Scientific Programming 22 (2014) 261–271
DOI 10.3233/SPR-140392
IOS Press

ELASTIC: A large scale dynamic tuning environment

Andrea Martínez *, Anna Sikora, Eduardo César and Joan Sorribes
Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain
E-mails: {amartinez, ania}@caos.uab.es, {eduardo.cesar, joan.sorribes}@uab.cat

Abstract. The spectacular growth in the number of cores in current supercomputers poses design challenges for the development of performance analysis and tuning tools. To be effective, such analysis and tuning tools must be scalable and be able to manage the dynamic behaviour of parallel applications. In this work, we present ELASTIC, an environment for dynamic tuning of large-scale parallel applications. To be scalable, the architecture of ELASTIC takes the form of a hierarchical tuning network of nodes that perform a distributed analysis and tuning process. Moreover, the tuning network topology can be configured to adapt itself to the size of the parallel application. To guide the dynamic tuning process, ELASTIC supports a plugin architecture. These plugins, called ELASTIC packages, allow the integration of different tuning strategies into ELASTIC. We also present experimental tests conducted using ELASTIC, showing its effectiveness to improve the performance of large-scale parallel applications.

Keywords: Dynamic tuning, scalability, tuning network, performance tools

1. Introduction

Supercomputers are a widely used resource in many areas of modern research. However they are a costly resource, and access is often limited to an allocation of execution hours.

Normally, parallel applications running on supercomputers do not make efficient use of resources. This provokes longer than expected running times, which "waste" computation hours and reduce the available time for further executions. In this context, analysis and tuning tools that identify, understand and fix performance problems are more valuable than ever.

To apply performance analysis and tuning to parallel applications executed on supercomputers, it is paramount that these tools have been specifically designed following a scalable and modular architecture that enables the control and analysis of an extremely large number of tasks.

Nowadays, there are automatic performance analysis tools, such as Scalasca [9], Periscope [1] or TAU [13], capable of scaling and looking for performance problems of parallel applications. Nevertheless, if the behaviour of the parallel applications depends on input data, or may even change during each execution due to data evolution, such analysis could not be enough. In these cases, dynamic analysis and tuning of the application during its execution, such as that performed by MATE [10], Active Harmony [14], and other tools, is necessary. However, most of these tuning tools do not scale well, mainly due to a centralised analysis process.

Taking into consideration these facts, this paper addresses the lack of large-scale dynamic tuning in the current performance analysis area. We present the design and implementation of ELASTIC, a dynamic tuning environment for MPI large-scale parallel applications. Its operation supports continuous and automatic monitoring, performance analysis and modifications while the application is running. ELASTIC's design is based on a novel model [4,5] whose scalability properties arise from the decentralisation of the dynamic tuning process. From this model, ELASTIC's architecture is structured as a hierarchical tuning network of distributed nodes that conduct dynamic tuning in a decentralised manner.

Using ELASTIC, an experimental evaluation has been carried out, consisting of dynamically detecting and resolving performance problems over a large-scale parallel application. From the obtained results, it can be concluded that ELASTIC is able to scale, and effectively improve the performance of large-scale parallel applications at runtime.

*Corresponding author. E-mail: amartinez@caos.uab.es.

1058-9244/14/$27.50 © 2014 – IOS Press and the authors. All rights reserved

The remainder of this paper is organised as follows. Section 2 presents an overview of the model from which ELASTIC's design arises. Section 3 describes ELASTIC's design, followed by details of its implementation in Section 4. The experimental evaluation performed using ELASTIC is depicted in Section 5. Section 6 describes the related work. The conclusions and outlook are presented in Section 7.

2. Model for hierarchical dynamic tuning

ELASTIC's design is based on a model that enables the hierarchical distribution of the analysis and tuning process. It uses an abstraction mechanism that offers a reduced representation of the application state, enabling global performance improvements to be achieved.

Let $\mathbb{A} := \{t_i\colon 0 \leqslant i \leqslant n-1\}$ be a parallel application composed of n distributed processes, which we will call *tasks*. When $\mathbb{A}$ is too large to be dynamically analysed and tuned in a centralised manner, the proposed model decomposes it into a number of non-empty subsets of tasks, called *domains*, which can be dealt with separately.

The *Decomposition Process* over $\mathbb{A}$, $\mathbb{A} \mapsto \{D_k\colon 0 \leqslant k \leqslant d-1\}$, must produce domains, D_k, that are disjoint and cover the entire application. Dynamic tuning of each of these separate domains will lead to local performance improvements in the parallel application.

To achieve global performance improvements, the model proposes the *Abstraction Process*. In this process, each domain obtained after decomposing the original parallel application is represented as a single *virtual* parallel application task, $v_k^{(0)} = \text{abstract}(D_k)$, $0 \leqslant k \leqslant d-1$.

This abstraction means that the performance of a virtual task is analysed using information collected from the domain that it represents and, similarly, tuning applied to a virtual task is actually carried out on the underlying domain.

When taken together, the virtual tasks representing all the domains of the original application form a new virtual parallel application, $\mathbb{V}^{(0)} = \{v_k^{(0)}\colon 0 \leqslant k \leqslant d-1\}$, which is composed of fewer tasks than the original.

While the number of tasks of the virtual parallel application is still too great to be analysed in a centralised manner, the decomposition and abstraction processes are repeated:

(1) *Decomposition*. A virtual parallel application at level ℓ is decomposed, $\mathbb{V}^{(\ell)} \mapsto \{D_k^{(\ell)}\colon 0 \leqslant k \leqslant d_\ell - 1\}$.
(2) *Abstraction*. The domains are abstracted to create new virtual tasks at level $\ell+1$, $v_k^{(\ell+1)} = \text{abstract}(D_k^{(\ell)})$, $0 \leqslant k \leqslant d_\ell - 1$, which form a virtual parallel application at this level, $\mathbb{V}^{(\ell+1)} = \{v_k^{(\ell+1)}\colon 0 \leqslant k \leqslant d_\ell - 1\}$.

