International Journal of Parallel Programming
https://doi.org/10.1007/s10766-020-00684-w

# Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming

Marco Danelutto[1] · Gabriele Mencagli[1] · Massimo Torquati[1] · Horacio González–Vélez[2] · Peter Kilpatrick[3]

Received: 13 August 2019 / Accepted: 8 October 2020


**Abstract**
This paper discusses the impact of structured parallel programming methodologies in state-of-the-art industrial and research parallel programming frameworks. We first recap the main ideas underpinning structured parallel programming models and then present the concepts of algorithmic skeletons and parallel design patterns. We then discuss how such concepts have permeated the wider parallel programming community. Finally, we give our personal overview—as researchers active for more than two decades in the parallel programming models and frameworks area—of the process that led to the adoption of these concepts in state-of-the-art industrial and research parallel programming frameworks, and the perspectives they open in relation to the exploitation of forthcoming massively-parallel (both general and special-purpose) architectures.

**Keywords** Algorithmic skeletons · Parallel design patterns · High performance computing · Multi-core architecture · Parallel computing

## 1 Introduction

In the last two decades, the number of parallel architectures available to the masses has substantially increased. The world has moved from clusters/networks of workstations—composed of individual nodes with a small number of CPUs sharing a common memory hierarchy—to the ubiquitous presence of multi-core CPUs coupled with different many-core accelerators, typically interconnected through high-bandwidth, low-latency networks. As a result, parallel application programmers now face the challenge of targeting hundreds of hardware-thread contexts, possibly associated to thousands of GP-GPU cores or, for top500-class architectures, millions of cores. These hardware features exacerbate the "software gap," as

Extended author information available on the last page of the article

Published online: 09 November 2020

they present more substantial challenges to skilled application programmers and to programming framework developers.

The need for programming models and programming frameworks to ease the task of parallel application programmers is therefore acute. A number of "*de facto* standards"—OpenMP for shared memory architectures, CUDA and OpenCL for GP-GPUs, MPI for distributed clusters—are widely recognised. Furthermore, other higher-level frameworks such as IntelTBB or Microsoft PPL have been recognised as emerging touchstones. Arguably, such frameworks build upon—and to different degrees, recognise roots in—research results from *structured parallel programming*.

Our chief contribution in this paper is to provide an outline of the main results from algorithmic skeletons and parallel design patterns that have been migrated to industrial-strength parallel programming frameworks. They have arguably contributed to the acceptance and success of these frameworks along with the outlining of the possibilities still to be explored in the area. The paper is not intended to be a comprehensive survey, but rather to trace how laboratory results from the research community have percolated through industrial strength programming models. As such, our contribution is quite different from both the contributions of survey papers (e.g. [35]) and of books dealing with pattern abstractions [42] or pattern implementations with existing parallel programming frameworks [43]. Aside from being of historical interest to those with an interest in structured parallel programming, the work may serve to inform the efforts of those charged with addressing the programmability of next generation computing systems.

## 2 Structured Parallel Programming

*Structured* parallel programming encompasses programming models where concurrency can only be expressed and orchestrated via *structured* compositions of parallel components representing notable "forms" of parallel computations. Similar to its sequential counterpart, its structural quality adds new organic constructs while banning programming practices that may be popular in the non-structured world. Structured parallel programming fosters not only more maintainable code, but also parameterisable constructs with predictable performance.

In sequential programming, *if-then-else*, *switch-case*, and *repeat-until* or *while-do* have replaced the indiscriminate use of *goto*s and made possible the implementation of compiler tools that have closed the gap between compiled code and handwritten assembly code in terms of efficiency.

In parallel programming, the use of algorithmic skeletons and parallel design patterns—such as the *map* (and *parallel for*), *pipeline*, *farm* (and *master worker*) and Google's *MapReduce*—has been gaining momentum, confining primitive message passing, shared memory mechanisms, and low-level programming models to ad-hoc deployments.

Launched in the '90s through the algorithmic skeleton concept, structured parallel programming was boosted in the early '00s with the introduction of parallel design patterns, which we briefly recap in the next two sections.

### 2.1 Algorithmic Skeletons

The algorithmic skeleton concept was introduced by Cole in his PhD thesis in the late '80s [13] and subsequently adopted by a number of research groups, mostly in Europe and Japan. Having arguably emanated from the HPC community, the skeleton concept is intended to address the difficulties relating to the implementation of efficient parallel applications by separating concerns:

- the application programmer's responsibility is to select suitable compositions[1] of algorithmic skeletons by modelling the parallel behaviour of the application using the set of available skeletons; and,
- the system programmer's responsibility is to provide suitable, efficient, and composable skeletons that support—alone or in composition—the efficient parallel implementation of the vast majority of real world parallel applications.

Algorithmic skeletons are therefore provided to application programmers as predefined abstractions analogous to a "host-based" sequential programming environment: higher-order functions, classes, or, simply, imperative library interfaces (Fig. 1 shows the way FastFlow [18, 29] presents parallel patterns to the application programmer as plain C++ objects, as an example). Since Cole's initial work, a number of parallel programming frameworks have been proposed and implemented [35], whose evolution is discussed in Sect. 3.

### 2.2 Parallel Design Patterns

Design patterns were introduced in the '90s by the software engineering community as a viable solution to increase programming efficiency as well as software quality and maintainability.

A design pattern is a "recipe" that addresses some specific programming scenario. The canonical "Gang of Four" book [30] describes object-oriented patterns targeting programming situations programmers face when designing object-oriented applications. A design pattern carefully describes a (set of) problem(s) along with the set of solutions that may be put in place to solve it (them) (see Table 1).

The design pattern concept migrated to the parallel programming world in the early '00s. It was welcomed by practitioners and researchers for the support of programming frameworks, mainly in the US with notable industry representatives. While there have been some incipient attempts to establish a mapping between parallel design patterns and algorithmic skeletons [12], it is now recognised that research on parallel design patterns has subsumed the fundamental idea of algorithmic skeletons, i.e. to provide programmers with ready-to-use parallel programming abstractions and currently various parallel "pattern" libraries exist and are used to implement parallel applications.

