Universitat Autònoma de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Dynamic Tuning for Large-Scale Parallel Applications

Thesis submitted by **Andrea Martínez Trujillo** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Sikora and Dr. Joan Sorribes Gomis

Barcelona, June 2013

Dynamic Tuning for Large-Scale Parallel Applications

Thesis submitted by **Andrea Martínez Trujillo** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Sikora and Dr. Joan Sorribes Gomis, at the Computer Architecture and Operating Systems Department.

Barcelona, June 2013

Author

Andrea Martínez Trujillo

Supervisors

Dr. Anna Sikora

Dr. Joan Sorribes Gomis

Abstract

The current large-scale computing era is characterised by parallel applications running on many thousands of cores. However, the performance obtained when executing these applications is not always what it is expected. Dynamic tuning is a powerful technique which can be used to reduce the gap between real and expected performance of parallel applications. Currently, the majority of the approaches that offer dynamic tuning follow a centralised scheme, where a single analysis module, responsible for controlling the entire parallel application, can become a bottleneck in large-scale contexts.

The main contribution of this thesis is a novel model that enables decentralised dynamic tuning of large-scale parallel applications. Application decomposition and an abstraction mechanism are the two key concepts which support this model. The decomposition allows a parallel application to be divided into disjoint subsets of tasks which are analysed and tuned separately. Meanwhile, the abstraction mechanism permits these subsets to be viewed as a single virtual application so that global performance improvements can be achieved.

A hierarchical tuning network of distributed analysis modules fits the design of this model. The topology of this tuning network can be configured to accommodate the size of the parallel application and the complexity of the tuning strategy being employed. It is from this adaptability that the model's scalability arises. To fully exploit this adaptable topology, in this work a method is proposed which calculates tuning network topologies composed of the minimum number of analysis modules required to provide effective dynamic tuning.

The proposed model has been implemented in the form of ELASTIC, an environment for large-scale dynamic tuning. ELASTIC presents a plugin architecture, which allows different performance analysis and tuning strategies to be applied. Using ELASTIC, experimental evaluation has been carried out on a synthetic and a real parallel application. The results show that the proposed model, embodied in ELASTIC, is able to not only scale to meet the demands of dynamic tuning over thousands of processes, but is also able to effectively improve the performance of these applications.

Keywords: performance analysis; dynamic tuning; scalability; performance tools; tuning network.

Resumen

La era actual de computación a gran escala se caracteriza por el uso de aplicaciones paralelas ejecutadas en miles de cores. Sin embargo, el rendimiento obtenido al ejecutar estas aplicaciones no siempre es el esperado. La sintonización dinámica es una potente técnica que puede ser usada para reducir la diferencia entre el rendimiento real y el esperado en aplicaciones paralelas. Actualmente, la mayoría de las aproximaciones que ofrecen sintonización dinámica siguen una estructura centralizada, donde un único módulo de análisis, responsable de controlar toda la aplicación paralela, puede convertirse en un cuello de botella en entornos a gran escala.

La principal contribución de esta tesis es la creación de un modelo novedoso que permite la sintonización dinámica descentralizada de aplicaciones paralelas a gran escala. Dicho modelo se apoya en dos conceptos principales: la descomposición de la aplicación y un mecanismo de abstracción. Mediante la descomposición, la aplicación paralela es dividida en subconjuntos disjuntos de tareas, los cuales son analizados y sintonizados separadamente. Mientras que el mecanismo de abstracción permite que estos subconjuntos sean vistos como una única aplicación virtual y, de esta manera, se puedan conseguir mejoras de rendimiento globales.

Este modelo se diseña como una red jerárquica de sintonización formada por módulos de análisis distribuidos. La topología de la red de sintonización se puede configurar para acomodarse al tamaño de la aplicación paralela y la complejidad de la estrategia de sintonización empleada. De esta adaptabilidad surge la escalabilidad del modelo. Para aprovechar la adaptabilidad de la topología, en este trabajo se propone un método que calcula topologías de redes de sintonización compuestas por el mínimo número de módulos de análisis necesarios para proporcionar sintonización dinámica de forma efectiva.

El modelo propuesto ha sido implementado como una herramienta para sintonización dinámica a gran escala llamada ELASTIC. Esta herramienta presenta una arquitectura basada en *plugins* y permite aplicar distintas técnicas de análisis y sintonización. Empleando ELASTIC, se ha llevado a cabo una evaluación experimental sobre una aplicación sintética y una aplicación real. Los resultados muestran que el modelo propuesto, implementado en ELASTIC, es capaz de escalar para cumplir los requerimientos de sintonizar dinámicamente miles de procesos y, además, mejorar el rendimiento de esas aplicaciones.

Palabras clave: análisis de rendimiento; sintonización dinámica; escalabilidad; herramientas de rendimiento; red de sintonización.

Resum

L'era actual de la computació a gran escala es caracteritza per l'ús d'aplicacions paral·leles executades en milers de cores. No obstant aix, el rendiment obtingut en executar aquestes aplicacions no sempre és l'esperat. La sintonització dinàmica és una potent tècnica que pot ser usada per reduir la diferència entre el rendiment real i l'esperat en aplicacions paral·leles. Actualment, la majoria de les aproximacions que ofereixen sintonització dinàmica segueix una estructura centralitzada, on un únic mòdul d'anàlisi, responsable de controlar tota la aplicació parallela, pot esdevenir un coll d'ampolla en entorns a gran escala.

La principal contribució d'aquesta tesi és la creació d'un model nou que permet la sintonització dinàmica descentralitzada d'aplicacions paral·leles a gran escala. Aquest model es basa en dos conceptes principals: la descomposició de laplicació i un mecanisme d'abstracció. Mitjanant la descomposició, l'aplicació parallela es divideix en subconjunts disjunts de tasques, les quals són analitzades i sintonitzades separadament. Mentre que el mecanisme d'abstracció permet que aquests subconjunts siguin vistos com una única aplicació virtual i, d'aquesta manera, es poden aconseguir millores globals de rendiment.

Aquest model es dissenya com una xarxa jeràrquica de sintonització formada per mòduls d'anàlisi distribuïts. La topologia de la xarxa de sintonització es pot configurar per ajustar-se a la mida de l'aplicació parallela i a la complexitat de l'estratègia de sintonització utilitzada. D'aquesta adaptabilitat sorgeix l'escalabilitat del model. Per aprofitar l'adaptabilitat de la topologia, en aquest treball es proposa un mètode que calcula topologies de xarxes de sintonització compostes pel mínim nombre de mòduls d'anàlisi necessaris per proporcionar la sintonització dinàmica d'una forma eficient.

El model proposat ha estat implementat com un entorn per a la sintonització dinàmica a gran escala, anomenat ELASTIC. Aquesta eina presenta una arquitectura basada en plugins i permet aplicar diferents tècniques d'anàlisi i de sintonització. Emprant ELASTIC, s'ha dut a terme una avaluació experimental sobre una aplicació sintètica i una aplicació real. Els resultats mostren que el model proposat, implementat en ELASTIC, és capaç d'escalar per acomplir els requeriments de sintonitzar dinàmicament milers de processos i, a més, millorar el rendiment d'aquestes aplicacions.

Paraules clau: anàlisi de rendiment; sintonització dinàmica; escalabilitat; eines de rendiment; xarxa de sintonització.

Acknowledgements

Primeramente, quiero darles las gracias a mis padres, Manolo y Mª Carmen. Gracias por apoyarme incondicionalmente durante estos últimos 4 años, por vuestras continuas muestras de ánimo y por quererme de esa manera tan especial. ¡Gracias por ser mis padres! También quiero dar especialmente las gracias a mis hermanas, Sonia y Mamen, a mis cuñados, Toni y Jorge, y a mis sobrinos, Pablo, Javier y Bruno. Gracias por animarme a luchar por conseguir mis metas y por la confianza depositada en mi. Además, gracias por hacerme sentir que, aunque nos separan cientos de kilómetros, siempre estaréis ahí para lo que necesite. Familia, si he llegado hasta aquí, ha sido gracias a vosotros. ¡Os quiero!

I want to thank to my research group. First of all, my thanks go to Ania and Joan, my advisors. Thank-you for giving me so much of your time over the past four years, for sharing your knowledge with me, for your suggestions and advice. I will never forget our discussions full of ideas. Ania, thank-you for being more than my advisor, and Joan, thank-you for encouraging me to be more positive. To Eduardo, who is for me, the third advisor of this thesis, thanks for always being willing to listen to me and trying to give me a solution when I interrupted you in your office with a *huge* problem. To Tomàs, for always providing me with a different point of view in our group meetings.

I would like to thank Prof. Michael Gerndt for his hospitality during my stay in Munich and for his interest and ideas about my work. I would also like to express my thanks to Prof. Barton Miller for his expert advice and to his MRNet team, especially to Michael Brim for answering many, many questions.

I also want to thank all the members of "CAOS". Special thanks go to Lola and Emilio for always helping me to enjoy research. Thanks to the PT guys, Dani and Xavi, and to Gemma, for their patience and disposition. Thanks to Moni, Maru, Alvaro, Zeynep, Gonza, Claudia, Ronald, Javier, Leo, Pili, Alvaro W., Claudio, Abel, Tharso, Joao, Sandra and Carlitos for their mutual support and care during this experience.

I want to thank to my five favourite *informáticas*, María, Conchi, Inma, Sandra y Mª Dolores, you are part of this work.

Very special thanks to Thea, for her valuable *aussie* support reviewing *my book*, and to Bruce for the limoncello.

Finally, my deepest thanks go to Hayden. I don't have the words to explain what you have meant to me in these last few years. You have taught me what it really means to love someone. Thank you for your understanding every day, your care, your constant wonderful smile, your hugs and for all the things that we share. This work would not be what it is without you. Thank-you, thank-you, thank-you!

A mis padres.

A Hayden.

A mi misma.

Contents

List of Figures

List of Tables

List of Algorithms

List of Equations

1

Introduction

"Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop."

– Lewis Carroll, *Alice in Wonderland*

In this chapter, we present a general overview of high performance computing. In particular, our work is focused on dynamic performance tuning for large-scale parallel applications. This chapter introduces the motivations inspiring this work, and details what its goals and contributions are. Finally, we present the organisation of this thesis.

1.1 High Performance Scientific Computing

Scientific computing is the interdisciplinary field located at the intersection of modelling scientific problems, and the use of computational resources to produce quantitative results from this models. In the current large-scale era, scientific computing has become high performance scientific computing, characterised by the use of supercomputer systems as computational resource.

Supercomputers are extremely powerful computers that offer massive parallelism and high capacity of processing, due to the combined power of hundred of thousands of individual processing units. This is combined with a large storage capacity and a fast and reliable interconnection network. Nowadays, supercomputers also exploit the development in computer technology integrating the use of accelerator blades, FPGAs (field-programmable gate array) or GPUs (graphics processing units) providing the high speed calculations important for scientific computation. Therefore, supercomputers are designed to work at high speed, handling huge amounts of data and performing multiple complex operations at the same time.

Supported by high performance systems, large-scale parallel applications can provide knowledge about many problems in intense research areas of science, engineering, industry and commerce. Some well-known examples of these applications are the determination of the human genome [47], climate study [54], simulation of molecular dynamics [18] or web search engines [3].

Unfortunately, it is difficult to develop parallel applications that can consistently exploit the capabilities of modern supercomputers. A great part of this is due to the complexity of the underlying hardware in these kinds of systems in terms of number of cores, various levels of communications paths, or even the presence of heterogeneous components. Moreover, a parallel application specifically designed to take advantage of the resources of one machine may present poor performance on another machine.

In this context, the ever widening gap between the theoretical and practical performance of parallel applications running on supercomputers makes tools that effectively identify, understand and fix performance problems more valuable than ever. These tools alleviate some of the burden of determining what is required to improve the performance of an application.

However, when working with truly large-scale applications, performance analysis tools can manifest scalability issues of their own. To avoid becoming a bottleneck and remain effective, these tools must follow a scalable and modular design that enables the control and analysis of an extremely large number of tasks.

This thesis is focused on ***providing an approach for scalable and dynamic performance tuning of parallel applications running on tens of thousands of processors***.

In order to provide a holistic vision of the scope of the thesis in this chapter, the following subsections present a short introduction about performance analysis and tuning of parallel applications and scalability in analysis tools, the pillars of this work.

1.1.1 Performance Analysis and Tuning

The development of efficient parallel applications is a challenging task that requires a high degree of expertise. Usually, when running a parallel application, performance problems appear that limit its efficiency, especially as the number of tasks involved increases. In large-scale parallel applications, there are several causes for performance degradation such as non-scalability, communication imbalance, load imbalance, cache misses, late sender and synchronisation overhead.

To reduce the effect of performance problems during the application execution, it is often necessary that, after the development of the application, the programmer or user carries out a performance improvement process. The performance improvement process includes three successive phases [24]. Firstly, behavioural information about the parallel application is gathered (***monitoring phase***). Then, through the analysis of the collected information, performance bottlenecks are detected and possible actions to overcome them are determined (***analysis phase***). Finally, the chosen changes are applied to the application code with the objective of resolving the problems and improving performance (***tuning phase***).

In recent decades, different approaches and tools have been developed with the aim of helping the programmer during the performance improvement process. These tools have been designed following different perspectives of performance analysis and improvements.

Some of these tools help users to find performance problems through the graphic visualisation of the behaviour of the parallel application using the performance data previously collected (*classical/static performance analysis approach*) [26]. Once the parallel application has finished its execution, other tools are able to automatically detect performance problems and provide the programmer with suggestions (on how) to improve performance (*automatic performance analysis approach*) [38][52]. Finally, there are tools that offer the capacity to dynamically find performance bottlenecks [4][37] and/or improve the performance of the parallel application [39][56] while it is running, without involving the programmer in this process (*dynamic performance analysis and/or tuning approach*).

This thesis is specifically centred on **automatic and dynamic performance analysis and tuning**. In this approach the three phases of the performance improvement process (monitoring, analysis and tuning) are performed automatically and continuously while the parallel application is running. The analysis phase uses performance measurements directly provided by the dynamic monitoring phase. Depending on the evaluation of the performance, the tuning actions are automatically and dynamically inserted into the application.

Therefore, dynamic tuning of a parallel application should provide the following aspects: 1) dynamic monitoring of its execution, 2) automatic performance evaluation, and 3) automatic tuning of the application while it is running. These three aspects permit dynamic tuning that exempts programmers and users from modifying the execution of the application manually, because the whole performance improvement process is carried out automatically.

Dynamic tuning permits the evaluation and adaptation of the parallel application according to the current execution's conditions. For this reason, this approach is the most suitable for improving the performance of parallel applications whose performance behaviour depends on input data, may change during each execution, or when the application is being executed in heterogeneous or time-sharing systems.

1.1.2 Scalability of Performance Analysis Tools

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. For this reason, the available processing time in these machines should be used as efficiently as possible.

Normally, parallel applications running on supercomputers do not make efficient use of resources. This translates into a longer than expected running time, which "wastes" computation hours and reduces the available allocated time for further executions. By improving performance, and thus reducing execution time of parallel applications, performance analysis tools can help to reduce the amount of time required on such large-scale systems.

Traditionally, performance analysis tools operate using a single centralised module where the analysis process and global tool control is performed. The collection of performance data and task control occur in the tool's daemons which usually are running on the nodes of the parallel system. The centralised module is responsible for communicating with all these tool's daemons to collect performance information from the application.

When working with large-scale parallel applications, a scalability barrier arises from this centralised operation, which becomes a bottleneck due to the large number of communication connections to be controlled and the increasing complexity of conducting a holistic performance analysis.

To apply performance analysis to parallel applications executed on supercomputers, it is paramount that the analysis tools have been specifically designed to operate in such environments. This design must focus on resolving scalability problems that can reduce the effectiveness of performance analysis tools. There are three primary scalability issues which must be overcome:

- How to manage the large volume of performance data generated when a large number of tasks are monitored.
- How to efficiently handle the communication channels with a large number of the tool's distributed daemons.
- How to conduct an effective performance analysis of thousands of parallel application tasks, avoiding that the complexity of the performance evaluation may increase the tool's time response.

Currently, there are several analysis tools which conduct an automatic performance analysis and are able to operate on large-scale systems. Well known examples are Paradyn [50], Scalasca [38], TAU [52], and Periscope [4]. All of them implement some sort of decentralised design, many of them hierarchical and in some cases using frameworks that enable an efficient management of data and communication. However, none of these tools, except for latest efforts in Periscope, consider application tuning.

Taking into consideration the scalability issues mentioned and insight gained from studying these performance analysis tools, in this thesis we concentrate on the design of an approach that allows performance analysis tools to provide scalable dynamic tuning to overcome performance problems in large-scale applications. In this manner, we strive to address the lack of large-scale dynamic tuning in the current performance analysis area.

1.2 Motivation

The current trend in supercomputers is to provide machines with more and more processors, with the aim of increasing computing power while maintaining energy consumption at minimal levels. It is now common to see supercomputers with tens or hundreds of

thousands of processors, which can perform multiple Petaflops (quadrillions of calculations per second). These kinds of machines have proven to be a fundamental resource in modern science.

Parallel applications running on supercomputers are able to calculate the results of extremely complex scientific models in a relatively short amount of time. Unfortunately, it is common that the performance expected of these large-scale parallel applications is not easily achieved in high performance machines. Several performance analysis tools are able to assist developers in analysing and improving the performance of these parallel application in large-scale contexts. Nevertheless, most of these analysis tools are less useful when applications have exceptionally long running times or behavioural patterns that change depending on the input data set or according to data evolution.

In this context, performance analysis tools based on the concept of automatic and dynamic tuning are necessary. These tools are able to perform automatic monitoring and analysis of the application, and then dynamically resolve performance issues during the same execution, without recompiling or restarting. As a counterpoint, developing such tools which can effectively operate over large-scale parallel applications is a challenging task.

If dynamic tuning is to continue to be a valid solution, then the scalability of dynamic tuning tools must keep up with the ever-increasing scale of modern supercomputers.

1.3 Objectives

The ultimate goal of this thesis is to design, implement and evaluate an approach that provides dynamic performance tuning for large-scale parallel applications.

We aim to address this problem by replacing the centralised analysis and tuning concept with a distributed one. This distributed approach should be capable of providing a coherent tuning environment which is able to scale to perform runtime analysis and tuning of parallel applications composed of tens of thousands of processes. Considering a parallel application running on a large-scale system, the tuning environment should improve its performance, detect existing bottlenecks and modify the application to resolve them without recompiling or restarting the application. The specific objectives of this work are as follows:

- Conduct a study of how current performance analysis tools are able to scale to work with large-scale parallel applications.

- Design a model that enables the distribution of the dynamic performance improvement process.
- Validate the scalability of the model for distributed dynamic tuning.
- Implement the first version of a tool that offers scalable dynamic tuning following the proposed model.
- Evaluate the viability of the distributed dynamic tuning model by means of experimental evaluation of the implemented tool to tune synthetic and real applications.

1.4 Contribution

The contributions in this work are focused on achieving the ultimate goal presented in the previous section. To this end, we have designed and implemented an approach for dynamic performance tuning of large-scale parallel applications. This thesis presents the following specific contributions:

- A model for scalable dynamic tuning. The model follows a decentralised scheme, that allows hierarchical performance analysis and tuning, based on application decomposition and an abstraction mechanism. Through the decomposition, parallel applications which are too large to be analysed and tuned in a centralised manner are decomposed into disjoint subsets of tasks, which can be managed separately. Meanwhile, the abstraction mechanism permits these subsets to be represented as a single virtual application such that global performance improvements can be obtained [31][32].

  To perform dynamic tuning, the model employs a collaborative approach where user knowledge is required in the form of an analytical performance model. Additionally, an abstraction model is necessary to guide the abstraction mechanism.

  The design of the model can be translated into a hierarchical tuning network of distributed analysis modules. The main characteristic of this network is that it presents a topology which is adaptable to the size of the parallel application being analysed and the complexity of the tuning strategy being employed.
- An explicit method for calculating an efficient topology for the proposed hierarchical tuning network. The method provides the minimum number of analysis modules needed to carry out a dynamic and automatic analysis and tuning process, considering specific parameters that characterise the behaviour of the tuning network and the application being analysed [33].
- ELASTIC, a tool that provides scalable dynamic tuning. This tool implements our model for scalable dynamic tuning. ELASTIC makes use of the framework MRNet [51] to establish the hierarchical communication in the tuning network and DynInst

[10] to perform dynamic instrumentation required for monitoring and tuning the parallel application at runtime.

- ELASTIC Packages definition. The knowledge required to carry out the performance analysis and tuning process, is codified in ELASTIC Packages. This allows for a generalised use of ELASTIC to resolve different performance issues by simply exchanging the ELASTIC Package, offering a pluggable design. To demonstrate this capability, we have developed two different packages which resolve performance problems in a synthetic application and a real agent-based application.

1.5 Thesis Organisation

The work presented in this thesis is divided into the following chapters.

Chapter 2: Related Work. This chapter introduces the main concepts that serve as the pillars of this work, *dynamic performance tuning* and *scalability in performance analysis*. An exhaustive review of performance tools in both areas is presented.

Chapter 3: Scalable Dynamic Tuning. This chapter explains in detail the proposed model for scalable dynamic performance tuning and its design. It also includes a method for calculating the minimum number of resources required by the hierarchical tuning network. Finally, the chapter concludes with an approximation of the time required to achieve global performance improvements considering the specifications of the proposed model.

Chapter 4: Model Scalability Validation. In this chapter, a validation of the scalability of the proposed model for dynamic tuning is given. The simulation environment developed to perform this evaluation is described. As a prior step required for the scalability validation, a study that verifies the correctness of the topology calculation method, presented in the previous chapter, is also detailed.

Chapter 5: ELASTIC. In this chapter, ELASTIC, the tool that implements the model presented in Chapter 3 is introduced. Details of its design and implementation are revealed. We describes all its modules and the APIs used for communication between them.

Chapter 6: Experimental Evaluation. This chapter details the experimental evaluation carried out on ELASTIC with two test cases, a synthetic SPMD application and a real agent-based application. It goes on to describe the process involved in the creation of the ELASTIC Packages used to tune each of these applications, and the experimental results that verify the scalability and effectiveness of the proposed model for scalable dynamic tuning implemented in ELASTIC.

Chapter 7: Conclusions. Finally, with this chapter we review the major details of the presented work and conclude this thesis. This is followed by an outline of open problems and a discussion of possible directions for future work.

2

Related Work

"Every new beginning comes from some other beginning's end."

– Seneca

This chapter presents a description of work related to the two pillars of this thesis: dynamic tuning and scalability in performance analysis tools. The characteristics of dynamic tuning are detailed, as well as a description of the current tools which implement this approach. We also describe how existing performance analysis tools are able to work in large-scale systems and analyse parallel application running on many thousands of cores.

2.1 Introduction

In the current era of large-scale computing, there are two principal challenges which must be solved simultaneously in order to make the current generation of performance analysis and tuning tools effective.

The first of these challenges consists of being able to develop tools that dynamically discover and tune performance problems for applications with heterogeneous behaviour, for which post-mortem analysis and tuning is not effective.

The second problem to consider is the scalability of the analysis tools. When working with truly large-scale applications, these tools can begin to present scalability issues of their own. Effective performance analysis tools must be designed in such a way that they do not become a bottleneck themselves.

The objective of this research is to provide an approach for scalable dynamic tuning of parallel applications. Our approach combines the potential of dynamic tuning with strategies to overcome the scalability barriers that analysis tools present nowadays.