Following these decomposition and abstraction processes, we obtain a hierarchy of virtual parallel applications, with the lowest level being the original parallel application and the highest level having few enough tasks to be analysed and tuned in a centralised manner. Figure 1 shows how the model works for an SPMD application. Level ℓ is the root when $D_0^{(\ell)} = \mathbb{V}^{(\ell)}$.

Application decomposition and the abstraction mechanism permit analysis and tuning to be performed separately on each domain composed of real or virtual tasks. This pattern leads to a hierarchical distribution of the analysis and tuning process of the parallel application. As such, the analysis and tuning conducted over the virtual parallel application at the highest level results in a global performance improvement over the real parallel application.

In order to offer effective distributed dynamic tuning, the model follows a collaborative approach which requires the integration of user knowledge to guide the performance analysis and tuning and to define the abstraction process. This information is codified in the form of a performance model and an abstraction model respectively.

Performance models are a set of analytical expressions that represent the behaviour of a parallel application with the aim of predicting its performance. To represent a performance model for dynamic tuning, we use terminology from MATE [10]:

- A set of measurement points, which determines the parameters of the application to be monitored and the points where they have to be measured.
- A set of evaluation strategies and/or expressions used for finding performance problems and giving solutions to them. Such strategies and expressions are evaluated on the data collected from the measurement points.
- A set of tuning points and actions, and a synchronisation method. A tuning point specifies what must be changed in the application, a tuning action is the change to be performed on that point, and the synchronisation method determines the conditions that must hold to perform the tuning action in a consistent manner.

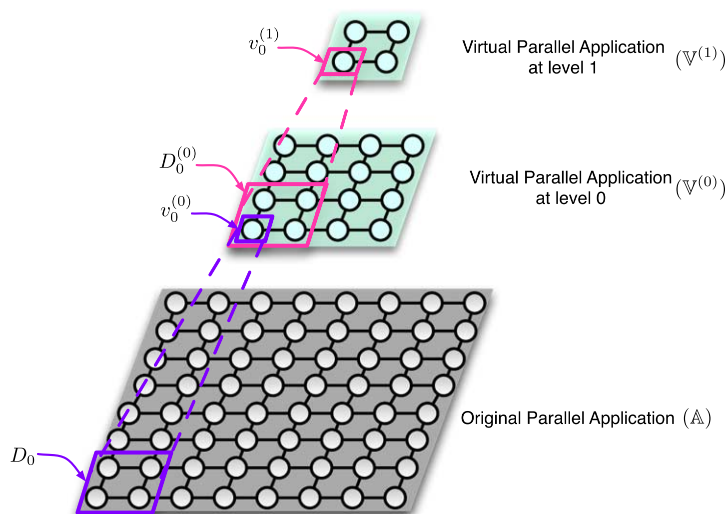


Fig. 1. Decomposition and abstraction of an SPMD application following the model for hierarchical dynamic tuning. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

The model for hierarchical dynamic tuning also requires a description about how to perform the abstraction between levels in the hierarchy. This knowledge forms the *Abstraction model*. When a virtual task is dynamically tuned following a performance model, the abstraction model must provide information about:

- How to calculate the measurement points of a virtual task from the events produced in the set of virtual or real tasks belonging to the domain that it represents.
- How to translate the set of tuning points, actions and synchronisation method of a virtual task to be applied to the set of virtual or real tasks from the domain that it represents.
- How to decompose a real or virtual parallel application into domains of tasks which can be analysed and tuned following the performance model to be used at its level in the hierarchy.

Therefore, using performance models, dynamic tuning can be applied to any domain of tasks, whether real or virtual. When dynamic tuning is applied to domains composed of virtual tasks, using the abstraction mechanism such process is actually occurring on the set of domains of real tasks that they represent.

The model presented can be used to attack certain problems that only require local analysis to be solved, for example, optimising the memory usage of an application task. In this case, no abstraction occurs as only analysis in the base of the hierarchy is necessary. Other problems can only be solved at a global level, while the rest of the hierarchy supports it via the abstraction mechanism. An example of such a situation is tuning MPI parameters, which must be the same for all application tasks.

However, many problems can be attacked in a hierarchically distributed manner, where analysis is performed at all levels in the hierarchy. Load balancing can be performed both locally and globally in such a way that local performance improvements can be achieved without worsening the global performance of the application.

3. Design

ELASTIC is an environment for large-scale dynamic tuning, whose design is based on the model proposed in [4] and summarised in Section 2. In this section we outline the translation of this model into the design of ELASTIC.

To create a decentralised analysis and tuning process, as well as to fit the model requirements, a tuning network of distributed nodes is structured as a hierarchical tree. Each node in the tuning network is formed of two modules: an Analysis and Tuning Module (ATM) and an *Abstractor* (see Fig. 2).