Further structuring in parallel patterns is introduced in [42]. Besides considering patterns modelling structured parallelism exploitation, it introduces a *design pattern*

[1] Compositionality has been added after Cole's initial proposal [13].

```
int main() {
   // declare three stages wrapping seq functions ...
   ff_node_F<data_t> first ([&](data_t* in, ff_node* thisnode) { ... });
   ff_node_F<data_t> middle([&](data_t* in, ff_node* thisnode) { ... });
   ff_node_F<data_t> last  ([&](data_t* in, ff_node* thisnode) { ... });

   ff_Pipe pipe(first, middle, last); // build a pipe of the three stages
   pipe.run_and_wait_end();           // run pipe and await termination
   return 0;                          // all done
}
```

**Fig. 1** Sample FastFlow 3 stage pipeline application code

**Table 1** Sketch of sample pipeline parallel pattern (the whole pattern may easily take tens of pages to be properly described [43])

| NAME | Pipeline |
|---|---|
| PROBLEM | Computations organised in stages, over large number of independent data sets |
| EXAMPLES | Assembly line, processor instruction fetch-decode-execute cycle, video frame processing with multiple filters, etc. |
| FEATURES | Input/output ordering, stage load balancing, parallel stage computation, ... |
| OPTIMISATION STRATEGIES | Merge adjacent stages with sum of service time smaller than service time of the previous or successive stage. Parallelise stages with higher service time, by replicating stage executor in parallel |
| IMPLEMENTATION | Set up a chain of parallel activities, each one processing a stage, receiving input data from previous stage and delivering results to next stage through proper single-producer single-consumer message queues. Alternatively, represent stages as tasks and organise pipeline computations with a thread/process pool computing "ready" stages |
| SAMPLE IMPLEMENTATION | Sketch of multi-threaded code computing an n-stage pipeline. Sketch of MPI code computing an n-stage pipeline |

The problem solved is outlined, along with possible implementation and optimisation strategies

*space* hierarchy. The hierarchy includes four spaces (finding concurrency, algorithm design, implementation structures, and execution mechanisms) where each of the spaces provides patterns solving specific problems. Namely:

- the problem of determining the possibilities to exploit parallelism (i.e. where is concurrency);
- the problem of modelling parallelism exploitation through suitable algorithms;
- the problem of efficient implementation of some parallel algorithm; and,
- the identification of the mechanisms to be used in the parallel application implementation, once all other aspects have been addressed in the higher level design spaces.

### 2.3 Strengths and Weaknesses of Structured Parallel Programming

Structured parallel programming frameworks, either based on the algorithmic skeleton or on the parallel design pattern concept, provide the application

programmer with advanced and useful abstractions that can be used to build parallel applications via a clear and productive workflow (see Fig. 2):

1. the application programmer identifies the parallelism exploitation opportunities of the application at hand;
2. he/she identifies which patterns or pattern compositions are suitable to model these parallelism opportunities;
3. he/she uses the knowledge base associated with the pattern(s), in the form of performance models or good practice heuristics, to identify the possibilities offered by the pattern (composition) on the target architecture considered;
4. he/she codes the application; and finally
5. performs functional (correctness of the business logic code[2]) and non-functional (performance) debugging with the usual compile-run-debug cycle.

The workflow is slightly different in the case of pure algorithmic skeletons and of pure parallel design pattern frameworks (see Fig. 3). The main differences consist in the effort required of the application programmer, which is lighter in the case of algorithmic skeletons, as in this case the effort needed to efficiently implement parallel exploitation patterns is completely the responsibility of the system programmer implementing the algorithmic skeletons rather than of the application programmer. The application programmer's major duty is to select the appropriate skeleton (skeleton composition). All the implementation effort is hidden in the skeleton library/DSL implementation. The workflow is not radically different from that adopted when using a classical, non-structured parallel programming environment. In that case, however, there is no pattern library: the pattern expression phase consists in a full implementation of the parallelism exploitation mechanism identified by the programmer and the refactoring phase may require substantial coding effort.

Clearly, the adoption of a structured parallel programming model has both strengths and weaknesses which are worth begin recapped and commented upon in detail.

*Expressivity* The availability of primitive mechanisms and abstractions for fully expressing complex parallel activity orchestrations greatly enhances the expressivity of the programming model. Programmers may use advanced abstractions to model the parallel activities they identified in the application at hand rather than being forced to use low level mechanisms (threads, processes, communications, synchronisations, shared data structures, etc.). Figure 1 evidences this fact. The simple use of a *ff_Pipe* object subsumes the fact that a thread is forked to host each of the three stages, communication queues are established to move data between stages and appropriate shared memory access mechanisms are used to implement communications and sharing. All this with no application programmer intervention at all!

*Rapid prototyping* The possibility to program parallel applications without the need to deal with all the necessary low level detail related to use of parallelism

[2] It is assumed that the correctness of the parallel pattern implementation has been confirmed by the system programmer.

