Autonomic management experiences in structured parallel programming

Marco Danelutto
Dept, Computer Science Univ. of Pisa)
marco.danelutto@unipi.it

Daniele De Sensi
Dept, Computer Science Univ. of Pisa)
desensi@di.unipi.it

Gabriele Mencagli
Dept, Computer Science Univ. of Pisa)
mencagli@di.unipi.it

Massimo Torquati
Dept, Computer Science Univ. of Pisa)
torquati@di.unipi.it

***Abstract*—Structured parallel programming models based on parallel design patterns are gaining more and more importance. Several state-of-the-art industrial frameworks build on the parallel design pattern concept, including Intel TBB and Microsoft PPL. In these frameworks, the explicit exposition of parallel structure of the application favours the identification of the inefficiencies, the exploitation of techniques increasing the efficiency of the implementation and ensures that most of the more critical aspects related to an efficient exploitation of the available parallelism are moved from application programmers to framework designers. The very same exposition of the graph representing the parallel activities enables framework designers to emplace efficient autonomic management of non functional concerns, such as performance tuning or power management. In this paper, we discuss how autonomic management features evolved in different structured parallel programming frameworks based on the algorithmic skeletons and parallel design patterns. We show that different levels of autonomic management are possible, ranging from simple provisioning of mechanisms suitable to support programmers in the implementation of *ad hoc* autonomic managers to the complete autonomic managers whose behaviour may be programmed using high level rules by the application programmers.**

***Index Terms*—Algorithmic skeletons, parallel design patterns, autonomic managers, rule based control, structured parallel programming.**

I. Introduction

Structured parallel programming is a methodology based on the adoption of composable, parametric parallel building blocks that capture and model simple parallelism exploitation patterns. At the very beginning, these building blocks were named *algorithmic skeletons* (after Cole's work [8]), and where provided to HPC application programmers as abstractions of sequential programming languages (libraries in imperative languages, objects in OO languages, higher order functions in functional languages, etc…). Later on, the concept of design pattern, developed within the software engineering community, has been moved in the parallel programming context with the *parallel design patterns* concept, promoting the very same idea of algorithmic skeletons, i.e. that parallelism exploitation in parallel application can be expressed though proper usage of parallel patterns, alone or in composition, completed by supplying proper "business logic code" parameters capturing the actual business logic of the application [18].

In both cases, the parallel architecture of the application is completely exposed by the skeleton/pattern composition chosen by the application programmer. This represents the key benefit of structured parallel programming models with respect to classical, unstructured ones. On the one hand, the availability of different, and in many cases alternative, skeleton/pattern compositions suitable to model the application increases the productivity of parallel application programmers and decreases the effort required to design the parallel application. On the other hand, the implementation toolchain (compilers, libraries, interpreters) may leverage the knowledge about the parallel structure of the application to exploit different well-known techniques aimed at providing efficient parallelism exploitation as well as efficient architecture targeting. Overall, structured parallel programming leaves the application programmers only in charge of figuring out the proper *qualitative* parallelism exploitation aspects and move to the system/framework programmers all the duties related to *quantitative* aspects and, even more important, of all those cumbersome and error-prone aspects more related to correct and efficient usage of base mechanisms (concurrent activities setup, communication, synchronization).

Structured parallel programming has been for a long time subject of activities of different research groups all around the world [7], [14], [15], [17], [21]. In [4], parallel design patterns have been mentioned as a realistic opportunity to overcome difficulties in parallel programming. After a while, this concept permeated to the industry, with several distinct "industrial" frameworks adopting some of the structured parallel programming concepts and methodologies.

For example, Intel Thread Building Block library [19] provides parallel patterns modelling different aspects of parallel applications (high level patterns and lower level communication and synchronization mechanisms) through patterns encapsulated in modern C++ classes. Similarly, Microsoft Parallel Pattern Library [20] provides different kinds of parallel patterns to programmers of standard Microsoft applications. All in all, even OpenMP [9] provides a kind of single data parallel pattern (the *parallel for*) that can be used to model quite a number of data parallel kernels as it may be used, by specifying proper additional parameters, as a *map*, a *reduce*, a *map-reduce* and a *stencil* data parallel pattern. Last but not least, the C++ standard committee is pushing forward the

idea of parallel execution of STD library "algorithms". In the forthcoming versions of the language, an additional parameter will be accepted by STD algorithms (sort, transform, etc.) making the computation of the algorithm parallel (multi-threaded). Although C++ STD algorithms are not exactly parallel patterns, the underlying idea is the very same of structured parallel programming community: provide the programmer with ready to use parallel solutions that can be used as building blocks speeding up execution of standard applications, with the parallelization effort moved from application programmers to library/language implementation programmers.