The base of the tuning network is made up of *Abstractor*–ATM pairs, where each ATM is responsible for improving the performance of a disjoint subset of parallel application tasks, called its *analysis and tuning domain*. To do it, the ATMs operate in three continuous phases: monitoring, performance analysis and modification. First, the ATMs instrument the parallel

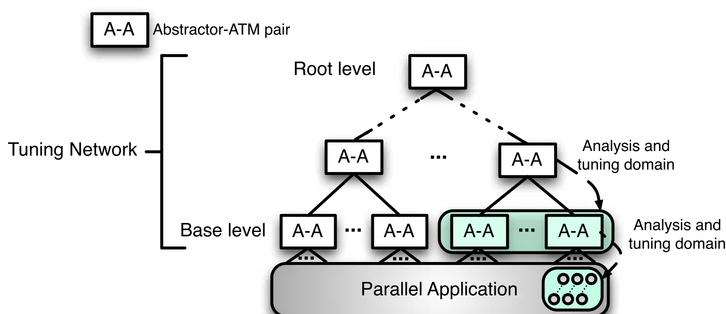


Fig. 2. General hierarchical tuning network. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

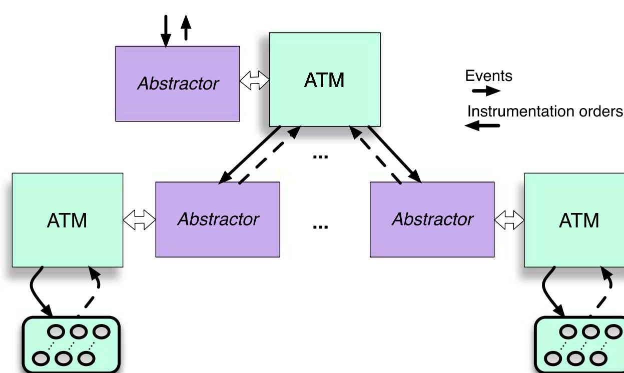


Fig. 3. Detail of Fig. 2 showing the *Abstractor*–ATM connections in the tuning network. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

application, using *instrumentation orders for monitoring*. These orders indicate the points in the applications that have to be measured to gather information about its behaviour. In the analysis phase, ATMs receive this information in form of *events*, search for bottlenecks, and give solutions for overcoming them. Events are messages generated by the parallel application tasks which contain the information previously requested by the monitoring orders. Finally, *instrumentation orders for tuning* are sent by the ATMs to apply the given solution. The tuning orders specify the points to be dynamically changed in the application to improve its performance.

The decentralised dynamic tuning process performed by the base level ATMs leads to local performance improvements in the parallel application. However, to obtain global performance improvements, dynamic tuning of larger segments of the application is required. Following the abstraction concepts of the model, it is necessary to abstract each domains of real tasks. In the design, this abstraction is provided by the *Abstractor* components associated to each of the base level ATMs.

The *Abstractor* receives events about the state of the children in the analysis and tuning domain of its associated ATM. Then, it summarises this information and sent it to its parent ATM in the form of a new event. When an *Abstractor* receives an instrumentation order for monitoring or tuning from its parent ATM, it must translate the order to be applied to the analysis and tuning domain of its associated ATM. As it can be seen in Fig. 3, the *Abstractor* makes the connection with the parent ATM, and the ATM associated to this *Abstractor* connects with its analysis and tuning domain, i.e., its immediate children. Consequently, the *Abstractor* is responsible for representing its associated ATM as a parallel application task (the virtual task of the model) to its parent ATM.

Following the proposed model, while the number of ATMs at the base level is too big to be analysed in a centralised manner, these ATMs are decomposed into domains that can be tuned separately. In the design, this tuning process will be carried out by the ATMs at the higher level in the hierarchy. Therefore, the base level ATMs become the analysis and tuning domains of the ATMs located at the higher level in the hierarchy.

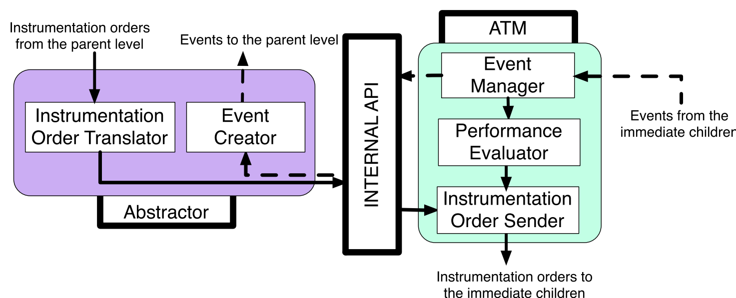


Fig. 4. Abstraction mechanism: *Abstractor*–ATM design. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

The abstraction and decomposition process continues until the set of ATMs at a specific level can be analysed in a centralised manner by a single ATM, the root ATM of the tuning network. Finally, the model is designed as a hierarchical tuning network of *Abstractor*–ATM pairs. Each ATM is able to dynamically tune its analysis and tuning domain, which leads to a hierarchical distribution of the analysis and tuning process. Each *Abstractor* also virtualises the domain of its associated ATM and presents it, to its parent ATM, as an application task.

The operation of the proposed hierarchical tuning network is based on the action of the *Abstractor*–ATM pair as a single entity. As it can be seen in Fig. 3, the *Abstractor* makes the connection with the parent ATM, and the ATM associated to this *Abstractor* connects with its analysis and tuning domain. The functional design of the *Abstractor*–ATM pair and its communication paths are shown in Fig. 4.

The *Abstractor* is composed of two main modules, in which the knowledge about the abstraction model is contained:

- *Instrumentation Order Translator*. Once an instrumentation order for monitoring or tuning is received by the *Abstractor*, the *Instrumentation Order Translator* transforms it into one or more new instrumentation orders which will be sent to the children.
- *Event Creator*. This module creates events using the data contained in events received from its associated ATM. Created events encapsulate the information requested by monitoring orders previously received by the *Abstractor* from its parent ATM.

