Supporting Advanced Patterns in GrPPI, a Generic Parallel Pattern Interface

David del Rio Astorga(✉), Manuel F. Dolz, Javier Fernández, and J. Daniel García

Computer Science and Engineering Department,
University Carlos III of Madrid, 28911 Leganés, Spain
david.rio@uc3m.es, {mdolz,jfmunoz,jdgarcia}@inf.uc3m.es

Abstract. The emergence of generic interfaces, encapsulating algorithmic aspects in pattern-based constructions, has greatly alleviated the development of data-intensive and stream-processing applications. In this paper, we complement the basic patterns supported by GrPPI, a C++ General and Reusable Parallel Pattern Interface of the state-of-the-art, with the advanced parallel patterns Pool, Windowed-Farm, and Stream-Iterator. This collection of advanced patterns is basically oriented to some domain-specific applications, ranging from the evolutionary to the real-time computing areas, where compositions of basic patterns are not capable of fully mimicking algorithmic behavior of their original sequential codes. The experimental evaluation of the advanced patterns on a set of domain-specific use-cases, using different back-ends (C++ Threads, OpenMP and Intel TBB) and pattern-specific parameters, reports remarkable performance gains. We also demonstrate the benefits of the GrPPI pattern interface from the usability and flexibility points of view.

Keywords: Parallel programming framework
Domain-specific parallel pattern · Data and stream computing
High-level API

1 Introduction

The advent of the heterogeneous HPC architectures in the last decade paved the way in improving performance of data-intensive and stream-processing applications [21]. This fact, however, posed a number of challenges to developers for exploiting available resources of parallel hardware. An example among these challenges is the variety of programming frameworks existing for multi-/many-core CPUs, GPUs, co-processors, DSPs or FPGAs units present in heterogeneous platforms [8]. Therefore, it becomes clear that additional expertise is required, not only to develop applications using those frameworks, but also to select and tune them optimally to operate on these architectures. The lack of unified interfaces, integrating available processor-specific programming frameworks in a standalone layer, makes the development an even more complex task.

© Springer International Publishing AG, part of Springer Nature 2018
D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 55–67, 2018.
https://doi.org/10.1007/978-3-319-75178-8_5

With the recent emergence of pattern-based programming frameworks, encapsulating algorithmic aspects using a building blocks approach, this aspect has been relieved when programming for parallel platforms [16]. Basically, parallel patterns offer a way to implement robust, readable and portable solutions while hiding away the complexity behind concurrency mechanisms, e.g., thread management, synchronizations or data sharing. Numerous examples of pattern-based programming frameworks, such as SkePU [9], FastFlow [3] or Intel TBB [19], can be found in the literature. Nevertheless, most of these frameworks are not generic enough nor offer unified pattern interfaces [5]. To tackle these issues, the recent interface GrPPI [20], accommodates a unified layer of generic and reusable parallel patterns on the top of existing execution environments and pattern-based frameworks. However, we find that the core patterns offered by this interface do not fully match in some domain-specific use cases, e.g., evolutionary and symbolic algorithms.

To deal with this issue, we extend GrPPI with a collection of advanced parallel patterns targeted to domain-specific applications and evaluate their performance on a set of use cases from different computing areas. Specifically, this paper contributes with the following:

– We complement the core patterns supported by GrPPI with the advanced parallel patterns: Pool, Windowed-Farm, and Stream-Iterator.
– We demonstrate the flexibility and the composability of the advanced patterns in the GrPPI interface context.
– We assess the usability of the patterns with respect to the number of lines of code (LOCs) that have to be modified in order to parallelize the selected use cases.
– We evaluate the performance gains by using these patterns on a set of domain-specific use cases and varying configurations of parallelism degree and problem-specific parameters.