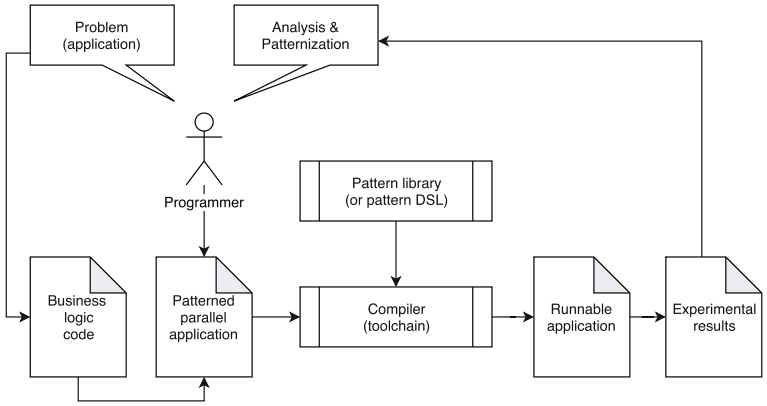


**Fig. 2** Structured parallel application development workflow

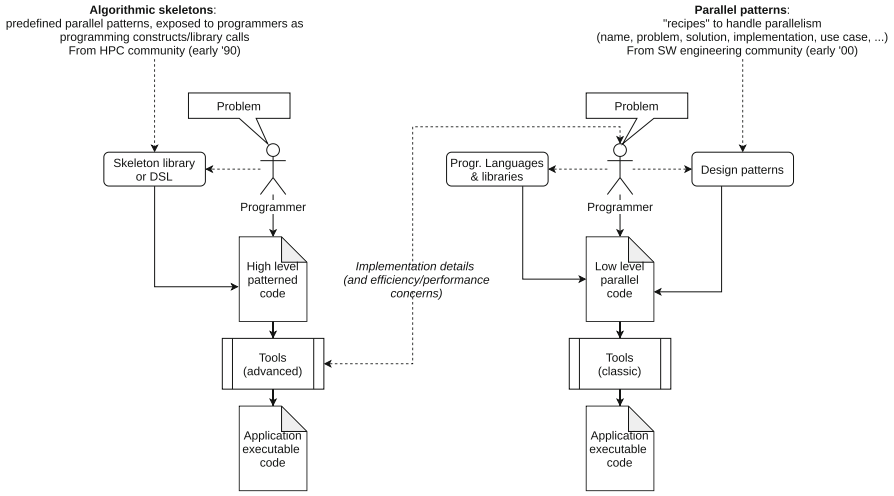


**Fig. 3** Algorithmic skeletons (left) and parallel design patterns (right) typical user workflow

clearly greatly enhances the possibility of achieving rapid prototyping. In addition, changes to the parallel structure of an application may be simply achieved by minimal changes in the abstractions used to program the pattern composition modelling the parallel behaviour of the application at hand. If we discover that the middle stage of the pipeline of Fig. 1 is too slow, we may simply replace the line 4 of the listing with the line:

```
ff_Farm<data_t> middle([&](data_t* in, ff_node* thisnode) { ... },
  nworkers);
```

that installs a parallel computation of the middle stage with parallelism degree equal to *nworkers*. This step *de facto* uses the refactoring rule:

$$seq(f) \equiv farm(seq(f), pardegree) \qquad pardegree > 1 \text{ and } stateless(f)$$

stating that farms can be freely introduced/eliminated (left to right and right to left usage of the rule) on stateless stages preserving functional semantics. *seq(f)* denotes generic sequential code computing function *f*.

*Parallelism exploitation correctness "by construction"* The correctness of the parallelism exploitation mechanisms and policies is guaranteed by the system programmer implementing the parallel patterns provided through the framework along with the pattern composition mechanisms. The application programmer is not required to consider any of the related problems. Rather, he/she can focus on guaranteeing that the business logic code used as parameter for the patterns respects the specifications stated by the pattern framework. As an example, the refactoring rule stating that a sequential pipeline stage may be safely replaced by a parallel farm with sequential workers—the one applied to Fig. 1 code in the previous paragraph—may be safely applied only in the case of "stateless" stages. It is up to the application programmer to check/ensure the *second_stage* is stateless, in this case. If this happens, then the correctness of the second version of the program—as the correctness of the first version—is guaranteed "for free" from the application programmer viewpoint.

*Performance portability* The exposition of the complete parallel structure of the application enables the use of any known effective and applicable rule, policy or heuristic when porting an application from one target architecture to a different one. As an example, moving from architecture *A* to architecture *B* such that the communication and synchronisation costs on *B* turn out to be significantly higher than those on *A*, any rules coarsening the grain of the parallel computations may be automatically applied to preserve application vertical scalability on architecture *B*. For example, consider a video stream processing application structured as a pipeline of filters, with each filter programmed as a data parallel map pattern over the frame pixels. When moving from a standard shared memory multi-core architecture to a high speed cluster, the data parallel pattern may be automatically refactored out of the pattern expression modelling application parallelism[3] or individual computations of different, consecutive filter stages may be merged into a single, coarser grain stage[4] to enhance grain and therefore increase application efficiency. Similar reasoning can also be applied to various hardware accelerators, including GP-GPUs and FPGAs [40, 46] and, potentially, enable a performance continuum into more distributed systems such as clouds [8]. It is worth pointing out that, provided the set of valid refactoring rules, the policies and the heuristics are suitably tagged with models qualitatively describing their effect w.r.t. target architecture features, the adaptation of the implementation of the parallel application may be performed independently of the declaration of its pattern structure as initially provided by the application programmer. Hardware targeting may or may not require pattern structure refactoring and may in general be

[3] Using the rule $(map(f))(X) \equiv (seq(\forall\, x \in X\ do\ x = f(x)))(X)$.

[4] Using the classical map fusion rule $map(f) \circ map(g) \equiv map(f \circ g)$, the composition of two maps computes the same result as a single map of a function which is the composition of the two map functions.

implemented in the compiler—runtime tool-chain, completely transparently to the application programmer.

*Optimisations* Pattern expressions, that is, the parallel structure of an application expressed using a single pattern or a pattern composition, are the only logically parallel code of a structured parallel application. A number of refactoring rules apply to stateless or stateful parallel pattern compositions. These rules may be used to explore a possibly large space of alternative pattern expressions that may be used to express the same functional computation with different non-functional features, e.g. different performance, power consumption, resource usage, fault tolerance or security properties, etc. Various tools have been developed to support structured parallel application refactoring [32, 34, 44]. Some of these tools apply or exploit rules that are well known in the application auto-tuning context. However, the exposition of the parallel structure of the program opens improved possibilities for optimisation through refactoring, and in particular:

- the possibility to implement semi-automatic exploration of the functionally equivalent pattern expression alternatives modelling the parallel behaviour of the application at hand;
- the possibility to automatically adapt the parallelism degree of the application to the available resources; and
- the possibility to implement automatic code generation starting from selected and certified "functional" business logic code portions.

As an example, the reader may consider the work proposed in [31] that outlines the main features of a tool that can be used to support application programmer directed exploration of the space of pattern alternatives available for a given application and also provides basic support for the automatic generation of structured parallel code out of a syntactically simple pattern expression—either provided by the application programmer or derived using the refactoring and optimisation rules included in the tool—and the original, suitably wrapped[5] business logic code.

*Rigidity of the pattern set* Very often the structured parallel programming frameworks have been criticised for the fact that the set of patterns provided to the user is fixed. Murray Cole already pointed out in his famous *manifesto* [14] that algorithmic skeletons should "integrate ad-hoc parallelism" and "accommodate diversity", meaning that:

> It is unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way.

concerning *ad-hoc* parallelism; and, with respect to the need to specify slightly different parameters in different computations employing the same general parallel pattern (accommodate diversity):

[5] The wrapping is needed to expose features to be used while composing the business logic components into general application code.

> We must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility. This is not a quantifiable trade-off.