The *Abstractor* communicates with its associated ATM via an internal API. Every ATM is composed of three modules:

- *Performance Evaluator*. This module contains the information necessary to carry out the monitoring, analysis and tuning of the parallel application, i.e., the performance model.
- *Event Manager*. This module is responsible for receiving and managing events generated by the parallel application tasks or other descendant ATMs. An event received by this module may be created as a result of the monitoring instrumentation orders generated by its *Performance Evaluator* or the *Performance Evaluator* of an ancestor ATM. In the first case, the *Event Manager* will transfer this event to its *Performance Evaluator*, and in the second case it will transfer it to the *Event Creator* module of its associated *Abstractor*. In this way the events required by the ancestor ATMs may flow through the hierarchy. The knowledge required to route incoming events is based on whether the monitoring order which provoked the generation of this event originated in the ATM or the *Abstractor* at this level.
- *Instrumentation Order Sender*. This module has to send the instrumentation orders received from its *Performance Evaluator*, or from its associated *Abstractor*, to its analysis and tuning domain.

4. Implementation

The current implementation of ELASTIC can be used to dynamically improve the performance of MPI parallel application running on UNIX systems.

4.1. Architecture

ELASTIC's architecture, shown in Fig. 5, is based on a hierarchical network composed of the following components:

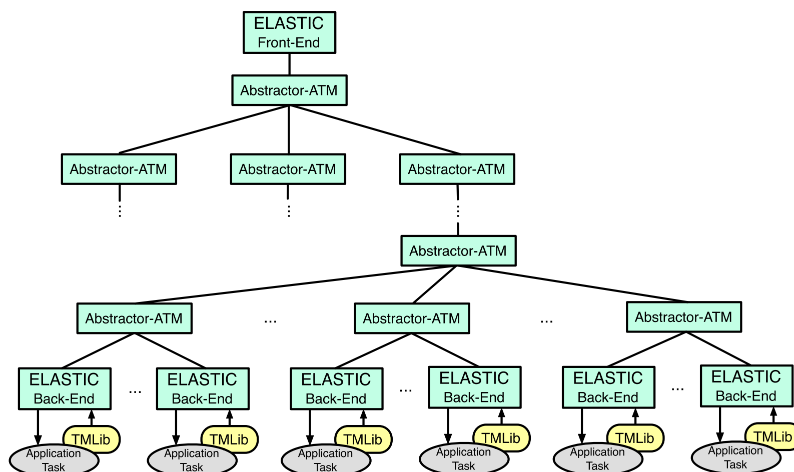


Fig. 5. ELASTIC's architecture. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

- *ELASTIC Front-End (EFE)*. The root process of the network, responsible for creating and instantiating the internal processes of the tuning network.
- *Abstractor–Analysis and Tuning Module (ATM) pairs*. Located at the internal nodes of the tuning network, they carry out the distributed performance analysis and tuning of the application, as well as providing the abstraction mechanism between levels in the hierarchy.
- *ELASTIC Back-End (EBE)*. A daemon that controls the execution and dynamic instrumentation of each parallel application task, receiving monitoring and tuning orders from the *Abstractor*–ATM pair.
- *Task Monitoring Library (TMLib)*. A shared library that is dynamically loaded by the EBE in each task, to support monitoring and performance data gathering.

These components cooperate to control and improve the execution of a parallel application through continuous monitoring, analysis and tuning. In the monitoring phase, ELASTIC uses event tracing to collect information about the application at the task level. This information is sent to the *Abstractor*–ATM pairs, where automatic performance analysis is conducted. After detecting a performance problem, tuning orders are sent to the EBE to be inserted into the application task at runtime.

The hierarchical communication layer of ELASTIC is established through MRNet [12]. This framework allows for the connection of the EFE with the EBEs as well as enabling the *Abstractor*–ATM pairs to be distributed across MRNet's TBON of internal processes, utilising the strength of filters. Filters are functions located at the internal nodes of the TBON that synchronise and aggregate the data that flows through the hierarchy. More details of how ELASTIC takes advantage of the features of MRNet can be found in [4].

ELASTIC uses the DynInst Library [3] to dynamically generate code and insert it into the application during runtime, without recompiling or restarting the parallel application. Specifically, via DynInst, ELASTIC inserts into the parallel application tasks (a) monitoring code to generate events to be traced, and (b) tuning code to apply the solution obtained after the performance analysis with the aim of improving the performance of the application.

It should be noted that the tuning network topology of ELASTIC can be configured to accommodate the size of the parallel application and the complexity of the tuning strategy being employed [6].

4.2. *ELASTIC package*

The knowledge required to guide the performance analysis and tuning process is integrated into ELASTIC in the form of plugins called ELASTIC packages. An ELASTIC package is a set of code and configurations that allow a tuning strategy. It is applied to improve the performance of a parallel application and is specified in terms of performance and abstraction models. To develop ELASTIC packages, ELASTIC provides the *Tuning and Abstraction API*. This API consists of a set of methods from different classes which represent the components of the *Abstractor*–ATM pair and their behaviour. Figure 6 shows the specific methods of the *Tuning and Abstraction API*

which must be overwritten, codifying the performance and abstraction models, to implement an ELASTIC package. Decomposition occurs prior to run time, and it must be manually defined outside the ELASTIC package.

The ELASTIC's plugin architecture gives the flexibility to tackle a wide range of performance problems just by interchanging the ELASTIC packages.

5. Experimental evaluation

We have carried out several experimental tests with the aim of proving the effectiveness and feasibility of ELASTIC as an environment that can be used to dynamically improve the performance of MPI large-scale parallel applications.

The evaluation consists of executing a parallel application which presents a specific performance problem and using ELASTIC to dynamically detect and resolve this problem. In this experimentation, we have used a synthetic parallel application with controlled load imbalance. It allows for testing different configurations in the most controlled manner.

The synthetic application has been developed following the SPMD programming paradigm using MPI as the library for inter-process communication.

The tasks that compose the application are logically arranged in a square two dimensional grid. Each task is only able to communicate directly with its neighbouring tasks.