A. Autonomic management of non functional concerns

Independently of the programming framework used, when developing parallel applications the main goal of the programmer is twofold: on the one hand the parallel application must be functionally correct, while on the other hand it must also be efficient, whatever measure (or mix of measures) we want to take into account (time, energy consumption, security, etc.). This second aspect is related to the efficient achievement of *non functional* features, that is features that do not concur to ensure *what* we compute (i.e. the application results), but rather concur to determine *how* we actually compute the application results. Typically, average performance (either considering latency or throughput), power management, load balancing as well as fairness may be considered *non functional* concerns in parallel application execution.

A more concrete instantiation of non functional concern may better explain the concept. Performance of a parallel application is definitely impacted by the parallelism degree used in the execution of taht application. Usually, a trade-off has to be found among higher parallelism degrees (hopefully leading to decreased completion/service times), and lower parallelism degrees (ensuring smaller parallelism management overheads). Any programmer writing parallel applications has experienced coming to the "knee" point where speedup stops increasing with parallelism degree due to overheads becoming important and/or computation grain of parallel activities becoming too light.

Another typical example of non functional concern is power management. Dynamic Voltage and Frequency Scaling (DVFS) [16] techniques may be used to improve execution times on standard multi-cores. However, the higher the frequency the higher the power consumption. Therefore a trade-off has to be ensured among performance and power consumption, especially in those cases were power consumption is a synonym of battery life.

Structured parallel programming helps programmers to emplace decent solutions for autonomic management of these non functional concerns. Being parallel application structure expressed as composition of patterns, the management of non functional concerns at application level may be reduced to management of non functional concerns at the single pattern level combined with rules that define non functional behaviour of composition of patterns.

Let's look at the autonomic management of the parallelism degree in a *parallel for/map* pattern. The programmer may indicate a parallelism degree of the map pattern that leads to inefficient parallelism exploitation in two ways:

- by expressing a too small parallelism degree, which will result in under utilization of the available resources, or
- by expressing a too large parallelism degree, which will result in a too high overhead.

The *parallel for*/map implementation can be easily equipped with additional code that monitors the initial part of the computation and possibly computes a better parallelism degree for the pattern. The reconfiguration of parallelism degree will take a predictable amount of time, in general, thus leaving the *parallel for*/map implementation the ability to decide whether or not to reconfigure itself depending on the estimated execution time. In case the *parallel for*/map is a component of another pattern (e.g. a pipeline processing a stream of data parallel tasks) the performance model of the topmost pattern may be used to figure out a) whether a reconfiguration is needed relative to the overall parallelism degree and b) where it has to be applied in case (e.g. replicating one or more data parallel pipeline stages or increasing the parallelism degree of the stages).

B. Contributions

Our contribution consists in a review of three different approaches used in structured parallel programming to include autonomic management of non functional features in structured parallel programming frameworks. In particular:

- we discuss different approaches providing different autonomic management possibilities
- we discuss solutions targeting different architectures
- we outline possibilities to include similar solutions in existing state-of-the-art parallel programming frameworks

We wish to point out that, despite the fact a number of domain specific or non structured parallel programming frameworks exist that include different kind of autonomic management features, we concentrate on the benefits coming from synergies deriving from joint adoption of *structured* parallel programming techniques *and* autonomic management techniques.

The rest of the paper includes three sections describing three different autonomic management experiences contributed by our research group (Sec. II to IV). Sec. VI outlines possibilities to include autonomic management in state-of-the-art programming frameworks. Finally Sec. VII draws conclusions and prospect future work.