The remainder of this paper is organized as follows. Section 2 revisits some related works about parallel programming frameworks and domain-specific patterns. Section 3 states the formal definition of the advanced parallel patterns supported by GrPPI. Section 4 describes the interface adopted for the new patterns presented in this paper. Section 5 evaluates these patterns from the usability and performance points of view under three different use cases. Section 6 gives a few concluding remarks and future works.

2 Related Work

In the literature, we found numerous works proposing parallel patterns targeted to modern architectures for developing applications. In a first place, we find several open-source pattern libraries oriented exclusively to multi-core processors, e.g., Intel Thread Building Blocks (TBB) [19], RaftLib [4] or Kanga [14], and others supporting also accelerators, such as, SkePU [9], which allows hybrid

CPU–GPU configurations. We also encounter commercial solutions in the state-of-the-art, such as Thrust [17] and SYCL [13] for CUDA and OpenCL devices, respectively. Simultaneously, standardized interfaces are being progressively developed. This is the case of C++ STL algorithms, available in the forthcoming C++17, that start defining parallel versions of already existing STL algorithms [11]. Similar implementations to the parallel STL can also be found as third-party libraries, e.g., HPX [12] and GrPPI [20].

All in all, we observe that these frameworks provide a collection of classic parallel patterns targeted to data and stream-processing applications, e.g., the Map, Reduce, MapReduce, Pipeline and Farm patterns. However, none of them natively supplies advanced patterns. As stated in the previous section, we refer to advanced patterns to those constructions that match the algorithmic behavior of some particular domain-specific applications coming from, e.g., the symbolic computing, control theory, biology, wireless sensor networks or real-time stream processing domains. In this sense, we find that only some pattern-based frameworks in the literature have pushed forward the development of complex, high-level patterns. For instance, the FastFlow [3] library recently provided the Pool [2] and the Windowed-Farm [7] patterns, two commonly used structures in evolutionary and stream-intensive applications, respectively. On the other hand, the MALLBA library [1] offers a collection of high-level skeletons for combinatorial optimization which deals with parallelism in a user-friendly and efficient manner. In any case, the high-level patterns offered by these frameworks are not generic enough to be easily leveraged when developing parallel applications. The contribution of this paper is mainly focused on complementing the GrPPI library of basic parallel patterns with a new set of advanced patterns matching the algorithms that commonly appear in, e.g., genetic, sensor networks or real-time applications.

3 Advanced Parallel Patterns

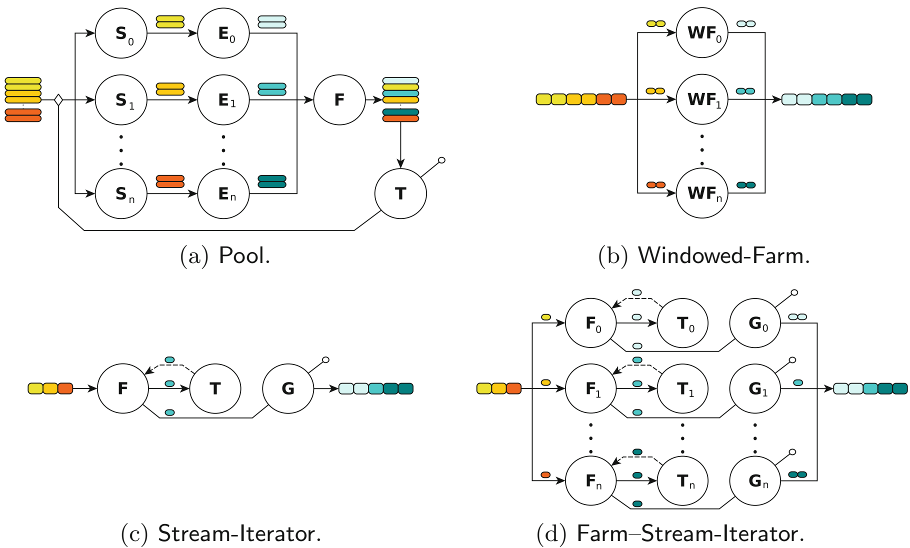
Patterns have been generally defined as recurring strategies for solving problems from a wide spectrum of areas, such as architecture, object-oriented programming and software architecture [15,16]. In our case, we take advantage of parallel software design patterns, since they provide a mechanism to encapsulate algorithmic features and are able to make applications more robust, portable and reusable. Also, if these patterns are properly tuned, they can achieve a good balance between parallel scalability and data locality.