Each iteration consists of a computation phase followed by a communication phase. Each task has a number of work units, which represent a fixed amount of computation to be performed in each iteration. In this way, the amount of work to be performed by each task in the computation phase is proportional to the number of *work units* it has. In the communication phase, each task exchanges messages with all of its neighbouring tasks. Each message has a fixed size. The number of iterations of the application is set to 100.

In order to ensure that the synthetic application was able to operate in a large-scale context, the amount of work to be performed in each iteration was kept equivalent to the number of parallel application tasks. In our experimental evaluation, each task started the execution with 20 work units. The dynamic imbalance inserted into the application also remains proportional to the size of the application.

To demonstrate the dynamic tuning capabilities of ELASTIC, load imbalance was introduced into the synthetic application at runtime in form of points of additional load, *hotspots*, localised to logical areas of the application grid.

(1) *Single Localised Hotspot*: This scenario is characterised by presenting a single area of additional workload located at the centre of the application grid. The introduced load accounts for an additional 10% of the initial application workload for all application sizes.
(2) *Multiple Hotspots*: In this scenario the additional load introduced is distributed amongst a number of smaller hotspots. Hotspots are introduced in three groups, each group at specific moments during the execution of the synthetic application.

Each group of hotspots represents an additional 7% of the initial application load (21% throughout the entire execution) for all the application sizes. The hotspots are randomly placed in the application grid.

5.1. Load balancing ELASTIC package

To resolve the inefficiencies related to the load imbalance problems, an ELASTIC package has been designed and integrated into ELASTIC. It should be note that the same ELASTIC package is used in all the *Abstractor*–ATM pairs. Figure 7 outlines the content of this ELASTIC package, divided into its two conceptual parts, the performance model and the abstraction model.

This ELASTIC package attempts to balance the number of work units evenly amongst all the tasks in the synthetic application. Each instance balances the load in the analysis and tuning domain of the ATM where it is located. Firstly, the package monitors the state of its domain and detects which tasks are underloaded and which are overloaded. Then, the package decides how to redistribute the work units according to the communication pattern presented in the synthetic application using local migrations between neighbouring tasks. The load balancing algorithm used in this ELASTIC package is proposed in [4].

5.2. Effectiveness evaluation

The experimental evaluation was executed on the supercomputer SuperMUC at Leibniz Supercomputing Centre, using up to three islands, each one composed of 512 nodes interconnected by Infiniband FDR10. The nodes have 2 8-core 2.7 GHz Intel Xeon processors and run SuSe Linux.

```
vector<Monitoring Order> PerformanceEvaluator::InitialMonitoringOrders()
```
Contains the monitoring points of the performance model.

```
bool PerformanceEvaluator::NewEvent(Event *e)
```
Manages the storage of events and recognise when the performance evaluation can be activated.

```
vector<Order> PerformanceEvaluator::EvaluatePerformance()
```
Conducts the performance problem detection process.

```
vector<Monitoring Order> InstrumentationOrderTranslator::
                         TranslateMonitoringOrder(MonitoringOrder *mo)
```
Converts monitoring orders from the parent level to be applied to the level below.

```
vector<Tuning Order> InstrumentationOrderTranslator::
                         TranslateTuningOrder(TuningOrder *to)
```
Converts tuning orders from the parent level to be applied to the level below.

```
bool EventCreator::NewEvent(Event *e)
```
Stores the events required to create a new event.

```
vector<Event> EventCreator::CreateEvent()
```
Creates a new event using the information about each of the events recorded in the EventCreator::NewEvent function.

Fig. 6. Tuning and abstraction methods required to implement an ELASTIC package.

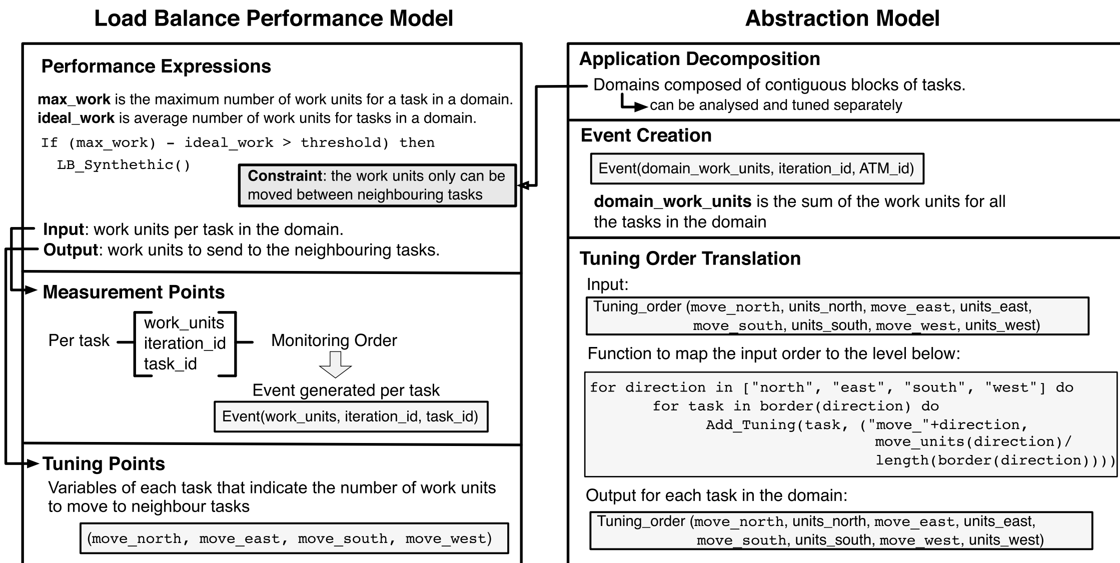


Fig. 7. ELASTIC package for load balancing in the synthetic application.

