P^{3}ARSEC: Towards Parallel Patterns Benchmarking

Marco Danelutto Tiziano De Matteis Daniele De Sensi
Gabriele Mencagli Massimo Torquati
Department of Computer Science, University of Pisa
Largo B. Pontevorvo 3, I-56127, Pisa, Italy
{marcod,dematteis,desensi,mencagli,torquati}di.unipi.it

ABSTRACT

High-level parallel programming is a de-facto standard approach to develop parallel software with reduced time to development. High-level abstractions are provided by existing frameworks as *pragma*-based annotations in the source code, or through pre-built *parallel patterns* that recur frequently in parallel algorithms, and that can be easily instantiated by the programmer to add a structure to the development of parallel software. In this paper we focus on this second approach and we propose P^{3}ARSEC, a benchmark suite for parallel pattern-based frameworks consisting of a representative subset of PARSEC applications. We analyse the programmability advantages and the potential performance penalty of using such high-level methodology with respect to hand-made parallelisations using low-level mechanisms. The results are obtained on the new Intel *Knights Landing* multicore, and show a significantly reduced code complexity with comparable performance.

CCS Concepts

•Computing methodologies → Parallel computing methodologies; •Software and its engineering → Parallel programming languages; Design patterns; Software development techniques;

Keywords

Parallel Patterns, PARSEC Benchmarks, Intel KNL

1. INTRODUCTION

The advent of Chip Multi-Processors (briefly, CMPs) has brought parallel computing into the mainstream by providing a decisive contribution to the emergence and ubiquity of parallel machines in our everyday life. The wide diffusion of parallel architectures goes together with academic and industrial efforts in the design and development of suitable and easy-to-use parallel programming methodologies.

Parallel programming frameworks offer suitable abstractions to improve the time-to-development and productivity of parallel software. Some of them adopt a *pragma-based* approach like OpenMP [1] and OmpSs [2], where the philosophy is to be as unobtrusive as possible in the modification of the legacy source code by relying on a compiler that translates *pragmas* into a corresponding parallel run-time code. Data parallel paradigms (parallel-for and reduce) and task parallelism (graphs of dependencies) can both be expressed by the programmer using proper pragma-based directives.

Another approach consists in using high-level *parallel patterns* that the programmer composes and nests to build parallel implementations. Each pattern applies a parallelism paradigm to solve recurrent problems [3, 4], e.g., map, reduce, farm, pipeline, scan, zip are some notable examples. Frameworks supporting this patterned vision are FastFlow [5], SkePU [6], Muesli [7], Microsoft PPL [8], Delite [9] and many others. Patterns are typically C++/Java classes that the programmer instantiates by providing the business logic code and some configuration parameters as input arguments of their constructors. The counterpart is the potential rigidity of the approach when parts of the application that have to be parallelised do not match any available patterns. For these reasons, some frameworks like Intel TBB [4] adopt a sort of hybrid approach, where besides some pre-defined patterns (e.g., pipeline, parallel-for, reduce, scan) the framework allows general graphs of tasks to be executed by respecting their precedence relations.

Recent approaches [9,10] try to raise the level of abstraction. A solution is based on DSLs (Domain Specific Languages) built on top of pattern-based frameworks, in order to help domain experts to easily prototype parallel variants and to introduce optimisations. A second approach [11–13] consists in annotating the sequential code with C++11 attributes in order to introduce a parallel pattern in a region of code (usually a compute-intensive kernel). A source-to-source compiler is responsible to translate the annotated code into a parallel code linked to a parallel run-time library.

Besides the programmability advantage, a crucial point is to understand which is the performance gap in using such frameworks. *What is the performance loss of a high-level approach with respect to developing a hand-made parallelisation using low-level mechanisms?* An interesting work that

SAC 2017,April 03-07, 2017, Marrakech, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04…$15.00
http://dx.doi.org/10.1145/3019612.3019745

tries to answer this question for a pragma-based framework is the one in [14], which shows that the OmpSs porting of some parallel programs of the PARSEC benchmark suite [15] do not suffer substantial performance degradation with respect to PTHREADS counterparts, and have a reduced code complexity in terms of lines of codes. Our contribution with this paper is to provide a similar analysis for pattern-based frameworks which are more expressive in exposing the parallel structure than pragma-based approaches. As far as we know, this analysis for the PARSEC is missing in the literature, and it is important because this suite is representative of a wide range of application domains, from HPC to desktop and server applications. Our contributions are:

- a representative subset of PARSEC applications has been implemented in the FastFlow pattern-based framework [5]. In this analysis we identify which pattern composition is useful to implement each application;
- we experimentally prove that the pattern-based methodology is capable of reducing the programming effort by achieving performance comparable to optimised hand-made parallelisations based on PTHREADS;
- the experimental evaluation has been performed on a novel parallel CMP, i.e. the second generation (*Knights Landing*) Intel Xeon Phi architecture. We highlight that PARSEC performance results on such CMP are meaningful themselves, because of the still limited availability of this multi-core architecture to most of the scientific community;
- we start building a benchmark for parallel patterns based applications (we call this benchmark `P`^{3}`ARSEC` – Parallel Patterns `PARSEC`), releasing it as open source[1]. Furthermore, the pattern characterisation described in this paper can be implemented in all the parallel frameworks that offer the needed patterns, thus not limited to FastFlow.

