided not to modify the original reference implementations since it is not the purpose of this work.

For the pattern-based implementation of `Ferret`, in Table 3 we report results of all alternative pattern implementations. The `pipe(seq1.1, farm(seq1.2; seq2; seq3; seq4; seq5), seq6)` version performs 80% better than the first pattern-based implementation, closer to the design of the Pthreads version, showing again the importance and flexibility of pattern composition

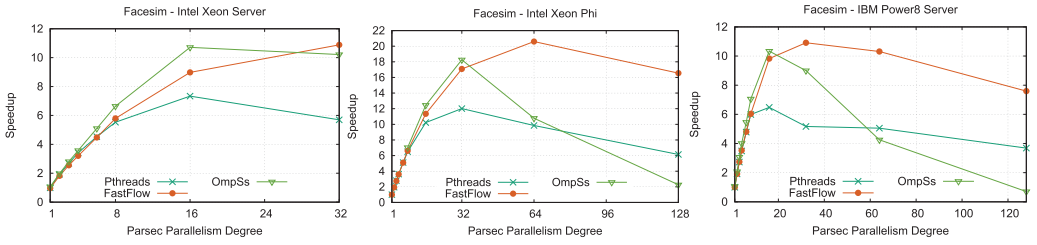

Fig. 6. `Facesim` speedup for different versions and on different architectures.

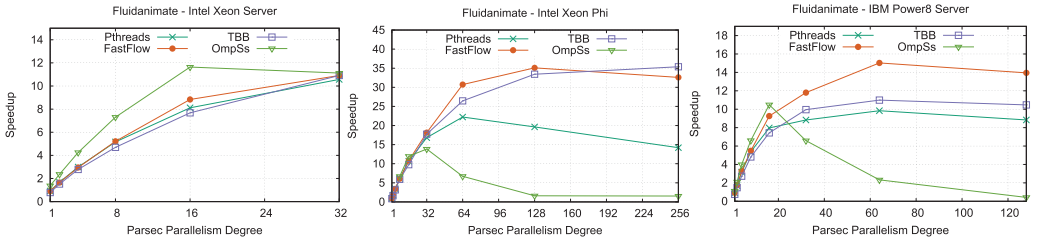

Fig. 7. `Fluidanimate` speedup for different versions and on different architectures.

to introduce optimizations. The OmpSs and FastFlow versions of `Ferret` produce the best performance gain over the Intel Xeon Phi and IBM Power 8 server. This happens because, differently from Pthreads and TBB, both versions parallelize the load of the images from the file (by separating `seq1.1` from `seq1.2`). The same effect does not occur on the Intel Xeon server due to the lower number of cores, as the images loading stage becomes a bottleneck only when using a high number of threads.

On `Fluidanimate`, we measured significant improvement in the pattern-based implementation with respect to the Pthreads one on the Intel Xeon Phi and IBM Power 8 server. This is mainly due to the different implementation of the barrier provided by the different frameworks. A barrier (implicit in the pattern-based approach) is executed after each parallel kernel. The one implemented in FastFlow is more efficient than the one used by Pthreads, thus leading to this performance gap. This performance difference is remarkable only at high concurrency levels, as shown in Figure 7, and does not occur on the Intel Xeon server, since it uses at most 48 hardware threads.

On `Streamcluster`, by parallelizing the kernels with `map` patterns, we greatly simplified the code. This made it possible to remove some unnecessary synchronizations (e.g., in the `pspeedy` function), leading to a performance improvement up to 40% on the Intel Xeon Phi. Such inefficiencies in the Pthreads implementation occur because of an intricate design, which led to a nonoptimized implementation. However, the pattern-based design of `Streamcluster` is simpler and more effective. Moreover, we did not parallelize some tiny functions that were parallelized in the Pthreads and OmpSs version. These functions are not worth to be parallelized since the overhead introduced by the parallelization would only slow down the entire application.