II. Full single concern management: Behavioural skeletons in Grid Component Model

Behavioural skeletons [1], [2] have developed in early '00, when the "fashion buzzword" in parallel and distributed computing community was *grid computing*. Within the EU funded Network of Excellence project CoreGRID and the spin-off STREP project GridCOMP, the concept of algorithmic skeleton has been extended to include some kind of

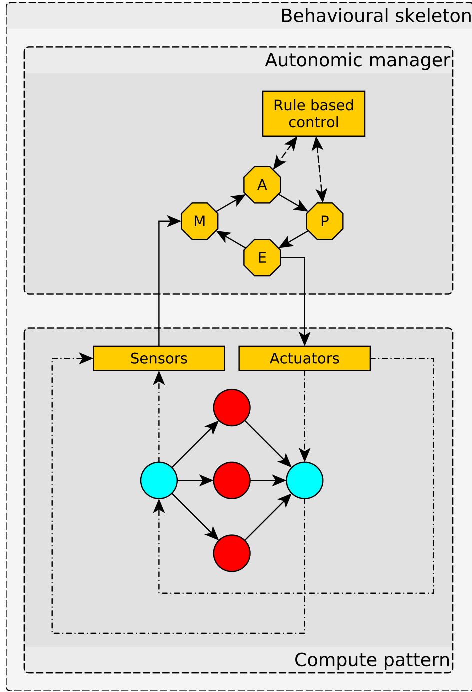

Fig. 1. Sample behavioural skeleton structure

autonomic control taking care of the performance of the parallel/distributed application after the programmer correctly expressed its parallel behaviour through proper algorithmic skeleton compositions (see Fig. 1).

Autonomic management of performance[1] was achieved by associating a further concurrent activity to each one of the algorithmic skeletons/parallel patterns used. This additional activity was running a *Monitor-Analyze-Plan-Execute* (MAPE) loop using proper sensors and actuators to monitor the running application and to implement the decisions taken after analysing the available data. The pairing of a computational pattern with its autonomic manager was called *Behavioural Skeleton*. Below we detail the two most important features introduced by Behavioural skeletons along with a brief summary of the results achieved with real applications on computational grids.

A. Sensor/actuator pattern instrumentation

The first important feature introduced by behavioural skeletons was the concept of structured sensors and actuators. In non-structured parallel programming frameworks the parallel structure of the application is not exposed to the tools. Therefore any sensor used to figure out the actual and current behaviour of an application must be directly programmed by the application programmer, the only one having a clear picture of the overall parallel schema of the application. In case parallelism is only expressed through (composition of) algorithmic skeletons/parallel patterns, proper sensoring instrumentation may be provided by the algorithmic skeleton/parallel pattern designer capturing the essential and worth properties of the skeleton/pattern execution. Similarly, actuators may be embedded in skeleton/patterns directly by their designer, providing meaningful actions for that particular parallelism exploitation structure.

[1]Completion time or service time of the application.

As an example, let us consider two common stream parallel patterns: pipelines and farms. The pipeline pattern applies a set of cascading functions $f_1, \ldots, f_k$ over all the items (x_i) appearing onto an input stream, delivering the results $(f_k(\ldots(f_1(x_i)\ldots))$ over its output stream. Parallelism is exploited in parallel computation of the different *stages* (activities computing the different f_j) over different items of the input stream. The *farm* pattern maps the same computation (say applying function f) over all the items x_i appearing onto an input stream, delivering results $(f(x_i))$ over its output stream, possibly maintaining the input/output ordering. Parallelism is exploited by using a set of identical "*workers*" computing f over different items of the input stream.

In these patterns, sensors may be easily programmed to report inter-arrival/departure times of input elements to/from the pipeline stages or to/from the farm workers, and pattern-specific actuators may be provided, such as:

- adding or removing a worker from the farm pattern implementation, thus increasing/decreasing its performance,
- merging consecutive stages of the pipeline that altogether do not represent bottlenecks for the overall computation (i.e. whose sum of service times is still smaller than the service time of at least one other stage in pipeline),
- splitting previously merged stages, in case their merged service time happens to be the largest in the pipeline,
- transforming a sequential pipeline stage into a farm, if its service time is the largest in the pipeline,
- removing a farm from a pipeline stage in case the service time of a single worker becomes smaller than the service times of other pipeline stages.

All this actions may be implemented only because the structure of the parallel computation is known and well-defined. The reader may easily figure out the effort required to implement the same kind of "actuators" when a parallel application is explicitly programmed using low level mechanism/libraries.

B. Rules-based non functional concern management

The availability of sensors and actuators specific to the used skeleton/patterns enables the implementation of efficient autonomic management policies for skeleton/pattern based applications. These policies are *de facto* the *control program* of our parallel application non-functional behaviour (performance, in case of behavioural skeletons).