In this chapter, we present the background concepts of this thesis related to dynamic tuning and scalability in performance analysis. As such, this chapter is divided into two sections. In Section 2.2, the concept of dynamic tuning is described. We also discuss a number of tools that implement different strategies in order to perform analysis and tuning of parallel applications at runtime. In Section 2.3 we survey existing analysis tools that address the scalability barriers presented in a large-scale context. We especially focus on the mechanisms that they employ in order to manage the various aspects that arise when analysing parallel applications running on many thousands of processors.

2.2 Dynamic Tuning

The performance improvement process of a parallel application, also called the tuning process, is the process followed in order to fix performance problems in the application, improving its behaviour through the modification of its critical parameters.

The tuning process involves several phases. First, during the *monitoring phase*, information about the behaviour of the application is collected. In the next phase, the analysis of the gathered information is conducted. The *analysis phase* finds performance bottlenecks, determines their causes and decides which actions have to be taken to eliminate such bottlenecks. Finally, in the *tuning phase*, appropriate changes are applied to the application code to overcome problems and improve the performance.

The model of dynamic tuning proposes that all the phases implicated in the performance improvement process are conducted continuously during the application execution (the closed tuning loop), as shown in Figure 2.1. Each of the phases requires particular adaptations:

- In the monitoring phase, the data containing information about the behaviour of the application has to be collected during runtime and passed directly to the analysis phase. This reduces the cost implied by storing the monitoring data generated throughout the entire application execution. To collect this information, the application must be instrumented, which consists of inserting code into the application at all the points that need to be monitored. This can be performed by the user manually or by linking with a monitoring library prior to application execution. An alternative technique allows instrumentation to be deferred until the application is running. An advantage of dynamic instrumentation technique is that it can be altered or removed during the application execution.
- The analysis phase has to be performed automatically during the application execution. To conduct an effective analysis it would be beneficial to integrate specific knowledge into the analysis process about the application and how to detect and overcome its bottlenecks.
- Once a solution has been provided by the analysis phase, the tuning phase is responsible for modifying the application to improve its performance. To perform the application modifications during its runtime, a dynamic instrumentation technique is required. This involves changing specific parameters that affect the performance of the application.

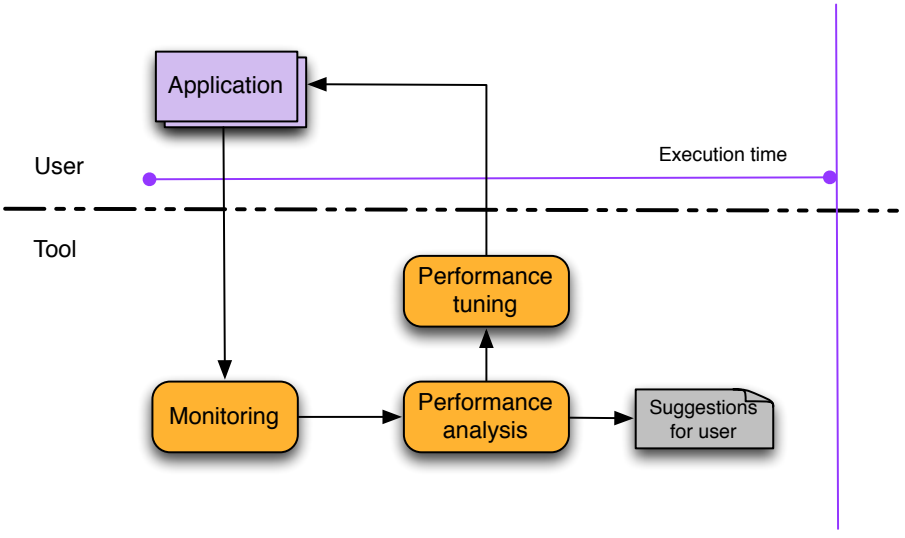


FIGURE 2.1: General model of the dynamic tuning approach.

Dynamic tuning is not only useful, but necessary, when applications present behaviour that varies greatly according to the input data or during runtime. This varying behaviour can be due to the application itself, or because it is being executed in heterogeneous or time-sharing systems. In these cases, the results obtained from a post-mortem analysis are not necessarily useful to improve the performance of subsequent executions.

Consequently, this type of performance analysis relieves the programmer to modify the execution of the application manually, because the whole process is carried out automatically by the analysis tool.

Currently, a number of performance tools exist that implement this approach in an automatic manner such as MATE [39], Active Harmony [56], Autopilot [48], PerCo [34] and the CPO Paradigm [12]. These tools can help developers and users of parallel applications by reducing or removing the difficult tasks involved in manual performance tuning.

Due to the influence that MATE had in the development of this work, we dedicate the next section to describing its design philosophies and implementation. In the section that follows, we detail the important aspects of the primary tools that implement a dynamic tuning approach.

2.2.1 MATE

MATE (Monitoring, Analysis and Tuning Environment) is a tool that performs dynamic and automatic tuning of MPI parallel applications. Its objective is to improve the performance of a parallel application at run-time, by adapting it to the variable conditions of the system. First, at run-time MATE instruments the application to gather information about its behaviour. During the analysis phase, MATE receives this information in the form of events, searches for bottlenecks and specifies solutions for solving the performance problems encountered. Finally, the application is dynamically modified by applying the given solutions. MATE uses dynamic instrumentation [10] to modify the application at run-time, so it does not need to be recompiled or restarted.

MATE is composed of the following modules which cooperate to control and improve the application's performance [41]:

- The **Application Controller** (AC) is a daemon that controls the execution and the dynamic instrumentation of each individual MPI task.
- The **Analyzer** is a centralised process that carries out the application performance analysis, and decides on monitoring and tuning. It automatically detects existing performance problems on the fly and requests appropriate changes to improve the application's performance.

- The **Dynamic Monitoring Library** (DMLib) is a shared library that is dynamically loaded by the AC in the application tasks to facilitate collecting data and delivering it to the Analyzer.

The knowledge required to perform analysis and tuning is encapsulated in MATE in a piece of software called a *tunlet*. Each tunlet implements the logic to overcome a particular performance problem by encapsulating knowledge about it in several terms that define the information required for the monitoring, analysis and tuning phases. These terms are:

- Measurement points, which indicate *what* is needed to detect the performance problem. Specifically, they are the places in the application code where the instrumentation must be inserted to gather information about the application's performance.
- Performance model, a set of expressions that model the application's behaviour and determine how to evaluate the collected information in order to detect bottlenecks.
- Tuning actions, which indicate *what*, *where* and *when* to change in the application execution in order to fix the detected bottleneck.

MATE has been demonstrated to be an effective and feasible tool to improve performance of real-world applications running on small size clusters [40]. However, scalability issues appear when running MATE on hundreds of processors. In this context, the centralised analysis becomes a scalability bottleneck because of the following factors:

- The volume of events to be processed by the Analyzer, and the number of connections and AC daemons that have to be managed, increase the tool's response time.
- The centralised performance analysis uses performance models for analysis and tuning. Although the models are quite simple, usually the complexity of the performance model's evaluation depends on the number of processes involved in the analysis phase. This fact limits the scalability properties of the centralised Analyzer.

An initial approach presented in [13] attempted to alleviate the identified scalability barriers of MATE. This approach is based on the distributed collection of events which reduces the workload of the original centralised manner in which such collection was carried out, and in the preprocessing of cumulative or comparative operations if possible. The experiments presented in this work were only conducted over 32 processors. The scalability properties of this approach are somewhat limited, as the entire analysis process must still be carried out in a centralised manner.

The scalability barriers found in MATE were the primary motivation that led to the development of the scalable dynamic tuning approach, presented in this thesis. For this reason, the proposed approach shares some characteristics with MATE. The proposed scalable dynamic tuning employs the same terms as MATE (measurement points, performance expressions, and tuning points and actions) to define the required knowledge to dynamically guide the analysis and tuning process. Moreover, the analysis and tuning modules of the hierarchical tuning network follow the same operation pattern as MATE's Analyzer, based on the three continuous phases of the closed tuning loop: monitoring, performance analysis and modifications.

The approach for scalable dynamic tuning presented in this thesis employs a more aggressive distribution in the performance analysis and tuning process than the initial approach that attempts to overcome the scalability barriers of MATE. We allow for the distribution, not only of the collection of performance information about the application, but also the analysis and tuning phases. Such a distribution leads to the ability to conduct dynamic tuning over larger-scale parallel applications than merely distributing the data collection process.

MATE assumes that the performance analysis, based on the global application view, is taking into consideration all the processes and their interactions. Such an approach is only feasible for environments with a relatively small number of nodes.

As well as offering a centralised analysis when the size of the application permits, our approach is able to resolve performance problems at much larger scales by distributing the analysis process throughout the tuning network. As a consequence of the decentralised process, the global view of the application is also distributed. This, in turn, implies that the problem being resolved can either be decomposed into smaller problems which can be resolved independently or hierarchically, or that the information required for global analysis can be abstracted effectively. The process of decomposition and abstraction is made possible by the abstraction mechanism presented in this thesis as part of the proposed hierarchical tuning model.

2.2.2 Other Dynamic Tuning Tools

Active Harmony

Active Harmony is an automated performance tuning infrastructure based on the closed loop of dynamic tuning. Specifically, this project is focused on the dynamic accommodation of the parallel application to the network and resource capacities of the execution

environment. This is achieved by automatic switching of algorithms and tuning of application libraries and parameters. The application must be Harmony-aware, that is, it must use the API provided by the system.

Active Harmony's architecture is based on a client-server model. The client is the "harmonised" parallel application, which sends performance information to the server. Using the collected information, the Harmony server carries out the tuning of the parallel application. The major components of the Active Harmony System are the Library Specification Layer and the Adaptation Controller.

The Library Specification Layer provides a uniform API, which integrates different libraries with the same or similar functionality. Using this API, the user develops an application and hence the application contains a set of libraries with different algorithms and tunable parameters to be changed. The goal of the Library Specification Layer is to help the application select the most appropriate underlying algorithm. During the execution of the parallel application, the Library Specification Layer collects information about performance metrics of the underlying libraries' execution.

The Library Specification Layer will send these measurements to the Adaptation Controller. Based on the observed performance, the Adaptation Controller selects the most appropriate library and changes tunable parameters to improve the application performance. Active Harmony automatically determines appropriate values for tunable parameters by searching the parameter value space using heuristic optimisation algorithms. In the last works with Active Harmony, the optimisation algorithm is based on the Parallel Rank Order algorithm proposed by Tabatabaee et al. [58].

The latest efforts of the Active Harmony project are focused in the field of online tuning of automatically generated code [57]. In these works, the Harmony system substitutes its normal collection of underlying libraries for different code variants. These variants are generated and compiled into a shared library by a standalone code-generation utility called CHiLL [15]. This new code is loaded and executed by the application being tuned. Following the initial philosophy of Active Harmony, the performance values obtained from these executions are utilised by the Adaptation Controller in order to discover the best configuration of tuning parameters and improve the performance of the application.

Active Harmony differs from the approach for dynamic tuning presented in this thesis in that the tuning of the parallel application is restricted to the functionality which it uses from the Library Specification Layer API. Therefore, the application must be implemented using this API. In our approach, dynamic instrumentation is used to monitor and tune the application, so any part of the application can be dynamically monitored or tuned. Related to the way in which the performance analysis is conducted,

as was mentioned, Active Harmony employs heuristic techniques, whereas our approach is based on analytical performance models in the form of a set of rules or algorithms.

Autopilot

Autopilot is an online tuning toolkit which bases its action on a closed loop control that allows for the adaptive control of applications and resource management policies on both parallel and 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 for assimilation of sensor inputs and control of actuator outputs.

Sensors are responsible for gathering performance data from the parallel application. During the monitoring phase sensors can gather data using two methods: non-threaded and threaded. Using the non-thread mode, a sensor records data in response to procedure calls that have been inserted into the application manually by the programmer. In the thread mode, a separate thread periodically awakes, reads application variables and returns to sleep. To allow the reduction of data, sensors support attached functions. These functions are invoked each time a sensor receives data, acting as a data filter, transforming the original data to an alternate reduced version.

Actuators are remotely controlled functions that can change local variable values and invoke local functions. Using actuators, a remote process can change the behaviour of an instrumented application. Such actuators can change, for example, parameter values or resource management policies (e.g. file caching policy).

The Autopilot decision infrastructure is based on a fuzzy logic engine that exploits real-time sensor inputs to dynamically select resource management policies. The fuzzy engine employs fuzzy sets to represent the semantic properties of each input (sensor) and output (actuator). Then a set of IF-THEN rules are used to map the input values to the output space. Depending of the result, specific actuators are activated to adapt the performance of the parallel application.

Moreover, Autopilot also provides mechanisms to manage local and remote tasks. The toolkit contains a sensor / actuator manager and set of remote clients. The manager serves as a network distributed name server and supports registration by remote sensors and actuators. A client controls both sensors and actuators in associated tasks, receiving data from sensors, and invoking actuators.

In Autopilot, the developer must prepare the application inserting sensors and actuators manually into the source code prior to the execution of the parallel application.

This technique is different to the monitoring process proposed in our approach, where dynamic instrumentation is used and no prior preparation is necessary. Rather than employing an analytical process to guide performance analysis, Autopilot uses fuzzy logic to automate the decision-making process.

PerCo

The PerCo system is a framework for controlling the performance of distributed applications in heterogeneous environments, such as computational Grids. It is capable of monitoring the progress of the application's execution and redeploying it to optimise performance. 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.

This framework is oriented to two HPC application domains: coupled models for scientific simulation [1] and distributed search for statistical disclosure control [35]. Functionally, these types of applications are composed of simulation components which run individually, sharing state information at regular intervals.

The structure of the PerCo system comprises three main modules: *PerCo loaders*, distributed *Control Performance Steerers* (CPS) and the *Application Performance Steerer* (APS).

There is a PerCo loader for each component of the application, and it is responsible for launching and migrating the components. During component migration, the communication between loaders is required to transfer input data and check-point files. So, the set of loaders constitute the redeployment architecture.

Each component presents an interface to communicate with a CPS. Using this interface a CPS controls its associate component, gathering performance information and applying performance commands.

The entity which takes control over the complete application, i.e. over all the components, is the APS. The APS receives data from the CPSs and maintains a historical database of performance data. Using this information, the APS is capable of determining improved component configurations, which will be applied in the components by the specific CPSs. To make redeployment decisions the APS policy is based on performance prediction. The performance prediction used in PerCo [23] combines both time-series and regression analysis.

In [16], Chen et al. present a model for a framework based on the PerCo system. This model focuses on dynamically adapting the execution environment to changes in resource availability. It distributes the functionality of the centralised APS in a hierarchical

structure, dividing decision making into local and global policies. The model is oriented towards providing fault tolerance and multi-level load balancing. This model has been simulated at a high level. Changes in the simulated resources are represented by random modifications of their simulated characteristics at run-time. A functional tool based on this model does not appear to have been developed.

Instead of being a general dynamic tuning system, as is the case of the approach in this thesis, PerCo is designed to operate over a specific type of application. In these applications the tuning performed by PerCo is restricted to the movement of processes between available resources, rather than using dynamic instrumentation to modify user defined points in the application employed by the proposed approach. The performance analysis conducted by PerCo differs from analytical performance models, in that it uses historical data combined with techniques based on time series.

Continuous Program Optimization Paradigm

In [12], Cascaval et al. present the conceptual model and the initial implemented prototype of a paradigm for *Continuous Program Optimisation* (CPO). This paradigm focuses on assisting in and automating the performance tuning of applications in current hardware and software environments. The CPO paradigm is based on two recurring phases, monitoring and optimisation.

Monitoring involves collecting and analysing performance data from the different layers of the system (from the hardware to the application). In the implementation of the CPO paradigm, the framework PEM [60] allows for vertically integrated performance monitoring information to be gathered from the hardware to the application levels.

Optimisation involves using the collected information to adapt the application to its current execution environment and adapt the execution environment to enhance application performance. Optimisations are implemented through agents that are instances of PEM clients. These CPO agents model the application behaviour based on performance data coming through PEM, and store this model in a database. Using this information, CPO agents negotiate resources that may either directly enhance performance or do so indirectly, by enabling further code adaptations. To complete the feedback pattern, the execution is continuously monitored to validate the previous problem diagnosis and verify that the applied modifications are having the expected effects.

In [11], Cascaval et al. presents a use case of the CPO paradigm where an offline agent and an online agent cooperate to optimise a large page usage. The offline agent performs a page size benefit analysis, storing the results. The action of the offline agent is used as a training phase, and in subsequent runs of the application the online agent uses

those stored results to determine which data categories should request the underlying system to map large pages to.

In this paradigm, historical data is the primary basis for conducting performance analysis and tuning. The model proposed in this thesis, does not uses prior execution data, as it is oriented towards tuning applications with highly variable runtime behaviour, a situation where data from previous execution is often not helpful.

2.2.3 Scalability of Dynamic Tuning Tools

The performance tools presented in the last section have aspects in their design that limit their operation over large-scale parallel applications. Such aspects are mainly related to a centralise component.

As it was mentioned in Section 2.2.1, MATE presents scalability barriers in its architecture due to the centralised module that guide the dynamic tuning process. Its centralised scheme becomes a bottleneck when the volume of communication and data to be managed increases due to the number of tasks of the parallel application. In [13] a first attempt to improve MATE's scalability was presented. A distributed collection of events is proposed and the experiments were conducted over a parallel application up to 32 tasks.

The client-server architecture of Active Harmony relegates the responsibility of carrying out performance analysis over the entire parallel application to the Adaptation Controller. This centralised component can take a long time to adjust the tuning parameters when the search space grows, due to having many parameters or having parameters with many possible values.

In [17], different techniques are presented in order to scale the analysis process in Active Harmony. These techniques are focused around selectively reducing the number of tuning parameters or values by prioritising certain parameters or looking for relationships between them. A separate technique is to group independent tuning parameters and assign each group to a separate Active Harmony Server. However, this is only possible when the parameters are completely independent and can be individually measured. The experiments presented in this work have been executed at a scale of around 400 cores. Also, the scalability barriers derived from the tuning of a single parameter across many thousands of cores is not considered.

Similarly to Active Harmony, Autopilot and PerCo make use of centralised modules to perform the analysis and tuning phases. In the case of Autopilot, no evidence of attempts to scale this tool have been found in the literature. For the PerCo tool a

conceptual model that distributes the Application Performance Steerer exists. However, it does not appear to have been implemented.

Continuous Program Optimisation presents a distributed analysis model through the CPO agents. However, the monitoring infrastructure PEM, is centralised and processes must contend for access to write to the centralised event log. In a large-scale context the PEM infrastructure would become a bottleneck.

2.3 Scalability of Performance Analysis tools

Currently, different performance analysis tools exist which are able to work with parallel applications involving hundreds or thousands of processes, such as Scalasca [38], Paradyn [37], Periscope [4] and the TAU performance system [52].

To achieve the objective of this work, which is to provide an approach for scalable dynamic tuning of large-scale applications, the techniques and schemes that such tools make use of in order to be scalable have been studied.

These techniques try to avoid saturated centralised points of processing. For instance, Scalasca exploits the parallelism offered by the environment where the parallel application is running. In the cases of Paradyn and TAU, a distributed communication framework is used to offload some of the processing tasks. Meanwhile, Periscope presents a hierarchical architecture designed specifically to be scalable.

In the following sections, the major scalable performance analysis tools along with their scalability properties are discussed.

2.3.1 Scalasca

Scalasca is a post-mortem performance analysis tool. It has been specially designed to analyse parallel application execution behaviour on large-scale systems such as Blue Gene and Cray XT, but is also well-suited for small and medium scale HPC platforms. The current version of Scalasca can be applied to simulation codes from science and engineering, based on the parallel programming interfaces MPI and/or OpenMP written in C/C++ and Fortran.

Scalasca offers an incremental performance analysis procedure that integrates runtime profiling and post mortem analysis of event traces, adopting a strategy of successively refined measurement configurations. Distinctive features are its ability to identify wait states in applications with very large number of processors [5] and to combine these with efficiently summarised local measurements.

To collect performance data, Scalasca offers a mix of manual and automatic instrumentation mechanisms to be applied to the target application. When running the instrumented code on the parallel machine, if profiling is chosen by the user, the Scalasca profiler emits only a single report file at the end of the application execution, which is collated in parallel using MPI collective operations.

When tracing is enabled, each process generates a trace file containing records for its process-local events. To developed a scalable pattern of analysis and wait states search, after program termination, Scalasca loads the trace files into main memory and analyses them in parallel by replaying the original communication on as many CPUs as have been used to execute the target application itself. The result consists of a report similar in structure to the summary report but containing higher-level communication and synchronisation inefficiency metrics.

To interactively examine the XML summary and pattern reports files generated, Scalasca also provides an analysis report explorer. These XML files can also be visualised using third-party profile browsers such as TAU's ParaProf [28].

The latest efforts in improving the Scalasca tool scalability [19] are mostly oriented towards a hierarchical reimplementation of the unification algorithm. This algorithm is used in order that the event data measured during the measurement acquisition phase is consistent across all processes of the parallel application.

The scalable design of Scalasca, exploiting its profiling pattern and parallel event tracing scheme, has facilitated performance analysis and tuning of a range of applications on Cray XT and XE systems [61]. It has managed unprecedented numbers of processes, such as 294,912 on IBM Blue Gene/P and 196,608 on Cray XT5 [62].

2.3.2 Paradyn

Paradyn is an on-line performance analysis tool for parallel and distributed applications. Paradyn supports monitoring MPI applications on IBM AIX, Solaris, and Linux platforms, providing program instrumentation and automatic performance analysis during the execution of the parallel application. Such functionality is achieved through the use of the dynamic instrumentation technique. Therefore, Paradyn is capable of inserting and modifying instrumentation during run-time, collecting only data that it needs for monitoring and automatic performance analysis.

The automatic search of performance bottlenecks in Paradyn is carried out by the Performance Consultant module. The Performance Consultant is a centralised process and its search is based on the W3 search model (why, where and when) [21]. This model is based on answering three separate questions: why is the application performing poorly,

where is the bottleneck and when does the problem occur. Such an approach allows quick and precise isolation of a performance problem, guiding Paradyn's instrumentation in search of it.

To eliminate the centralised nature of the Performance Consultant and extend the scalability of Paradyn, later works [50], [49] present and evaluate a distributed approach for the Performance Consultant. The performance bottleneck search strategy defined in this new approach deals with local and global application behaviour. For local behaviour, the search examines a specific process behaviour, delegating control to a search agent running on that process host. To examine global application behaviour, the strategy uses MRNet [51] for efficient aggregation of performance data collected from all application processes, as well as for overall tool control during the search. In this work experimental results show how they applied the proposed distributed search on 1024 application processes.

2.3.3 Periscope

Periscope is an on-line distributed performance analysis tool. It allows analysing of performance issues of parallel MPI applications as well as evaluating single node performance. The performance problem detection conducted by Periscope is automatic and occurs while the application is running. Such detection is based on summarised information, and in this phase the structure of the application is known and exploited.

To be scalable, the architecture of Periscope is composed of a *analysis agent network*. This network consists of three different types of agents: the master agent, communication agents and analysis agents.

The *analysis agents* are the leaves of network and search autonomously for predefined performance problems in a subset of the application's processes. The application processes are linked with a monitoring interface that allows the analysis agents to configure the measurements, collect performance data and control when the application starts, halts and resumes.

The performance problem detection is carried out in one or more experiments, which are represented by an iterative phase of the application. The detection strategy is based on a set of hypotheses, that define an initial set of problems or behavioural patterns that are to be checked in the first experiment, as well as a refinement process that permits the creation of new hypotheses from the problems already found. Therefore, the analysis agents start from the initial hypotheses, execute the specific phase of the application in order to gather the performance data and finally evaluate which hypotheses hold. When necessary, the hypotheses can be refined and a new detection cycle is performed.

At the end of the local search, the detected performance problems are communicated through the network of *communication agents* to the *master agent* which interacts with the user. Communication agents combine similar performance problems found in their child agents and forward only the combined properties.