The original TBB implementation of `Swaptions` produced poor performance results with respect to other parallel implementations. We were able to reduce this gap by changing the size of the block scheduled to the different threads.

4.3 Summary of Results

To summarize the results, we achieved an average reduction of 26% in the LOC (in both FastFlow and SkePU) compared to the original Pthreads implementation, and an average reduction of 3% with respect to the OmpSs implementations. In the best case, we reduced the LOC up to 87% with respect to Pthreads and 14% compared to the OmpSs version. The code churn is an average of 58% lower than Pthreads and 34% lower than the OmpSs version. Concerning the performance, the FastFlow implementations obtained an average performance gain of 14%, with a maximum gain of 42% and a maximum loss of 9% with respect to the Pthreads one. Considering the benchmarks implemented with SkePU, we obtained an average performance gain of 7% (maximum gain of 45%, maximum loss of 11%). Finally, OmpSs implementations obtained an average gain of 2% (maximum gain of 37%, maximum loss of 23%)[8] with respect to the Pthreads implementation.

This evaluation confirmed that the pattern-based parallel programming approach reduces the LOC and code churn without impairing performance. In addition, in several cases, we were able to improve the performance by rapidly prototyping alternative pattern compositions.

5 RELATED WORK

Pattern-based programming has become a widely used coding practice in software engineering, both for sequential programming [33] and, with smaller acceptance, in the parallel computing domain [48]. The main reason around this shift is that parallel patterns simplify coding and maintainability and increase software portability, yet provide a good level of performance, often close to the one obtained with hand-tuned code. Notwithstanding, a comprehensive analysis that demonstrates the feasibility of using the pattern-based approach for parallelizing real-world applications is still missing. We tried to fill this gap for multicore platforms by parallelizing the benchmarks of the PARSEC suite using the pattern-based approach. The PARSEC suite covers a wide range of working set size, locality patterns, data sharing, synchronizations, and memory bandwidth requirements, which have made it particularly attractive for several research works [14, 23, 42].

In Chasapis et al. [17], the authors propose PARSECSs, a significant subset of the PARSEC suite (10 benchmarks) implemented using a task-based parallel programming model (OmpSs). They demonstrated that on average, the task-based approach is able to reduce the LOC needed to develop the PARSEC applications with respect to the native Pthreads implementations. Furthermore, they found that the overall performance is not degraded. Our work takes inspiration from PARSECSs with the aim to prove that a pattern-based parallel programming model is a well-suited candidate for high-level parallelization of applications. In our study, we also validated the results obtained in Chasapis et al. [17] by running the PARSECSs benchmarks on different multicore platforms, finding that in some cases they added optimizations that changed the original PARSEC sequential semantics, thus improving the original performance.

In Lee et al. [42], the authors studied pipeline parallelism proposing an extension to the Cilk programming model [8]. Results are validated using three PARSEC benchmarks (`Ferret`, `Dedup`, and `x264`), and they compared their approach to Pthreads and TBB. Other research works evaluated pattern-based programming frameworks by using only microbenchmarks [26, 32, 44]. In a previous work, we carried out a preliminary evaluation on a small subset of PARSEC applications [21]. To the best of our knowledge, no previous study has been conducted to thoroughly assess both the performance and programming effort of the pattern-based approach.

FastFlow has been used to parallelize several applications/algorithms in different application domains: bioinformatics [1, 9], data mining [4], data streaming processing [50], parallel numerical

[8]For the sake of fairness, in this comparison, we did not consider the results of OmpSs implementations of `Blackscholes`, `Canneal`, and `Fluidanimate` on the Intel Xeon Phi.

kernels [11], and network monitoring [19]. These works mainly focus on performance, and they can hardly be used to demonstrate general insights into the effectiveness of the pattern-based parallel programming model. With this work, we tried to provide a first answer in this direction.