The synthetic application was executed with different numbers of tasks from 256 up to 16,384 tasks. To dynamically tune the application, the topology of ELASTIC's tuning network was chosen following the method proposed in [6], which choses topologies composed by the minimum number of non-saturated *Abstractor*–ATM pairs. The details of these topologies are given in Table 1.

The 256 task application is dynamically tuned by a centralised tuning network, for the rest of the number of tasks, the application is dynamically tuned by networks of two levels.

ELASTIC's tuning network is located on different physical nodes from those running the synthetical parallel application, giving each *Abstractor*–ATM four cores due to multi-threaded nature of MRNet internal nodes [2]. This does not include the EBEs, which run in the same core as the task they control.

Table 1
ELASTIC tuning network topologies over the synthetic application

Number of application tasks	Level 0 number of ATMs	Level 1 number of ATMs
256	1	–
1,024	4	1
2,304	9	1
4,096	16	1
9,216	36	1
16,384	64	1

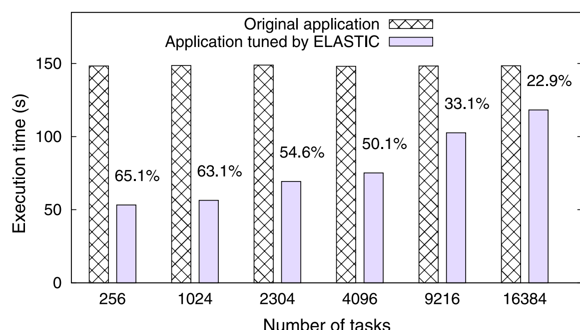


Fig. 8. Centralised hotspot scenario: execution time of the original and tuned synthetic application. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

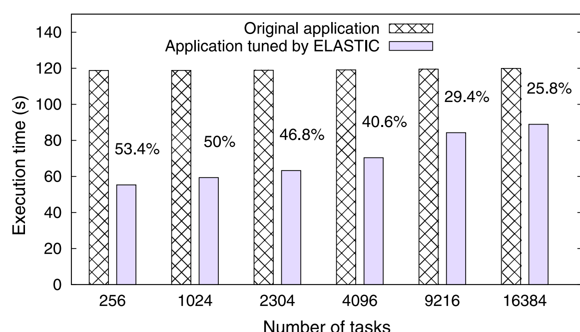


Fig. 9. Multiple hotspot scenario: execution time of the original and tuned synthetic application. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140392.)

Figures 8 and 9 present the execution time of the synthetic application with and without ELASTIC performing dynamic tuning, for the two considered scenarios, single and multiple hotspot respectively. The percentage presented depicts the reduction in the execution time when the application is tuned by ELASTIC compared to the original application.

For both scenarios, the effects of the load balance conducted by ELASTIC are reflected in an improvement of the application performance in terms of a reduction in the total execution time of the synthetic application for all sizes tested.

The improvement is reduced with the size of the parallel application. This is due to the restrictions placed on the migration process, where work units can only be migrated between neighbouring tasks. As the application grid size increases, it takes longer to spread the load throughout the grid, irrespective of the load balancing strategy employed. This is a problem with the lack of scalability of the local migrations available to this application, and not an artefact of ELASTIC's scalability.

It can be seen that the reduction in the performance improvement gained with the size of the synthetic application is not as pronounced in the multiple hotspot scenario as in the case of the centralised hotspot. This is because the multiple hotspots are introduced in a more distributed manner and so it does not take as long to spread the additional load throughout the application grid, as if the additional load is located in a single hotspot.

Using ELASTIC, experimental tests were also performed over a real agent-based MPI parallel application, which describes how an epidemic spreads throughout a population. In the functional behaviour of the agent-based application, the computation associated with an agent does not depend on the application task where it is performed. This gives rise to an "any-to-any" communication pattern, wherein each task may exchange information with any other task in each iteration. For the same reason, migration can be performed between any tasks in the application.

The agent-based application suffers from load imbalance due to the agent life cycle. To solve this problem, an ELASTIC package was developed according to the characteristics of the application. The load balancing algorithm implemented in this ELASTIC package takes advantage of the application's communication pattern to conduct an immediate distribution of the load between tasks, because the restriction of local communication does not exist. This behaviour permits ELASTIC to reach its full potential, achieving reductions of around 30% in the execution time when dynamically tuning this application. The full details of this experimentation can be found in [4].

The important outcome of the results presented in this section is that ELASTIC is able to manage the data generated during the monitoring process and use its tuning network to effectively detect load imbalances and correct them at runtime. From this, it can be concluded that ELASTIC is able to scale to dynamically tune large-scale parallel applications, its primary objective.

5.3. ELASTIC overhead

A primary consideration when designing ELASTIC was limiting the overhead that the dynamic tuning process introduces into the parallel application.

The overhead in ELASTIC comes from (1) inserting and removing code during the monitoring and tuning phases, (2) the execution of this code, and (3) the con-

Table 2
ELASTIC overhead measured in the synthetic application

Number of application tasks	With ELASTIC (s)	Without ELASTIC (s)	Overhead (%)
256	148.886	148.813	0.049
1,024	148.891	148.820	0.048
2,304	148.918	148.826	0.062
4,096	148.931	148.831	0.067
9,216	148.957	148.853	0.070
16,384	148.952	148.851	0.068

current operation of the EBE on the same core as the task that it controls.

Additional overhead may be introduced by user code required by the tuning process, such as the migration of work units between tasks in the synthetic application. This overhead does not come from ELASTIC, but from the specific tuning strategy being employed.