Periscope also offers a user-interface that displays the results of the performance problem detected on runtime behaviour of the parallel application.

Two years ago, as an extension of Periscope, the Periscope Tuning Framework (PTF) arose under the AutoTune project [36]. Such an extension aims to help developers in the process of tuning a parallel application. PTF identifies tuning alternatives based on codified expert knowledge and evaluates the alternatives within the same run of the application (online). At the end of the application execution, PTF produces a report on how to improve the code, which can be manually or automatically applied. Currently, PTF includes plugins for tuning the performance and the energy consumption.

2.3.4 TAU Performance System

The TAU performance system is an integrated toolkit for performance instrumentation, measurement, offline analysis, and visualisation of parallel applications. TAU can be executed in the majority of the current HPC platforms and supports applications written in C, C++, and Fortran languages, as well as the use of standard message passing (e.g., MPI) and multi-threading (e.g., Pthreads) libraries.

TAU implements a flexible instrumentation model that is applied at different stages of program compilation and execution. Different instrumentation techniques are supported, including dynamic instrumentation using the DynInst API [10].

The TAU measurement library supports scalable performance profiling and tracing techniques. When tracing is enabled, every node/context/thread will generate a trace for instrumented events. TAU writes performance traces for post-mortem analysis, but also supports an interface for online trace access. This includes mechanisms for online and hierarchical trace merging [8], [9].

As in tracing, profiles are collected and stored on a per-thread basis. TAU profiling system support two types of profiling: *flat profiling* and *event path profiling*. Flat profiling is able to give information about a specific event, but not within the context of other events. On the other hand, event path profiling allows more specific context information about the relationships between events.

TAU parallel profile analysis environment consists of a framework for managing the profile data, PerfDMF [22], and a profile tool, ParaProf [28]. After the measurement, the profile data can be loaded in the profile experiment database of PerfDMF. The

use of this database not only allows a wide range of analyses of the collected profile data but also comparisons between experiments. ParaProf provides the user with a graphical visualisation of the parallel profile data. The visualisation is based on scalable histogram and three-dimensional displays for large profiles. The input to ParaProf can come directly from the PerfDMF database.

TAU leverages third-party software packages, such as Vampir [44] for sophisticated trace analysis and visualisation.

Recent works [45] [46] [27] show the extension of the TAU performance system in order to conduct scalable, low-overhead online performance monitoring and analysis of parallel applications. The first approximation was shown in [45] using the scalable monitoring system Supermon as a transport runtime system for monitoring data. Supermon [55] presents a hierarchical structure of servers that gather, concentrate and transport monitoring data to the root of the hierarchy, which provides the data to the clients (Performance database, online visualiser or application steerer).

In [46] an extension of the latest work based on MRNet was presented. In this case MRNet is also used as a transport runtime system but the capabilities of the filters in the internal nodes of the network are used to provide distributed runtime performance analysis while the data flows upstream in the network.

In [27] extending the last two works, it is shown how online analysis operations can also be supported directly and scalably using the parallel infrastructure provided by an MPI application instrumented with TAU. In this work successful experiments with 131K cores show the scalability of this tool.

2.4 Conclusions

In this chapter, a range of dynamic performance tuning tools - MATE, Active Harmony, Autopilot, PerCo and CPO - have been outlined. Each of these 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.

The existence of performance analysis tools, such as Scalasca, Periscope, Paradyn and TAU, that are capable of analysing parallel applications executed on thousands of processes has also been detailed. This scalability is achieved in different ways, however in all cases it is based on distributing the analysis process amongst multiple components.

These tools offer different kinds of analysis, offline and online. However, until now, these tools do not offer dynamic tuning.

It is at this intersection, where dynamic tuning meets scalability, that the contribution presented in this work provides an advance in the state of the art of performance analysis and tuning.

3

Scalable Dynamic Tuning

"The major achievement of modern science is to demonstrate the links between phenomena at different levels of abstraction and generality, from quarks, particles, atoms and molecules right through to stars, galaxies, and (more conjecturally) the entire universe. On a less grand scale, the computer scientist has to establish such links in every implementation of higher level concepts in terms of lower. Such links are also formalised as equations or more general predicates, describing the relationships between observations made at different levels of abstraction."

– Charles Antony Richard Hoare, *Mathematical Models for Computing Science*

This chapter introduces a model that enables dynamic tuning for large-scale parallel applications. This model is designed as a hierarchical tuning network of modules which independently perform analysis and tuning in a distributed manner. The challenge of estimating the additional resources required for dynamic tuning is also addressed in this chapter, through the calculation of efficient tuning network topologies. To finalise, an approximation of the time required to achieve a global performance improvement utilising the proposed tuning network is presented.

3.1 Introduction

Typically dynamic tuning tools are organised following a centralised architecture, as we discussed in Section 2.2.3. Such tools are composed of a single tool component, which can be called the analysis and tuning module, that controls and interacts with a large number of tool daemons, the back-ends, which are responsible for data collection and application control. Generally, the execution of the tool back-ends is distributed because they can run in the same node where the application task is executing. The analysis and tuning module acts as a coordinator allowing the daemons to work together. It is normally also in charge of managing the collected performance data and conducting performance analysis and tuning over the entire parallel application.

When working with parallel applications involving hundreds or thousands of tasks, the centralised analysis and tuning module becomes a bottleneck due to centralised computation and communication with all back-ends daemons, and no longer provides effective analysis and tuning. Therefore, providing effective dynamic tuning in large-scale contexts requires:

- The elimination of a single centralised point responsible for analysing and tuning the application.
- The distribution of the analysis and tuning process in such a way that it remains effective.

The capacity of a centralised tool will always be limited in terms of the size of the parallel application that it can support. Even the most efficient centralised tool will become a bottleneck given a sufficiently large number of tasks to analyse. So, in order for an analysis and tuning tool to be truly scalable, it must present a design that is able to be adapted to the size of the parallel application that it is operating over.

On the other hand, the performance problems being resolved are often highly complex and require global knowledge about the state of the parallel application. In this case the analysis and tuning process must be designed to use mechanisms that permit the resolution of performance problems without resorting to a single point of absolute control.

In this chapter, we propose a new approach to perform dynamic tuning for large-scale parallel applications, overcoming the limitations that exist in a centralised design.

We start by discussing the different schemes to support scalability for dynamic performance analysis and tuning in Section 3.2. Then, in Section 3.3 we present our proposal, a model that enables decentralised dynamic tuning of large-scale parallel applications. The decentralised scheme takes the form of hierarchical performance analysis

and tuning, based on the decomposition of the parallel application and the abstraction of its behaviour. The requirements of the proposed model are satisfied by a hierarchical tuning network of analysis and tuning modules whose design is detailed in Section 3.4.

The topology of a hierarchical tuning network can be adapted to the size of the parallel application being analysed and tuned. In Section 3.5 we address the challenge of selecting this topology according to the characteristics of the performance analysis and tuning process and the parallel application being analysed.

We finalise the chapter in Section 3.6 where we discuss the time required to achieve global performance improvements following the specification of the model proposed in this chapter.

3.2 Distributed Approaches to Support Scalability

In order to develop a scalable dynamic analysis and tuning design, it is necessary to avoid a single centralised analysis and tuning module. We investigated two possible alternatives that allow us to achieve the decentralisation of this component.

The two possible options considered for the decentralisation are a fully distributed approach or a hierarchically distributed approach. In this section, the advantages and drawbacks of each option are evaluated in the context of performance analysis and tuning.

A fully distributed analysis is characterised by not presenting any centralised component. This type of analysis would be conducted by a set of analysis and tuning modules that would be expanded on one level over the parallel application tasks. Each analysis and tuning module monitors, analyses and tunes a subset of application tasks. To achieve global improvements of the application's performance, it is necessary to establish a communication pattern between analysis and tuning modules for transferring and exchanging performance data about the application. Using its own information and the information from its neighbours, each analysis and tuning module can infer a global performance improvement and tunes its set of application tasks.

The hierarchical alternative consists of analysis and tuning modules structured as a hierarchical tree. Performance information about the parallel application would flow from the tasks of the application to the analysis and tuning modules located at the base of the hierarchy, and from them to the analysis and tuning modules at the remaining levels. In this way, the higher an analysis and tuning module is in hierarchy, the larger the segment of the application which it has knowledge about. However, analysis and tuning modules located at higher levels of the hierarchy will most likely have a coarser view of the application than those modules which reside at lower levels.

In a hierarchical distribution, problems which require a global view of the application can be resolved by the analysis and tuning module which resides at the root of the hierarchy. The lower levels can be responsible for resolving the same problem over the subset of the application which they control, with a restricted but more detailed view. Alternatively, each level in the hierarchy can resolve a distinct performance problem depending on the view of the application that it has.

The fully distributed analysis approach is inherently scalable, but it tends to generate poor global performance improvements. Due to the absence of a centralised component, this design takes longer to spread information about the global state of the application to all analysis and tuning modules. For this reason, it takes longer to achieve quality improvements that consider the overall state of the application when it has a large number of tasks.

As with the fully distributed design, the hierarchical option provides effective local performance improvements. However, the hierarchical approach has features that make it somewhat less scalable because a centralised analysis point exists. This is offset by the fact that the hierarchical structure makes it possible to achieve better global performance improvements, and achieve them faster than the fully distributed option. This is possible because the centralised analysis and tuning module has a holistic vision of the state of the parallel application.

Considering the advantages and disadvantages of each approach discussed, we need to choose the best option to scale the analysis and tuning process, maintaining its quality while avoiding the introduction of unnecessary complexity.

The hierarchical structure has been chosen because it offers the best compromise between scalability and the effectiveness of the analysis and tuning process. While it is true that the ultimately centralised nature of a hierarchical structure makes it somewhat less scalable than a fully distributed configuration, the trade-off is considered worthwhile given that the hierarchical approach will achieve quality global improvements faster on applications with a large number of tasks.

3.3 Model for Hierarchical Dynamic Tuning

To address the challenge of tuning large-scale parallel applications at runtime, we propose a model that follows a decentralised scheme for dynamic tuning. This model is based on the hierarchical distribution of the analysis and tuning process, and uses an abstraction mechanism that offers a reduced representation of the application state, enabling global performance improvements to be achieved.

The model involves decomposing the parallel application, which is too large to analyse and tune in a centralised manner, into a number of disjoint subsets of tasks which can be dealt with separately. Analysing and tuning each of these separated subsets, called ***domains***, will lead to local performance improvements in the parallel application.

However, in order to achieve global performance improvements, the application must be viewed as a whole. To achieve this, in this model we propose that each separate domain has to be abstracted so that it can also be treated as a single parallel application task. When taken together, the tasks representing all the domains form a new *virtual* parallel application 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 process is repeated and an additional virtual parallel application is formed.

Following this decomposition and abstraction process, we obtain a hierarchical tree of virtual parallel applications, with the lowest level being the real parallel application and the highest level having few enough tasks to be analysed and tuned in a centralised manner, as shown in Figure 3.1 for an SPMD application. At each level in the hierarchy, the virtual parallel application is composed of tasks which abstract the state of a domain in the application at the level below. In this case, the virtual application follows the same programming paradigm as the original application. At each higher level in the hierarchy the virtual application presents a coarser representation of the original application state.

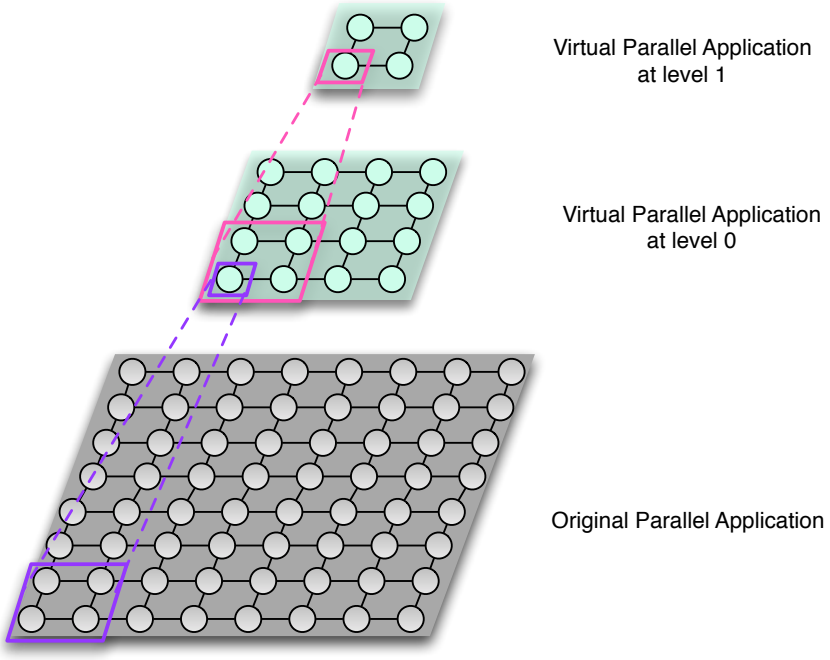


FIGURE 3.1: Decomposition and abstraction of an SPMD application following the model for hierarchical dynamic tuning.

Analysis and tuning is performed separately on each subset in each virtual parallel application. The results of this analysis are actually carried out, via the abstraction mechanism, on the underlying segment of the original application which is represented by this subset. 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.

The model can also be applied to less homogeneous applications. As an example, Figure 3.2 presents a master-worker of pipelines application and its corresponding decomposition and abstraction. In this case, only the pipelines are decomposed and abstracted between levels, the master task in each virtual application merely represents its real counterpart. The decomposition allows the original application to be analysed locally for each piece of the pipeline. The abstraction allows the highest level virtual application to be analysed in terms of its behaviour under a master-worker paradigm.

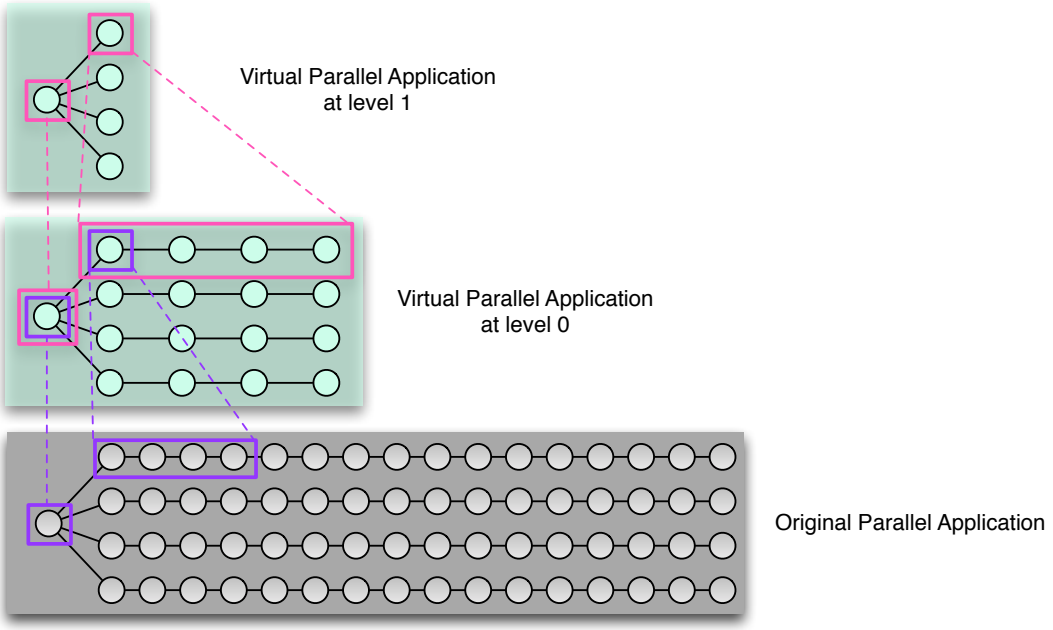


FIGURE 3.2: Decomposition and abstraction of a master-worker of pipelines application following the model for hierarchical dynamic tuning.

Dynamic tuning of large-scale parallel applications arises from the two main processes of the proposed model. The decomposition permits the analysis and tuning over manageable sized portions of the application; and the abstraction mechanism joins these separate application domains to carry out global performance improvements.

3.3.1 Knowledge Required for Collaborative Hierarchical Tuning

Dynamic performance analysis and tuning are conducted during the execution of the parallel application. This fact implies that analysis and modification must be as simple as possible. The resulting decisions from the performance analysis phase have to be taken in a short period of time in order to effectively tackle the detected bottlenecks.

Additionally, an important aspect to consider is that the intrusion generated by the monitoring and tuning process must be minimised, because it can affect the application behaviour and could even generate previously non-existent performance problems. For this reason, the monitoring and tuning process must not present a high degree of complexity and must be carried out carefully.

All of these restrictions are very complex to meet if there is no previous knowledge about the structure and behaviour of the application. Approaches based on *blind* dynamic tuning show poor results and the effectiveness of dynamic modifications might be significantly reduced. With the aim of avoiding such limitations it would be beneficial to ask for user collaboration in order to provide specific information about the application and how to detect and fix its performance problems. In this work, such a *cooperative* approach is followed.

Historically, our research group has proposed a collaborative approach [40] [14] [43] for tackling these issues. This approach consists on the one hand, a framework tool for driving the analysis and tuning process and, on the other hand, on integration of knowledge provided by the user about the application behaviour.

Following this collaborative approach, users will have to provide knowledge for the proposed hierarchical tuning model about how to conduct performance analysis and tuning (the performance model), as well as how to carry out the decomposition and abstraction process (the abstraction model).

Performance Model

The analysis and tuning process aims to evaluate the application behaviour by collecting performance data, detecting bottlenecks and giving solutions to overcome them. Different possibilities exist to evaluate the application behaviour, such as analytical performance models, heuristic techniques, historical performance information, fuzzy logic, regression analysis or time-series techniques.

The approach used in the proposed model is based on analytical performance models and rules. Analytical performance models are a set of formulae, expressions or algorithms that determine the optimal performance conditions as well as predict the performance

of an application. These models are usually parameterised by application and system characteristics. This approach can be application-specific or based on simple models.

To represent a performance model for dynamic tuning, we have adopted the terminology used by MATE. A performance model defined for dynamic tuning must consist of:

- A set of measurement points, which determines the parameters of the application to be monitored and the points where they have to be measured. The values of the data gathered from the measurement points are the inputs to the performance model.
- 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 previously mentioned inputs of the performance model.
- 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.

As previously discussed, the model for hierarchical dynamic tuning proposed in this work permits a distributed performance analysis. So, at each level of the hierarchy a performance model is used. Depending on the performance problem to be overcome, as well as the structure of the original parallel application being analysed, each level in the hierarchy may be analysed using the same performance model, as in the case of an SPMD application, or different models, as for a master-worker of pipelines application. In this last case, returning to Figure 3.2, a master-worker performance model would be used at the highest level in the hierarchy, while a pipeline performance model would be employed at the remaining levels.

Abstraction Model

To conduct a distributed dynamic analysis and tuning process, our model also requires a description about how to perform the abstraction between levels in the hierarchy. This knowledge forms the *Abstraction model* and is composed of 4 parts:

- How to translate the monitoring points between different levels in the hierarchy, in order to apply monitoring points from one level to its child level.
- How to summarise the information collected from monitoring points, so that a subset of tasks from one level in the hierarchy can be represented as a single *virtual* parallel application task.

- How to translate the set of tuning points, actions and synchronisation method between different levels in the hierarchy, in order to apply tuning actions from one level to its child level.
- How to decompose a real or virtual parallel application into subsets of tasks which can be analysed and tuned following the performance model to be used at its level in the hierarchy.

These four aspects must take various details into consideration, such as the structure and the behaviour of the parallel application, specific characteristics of the performance issue that is being resolved, and differences in the performance model being used at each level of the hierarchy.

3.3.2 Decomposition and Abstraction of Performance Problems

The model proposed in Section 3.3 allows dynamic tuning over large-scale parallel applications by distributing the analysis and tuning process. This distribution influences the possible strategies which can be followed to overcome performance problems in a parallel application. Specifically, a performance model must have certain characteristics if it is to be used effectively when operating over large-scale parallel applications.

First, the proposed model calls for the decomposition of the parallel application into separate subset of tasks, each of which will be analysed and tuned separately. As such, it is necessary that the performance model employed to conduct analysis and tuning can be applied to these disjoint subsets.

Second, in order to achieve a global vision of the parallel application, subsets of tasks are abstracted and represented as if they were a single parallel application task. Consequently, it must be possible to apply the performance model over the resulting virtual parallel application.

The complete functionality of the proposed hierarchical dynamic tuning model is based on using performance models that meet both of these requirements, that is they are *decomposable* and *abstractable*. However, under the proposed model it is possible to resolve problems in a parallel application that require a performance model that only fulfils one of these requirements.

As an example, optimising the memory usage of the application tasks only requires local analysis. In this situation no abstraction occurs as the problem can be solved for each task individually. On the extreme, other problems can only be solved at a global level, supported by the abstraction mechanism. An example of such a situation is tuning the MPI parameter `MP_EAGER_LIMIT`, which consists of setting the optimal maximum size for sending messages without the handshaking protocol.

However, many problems can be attacked in a hierarchically distributed way in order to quickly solve performance issues, in this case analysis is performed at all levels in the hierarchy. For example, 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.

3.4 Hierarchical Tuning Network Design

The model for large-scale dynamic tuning presented in Section 3.3 is based on the decomposition of the parallel application to be analysed and tuned, and an abstraction mechanism which allows the representation of the application state with different granularity. Using this decomposition and abstraction, the proposed model takes the form of a hierarchy of disjoint levels composed of virtual parallel applications. At each level in the hierarchy, a distributed analysis and tuning process is performed, the result of which is applied to the real parallel application.

The design of a tool based on the model must include mechanisms which drive the inter-level communication and allow the coordinated operation of this analysis and tuning process throughout the levels in the hierarchy, in order to improve the global performance of a parallel application.

Based on the concepts and architecture of analysis tools, this design structures the hierarchy as a set of **analysis and tuning modules** (ATMs) configured as a hierarchical tree. These ATMs form the ***hierarchical tuning network***.

In this section, we present the scheme of this hierarchical tuning network, which arises from the decomposition and abstraction process defined by the hierarchical tuning model. We also show how the required knowledge, specified in Section 3.3.1, is introduced into the tuning network. Finally, the synchronisation methods to coordinate the holistic operation of the network are depicted.

3.4.1 Hierarchical Tuning Network

In order to decentralise the analysis process, our model for hierarchical dynamic tuning calls for a tuning network of distributed analysis and tuning modules (ATMs), structured as a hierarchical tree over a parallel application.

These ATMs provide analysis and tuning as well as representing the tasks of the virtual parallel applications described in Section 3.3. This structure is shown in Figure 3.3.

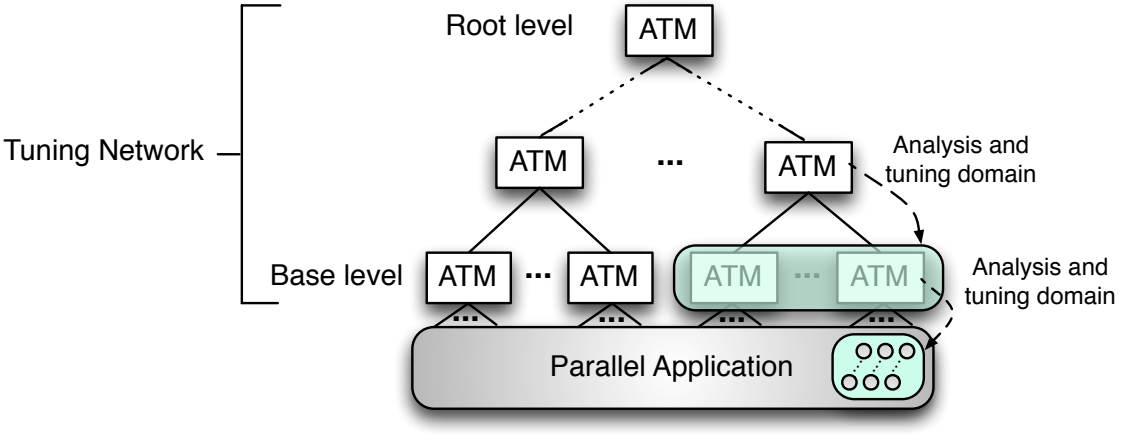


FIGURE 3.3: General hierarchical tuning network.