Indeed, most of the structured parallel programming frameworks currently available either prohibit mixing the patterns provided with primitive parallel constructs or warn users that the efficiency and the optimisation of the framework patterns may be impaired/contaminated by this "hybrid" usage of the framework. Various kinds of workaround have been proposed to overcome the problem, as this is a really important issue in those cases where the patterns provided by the structured parallel programming framework do not suffice to match application programmer expectations. Proposed solutions include "escape" patterns encapsulating and controlling the interaction with the rest of the system of user-defined parallel patterns using base parallel mechanisms, or the provision of access for application and system programmers to a lower level of pattern building blocks that can be used to compose, orchestrate and implement alternative patterns in a flexible, safe, and efficient way [1].

Moreover, in many structured parallel programming frameworks, such as Intel TBB [47] or SkePU [27], nothing prevents programmers from using threads outside the parallel patterns they provide.

*Minimal disruption effort* Structured parallel programming frameworks have been provided in several distinct ways, including new languages and libraries. In the former case, application programmers are required to learn brand new syntax rules and to set aside their favorite programming abstractions to embrace the model imposed by the "new language" structured programming framework. In the latter case, application programmers may preserve all their knowledge and best programming practices as the structured abstractions are provided as abstractions of the existing (sequential) programming language/model, thus ensuring the "minimal disruption" principle advocated by Cole in his *manifesto*. It is of fundamental importance that the new programming environments preserve the possibility to re-use the existing programmer knowledge base, for two fundamental reasons:

1. first, due to the fact that changing to "yet another programming language" would require substantial time to have a decent set of expert programmers, independently of the importance and effectiveness of the new parallel programming model; and
2. second, due to the fact that code reuse may be of great help in guaranteeing the possibility to move into a new programming model, and code reuse is definitely much easier when libraries are involved than when new programming languages are designed and implemented.

## 3 Research Programming Framework Evolution

We present a short review of the history of structured parallel programming models and frameworks in order to introduce the discussion on state-of-the-art parallel programming environments in relation to the "structured" parallel programming concepts of Sect. 4. We distinguish three different phases: *pioneering*, *colonisation* and *integration* phases.

*Pioneering* In the late '90s, several structured parallel programming environments based on the algorithmic skeleton concept were proposed. These were the first examples of this kind of framework, often conceived as "proof-of-concept" functional programming prototypes [49]. Some of them did not support composition of skeletons [6]. Some were conceived as brand new programming languages [45] or extension to existing ones [19], and some supported business code reuse from other languages [7]. No optimisations, affecting and possibly improving the skeleton expression used by the application programmer, were supported, either static or dynamic. In most cases, the number of patterns supported as skeletons was limited and basically none of these frameworks provided the possibility to program your own skeletons/patterns if those provided did not fulfil your needs. The success of these early programming frameworks was limited, their impact lying chiefly in the fact that several research groups became interested in the concept and started evolving the design and development of structured parallel programming frameworks along various lines.

*Colonisation* From the late '90s on the algorithmic skeleton concepts were sufficiently mature to lead to the design and implementation of programming frameworks that provided good programmabilty, excellent performance and the capacity to target different kinds of parallel architectures. Meanwhile, the parallel design patterns community emerged and began influencing the parallel programming landscape. However, the two research communities operated more or less independently, with little evidence of synergy. The net result was the appearance of quite a number of frameworks that presented very attractive features even if they were still restricted in usage and adoption by groups not already involved in structured parallel programming. This notwithstanding, Muesli [26] provided support to target both shared memory and cluster architectures, SkeTo [24] introduced automatic optimisations of data parallel computations through conscious exploitation of a map fusion refactoring rule, OSL [38] brought into structured parallel programming the BSP model, SkePU [25] supported GP-GPU accelerators, Lithium [17] introduced the macro data flow based implementation model and Muskel [15] introduced the concept of autonomic management of non-functional features as well as the first opportunity for user defined customisation of the skeleton set.

We regard this phase as a kind of "colonisation" era, as the key point is that the main concepts that make the structured parallel programming models attractive and efficient had mostly been developed but their exploitation was still largely confined to the community.

*Integration* From the late '00s, several new developments changed the scene, in our opinion. Different structured programming frameworks fully embraced modern sequential "host" languages, such as C++. Muesli, FastFlow and SkePU adopted and integrated the programming facilities provided through the C++11 standard, both as valid business logic code abstractions and as implementation tools and mechanisms. This resulted in more modern and effective opportunities to convince parallel application programmers to adopt the structured parallel programming model fully, thus realising Cole's minimal disruption principle.

Different target architectures are fully addressed, including GP-GPUs but also FPGAs, which makes the structured programming model effective in dealing with the more advanced architectures available. Finally, structured parallel programming models have been adopted in several EU-funded research projects (SkePU in Peppher [4] and Excess[6], FastFlow in ParaPhrase [37], REPARA[7] and RePhrase[8] projects) that spread adoption across different industrial contexts and therefore ensured a wider diffusion of their concepts.

This is the "integration" phase, where concepts born, grown and matured in the structured parallel programming research enclave begin to permeate other communities, based on the consolidated results achieved so far. A notable result is the porting of the PARSEC benchmark suite onto a structured parallel programming framework with excellent performance results achieved in the recent past [20].

## 4 Industrial Strength Programming Frameworks

While different phases were elapsing in the structured parallel programming scenario, developments were also occurring in industry based parallel programming frameworks. In this section we discuss examples of programming frameworks that inherited from structured parallel programming research results, even if in most cases the common features have been proposed as brand new and the research results from structured programming community have been little acknowledged. For example, Google's MapReduce clearly builds on results that were investigated in the '80s (as discussed later in this Section). Besides, Microsoft PPL includes most of the patterns that were commonly included in algorithmic skeleton parallel programming frameworks already available in the last years of the past century, such as Muesli, SkeTo, SkePu, Lithium, as well as in P3L [16] and, more recently, in FastFlow and GrPPI [22]. We aim to cover the widest scenario respecting the paper size, and we selected different kinds of programming framework: Google MapReduce is included as it was the first widely used programming framework leveraging the parallel pattern concept [21]. OpenMP and MPI represent *de facto* standards in HPC (distributed and shared memory contexts [2, 23]), Microsoft PPL [9] and Intel TBB[9] are the only industrial frameworks actually leveraging the

[6] http://www.excess-project.eu/.

[7] http://repara-project.eu/.