6 CONCLUSIONS AND FUTURE WORK

This article presented P^{3}ARSEC, a suite based on PARSEC for benchmarking parallel pattern–based frameworks. Each application has been described from the pattern perspective as a composition and nesting of recurrent parallel patterns. This provided a guideline to parallelize such applications in different frameworks that offer the patterns we used in our analysis and confirmed that relatively few parallel patterns are sufficient to model complex real-world applications.

In addition to providing a benchmarking suite for pattern-based parallel programming, which was missing in the literature, this article also proposed an analysis aimed at evaluating the effectiveness of the parallel pattern–based programming methodology in terms of programmability and performance. To evaluate the programming effort, we computed specific metrics for all implemented P^{3}ARSEC benchmarks. The final result is that LOC and code churn using parallel patterns are reduced with respect to using Pthreads, and it is in most cases comparable to other parallel programming approaches based on `#pragma`-based annotations (i.e., OpenMP and OmpSs).

The performance has been accurately evaluated on three different multicore systems, which represent different classes of general-purpose shared-memory platforms. The analysis showed that the performance achieved by the patterned versions is generally similar to the one of the other implementations based on Pthreads and OmpSs. Furthermore, there are some specific cases where the flexibility of the pattern-based approach allows the programmer to easily prototype variants of the parallel implementations, which perform better than the initial and simpler versions.

In the future, we will extend the work by including GPUs as target platforms and evaluating the performance and programmability of the pattern-based programming framework on many-core systems. Moreover, we would like to evaluate the impact on the performance of using different C/C++ compilers, such as by using the `icc` compiler on Intel-based architectures.

REFERENCES

[1] Marco Aldinucci, Mario Coppo, Ferruccio Damiani, Maurizio Drocco, Massimo Torquati, and Angelo Troina. 2011. On designing multicore-aware simulators for biological systems. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'11)*. 318–325. DOI: http://dx.doi.org/10.1109/PDP.2011.81

[2] Marco Aldinucci and Marco Danelutto. 1999. Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*. 966–962.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2012. An efficient unbounded lock-free queue for multi-core systems. In *Euro-Par 2012 Parallel Processing*. Lecture Notes in Computer Science, Vol. 7484. Springer, 662–673. DOI: http://dx.doi.org/10.1007/978-3-642-32820-6_65

[4] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. 2014. Decision tree building on multi-core using FastFlow. *Concurrency and Computation: Practice and Experience* 26, 3, 800–820. DOI: http://dx.doi.org/10.1002/cpe.3063

[5] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. 1995. P3L: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience* 7, 3, 225–255. DOI: http://dx.doi.org/10.1002/cpe.4330070305

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 72–81. DOI: http://dx.doi.org/10.1145/1454115.1454128

[7] Fischer Black and Myron Scholes. 1973. The pricing of options and corporate liabilities. *Journal of Political Economy* 81, 3, 637–654.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices* 30, 8, 207–216. DOI: http://dx.doi.org/10.1145/209937.209958

[9] Andrea Bracciali, Marco Aldinucci, Murray Patterson, Tobias Marschall, Nadia Pisanti, Ivan Merelli, and Massimo Torquati. 2016. PWHATSHAP: Efficient haplotyping for future generation sequencing. *BMC Bioinformatics* 17, S-11, 342. DOI: http://dx.doi.org/10.1186/s12859-016-1170-y

[10] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, Los Alamitos, CA, 89–100. DOI: http://dx.doi.org/10.1109/PACT.2011.15

[11] Daniele Buono, Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. 2014. A lightweight run-time support for fast dense linear algebra on multi-core. In *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Networks*.

[12] Colin Campbell and Ade Miller. 2011. *A Parallel Programming With Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, Redmond, WA.

[13] Denis Caromel, Ludovic Henrio, and Mario Leyton. 2008. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'08)*. 45–53. DOI: http://dx.doi.org/10.1109/PDP.2008.29

[14] Juan M. Cebrian, Magnus Jahre, and Lasse Natvig. 2015. ParVec: Vectorizing the PARSEC benchmark suite. *Computing* 97, 11, 1077–1100. DOI: http://dx.doi.org/10.1007/s00607-015-0444-y

[15] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 363–375. DOI: http://dx.doi.org/10.1145/1806596.1806638

[16] Barbara Chapman. 2007. *The Multicore Programming Challenge*. Springer, Berlin, Germany.