To form the base of the tuning network, it is first necessary to decompose the parallel application into disjoint subsets of tasks, according to the decomposition process of the proposed model. Each subset of application tasks then makes up the *analysis and tuning domain* of an ATM at the base level of the tuning network.

These base level ATMs are directly responsible for improving the performance of the application tasks in their domain. Following the dynamic tuning process presented in Section 2.2, these ATMs operate in three continuous phases: monitoring, performance analysis, and modification. First, the ATMs instrument the parallel application, using *instrumentation orders for monitoring*. These orders indicate the parameters of the applications that have to be measured and the points where they can be found to gather information about the behaviour of the application. In the analysis phase, ATMs receive this information in the form of *events*, search for bottlenecks, and give solutions for overcoming them. Events are messages originated in the parallel application tasks which contain performance information previously requested by a monitoring order. 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.

In order to obtain a homogeneous behaviour in all the ATMs in the tuning network, the analysis scheme of the ATMs at the hierarchy's base is replicated in the higher levels of the hierarchy. These higher level ATMs also operate over their child ATMs following the three phases previously mentioned: monitoring, analysis and tuning. From the point of view of these ATMs, their child ATMs form a parallel application.

To do this, the *abstraction mechanism* previously introduced is used. In the tuning network, this mechanism consists of representing each ATM as a parallel application task to its parent ATM. Therefore, the ATMs of one level behave as the *analysis and*

tuning domain of their parent ATM. A detailed description of how this mechanism fits into the tuning network is presented in Section 3.4.2.

We use the SPMD application shown in Figure 3.4 to illustrate these concepts. This application consists of 16 tasks connected in a 4×4 grid. According to the decomposition process of the proposed model, the 4×4 grid is split into subgrids of size 2×2. These subsets of tasks can be analysed and tuned separately under the SPMD paradigm. So, each 2×2 subgrid is the analysis and tuning domain of an ATM located at the base level of the tuning network. Therefore, this level of the network is composed of 4 ATMs (A1.1, A1.2, A1.3 and A1.4) which, following the abstraction mechanism, form a virtual application from the point of view of the parent level in the hierarchy. This virtual application also follows an SPMD paradigm, so these 4 ATMs should act as parallel application tasks *abstracting the behaviour* of the real tasks within their domain. These ATMs will become the analysis and tuning domain of the root ATM A1.

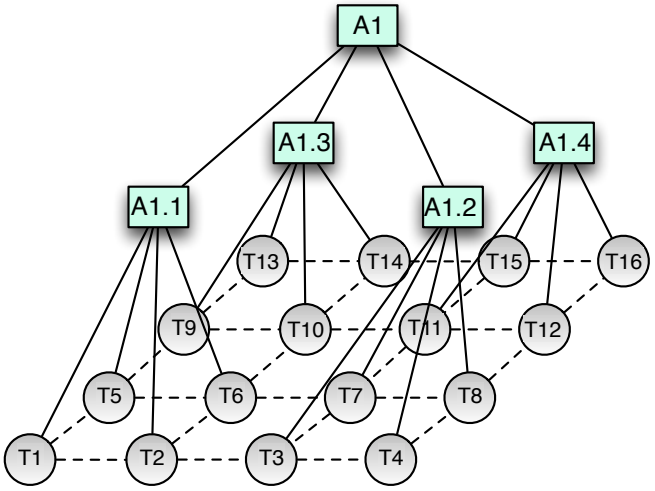


FIGURE 3.4: Hierarchical tuning network over an SPMD application.

As it was mentioned in Section 3.3, our approach also works with other programming paradigms such as hierarchical master-worker or master-worker of pipelines. The structured nature of these kinds of applications allows an analysis and tuning process using different tuning techniques throughout the analysis hierarchy. Considering the second case, a master-worker of pipelines where each worker is a pipeline, we can decompose the application so that parts of the analysis can be performed independently for each pipeline, and globally for the whole master-worker setup [20]. Figure 3.5 presents an example of a master-worker of pipelines composition and the associated hierarchy of ATMs that would be defined according to our proposal. Each ATM that controls a pipeline has to abstract the behaviour of its domain and act as a virtual worker task from the point of view of the root ATM. In this case, the virtual application is composed of three

virtual workers (A1.1, A1.2 and A1.3) and a virtual master (A1) which represents the task T1.

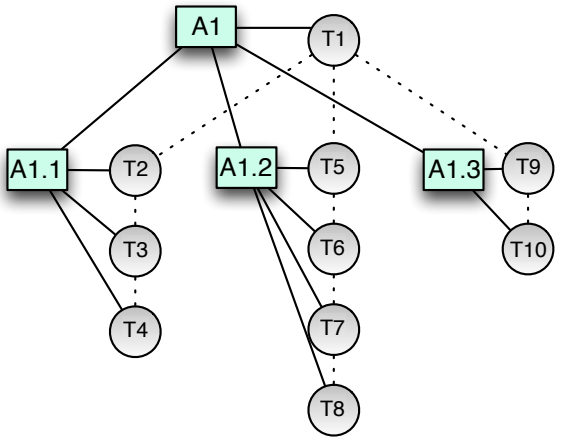


FIGURE 3.5: Hierarchical tuning network over a master-worker of pipelines application.

3.4.2 Abstraction Mechanism

The abstraction mechanism between levels in the hierarchy is the key principle for both, a) conducting a scalable and decentralised analysis, and b) efficiently managing the volume of data necessary to perform analysis and tuning.

The functionality required to carry out the abstraction mechanism is provided by a component called the *Abstractor*, associated to each ATM. This *Abstractor* is responsible for representing its associated ATM as a parallel application task to its parent ATM.

The *Abstractor* abstracts the performance information which is received in the form of events from the analysis and tuning domain of its associated ATM, and sends this abstracted information 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. These two processes allow performance analysis over the entire parallel application.

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 Figure 3.6, 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.

The functional design of the *Abstractor*-ATM pair and its communication paths are shown in Figure 3.7.

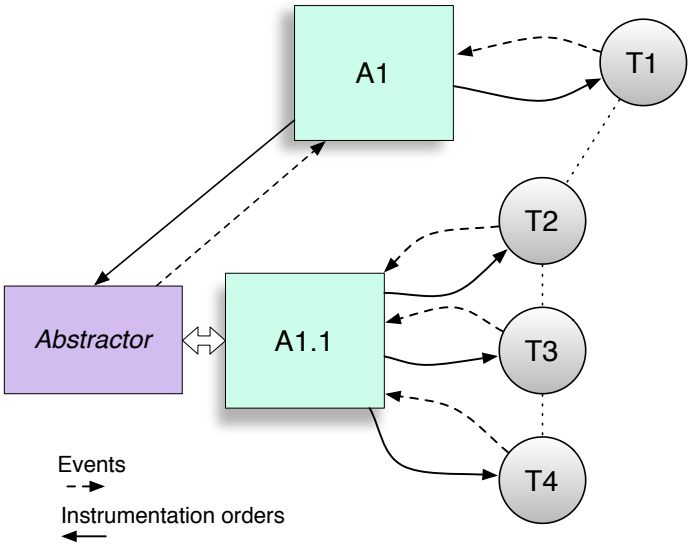


FIGURE 3.6: Detail of Figure 3.5 showing the tuning network using the Abstractor module.

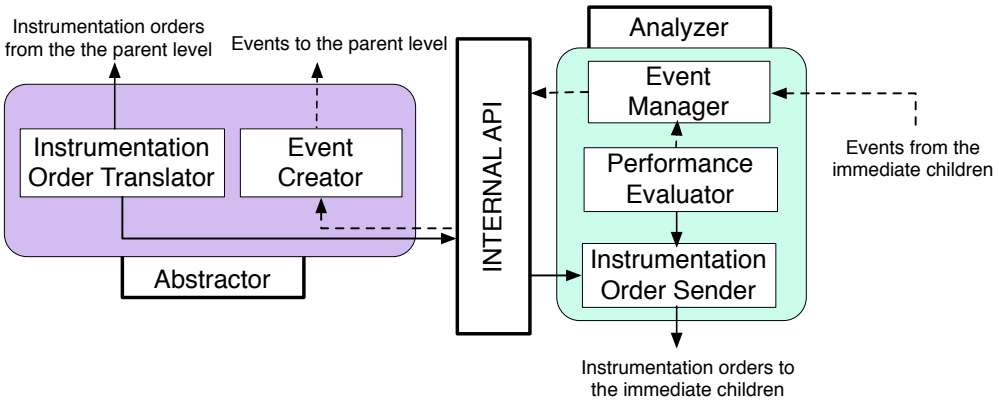


FIGURE 3.7: Abstraction mechanism: *Abstractor*-ATM design.

The *Abstractor* is composed of two main modules:

- **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 main modules:

- **Performance Evaluator**. This module contains the information necessary to conduct the monitoring, analysis and tuning of the parallel application.
- **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.

3.4.3 Knowledge in the Hierarchical Tuning Network

To conduct dynamic performance analysis and tuning according to the proposed model, the integration of the knowledge detailed in Section 3.3.1 into the tuning network is required. This knowledge is made up of a performance model, that guides the performance analysis and tuning of the parallel application, and the abstraction model which allows the definition of virtual applications at each level of the tuning network.

The performance model drives the behaviour of the ATMs. It defines what should be measured in the parallel application, how to analyse its behaviour and what changes must be applied to improve its performance. Specifically, this knowledge must be integrated in the *Performance Evaluator* component of the ATMs.

The abstraction model is primarily based on components of the *Abstractor* module. The translation of monitoring orders, which encapsulate monitoring points, should be located in the *Instrumentation Order Translator*. This translation mechanism takes the form of a function that maps a single instrumentation order for monitoring into others, one for each child in the analysis and tuning domain. The new monitoring orders must generate information to fulfil the information requested in the original order.

The *Abstractor* receives information about the state of the children in the analysis and tuning domain of its associated ATM in the form of events. This information has to be summarised using a reduction function that will result in a new event representing the state of the analysis and tuning domain of the associated ATM as if it were a single

application task. The *Event Creator* component contains the knowledge for summarising this information.

The translation of tuning orders (tuning points, actions and synchronisation) is hosted by the *Instrumentation Order Translator*. This translation process implies that when a tuning order is received by the *Abstractor* from the parent level, this order must be translated into orders for one or more of the children in the analysis and tuning domain of its associated ATM. The translation of these orders has to be performed in such a way that the resulting instrumentation produces the desired change in the application.

The knowledge required for the decomposition of the real or virtual application into domains is not hosted in any component of the *Abstractor* module. It is used before starting the analysis and tuning process when the configuration of the tuning network is established.

3.4.4 Synchronisation Policies

The behaviour of the hierarchical tuning network proposed in this work is based on the actions of the ATMs in various levels. The question that arises is how to combine the operation of all ATMs in order to provide an effective dynamic tuning.

Various synchronisation policies can be employed in order to ensure that the decisions made by each ATM are applied in such a way that their combined effect produces an improvement in the parallel application's performance.

In this section, three different classes of synchronisation policies are explored. These three policies have been called *Synchronised Analysis*, *Synchronised Tuning*, and *Asynchronous*.

Synchronised Analysis. Using this policy, the actions of the ATMs of the tuning network are synchronised by level. At a given time, only the ATMs in a single level are actively performing analysis and tuning. The tuning process begins at either the top or the bottom of the hierarchy. Assuming that it is the top, the root ATM acts until it is satisfied with the performance of its analysis and tuning domain. The root ATM then signals its children ATMs, and they become active and begin their own analysis and tuning process. Once an ATM has finished improving the performance of its analysis and tuning domain it notifies the root ATM. Once the root ATM has received this signal from all the ATMs at that level, it triggers the activation of the next level of ATMs. This pattern is repeated for each level in the hierarchy until the ATMs at the base level have finished, and the cycle begins again.

During the synchronised analysis cycle, the ATMs in inactive levels will not be functioning, however their associated *Abstractors* will still be operational, creating new events that flow upstream through the tuning network and translating the instrumentation orders for monitoring and tuning that go downstream. Due to the periods of inactivity that ATMs suffer during the cycle, this policy may leave performance problems undetected for unacceptable lengths of time because they can only be discovered by an ATM at a specific level.

This policy could be a good option in situations where the tuning orders from one level would counteract those of a child or parent level if they were blindly applied at the same time, or if tuning performed by an ATM at one level would drastically change the apparent state of the analysis and tuning domain of an ATM at a child or parent level. However, synchronised analysis does not make the best use of resources, as ATMs are inactive during the synchronisation cycle. Moreover, an explicit communication mechanism between ATMs in different levels is required to implement the signal that activates the analysis and tuning process at each level. This extinguishes part of the beauty of the abstraction mechanism, as ATMs must be aware that they have other ATMs in their analysis and tuning domain.

Synchronised Tuning. With a synchronised tuning policy, all ATMs are actively performing analysis in parallel. However, instrumentation orders for tuning are not sent until a corresponding tuning order has been received from the parent ATM. When an *Abstractor* receives a tuning order, it is responsible for integrating this order with the ones generated by its associated ATM. This integration consists of taking the ATM's more finely grained knowledge of its analysis and tuning domain into account when performing the instrumentation order translation, as well as combining the newly created tuning orders with those already generated by its associated ATM.

This policy is only applicable if higher levels in the hierarchy have high priority, i.e. it is a top-down scheme. Due to the integration of tuning orders this policy may be able to obtain significant performance improvements with fewer changes in the application. On the other hand, if the integration mechanism during the instrumentation order translation is complex and costly, tuning orders will incur delays as they flow downstream through the tuning network, which may reduce their effectiveness. This integration mechanism also requires additional knowledge from the user to conduct the performance analysis and tuning process.

Asynchronous. The final possibility is an asynchronous policy. In this case ATMs are continually active during the application execution. Each ATM acts completely independently, performing analysis and sending tuning orders when necessary. All tuning orders are received by the application task and applied.

The asynchronous characteristic of this policy provides dynamism in the behaviour of the ATMs at each level of the hierarchy. Because of this, the detection of performance problems takes place as quickly as each ATM is able to react, they are not forced to wait for any signal or synchronisation coming from parent or child levels. Furthermore, no additional knowledge or logic is required in the *Abstractor*-ATM pair. When using this policy, it is important that the combination of tuning orders coming from different levels converge towards an improvement of application performance, otherwise the application performance could be continually oscillating.

Because all ATMs work in parallel and tuning orders from different levels can be received by a single application task at the same time, a coherence mechanism is necessary at the point where the application is instrumented for tuning. This mechanism controls priority of tuning orders from different levels and also ensures that tuning orders are only applied if they have been generated based on a reasonably recent parallel application state. In a performance tool this mechanism would be located in the module that controls the insertion of instrumentation orders for tuning in each parallel application task.

The advantage of this coherence mechanism is that it can be generalised, does not require specific knowledge about the performance analysis and tuning process.

Tuning Coherence Mechanism

A hierarchical tuning network, performing asynchronous analysis and tuning, requires a coherence mechanism. This will reside in the performance environment module that controls the insertion of instrumentation orders for tuning into each parallel application task.

To provide coherence, this mechanism places two constraints on instrumentation orders for tuning:

- **Non-stale tuning orders**. It is possible that while an ATM from one level collects performance data, analyses them and sends a tuning order, an instrumentation order for tuning from an ATM at another level in the hierarchy produces a significant change in the state of the application. When this situation occurs the information upon which the tuning order was based is no longer valid, and it is denoted *stale*. The coherence mechanism must keep track of the last instrumented change in its associated task, and discard tuning orders based on an application state measured prior to this change.
- **Inter-level priority**. When a tuning order is received by the module that controls the instrumentation, it is not applied until the parallel application task passes

through the associated tuning point. Until this moment, this order is *pending*. While an order is pending, there is the possibility that an order from another level arrives. The coherence mechanism compares the priority of the ATM that sent each order to decide which order should be kept and which order should be discarded. ATMs have a priority based on their level in the hierarchy. Levels in the hierarchy will either have ascending or descending priority, depending on the characteristics of the performance issue to be resolved.

Figure 3.8 shows a flow diagram detailing the combined application of these two constraints, which form the coherence mechanism.

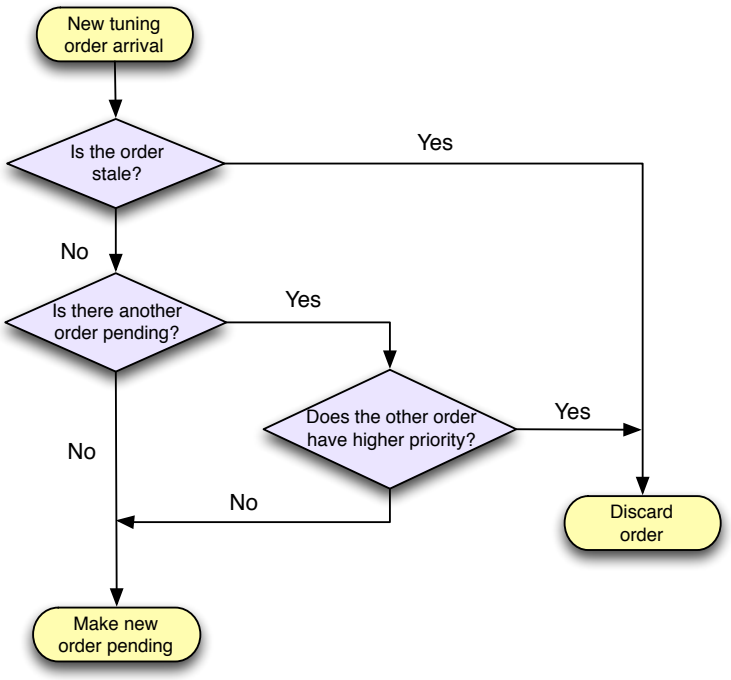


FIGURE 3.8: Coherence mechanism.

3.4.5 Case Studies

In this section, we use two theoretical examples to describe how the hierarchical dynamic performance analysis and tuning process is performed. These examples are focused on the three phases of dynamic tuning and how the hierarchical tuning network behaves in each case.

The first example illustrates a master-worker of pipelines application analysed by a tuning network which employs abstraction and performance models focused on energy efficiency. The second example shows how a tuning network could be structured over an SPMD application to conduct load balancing using the appropriate abstraction and performance models.

Monitoring

The monitoring phase allows the instrumentation of the parallel application tasks to obtain information related to the application's behaviour.

Instrumentation orders for monitoring are generated by the ATMs' *Performance Evaluator* module and are transferred to the *Instrumentation Order Sender*. This module sends these orders to the analysis and tuning domain of its ATM, triggering their propagation through the analysis hierarchy.

As they flow through the hierarchy, monitoring orders will be received by parallel application tasks or by *Abstractors* associated to descendant ATMs. In the first case, the orders will be inserted into the application tasks. In the second case, a monitoring order translation phase occurs. At the time of receiving a monitoring order, the *Abstractor's Instrumentation Order Translator* module will translate it. The new order, or orders, must provide data equivalent to what was requested in the original monitoring order. The translated orders will be transferred from the *Instrumentation Order Translator* to the *Instrumentation Order Sender* of its associated ATM. Then, these orders will be sent to the analysis and tuning domain and continue their propagation down through the hierarchy.

To illustrate this translation phase, we will consider the master-worker of pipelines example shown previously in Figure 3.5. Suppose that the ATM A1 requires the energy consumption of each worker in its analysis domain (ATMs A1.1, A1.2, A1.3) in order to evaluate an energy efficient master-worker performance model. To get this information, A1 generates an instrumentation order for monitoring each worker. This order is received by the *Abstractors* associated with the ATMs A1.1, A1.2 and A1.3, which work with an energy efficiency pipeline performance model.

Let's focus on the *Abstractor* action associated with the ATM A1.1 (the actions for A1.2 and A1.3 are equivalent). The *Abstractor* interprets the received order so that the worker energy consumption is calculated from the pipeline energy consumption. In order to accomplish the requirements of the received monitoring order, the *Abstractor* creates a new order for each stage of the pipeline (T2, T3 and T4) to collect their energy consumption. These orders are transferred to the *Instrumentation Order Sender* and sent to T2, T3 and T4.

In the case of the SPMD example shown in Figure 3.4, suppose that the ATM A1 requires the computation time and the number of work units of each task in its analysis and tuning domain (A1.1, A1.2, A1.3, A1.4) in order to evaluate a performance model to balance the application load. To get this information, A1 generates an instrumentation order for monitoring for each child in its domain. The order is received by the *Abstractor*

associated to A1.1, A1.2, A1.3 and A1.4, which work with a load balancing abstraction model for SPMD.

Let's focus on the *Abstractor* action associated with A1.1. In order to accomplish the requirements of the received monitoring order, the *Abstractor* needs the computation time and the work units of each child in its analysis and tuning domain. So, the *Abstractor* creates a new order for each task in the analysis and tuning domain of its associated ATM, requesting this information. These orders are transferred to the *Instrumentation Order Sender*, which sends them to T1, T2, T5 and T6.

Analysis

In this phase, the *Performance Evaluator* module evaluates its performance model with the information about the analysis domain behaviour received in the form of events from its *Event Manager*. Events are generated during the execution of the parallel application as a result of previously inserted monitoring orders. These events will flow through the analysis hierarchy, being transformed, according to user provided knowledge, by an *Event Creator* at each intermediate level, until they reach the *Performance Evaluator* which originally requested the information they contain.

In the case of the master-worker of pipelines application, events containing the energy consumption of pipeline stages 2, 3 and 4 are received by A1.1. This ATM knows that this information was requested by its *Abstractor*. So, it passes these events to the associated *Abstractor*. The user provided knowledge will be used by the *Abstractor* to sum the energy consumption of each stage. Then, the *Abstractor* will encapsulate this energy information into an event which will be sent to the ATM A1. For this ATM, the received event contains the worker energy consumption, which was the information previously requested in the monitoring phase.

For the SPMD application example, events containing the computation time and the number of work units of tasks T1, T2, T5 and T6 are received by A1.1. The *Event Manager* knows that this information was requested by its *Abstractor*. Using the information specified in the abstraction model, the *Event Creator* module creates a new event containing a state representation of its analysis and tuning domain. This contains the sum of the number of work units and average computation time, for each child in its domain. The generated event is sent to the ATM A1 to satisfy the previously requested monitoring order.

Tuning

After finishing the performance model evaluation, the *Performance Evaluator* generates one or more instrumentation orders for tuning in order to fix the detected performance problems in its analysis domain. These orders will flow through the analysis hierarchy, being transformed into one or more new orders by the *Instrumentation Order Translator* at each intermediate level. The *Instrumentation Order Sender* will then inject these new orders downstream, until they reach the parallel application task.

For the master-worker of pipelines example, the ATM A1 may decide that one or more of its workers should reduce their energy consumption. Suppose that it sends such a tuning order to its child A1.1. The *Abstractor* of A1.1 will receive this order, which it must translate to be applied to the analysis and tuning domain of its associated ATM. In this case, the abstraction model indicates that the energy consumption of a pipeline can be reduced by deactivating some of the tasks and redistributing their pipeline stages across the remaining tasks. So, the *Abstractor* create an order for T4 ordering it to shut down, and other tuning orders for T2 and T3 telling them to host the stages that were previously hosted by T4. These orders are transferred to the *Instrumentation Order Sender* and sent to the stages T2, T3 and T4.