As observed, several solutions in the state-of-the-art offer collections of basic parallel patterns as a "building blocks" modeling strategy for developing stream processing and data-intensive applications. However, while many of the algorithms found in general-purpose applications match directly those patterns, there exist situations in which those have to be composed among them in order to comply with the algorithm requirements. Furthermore, we identify some domain-specific algorithms, in which those basic patterns do not match any of these constructions or have to be composed in a very complex way in order to satisfy the problem prerequisites. This occurs in many algorithms that come from

the evolutionary and symbolic computing [10] domain, wireless sensor networks algorithms [6] or in real-time processing engines [18]. Therefore, we determine the need for supporting advanced patterns in order to simplify the development of complex algorithms related to the aforementioned application domains.

In the following, we describe formally three new parallel patterns that can be eventually incorporated during the parallelization task of such applications: Pool, Windowed-Farm and Stream-Iterator.

Pool. This pattern models the evolution of a population of individuals matching many evolutionary computing algorithms in the state-of-the-art [2]. Specifically, the Pool pattern is comprised of four different functions that are applied iteratively to a population P of individuals of type α (see Fig. 1(a)). First, the *selection* function S: $\alpha^* \rightarrow \alpha^*$ selects a subset of individuals belonging to P. Next, the selected individuals are processed by means of the *evolution* function E: $\alpha^* \rightarrow \alpha^*$, which may produce any number of new or modified individuals. The resulting set of individuals computed by E are filtered through a *filter* function F: $\alpha^* \rightarrow \alpha^*$, and eventually inserted into the population. Finally, the *termination* function T: $\alpha^* \rightarrow \{true, false\}$ determines in each iteration whether the evolution process should be finished or continued. To guarantee the correctness of the parallel version of this pattern, both functions S and E should be pure, i.e., they can be computed in parallel with no side effects.



(a) Pool. (b) Windowed-Farm. (c) Stream-Iterator. (d) Farm–Stream-Iterator.

Fig. 1. Advanced parallel patterns.

Windowed-Farm. This stream-oriented pattern delivers "windows" of processed items to the output stream. Basically, this pattern applies the function WF over consecutive contiguous collections of x input items of type α and delivers the

resulting windows of y items of type β to the output stream (see Fig. 1(b)). Optionally, these windows can have an overlap factor, i.e., the number of items in the window w_i that are also part of the window w_{i+1}. The parallelization of this pattern requires a pure function WF: $\alpha^* \rightarrow \beta^*$ for processing item collections.

Stream-Iterator. This stream pattern is intended to recurrently compute the pure function F: $\alpha \rightarrow \alpha$ on a single stream input item until a specific condition, determined by the boolean function T: $\alpha \rightarrow \{true, false\}$, is met. Additionally, in each iteration the result of the function F is delivered to the output stream, depending on a boolean output guard function G: $\alpha \rightarrow \{true, false\}$ (see Fig. 1(c)). Note that this pattern, due to its nature, does not provide any parallelism degree by itself and can be classified as a pattern modifier. Therefore, the parallel version of this construction is only achieved when it is composed with some other core stream pattern, e.g., using Farm or Pipeline as for the function F. An example of Stream-Iterator composed with a Farm pattern is shown in Fig. 1(d).