In order to provide maximum flexibility, behavioural skeletons adopted a business rule system (JBoss at that time) to provide the user the possibility of programming the behavioural skeleton's application control program through condition-action rules where:

- conditions where expressed as formulas involving read through sensors, and

- actions where expresses using (proper compositions of) actuator calls.

Rules were prioritized and additional state variables were supported to make control program stateful, which was used–as an example–to avoid useless cycling among pairs of configurations equally far away (or close to) from the optimal steady state configuration.

In addition, the user could express target performance values (i.e. service level agreements) together with manager rules to drive the manager actions.

C. Results

The most important consequence of the rule based management in the MAPE loop was the simplicity observed in the implementation of quite complex autonomic parallelism management policies.

At the end of the GridCOMP project, a long running application processing images from medical equipment was implemented using behavioural skeletons and was demonstrated to be able to automatically adapt the parallelism degree of its different components, correctly and timely enforcing the user supplied frame rate (service level agreement) [2]. In particular, in a pipeline where the computing stages were parallelized using farms, the detection of an under utilized farm stage first led to the enforcement–by the top level pipeline rules–of an increase of performance in the stage supplying the input stream items, and then dynamically adjusted the parallelism degrees of the other stages such that a) the user supplied frame rate service level agreement was met and b) no load imbalances were observed among parallel pipeline stages, thus increasing the efficiency of the application.

Overall, the Behavioural skeleton experience showed that autonomic management of non functional features through properly programmed MAPE loops perfectly pairs with the structured parallel programming principles. The exposition of the parallel structure of the program enables programmers to include in the MAPE loop notable rules and efficient control policies. By expressing application parallelism through skeletons/parallel patterns, notable and efficient sensors and actuators can be exploited, while not increasing the programming effort required to the application programmer compared to the case where autonomic management was not considered at all.

III. Providing mechanisms for *ad hoc* autonomic management: FastFlow

FastFlow is a structured parallel programming framework designed, developed and maintained by members of Univ. of Pisa and of Torino since early '00s mainly targeting shared memory architectures, with limited support for accelerators[2]. FastFlow provides the programmer with three distinct types of parallel design patterns: i) "core" patterns, ii) high level patterns and iii) parallel building blocks. All those patterns are provided via fully C++14 compliant template classes, so that programmers of a parallel application may build a pattern expression completely modelling the parallel behaviour of the application, and then invoke pattern expression computation separately. The differences in between the three distinct class of patterns can be stated as follows:

- core pattern model simple parallel patterns (e.g. pipeline and farms in the stream parallel pattern set or map and reduce in the data parallel set). They are usually used in composition to model more complex/realistic parallel execution scenarios.
- high level patterns model more complex patterns (e.g. divide and conquer or pool evolution patterns) and are more frequently used as the sole pattern modelling the complete parallelism within an application, even if they can be freely nested in/with other patterns, if needed.
- finally parallel building blocks present a lower level of abstraction with respect to the other two kind of patterns and are intended as "parallel bricks" to be used to implement other core or high level pattern to be provided to the application programmer.

Independently of the patterns used, the elementary abstraction provided by FastFlow is the *node* abstraction: a thread with an input task queue and an output task queue which executes an infinite loop. At each iteration the node looks for a task in the input queue, computes the task and subsequently delivers the result over the output queue. As an example, pipelines are built by chaining nodes and using a single queue item as output queue of node i and input queue of node $i+1$. Farms and maps may be implemented in different ways, including using a set of nodes all fetching tasks from a single queue and delivering tasks to another single queue or using two additional nodes, one actively scheduling input tasks from the input queue to the input queues of a set of worker nodes, and one gathering results from the worker nodes and delivering them into its own output queue.

A. Alternative communication implementation mechanism

FastFlow communications happen to leverage the shared memory model of the target architecture. Items passed through node input/output queues are pointers to data and the FastFlow queues are implemented in a very efficient lock-free way, such that sending or receiving a single item to/from a queue is an operations that takes from a few nanoseconds to about one hundred nanoseconds on state-of-the-art shared memory architectures[3]. The conceptual model behind this implementation is that when a node communicates a pointer to another node, it formally gives the second node the *capability* to operate on the pointed data, implicitly subscribing the fact it (the first node) will not access any more that data. Being the ultra fast FastFlow communications implemented with the usage of additional threads and active wait spin-locks[4], alternative

[2]Framework web page is at http://calvados.di.unipi.it/fastflow and the code may be accessed via Github at https://github.com/fastflow/fastflow

[3]depending on the relative location of the sending and the receiving node: same/different core/socket

[4]That only take place on worst case scenarios, however.

more classic communication mechanisms are provided based on classical passive, shared communication buffer data structures, that may be used when computational grain of parallel activities is sufficiently large.