In the case of the SPMD example, suppose that after evaluating its performance model, A1 notices that A1.1 is more loaded than A1.3 and it decides to move load from one to the other. The *Abstractor* associated to A1.1 receives a tuning order to move load to A1.3. In this example, to successfully complete the tuning order translation process, the *Abstractor* knows, using the knowledge provided by the user, that this order has to be transformed into another tuning order for moving load from the application tasks T5 and T6 to tasks T9 and T10 respectively, because T5 and T6 are the only ones that share a common border with the analysis and tuning domain of A1.3.

3.5 Topology of the Hierarchical Tuning Network

Dynamic tuning tools must be active during the execution of the analysed application. As such, they require additional resources in order to reduce or eliminate interference with the application. However, in a time when reducing energy consumption costs is an important consideration in high performance systems, a careful balance must be struck when choosing the number of additional resources to dedicate to the analysis and tuning process. Using too few resources will hamper the effectiveness of the tuning tool, but unnecessary use of resources should be avoided to reduce power consumption.

In this section, we present an explicit method for calculating an efficient topology for our proposed hierarchical tuning network. The method provides the minimum number

of resources needed to carry out a dynamic and automatic analysis and tuning process, considering specific parameters that characterise the behaviour of the tuning network and the application being analysed. We also discuss the management of resource usage carried out by current performance analysis tools.

3.5.1 Method for Determining an Efficient Topology

Given a parallel application, the proposed tuning network may have several topologies. The structure of the topology will depend on the number of levels in the hierarchy and the number of *Abstractor*-ATM pairs in each level.

In this work, we propose the creation of tuning networks that use the fewest possible resources, so a topology with the minimum number of *Abstractor*-ATM pairs must be found. However, these *Abstractor*-ATM pairs cannot become saturated. An *Abstractor*-ATM pair is saturated if it cannot process and act upon the information that it receives from the application at the frequency with which it is sent.

The amount of work that an *Abstractor*-ATM pair performs is determined by characteristics of the performance and the abstraction models and the number of nodes which make up its analysis and tuning domain. In order to determine the domain size that an *Abstractor*-ATM pair can manage without becoming saturated, we have developed an expression that models the work performed by the *Abstractor*-ATM pair during each analysis and tuning process.

An analysis and tuning process is defined as the work that an *Abstractor*-ATM pair has to complete to perform the detection of performance problems and make decisions to improve the performance of the parallel application. Considering the variables presented in Table 3.1, we can define the total time that an *Abstractor*-ATM pair requires to carry out an analysis and tuning process as:

$$N \cdot E_a \cdot T_m + T_a(N) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp} \tag{3.1}$$

This work can be divided into several actions:

1. Gathering behavioural information about the nodes that make up the analysis and tuning domain of the ATM. The time required to gather an event and transfer it to the *Performance Evaluator* or the *Event Creator* is T_m. This action is repeated for each event received before detecting performance problems, i.e $N \cdot E_a$.
2. Detecting performance problems and making the necessary decisions which will improve the performance of the application. According to the characteristics of the performance model, the time taken by an ATM to detect performance problems,

TABLE 3.1: Description of variables which represent the performance and abstraction models.

Variable	description
N	the number of nodes in the analysis and tuning domain for each ATM
E_a	the number of events received from each node of the analysis and tuning domain before performing analysis to detect performance problems
E_c	the number of events received from each node of the analysis and tuning domain before creating a new event which will be sent to the level above in the hierarchy
$T_a(N)$	the time required to detect performance problems. This time can be constant or dependent of N
T_m	the time required to transfer each event received by the ATM to the correct module, where it is stored
T_c	the time required to create a new event which will be sent to the level above in the hierarchy
T_t	the time required to translate an instrumentation order which will be sent to the level below in the hierarchy
f_{rp}	the tuning order reception frequency from the parent *Abstractor*-ATM pair
f_{rc}	the event reception frequency from each child node of the analysis and tuning domain
f_e	the event generation frequency of the parallel application

$T_a(N)$, can be constant or depend on the number of nodes in its analysis and tuning domain.

3. Creating a new event using the behavioural information previously gathered, taking a time equal to T_c. The number of occurrences of this action in an analysis and tuning process is determined by E_a/E_c.
4. Translating an instrumentation order received from the parent *Abstractor*-ATM pair, taking a time of T_t. How often this action is performed depends on the arrival frequency of tuning orders sent by the parent *Abstractor*-ATM pair, f_{rp}.

The frequency at which an analysis and tuning process takes place in an *Abstractor*-ATM pair is $f_a = f_{rc}/E_a$. This value ultimately depends on the frequency with which events are generated in the parallel application task, f_e.

In order to avoid saturation, the maximum time that an *Abstractor*-ATM pair can dedicate to carry out an analysis and tuning process is $1/f_a$, the analysis period. This constraint is defined by the following expression:

$$N \cdot E_a \cdot T_m + T_a(N) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp} \leq \frac{1}{f_a} \tag{3.2}$$

If the time spent performing the work of an analysis and tuning process is equal to $1/f_a$, the *Abstractor*-ATM pair does not have free time between two analysis and tuning processes. In other words, it is occupied 100% of the time, making efficient use of the resources. A hierarchical tuning network composed of *Abstractor*-ATM pairs 100% occupied (but not saturated) will use the smallest possible number of resources and consequently, will have the minimum number levels in the hierarchy.

Knowing the characteristics of the performance and the abstraction models that guide the operation of an *Abstractor*-ATM pair, expression 3.2 can be used to determine the maximum domain size of an ATM without saturating it. The required variables can be obtained through prior evaluation of the application and the analysis process. Algorithm 3.1 shows the procedure for determining the topology of a tuning network. Beginning with the number of parallel application tasks, the first iteration of the loop calculates the number of *Abstractor*-ATM pairs which will form the base of the hierarchy. This is done by calculating the maximum domain size, N, at this level according to expression 3.2. These *Abstractor*-ATM pairs will become the tasks to be distributed amongst the ATMs in the next iteration of the loop. This procedure is repeated until the number of ATMs at level i in the hierarchy is equal to 1 - this ATM is the root of the tuning network.

Algorithm 3.1 Calculating the topology of the hierarchical architecture.

Input: $\#Tasks$ // Number of tasks of the parallel application.
Input: $E_a^{(i)}, E_c^{(i)}, T_a^{(i)}(N), T_m^{(i)}, T_c^{(i)}, T_t^{(i)}, f_{rp}^{(i)}, f_{rc}^{(i)}$ // Analysis and tuning process variables for each level i
Output: $Topology[]$

$i = 0$ // i indicates the level of the hierarchy. $i = 0$ means the base of the hierarchy.
$Current\_level\_tasks = \#Tasks$ // The base level tasks are all the tasks of the parallel application.

repeat
$N = \frac{\frac{1}{f_a^{(i)}} - \frac{E_a^{(i)}}{E_c^{(i)}} \cdot T_c^{(i)} - T_a^{(i)}(N) - T_t^{(i)} \cdot f_{rp}^{(i)}}{E_a^{(i)} \cdot T_m^{(i)}}$ // Solve for N
$\#AbstractorATM\_modules = (int)\frac{Current\_level\_tasks}{N}$
$Topology[i] = \#AbstractorATM\_modules$
$Current\_level\_tasks = \#AbstractorATM\_modules$
$i++$
until $\#AbstractorATM\_modules == 1$

It is possible that the variables which define the analysis and tuning process are different at each level i in the hierarchy. For example, a master-worker of pipelines would require different performance and abstraction models for each paradigm employed. In this case, the appropriate values for these variables must be used for each level i.

3.5.2 Resource Usage of Related Performance Analysis Tools

As we have detailed in Section 2.2, there are a number of tools which perform dynamic tuning of parallel applications, such as Active Harmony [56], Autopilot [48], and PerCo [34]. To perform dynamic tuning, these tools make use of a centralised analysis architecture, thereby using one additional machine to perform analysis.

To find examples of similar analysis architectures that exploit the capabilities of multiple resources, we examined the analysis tools presented in Section 2.3 that provide automatic performance analysis without dynamic tuning and work in large-scale systems.

The Scalasca toolset [38] performs a post-mortem analysis based on a parallel trace-analysis scheme. After application termination, Scalasca loads trace files into main memory and analyses them, making use of the same resources which were used to execute the parallel application.

In order to work in large-scale systems, efforts presented by the tools TAU [46] and Paradyn [50], use hierarchical architectures for analysis using MRNet. These hierarchical analysis architectures are run on additional resources. In these works, the objective is that analysis network nodes provide a reasonable performance, avoiding the resource saturation that would lead to a scalability bottleneck in the tool itself. However, in the architectures used in the experimental tests, no method is given to select the number of required resources or to justify the topologies presented from the point of view of resource usage.

In Periscope [4] the automatic analysis is performed by analysis agents structured according to a hierarchical scheme. Currently, the number of additional resources used to allocate the agents is a user defined parameter. The analysis agents notify the user if they were saturated during the last experiment, which allows the user to add additional analysis resources in future executions.

Unlike the tools presented in this section, our proposal constructs tuning networks considering the characteristics of the analysis and tuning process and the application itself. This gives the user much better control over balancing the use of resources with the quality of the tuning environment.

3.6 Hierarchical Tuning Performance Characterisation

Global performance improvements carried out by the tuning network may require the combined operation of the ATMs at all levels in the hierarchy. This is because while an ATM located at a given level seeks to improve the performance of its analysis and tuning domain, its actions may lead to instability in the state of the analysis and tuning domains of its children. The child ATMs will then need to carry out analysis and tuning to restabilise their domains through performance improvements. Therefore, a tuning action generated at a specific level will lead to a stabilisation process that descends, level by level, through the hierarchy.

A model of the time required to achieve global performance improvement in the proposed tuning network would give insight into the best manner to reduce the impact of

this iterative stabilisation process. The time required for the proposed tuning network to achieve a global performance improvement is heavily dependent on 1) the performance problem, 2) the performance model used to resolve this problem, and 3) the current state of the parallel application. Consequently, modelling the global performance improvement process is extremely complicated without providing specific details about the performance model.

However, it is possible to provide a detailed model of a single cycle required by an *Abstractor*-ATM pair to analyse the state of its analysis and tuning domain and apply an instrumentation order for tuning. Such a model is developed in Section 3.6.1, and it can be used together with specific information about the performance model, to give an approximation of the time required to achieve a global performance improvement, the motivation of this section. The outline for this approximation is given in Section 3.6.2

3.6.1 Hierarchical Tuning Cycle Model

The time required to apply a tuning order generated at level i is called the *tuning cycle* time at level i, and can be defined as:

$$T_{TC}^{(i)} = T_g^{(i)} + T_{mb}^{(i)} + T_a^{(i)} + T_{tu}^{(i)} \tag{3.3}$$

Where:

- $T_g^{(i)}$ is the time that it takes for the final event required to perform analysis to arrive at the ATM at level i.
- $T_{mb}^{(i)}$ is the time required to manage a batch of events at level i, after the last event required for the analysis phase has arrived.
- $T_a^{(i)}$ is the time required at level i to perform the analysis phase, detecting performance problems and giving solutions to overcome these problems.
- $T_{tu}^{(i)}$ is the time required to apply the tuning order generated in the analysis phase at level i.

Having in mind the abstraction mechanism between levels in the hierarchy, $T_g^{(i)}$ and $T_{tu}^{(i)}$ include processing and network times at each level below i.

As such, $T_g^{(i)}$ can be calculated as

$$T_g^{(i)} = \begin{cases} T_n^{(i)} + T_g^{(i-1)} + T_c^{(i-1)} + T_{mb}^{(i-1)} & \text{if } i > 0 \\ T_n & \text{if } i = 0 \end{cases}$$

Where:

- $T_n^{(i)}$ is the time required for an event to travel between levels $i-1$ and i.
- $T_c^{(i-1)}$ is the time required to create a new event to be sent to level i using the information received by the ATM at level $i-1$.

Equivalently, $T_{tu}^{(i)}$ can be expressed by

$$T_{tu}^{(i)} = T_n^{(i)} + T_I^{(i)}$$

where $T_n^{(i)}$ is defined as previously and $T_I^{(i)}$ is the time required to instrument the application given as

$$T_I^{(i)} = \begin{cases} T_t^{(i-1)} + T_n^{(i-1)} + T_I^{(i-1)} & \text{if } i > 0 \\ T_I & \text{if } i = 0 \end{cases}$$

where $T_t^{(i-1)}$ is the time required for the Abstractor at level $i-1$ to transform the tuning order generated at level i. This new order will be sent to the analysis and tuning domain of the ATM at level $i-1$.

At level 0, $T_I^{(0)}$ is simply the time required to instrument the actual parallel application tasks.

Having modelled the complete tuning cycle time for a given level i, $T_{TC}^{(i)}$, it can be seen that its value is proportional to the level at which the tuning decision is made. So, if we consider this time at the root ATM to be the global tuning cycle time, then we see that it is proportional to the total number of levels in the tuning network topology. This shows that, exploiting the scalability properties of hierarchical tree structures, the global tuning cycle time will grow logarithmically with the number of tasks in the parallel application being tuned.

3.6.2 Time Approximation for a Global Performance Improvement

The level by level stabilisation process described at the beginning of this section, leads to iterative behaviour, where a global performance improvement can only be assured after a series of tuning cycles are performed by ATMs located at inferior levels in the hierarchical tuning network.

As such, the time required to obtain an improvement in the application performance when the tuning order has been generated at level i is:

$$T_{PI}^{(i)} = T_{TC}^{(i)} + \sum_{k=0}^{i-1} X_k \cdot T_{TC}^{(k)} \tag{3.4}$$

Where X_k is the number tuning cycles required at each level k, and depends on the three points defined at the beginning of this section. $T_{PI}^{(i)}$ for the root level in the hierarchy, gives an approximation of the time required to obtain a global performance improvement.

Considering this behaviour, the number of levels of the tuning network topology will influence the time required to achieve a global performance improvement. Each level above the base may provoke destabilisation in the analysis and tuning domains of its child level. In terms of the interlevel destabilisation, the total number of levels in the hierarchy should be kept to a minimum, as is proposed in our method to calculate tuning network topologies.

At the same time, care must be taken in the design of the performance and *abstraction* models in order to reduce the secondary effects that a decision, generated at a specific level to improve the performance, will have on lower levels. Tuning actions that provoke serious instability will lead to longer stabilisation periods, and therefore the original tuning action will be less effective since the performance state of the application could have evolved in this period.

3.7 Conclusions

Dynamic tuning of parallel applications is a challenge in large-scale contexts. The majority of the current approaches that offer dynamic tuning follow a centralised scheme where a single component is responsible for improving the performance of the entire parallel application. When tuning parallel applications running on many thousands of processes, this centralised component becomes a bottleneck.

In this chapter, a solution to the problem of dynamic tuning for large-scale parallel application has been presented. This solution takes the form of a model that enables a decentralised scheme for dynamic tuning. Application decomposition and an abstraction mechanism are the two key concepts which support this model. The decomposition allows a parallel application to be divided into smaller parts to be analysed and tuned individually, while the abstraction mechanism allows these subsets to be viewed as a single virtual application so that global performance improvements can be achieved.

The model follows a collaborative approach to perform dynamic tuning. In order to operate it requires the integration of knowledge to guide the performance analysis and tuning process (the performance model) and the abstraction mechanism (abstraction model).

To design a dynamic tuning tool, the model is translated into a hierarchical tuning network composed of modules which perform analysis and tuning in a distributed manner. This effectively decentralises the responsibility for the improvement of the performance of the parallel application. Meanwhile, the abstraction mechanism permits an analysis and tuning process which does not need to be aware of the hierarchical structure in which it exists.

One of the principal advantages of this scheme is that the tuning network is adaptable to the size of the parallel application and the characteristics of the analysis and tuning process. In this chapter we have also presented a method that, considering these two aspects, calculates topologies composed of the minimum number of resources necessary to provide an effective tuning network.

Finally, by modelling the time required to achieve global performance improvements in the application, some insights have been drawn into best practices for designing performance and abstraction models.

4

Model Scalability Validation

"To climb steep hills requires a slow pace at first."

– William Shakespeare

This chapter presents the validation of the scalability of the proposed model for hierarchical dynamic tuning. To perform this validation, a simulation environment has been developed. To support the scalability validation, it has been verified that the proposed method to calculate efficient tuning networks produces topologies composed of the minimum number of resources required to provide an effective tuning environment.

4.1 Introduction

The model for hierarchical dynamic tuning presented in this work has been designed in order to permit tuning of large-scale parallel applications. The model's design is based on a hierarchical tuning network of analysis modules whose topology can be adapted in order to operate over different sized applications and achieve the required scalability.

When considering a scalable environment for dynamic tuning, it is important to validate the desired scalability without the distraction of tuning effectiveness. This is because, while a tuning strategy requires a quality environment in which to operate, its effectiveness is often influenced by many factors external to this environment. With this in mind, a simulation environment has been developed to validate the scalability of the proposed model for hierarchical tuning prior to performing actual dynamic tuning.

The simulation environment takes the form of a tuning network based on the proposed model, which directly implements the hierarchical communication aspects of the tuning network and simulates the analysis and tuning process. The design and implementation of the simulation environment and its parameterisation is explained in detail in Section 4.2.

An important aspect of the scalability of the hierarchical tuning network is its adaptable topology. Previously, in Section 3.5, we proposed a method to calculate efficient topologies for tuning networks. These topologies are composed of the minimum number of resources required to provide a quality tuning environment. The proposed method takes into consideration the size of the parallel application being analysed and the characteristics of the analysis and tuning process.

As a prior step to validating the scalability of the proposed model, a study was performed using the simulation environment in order to verify that the topologies calculated following the proposed method are indeed composed of the minimum possible number of resources while providing a quality tuning environment. This evaluation is described in Section 4.3.

Finally, the principal objective of this chapter, validating the scalability of the proposed model for hierarchical dynamic tuning, is detailed in Section 4.4.

4.2 Hierarchical Tuning Network Simulation Environment

A simulation environment has been developed in order to validate the scalability of the proposed model for hierarchical dynamic tuning. Based on the design of the hierarchical tuning network, presented in Section 3.4, this simulation environment implements a

tree based communication structure. The behaviour of this environment simulates all the actions that take place during the analysis and tuning process. These actions are parameterised in order to simulate the effects of different performance and abstraction models.

To implement the hierarchical communication scheme the MRNet framework is employed [51]. In summary, MRNet allows tools to use a hierarchical network of internal processes as a communication substrate between a front-end process and back-end processes via logical channels called streams. At MRNet internal processes, filters synchronise and aggregate dataflows, where they can efficiently compute averages, sums, and other more complex aggregations and analyses on tool data. An extensive description of MRNet will be provided in Section 5.3.2.

The parameterised simulation is guided by the parameters previously presented in Section 3.5.1 in Table 3.1. These parameters permit the configuration of the simulated behaviour of the *Abstractor*-ATM pair and application tasks.

In the simulation environment, application tasks are simulated by the MRNet back-end processes. Each back-end generates upstream packets (simulated events) at a certain frequency, f_e, following a normal distribution in time. As the principal aim of this environment is to evaluate the proposed hierarchical tuning network, the effect of tuning on these tasks is not considered. This is because we are focused on the scalability of the model and not the effectiveness of specific tuning strategies.

The MRNet internal processes, that represent the *Abstractor*-ATM pairs, are each divided into two components: the Upstream Filter (UF) and the Downstream Filter (DF). These filters simulate the behaviour of the performance and abstraction model integrated into the *Abstractor-ATM* pair. The UF works with data that flows upstream through the network (simulated events), and the DF works with data that flows in the opposite direction (simulated instrumentation orders).

The Upstream Filter performs a parameterised simulation of the three components of the *Abstractor*-ATM pair that operate on events that flow upwards through the network. These are the *Event Manager* (T_m) and the *Performance Evaluator* (E_a, $T_a(N)$) from the ATM, and the *Event Creator* (E_c, T_c) from the *Abstractor*. The UF also generates downstream packets that represent instrumentation orders generated as a result of the performance evaluation process.

Algorithm 4.2 describes the work simulated for each of these components. This algorithm defines the average amount of work that must be completed by an UF for each received batch of events. A batch of events is composed of one event from each child in the analysis and tuning domain.

The function `Work(t)` simulates `t` seconds of computational work.

Algorithm 4.2 Upstream filter pseudocode.

Input: $Events[], N, E_a, E_c, T_a(N), T_m, T_c$
Output: $TranslatedEvents[], TuningOrders[]$

```
Event_count = 0 // Received event counter
for all event in Events[] do
  Event_count++
  Work(T_m)
  if (Event_count%(E_a · N)) == 0 then
    Work(T_a(N))
    Append(TuningOrders[], newOrder())
  end if
  if (Event_count%(E_c · N)) == 0 then
    Work(T_c)
    Append(TranslatedEvents[], newEvent())
  end if
end for
```

The Downstream Filter simulates the *Instrumentation Order Translator* (T_t) from the *Abstractor* module. As a result of this operation, the DF generates one or more downstream packets, representing the translated instrumentation orders. In Algorithm 4.3, the work required to perform this operation is defined.

Algorithm 4.3 Downstream filter pseudocode.

Input: $InstrumentationOrders[], T_t$
Output: $TranslatedOrders[]$

```
for all order in InstrumentationOrders[] do
  Work(T_t)
  Append(TranslatedOrders[], newOrder())
end for
```

The front-end of the tuning network prototype is responsible for launching the hierarchical network, and from then on it simulates the root *Abstractor-ATM* pair.

4.3 Validation of Topology Selection Method

The goal of this study is to assess the efficiency of tuning networks built following the method described in Section 3.5.1. Specifically, we wish to verify that the proposed method is able to calculate topologies that are composed of the minimum number of non-saturated *Abstractor*-ATM pairs. The experiments to perform this validation have been conducted using the hierarchical tuning network simulation environment presented in the previous section.

The supercomputer SuperMUC at Leibniz Supercomputing Centre was used to perform the experimental tests. Each experiment used a subset of a single island composed

of 512 nodes interconnected by Infiniband FDR10. The nodes have 2 8-core 2.7GHz Intel Xeon processors and run SuSe Linux.

Each node on SuperMUC hosts 16 back-ends (one per core). On the other hand, each *Abstractor*-ATM pair[1] is assigned 4 cores because each of the MRNet internal nodes is multi-threaded and manages two threads for every directly connected node (children and parent) in the hierarchy [7].

The experimental assessment is structured in two use cases, each one presenting different circumstances which can arise when using a hierarchical tuning network.

4.3.1 Local and Global Analysis Process

Many problems can be attacked in a distributed manner in order to more efficiently solve performance issues. As an example, the analysis and tuning process required to perform load balancing can often be conducted both locally and globally. In this performance problem, local improvements can be achieved without worsening the global performance of the application.

In this use case, the hierarchical tuning network will perform both local analysis, at intermediate *Abstractor*-ATM pairs and global analysis at the root ATM. The abstraction mechanism allows the ATMs to carry out the same analysis and tuning process, regardless of which level they are in.

To represent this use case, we suppose a parallel SPMD application with 2048 tasks and performance and abstraction models characterised by the following values:

- f_e of 10 events per second per simulated task.
- E_a of 5, a performance problem detection phase is performed every 5 batches of events.
- E_c of 1, a single new event is sent to the parent level every batch of events, as such $f_{rc} = f_e$ at all levels.
- Linear $T_a(N)$ of $1ms$, i.e. the performance problems detection phase lasts $1ms$ per node within the analysis and tuning domain of the ATM.
- T_m of $1.1ms$.
- T_c and T_t of $0.1ms$.
- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds.