4 Description

In this section, we extend our generic and reusable parallel pattern interface (GrPPI) for C++ applications, previously presented in [20], with the advanced parallel patterns described in Sect. 3. In general, GrPPI takes full advantage of modern C++ features, metaprogramming concepts and generic programming to act as switch between the parallel programming models OpenMP, C++ threads and Intel TBB. Its design allows users to leverage the aforementioned execution frameworks just in a single and compact interface, hiding away the complexity behind the use of concurrency mechanisms with negligible overheads. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while composing them to arrange more complex ones. Thanks to these properties, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relative small efforts, having as a result portable codes that can be executed on multiple platforms.

Next, we describe in detail the interfaces of the advanced parallel patterns offered by GrPPI and demonstrate its composability.

Pool. The GrPPI interface designed for the Pool pattern, shown in Listing 1.1, receives the execution model, the population (`popul`), the selection (`select`), evolving (`evolve`), filtering (`filter`) and termination (`term`) functions, and the number of selections that will be performed. Initially, the parallel pattern implementation of GrPPI divides the number of selections among the concurrent processing entities that will select and evolve the population individuals. Afterwards, the resulting individuals are merged and forwarded to the sequential filtering and termination functions. Finally, only if the termination condition is met, the Pool parallel pattern finishes and delivers the resulting population. On the contrary, the whole process is repeated again with the evolved population.

The parallelism of this pattern is controlled via the execution model parameter, which can be set to operate in sequential or in parallel, through the different supported frameworks; e.g. to use C++ threads, the parameter should be set to `parallel_execution_thr`. In this case, any execution model can optionally receive, as an argument, the number of entities to be used for the parallel execution, e.g., `parallel_execution_thr{6}` would use 6 worker threads. If this argument is not given, the interface takes by default the number of threads set by the underlying platform.

Listing 1.1: Pool interface.

```
1 template <typename EM, typename P, typename S, typename E, typename F, typename T>
2 void Pool(EM exec_mod, P &popul, S &&select, E &&evolve, F &&filt, T &&term, int num_select);
```

Windowed-Farm. The interface for the Windowed-Farm pattern, described in Listing 1.2, receives the execution model, the stream consumer (`in`), the Farm (`task`) and the producer (`out`) functions. This pattern also receives the size and the overlap factor of the windows.[1] Specifically, the `in` function reads from the input stream as many items as required to fill the window buffer. Next, this buffer is forwarded to one of the concurrent entities, which will compute the function `task` in a Farm-like fashion. Therefore, the parallel implementation of this GrPPI pattern is offered by the Farm construction. Finally, the items collections resulting from the `task` function are delivered to the output stream. Note that, depending on the user requirements, this pattern can deliver items windows in an ordered way by properly configuring the execution model.

Listing 1.2: Windowed-Farm interface.

```
1 template <typename EM, typename I, typename WF, typename O>
2 void WindowedFarm(EM exec_mod, I &&in, WF &&task, O &&out, int win_size, int overlap);
```

Stream-Iterator. The GrPPI interface for the Stream-Iterator pattern, detailed in Listing 1.3, takes the execution model, the stream consumer (`in`), the kernel (`task`) and the producer (`out`) functions. This pattern also receives two boolean functions: the termination (`term`) and output guard (`guard`) functions. In the first step, the `in` function reads items from the input stream and a worker thread executes the kernel `task` function for each item. Next, the termination function `term` is evaluated with the resulting item to determine if the kernel should be re-executed on the same input item. Additionally, the output `guard` function decides whether an item should be delivered to the output stream or not.

Listing 1.3: Stream-Iterator interface.

```
1 template <typename EM, typename I, typename F, typename O, typename T, typename G>
2 void StreamIteration(EM exec_mod, I &&in, F &&task, O &&out, T &&term, G &&guard);
```

As stated in the previous section, the parallelism of the Stream-Iterator pattern is only obtained when it is composed with a basic GrPPI parallel pattern, e.g., Farm or Pipeline. As an example of composition, the code in Listing 1.4

[1] Note that while the current Windowed-Farm pattern only supports count-based windows, in the future we plan to extend its interface to cover time-based, slide-by-tuple and delta-based windowing models.

implements a Stream-Iterator, in which the kernel `task` function has been composed with the Pipeline pattern. Therefore, the kernel is computed in parallel by 2 worker threads. Note that the `optional`, as for the return type in the consumer function lambda, is used to indicate the end of the stream when constructed without arguments. As can be seen, thanks to GrPPI, it is possible to compose advanced with basic parallel patterns in order to increase the parallelism degree.