The choice relative to which communication mechanisms has to be used is up to the programmer. FastFlow patterns provide by default the ultra-fast spin-lock based mechanism. Programmers may ask to use the other communication mechanisms for the whole program or for portions (pattern sub expressions) of the program both before running the pattern expression representing the parallel application and *during* execution. The requests are issued using the FastFlow communication type "actuators" that operate on single patterns or on composition of FastFlow patterns. This opens perspectives to the possibility of implementing autonomic management of the communications mechanisms that use the default mechanisms when the application starts and then switch to other mode in case the grain of parallel computations turns out to be large enough. This is possible as FastFlow frameworks provides interfaces to access "sensor" data measuring the input pressure of a node (via number of items in the input queue and/or number of failed pops from the queue) as well as (average) time spent in computing the single task in a node.

B. Concurrency throttling mechanisms

FastFlow also provides mechanisms to dynamically vary the parallelism degree of core patterns. As an example, FastFlow farms may be started providing two parameters: the maximum parallelism degree possible for the pattern (nw_{max}) and the actual parallelism degree required once the pattern expression to which the farm pattern belongs will be executed (nw). While executing, calls may be made to farm implementation "actuators" to increase or decrease by a given amount (1 or larger) its parallelism degree. This enables the usage of "dynamic" farms whose parallelism degree nw may vary in the interval $[1, nw_{max}]$. Actually, when the farm is shrinked down to $nw = 1$ it can be dynamically re-mapped to a single node with no scheduler and gatherer additional nodes.

This mechanism, enables, as the other one relative to communication implementation, the possibility for the programmers to implement *ad hoc* autonomic management policies dynamically tuning the parallelism degree of an application, and it may be incredibly useful in at least two different cases:

- in long running applications with notably different phases (e.g. a stream processing application computing tweet analytics that needs to properly take case of "hot spot" phases), or
- when applications run in non exclusive mode on the target architecture and additional, external loads may impair proper dimensioning of the parallelism degree of the application at hand.

C. Topology optimization mechanisms

Last but not least, the most recent release of FastFlow provides a further mechanism that may be useful in the perspective of providing programmers with mechanisms suitable to implement autonomic management of non functional features, namely some actuators optimizing the FastFlow node graph with respect to resource (threads) usage or to the overall number of resources required to run the pattern expression at hand.

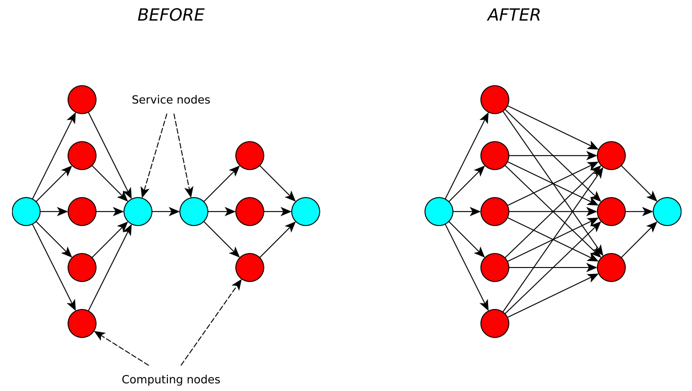

Fig. 2. Sample FastFlow node graph optimizations

FastFlow node graphs happen to be built out of the merge of the sub-graphs used to implement the outermost pattern expression. As an example, a pipeline expression with two consecutive farm stages is trivially synthesized with a node graph where the output gather node of the first farm is directly connected to the input scheduler node of the second farm. In case the gather policy of the first farm and the schedule policy of the second farm are standard (not re-defined by the programmer), the two nodes may be easily merged, sparing one thread. Under some more specific conditions on gathering/scheduling policies in the two farms, the two nodes (or the merged node) may be completely eliminated, connecting workers of the first farm to the workers of the second farm with a specific "all-to-all" parallel building block (see Fig. 2).