[8] https://rephrase-eu.weebly.com/.

[9] TBB is no longer supported, as is, but it has been included in the new OneAPI framework (see `https://spec.oneapi.com`) as an integral part of the library, also providing classical data parallel

parallel pattern concepts. Finally, Storm is mentioned here as a representative of a range of programming environments providing stream processing support (e.g. [50, 51]). These frameworks support a similar set of "structured patterns" while delivering different levels of efficiency in their implementation on distributed architectures. Table 2 outlines some relevant features of the frameworks, discussed qualitatively in the following sections.

The table rows capture different features of the programming frameworks considered. Unless otherwise stated, the contents of each row are derived from the framework's available documentation. *Pattern set* lists the range of parallel patterns provided by the framework. *Expressiveness* and *rapid prototyping*, whose evaluation is based on authors' direct experience, discuss the expressive power of the frameworks, and in particular the effort required to implement applications whose parallel behaviour is modeled with (a composition of) the provided parallel patterns, once the business logic code is available. *High level refactoring* indicates which frameworks support the possibility to implement the refactoring of the parallel structure of the application without the need to completely rewrite the application code, either the business logic or the system/parallel code.

*Performance portability* comments on the portability of the programming environments across fairly different, common parallel and distributed architectures, mainly outlining the possibility to run on multi-core hardware (shared memory architectures in the table) and on clusters/networks of workstations (distributed architectures in the table). *Support for extra patterns* considers whether the framework includes support for the programming of extra patterns to be used alone or in conjunction with those that are natively provided. Finally (the *Designer and maintainers* row is included to acknowledge the maintainers of these frameworks), the *ease of use* row reports the authors' personal experience of the "minimal disruption" principle defined in Cole's manifesto [14], that is, the provisioning of parallel programming abstractions through mechanisms and tools as uniform and consistent as possible with the mechanisms and tools provided by the "hosting" (sequential) programming environment.

*OpenMP* OpenMP has been proposed with the intent of fulfilling a "minimal disruption" principle while providing task and data parallel patterns to the shared memory architecture programmer. The initial parallel region and parallel for concepts, introduced through pragmas preserving the sequential semantics of the code, have been immediately successful. The simple pragma enabling independent iterations of a for loop to be executed in parallel and supporting the "reduction" of simple variables through common arithmetic operators provided the means to easily and immediately parallelise a number of existing applications on standard shared memory multi-cores. The section and task constructs support also the possibility to implement task parallel computations, although the parallelisation effort is not as small as that required to introduce data parallelism. OpenMP actually provides map and reduce patterns, with very efficient implementation on shared memory

Footnote 9 continued
patterns, collective communication patterns as well as specific patterns to address coprocessors (GP-GPUs and FPGAs).

**Table 2** Parallel design pattern/algorithmic skeletons inherited properties in *state-of-the-art* parallel programming frameworks

| | Google MapReduce | OpenMP | MPI | Microsoft PPL | Intel TBB | Storm |
|---|---|---|---|---|---|---|
| Pattern SET | Mapreduce only | Parallel for (with reduce options), task (with dependencies ->macro data flow pattern) can be used to implement different patterns | Collective communications (only) + SPMD (implicit orchestration) | Task, data, stream and data flow patterns + parallel loops | High level (parallel for (map), reduce, scan, data flow, pipeline (with parallel stages), ) and low level (mechanisms) | Stream processor (filter network) only |
| Expressiveness | Excellent: Programmers only need to provide map and reduce functions | Good: Mainly granted by pragma based syntax | Very limited: Assortment of implementation and business logic code + implicit SPMD orchestration | Very good: Clean and high level library API(s). Available for different front-end languages | Limited: Full C++ (template) class library. Patterns from different spaces at the same level | Good: few elementary concepts supported (spouts & bolts) for the single pattern provided |
| Rapid prototyping | Fully supported by the single pattern provided | Easy prototyping of data parallel appls. Fine tuning is expensive | Definitely not supported | Yes. Different pattern spaces (task & data parallel, stream) | Requires deep insight in the provided C++ classes | Requires full knowledge of tuple mechanisms |
| High level refactoring | Chains of mapreduce optimisation (Flume) | Nested loop parallelisation only | Communication primitives refactoring | N/A | Automatic pipe of farm implementation | N/A |
| Performance portability (Automatic HW targeting) | Distributed or shared memory HW (in principle) | Shared memory HW only | Distributed and shared memory architectures | Shared memory HW only | Shared memory HW only | Distributed or shared memory HW (in principle) |

**Table 2** continued

| | Google MapReduce | OpenMP | MPI | Microsoft PPL | Intel TBB | Storm |
|---|---|---|---|---|---|---|
| Rigidity of the pattern set: support for extra patterns | No support | Regions & tasks + shared memory synch & comm mechanisms | Through external pattern libraries. Single node intra parallelism via OpenMP (or other libs) | Via Task/Data flow patterns | Via independent task submission to TBB scheduler | No support |
| Ease of use (Minimal disruption) | Good | Good | Poor | Good | Fair | Fair |
| Designer & maintainers | Google (+ open source, Apache) | OpenMP Architecture Review Board (or OpenMP ARB) | MPI Forum | Microsoft | Intel (+ open source version) | Open Source - Apache Foundation |

architectures based on threads, but not providing any kind of global optimisation (e.g. exploitation of map fusion across subsequent parallel for loops). Due to the simplicity of the code required to introduce the provided patterns, OpenMP has become the *de facto* standard for shared memory architectures in parallel programming.

*Intel TBB* Intel TBB provides a number of patterns including stream parallel (pipeline, task farm) and data parallel (parallel for and reduce) patterns. The library also provides lower level patterns such as synchronisation or memory allocation patterns and higher level parallel algorithms (e.g. sort) thus making available a mix of tools that all contribute to support the parallel application programmer in solving a number of the problems he/she has to face when dealing with parallel execution. Intel TBB fully embraces modern C++11 concepts. The availability of most of the commonly needed parallel patterns requires anyway some non-trivial programming effort from the application programmer. However, separation of concerns is guaranteed, modulo the existence of different levels of abstraction in the patterns provided to the application programmer. Efficiency is mainly guaranteed by the extremely efficient thread pool that manages all the parallel computations set up via the library patterns. No automatic optimisation of the parallel structure of the application is supported.