The outline of this paper is the following. Sect. 2 introduces a brief description of the selected PARSEC applications. Sect. 3 presents the used patterns and their implementation in FastFlow. Sect. 4 provides details of the new Intel Phi and presents the experimental results. Finally, Sect. 5 provides the conclusions of this work.

2. APPLICATIONS

In this section we introduce the basic characteristics of the PARSEC benchmarks. Then, we describe in detail the subset of benchmark applications selected for the performance evaluation of the pattern-based methodology that will be described in Sect. 3.

2.1 The PARSEC Benchmark Suite

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [15] is a benchmark suite of state-of-the-art, computationally intensive multi-threaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for CMPs. PARSEC[2] is composed of 13 programs coming from different areas of computing. Each application is provided with several input sets, including a *native* set which is representative of real applications.

The PARSEC suite is interesting from the high-level parallel programming perspective, because its applications are characterised by different memory access behaviours, different data sharing patterns using different synchronisation mechanisms and workloads. Table 1 reports the official name of the benchmarks and their parallel execution model.

Parallelism Model	Applications
Data Parallel	blackscholes, fluidanimate,freqmine, streamcluster, swaptions, facesim, vips, raytrace, bodytrack
Unstructured Parallelism	canneal
Stream Parallel	dedup,ferret, x264

Table 1: Parallelism models of the PARSEC applications.

Data parallel applications work on large data structures partitioned among threads that execute all the computation phases on their partitions. Stream parallelism characterises applications where threads execute distinct computation phases on different data items in parallel. The case of `canneal` is representative of applications that do not straightforwardly follow any common parallelism model.

2.2 Selected PARSEC Benchmarks

Among all PARSEC benchmarks, we have selected a representative subset for our analysis which consists of five programs: `swaptions`, `blackscholes`, `ferret`, `dedup` and `canneal`. This subset has been carefully selected in order to show the best and worst-case performance on the new Intel Xeon Phi (Knights Landing) and to exemplify the application of the pattern-based methodology to a set of computations belonging to all the parallelism models present in PARSEC. The first two applications, `swaptions` and `blackscholes`, are both data-parallel applications. However, differently from `swaptions`, `blackscholes` computation is also iterative. Instead, `ferret` and `dedup` both use the stream parallelism model, however they have typically shown different scalability results on past PARSEC evaluations [14] owing to the different synchronisation mechanisms used in the computation (`dedup` makes use of locks which are absent in `ferret`) and to intrinsic applications characteristics. Finally, `canneal` has been chosen because it is the only example of unstructured parallelism in the suite.

In the following part of this section we provide a brief description of the five selected applications. A summary of the main characteristics of these benchmarks together with the available PARSEC implementations, is reported in Table 2.

`Swaptions` this application uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The PTHREADS version partitions the array into a number of

[1]https://github.com/paragroup/p3arsec

[2]In this paper we refer to the PARSEC version 3.0.

Benchmark	**Domain**	**Parallelism**			**Versions**
		Model	*Synchronisation*	*Grain*	
swaptions	Financial Analysis	Data Parallel	dataflow	coarse	Pthreads, Intel TBB
blackscholes	Financial Analysis	Data Parallel	dataflow	coarse	Pthreads, OpenMP, Intel TBB
ferret	Similarity Search	Stream Parallel	dataflow	medium	Pthreads, Intel TBB
dedup	Enterprise Storage	Stream Parallel	dataflow/locks	medium	Pthreads
canneal	Simulated Annealing	Unstructured	locks/atomic	fine	Pthreads

Table 2: Characteristics of the selected PARSEC v3.0 benchmark applications.

blocks equal to the number of threads and assigns one block to every thread. Each thread computes the price of all the swaptions in its work unit.

`Blackscholes` this application is an Intel RMS benchmark. It analytically calculates the prices for a portfolio of European options with the Black-Scholes partial differential equations (PDE). The PTHREADS implementation divides the portfolio into work units (one for each available thread). Then, each thread calculates the prices for its corresponding options. The algorithm is iterated multiple times to obtain the final estimation of the portfolio.