The FastFlow node graph optimization actuator provides different operations that may be required by the programmer, whose actual effects on FastFlow application performance vary and depend on business logic properties that cannot be automatically figured out by the framework. Therefore, the possibility offered to invoke different kind of node graph optimizations opens perspectives to implement autonomic management of the node graph as the application programmers. However, the node graph optimization has to be invoked before the pattern expression is computed. This notwithstanding application programmers may set up different versions of the application (optimized versions A, B, ... and non optimized version) or of parts of the application and then use the ones identified as the most suitable through parameters figured out from FastFlow sensors.

IV. Multi-concern management: Nornir

Nornir[5] is a customizable C++ framework supporting the development of autonomic and power-aware algorithms for

[5] http://danieledesensi.github.io/nornir/

parallel applications running on shared memory multicore machines [11]. Among the other features, Nornir provides ways to manage structured and unstructured parallel applications and, particularly, it supports applications written in FastFlow. This means that if you have a FastFlow application, you may simply add a few lines of code creating a `Manager` and passing the manager a user contract describing the expectations in terms of non functional concerns of the application user along with the handle of the FastFlow application (that is the pattern expression defined the parallel structure of the application). Nornir has been designed to manage long running applications, that is applications where time needed to reconfigure the parallel application (e.g. in terms of adding or removing computing resources or varying DVFS parameters) may be considered negligible with respect to the time used to run the application. Stream parallel applications computing ideally infinite stream of tasks are also targeted by Nornir. The manager then takes care of all the activities needed to ensure (best effort) the user provided contract. In order to ensure the contract, Nornir autonomically manages different aspects related to the parallel execution of the application, including voltage/frequency parameters, thread pinning and concurrency throttling (dynamic adaptation of application parallelism degree).

A. Multiple non functional concern management

One of the more notable contributions of Nornir consists in the fact that its autonomic manager ensures (best effort) a trade-off in between performance and power consumption, that is the autonomic management is actually pursuing multiple concerns. Moreover, the multiple concerns taken into account in Nornir normally push manager decisions in opposite directions: looking for performance favours increasing resource usage and higher frequencies, while looking for power consumption favours exactly opposite mechanisms, such as decrease resource usage and operating frequencies of the available cores. This makes the autonomic management of both non functional concerns (power and performance) completely different from the management of a single concern, such as performance in Behavioural skeletons as outlined in Sec. II.

Nornir relies on a lower level library (Mammut[6]) providing different mechanisms that include sensors and actuators. In particular, the sensors in Mammut report power consumed in applications, as well as different machine parameters that affect the parallel behaviour of applications, such as the number of available cores and the core/cache topology. The actuators provide, among the others, mechanisms to pin threads to cores, to move threads across contexts, to enforce particular DVFS parameters, etc.

B. Predictive modelling

Autonomic management policies implemented in Nornir consider moves in a space of "configurations", i.e. tuples of values for the parameters of interest (parallelism degree, thread pinning, frequency values, etc.). In the initial part of the computation, a small number of configurations are used and the relevant non functional properties (time and power spent in the computation of a given amount of tasks) are monitored using Mammut sensors. Then the measured values are used to build a simple interpolation model which is subsequently used to figure out which are the configurations suitable to match the contract provided to the manger by the programmer/user. Among those configurations, the manager picks up the more convenient one and executes the rest of the application with that configuration. The manager keeps collecting the measures of interest through Mammut sensors throughout the entire application execution, to ensure that the optimal configuration is selected even in presence of performance and power consumption fluctuations, either caused by external factors, or due to intrinsic differences between application phases.

This can be considered a learning based approach to parallel application modelling, of course. However, differently from other approaches in literature, the learning process does not require any previous knowledge relative to application behaviour. Rather, the knowledge is build while the application is running. Experiments demonstrated that the initial run of the application with random configurations has a negligible impact on application performances (a few tasks are executed in each of the configurations experimented), while the interpolation model derived demonstrates to be precise enough to identify a worth configuration to continue the execution of the parallel application matching the user requirements stated in the manager contract [12].