*Microsoft parallel pattern library* Microsoft Parallel Pattern Library (PPL) provides patterns that are available across all the languages supported by Microsoft development tools through CLR (Common Language Runtime). The pattern set provided is wide. It includes data and stream parallel patterns, the possibility to model data flow graphs (workflows), support for task parallelism and constructs similar to those provided originally by OpenMP pragmas to parallelise loops are also present.

The library supports the minimal disruption principle in that it is provided as any other library of the CLI programming environment and users may seamlessly use entries of the library to parallelise complex applications according to the pattern implemented by the library entry.

As far as we know, there is no support for any kind of static or dynamic pattern optimisation in PPL, nor is there support to introduce new patterns when those provided do not fit the application programmer needs. This notwithstanding, the PPL library has proven to be effective in the parallelisation of applications [10] and guidelines are provided to help programmers in using the possibilities of the library[10].

*MPI* MPI still represents the *de facto* standard in the field of (massively) parallel applications targeting distributed architectures, such as networks or tightly coupled clusters of workstations. The standard was designed to include different variants of point to point communications as well as a comprehensive set of collective communications to be used to orchestrate MIMD[11] computations in a SPMD[12] execution environment. While decent efficiency may be obtained by using a

[10] See https://msdn.microsoft.com/en-us/library/ff601930.aspx.

[11] Multiple Instruction Multiple Data.

[12] Single Program Multiple Data.

reasonably small subset of MPI calls, the library requires the application programmer to understand the hundreds of different functions in the MPI collection to optimise applications. Also, MPI leads to code which is a global mix of communication, orchestration and business logic code, making maintenance, debugging or tuning operations much more difficult to tackle than those used in other kinds of parallel environments and frameworks.

This notwithstanding, the advent of multi-core machines and the global availability of GP-GPU accelerators has forced a global evolution of the way MPI programs are architected and so brought the whole story much closer than expected to the structured parallel programming concepts. It is widely recognised [14, 42, 43] that most of the more performant applications running on massively parallel architectures, such as those of the www.top500.org list, share a common parallel structure including:

- some general data parallel orchestration of the overall computations distributed on the cluster nodes implemented using the MPI SPMD model, and widespread use of both collective and point-to-point communications, reflecting the fact that massively parallel computations are basically embarrassingly or quasi-embarrassingly data parallel computations (e.g. computations implementing various kinds of stencil data parallel pattern);
- a finer grain parallel orchestration of the single node sub computation, exploiting multi-core and multi-threading facilities of the node through OpenMP, not necessarily but often in yet another context of data parallel computation;
- possibly, some offloading of the third level of data parallel computations to one or more node-attached GP-GPU nodes.

Overall this contributes to the definition of parallel computations that may be re-read as patterned data parallel computations (maps of maps of maps) where the known data parallel pattern optimisation rules have been deployed to target any level of the hardware available with the most suitable computation grain that can be used with the given business logic code.

However, even in the case of MPI computations, optimisations are still an “art” which is required of the application programmer, although some research tracks are actively seeking possible automatic optimisations of MPI computations (see for instance [28])

mostly looking for automatic identification of the most appropriate communication mechanisms to be used once the overall parallel structure of the application and, possibly, the related computation grains are known.

*Google MapReduce* Google’s MapReduce has many features that can be obviously related to structured parallel programming concepts. Optimisation of maps followed by reduces has been subject to intense mathematical work in the ’80s [5] and the same results have been employed in the structured parallel programming frameworks in the ’90s by Gorlatch [36], as an example. Google’s main addition to these results consists in two simple but extremely effective enhancements: (1) the addition of the key—value pair concept as the result of the map phase, which, with the shuffling and key-directed reduce phases, enables several distinct opportunities

to enhance the exploited parallelism, and (2) the in-place application of the map-reduce pattern to massive amounts of data spread among different cluster components to incorporate fault resilience. This is complemented by an efficient implementation of the pattern, which includes a very effective "shuffle" communication pattern connecting the map with the reduce phases and the support for large ("big data") datasets.

The success of the MapReduce framework has been notable. So many different applications have been developed and successfully run on top, which is a perfect illustration of the fact that requiring application programmers to provide just the business logic code portions and using them to fill parameters of an existing, efficient implementation of a pattern is a real success story. In addition to this, Google MapReduce has been subject to research aimed at optimising compositions of map-reduces (e.g. see Flume work [11]) much in the perspective of the algorithmic skeleton optimisation.

*Storm* Storm provides stream processing as the main programming model. The graph of filters and transformers applied to a set of possibly infinite input streams may be arbitrarily structured by merging and splitting streams in different places according to the application needs, or even processing any one of the input streams with its own dedicated filters and transformers. The emphasis in Storm is on throughput. Despite the fact that the user may define arbitrary topologies made of "spouts" (stream sources) and "bolts" (stream processors), the structure of Storm parallel computations is somehow restricted by the stream item flow and by the data flow semantics associated with stream items. Simple performance models can be built for Storm applications once the semantics and timings of the business logic of the spouts and bolts are known. Also, Storm topologies may be refactored and, possibly, optimised due to the clear semantics of the different components used to build these topologies and to the complete absence of topology side effects.

## 5 Programming Frameworks Perspectives

We have discussed how in a number of cases results achieved and assessed in the structured parallel programming research arena have been absorbed, adopted and exploited in programming frameworks used throughout the software industry.

Putting the trend in perspective, we now outline three different evolutions of this scenario, characterised by the focus taken when dealing with design, implementation and deployment of parallel programming frameworks.

*User centric view* Users of parallel programming frameworks are application programmers. Application programmers are typically comfortable in using some programming language *and* are experts in the domain of the applications they program. They rarely are parallel programming experts nor, typically, do they know all the hardware details of the target architecture, i.e. those worth careful consideration and those causing inefficiencies during parallel execution unless particular attention is paid while programming the application. Structured parallel programming may help in this case by leveraging natural possibilities offered for high level syntax design, once a complete set of general purpose parallel patterns is

provided and efficiently implemented. The parallel pattern set may be used to provide the exact pattern compositions of interest in the application domain at hand as further level programming abstractions. These abstractions will be closer to the application programmer's way of reasoning, in that they provide more general parallel abstractions w.r.t. those provided by the underlying parallel libraries. Therefore they are easier to use, they may be directly compiled to the underlying pattern set and, as a consequence, they will be easy to implement (efficiently). The same process requires, in classical parallel programming environments, the design and implementation of full DSLs, either internal or external, which usually is a more demanding process. It is worth pointing out that hardware advances will, in the near future, follow a similar approach, using the available resources to provide specialised architectures rather than more powerful general purpose architectures (see [39]).