`Ferret:` this application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes. The toolkit has been configured for image similarity search. The PTHREADS implementation decomposes the application into six pipeline stages. The first and last stages are implemented as a single thread. The middle stages are configured with a thread pool each. Communication channels between pools are implemented using queues of fixed size (the default is 20 entries). The pipeline is sketched in Fig. 1.

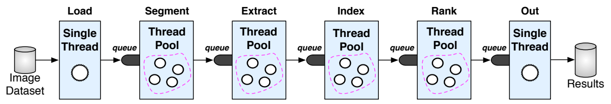


Figure 1: General scheme of the Ferret application.

`Dedup:` this application compresses a data stream with a combination of global and local compression called "deduplication". The PTHREADS version implements a pipeline with five stages, 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 stages is implemented using multiple queues of fixed size. Each queue is then assigned to a subset of threads in the pool. Fig. 2 shows a representation of the `dedup` processing pipeline. Differently from `ferret`, it is not a linear chain of stages, as results coming from the third stage can be transmitted directly to the last stage bypassing the fourth stage. Furthermore, the second stage can generate more output items per input item.

`Canneal:` this application uses cache-aware simulated annealing to minimise the routing cost of a chip design. It uses fine-grained parallelism with a lock-free algorithm and a very aggressive synchronisation strategy based on data race recovery instead of avoidance. The PTHREADS implementation picks pairs of pseudo-random elements of the graph and swaps them until the whole computation converges to an optimal solution. If the swap decreases the routing cost, the swap is automatically accepted. No locks are used to protect the list from concurrent accesses and swap operations are done using atomic CAS instructions. However, the evaluation of the elements to be swapped is not done atomically, thus, due to simultaneous swaps performed by other threads, it may happen to find a solution that increases the routing cost. In that case the algorithm performs a rollback step by swapping the elements again.

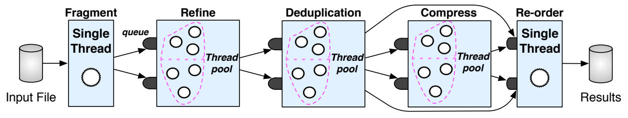


Figure 2: General scheme of the Dedup application.

3. PARALLEL PROGRAMMING WITH PATTERNS

Pattern-based frameworks provide a large set of parallel patterns that solve recurrent problems. Some notable examples are: *sequential composition*, *map*, *reduce*, *pipeline*, *farm*, *divide-and-conquer*, *scan*, *stencil*, *parallel-for*. In the following we briefly review the patterns that we used in the parallelisation of the selected PARSEC benchmark applications described in Sect. 2.2.

3.1 Used Patterns

Sequential (`seq`): this simple pattern describes a portion of the "business logic" code of the application. The implementation usually requires to wrap the code in a function hosting input and output parameters ($f : \alpha \to \beta$). The function f is executed sequentially.

Sequential Composition (`seqcomp`): this pattern describes the composition of two `sequential` patterns. Given two `sequential` patterns $f_1 : \alpha \to \beta$ and $f_2 : \beta \to \gamma$, then the `sequential composition` is $f_2(f_1(x)) : \alpha \to \gamma$ where $x : \alpha$ is the input data. The `seqcomp` is executed sequentially.

Pipeline (`pipe`): the pattern computes in parallel several stages on a stream of items. Each stage processes data produced by the previous stage in the pipe and delivers results to the next stage. If the i-th stage of the pipeline computes a function f_i, for each stream item x an item

$f_n(f_{n-1}(\ldots f_1(x)\ldots))$ is delivered in the pipeline output stream. Pipeline stages are executed in parallel.

Task-Farm (`farm`): This pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing in an input stream of type α `stream` delivering the results on the output stream of type β `stream`. The model of computation of the `task-farm` pattern consists of three logical entities: the *Emitter* that is in charge of accepting input data streams, in a data-flow or non-deterministic way, and to assign the data to the Workers; a pool of *Workers* which compute the function f in parallel over different stream elements; the *Collector* that non-deterministically gathers Workers' partial results and eventually produces the final result. The Emitter, the set of Workers and the Collector interact in a pipeline fashion using a data-flow model which can be implemented in several different ways depending on the target platform. For example, the Emitter and Collector, could be implemented in a *centralised* way using a single thread, or in a partially or fully *distributed* way. In some cases, the Emitter and Collector are implemented using a passive data structure. A farm's workers can be any other patterns. An interesting result concerning composition of `pipe` and `farm` patterns is the following [16]:

`pipe(seq(f1), seq(f2))` $\equiv$ `farm(seqcomp(f1,f2), n)`

where `n` is a non-functional parameter representing the number of Workers in the `farm` pattern. In the general case, input/output data ordering may be altered due to the different relative speeds of the workers executing the distinct stream items. If ordering is important, it can be enforced by the Collector or by the scheduling/gathering policies of the `farm` pattern. We call `ofarm` the instance of the `farm` pattern that preserves input/output ordering.