Is worth pointing out another couple of points relatively to Nornir. Nornir can be customized by implementing new prediction policies, by relying on the monitor and actuation mechanisms already provided by the framework in an abstract way. However, the Nornir manager policies are coded in the framework. That is, in case different management policies need to be implemented, the new code must be implemented in the framework and it needs to be re-compiled. This represents quite a difference with respect to Behavioural skeletons discussed in Sec. II. In addition, the Nornir frameworks may be used to manage also unstructured applications. The manager follows the same steps outlined for the management of the structured, FastFlow applications mentioned above. However, sensors may be used only to figure out general, application wide measures, as the patterns used to exploit parallelism within the application are not known. Furthermore, only general purpose, application wide actuators may be used for the very same reason (i.e. it would not be possible to dynamically change the number of threads used). This means that overall the efficiency of the Nornir autonomic manager may be sensibly decreased with respect to efficiency achieved while taking care of structured parallel applications.

V. Comparison and discussion

The three experiences discussed in the previous sections may be overall summarized and compared as follows:

[6]http://danieledesensi.github.io/mammut/

	GCM	FastFlow	Nornir
Control	Autonomic	User centric	Autonomic
User SLA	Yes	No (specific code needed)	Yes
NF Concerns	Single: Performance (Service time)	Single: Performance (service time or latency)	Multiple: Power + Performance
Strategies	Optimize service time through dynamic parallelism degree adaption	Perform known refactorings of concurrent activity graphs, change implementaiton mechanisms, all upon user/programmer specific request	Heuristics to identify performance/power consumption tradeoff, exploiting from monitoring info
Systems targeted	COW/NOW/GRID	Shared memory multi/many core (with accelerators)	Shared memory multi/many core (with accelerators)
Project	CoreGRID and GRIDcomp EU funded FP6 projects (late '00s)	UNIPI/UNITO project, adopted in ParaPhrase, REPARA and RePhrase EU funded projects (FP7 and H2020) & PhD thesis, early '10s → late '10s	UNIPI project & PhD thesis, late '10s

Fig. 3. Summary of features of the three examples discussed

a) GCM Behavioural skeletons: BS represent some how the initial step in autonomic management of non functional features in structured parallel application. They build on previous experiences where autonomic management was already used in more limited ways, such as the Muskel Java based framework [10] using autonomic managers to get rid of failures in COW/NOW execution of structured parallel applications, or ASSIST [6] that also used a kind of autonomic managers to adapt dynamically parallelism degree.

b) FastFlow: FastFlow mechanisms provide elementary mechanisms that programmers of parallel applications may use to implement *ad-hoc* autonomic managers, possibly embedded in the system code providing high level, autonomic parallel design patterns to the application programmers.

c) Nornir: Nornir introduces synergic management multiple concerns building on previous experiences from the group that considered the possibility to implement strategies to deal with multiple concerns, possibly pushing in different directions the non functional parameters considered when optimizing parallel execution behaviour [3].

Overall, the three experiences reported in this paper contributed to show that more and more decisions affecting the efficiency of a parallel computation may be taken when the parallel structure of the application is completely exposed to the parallel framework implementation tools. Behavioural skeletons demonstrated the feasibility of the concept in a distributed environment. Despite the fact they where only developed to take care of service time of a structured parallel application, the research on BS opened perspectives on suitable ways to manage multiple non functional concerns that later on was actually implemented in Nornir. All the mechanisms provided in FastFlow to support user directed autonomic management are inspired by previous work on autonomic managers in structured parallel computations, including Behavioural skeletons.

VI. AUTONOMIC MANAGERS IN PERSPECTIVE

The three examples discussed in the previous Sections show different aspects relative to autonomic management of non functional features in parallel/distributed computations (see Fig. 3): single or multiple non functional concern management, complete autonomic managers vs. set of mechanisms supporting *ad hoc* manager design, different techniques to implement manager control programs. The common factor of all the mentioned frameworks and tools is that they work and are that efficient because they are managing applications whose parallel structure is known, such as those programmed using skeleton/pattern based structured parallel programming frameworks. We hope this important fact may be recognized and exploited also in more traditional, state-of-the-art parallel programming frameworks.

Following an approach similar to the ones adopted in Nornir or in Behavioural skeletons, it would be relatively simple to add some kind of autonomic management of performance (time) and power consumption in application written using Microsoft Parallel Pattern Library or Intel TBB, as those frameworks already support programming of parallel applications through usage of properly nested patterns.