Listing 1.4: Example of Stream-Iterator-Pipeline composition.

```
1  StreamIteration( parallel_execution_thr{4},
2      [&]() -> optional<int> { // Consumer function
3          auto value = read_value(is);
4          return ( value > 0 ) ? value : {};
5      },
6      Pipeline( // Kernel function
7          []( int e ) { return e + 2*e; },
8          []( int e ) { return e - 1; }
9      ),
10     [&]( int e ){ os << e << endl; }, // Producer function
11     [] ( int e ){ return e < 100; }, // Termination function
12     [] ( int e ){ return e % 2 == 0; } // Output guard function
13 );
```

5 Evaluation

In this section, we perform an experimental evaluation of the three novel advanced patterns from the usability and performance points of view. To do so, we use the following hardware and software components:

- *Target platform.* The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57.
- *Software.* To develop the parallel versions and to implement the proposed interfaces, we leveraged the execution environments C++11 threads and OpenMP 4.5, and the pattern-based parallel framework Intel Threading Building Blocks (TBB). The C++ compiler used to assemble GrPPI is GCC v5.0.
- *Use cases.* To evaluate the advanced patterns, we use three different synthetic use cases targeting problems from different domains.
    - The Pool pattern has been evaluated on a benchmark that solves the *traveling salesman* problem (TSP) using a regular evolutionary algorithm. This NP-problem computes the shortest possible route among different cities, visiting them only once and returning to the origin city.
    - To evaluate the Windowed-Farm, we use a benchmark that computes average window values from an emulated sensor readings.
    - For the Stream-Iterator, we leverage a benchmark that reduces the resolution of the images appearing in the input stream, and produces the images with concrete resolutions to the output stream.

In the following sections, we analyze the usability, in terms of lines of code, and the performance of the GrPPI advanced patterns using the above-mentioned benchmarks with varying configurations of parallelism degree, problem size and execution frameworks.

5.1 Usability Analysis

In this section we analyze the usability and flexibility of the advanced pattern interfaces. To analyze these aspects, we assess the number of modified lines of code (LOCs) required to implement the parallel versions of the use case algorithms. Then, we compare the modified LOCs leveraging the GrPPI interface with respect to using directly the supported frameworks. Table 1 summarizes the percentage of modified LOCs in the sequential algorithm in order to implement the parallel versions of the use cases algorithms. As observed, the OpenMP and TBB versions require less LOCs, given that these frameworks provide high-level interfaces hiding away the complexity behind concurrency mechanisms. For instance, OpenMP 4.5 offers the `depend` clause in `task` directives which enforces additional constraints on the scheduling of tasks. However, the analogous implementation in C++ threads requires the use of explicit communication channels (e.g. multiple-produce/multiple-consumer queues) and synchronization mechanisms (e.g. locks, condition variables and atomic variables). On the other hand, using the GrPPI interface for parallelizing a given application is simpler than using directly the above-mentioned programming frameworks. On average, the LOCs that have to be modified in order to incorporate an advanced GrPPI pattern, is 28%. An additional advantage of GrPPI is its capability to easily switch among execution frameworks, since it is only required to replace a single argument in the pattern function call.

Table 1. Percentage of modified lines of code w.r.t. the sequential version.