To measure the intrusion caused by ELASTIC when tuning the synthetic application, we have used the single localised hotspot scenario. For each application size, the synthetic application was executed with ELASTIC performing all analysis and tuning functions, but without the migration process. In this way, the application remained imbalanced, and the difference between this execution time and the execution time of the synthetic application without ELASTIC gives a measurement of the overhead introduced.

Table 2 shows the execution times for each application size executed with and without ELASTIC. The overhead is given as a percentage of the execution time without ELASTIC.

It can be seen that the overhead introduced by ELASTIC is only a fraction of a second, which represents less than 0.1% of the execution time for all application sizes.

We have shown that ELASTIC's strategy of performing all costly operations in a distributed manner without using application resources ensures that the intrusion required to perform dynamic tuning is kept to a minimum.

6. Related work

In the performance analysis area, there are a number of tools that perform dynamic tuning of parallel applications.

MATE [10] offers dynamic analysis and tuning of MPI parallel applications through three basic and continuous phases: monitoring, performance analysis and modification. All these steps are performed automatically, dynamically, and continuously throughout the application execution. MATE uses dynamic instrumentation to modify the application at runtime and its analysis process is based on performance models.

Autopilot [11] is an online tuning toolkit that allows for the adaptive control of applications and resource management policies on a wide area distributed systems. The Autopilot infrastructure includes distributed sensors for performance data acquisition, distributed actuators for implementing performance optimisation decisions, and a decision-making mechanism, based on fuzzy logic, for assimilation of sensor inputs and control of actuator outputs.

Active Harmony [14] allows dynamic adaptation of an application to the network and resource capacities of the execution environment. In this tool, the monitoring process gathers measurements from various libraries with the same functionality. Then, it uses heuristic techniques to explore the application optimisation space and finally chooses the best implementation among the libraries or the best combination of tuning parameters.

PerCo [7] is a framework for performance monitoring in heterogeneous environments. It is capable of monitoring the progress of the application's execution and redeploying it to optimise performance through process migration. To allow for the redeployment, the controlled application could be interrupted in one platform and restarted in another from the point of the interruption, using, for example, check-pointing files. The performance analysis and tuning process is performed using historical data, and combining time series and data adjustment methods.

Each of the previous tools employs different techniques to gather and analyse performance data, and use these data to make decisions in order to improve the performance of a parallel application. However, they all share a common trait, which is the existence of a centralised analysis component in their design. It is due to this fundamentally centralised scheme that none of these tools are able to scale to operate on large-scale parallel applications.

Currently, there are several automatic performance analysis tools that work on large-scale systems, such as Scalasca [9], TAU [13] and Periscope [1]. All of them implement some sort of decentralised mechanism. However, none of them, except for latest efforts in Periscope under the AutoTune Project [8], consider application tuning.

Having these facts in mind, in this work we address the lack of large-scale dynamic tuning in the current performance analysis area.

7. Conclusion

The principal objective of this work is to face the challenges posed by performing dynamic tuning on parallel applications composed of many thousands of tasks. This objective has been met in the form of ELASTIC. The scalability of ELASTIC arises from its architecture, structured as a hierarchal tree of nodes (the tuning network) whose topology can be adapted to accommodate the size of the parallel application.

ELASTIC operates following the closed tuning loop of automatic and continuous monitoring, analysis and tuning of a parallel application without stopping, re-compiling or re-running it. In the monitoring phase, ELASTIC uses event tracing to collect information about the application at the task level. This information is sent to the nodes of the tuning network where automatic performance analysis is conducted. After detecting a performance problem, tuning orders are inserted into the application tasks at runtime with the aim of improving its performance.

The knowledge required to guide the performance analysis and tuning process is integrated into ELASTIC in the form of ELASTIC packages. The authors of ELASTIC packages have access to a rich array of features via the *Tuning and Abstraction API*, which also defines the structure of the packages themselves. The resolution of different kinds of performance problems using ELASTIC is completely viable, owing to the plugin architecture granted by these ELASTIC packages.

ELASTIC was used to balance the load in a synthetic parallel application. In all the cases tested up to 16,384 tasks, ELASTIC improves the application execution time for 22%. These results highlight the viability of using ELASTIC for dynamic tuning in the large-scale computing area.

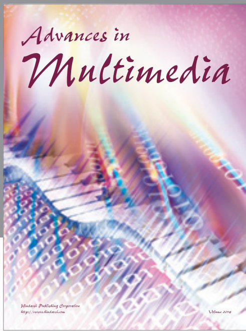
The most direct extension of this work is the creation of a library of generalised ELASTIC packages which model different performance issues. These ELASTIC packages would only need small modifications to be applied to specific parallel applications with the same performance problem.

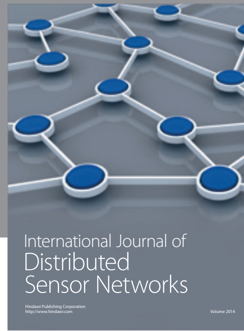
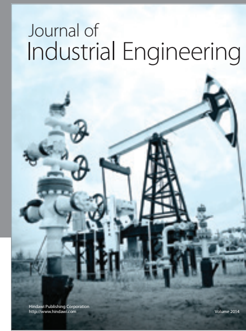
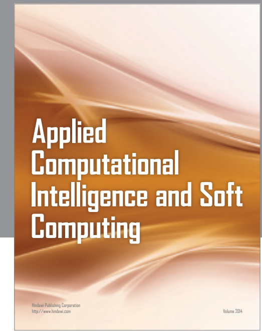
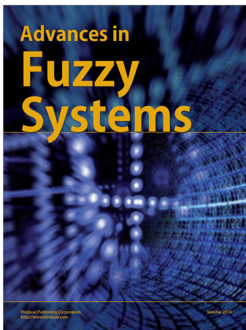
Acknowledgements