Similarly, some of the techniques discussed in the previous Sections may be applied to those patterns included in parallel programming frameworks that *per se* do not support structured parallel programming. OpenMP is one of such frameworks. The only pattern(s) supported is the *parallel for*, which in turn can be used to model map, reduce and map-reduce patterns. Some of the different `schedule` parameters that can be indicated in a `#pragma omp for` clause in fact already try to do something to optimize task (i.e. iteration) scheduling on available thread from the OpenMP thread pool (e.g. the `dynamic` clause). In a sense, picking up one of the available schedule clause parameters corresponds to the request of a specific contract in terms of *parallel for* performance by the application programmer. A similar approach could be

implemented in other frameworks providing similar parallel loop constructs, such as Microsoft TPL or Intel TBB.

Moreover, in some cases more complex policies could be programmed, for example by using the OMPT tracing library [13] to intercept and monitor the main OpenMP routines. For example, it could be possible to monitor each *parallel for* iteration by collecting information through proper sensors (reporting average time spent computing iterations, load of the different computing resources, etc) and by applying decisions with appropriate actuators (adding/removing threads to the *parallel for* execution, or varying other thread features, such as thread pinning, core frequency and others). In addition to that, in task-based programming environments, tasks are usually executed by a pool of threads, which could be dynamically added or removed from the pool by using mechanisms and techniques similar to those provided by FastFlow and Nornir. The thread number to be used to execute a specific parallel section could be selected by using a `task_scheduler_init` in Intel TBB or by calling a `ThreadPool.SetMaxThreads` in Microsoft TPL.

Finally, a concept somehow related to autonomic computing is that of *"auto tuning"*, which can be applied by frameworks and library to optimize execution by selecting a close to optimal set of running parameters once the target architecture or to the software configuration of the target machine are known. As a notable example is that of linear algebra libraries that implement auto tuning taking into account hardware features such as cache sizes or ALU/SIMD sizes of target architectures [5].

VII. Conclusions

We outlined different experiences contributed by our research group that introduced different autonomic management features in structured parallel programming frameworks based on the algorithmic skeleton and parallel design pattern concepts. We argued that the effectiveness of the different autonomic management features is fundamentally due to the fact the exact parallel structure of the applications is completely exposed to the tool-chain supporting application execution. We finally briefly discussed how the techniques experimented in the structured parallel programming frameworks scenario may be actually and, probably, seamlessly migrated to mainstream, state-of-the-art parallel programming framework.

ACKNOWLEDGMENTS This work has been partially supported by Univ. of Pisa PRA_2018_66 DECLware: Declarative methodologies for designing and deploying applications

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons for component autonomic management on grids. In *Making Grids Work: Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments, 12-13 June 2007, Heraklion, Crete, Greece*, pages 3–15, 2007.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons in GCM: autonomic management of grid components. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 13-15 February 2008, Toulouse, France*, pages 54–63. IEEE Computer Society, 2008.

[3] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *23rd IEEE Int'l Symp. on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy*, pages 1–12, 2009.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[5] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, Nov 2018.

[6] Carlo Bertolli, Daniele Buono, Gabriele Mencagli, and Marco Vanneschi. Expressing adaptivity and context awareness in the ASSISTANT programming model. In *Autonomic Computing and Communications Systems, Third Int'l ICST Conf., Autonomics 2009, Limassol, Cyprus, September 9-11, 2009*, pages 32–47, 2009.

[7] Philipp Ciechanowicz and Herbert Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*, pages 108–113, 2010.

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

[9] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[10] Marco Danelutto. Qos in parallel programming through application managers. In *13th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2005), 6-11 February 2005, Lugano, Switzerland*, pages 282–289, 2005.

[11] Daniele De Sensi, Tiziano De Matteis, and Marco Danelutto. Simplifying self-adaptive and power-aware computing with nornir. *Future Generation Computer Systems*, pages –, 2018.

[12] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM Trans. on Architecture and Code Optimization*, 13(4):43:1–43:25, dec 2016.

[13] Alexandre E. Eichenberger, John M. Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT: an openmp tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators - 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, pages 171–185, 2013.

[14] August Ernstsson, Lu Li, and Christoph Kessler. Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, Feb 2018.

[15] Noman Javed and Frédéric Loulergue. A formal programming model of orléans skeleton library. In *Proceedings of the 11th International Conference on Parallel Computing Technologies*, PaCT'11, pages 40–52, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[17] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Parallel programming with a pattern language *. *International Journal on Software Tools for Technology Transfer*, 3(2):217–234, May 2001.

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

[19] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[20] Parallel Pattern Library, 2019. https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=vs-2019.

[21] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.