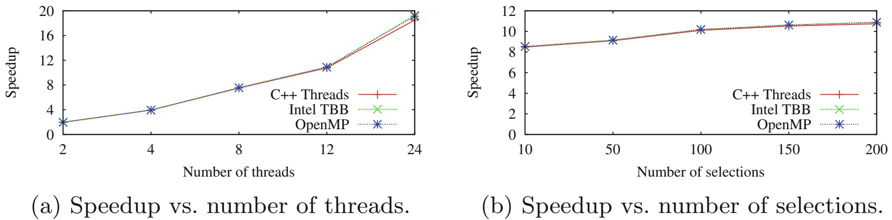
Advanced pattern	% of modified lines of code			
	C++ Threads	OpenMP	Intel TBB	GrPPI
Pool	+55.0%	+70.0%	+55.0%	+22.5%
Windowed-Farm	+152.1%	+75.8%	+51.7%	+31.0%
Stream-Iterator	+153.5%	+56.4%	+46.1%	+30.8%

5.2 Performance Analysis of the Pool Pattern

Next, we evaluate the Pool pattern on a benchmark that solves the TSP problem using a population of 50 individuals representing feasible routes. We also set the benchmark to perform a total of 200 iterations, each of them making 200 selections. Figure 2(a) shows the performance gains when varying the number of threads, from 2 to 24, and using the three available GrPPI back-ends: C++ threads, OpenMP and Intel TBB, with respect to the sequential version. As

can be seen, the speedup increases roughly at a linear rate when increasing the number of threads for all frameworks. Concretely, we observe that between 2 and 12 threads the efficiency is sustained in the range of 91%–98%. However, for 24 threads the frameworks OpenMP and Intel TBB deliver an efficiency of 80%, while for C++ threads it slightly decreases to 77%. This is mainly due to the better resource usage made by the OpenMP and Intel TBB runtime schedulers.

As a complementary evaluation, we set the number of threads to 12 and vary the number of selections from 10 and 200. According to the results shown in Fig. 2(b), the speedup grows hand in hand with the number of selections, since the Pool pattern only parallelizes the selection and evolution functions. This indicates that increasing the number of selections improves the load balance among the worker threads and pays off the parallelization overheads related to thread synchronizations and communications.

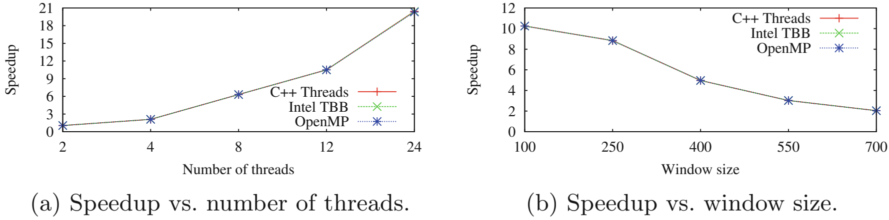


(a) Speedup vs. number of threads. (b) Speedup vs. number of selections.

Fig. 2. Pool speedup varying with varying number of threads and selections.

5.3 Performance Analysis of the Windowed-Farm Pattern

In this section, we evaluate the performance of the Windowed-Farm using a synthetic benchmark that computes average window values from an input stream of sensor readings. Specifically, the sensor in this benchmark has been configured to read samples at 1 kHz and the pattern window size has been set to 100 elements with 90% of overlap among windows. Figure 3(a) shows the speedup when the number of threads increases from 2 to 24. The main observation is that all execution frameworks scale with the increasing number of threads and behave similarly, given that the OpenMP and Intel TBB runtime schedulers do not provide any major advantage over the C++ threads implementation in this concrete use case. This is because the internal Farm pattern leads, by nature, to well balanced workloads among threads. Note that a Farm is comprised of a pool of threads that constantly retrieve items from the input stream and apply the same function over them. On the other hand, we also observe an almost linear scaling for increasing number of threads. This is mainly caused because the Farm pattern can theoretically scale up to $\frac{T_f}{T_a}$, being T_f the computation time of the window average value and T_a the interarrival time of windows in the input stream. To demonstrate this strong scaling, we experimentally measured the computation time of the average function, which was, on average, 220 ms



(a) Speedup vs. number of threads. (b) Speedup vs. window size.