These values have been selected as coherent values which enables us to explore the capabilities of the tuning network with the available resources. However, we should

[1] In the experimental evaluation presented in this chapter, MRNet internal processes are referred to as *Abstractor*-ATM pairs, the components that they simulate the behaviour of.

not place too much importance on the specific values chosen, since we can calculate a topology for any set of values.

In this use case, we suppose that *Abstractor*-ATM pairs at all levels in the tuning network use the same performance and abstraction models. For this reason, these same values are used to parameterise the behaviour of the UFs and DFs at all levels in the hierarchy.

To calculate the predicted topology, the Algorithm 3.1 presented in Section 3.5.1 is used. Since E_c has a value of 1, the frequency at which batches of events are received by *Abstractor*-ATM pairs at all levels in the hierarchy is the same. Coupled with the fact that all *Abstractor*-ATM pairs in the hierarchy are simulated according to the same parameter values, this means that the maximum domain size will be the same for all *Abstractor*-ATM pairs independent of their level. For this reason, we only need to calculate the first iteration of the algorithm.

Substituting these values into Expression 3.2 (repeated below), the value of N_{max} can be calculated. This will give the number of children that an *Abstractor*-ATM pair can manage without becoming saturated.

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$

$$N_{max} = \left\lfloor \frac{\frac{1}{2} - \frac{0.5}{1000} - \frac{0.1}{1000} \cdot 0.1}{\frac{5.5}{1000} + \frac{1}{1000}} \right\rfloor$$

$$N_{max} = 76$$

With N_{max} equal to 76, each analysis and tuning domain controlled by an *Abstractor*-ATM pair at the base of the hierarchy will be composed of at most 76 application tasks. With a total of 2 048 application tasks, this gives 27 base level *Abstractor*-ATM pairs. As it was mentioned previously, *Abstractor*-ATM pairs at all levels in the hierarchy use the same performance and abstraction models and can also support 76 child nodes, so only one additional level is required, the root of the hierarchy.

This predicted topology is given for Experiment 1 in Table 4.1.

TABLE 4.1: Tuning network topologies for local and global analysis.

	Level 0		Level 1	
Experiment	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size
1	27	76	1	27
2	28	74	1	28
3	26	79	1	26

In order to achieve the goal of the experimental study, it is necessary to compare the predicted tuning network topology with other topologies, without changing the characteristics of the parallel application or the performance and abstraction models. Experiment 2 in Table 4.1 presents a topology with an additional level 0 *Abstractor*-ATM pair, meanwhile, Experiment 3 has one less *Abstractor*-ATM pair at level 0.

To determine whether or not the hierarchical tuning network is structured optimally to use the smallest number of non-saturated *Abstractor*-ATM pairs, we define a measurement called *analysis lag*. This is defined as the time between the generation of the final event required for an analysis and tuning process and the beginning of that process, as shown in Figure 4.1.

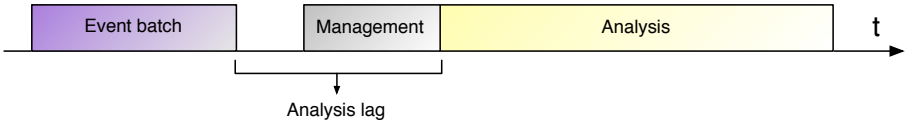


FIGURE 4.1: Analysis lag.

A saturated *Abstractor*-ATM pair presents steadily increasing analysis lag during the experiment's execution. This is because the ATM has an increasing backlog of events to process due to being unable to complete an analysis and tuning process within the analysis period. On the other hand, a stable analysis lag, one that does not change significantly over time, indicates a non-saturated ATM.

Figure 4.2 shows the analysis lag over time for the *Abstractor*-ATM pairs located at level 0 in the hierarchy for the three compared topologies. These results show the expected behaviour, wherein the analysis lag from Experiment 3 reflects its saturated state (due to having fewer level 0 *Abstractor*-ATMs than necessary). Experiments 1 and 2, are not saturated as demonstrated by the stability of their analysis lag.

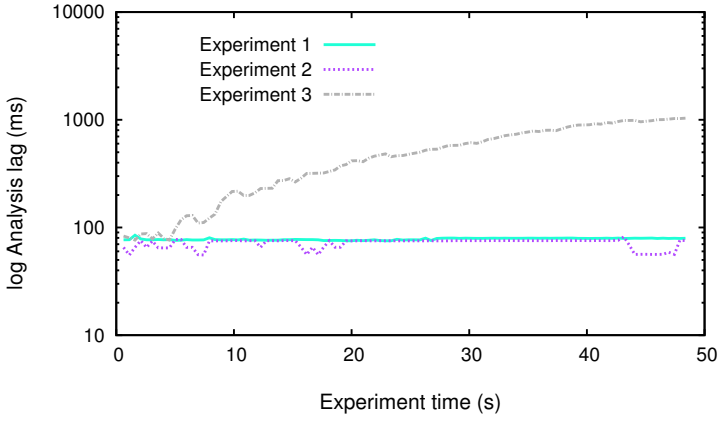


FIGURE 4.2: Analysis lag of the level 0 *Abstractor*-ATM pairs in milliseconds for local and global analysis.

Figure 4.3 shows that the analysis lag in level 1 does not change significantly between experiments. The root ATM is operating far below its saturation point, however the quality of the performance analysis and tuning process carried out by the root ATM in Experiment 3 will be greatly degraded by the saturated state of the level 0 *Abstractor*-ATM pairs.

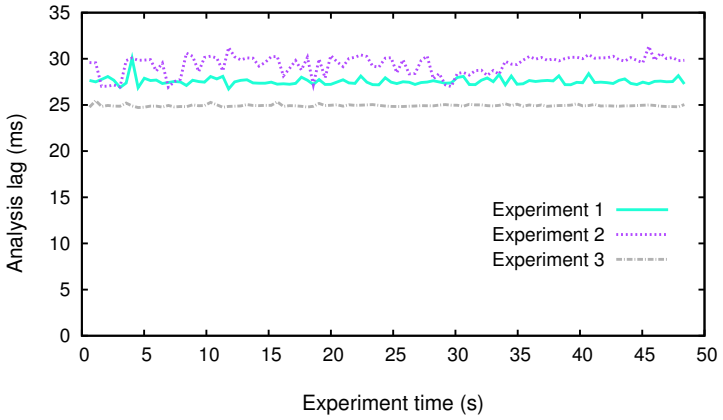


FIGURE 4.3: Analysis lag of the root ATM (level 1) in milliseconds for local and global analysis.

Figure 4.4 shows the average occupation of the *Abstractor*-ATM pairs for each level in the hierarchy. *Abstractor*-ATM pair occupation is used to measure the efficiency of the tuning network from the point of view of resource usage. This metric is calculated as the proportion of time that an Abstractor -ATM pair is carrying out abstraction and performance analysis and tuning actions against the total time that it is active. Experiments 1 and 3 have level 0 *Abstractor*-ATM pairs that make efficient (near optimal) use of the resources, but the level 0 *Abstractor*-ATM pairs in Experiment 2 are less occupied. This is due to having fewer nodes in the analysis and tuning domain than the maximum it can manage, whereas Experiment 1 has a number of nodes near the maximum and Experiment 3 exceeds this limit. Given that the level 1 ATM, the root, is operating below its saturation point, it is unavoidably under occupied in all experiments.

Taking into account the analysis lag and occupation, the predicted tuning network topology, Experiment 1, is the best candidate. It provides near optimal efficiency with respect to the use of resources while employing the minimum number of non-saturated *Abstractor*-ATM pairs. While Experiment 3 has level 0 *Abstractor*-ATM pairs with slightly higher efficiency, their saturated state would lead to a severely degraded analysis and tuning process. Meanwhile, Experiment 2 presents a small reduction in the analysis lag, but does not make efficient use of resources.

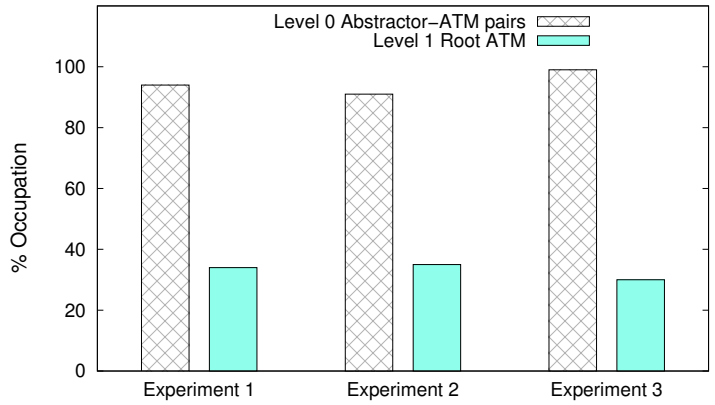


FIGURE 4.4: Percentage of occupation of the *Abstractor*-ATM pairs for each conducted experiment for local and global analysis.

4.3.2 Centralised-only Analysis Process

Certain problems can only be solved with a global view of the application, by the root ATM, while the rest of the lower level *Abstractor*-ATM pairs support it via the abstraction mechanism. An example of such a situation is tuning the MPI parameter `MP_EAGER_LIMIT`, which consists of setting the optimal maximum size for sending messages without the handshaking protocol.

In these cases the hierarchical tuning network would be heterogeneous, because the behaviour of the *Abstractor*-ATM pairs is different depending on the level at which they are located. The level 0 *Abstractor*-ATM pairs will aggregate data from their analysis and tuning domains to pass on to the root ATM, which will perform the actual analysis.

Once again we suppose a parallel SPMD application composed of 2 048 tasks. Parameter values used in the simulation are:

- f_e of 10 events per second per simulated task.
- E_a of 5, a performance problem detection phase is performed every 5 batches of events.
- E_c of 1, a single new event is sent to the parent level every batch of events, as such $f_{rc} = f_e$ at all levels.
- Linear $T_a(N)$ of $50ms$ for level 1 only, i.e. the performance problems detection phase lasts $50ms$ per node within the analysis and tuning domain of the root ATM.
- T_m of $0.42ms$.
- T_c and T_t of $0.1ms$.
- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds.

In this use case, the level 0 *Abstractor*-ATM pairs do not perform analysis, and so the $T_a(N)$ term can be removed from Expression 3.2, before calculating the maximum

number of child nodes that an *Abstractor*-ATM pair can support without becoming saturated. This maximum N_{max} is calculated as follows.

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$
$$N_{max} = \left\lfloor \frac{\frac{1}{2} - \frac{0.5}{1000} - \frac{0.1}{1000} \cdot 0.1}{\frac{2.1}{1000}} \right\rfloor$$
$$N_{max} = 237$$

So, each level 0 *Abstractor*-ATM pair can manage up to 237 child nodes. This gives a total of 9 *Abstractor*-ATM pairs at the base level of the hierarchy.

We now need to calculate N_{max} for the next level in the hierarchy. As the base level only contains 9 *Abstractor*-ATM pairs, we will assume that the next level will be the root. In order to confirm this, we will calculate the maximum domain size for this supposed root ATM. If $N_{max} \geq 9$, then this is our root ATM and the tuning network will be formed of two levels. Giving the assumption that this will be the root ATM, the times for event creation and instrumentation order translation are not included:

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a(N)$$
$$N_{max} = \left\lfloor \frac{\frac{1}{2}}{\frac{2.1}{1000} + \frac{50}{1000}} \right\rfloor$$
$$N_{max} = 9$$

This confirms our assumption that the hierarchy would be composed of two levels. This topology is given for Experiment 1 in Table 4.2. The evaluation process is the same as in the previous use case, where Experiments 2 and 3 represent tuning networks with one more and one less level 0 *Abstractor*-ATM pairs respectively.

TABLE 4.2: Tuning network topologies for centralised-only analysis.

	Level 0		Level 1	
Experiment	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size
1	9	228	1	9
2	10	205	1	10
3	8	256	1	8

Since level 0 *Abstractor*-ATM pairs do not perform analysis, we substitute the analysis lag measurement for *event creation lag*. The event creation lag is the time between the generation of the final event required to create a new event to be sent to the parent level in the hierarchy and the beginning of that event creation process.

Figure 4.5 shows the event creation lag for the level 0 *Abstractor*-ATM pairs. The most important observation here is that an *Abstractor*-ATM pair cannot handle, even theoretically, an infinite number of application tasks even when it has no analysis time. This is due to the time it takes to perform other duties, such as event management and the creation of events to be sent to the parent level in the hierarchy. Figure 4.5 shows that level 0 *Abstractor*-ATM pairs in Experiment 3 are saturated, as expected.

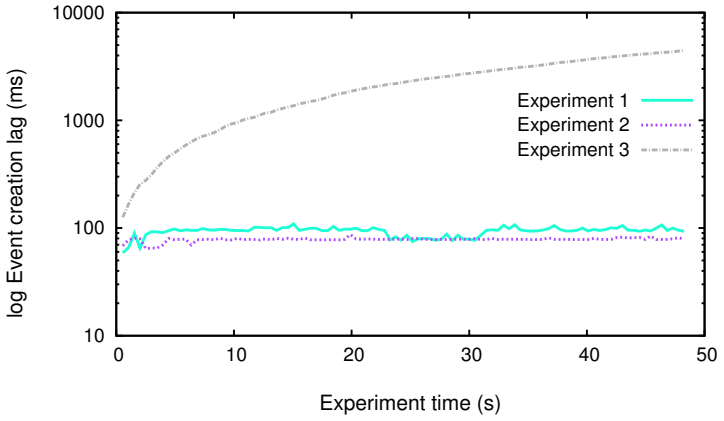


FIGURE 4.5: Event creation lag of the level 0 *Abstractor*-ATM pairs in milliseconds for centralised-only analysis.

The analysis lag for the root ATM, at level 1, is presented in Figure 4.6. In this case, the root ATM in the predicted tuning network is also close to saturation. As such, when an additional ATM is added in Experiment 2, alleviating the level 0 *Abstractor*-ATM pairs, this root ATM becomes saturated. This shows a border line case where, if only a few more application tasks were added, a third level in the hierarchy would be required.

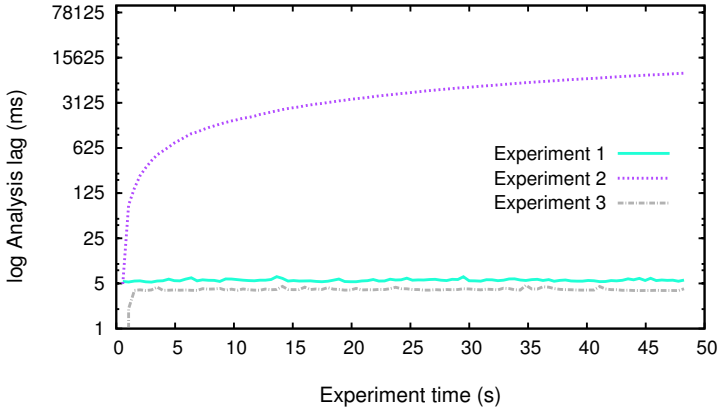


FIGURE 4.6: Analysis lag of the level root ATM in milliseconds for centralised-only analysis.

In contrast to the previous use case, the characteristics of this use case generate high occupation in both levels in the hierarchy, as can bee seen in Figure 4.7. In Experiment 1, both levels are almost 100% occupied because their analysis and tuning domains

present the maximum size. In Experiment 2, level 0 *Abstractor*-ATM pairs are slightly less occupied than in Experiment 1, while the level 1 ATM remains almost completely occupied. In fact the level 1 ATM is saturated, as we saw in Figure 4.6 - however the occupation measurement does not reflect this saturated state. This situation is reversed in Experiment 3, where the level 1 ATM is slightly less occupied than in Experiment 1 and level 0 *Abstractor*-ATM pairs show the same occupation level even though they are saturated.

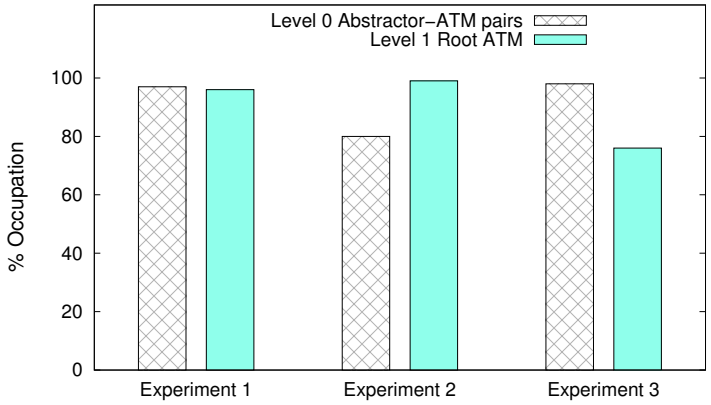


FIGURE 4.7: Percentage of occupation of the *Abstractor*-ATM pairs for each conducted experiment for centralised-only analysis.

As in the previous use case, the predicted hierarchical tuning network, given in Experiment 1, is in fact the topology that makes the best use of resources. Additionally, with this use case we show how the proposed method calculates the most efficient non-saturated topology, including heterogeneous situations where the analysis process differs from one level to another, up to extreme cases where some levels do not perform any analysis.

4.4 Validation of the Model's Scalability

In this section, we present an experimental evaluation that verifies the scalability of the proposed hierarchical dynamic tuning model. In order to perform this evaluation, we employ the prototype previously described in Section 4.2.

The primary measure of scalability is the time required by the tuning network to react to a performance problem in the parallel application, and how this time changes with the number of parallel application tasks. For this purpose, the global decision time is used.

The decision time at level i is defined as the segment of the tuning cycle time presented in Section 3.6, which ends at the level i ATM with the generation of a tuning

action. Using the same terminology, it can be defined as:

$$T_d^{(i)} = T_g^{(i)} + T_{mb}^{(i)} + T_a^{(i)}$$

Since both the tuning cycle time and the decision time include recursive terms, they are proportional. In this case, the decision time is used to reduce the complexity of the measurement process. The decision time is measured at each level i as the interval between the generation of the final event required for an analysis and tuning process in an ATM at level i, and the end of that process, when a tuning order is prepared. The global decision time, analogous to the global tuning cycle time, is the decision time at the root level of the tuning network.

The decision times presented in this evaluation represent the worst case because an analysis and tuning process is activated at each level in the hierarchy by the same event. As well as creating a new event to be sent to the parent level, the same event will also activate a performance problem detection process at the same level. Additionally, MRNet filters, which host the *Abstractor*-ATM pair functionality, operate sequentially. So, events created to be sent to higher levels in the hierarchy are not emitted until the simulated analysis and tuning process has been finalised. For this reason the decision time at any level i, includes the time required for an analysis and tuning process at each descendant level, as shown in Figure 4.8.

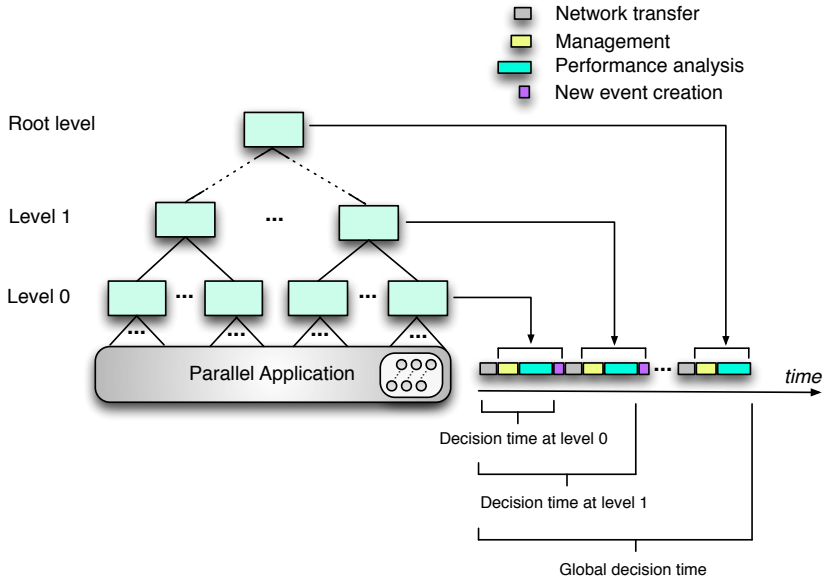


FIGURE 4.8: Decision time at each level in the hierarchy.

To carry out the desired validation, four experiments have been conducted. In the first three, we assess the behaviour of the tuning network with respect to different analysis time patterns. These patterns represent analysis times with different dependencies on the number of children to be analysed. The first experiment simulates an analysis time,

T_a, that does not depend on the size of the analysis and tuning domain. This constant time represents a case where the analysis time scales perfectly. The second and third experiments simulate analysis time that is linear and quadratic with respect to the size of the analysis and tuning domain. These cases represent good and poor analysis time scalability respectively. In the final experiment, we check that the scalability properties of the tuning network remain independent to the execution environment and continues for tests conducted in larger scale scenarios.

The experiments were performed using two different supercomputers, MareNostrum at the Barcelona Supercomputing Center and SuperMUC at Leibniz Supercomputing Centre. The first three experiments were conducted on MareNostrum, and the last experiment used both machines.

On MareNostrum we had access to a maximum of 256 compute nodes connected by Myrinet Network. Each node, running SuSe Linux, has 2 dual-core 2.3 GHz PowerPC 970MP processors. On SuperMUC we used an island composed of 512 nodes interconnected by Infiniband FDR10. Each node, running SuSe Linux, has 2 8-core 2.7GHz Intel Xeon processors.

In the experimental tests presented in this section, each back-end is allocated one core, whereas each *Abstractor*-ATM pair uses four cores. This is because each of the internal nodes of MRNet is multi-threaded and manages two threads for every directly connected node (children and parent) in the hierarchy [7].

In each of the experiments, the variables representing the abstraction and performance models were kept the same while the number of parallel application tasks was changed. Additionally, the same values were used for these variables for all levels within each tuning network. For each number of application tasks the tuning network topology was chosen using Algorithm 3.1 from Section 3.5.1. In the first three experiments, application sizes from 16 to 768 tasks were considered, while in the final experiment the size of the simulated application was between 25 and 6400 tasks.

4.4.1 Constant Analysis Time

In this experiment we explore a case where the analysis time is constant irrespective of the size of an ATM's analysis and tuning domain. This provides an example of a situation where the size of the tuning network is controlled, not by the analysis time, but by the management time, T_m.

Suppose a performance and abstraction model simulated by the prototype with the following parameters:

- f_e of 10 events per second per simulated task.

- E_a of 10, a performance problem detection phase is performed every 10 batches of events.
- E_c of 1, a single new event is sent to the level above in the hierarchy every batch of events, as such $f_{rc} = f_e$ at all levels.
- Constant $T_a(N)$ of $780ms$.
- T_m of $1ms$.
- T_c and T_t of $0.1ms$.
- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds

Using these values, it is necessary to calculate the maximum size of an analysis and tuning domain in order to calculate the tuning network topology for each parallel application size. Since E_c has a value of 1, this maximum domain size will be the same for all levels in the hierarchy, and only needs to be calculated once.

Proceeding in the same manner as in the previous section, the values presented are substituted into Expression 3.2 (repeated below), and the value of N_{max} can be calculated.

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$

$$N_{max} = \left\lfloor \frac{1 - \frac{780}{1000} - 10 \cdot \frac{0.1}{1000} - \frac{0.1}{1000} \cdot 0.1}{10 \cdot \frac{1}{1000}} \right\rfloor$$

$$N_{max} = 21$$

So, the maximum size of the analysis and tuning domain that an *Abstractor*-ATM pair can manage without becoming saturated is 21 children. This value is used to calculate the tuning network topologies, which are presented in Table 4.3.

Figure 4.9 depicts the average decision time measured for each level in the hierarchy for each application size.