Master-Worker : this is a specialisation of the `task-farm` pattern where the Emitter and Collector are collapsed in a single entity (called master). The Workers deliver computed results back to the master. The master schedules received input tasks toward the pool of workers trying to balance their workload.

Map: this is a data parallel pattern. It computes a given function $f : \alpha \rightarrow \beta$ over all the data items of an input collection whose elements have type α. The output produced is a collection of items of type β. Given the input collection $x_1, x_2, \ldots, x_N$, the output collection is $y_1, y_2, \ldots, y_N$ where $y_i = f(x_i)$ for $i = 1, 2, \ldots, N$. Since each data item in the input collection is independent from the other items, all the elements can be computed in parallel. The model of computation of the `map` pattern is very similar to the one described for the `farm` pattern. The difference lies in the fact that the `farm` patterns works on a stream of independent data (the stream may be unbounded), while the `map` pattern receives a data collection of a fixed number of items that is *partitioned* among the available computing resources.

Pattern Iterator (`p-iterator`): The pattern iterates the computation of a nested pattern until a given condition is true. Given a pattern computing the function $f : \alpha \rightarrow \alpha$, a state S, a boolean predicate $\mathcal{P} : \alpha \times S \rightarrow \{true, false\}$ and a state update function $\mathcal{U} : \alpha \times S \rightarrow S$, this pattern computes the pattern function f until the predicate is not true. If the predicate is false, the output element is used as input to compute the nested pattern again. The parallel semantics of this pattern is the one of the inner pattern, which can be a stream-based parallel pattern as a `farm` or a `pipe` or a data-parallel pattern like a `map`. This pattern is often used for iterating data-parallel kernel computations up to a convergence criterion.

3.2 Benchmarks Parallelisation

In this section, we discuss how the selected PARSEC applications can be parallelised by using the parallel patterns introduced in the previous section. For each benchmark we first describe the implementations closest to the native PTHREADS version. Then, we briefly outline alternative implementations that can be obtained by exploiting parallel patterns composability. To exemplify the pattern-based methodology, we use the FastFlow framework [5].

`Swaptions:` this benchmark models a data parallel computation. It can be implemented as a single `map` pattern since it works on a collection of items representing the swaptions portfolio. The FastFlow implementation is based on a *parallel-for* loop that implements the `map` pattern when it is not nested in a `pipe` or `farm` patterns.

`Blackscholes:` this benchmark models an iterative computation. Its parallel behaviour can be modelled as a `p-iterator` pattern, where the internal pattern is a `map` computation because the input is a collection of items composing the portfolio. In the FastFlow framework, this benchmark can be implemented using different parallel patterns. We selected the *ff_Map* pattern that allows iterating a parallel-for (and a parallel-for-reduce) pattern inside a sequential loop as in Listing 1. It is worth pointing out that, the FastFlow `map` can also be used as a *pipeline* stage or a *farm* worker to model complex streaming networks of nested patterns. In the code the *svc* method executes the code of the pattern.

Listing 1: FastFlow implementation of the `blackscholes` benchmark.

```
struct BlackScholes: ff_Map<vec_wi> {
  BlackScholes(..<input>...) { ... }
  vec_wi *svc(vec_wi *in) {
    for(long i=0;i<nIter;++i)
      parallel_for(0,in->size,
        [&in](const long k) { ... });
    return in;
}} BS(<input>); // instance of the ff_Map
BS.run_and_wait_end();
```

`Ferret:` this is a streaming computation that can be modelled using the `pipe` pattern. Pipeline stages do not update any shared state concurrently and stream elements can be computed independently. Therefore, the four middle stages can be parallelised with a `farm` pattern, whereas the first and last stages, that are in charge of I/O operations, are `seq` patterns. We have the following nest of patterns that describe the `ferret` benchmark:

```
pipe(seq(Load), farm(Segment,n), farm(Extract,n),
     farm(Index,n), farm(Rank,n), seq(Output))
```

This nesting of patterns can be easily implemented in FastFlow by encapsulating each function in a sequential FastFlow node (`ff_node`) and using the `ff_Farm` and `ff_Pipe` patterns directly provided by the framework (see Listing 2).