Fig. 3. Windowed-Farm speedup with varying number of threads and window size.

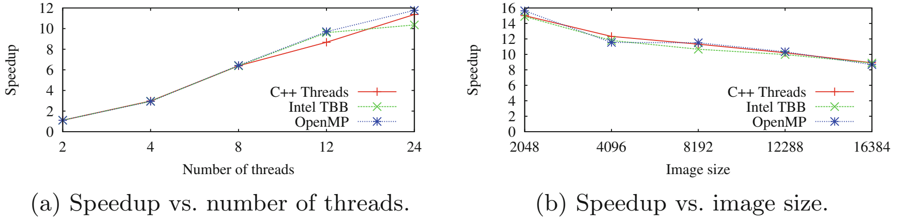
and the interarrival window time that was 10 ms. Therefore, applying the aforementioned formula, we get 22 as for the maximum theoretical speedup.

As an additional experiment, we evaluate the behavior of the Windowed-Farm pattern when increasing the window size, using 12 threads and the aforementioned configuration that uses a fixed overlapping factor of 90%. As can be observed from Fig. 3(b), the speedup decreases for increasing window sizes, as the number of non-overlapping items among windows also increases. This basically occurs because the interarrival time of window T_a increases, restricting proportionally the maximum parallelism degree.

5.4 Performance Analysis of the Stream-Iterator Pattern

Finally, we analyze the performance of the GrPPI Stream-Iterator pattern using the above-mentioned benchmark, in charge of processing square images and halving their sizes on each iteration until reaching concrete resolutions. Specifically, the size of the input images is fixed to 8,192 pixels, and the output images, for each input, have sizes of 128, 512 and 1,024. Figure 4(a) illustrates the benchmark speedup when varying the number of threads from 2 to 24 for the different GrPPI back-ends. In this case, when the number of threads ranges between 2 and 12, the efficiency attained is roughly 75%, while for 24 this is degraded to 48% for all programming frameworks. This effect is mainly caused by the fact that each of the threads involved in the Farm pattern, part of the Stream-Iterator, are simultaneously accessing to different input images. Therefore, these memory accesses become a bottleneck due to constant cache misses when the threads perform the computation of the `task` function of the pattern. In general, these results suggest a memory bandwidth limitation in this particular benchmark.

To gain insights into the performance degradation detected in the previous analysis, we perform an additional experiment in which we set the number of threads to 24 and vary the input image sizes from 2,048 to 16,384. Figure 4(b) depicts the performance gains for the different execution frameworks when varying the image size in the preceding range. Again, we observe a slight speedup decrease for increasing image sizes, which confirms our prior impressions. As an example, if we assume 22 worker threads in the internal Farm pattern, individually processing images with resolution of 2,048 × 2,048 pixels (represented with matrices of integers), these require about 352 MiB of memory. Therefore, not



(a) Speedup vs. number of threads. (b) Speedup vs. image size.

Fig. 4. Stream-Iterator speedup with varying number of threads and image size.

fitting in any of the available cache levels and leading to an increased L2/L3 cache miss rate when they are simultaneously accessed. All in all, this issue is mainly due to the inherent memory-bound nature of this specific use case.

6 Conclusions

In this paper, we have extended GrPPI, a generic and reusable parallel pattern interface, with the advanced parallel patterns Pool, Windowed-Farm and Stream-Iterator, targeted to domain-specific applications. With the unified interface, thanks to the use of C++ templates and metaprogramming techniques, these patterns can be executed in parallel using any of the currently supported back-ends: C++ threads, OpenMP and Intel TBB. Furthermore, their compact design facilitates the development of the domain-specific applications, improving at the same time their portability and maintainability.