[17] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article No. 41, 22 pages. DOI: http://dx.doi.org/10.1145/2829952

[18] Murray Cole. 2004. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing* 30, 3, 389–406.

[19] Marco Danelutto, Luca Deri, Daniele De Sensi, and Massimo Torquati. 2013. Deep packet inspection on commodity hardware using FastFlow. In *Proceedings of the 15th International Parallel Computing Conference (ParCo'13)*. 92–99. DOI: http://dx.doi.org/10.3233/978-1-61499-381-0-92

[20] Marco Danelutto, José Daniel Garcia, Luis Miguel Sanchez, Rafael Sotomayor, and Massimo Torquati. 2016. Introducing parallelism by using REPARA C++11 attributes. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'16)*. 354–358. DOI: http://dx.doi.org/10.1109/PDP.2016.115

[21] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, and Massimo Torquati. 2017. P^3ARSEC: Towards parallel patterns benchmarking. In *Proceedings of the Symposium on Applied Computing (SAC'17)*. ACM, New York, NY, 1582–1589. DOI: http://dx.doi.org/10.1145/3019612.3019745

[22] Marco Danelutto and Massimo Torquati. 2015. Structured parallel programming with "core" FastFlow. In *Central European Functional Programming School*. Lecture Notes in Computer Science, Vol. 8606. Springer, 29–75.

[23] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. 2016. A reconfiguration algorithm for power-aware parallel applications. *ACM Transactions on Architecture and Code Optimization* 13, 4, Article No. 43, 25 pages. DOI: http://dx.doi.org/10.1145/3004054

[24] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. 2017. Mammut: High-level management of system knobs and sensors. *SoftwareX* 6, 150–154. DOI: http://dx.doi.org/10.1016/j.softx.2017.06.005

[25] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1, 107–113. DOI: http://dx.doi.org/10.1145/1327452.1327492

[26] David del Rio Astorga, Manuel F. Dolz, Javier Fernndez, and J. Daniel Garca. 2017. A generic parallel pattern interface for stream and data processing. Available at http://dx.doi.org/10.1002/cpe.4175

[27] Antonio J. Dios, Rafael Asenjo, Angeles Navarro, Francisco Corbera, and Emilio L. Zapata. 2010. Evaluation of the task programming model in the parallelization of wavefront problems. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC'10)*. 257–264. DOI: http://dx.doi.org/10.1109/HPCC.2010.78

[28] Pradeep Dubey. 2005. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine* 9, 2, 1–10.

[29] Kento Emoto and Kiminori Matsuzaki. 2014. An automatic fusion mechanism for variable-length list skeletons in SkeTo. *International Journal of Parallel Programming* 42, 4, 546–563. DOI: http://dx.doi.org/10.1007/s10766-013-0263-8

[30] Johan Enmyren and Christoph W. Kessler. 2010. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the 4th International Workshop on High-Level Parallel Programming and Applications (HLPP'10)*. ACM, New York, NY, 5–14. DOI: http://dx.doi.org/10.1145/1863482.1863487

[31] Steffen Ernsting and Herbert Kuchen. 2012. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *IInternational Journal of High Performance Computing and Networking* 7, 2, 129–138. DOI: http://dx.doi.org/10.1504/IJHPCN.2012.046370

[32] August Ernstsson, Lu Li, and Christoph Kessler. 2017. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Available at https://doi.org/10.1007/s10766-017-0490-5

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Longman, Boston, MA.