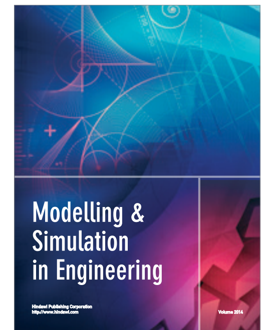
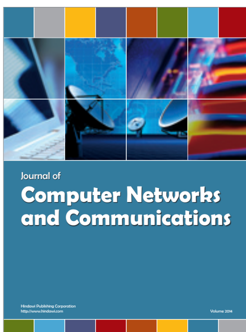
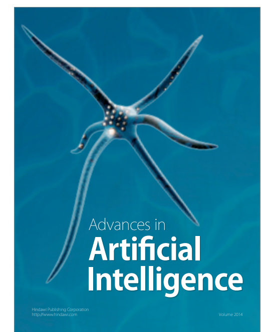
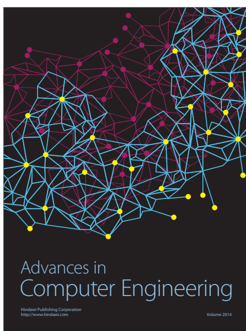
This research has been supported by the MICINN-Spain, under contract TIN2011-28689. The author thankfully acknowledges the computer resources, technical expertise and assistance provided by the Leibniz Supercomputing Centre.

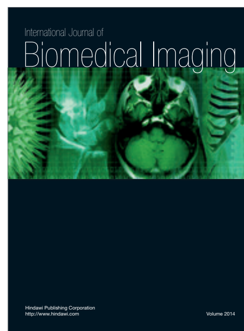
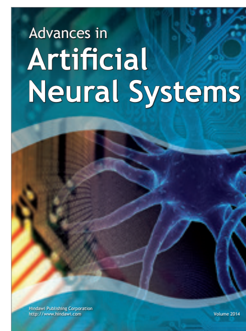
References

[1] S. Benedict, V. Petkov and M. Gerndt, PERISCOPE: An online-based distributed performance analysis tool, in: *Parallel Tools Workshop*, 2009, pp. 1–16.
[2] M.J. Brim, L. DeRose, B.P. Miller, R. Olichandran and P.C. Roth, MRNet: A scalable infrastructure for the development of parallel tools and applications, in: *Proceeding of Cray User Group 2010*, May 2010.
[3] B. Buck and J. Hollingsworth, An API for runtime code patching, *The International Journal of High Performance Computing Applications* **14**(4) (2000), 317–329.
[4] A. Martínez, Dynamic tuning for large-scale parallel applications, PhD dissertation, Universitat Autònoma de Barcelona, 2013.
[5] A. Martínez, A. Sikora, E. César and J. Sorribes, How to scale dynamic tuning to large-scale applications, in: *Proceedings of International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) – IPDPS*, 2013, pp. 355–364.
[6] A. Martínez, A. Sikora, E. César and J. Sorribes, How to determine the topology of hierarchical tuning networks for dynamic auto-tuning in large-scale systems, in: *Proceedings of International Workshop on Automatic Performance Tuning (IWAPT) – ICCS*, 2013, pp. 1352–1361.
[7] K. Mayes, M. Luján, G.D. Riley, J. Chin, P.V. Coveney and J.R. Gurd, Towards performance control on the grid, *Philosophical Transactions of the Royal Society: Series A* **363**(1833) (2005), 1975–1986.
[8] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin and F. Bodin, AutoTune: A plugin-driven approach to the automatic tuning of parallel applications, in: *Proceeding of the 11th Int. Workshop on the State-of-the-Art in Scientific and Parallel Computing (PARA 2012)*, Vol. 7782, 2012, pp. 328–342.
[9] B. Mohr, B. Wylie and F. Wolf, Performance measurement and analysis tools for extremely scalable systems, *Concurrency and Computation: Practice and Experience* **22** (2010), 2212–2229.
[10] A. Morajko, T. Margalef and E. Luque, Design and implementation of a dynamic tuning environment, *Journal of Parallel and Distributing Computing* **67** (2007), 474–490.
[11] R.L. Ribler, H. Simitci and D.A. Reed, The autopilot performance-directed adaptive control system, *Future Generation Computer Systems* **18**(1) (2001), 175–187.
[12] P.C. Roth, D.C. Arnold and B.P. Miller, MRNet: A software-based multicast/reduction network for scalable tools, in: *Proceeding of the ACM/IEEE Conference on Supercomputing*, 2003, p. 21.
[13] S.S. Shende and A.D. Malony, The TAU parallel performance system, *International Journal of High Performance Computing Applications* **20**(2) (2006), 287–311.
[14] C. Tapus, I.-H. Chung and J. Hollingsworth, Active harmony: Towards automated performance tuning, in: *Proceeding of the Conference on High Performance Networking and Computing*, 2003, pp. 1–11.

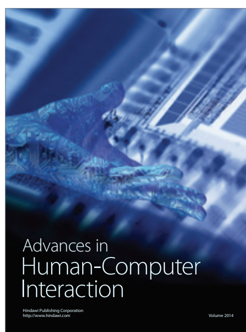
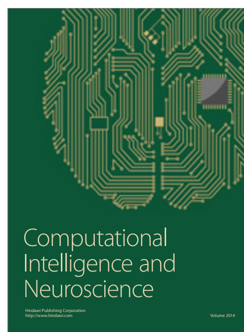

Advances in
Multimedia

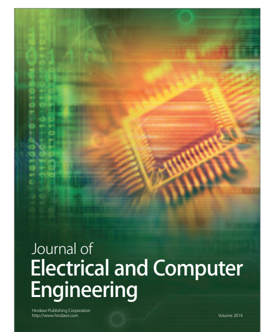
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering