Self-Configuration and Self-Optimization Autonomic Skeletons using Events

Gustavo Pabon, Ludovic Henrio

Self-Configuration and Self-Optimization Autonomic Skeletons using Events *

Gustavo Pabón
NIC Research Labs, Chile
gustavo.pabon@gmail.com

Ludovic Henrio
INRIA-I3S-CNRS,University of Nice Sophia
Antipolis, France
ludovic.henrio@cnrs.fr

ABSTRACT

This paper presents a novel way to introduce self-configuration and self-optimization autonomic characteristics to algorithmic skeletons using event driven programming techniques. Based on an algorithmic skeleton language, we show that the use of events greatly improves the estimation of the remaining computation time for skeleton execution. Events allow us to precisely monitor the status of the execution of algorithmic skeletons. Using such events, we provide a framework for the execution of skeletons with a very high level of adaptability. We focus mainly on guaranteeing a given execution time for a skeleton, by optimizing autonomically the number of threads allocated. The proposed solution is independent from the platform chosen for executing the skeleton for example we illustrate our approach in a multicore setting, but it could also be adapted to a distributed execution environment.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Design, Languages

Keywords

Autonomic computing, Algorithmic skeletons, Event driven programming, Self-configuration, Self-optimization.

1. INTRODUCTION

Large-scale parallel and distributed environments allow the resolution of large-scale problems. However, parallel software development is hard, and curently, we are facing an increasing challenge due to the increasing number of cores or machines available for a single computation. Indeed both million-core supercomputers and cloud infrastructures are almost impossible to program efficiently, and those architectures are even more difficult to maintain. According to Gartner [21], IT operations management costs are the 36% of the total operation IT budget. As an alternative solution, IBM, in 2001, introduced the concept of Autonomic Computing (AC) [16]. It refers to the self-managing (self-configuration, self-optimization, self-healing, and self-protecting) characteristics of computing resources. In autonomic computing, each entity is able to adapt itself to changes in the runtime environment, or in the quality of service desired. The vision of autonomic computing is based on the idea of self-governing systems. These systems can manage themselves given high-level objectives from an administrator [17].

Algorithmic Skeletons [14] (skeletons for short) are a high-level parallel programming model introduced by Cole [8]. Skeletons take advantage of recurrent parallel programming patterns to hide the complexity of parallel and distributed applications. Lately, the use of skeletons has raised due to the increasing popularity of MapReduce pattern [7] for data-intensive processing. The introduction of autonomic characteristics to skeletons is an active research topic in the parallel and distributed computing area [2, 9, 15].

Skeletons use inversion of control to provide a high-level programming model, and hide the complexity of parallel and distributed programming. But inversion of control hides, at the same time, the actual execution flow to the programmer; which is essential to the implementation of non-functional concerns like autonomic computing. In our previous work [20], we proposed a solution to tackle skeleton's inversion of control by introducing a novel separation of concerns (SoC) based on event driven programming. This paper shows how to use such SoC for developing self-configuring and self-optimizing skeletons. Events allow us to precisely monitor the status of the skeleton execution with a very high level of adaptability. In this paper, we focus mainly on guaranteeing a given execution time for a skeleton, by optimizing autonomically the number of threads allocated, but the approach is clearly applicable to other autonomic characteristics.

Our proposal was implemented above Skandium framework [18]. Skandium is a Java based Algorithmic Skeleton library for high-level parallel programming of multi-core architectures. Skandium provides basic nestable parallelism patterns, which can be composed to program more complex

*This work is part of the SCADA (Safe Composition of Autonomic Distributed Applications) Associate Team OASIS/NIC-Labs (*https://team.inria.fr/scada/*)

PMAM 2014 The 2014 International Workshop on Programming Models and Applications for Multicores and Manycores. February 15-19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2560683.2560699.

applications.

This paper is organized as follows. Section 2 presents related works. Section 3 gives an introduction to programming in Skandium using skeletons and events. Section 4 shows our proposed solution to introduce autonomic characteristics to skeletons using events. Section 5 presents an example of our approach in action. Section 6, concludes and presents some future work.

2. RELATED WORKS

Autonomic capabilities have often be associated to component models [5]. Among them, the ParaPhrase project [15] is probably the closest to ours. It aims to produce a new design and implementation process based on adaptable parallel patterns. However, algorithmic skeletons are better adapted to express pure computational application patterns, compared to the structural pattern described by components. Thus our approach is better adapted to provide autonomic features for the computational aspects, while the ParaPhrase project gives those possibilities at the level of the application architecture. Also in the domain of components, several efforts have been made to give autonomic features in hierarchical structured component models like Fractal [6] and GCM [4] (Grid Component Model). Those works generally focus on the structure of the autonomic features and also the application architecture [22, 12]. Again, our approach here is much more focused on the computational aspects, and thus is complementary with these component-oriented approaches. The ASSIST framework [3] showed the complementarity of the two approaches by providing both structural support and computational support for autonomic aspects. Our work only focuses on the computational aspect, and improves the state of the art on the autonomic adaptation of skeletons; consequently, it can be used in a framework *Ãă la* ASSIST to improve the autonomic adaptation capacities of the framework, and finally obtain large-scale complex and structured component applications with efficient autonomic adaptations.

The second solution that we would like to highlight is part of the ASPARA project [13] leaded by Murray Cole and Horacio Gónzales-Vélez. This work proposes a methodology to introduce adaptability in skeletons. On ASPARA, structural information is introduced during compilation. Compared to ASPARA project, our solution allows the introduction of structural information during execution. This produces a higher level of adaptability because we can react faster to mismatch in the quality of service (QoS) desired.

The third related work is the Auto-tuning SkePU [11]. Here the prediction mechanism at execution time uses online pre-calculated estimates to construct an execution plan for any desired configuration with minimal overhead. Our solution does not need pre-calculated estimates, it calculates estimates at runtime. Again from the dynamic estimation of runtime and the dynamic adaptation of the skeletons, we are able to react faster to unsatisfactory quality of service.

As part of our work it is needed to estimate execution time for a skeleton and there is also related work on estimating parallel performance, [19]. Here, authors introduce a estimation method for parallel execution times, based on identifying separate "parts" of the work done by parallel programs. The time of parallel program execution is expressed in terms of the sequential work, called muscles in skeletons programming, and of the parallel penalty. Our parallel work estimation uses a different approach based on Activity Dependency Graphs (ADGs) which models the work estimation as an activity scheduling problem. The sequential work estimation uses a history-based estimation algorithm that allows on-the-fly estimation tuning.

To summarise, our approach here is much more focused on the computational aspects, it allows the introduction of structural information during at execution time, it does not need pre-calculated estimates since it can calculate them at runtime, and thus is complementary with the current approaches.

3. SKANDIUM LIBRARY

In Skandium, skeletons are provided as a Java Library. The library can nest task and data parallel skeletons according to the following syntax:

$$\begin{aligned}\Delta ::= \; & seq(f_e)|farm(\Delta)|pipe(\Delta_1,\Delta_2)|while(f_c,\Delta)| \\ & if(f_c,\Delta_{true},\Delta_{false})|for(n,\Delta)|map(f_s,\Delta,f_m)| \\ & fork(f_s,\{\Delta\},f_m)|d\&C(f_c,f_s,\Delta,f_m)\end{aligned}$$

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern, hidden away from the programmer. The task-parallel skeletons are: *seq* for wrapping execution functions; $farm$ for task replication; *pipe* for staged computation; $while/for$ for iteration; and if for conditional branching. The data-parallel skeletons are: *map* for single instruction multiple data; $fork$ which is like map but applies multiple instructions to multiple data; and $d\&C$ for divide and conquer.

The nested skeleton pattern (Δ) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. We denominate these blocks"muscles", as they provide the real (non-parallel) functionality of the application.

In Skandium, muscles come in four flavors: (1)"Execution", $f_e : P \rightarrow R$; (2) "Split", $f_s : P \rightarrow \{R\}$; (3)"Merge", $f_m : \{P\} \rightarrow R$; and (4); "Condition", $f_c : P \rightarrow boolean$. Where P is the parameter type, R the result type, and $\{X\}$ represents a list of elements of type X. Muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined Δ. The result of a muscle is passed as a parameter to other muscle(s) following dependencies defined by the skeleton program. When no further muscles need to be executed, the final result is delivered. Listing 1 shows an example of a common Skandium program; it is a skeleton with the following structure: $map(f_s, map(f_s, seq(f_e), f_m), f_m)$.

Separation of Concerns using Events

There are different programming models or strategies to address separation of concerns (SoC). Aspect Oriented Programming (AOP) is the preferred and widely used programming model. However, we have chosen to address SoC on Algorithmic Skeletons by using Event Driven Programming (EDP) for two reasons. First, there is no need to weave non-functional (NF) code as we can statically create event hooks as part of the Skeleton Framework. Second, the reactive nature of non-functional concerns is better reflected by events. For more details about this SoC approach, and

```
// Muscle's definition
Split<P,R> fs = new Split<P,R>() {
    @Override
    public R[] split(P p) {
        ...
    }
};

Execute<P,R> fe = new Execute<P,R>() {
    @Override
    public R execute(P p) {
        ...
    }
};

Merge<P,R> fm = new Merge<P,R>() {
    @Override
    public R merge(P[] p) {
        ...
    }
};

// Skeleton's definition
Map<P,R> nestedSkel = new Map<P,R>(fs, fe, fm);
Map<P,R> mainSkeleton =
  new Map<P,R>(fs, nestedSkel, fm);

// Input parameter
Future<R> future = mainSkeleton.input(new P(...));

// do something else

// Wait for result
R result = future.get();
```

Listing 1: Example: skeleton processing

its implementation on Skandium, refer to our previous work [20].

In short, skeleton's SoC can be addressed using EDP as follows. Events are triggered during a skeleton execution. Those events are statically defined during the skeleton's design and provide information on the actual skeleton execution (e.g., partial solutions and skeleton's trace). By means of event listeners, the NF programmer can implement NF concerns without touching the business logic code, i.e., the muscles. For example, Seq skeleton has two events defined: $seq(f_e)@b(i)$ (Seq Before), and $seq(f_e)@a(i)$ (Seq After). We use the notation $\Delta@event(information)$ to represent an *event* related to the skeleton Δ that provides the $information$ of the actual execution as event parameters. Map skeleton has eight events defined: the beginning of the skeleton, before and after the split muscle, before and after the nested skeleton, before and after the merge muscle, and at the end of the map. All events provide partial solutions, skeleton's trace, and an event identification, i, which allows correlation between Before and After events. Events also provide additional runtime information related to the computation; e.g., "Map After Split" provides the number of sub-problems created when splitting.

Listing 2 shows an example of a simple logger implemented using a generic listener. A generic listener is registered on all events raised during a skeleton execution. As handler parameters there is information of the event identification: skeleton trace, when (before or after), where (e.g. skeleton, split, merge), and i parameter. Additionally the partial solution, *param*, is sent and should be returned. This allows the possibility to modify partial solutions which could be very useful on non-functional concerns like encryption during communication. It is guaranteed that the handler is

```
mainSkeleton.addListener(new GenericListener() {
    @Override
    public Object handler(Object param,
      Skeleton[] st, int i, When when,
      Where where) {

        logger.log(Level.INFO,
          "CURRSKEL:"+st[st.length-1].getClass());

        logger.log(Level.INFO,
          "WHEN/WHERE:"+when + "/" + where);

        logger.log(Level.INFO,
          "INDEX:" + i);

        logger.log(Level.INFO,
          "PARTIAL_SOL:"+param.toString());

        logger.log(Level.INFO,
          "THREAD:"+Thread.currentThread());

        return param;
    }
});
```

Listing 2: Example: a simple logger

executed on the same thread than the related muscle (i.e. the next muscle to be executed after a $before$ event, and the previous muscle executed before an $after$ event).

As you can see, events allow us to precisely monitor the status of the execution of algorithmic skeletons. In the next section we show how using such events to provide a framework for the execution of skeletons with a high level of adaptability.

4. AUTONOMIC SKELETONS

AC is often used to achieve a given quality of service (QoS) that the system has to guarantee as much as possible. For example, a type of QoS is Wall Clock Time (time needed to complete the task). On an AC system, that supports the WCT QoS, it is possible to ask for a WCT of 100 seconds for the completion of a specific task. This means that the system will try to finish the job within 100s by means of self-managing activities (e.g., modifying process priorities or number of threads).

The current version of Skandium, 1.1b1, includes autonomic characteristics for two types of QoS: (1) Wall Clock Time, WCT, and (2) Level of Parallelism, LP. Actually these two types of QoS are related. If the system realizes that it won't target the WCT goal with the current LP, but it will do if the LP is increased, it autonomically increases the LP. However, if the system realizes that it will target the WCT goal even if it decreases the LP, it autonomically decreases the LP. To avoid potential overloading of the system, it is possible to define a maximum LP.

Why would one not always use the maximum number of threads in order to get fastest possible WCT? There are several reasons that drives the decision to not do so. First, energy consumption and heat production. The more work a processor does, the more energy it consumes, and the more heat it produces, implying even more energy needed for the cooler system. Another reason is to improve the overall system performance over the performance of a single application, when it is possible to free resources that could be used by other processes.

It is not always true that the WCT decreases if the num-

ber of active threads increases. The hardware cache system could lead to situations where higher performance is achieved with a number of threads even smaller than the available hardware threads. Even though, for simplicity, on this paper, we assume that the LP produces a non-strictly increasing speedup. The simplification obeys to the complexity of inferring the memory access pattern of the functional code.

The principles of our autonomic adaptation framework are the following. Using events, we can observe the computation and monitor its progress, we know when tasks start and finish and how many tasks are currently running. Thanks to the use of events, we can monitor this information without modifying the business code, and extremely dynamically: the execution of each individual skeleton is monitored. Consequently, we are able to adapt the execution as soon as we "detect" that the quality of service expected will not be achieved. In practice, we use functions to estimate the size and the duration of the problem to be solved, and, if necessary, allocate more resources to the resolution of a given skeleton. This way, the autonomic adaptation targets the currently evaluated instance, and not the next execution of the whole problem as in most other approaches.

We use events both to build up an estimation of the execution time, and to react instantly if the current execution may not be achieve in time. In the last case, more resource are allocated to improve the execution time. On the contrary, if the execution is fast enough or if the degree of parallelism must be limited, the number of threads allocated for the task can be decreased.

The algorithm to calculate the optimal WCT is a greedy one, while the algorithm to calculate the minimal number of threads to guarantee a WCT goal is NP-Complete. Therefore Skandium does not reduces the LP as fast as it increases it. The algorithm implemented for decreasing the number of threads first checks if the goal could be targeted using half of threads, if it can, it decreases the number of threads to the half.

The remaining of this section shows how we can estimate in advance the WCT. Let us introduce two simple functions: $t(m)$ and $|m|$. The former represents the estimated execution time of the muscle m, and the later represents the estimated cardinality of the muscle m. The estimated cardinality is only defined for the muscles of type Split and Condition. The cardinality of a muscle of type Split is the estimated size of the sub-problem set returned after its execution; the cardinality of a Condition muscle is the estimated number of times the muscle will return "true" over the execution of a While skeleton, or the estimated depth of the recursion tree of a Divide & Conquer skeleton. In a first time, we assume that we know in advance the values of the functions $t(m)$ and $|m|$. Then , it is possible to draw a Activity Dependency Graph (ADG) like the one shown on Figure 1.

Figure 1 shows an example of an ADG related to an actual skeleton execution similar to the example used in Section 3: two nested Map skeletons, $map(f_s, map(f_s, seq(f_e), f_m), f_m)$. Let's assume that $t(f_s) = 10$, $t(f_e) = 15$, $t(f_m) = 5$, and $|f_s| = 3$. Each activity, rectangle with three columns, corresponds to a muscle execution. The first and third columns represent the start and end time respectively. They could be an actual time (already passed) represented by a single light-gray box; or a best effort estimated time, represented by top boxes; or a limited LP estimated time, represented by a bottom boxes. The second column shows the muscle related to the activity. The skeleton have been executed using a LP of 2 threads, and the ADG has been taken at WCT of 70. In both cases, best effort or limited LP, estimated end time, t_f, is calculated as follows: $t_f = t_i + t(m)$, but if $t_i + t(m)$ is in the past, $t_f = currentTime$.

For example, the ADG of figure 1 shows that the execution started at time 0 where the first *split* muscle was executed, and finished its execution at time 10, producing 3 sub problems. Two of the three sub-problems started at time 10, but one started at time 65. As the ADG represents the situation at time 70, it is shown that the *split* that started at 65 has not yet finished, but it is expected to finish at 75 in either, the best effort case, or in the $LP(2)$ case.

The best effort strategy for estimating t_i uses the following formula: $t_i = max_{a \in A}(a_{t_f})$, where A is the set of predecessor activities and a_{t_f} is the end time of activity a. If $max_{a \in A}(a_{t_f})$ is in the past, $ti = currentTime$. Best effort strategy assumes an infinite LP; it calculates the best WCT possible, i.e. the end time of the last activity with a best-effort strategy.

Optimal LP is calculated using a time-line like the one presented on Figure 2. Figure 1 and figure 2 shows the same situation but in a different way. Figure 2 shows the estimated LP changes during the skeleton execution. It is possible to build the timetable, on the best effort case, using the start and end times estimated using the above formulas. It shows a maximum requirement of 3 active threads during the interval $[75, 90)$. Therefore the optimal LP for this example is 3 threads.

Limited LP strategy is used to calculate the total WCT under a limit of LP. In this case LP is not infinite, therefore the t_i calculation has an extra constraint: any point of time LP should not be over the limit. As you can see on Figure 2, the LP for the Limited LP case never exceeds 2 threads, and the total WCT will be 115.

If we set the WCT QoS goal to 100, Skandium will autonomically increase LP to 3 in order to achieve the goal.

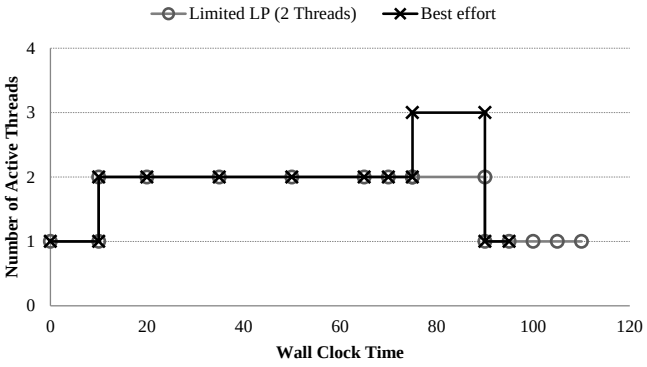

Figure 2: Example of timeline used to estimate the total WCT and the optimal level of parallelism

All this previous analysis have been done under the assumption that we already know the values of $t(m)$ and $|m|$. We will explain below how to estimate these values. The estimation algorithm implemented on skandium is based on history: "the best predictor of the future behaviour is past behaviour". The base formula for the calculation of the es-

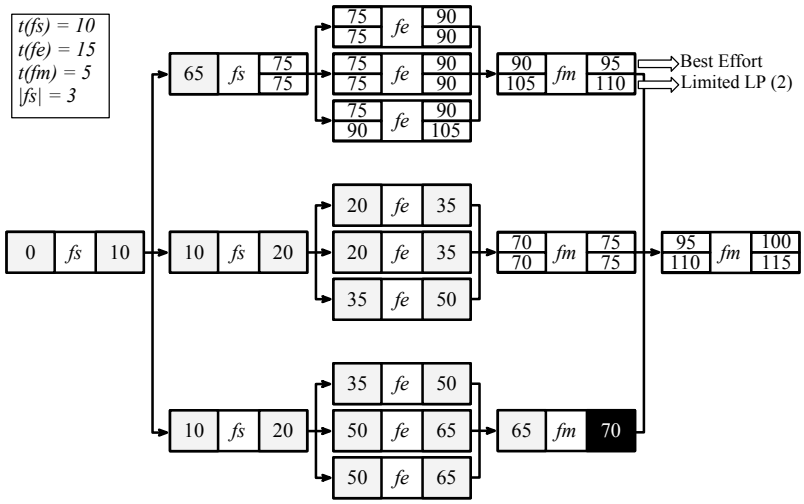

Figure 1: Example of an Activity Dependency Graph

timated new value is:

$$newEstimatedVal = \rho \times lastActualVal + (1-\rho) \times previousEstimatedVal$$

where ρ is a system parameter between 0 and 1, that defines the weight of the last actual value with respect to the previous ones. Its default value is 0.5 meaning that the estimated time is the average between the length of the previous execution, and the previous estimation. A proper value for ρ depends on the relation among previous values and the new expected value. For example, if ρ is set to 1, then only the last measure will be taken into account; but, if ρ is set to 0, then only the first value will be taken into account. Overall if ρ is closed to 0 then the value will not be too much sensitive to recent variations and the adaptation will be triggered slowly, following a stable tendency of results; whether if it is close to 1 the framework will quickly react to recent values.

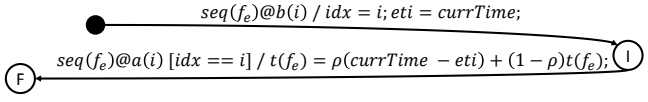

Figure 3: $StateMachine(seq(f_e))$ **definition**

Using events, it is possible to trace a skeleton execution without touching functional code. The natural way to design a system based on events is by State Machines. We use State Machines for tracking the Skeleton execution to (i) calculate the estimations based on the above formula, and (ii) create or update the ADG. Figures 3 and 4 show the state machines implemented for Seq and Map skeletons respectively.

Each type of skeleton included on Skandium has its corresponding state machine except If and $Fork$ skeletons which are not yet supported. If skeleton produces a duplication of the whole ADG that could lead to performance issues, and $Fork$ skeleton produces a non-deterministic state machine. The support for those types of skeletons are under construction.

The state machine for Seq skeleton is shown in Figure 3. It is activated once the $Seq\ Before$ event, $seq(f_e)@b(i)$, have been triggered. $Seq\ Before$ event has a parameter, i, that identifies the executed skeleton. Its value is stored on the local variable idx. Another local variable, eti, holds the time-stamp of the start of the muscle execution; once $Seq\ After$ event, $seq(f_e)@a(i)$, of the corresponding idx is triggered, the $t(f_e)$ value is computed and updated.

The state machine for Map skeleton, Figure 4, is a little more complex but its general idea is the same. Its goal is to trace the skeleton's execution, and to update the values of $t(f_s)$, $t(f_m)$, and $|f_s|$ as follows.

Map State Machine starts when a $Map\ Before\ Split$ event, $map(f_s, \Delta, f_m)@bs(i)$, is triggered. Similarly to Seq case, it has an identification parameter, i, that is stored in a local variable idx that serves as guard for the following state transitions. The time-stamp of the start of split muscle is stored in a local variable sti. The transition from state I to S occurs when the $Map\ After\ Split$ event,$map(f_s, \Delta, f_m)@as(i, fsCard)$, is triggered, where the $t(f_s)$ and $|f_s|$ estimations are updated. At this point all the children State Machines, $StateMachine(\Delta)$, are in hold waiting to start. When all children State Machines are in F state, the Map State Machine is waiting for $Map\ Before\ Merge$ event, $map(f_s, \Delta, f_m)@bm(i)$. Once it is raised, the mti local variable stores the time-stamp just before the execution of merge muscle. The transition from M to F state occurs when the $Map\ After\ Merge$ event, $map(f_s, \Delta, f_m)@am(i)$, occurs, where the $t(f_m)$ estimate is updated.

This estimation algorithm implies that the system has to wait until all muscles have been executed at least once. In the example, Figure 1, it is possible to estimate the work left of the skeletons still running because all muscles have been executed at least once at the moment of ADG analysis (black box). However, Skandium also supports the initialization of $t(m)$ and $|m|$ functions.

As presented in this section, our work guarantees a given execution time for a skeleton by optimizing autonomically the number of threads allocated to its execution and estimating the remaining execution time while the skeleton is running. We will show in the next section an execution example of this approach, showing that it is indeed efficient and allows autonomic adaptation to occur while the skele-

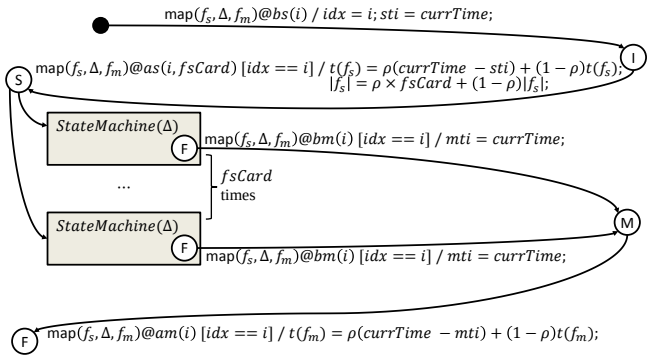

Figure 4: $StateMachine(map(f_s, \Delta, f_m))$ **definition**

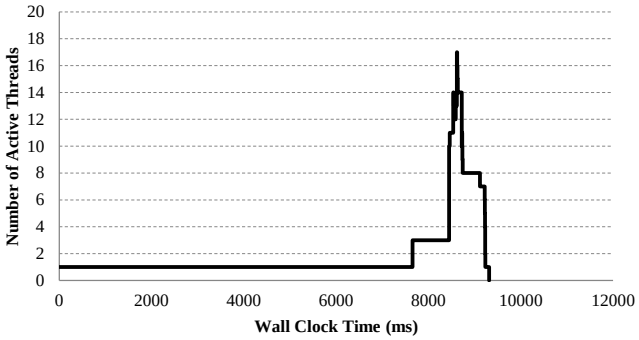

Figure 5: "Goal Without Initialization" execution

ton is being evaluated.

The proposed solution here is independent from the platform chosen for executing the skeleton. We illustrated our approach in a multicore setting, but it could also be adapted to a distributed execution environment. It could be achieved by a centralised distribution of tasks to distributed set of workers, adding or removing workers like adding or removing threads in a centralised manner. Taking decisions in a distributed manner would require more work. We believe that probably a hierarchical distributed algorithm would be more feasible than a pure distributed one.

5. EXECUTION EXAMPLE

This section presents an execution example to show the feasibility of our approach.

The example is an implementation a Hashtag and Commented-Users count of 1.2 million Colombian Twits from July 25Th to August 5Th of 2013 [10]. The problem was modelled as two nested Map skeletons: $map(f_s, map(f_s, seq(f_e), f_m), f_m)$, where f_s splits the input file on smaller chunks; f_e produces a Java HashMap of words (Hashtags and Commented-Users) and its corresponding partial count; and finally f_m merges partial counts into a global count.

The executions were done on an Intel(R) Xeon(R) E5645 at 2.4 GHz each, with a total of 12 cores and 24 CPU Threads, 64 GB RAM Memory, running Skandium v1.1b1.

To show the feasibility of our approach we present the comparison of three different execution scenarios:

1. "Goal without initialization". It is the autonomic execution with a WCT QoS set at 9.5 secs without initializing the estimation functions;
2. "Goal with initialization". it is the autonomic execution with a WCT QoS goal set at 9.5 secs with initialization of estimation functions.
3. "WCT Goal of 10.5 secs", it is the autonomic execution with WCT QoS goal of 10.5 secs.

Goal without initialization (figure 5)

The purpose of the first scenario is to show the behaviour of the autonomic properties without any previous information with an achievable goal.

In this scenario it is needed to wait until the first Merge muscle is executed in order to have all the information necessary to build the ADG. This occurs at 7.6 secs.

At that point, the first estimation analysis occurs and Skandium increments to 3 threads. It reaches its maximum number of active threads, 17, at 8.6 secs when almost all the f_e muscles can be executed in parallel. This scenario finishes its execution at a WCT of 9.3 secs reaching its targeted goal.

But, why we chose a goal of 9.5 secs? The total sequential work (WCT of the execution with 1 thread) takes 12.5 secs, therefore any goal greater than 12.5 secs won't produce the necessity of an LP increase. On the other hand, Skandium took 7.6 secs to execute at least the first Split, one other Split, all the execution muscles of second Split, and one Merge. The first split took 6.4 secs (as we will see on scenario 2), and it is expected to have the second level split 7 times faster than the first one, and 0.04 secs per Execution and Merge muscles, therefore in the best case it is expected that the system could finish at 8.63 secs. What could occur is that the increment of threads happened after any of the left splits has already started its execution in such case we cannot achieve the maximum LP and it is needed to wait an extra split execution (in the worst case) which implies a WCT of 9.54 secs.

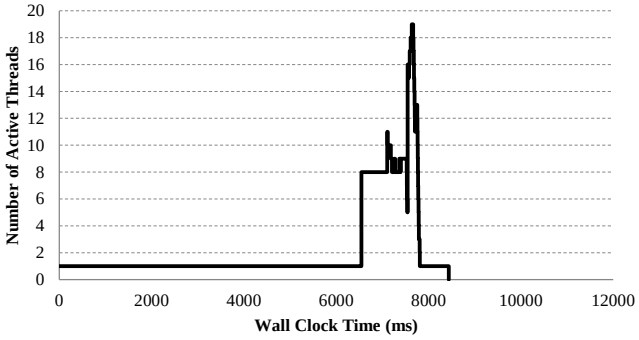

Figure 6: "Goal With Initialization" execution

Goal with initialization (figure 6)

The purpose of the second scenario is to show how a previous execution could give important information that can be used in order to improve the estimations. In this scenario we chose a goal of 9.5 in order to do the comparison with the same parameters of the first scenario except the initialization of variables.

Here the $t(m)$ and $|m|$ functions are initialized with their

corresponding final value of a previous execution. Figure 6 shows that Skandium increases the number of threads to 8 at 6.4 secs of WCT execution. As you can notice it is before the first Merge muscle has been executed. Skandium does not increase the number of threads before because it is performing I/O tasks, i.e., reading the input file stream on the first split muscle where there is no need for more than one thread. The execution reaches its maximum LP, 19 active threads, at 7.6 secs. This scenario finishes its execution at a WCT of 8.4 secs. It shows a better execution time than experiment (1) where Skandium needed some time to detect that the WCT could not be achieved. One can notice that experiment (1) shows the additional cost paid during the first execution in order to initialize the cost functions.

This experiment finishes at 1.1 secs before the targeted goal. The reason is the algorithm implemented for decreasing the number of threads. As described in previous section, Skandium does not reduces the LP as fast as it increases it producing an early WCT.

It may be expected that this execution uses all 24 threads in its maximum LP. Theoretically all the execution muscles should have been executed in parallel and therefore all physical threads should be used, but in practice some execution muscles took less time than others, and at the end the scheduler did not have the necessity of activate all threads.

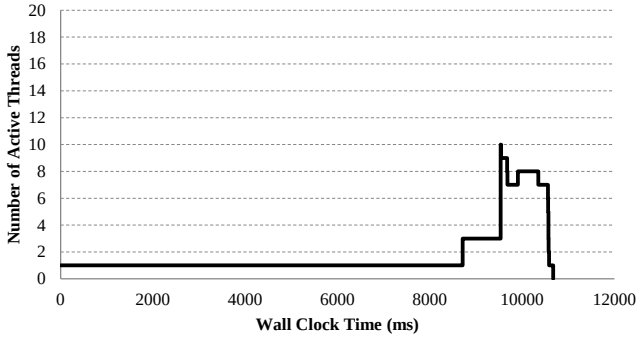

Figure 7: "WCT Goal of 10.5 secs" execution

WCT goal of 10.5 secs (figure 7)

The purpose of this scenario is to show the behaviour of the LP when it is needed to increase the number of threads but not as much as in the first scenario. In this scenario Skandium has more clearance and it is expected that the maximum LP is lower than the maximum LP of the two previous executions.

Here, figure 7 shows that Skandium, at 8.7 secs of execution, realizes that it won't target its goal with the current LP, therefore it increases the LP to a maximum of 10 active threads.

Note that the maximum LP of this execution is lower than the used on the two previous executions because the WCT goal has more room to allocate activities with less number of threads. It finishes at 10.6 secs.

This example has shown the feasibility of our proposed solution. We illustrated how instrumenting skeletons with events allowed us to discover that the execution of a skeleton might be too slow or too fast and to adapt the degree of parallelism dynamically, during the execution of this same skeleton. Not only our methodology allows us to adapt faster the degree of parallelism, but it is also adapted to cases where the execution time is dependent on the input data, while strategies using the execution time of the previous instance to adapt the degree of parallelism of the next skeleton are unable to achieve this.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed how skeletons together with autonomic computing present a promising solution for the autonomic adaptation of applications. We illustrated our approach in the setting of a multi-core execution environment independent from the platform chosen for executing the skeleton.

Our approach relies on the use of events both to build up execution time estimators, and to adapt rapidly the degree of parallelism in case the desired execution time cannot be achieved with the resources currently allocated to the computation. We have described and shown the feasibility our proposal by introducing self-configuration and self-optimization autonomic characteristics to skeletons using event driven programming techniques. We have shown that the use of events allow us to precisely monitor the status of the execution of skeletons and improves the estimation of the remaining computation time. If the expected run time can not be reached, then additional threads can be allocated to the execution in order to achieved the required QoS.

The proposed solution is independent from the platform chosen for executing the skeleton. We illustrated our approach in a multicore setting, but it could also be adapted to a distributed execution environment.

Here we discuss about the QoS of level of parallelism and Wall Clock Time, but as discussed by M.Aldinucci et al on [1], there are different QoS and non-functional concerns that are widely studied and incorporated as autonomic characteristics (e.g. dynamic load balancing, adaptation of parallelism exploitation pattern to varying features of the target architecture and/or application, among others). Our current research is focusing on the study of this different QoS in order to improve the scalability and maintainability of the systems build on Skandium, or in general, on Skeletons.

Other initiatives based on this work involves the analyses of different WCT estimation algorithms comparing its overhead costs, and more experiments are conducted on other benchmarks.

7. REFERENCES

[1] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, 2009.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Carlo Montangero, and Laura Semini. Managing adaptivity in parallel systems. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 199–217. Springer Berlin Heidelberg, 2013.

[3] Marco Aldinucci, Marco Danelutto, and Marco Vanneschi. Autonomic qos in assist grid-aware components. In *14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2006)*.

[4] Françoise Baude, Denis Caromel, CÃl'dric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian PÃl'rez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1-2):5–24, 2009.

[5] Françoise Baude, Ludovic Henrio, and Paul Naoumenko. A Component Platform for Experimenting with Autonomic Composition. In *First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007). Invited Paper*. ACM Digital Library, October 2007.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien QuÃl'ma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[7] D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010. <ce:title>ICCS 2010</ce:title>.

[8] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[9] Alexander Collins, Christian Fensch, and Hugh Leather. Auto-tuning parallel skeletons. *Parallel Processing Letters*, 22(2), 2012.

[10] Data created using Twitter4j (http://twitter4j.org). Source raw data used for the example on section 5: 1.2 million colombian twits from july 25th to august 5th of 2013. https://drive.google.com/file/d/0B_KljwYYwPn0S0Nob3NTX29XcHc, August 2013. [Online; accessed 7-January-2014].

[11] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[12] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006. 10.1007/11821946 6.

[13] Horacio González-Vélez and Murray Cole. Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. *Concurr. Comput. : Pract. Exper.*, 22(15):2073–2094, October 2010.

[14] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[15] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio GonzÃąlez-VÃl'lez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 218–236. Springer Berlin Heidelberg, 2013.

[16] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, 2001.

[17] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, January 2003.

[18] M. Leyton and J.M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, 2010.

[19] Oleg Lobachev, Michael Guthe, and Rita Loogen. Estimating parallel performance. *Journal of Parallel and Distributed Computing*, 73(6):876 – 887, 2013.

[20] G. Pabon and M. Leyton. Tackling algorithmic skeleton's inversion of control. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 42–46, 2012.

[21] Kurt Potter, Michael Smith, Jamie K. Guevara, Linda Hall, and Eric Stegman. It metrics: It spending and staffing report, 2011. Technical report, Gartner, Inc.

[22] Cristian Ruz, Françoise Baude, and Bastien Sauvan. Using Components to Provide a Flexible Adaptation Loop to Component-based SOA Applications. *International Journal on Advances in Intelligent Systems*, 5(1 2):32–50, July 2012. ISSN: 1942-2679.