TABLE 4.3: Tuning network topologies for constant analysis time for parallel applications composed of different numbers of tasks.

# Tasks of the parallel application	Level 0		Level 1		Level 2	
	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size
16	1	16	-	-	-	-
32	2	16	1	2	-	-
64	4	16	1	4	-	-
128	7	19	1	7	-	-
256	13	20	1	13	-	-
512	25	21	2	13	1	2
768	37	21	2	19	1	2

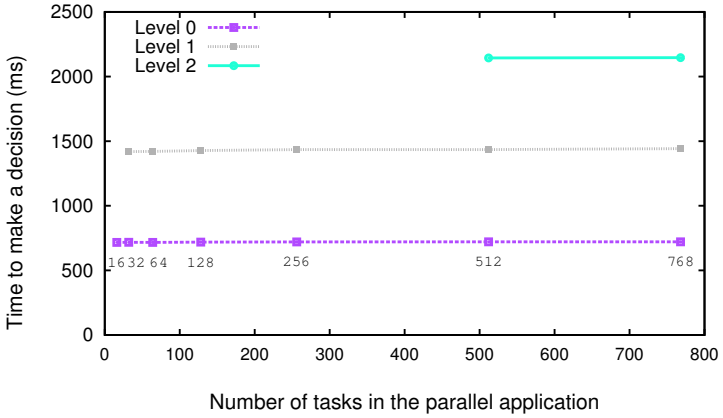


FIGURE 4.9: Scalability pattern with constant analysis time. The time required to make a decision is shown. The time is measured for each level in the hierarchy.

As it can be seen, the management time, which is essentially linear, does not greatly influence the global decision time due to its small value relative to the analysis time. For this reason, the global decision time remains almost constant for tuning networks with topologies composed of the same number of levels, even though the size of the root ATM's analysis and tuning domain may be different. This is the case for application sizes from 32 to 256 tasks (a two level tuning network) and from 512 to 768 tasks (a three level tuning network).

One may be led to think that an ATM with a constant analysis time could manage an analysis and tuning domain composed of infinite tasks. However, this is not the case. Because 10 batches of events are required for each analysis process, in this case the management time leads to the addition of new levels in the hierarchy. Furthermore, managing an infinite domain size is obviously not possible, as some degree of limitation will always exist at the hardware or operating system level. As such, the global decision time presents the same logarithmic growth as levels are added to the hierarchy. This will be observed in the linear and quadratic cases.

4.4.2 Linear Analysis Time

The major characteristic of this experimental evaluation is that the analysis time, T_a, is linear, i.e. the time required to perform an analysis process increases linearly with the size of an ATM's analysis and tuning domain.

Suppose a performance and abstraction model simulated by the prototype with the following parameters:

- f_e of 10 events per second per simulated task.
- E_a of 10, a performance problem detection phase is performed every 10 batches of events.

- E_c of 1, a single new event is sent to the level above in the hierarchy every batch of events, as such $f_{rc} = f_e$ at all levels.
- Linear $T_a(N)$ of $40ms$, i.e. the performance problems detection phase lasts $40ms$ per node within the analysis and tuning domain of the ATM.
- T_m of $1ms$.
- T_c and T_t of $0.1ms$.
- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds

The tuning network topologies are calculated using these values. Once again, the maximum domain size will be the same for all levels in the hierarchy because E_c has a value of 1.

Employing Expression 3.2, we can obtain the value of N_{max}.

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$

$$N_{max} = \left\lfloor \frac{1 - 10 \cdot \frac{0.1}{1000} - \frac{0.1}{1000} \cdot 0.1}{10 \cdot \frac{1}{1000} + \frac{40}{1000}} \right\rfloor$$

$$N_{max} = 19$$

The details of each topology calculated for this experiment are given in Table 4.4.

Figure 4.10 presents the average time required to make a decision for each level in the hierarchy for each application size.

For level 0, it can be seen that, when increasing the number of parallel application tasks, the ATMs quickly have their analysis and tuning domain filled up. From this point (64 application tasks) onwards the decision time at level 0 is roughly the same for the remainder of the application sizes. This situation occurs because, as the parallel application grows, additional level 0 *Abstractor*-ATM pairs are simply added, each one having the same, near maximum, domain size.

TABLE 4.4: Tuning network topologies for linear analysis time for parallel applications composed of different numbers of tasks.

# Tasks of the parallel application	Level 0		Level 1		Level 2	
	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size
16	1	16	-	-	-	-
32	2	16	1	2	-	-
64	4	16	1	4	-	-
128	7	19	1	7	-	-
256	14	19	1	14	-	-
512	27	19	1	27	-	-
768	41	19	2	21	1	2

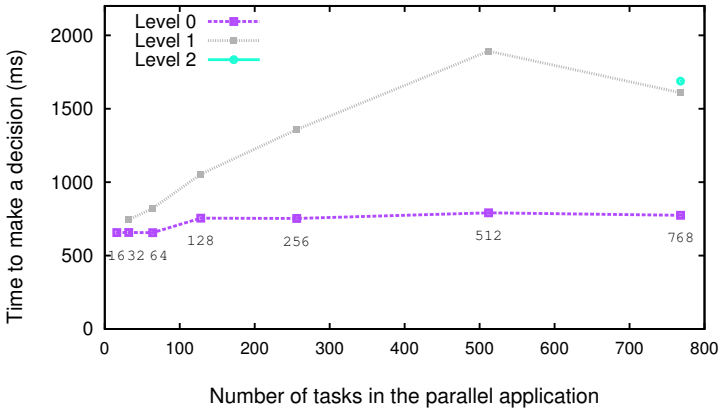


FIGURE 4.10: Scalability pattern with linear analysis time. The time required to make a decision is shown. The time is measured for each level in the hierarchy.

At level 1, the analysis and tuning domain of the single ATM, which is presented with a number of application tasks from 32 to 512, is composed of all *Asbtractor*-ATM pairs located at level 0. For this reason, the time to make a global decision grows by an amount proportional to the size of the analysis and tuning domain at this level in the hierarchy.

The described growth results in the time required to make a global decision for a parallel application composed of 512 tasks, being only double the time required for one composed of 64 tasks. This depicts a global decision time that appears to grow logarithmically with respect to the number of tasks in the application.

In the case of a parallel application with 768 tasks, the tuning network requires a third level in the hierarchy. It can be seen that the global decision time decreases with respect to the case of 512 tasks. This occurs because the analysis time increases linearly with the size of the analysis and tuning domain, and the two level 1 ATMs, which are working in parallel, each have a smaller analysis and tuning domain than the root ATM for 512 tasks. Meanwhile, the new root ATM (at level 2) only has these two *Abstractor*-ATM pairs in its own analysis and tuning domain.

4.4.3 Quadratic Analysis Time

In many cases, the analysis time increases quadratically with respect to the number of tasks being analysed. In a centralised tuning network, the single ATM would quickly become a bottleneck, making a non-centralised tuning network even more important.

The same parameters are used for this case as in the linear experiment, with the following exception:

- f_e of 10 events per second per simulated task.

- E_a of 10, a performance problem detection phase is performed every 10 batches of events.
- E_c of 1, a single new event is sent to the level above in the hierarchy every batch of events, as such $f_{rc} = f_e$ at all levels.
- Quadratic $T_a(N)$ of $12ms$, i.e. the performance problems detection phase lasts $N^2 \cdot 12ms$, where N is the size of the analysis and tuning domain of the ATM.
- T_m of $1ms$.
- T_c and T_t of $0.1ms$.
- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds

The maximum domain size is calculated in order to determine the topologies of the tuning networks for each application size. Since the analysis time is quadratic, N_{max} can be found using Expression 3.2 and the quadratic formula.

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$

$$N_{max} = \left\lfloor \frac{-\left(10 \cdot \frac{1}{1000}\right) \pm \sqrt{\left(10 \cdot \frac{1}{1000}\right)^2 - 4 \cdot \left(\frac{12}{1000}\right) \cdot \left(10 \cdot \frac{0.1}{1000} + \frac{0.1}{1000} \cdot 0.1 - 1\right)}}{2 \cdot \left(\frac{12}{1000}\right)} \right\rfloor$$

$$N_{max} = 8$$

The details of each topology calculated for each size of the parallel application are given in Table 4.5. In this case, the size of the parallel application only reaches 512 tasks, this is because the combined number of nodes required for the application and the hierarchy in the case of 768 tasks exceeds the resources available in the MareNostrum supercomputer.

Figure 4.11 presents the average time required to make a decision for each application size and for each level in the hierarchy.

In this case, the size of the analysis and tuning domain that each ATM can manage is considerably smaller than in the previous linear experiment. As such, an application

TABLE 4.5: Tuning network topologies for quadratic analysis time for parallel applications composed of different numbers of tasks.

# Tasks of the parallel application	Level 0		Level 1		Level 2	
	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size	#*Abstractor*-ATM pairs	Domain size
16	2	8	1	2	-	-
32	4	8	1	4	-	-
64	8	8	1	8	-	-
128	16	8	2	8	1	2
256	32	8	4	8	1	4
512	64	8	8	8	1	8

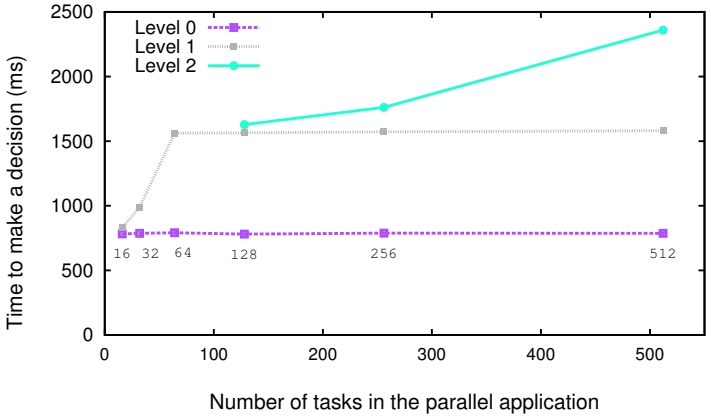


FIGURE 4.11: Scalability pattern with quadratic analysis time. The time required to make a decision is shown. The time is measured for each level in the hierarchy.

with 16 tasks already requires a tuning network composed of two levels. For the same reason, an application size of 128 tasks already requires a three level hierarchy.

The quadratic behaviour of the analysis time means that the global decision time increases substantially faster while the tuning network has the same number of levels, for example from 128 tasks to 512 tasks. However, as levels are added to the hierarchy, the global decision time presents the same logarithmic growth as in the experiment with linear analysis time.

4.4.4 Multi-environment Validation

In order to verify the independence of the results from the underlying execution environment we have performed the same experiment on two different supercomputer architectures. Taking advantage of the additional nodes available on SuperMUC, the experiment has also been extended to a larger parallel application size.

The characteristics of the performance and abstraction model simulated by the prototype are the following:

- f_e of 10 events per second per simulated task.
- E_a of 10, a performance problem detection phase is performed every 10 batches of events.
- E_c of 1, a single new event is sent to the level above in the hierarchy every batch of events, as such $f_{rc} = f_e$ at all levels.
- Linear $T_a(N)$ of $30ms$, i.e. the performance problems detection phase lasts $30ms$ per node within the analysis and tuning domain of the ATM.
- T_m of $1.5ms$.
- T_c and T_t of $0.1ms$.

- f_{rp} of $f_e/(10 \cdot E_a)$, a tuning action is received every 10 seconds

All levels in the hierarchy will present the same maximum domain size ($E_c = 1$). As in the previous sections, the value of this maximum domain size, N_{max}, is calculated using Expression 3.2

$$\frac{1}{f_a} \geq N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp}$$

$$N_{max} = \left\lfloor \frac{1 - 10 \cdot \frac{0.1}{1000} - \frac{0.1}{1000} \cdot 0.1}{10 \cdot \frac{1.5}{1000} + \frac{30}{1000}} \right\rfloor$$

$$N_{max} = 22$$

Using N_{max} with value equal to 22, tuning networks topologies for each application size has been calculated. The details of the topologies used on both MareNostrum and SuperMUC are shown in Table 4.6.

TABLE 4.6: Tuning network topologies for parallel applications composed of different number of tasks.

# Tasks of the parallel application	Level 0		Level 1		Level 2	
	#Abstractor-ATM pairs	Domain size	*#Abstractor*-ATM pairs	Domain size	*#Abstractor*-ATM pairs	Domain size
Tuning Networks executed in MareNostrum and SuperMUC						
25	2	13	1	2	-	-
50	3	16	1	3	-	-
100	5	20	1	5	-	-
200	10	20	1	10	-	-
400	19	22	1	19	-	-
800	37	22	2	19	1	2
Tuning Networks executed in SuperMUC						
1600	73	22	4	20	1	4
3200	146	22	7	21	1	7
6400	292	22	14	21	1	14

Figure 4.12 shows the average decision time for each level in the hierarchy from 25 to 800 application tasks in both MareNostrum and SuperMUC. The differences between the time measured in each system were less than 0.8%, and as such can be considered negligible. This figure shows that the time to make a decision at level 0 rapidly reaches a ceiling, when the ATMs are controlling the maximum number of tasks. This situation occurs because, as the parallel application grows, additional ATMs are simply added. The decision time at level 1 increases proportionally to the size of the analysis and tuning domain at this level in the hierarchy. When the application reaches 800 tasks, a new level is required. However, in this last experiment, since the two ATMs at level 1 are working in parallel, the global decision time is only slightly larger than for 400 application tasks. This small increase from 400 to 800 tasks highlights the key benefit of the hierarchical architecture.

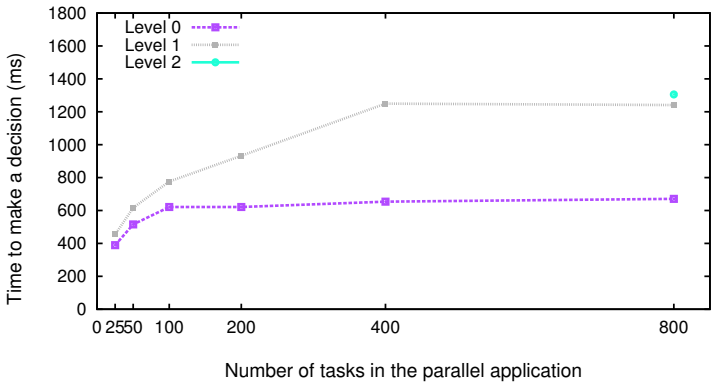


FIGURE 4.12: Scalability at MareNostrum. The time required to make a decision is shown. The time is measured for each level in the hierarchy.

The experiment was continued in SuperMUC up to 6400 parallel application tasks. The average decision times for each level are shown in Figure 4.13.

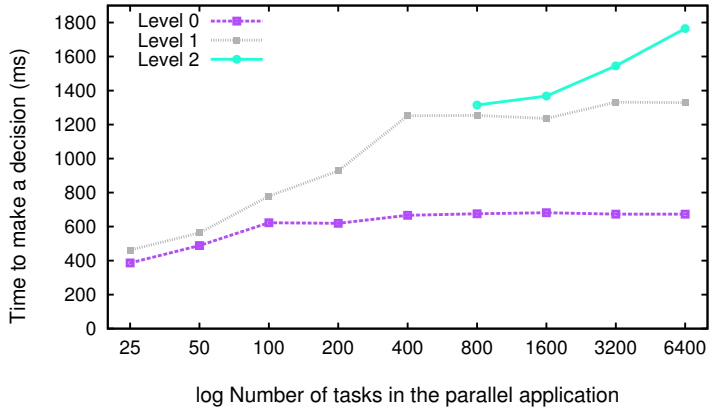


FIGURE 4.13: Scalability at SuperMUC. The time required to make a decision is shown. The time is measured for each level in the hierarchy.

This experiment demonstrates that the pattern of the previously discussed results continues as the application grows. Given that all the ATMs work in parallel, except for the root, the overall increase in global decision time grows logarithmically as the number of application tasks increases and new levels are filled up or added to the hierarchy.

4.5 Conclusions

In this chapter a validation of the scalability of the hierarchical tuning model proposed in this thesis has been presented.

To perform this validation, a simulation environment that reproduces the behaviour of a tuning network has been used. This has allowed an evaluation independent of the properties of a specific tuning strategy to be carried out.

The first step to validate the scalability, was to verify the correctness of the method to calculate an efficient topology, which was presented in Section 3.5. In order to achieve this, an experimental evaluation has been performed using two distinct use cases which demonstrate that the proposed method is able to adapt to different classes of analysis and tuning processes. In all cases, the method selects hierarchical tuning networks characterised by providing a high quality analysis and tuning environment, using the minimum number of resources required to do so.

In validating the scalability of the model, it has been demonstrated that irrespective of the analysis time pattern and the underlying execution environment, the global decision time presents a logarithmic growth with the size of the parallel application. Moreover, while the parameters that define the performance and *abstraction* model will change the values of the results observed, the underlying behaviour will remain. From this behaviour arises the scalable properties of the proposed hierarchical dynamic tuning model, which does not depend upon the specific characteristics of the application and the performance and *abstraction* models' parameters.

5

ELASTIC

"Elastic: able to encompass much variety and change; flexible and adaptable."

– Oxford Dictionary

In this chapter we present ELASTIC, a tool for large-scale tuning. ELASTIC implements the hierarchical model for dynamic tuning presented in Chapter 3. First, we introduce the motivation and goals of our tool. Subsequently, functional and design requirements of ELASTIC are presented. Finally, ELASTIC is described and its architecture and its main components are detailed.

5.1 Motivation and Goals

In order to bring dynamic performance analysis and tuning to high performance computers, in this thesis we have presented a model for scalable dynamic tuning. This model is based on a hierarchical tuning network of analysis modules, combined with an abstraction model, which will allow for scalable analysis and tuning through a combination of local and global improvements.

To complete the development of this model, it is necessary to implement the conceptual design of the model in a real tool that enables the dynamic improvement of the performance of large-scale parallel applications.

Several performance tools have been studied that provide, by employing different approaches, dynamic tuning of parallel applications. The operation of these tools is focused on tuning medium scale applications, and does not extend to parallel applications running on tens of thousands of processors, a situation which is becoming increasingly common. Currently, there is no tool that, operating on large-scale systems, provides continuous monitoring, analysis and tuning phases in a dynamic manner.

Our goal is to develop a tuning tool, based on the model proposed in this thesis, that is able to scale to provide effective hierarchical dynamic tuning on large-scale parallel applications. The tool must be able to collect and utilise the large amounts of data involved in the analysis process as well as being able to effectively "distribute" instructions to modify the parallel application to improve its performance. This must all be possible, while allowing user defined performance and abstraction models to drive the analysis and tuning process, an important part of a general purpose tuning tool.

5.2 Functional and Design Requirements

The goal of the tuning tool is to improve a parallel application's performance, minimise its execution time and exploit the potential of the execution system. Moreover, the tuning tool has to be scalable in order to operate with parallel applications composed of tens of thousands of tasks. In order to improve the application at runtime, as well as to be efficient, useful and easy to extend, the tuning tool must consider several functional requirements that will be described in the following sections.

Hierarchical Architecture

Following the model for hierarchical dynamic tuning presented in this thesis, the tuning tool has to fit a hierarchical structure. It should be composed of a front-end process

connected to a number of back-end processes via a hierarchical tuning network composed of analysis modules. The communication established in the tuning network must present mechanisms that can efficiently transport the necessary data between levels in the hierarchy during the performance analysis and tuning process. It must be possible to configure the topology of the hierarchical tuning network, depending on the size and structure of the underlying parallel application.

The MRNet framework has been chosen to meet these requirements, it will be described in Section 5.3.2.

Control of the Application Lifecycle

In order to perform dynamic performance improvement, the tuning tool must be able to control the entire execution of the application, from start to finish. The tool must launch the parallel application and then connect to it, allowing dynamic instrumentation. It must also be able to detect when the parallel application terminates its execution.

This is achieved by the back-ends of the tuning tool, each of which controls a single parallel application task.

Automatic and Dynamic Improvement of the Parallel Application

The primary purpose of the proposed tuning tool is the automatic and dynamic tuning of a parallel application. This process can be divided into three phases: monitoring, analysis and tuning.

The performance monitoring phase requires the instrumentation of the parallel application and the collection of the resulting monitoring data. These actions must be performed automatically while the application is running.

The insertion of the monitoring code in the application is carried out by the back-ends of the tuning tool using DynInst, a tool that will be explained in more detail in Section 5.3.1. This monitoring code is constructed using information sent by an analysis module (Instrumentation Order for Monitoring) that indicates points in the application that have to be measured to gather information about its behaviour.

Following the conceptual design of the hierarchical tuning model proposed in this work, the dynamic monitoring is based on event tracing. Event tracing makes use of events to obtain information about the state of the application. Each event has to contain information about 1) what action occurred, 2) where this action occurred in the application execution, 3) when this action occurred, and 4) additional information related to the specific action.

So, the instrumentation for monitoring must be able to generate events containing information about the state of the application. This information will be injected into the tuning network by the back-ends to be received by analysis modules. Using this information, the analysis modules are able to analyse the performance state of the application.

In accordance with the proposed hierarchical tuning model, the performance analysis process is guided by the performance and abstraction models. The tuning tool must enable a user to load the necessary codified knowledge to guide this process into the analysis modules. This must be done in such a way that no user intervention is required during the dynamic tuning process.

In order to implement the proposed changes in the application as a result of the analysis phase, the analysis modules must be able to communicate these changes, in the form of Instrumentation Orders for Tuning, through the tuning network to the back-ends, which must then instrument their application tasks. These Tuning Orders must contain all the required information, specifically which task is to be tuned, what must be changed (the tuning action), where the change must occur in the application (the tuning point), and at which moment the change should be applied (synchronisation). Applying the chosen solution is performed by means of the DynInst library, since this operation must be done during runtime, without recompiling or restarting the parallel application.

Intrusion Reduction

A primary concern of any tuning tool is maintaining the intrusion on the parallel application to a minimum. This ensures that the benefits of the performance tuning are not outweighed by the overhead created by the presence of the tuning tool.

In our dynamic tuning process, there are several possible sources of intrusion.

One source is in the event tracing mechanism. Since event tracing can produce large amounts of data, it can be reasonably invasive, and should therefore only be active when necessary. For this reason, using the capacity of dynamic instrumentation, our tuning tool is able to add and remove monitoring code during the execution of the application. This allows it to control the intrusion originating from event tracing.

However it should be noted that dynamic instrumentation is also a source of intrusion. This includes the instrumentation performed to add and remove monitoring code, as well as the instrumentation to change the application to improve its performance. The overhead exists because the instrumentation involves stopping the execution of the

application task to be instrumented while its code is modified. To mitigate this intrusion, it is important to keep the instrumentation as simple as possible.

The final source of intrusion is in the overlying tuning network. To reduce this intrusion, the front-end and analysis modules of the tuning tool should be located in different machines to those used by the application. Due to functional requirements, the back-ends must be running together with their associated parallel application tasks. For this reason, the back-ends should present a light-weight design.

Target environment and portability

Currently, the great majority of large-scale scientific parallel applications are implemented using the Message Passing Interface (MPI) standard to perform interprocess communication. With the aim of offering performance analysis over these kinds of applications, the tuning tool should support MPI based parallel applications.

In a similar vein, most large-scale parallel computers run UNIX operating systems (over 90% of the Top 500 supercomputer list run a Linux based operating system [59]). As our main objective is to propose a model for dynamic tuning of applications running on these kinds of large-scale systems, our target environment is UNIX based supercomputers.