Listing 2: FastFlow implementation of the `ferret` benchmark.

```
 1 struct Load: ff_node_t<long,load_data> {
 2     load_data *svc(long*) { ... };
 3 } In;
 4 struct Segment:ff_node_t<load_data,seg_data> {
 5     seg_data *svc(load_data *in) { ... };
 6 };
 7 struct Extract:ff_node_t<seg_data,extr_data> {
 8     extr_data *svc(seq_data *in) { ... };
 9 };
10 struct Index:ff_node_t<extr_data,vec_query_data> {
11     vec_query_data *svc(extr_data *in) { ... };
12 };
13 struct Rank:ff_node_t<vec_query_data,rank_data> {
14     rank_data *svc(vec_query_data *in) {....};
15 };
16 struct Output:ff_node_t<rank_data> {
17     void *svc(rank_data *in) { ... };
18 } Out;
19
20 ff_Pipe<> pipe(In,
21               make_Farm<Segment,n>(),
22               make_Farm<Extract,n>(),
23               make_Farm<Index,n>(),
24               make_Farm<Rank,n>(), Out);
25 pipe.run_and_wait_end();
```

Finally, it is worth noting that a **`pipe`** with parallel stages (each one instantiating a **`farm`**) can be easily transformed into a **`farm`** whose workers are internally parallel (they are **`pipe`** of sequential stages) or sequential (**`comp`**). Other transformations are possible. The implementation of this last transformation in FastFlow can be obtained by replacing lines 20-25 in Listing 2 with the code in Listing 3.

Listing 3: **`ferret`** implementation as a single farm.

```
ff_Farm<> farm(
  make_Comp<Segment, Extract, Index, Rank>(),
  n, In, Out);
farm.run_and_wait_end();
```

`Dedup:` the PTHREADS version of this benchmark has been parallelised by using a pipeline model with five stages. The first stage (*Fragment*) reads the data stream from the disk and then partitions the data at fixed positions; then, it produces in output a stream of data chunks. Each chunk can be processed independently from the other chunks. The second stage (*Fragment Refine*) further partitions the input chunk into smaller fine-grain chunks generating a nested stream. The third stage is *Deduplication*. It checks if the chunk has already been compressed in the past by accessing a hash table. If so, the chunk is marked as *duplicate*. *Compress* is the fourth stage where chunks, if not marked as *duplicate*, are compressed and the corresponding table entry is updated. To ensure correctness in the access to the database performed by *Deduplication* and *Compress* threads, each bucket in the hash table is protected with a *lock*. Eventually, the *Reorder* stage writes the final compressed output data into the output file. If the 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. This stage represents the main bottleneck of the **`dedup`** pipeline, both due to data reordering and to I/O.

With minimal modification to the original PARSEC code we can implement it with a FastFlow pipeline pattern with the following logical structure:

```
pipe(seq(Fragment), farm(Refine,n),
     farm(Deduplication,n),farm(Compress,n),seq(Reorder))
```

As in the **`ferret`** benchmark, the original pattern scheme of the application can be straightforwardly implemented in FastFlow. Also in this case it is possible to implement several parallel variants resulting from different nesting of **`pipe`** and **`farms`** and different composition of stages. Composition is possible even though some of them keep an internal state because the concurrent access to the state is lock-protected.

Since obtaining the **`farm`** version is trivial, we can derive as well an ordered **`farm`** version called **`ofarm`** in which the pattern guarantees to maintain in output the same ordering of the input. In such a case, since the ordering of the elements is implicit in the pattern, the *Reorder* stage can be replaced by a simpler stage that only performs the writing of the elements on the disk. Exploiting the ordering property of the pattern instead of performing it from scratch leads to advantages both in terms of code complexity and performance, as we will show in Section 4.

`Canneal:` the PTHREADS version of this benchmark follows an unstructured interaction model among threads. This application can be implemented by using a **`master-worker`** pattern where the workers are sequential pattern instances and the master is in charge of: *i)* implementing the barrier between two phases; *ii)* evaluating the termination condition; *iii)* (re-)starting the workers computation if the termination condition is false. In FastFlow the **`master-worker`** pattern is provided with the **`make_MasterWorker`** function as shown in Listing 4.

Listing 4: FastFlow implementation of **`canneal`**.

```
// Worker definition subclassing ff_node_t
// Master definition subclassing ff_node_t
auto mw = make_MasterWorker<Master, Worker, n>();
mw.run_and_wait_end();
```

4. EVALUATION

In the first part of this section, we provide detailed information about the second-generation Intel Xeon Phi based on the Knights Landing (KNL) architecture. In the second part, we describe the performance and code complexity results of the selected PARSEC applications run on the KNL.

4.1 Knights Landing Architecture

With respect to the first-generation Intel Xeon Phi (codename Knights Corner, KNC), the KNL version is also shipped as a standard standalone architecture [17]. The KNL architecture represents a performance jump compared with the previous KNC version, with a renewed on-chip interconnection network, memory sub-system with empowered performance for scalar and vector instructions. The KNL

CPU has at most 36 active tiles (see Fig. 3), each one composed of two cores, two vector processing units (VPUs) per core and 1MB L2 Cache shared between the two cores. The core internal architecture is derived from Intel Atom core: it is a two-wide out-of-order core with 4 hyper-threaded contexts. In addition, each core has a private L1d cache of 32KB.