*Tool centric* The exposition of the full parallel structure of an application through the pattern expression used to model its parallel semantics allows application of different optimisation strategies. On the one hand, the application programmer may be given a rough description of the parallel semantics of the patterns, modelling the bare minimum notions relative to computation dependency necessary to express the semantics of what is going to happen in parallel. Then, the application programmer may be asked to provide an initial pattern expression modelling as much knowledge as he/she has on the application. Finally, an exploration of different alternatives derived by the use of different refactoring and optimisation rules may be used to optimise this pattern expression. On the other hand, the established knowledge base relating to optimisation of non-functional features affecting parallel execution of applications (e.g. cache friendliness, thread affinity, generic locality enhancing techniques, etc.) may be much better employed than in traditional, non structured parallel programming environments, due to the fact that both local (single pattern) and global (pattern composition) parallelism structure is known. Finally, JIT-like techniques may be put in place to optimise implementation of the patterns used in a given application only when the features of the target architecture are known. This may involve both picking up different known implementation templates for single patterns or pattern compositions as well as the refactoring of the whole parallel application structure by applying known rules to implement "better"—w.r.t. current target hardware features—parallel patterned code.

*Hardware centric* Using available hardware acceleration has always been subject to different phases. Initial availability of hardware accelerators requires specific, low level programming techniques to benefit from the accelerator's features. Over time, by analysing the existing code base targeting the particular accelerators, we can perceive that most of the applications use these low level tools to model particular parallel patterns, and we then can provide higher level, specific programming tools. Eventually, ways to identify and express the specific accelerator patterns in standard (sequential) code are developed and this brings accelerators to mainstream usage. This happened most notably with GP-GPUs. Initially very low level CUDA/OpenCL code was needed to implement data parallel computations on GP-GPUs. Later, specific programming environments evolved to relieve the application programmer from the effort of dealing with the low level details *and*

support for more common and useful situations was transferred to hardware (e.g. memory management or remote (host) memory uniform access). Finally, the community is trying to devise efficient techniques and tools that allow the programmer to annotate sequential code computations as data parallel and leave the underlying tool-chain to take care of the details needed for efficient implementation on accelerators (e.g. OpenACC[13]). Structured parallel programming adds additional possibilities to this process. By requiring the application programmer to identify the opportunities for parallelism expression through patterns, the data parallel computations identified may be easily implemented on GP-GPUs using an implementation approach very similar to that adopted in OpenACC [52] *and* to that adopted to target GP-GPUs in more classical and older skeleton based frameworks (e.g. SkePU). However, the identification of data parallel computations and the possibility to implement them on both CPUs and GP-GPUs (and on different kinds of accelerators) opens perspectives for cross accelerator (that is CPU+GP-GPU) implementation and optimisation of the same data parallel computations [33, 41, 48]. As an example, the time usually spent waiting for GP-GPU data parallel task computation may be used on the (multi-core) CPU to execute part of the data parallel task such that the overall execution time of the data parallel computation is decreased/optimised. While this may result in minimal improvements in cases where GP-GPUs clearly outperform CPUs, good improvements may be achieved in cases where the GP-GPU offloading overhead is not negligible.

## 6 Conclusions

Structured parallel programming models have been investigated in two distinct research communities for quite a long time: algorithmic skeletons in the HPC and parallel design patterns in the Software Engineering academic research community. Both research areas have contributed a significant number of results and achievements that are currently being integrated and exploited in industrial strength parallel programming frameworks. In this work we outlined the main qualitative achievements in this area as well as the way in which they have been adopted in different state-of-the-art parallel programming environments. We claim that the adoption of the full range of results coming from structured parallel programming communities may be considered a viable roadmap to reduce the software gap [3], may lead to availability of new and more advanced tools supporting parallelism exploitation, and may help achieve better hardware targeting.

**Acknowledgements** This work has been partially supported by Univ. of Pisa project "DECLWARE: Metodologie dichiarative per la progettazione e il deployment di applicazioni" (PRA_2018_66) and EU COST Action IC1406 High Performance Modelling and Simulation for Big Data Applications (cHiPSet).



[13] https://www.openacc.org/.

**Funding** Open access funding provided by Università di Pisa within the CRUI-CARE Agreement..

## References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. Int. J. Parallel Program. **42**(6), 1012–1031 (2014)
2. Amaral, V., et al.: Programming languages for data-Intensive HPC applications: a systematic mapping study. Parallel Comput. **91**, 102584 (2020)
3. Asanovic, K., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
4. Benkner, S., et al.: PEPPHER: Efficient and productive usage of hybrid computing systems. IEEE Micro **31**(5), 28–41 (2011)
5. Bird, R.S.: Lectures on constructive functional programming. In: Broy, M. (ed.) Constructive Methods in Computing Science, volume 55 of NATO ASI Series. F: Computer and Systems Sciences. Springer, Berlin (1989)
6. Botorog, G.H., Kuchen, H.: Efficient high-level parallel programming. Theor. Comput. Sci. **196**(1), 71–107 (1998)
7. Bromling, S., MacDonald, S., Anvik, J., Schaeffer, J., Szafron, D., Tan, K.: Pattern-Based Parallel Programming. In: ICPP '02. IEEE Computer Society, Washington, pp. 257– (2002)
8. Campa, S., Danelutto, M., Goli, M., González-Vélez, H., Popescu, A.M., Torquati, M.: Parallel patterns for heterogeneous CPU/GPU architectures: structured parallelism from cluster to cloud. Future Gener. Comput. Syst. **37**, 354–366 (2014)
9. Campbell, C., Johnson, R., Miller, A., Toub, S.: Parallel Programming with Microsoft.NET: Design Patterns for Decomposition and Coordination on Multicore Architectures, 1st edn. Microsoft Press, Redmond (2010)
10. Campbell, C., Miller, A.: A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures, 1st edn. Microsoft Press, Redmond (2011)
11. Chambers, C., et al.: FlumeJava: easy, efficient data-parallel pipelines. SIGPLAN Not. **45**(6), 363–375 (2010)
12. Chis, A.E., González-Vélez, H.: Design patterns and algorithmic skeletons: a brief concordance. In: Kołodziej, J., Pop, F., Dobre, C. (eds.) Modeling and Simulation in HPC and Cloud Systems, Number 36 in Studies in Big Data, pp. 45–56. Springer, Cham (2018)
13. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge (1991)
14. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
15. Danelutto, M.: QoS in parallel programming through application managers. In: PDP 2005, pp. 282–289. IEEE Computer Society, Lugano (2005)
16. Danelutto, M., Di Meglio, R., Orlando, S., Pelagatti, S., Vanneschi, M.: A methodology for the development and the support of massively parallel programs. Future Gener. Comput. Syst. **8**(1–3), 205–220 (1992)
17. Danelutto, M., Teti, P.: Lithium: A structured parallel programming environment in Java. In: ICCS '02, pp. 844–853. Springer, London (2002)
18. Danelutto, M.: Structured parallel programming with "core" FastFlow. In: Zsók, V., Horváth, Z., Csató, L. (eds.) Central European Functional Programming School, volume 8606 of LNCS, pp. 29–75. Springer, Berlin (2015)