As demonstrated through the experimental evaluation, the use cases implemented with the proposed patterns attain remarkable speedup gains compared with their corresponding sequential versions. Although in some cases, the parallelism degree is limited by the pattern nature. We also proved that leveraging GrPPI reduces considerably the number of LOCs that have to be modified in the original codes to turn them parallel with respect to using the parallel frameworks directly. In general, we believe that these advanced patterns can eventually be incorporated in domain-specific applications so as to easily parallelize them, without having a deep understanding of existing parallel programming frameworks or third-party interfaces.

As future work, we plan to support other advanced parallel patterns in GrPPI, such as the *keyed stream farm*, *stream pool* and *image convolution*. Furthermore, we intend to include other execution environments as for the offered parallel frameworks, e.g., FastFlow or SkePU. An ultimate goal is to provide support for accelerators via CUDA Thrust and OpenCL SYCL.

Acknowledgements. This work was partially supported by the EU project ICT 644235 "RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the project TIN2013-41350-P "Scalable Data Management Techniques for High-End Computing Systems" from the *Ministerio de Economía y Competitividad*, Spain.

References

1. MALLBA geographically distributed environments: combinatorial optimization library (2000). http://www.cs.upc.edu/~mallba
2. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Pool evolution: a parallel pattern for evolutionary and symbolic computing. Int. J. Parallel Program. **44**(3), 531–551 (2016)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, pp. 261–280. Wiley, Hoboken (2017)
4. Beard, J.C., Li, P., Chamberlain, R.D.: RaftLib: a C++ template library for high performance stream parallel processing. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2015, pp. 96–105. ACM, New York (2015)
5. Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., Loidl, H.W.: A survey of high-level parallel programming models. Technical report HW-MACS-TR-0103. Department of Computer Science, Heriot-Watt University, December 2013
6. Bucur, D., Iacca, G., Squillero, G., Tonda, A.: An evolutionary framework for routing protocol analysis in wireless sensor networks. In: Esparcia-Alcázar, A.I. (ed.) EvoApplications 2013. LNCS, vol. 7835, pp. 1–11. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37192-9_1
7. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. Int. J. Parallel Program. **45**(2), 382–401 (2017)
8. Diaz, J., Muoz-Caro, C., Nio, A.: A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans. Parallel Distrib. Syst. **23**(8), 1369–1386 (2012)
9. Enmyren, J., Kessler, C.W.: 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, HLPP 2010, pp. 5–14. ACM, New York (2010)
10. Gajda-Zagórska, E.: Multiobjective evolutionary strategy for finding neighbourhoods of pareto-optimal solutions. In: Esparcia-Alcázar, A.I. (ed.) EvoApplications 2013. LNCS, vol. 7835, pp. 112–121. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37192-9_12
11. ISO/IEC: Programming Languages - Technical Specification for C++ Extensions for Parallelism, July 2015. iSO/IEC TS 19570:2015
12. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: Hpx: a task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 6:1–6:11. ACM, New York (2014)
13. Khronos OpenCL Working Group: SYCL: C++ Single-source Heterogeneous Programming for OpenCL. https://www.khronos.org/sycl. (Accessed May 2015)
14. Kist, D., Pinto, B., Bazo, R., Bois, A.R.D., Cavalheiro, G.G.H.: Kanga: a skeleton-based generic interface for parallel programming. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), pp. 68–72, October 2015
15. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Boston (2004)
16. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)

17. NVIDIA Corporation: Thrust. https://thrust.github.io/
18. Popovic, V., Seyid, K., Pignat, E., Çogal, Ö., Leblebici, Y.: Multi-camera platform for panoramic real-time HDR video construction and rendering. J. Real-Time Image Process. **12**(4), 697–708 (2016)
19. Reinders, J.: Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly, Sebastopol (2007)
20. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurr. Comput.: Pract. Exp. **29**, e4175-n/a (April 2017)
21. Shan, A.: Heterogeneous processing: a strategy for augmenting Moore's law. Linux J. **2006**(142), 7 (2006)