[34] Buğra Gedik, Habibe G. Özsema, and Özcan Öztürk. 2016. Pipelined fission for stream programs with dynamic selectivity and partitioned state. *Journal of Parallel and Distributed Computing* 96, C, 106–120. DOI: http://dx.doi.org/10.1016/j.jpdc.2016.05.003

[35] Horacio González-Vélez and Mario Leyton. 2010. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience* 40, 12, 1135–1160. DOI: http://dx.doi.org/10.1002/spe.v40:12

[36] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3, 353–401. DOI: http://dx.doi.org/10.1017/S0956796805005538

[37] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for high-level and productive stream parallelism. *Parallel Processing Letters* 27, 1, 1–20. DOI: http://dx.doi.org/10.1142/S0129626417400059

[38] Michael Haidl and Sergei Gorlatch. 2017. High-level programming for many-cores using C++14 and the STL. Available at http://dx.doi.org/10.1007/s10766-017-0497-y

[39] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM SIGMOD Record* 29, 2, 1–12. DOI: http://dx.doi.org/10.1145/335191.335372

[40] David Heath, Robert Jarrow, and Andrew Morton. 1992. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica* 60, 1, 77–105.

[41] Vladimir Janjic, Chris Brown, Kenneth Mackenzie, Kevin Hammond, Marco Danelutto, Marco Aldinucci, and José Daniel Garcia. 2016. RPL: A domain-specific language for designing and implementing parallel C++ applications. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'16)*. 288–295. DOI: http://dx.doi.org/10.1109/PDP.2016.122

[42] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing* 2, 3, Article No. 17, 42 pages. DOI: http://dx.doi.org/10.1145/2809808

[43] Joeffrey Legaux, Frdric Loulergue, and Sylvain Jubertie. 2013. OSL: An algorithmic skeleton library with exceptions. *Procedia Computer Science* 18, 260–269. DOI: http://dx.doi.org/10.1016/j.procs.2013.05.189 2013 International Conference on Computational Science.

[44] Mario Leyton and José M. Piquer. 2010. Skandium: Multi-core programming with algorithmic skeletons. In *Proceedings of the18th Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'10)*. 289–296. DOI: http://dx.doi.org/10.1109/PDP.2010.26

[45] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2006. Ferret: A toolkit for content-based similarity search of feature-rich data. *ACM SIGOPS Operating System Review* 40, 4, 317–330.

[46] Kirk Martinez and John Cupitt. 2005. VIPS—a highly tuned image processing software architecture. In *Proceedings of the IEEE International Conference on Image Processing*, Vol. 2. II–574–7. DOI: http://dx.doi.org/10.1109/ICIP.2005.1530120

[47] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. DOI: http://dx.doi.org/10.1145/2851141.2851148

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

[49] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming*. Morgan Kaufmann, San Francisco, CA.

[50] Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Tiziano De Matteis. 2017. Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1. DOI: http://dx.doi.org/10.1109/TPDS.2017.2679197

[51] John C. Munson and Sebastian G. Elbaum. 1998. Code churn: A measure for estimating the impact of code change. In Proceedings of the International Conference on Software Maintenance. 24–31. DOI: http://dx.doi.org/10.1109/ICSM.1998.738486

[52] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. 284–292. DOI: http://dx.doi.org/10.1109/ICSE.2005.1553571

[53] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE, Los Alamitos, CA, 281–290. DOI: http://dx.doi.org/10.1109/PACT.2009.28

[54] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, efficient, parallel execution of parallel programs. *ACM SIGPLAN Notices* 49, 6, 169–180. DOI: http://dx.doi.org/10.1145/2666356.2594292

[55] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL—a portable skeleton library for high-level GPU programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11)*. 1176–1182. DOI: http://dx.doi.org/10.1109/IPDPS.2011.269

[56] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. In *ECOOP 2013—Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 7920. Springer, 52–78.

[57] Marco Vanneschi. 2002. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* 28, 12, 1709–1732.

[58] Elaine J. Weyuker. 1988. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 14, 9, 1357–1365. DOI: http://dx.doi.org/10.1109/32.6178

[59] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1, 20–24.

Received June 2017; revised July 2017; accepted August 2017