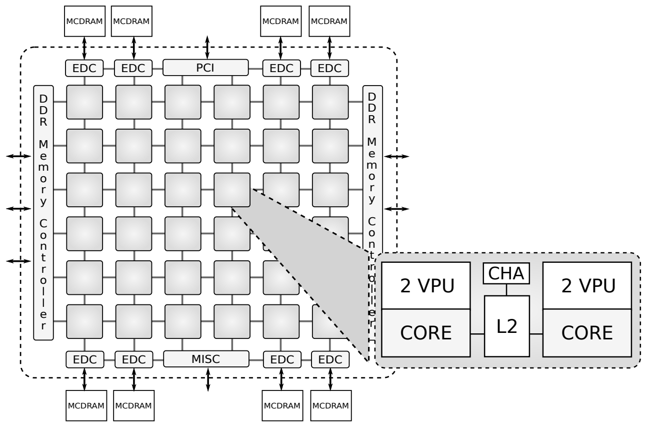


Figure 3: Knights Landing internal architecture.

KNL supports legacy x86 instructions. In addition, it introduces (as its predecessor) the AVX-512 instruction set which provides support for 512-bit-wide vector instructions. Tiles, memory controllers, I/O controllers and other chip components are interconnected through a 2D mesh. The mesh supports the MESIF cache coherent protocol. The KNL supports two levels of memory: multi-channel 3D-stacked DRAM (MCDRAM) and double rate (DDR4) memory. The former, is a 16GB high bandwidth memory offering more than 400 GB/sec of bandwidth. It is composed by 8 modules of 2 GB each, integrated on the same package and directly connected to the processor. In addition, KNL has six DDR4 channels for standard external memory.

The processor used for the experiments in this paper is the Intel Xeon Phi KNL model 7210. The CPU has 64 cores, running at 1.30 GHz. The machine is equipped with 96 GB of RAM. The installed OS is a Linux version kernel 3.10. For the experiments we disabled the power saving feature of the CPU, i.e. the cores run at the maximum frequency. In the used configuration, the mesh is used in *quadrant* mode [17], while the 16 GB MCDRAM is used in *cache mode* [17] (standard configuration suggested by the vendor for most of the workloads). The code has been compiled with the `gcc` compiler version `4.8.5` with the `-O3` optimisation flag.

4.2 Results

For the evaluation we used the PARSEC *native* input set, in order to obtain results representative of real-world program executions. Among the various parameters, the tool takes as input the number n of threads to use. Actually, this number does not always coincide with the total number of threads effectively used. For example, in `dedup` and `ferret`, this number corresponds to the number of threads activated on each parallel stage of the pipeline. We kept the same semantics even for FastFlow versions. As for all the PARSEC benchmarks, the time measured is the one spent in the so called *region of interest* (ROI), which includes all part sensible to the parallelisation. That part includes the setup time of the run-time framework. Each program has been run multiple times and average results are shown (the standard deviation is always small). All the benchmarks have been executed with the original parameters provided by PARSEC. If present in the PARSEC distributions, the OpenMP and TBB versions have also been executed.

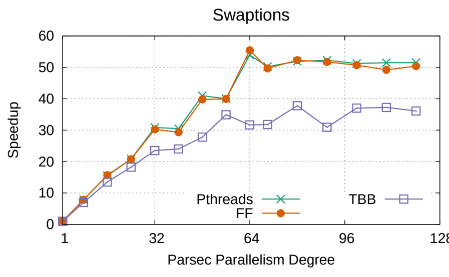
This work has been engineered as a PARSEC plug-in. Accordingly, it is possible to add the parallel patterns implementation of the benchmarks along with the already present Pthreads, OpenMP and TBB version, and to run them using the `parsecgmt` tool provided by PARSEC.