19. Darlington, J., Guo, Y., To, H.W., Yang, J.: Parallel skeletons for structured composition. In: PPoPP'95, pp. 19–28. ACM, Santa Barbara (1995)
20. De Sensi, D., De Matteis, T., Torquati, M., Mencagli, G., Danelutto, M.: Bringing parallel patterns out of the corner: the P$^3$ARSEC benchmark suite. ACM Trans. Archit. Code Optim. **14**(4), 33:1–33:26 (2017)
21. Dean, J., Ghemawat, S.: MapReduce: A flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
22. 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**(24), e4175 (2017)
23. Diaz, J., Munoz-Caro, C., Nino, 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)
24. Emoto, K., Matsuzaki, K.: An automatic fusion mechanism for variable-length list skeletons in SkeTo. Int. J. Parallel Program. **42**(4), 546–563 (2014)
25. Enmyren, J., Kessler, C.W.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: HLPP '10, pp. 5–14. ACM, Baltimore (2010)
26. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw **7**(2), 129–138 (2012)
27. Ernstsson, A., Li, L., Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Int. J. Parallel Program. **46**(1), 62–80 (2018)
28. Faraj, A., Yuan, X.: Automatic generation and tuning of MPI collective communication routines. In: ICS '05, pp. 393–402. ACM, Cambridge (2005)
29. FastFlow home page. http://calvados.di.unipi.it/fastflow (2018). Accessed Apr 2020
30. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc, Boston (1995)
31. Gazzarri, L., Danelutto, M.: A tool to support FastFlow program design. In: ParCo 2017, pp. 687–697. Bologna (2017)
32. Gazzarri, L., Danelutto, M.: Supporting structured parallel program design, development and tuning in FastFlow. J. Supercomput. **75**(8), 4026–4041 (2019)
33. Goli, M., González-Vélez, H.: Autonomic coordination of skeleton-based applications over CPU/GPU multi-core architectures. Int. J. Parallel Programm. **45**(2), 203–224 (2017)
34. Goli, M., González-Vélez, H.: Formalised composition and interaction for heterogeneous structured parallelism. Int. J. Parallel Program. **46**(1), 120–151 (2018)
35. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exp. **40**(12), 1135–1160 (2010)
36. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Euro-Par, Vol. II, volume 1124 of Lecture Notes in Computer Science, pp. 401–408. Springer (1996)
37. Hammond, K et al.: The paraphrase project: parallel patterns for adaptive heterogeneous multicore systems. In: FMCO 2011 (Revised Selected Papers), volume 7542 of Lecture Notes in Computer Science, pp. 218–236. Springer, Turin (2011)
38. Javed N, Loulergue, F.: A formal programming model of Orléans skeleton library. In: PaCT'11, pp. 40–52. Springer, Kazan (2011)
39. Jouppi, N.P., Young, C., Patil, N., Patterson, D.: A domain-specific architecture for deep neural networks. Commun. ACM **61**(9), 50–59 (2018)
40. Korinth, J., de la Chevallerie, D., Koch, A.: An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: FCCM 2015, pp. 195–198. IEEE Computer Society, Vancouver (2015)
41. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: PACT '13, pp. 245–256. IEEE Press, Edinburgh (2013)
42. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Boston (2004)
43. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
44. Paraformance home page. https://www.paraformance.com/ (2018). Accessed Jan 2019
45. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis, Inc., Bristol (1998)
46. Prabhakar, R et al.: Generating configurable hardware from parallel patterns. In: ASPLOS '16, pp. 651–665. ACM, Atlanta (2016)

47. Reinders, J.: Intel Threading Building Blocks, 1st edn. O'Reilly & Associates, Inc., Sebastopol (2007)
48. Serban, T., Danelutto, M., Kilpatrick, P.: Autonomic scheduling of tasks from data parallel patterns to CPU/GPU core mixes. In: HPCS 2013, pp. 72–79. Helsinki (2013)
49. Sérot, J., Ginhac, D.: Skeletons for parallel image processing: an overview of the SKIPPER project. Parallel Comput. **28**(12), 1685–1708 (2002)
50. Spark Streaming home page. https://spark.apache.org/streaming/ (2019). Accessed Jan 2020
51. Streamit home page. http://groups.csail.mit.edu/cag/streamit/ (2016). Accessed Jan 2019
52. Wienke, S., Springer, P.L.: Christian terboven, and dieter an Mey. OpenACC—first experiences with real-world applications. In: Euro-Par 2012, volume 7484 of LNCS, pp. 859–870. Rhodes Island. Springer (2012)



## Affiliations

**Marco Danelutto[1] · Gabriele Mencagli[1] · Massimo Torquati[1] · Horacio González–Vélez[2] · Peter Kilpatrick[3]**

✉ Marco Danelutto
marco.danelutto@unipi.it

Gabriele Mencagli
mencagli@di.unipi.it

Massimo Torquati
torquati@di.unipi.it

Horacio González–Vélez
horacio@ncirl.ie

Peter Kilpatrick
p.kilpatrick@qub.ac.uk

[1] Department of Computer Science, University of Pisa, Pisa, Italy

[2] Cloud Competency Centre, National College of Ireland, Dublin, Ireland

[3] Department of Computer Science, Queen's University of Belfast, Belfast, UK