Fig. 4 shows the speedups achieved for the different benchmarks. It has been computed considering the execution time of the PARSEC serial version of the program. In `swaptions` (Fig. 4a), due to the specific *native* input provided by PARSEC, it is possible to run the benchmark with at most 128 threads. TBB version exhibits the lowest speedup among the existing implementations, probably due to the excessive overhead required to manage small chunks of swaptions. The parallel pattern implementation is characterised by a performance very close to the Pthreads version. For `blackscholes` (Fig. 4b) all the different implementations exhibit a good speedup. This is a consequence of the fact that this is essentially an embarrassingly parallel computation. Even for `ferret` (Fig. 4c) the parallel pattern implementation is comparable to the Pthreads one. The TBB version is not shown since it has a different structure with respect to the Pthreads one. However, we will show its best performance result in Fig. 5. In the `dedup` plot (Fig. 4d), the Pthreads version is compared with its corresponding implementation using patterns (that is a `pipe` of `farms` stages). The parallel pattern FastFlow version is better performing than the Pthreads one owing to the characteristics of the FastFlow run-time. In this particular case, this is caused by the different implementations of the communication channels between pipeline stages. In `canneal` (Fig. 4e), the Pthreads and FastFlow versions achieve a comparable speedup. The performance peak is reached at different concurrency levels. This effect is caused by the difference in the mechanisms used to perform a global synchronisation between threads. Indeed, in the FastFlow version the synchronisation is performed by the *master* thread, which is slightly more efficient for lower parallelism degree than the `pthread_barrier` used by the Pthreads version.

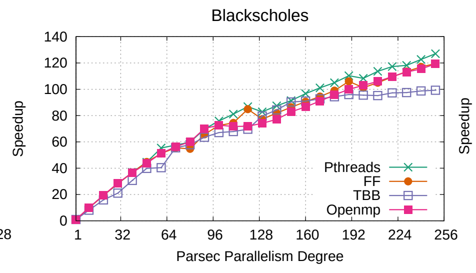
4.3 Alternative Versions

After we compared the existing implementations provided by PARSEC with the most similar parallel pattern implementations, we now analyse different alternative parallel pattern solutions for `ferret` and `dedup`. These versions have been derived by implementing a different nesting of patterns without adding extra code, such as for example the `farm` of `pipes` (`farm(pipes)`) and `farm` of `seqcomp` (`farm(seqcom)`).

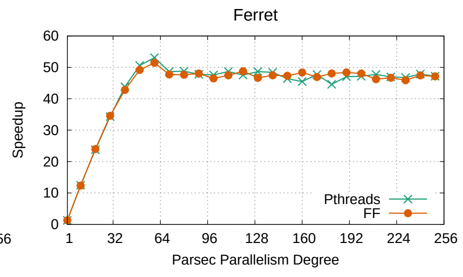
For `dedup`, we also implemented a new version based on the `ofarm(seqcomp)` pattern. This version required: i) to add a few lines of extra code in the *Compress* stage; ii) to remove the code used for data reordering management in the



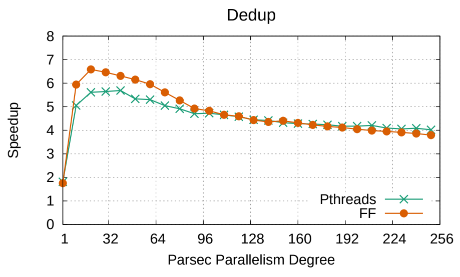
(a) Swaptions



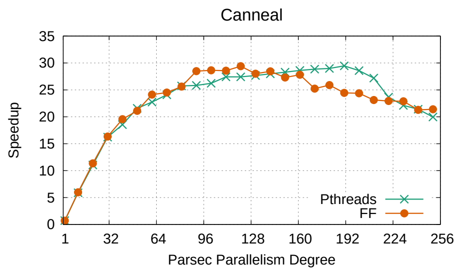
(b) Blackscholes



(c) Ferret



(d) Dedup



(e) Canneal

Figure 4: Speedup of the five selected applications using different parallel frameworks and PTHREADS.

Reorder stage. The `ofarm` version is not shown for `ferret` since data ordering is not needed by this application. Table 3 reports the maximum speedup achieved in the different implementations. The best `dedup` implementation is characterised by a 25% performance improvement with respect to the less performing one, thus highlighting the importance of programming abstractions to allow a rapid prototyping of different alternative versions of the application.

	`pipe(farms)`	`farm(pipes)`	`farm(seqcomp)`	`ofarm(seqcomp)`
`dedup`	6.69	5.64	6.56	**7.02**
`ferret`	**51.43**	49.72	51.30	N.D.

Table 3: Best speedups of different parallel patterns for `dedup` and `ferret`.

4.4 Final Analysis

We compared the code developed using FastFlow with the one of the PTHREADS implementation. We used two metrics: Lines of Code (LOC) and the Cyclomatic Complexity (CC), which measures the number of linearly independent paths through a source code, i.e. the higher the CC index the higher the code complexity. For this comparison, we considered only the source files which differ between the two implementations, excluding comments and empty lines.

The results are summarised in Table 4. For each benchmark, we consider the best performing parallel pattern implementation among those described and we focus on the lines of codes of the parallel region. For all the benchmarks we reduced the lines of code with respect to the PTHREADS version (up to 42.4% for `ferret`). Also the CC has been reduced for all the benchmarks, with a significant reduction in the `dedup` case, where the complexity due to the ordering of the results and explicit management of communication between stages has been reduced since it is embedded in the pattern itself. In addition to that, the performance are usually comparable with those obtained by the hand-written PTHREADS version. In the `dedup` case, a performance improvement of 17% is obtained. This is caused both by intrinsic differences in the run-time but also by a more efficient management of the reordering of the processed elements (`ofarm` version).

Benchmark	**Pattern**	**LOC Reduction**	**CC Pthreads**	**CC FastFlow**	**Perf. Gain**
Swaptions	`map`	17.9%	22	20	3.7%
Blackscholes	`p-iterator(map)`	1.2%	244	238	-6.3%
Canneal	`master-worker`	2.1%	21	23	0
Dedup	`ofarm(seqcomp)`	32.04%	284	124	17.14%
Ferret	`pipe(farms)`	42.4%	92	42	-3.2%

Table 4: Summary comparison of the original PTHREADS versions against the best parallel pattern versions.

Finally, Fig. 5 shows, for all benchmarks, the best execution times, normalised to the PTHREADS time, obtained by varying the PARSEC `-n` parameter up to 256. Accordingly, values lower than 100 represent solutions with an execution time lower than the one of the PTHREADS implementation.

5. CONCLUSIONS

In this paper we proposed $\mathtt{P^3ARSEC}$, a benchmark for parallel pattern-based frameworks. The selected applications are a representative subset of the standard PARSEC benchmarks suite. We have identified which pattern composition is useful to parallelise each application, then we experimentally showed that the pattern-based methodology is able to

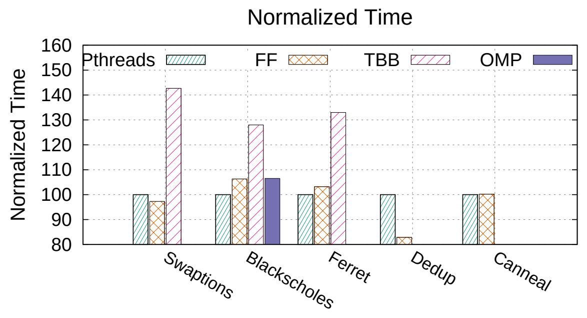


Figure 5: Best execution times normalised to the PTHREADS time.

reduce the code complexity measured in terms of Lines-Of-Code and Cyclomatic Complexity.
The experimental results showed that the performance obtained are comparable with those achieved by native PARSEC implementations based on PTHREADS. Furthermore, we showed how the proposed methodology allows alternative parallel solutions to be easily defined with a limited effort, obtaining in some cases even better performance with respect to the reference parallel solution. The experiments has been conducted on the new Intel *Knights Landing* CMP.

Several future extensions of this work can be devised, the most interesting are: *i)* the integration in $\mathtt{P^3ARSEC}$ of other benchmarks to further validate the results obtained in this work; *ii)* to compare the programmability effort and performance figures with respect to other parallelisation approaches as for example the *pragma*-atic one proposed in [14].

Acknowledgments

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235).

6. REFERENCES

[1] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 2007.

[2] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011.

[3] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[4] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[5] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow," in *Central European Functional Programming School*, ser. LNCS. Springer, 2015, vol. 8606, pp. 29–75.

[6] J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, ser. HLPP '10. New York, NY, USA: ACM, 2010, pp. 5–14.

[7] S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-gpu systems and clusters," *Int. J. High Perform. Comput. Netw.*, vol. 7, no. 2, pp. 129–138, Apr. 2012.

[8] C. Campbell and A. Miller, *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*, 1st ed. Microsoft Press, 2011.

[9] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *2011 Inter. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '11. IEEE, 2011, pp. 89–100.

[10] T. Sujeeth, Arvind K.and Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, *Composition and Reuse with Compiled Domain-Specific Languages.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 52–78.

[11] M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati, "Introducing parallelism by using repara c++11 attributes," in *24th Euromicro Inter. Conf. on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 354–358.

[12] M. Danelutto, T. D. Matteis, G. Mencagli, and M. Torquati, "Parallelizing high-frequency trading applications by using c++11 attributes," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, Aug 2015, pp. 140–147.

[13] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati, "Data stream processing via code annotations," *The Journal of Supercomputing*, pp. 1–15, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11227-016-1793-9

[14] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero, "PARSECSs: Evaluating the impact of task parallelism in the parsec benchmark suite," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 41:1–41:22, Dec. 2015.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *17th Inter. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '08. ACM, 2008, pp. 72–81.

[16] M. Aldinucci and M. Danelutto, "Stream parallel skeleton optimization," in *Proc. of PDCS: Intl. Conf. on Parallel and Distributed Computing and Systems*, IASTED. ACTA press, Nov. 1999, pp. 955–962.

[17] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.