It should be noted that the portability of our tuning tool is dependant on the constraints of some of the technologies used in its development. On one hand, an MPI environment is necessary as mentioned above. On the other hand, the technologies used by the tuning tool have certain requirements, which must be met in the target system. These are:

- Multiple threads per core. In order to make efficient use of resources no additional cores are allocated for the tuning tool back-ends. For this reason, multiple threads are used in each core, one for the application task and one or more for the back-end. Additionally, the internal nodes of the tuning network use multiple threads to manage communication.
- Socket based communication. Both the event collection in the back-ends and the hierarchical communication utilise sockets.
- ELF or DWARF binary format. The dynamic instrumentation library is only capable of reading and writing binaries which have been compiled in either of these formats.

5.3 Employed Technologies

There are two key technologies required for the implementation of the hierarchical tuning network for which mature solutions already exist. These are dynamic instrumentation and efficient hierarchical communication.

The dynamic instrumentation of the parallel application carried out by the tuning tool presented in this chapter is performed using the DynInst library. DynInst is a dynamic instrumentation library that provides a machine independent interface to permit the creation of tools and applications that use runtime code patching. Its potential has been exploited in a broad range of uses, such as performance measurement and debugging tools, execution drive simulations and computational steering.

In the case of hierarchical communication, the MRNet framework has proven to be an efficient communication substrate in large-scale systems. Its use has provided positive results in areas like clock synchronisation, equivalence classification, time-aligned data aggregation, performance analysis and debugging. As MRNet meets the requirements of the hierarchical communication employed by the tuning network, it has been used for the development of the tuning tool presented in this chapter.

The following sections describe the main aspects of these two technologies.

5.3.1 Dynamic Instrumentation via DynInst

The main idea of dynamic instrumentation is to postpone the application instrumentation until it is running and insert, change or remove this instrumentation dynamically during the application execution. DynInst is a post-compiler program manipulation tool that allows this dynamic instrumentation. Using this library, it is possible to instrument and modify applications during execution [10].

The DynInst API is based on object-oriented technology and provides a set of classes and methods that support the programmer in the process of developing an application that will instrument another application during runtime. Specifically, DynInst allows the programmer to:

- Modify a running application or to start a new one.
- Create a new piece of code and insert it into an executing application.
- Access and use existing code and data structures.
- Remove inserted code from a running application.

In order to modify a running application, DynInst manipulates the address-space image of the application. For this reason, this library only needs access to the running application, not its source code. Therefore, a running application modified using DynInst does not need to be re-compiled, re-linked, or re-started in order to be able to continue running. However, it should be noted that for DynInst to modify an application at runtime, the application must be compiled with the option that enables debug information. DynInst needs this information to locate functions, data structures and variables in the instrumented application.

The goal of this API is to provide a machine independent interface to permit the creation of tools and applications that use runtime code patching and work on different types of execution environments. The last version of DynInst, 8.1.1, supports the following platforms: Tru64 Unix on the Alpha processor, Linux 2.4/2.6 on Intel x86, IA-64, or AMD-64 based processors, Windows XP/2000/2003 on Intel x86 based processors, AIX Version 5.1, and Solaris 2.9 on SPARC processors.

The DynInst API is based on several abstractions with interactions, as shown in Figure 5.1, that permit the modelling of the components necessary in the dynamic instrumentation mechanism:

- **Mutatee**. The application to be instrumented.
- **Mutator**. A separate application which controls the mutatee and instruments it via DynInst.
- **Thread**. A thread corresponding to the application in execution.
- **Image**. The static representation of the application on a disk. Each thread is associated with an image.
- **Point**. A specific place in the application where the instrumentation could be inserted. For instance, this place could represent a function entry, function exit, a loop, etc.
- **Snippet**. The representation of a piece of executable code to be inserted into the application at a point.

To dynamically instrument a parallel application via DynInst, several steps have to be performed during the development phase and during the execution time phase.

During the development phase, the programmer has to check that:

- A mutatee, compiled with debug information, is available.
- The mutator implements snippets using the DynInst API.
- The mutator has been compiled and linked with DynInst Library.
- The mutator can be launched.

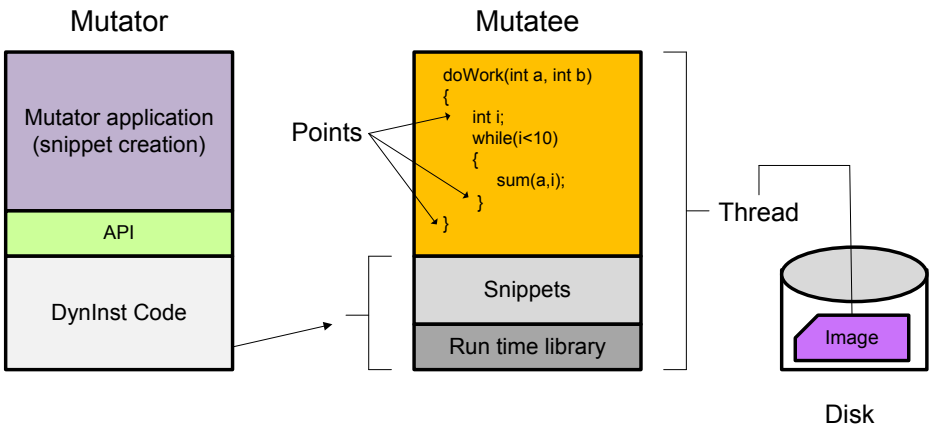


FIGURE 5.1: Abstractions used in DynInst.

Secondly, at runtime, DynInst is responsible for:

- Dynamically loading the DynInst Library into the address space of the mutator.
- Using the DynInst API, the mutator can create the mutatee.
- Automatically attaching its runtime library to the address of the mutatee.
- Inserting calls to the snippet code at specific points in the mutatee.

The tuning tool presented in this chapter uses DynInst to dynamically instrument the application. The tuning tool acts as the mutator and the parallel application acts as the mutatee. Snippets and points depend on the information necessary to evaluate the behaviour of the application which is determined by the performance model. Snippets are the pieces of software used to obtain the information and make the events, which have to be inserted in specific points of the parallel application.

5.3.2 MRNet

MRNet [51] is a communication software infrastructure for parallel tools and applications with a master/slave architecture. MRNet acts to reduce the cost of centralised tools' activities by incorporating a hierarchical network of processes (tree-based overlay network-TBON) between the tool's front-end and back-ends, as is shown in Figure 5.2. MRNet uses these *internal processes* to distribute many important tool communication and computation activities.

To build scalable tools, the major characteristics that MRNet offers are:

- Flexible organisation of the TBON of internal processes. The organisation is determined by a configuration file.
- Scalable data aggregation provided by the internal nodes of the network.

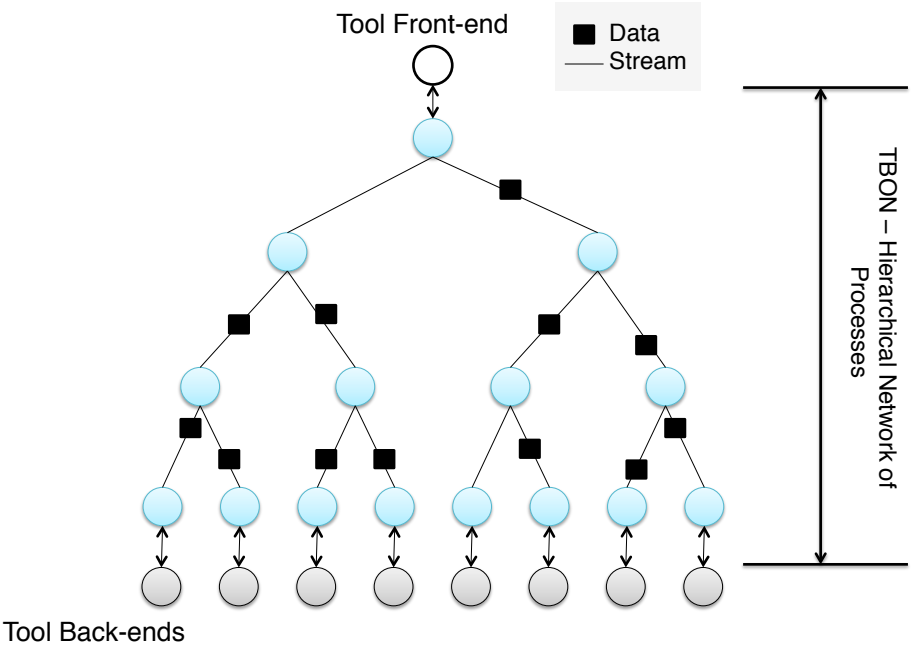


FIGURE 5.2: Typical architecture of a tool built using MRNet.

- Scalable and efficient multicast communication, reducing the communication cost between the front-end and all its back-ends.
- Multiple and concurrent communication data channel between tool components.

As we explained in Chapter 2, MRNet has been integrated into several existing tools in order to provide scalable properties, such as Paradyn [50] and TAU [46]. Moreover, other new tools, such as STAT [2] and TBON-FS [6], have made direct use of MRNet as an scalable platform.

The last version of MRNet, 4.0.0, has been successfully tested on the following platforms: x86, x86_64, powerpc32 and powerpc64 for Linux, and x86 for Windows.

MRNet is based on two main components: the *library API* and the *mrnet_commnode* program. Tool's front-end and back-ends are linked to the library API in order to establish communication between them via MRNet. The mrnet_commnode program is run on the internal nodes between the front-end and the back-ends, enabling distributed data processing and implementing scalable group communications.

MRNet is based on the following abstractions:

- **End-points**. The tool's back-ends processes, i.e. the leaves of the tree network.
- **Communicators**. Groups of end-points. Using communicators, the front-end can communicate with back-ends in unicast or multicast mode.

- **Streams**. Logical communication channels that connect the front-end to the end-points of a specific communicator. Streams transport data packets downstream, from the front-end to the back-ends, and upstream, from the back-ends to the front-end.
- **Filters**. The functions located at the internal processes of the network, to synchronise and aggregate data packets sent to the front-end. Filters are defined as a dynamic function that is bound to a stream during the stream creation. In MRNet, there are two types of filters: synchronisation filters and transformation filters. Synchronisation filters organise upstream data packets into synchronised waves of data packets, while transformation filters operate on the synchronised upstream or downstream data packets, generating one or more output packets. MRNet offers some general purpose filters as well as facilities for adding user-defined filters to perform tool-specific aggregation operation.

It should be noted that in order to use MRNet, the front-end and the back-ends have to be compiled and linked with the MRNet library. If the filter used is user-defined, it must be compiled into a valid shared object, and front-end and back-ends which dynamically load these kinds of filters must be built with compiler options that notify the linker to export global symbols.

In the proposed model for hierarchical tuning, the instrumentation orders and the events containing behavioural information about the application must flow throughout the hierarchical communication network. To implement this kind of communication, the tool presented in this chapter employs the MRNet framework. Also, the strength of the filters has been used to place the *Abstractor*-ATM pair at the internal processes of the MRNet TBON. The manner in which the tuning tool presented in this work uses MRNet is detailed in Section 5.4.7.

5.4 ELASTIC

The proposed model for hierarchical dynamic tuning presented in Chapter 3 has been implemented as a performance analysis tool called **ELASTIC**.

ELASTIC provides dynamic performance tuning of large-scale parallel applications. The steering of the application is based on three phases: performance monitoring, performance analysis and tuning. These phases are carried out automatically and continuously by ELASTIC while the parallel application is running on large-scale systems.

5.4.1 Architecture

ELASTIC is composed of the following components which cooperate to control and to improve the execution of the application:

- **ELASTIC Front-End (EFE)**. The root process of the network, responsible for creating and instantiating the internal processes of the tuning network.
- ***Abstractor*-ATM pair**. Located at the internal nodes of the tuning network, they carry out the performance analysis and tuning of the parallel 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 individual task of the parallel application, 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 application task, to support monitoring and behavioural data gathering.

The knowledge required to perform dynamic tuning in ELASTIC is built into a set of code and configuration called an **ELASTIC Package**. To conduct the hierarchical dynamic tuning proposed in this thesis, this package must contain information that (1) guides the performance analysis and tuning process and (2) determines how to transform information between different levels in the tuning network. As was detailed in Section 3.4.3, such required knowledge takes the form of a performance model and abstraction model respectively.

Figure 5.3 shows the ELASTIC architecture. The hierarchical communication layer is established through MRNet. This framework allows for the connection of the ELASTIC front-end with ELASTIC back-ends as well as enabling the *Abstractor*-ATM pairs to be distributed across MRNet's TBON of internal processes, utilising the strength of filters. The details of how ELASTIC takes advantage of the features of MRNet can be found in Section 5.4.7.

Each of the ELASTIC components are explained in detail in the following sections. Their design, functionality and interfaces are presented.

5.4.2 ELASTIC Front-End

The front-end is the root process of the ELASTIC architecture. Its main responsibilities are the initialisation and finalisation of the ELASTIC infrastructure. Specifically, the EFE is in charge of the following actions:

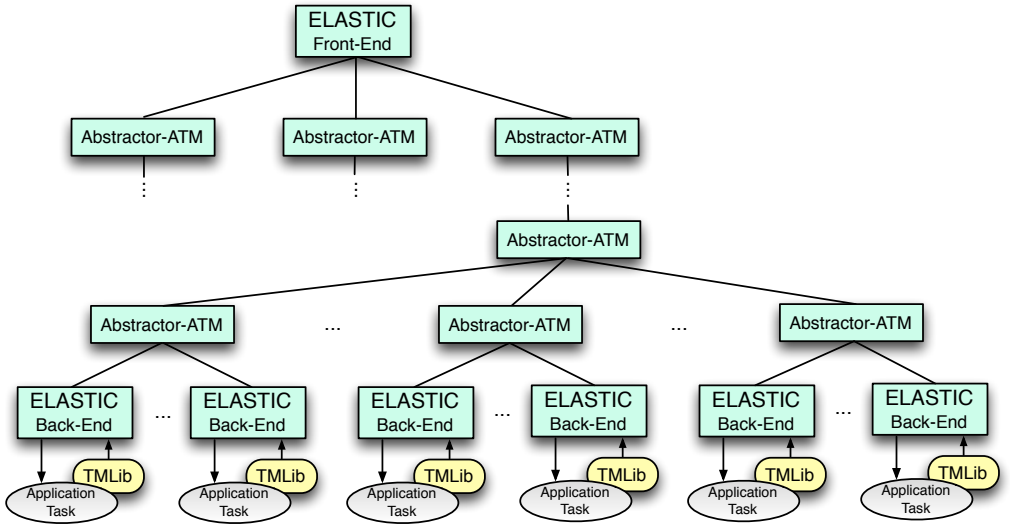


FIGURE 5.3: ELASTIC architecture.

- Instantiate the hierarchical tuning network.
- Control the correct connection of EBEs.
- Create the communication channels between it and EBEs.
- Load the functionality of the *Abstractor*-ATM pair in each internal node of the hierarchy.
- Initialise the performance analysis and tuning.
- Delete the tuning network when the execution of the parallel application has finished.

To conduct all these actions, the EFE uses the MRNet API. The details of how these processes are performed will be detailed in Section 5.4.7.

5.4.3 *Abstractor*-ATM Pair

The *Abstractor*-ATM pair is the fundamental component of the ELASTIC architecture. It enables a distributed dynamic performance analysis and tuning process through the tuning network. Each *Abstractor*-ATM pair is located at an internal node of MRNet's TBON architecture.

The dynamic tuning cycle begins with the monitoring order generation carried out by the ATMs at each level in the hierarchy. The ATM sends these orders to its children, i.e. the nodes of its analysis and tuning domain. In the case that these nodes are other *Abstractor*-ATM pairs, the *Abstractor* in the child node is responsible for translating the received monitoring orders and continuing their propagation downstream through the tuning network. When these orders arrive at an ELASTIC back-end, they are applied to the application task.

Still in the monitoring phase, events are sent from the application task upstream to the *Abstractor*-ATM pairs. These events can be used for an analysis process or, if they are destined for an *Abstractor*-ATM pair at a higher level, the *Abstractor* will transform and send them to the parent level in the hierarchy.

If, as a result of the analysis process, tuning is required, an instrumentation order will be sent by the *Abstractor*-ATM pair to the necessary children in the analysis and tuning domain. During this tuning phase, the orders received by an *Abstractor*-ATM pair from the parent level will be translated to be applied to the child level, until these orders reach the ELASTIC back-ends.

Based on the hierarchical dynamic tuning model presented in Chapter 3, the *Abstractor*-ATM pair is composed of several cooperating modules, shown in Figure 5.4.

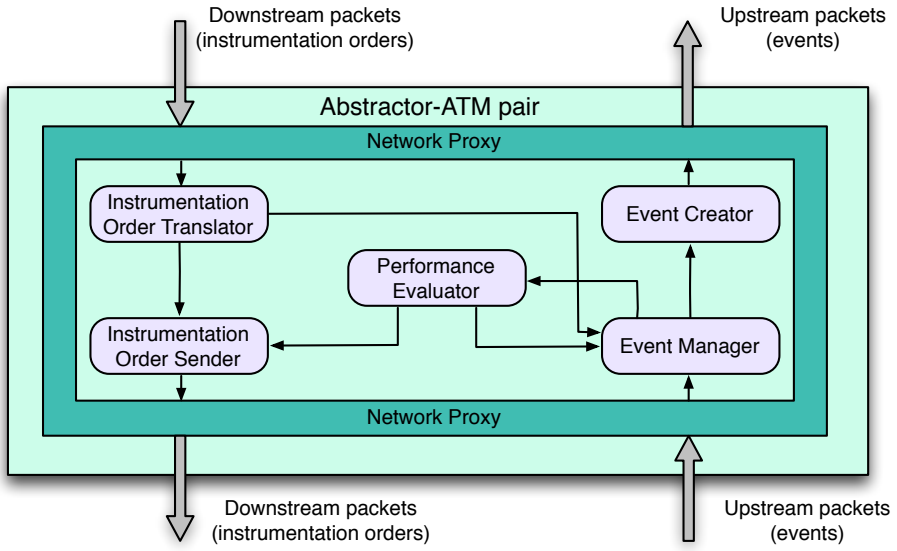


FIGURE 5.4: Internal architecture of the *Abstractor*-ATM pair.

The functionality of each of the components presented in Figure 5.4 is implemented as a C++ class.

As mentioned in Section 3.4.3, to carry out the proposed hierarchically distributed performance analysis and tuning process, the user has to provided the knowledge related to the performance model and the abstraction model. The integration of this knowledge in ELASTIC is provided in the form of concrete subclasses of the C++ classes which implement the *Abstractor*-ATM pair components. The collection of these subclasses and some additional configuration options form the ELASTIC Package, the mechanism through which the user interacts with ELASTIC. From within these subclasses, the user will have access to the public properties and methods of each of the described classes, which form the ***Tuning and Abstraction API***.

The functional description of each of the main components of the *Abstractor*-ATM pair is presented below, followed by the specification of the *Tuning and Abstraction API*.

Functional Component Description

Performance Evaluator. This module is the core of the *Abstractor*-ATM pair because it directs the performance analysis and tuning process using the integrated knowledge. Beginning with the monitoring phase, the *Performance Evaluator* creates monitoring orders to request specific information required in the analysis phase. These orders must be registered with the *Event Manager* before being sent to the *Instrumentation Order Sender*. This is necessary so that when events containing the requested information are received by the *Event Manager*, they can be correctly routed to the *Performance Evaluator*.

When the *Performance Evaluator* receives an event it must store the information contained in this event. Having received events containing the complete information about the state of the analysis and tuning domain, the *Performance Evaluator* activates the analysis and tuning process. As a result of this process, a tuning order may be passed to the *Instrumentation Order Sender*.

Instrumentation Order Sender. This module works with instrumentation orders for monitoring and for tuning. In both cases it is responsible for converting their received orders into a format that can be sent across the network. The *Instrumentation Order Sender* must pack one or more orders into a serialised format for network transfer by the *Network Proxy*.

Instrumentation Order Translator. When instrumentation orders are sent by an *Abstractor*-ATM pair which is not in the base level of the hierarchy, it will need to be translated at each intermediate level to be applied coherently to the final parallel application tasks. Therefore, the *Instrumentation Order Translator* is responsible for this operation at each level of the tuning network. Based on user-provided knowledge, each instrumentation order for monitoring or tuning received, results in orders that can be applied to the level directly below. These orders are sent to the *Instrumentation Order Sender*.

In the case of monitoring orders, the *Instrumentation Order Translator* must register each type of order with the *Event Manager* to ensure that the corresponding generated events are passed to the *Event Creator* in this *Abstractor*-ATM pair.

Event Manager. During the monitoring phase the *Event Manager* handles the registration of the monitoring orders generated by the *Performance Evaluator* and the

Instrumentation Order Translator. The information from this registration is later used, when packets containing events coming from the analysis and tuning domain are received, in order to decide the destination of each event. An event received by this module will be passed to the *Performance Evaluator*, the *Event Creator* or both components.

Event Creator. This module is in charge of managing events received as a result of monitoring orders created by the *Instrumentation Order Translator* which resides in the same *Abstractor*-ATM pair. Using knowledge provided by the user, the events containing information about the analysis and tuning domain are *abstracted* and a new event is created and sent to the parent level in the tuning network.

To inject the newly created event into the tuning network, the *Event Creator* must serialise it for communication and pass the serialised representation to the *Network Proxy*.

Network Proxy. In order to abstract the communication with the tuning network via MRNet, the *Network Proxy* module implements network specific functionality. This module controls the connection of the *Abstractor*-ATM with the tuning network. Its capability resides in the conversion between the serialised versions of instrumentation orders and events, and the MRNet packets.

MRNet packets containing instrumentation orders are received as downstream packets, and new instrumentation orders are also sent as downstream packets. On the other hand, MRNet packets containing events are received as upstream packets, and likewise newly created events are sent as upstream packets.

Tuning and Abstraction API

This API encapsulates the aspects that implement the model for hierarchical dynamic tuning proposed in this thesis, which are located in the *Abstractor*-ATM pair. With this API, the components of the *Abstractor*-ATM pair are represented as objects of specific classes and the interactions between them are established.

In this section, the specification of the *Tuning and Abstraction API* is detailed.

- **PerformanceEvaluator Class.**
  Properties
  - `Domain *d` - representation of the analysis and tuning domain.
  - `InstrumentationOrderSender *ios` - the *Instrumentation Order Sender* instance used by this object.

- `EventManager *em` - the *Event Manager* instance used to register instrumentation orders for monitoring.

Methods

- `InitialMonitoringOrders` - called once, when the analysis and tuning process begins, should return at least one initial monitoring order.
- `HandleEvent` - called by the *Event Manager* for each new event.
- `NewEvent` - called to manage and store each new event, should return `true` if the complete information about the state of the analysis and tuning domain has been collected to initiate performance evaluation.
- `EvaluatePerformance` - called to evaluate the performance of the analysis and tuning domain, should return a vector of tuning and/or monitoring orders (if any) which will be sent to the *Instrumentation Order Sender*.

- **Event Manager Class.**
  Properties
  - `PerformanceEvaluator *pe` - the *Performance Evaluator* instance that receives events for performance evaluation in this *Abstractor*-ATM pair.
  - `EventCreator *ec` - the *Event Creator* instance that receives events for abstraction to a higher level in the tuning network.

  Methods
  - `ManageEvent` - called for each event that is received by this *Abstractor*-ATM pair. This event will be forwarded to either the *Performance Evaluator*, the *Event Creator* or both.
  - `RegisterEventForPerformanceEvaluator` - called by the *Performance Evaluator* to register a monitoring order. This instructs this object to forward the associated event to the *Performance Evaluator*.
  - `UnregisterEventForPerformanceEvaluator` - called by the *Performance Evaluator* to unregister a monitoring order.
  - `RegisterEventForEventCreator` - called by the *Instrumentation Order Translator* to register a monitoring order. This instructs this object to forward the associated event to the *Event Creator*.
  - `UnregisterEventForEventCreator` - called by the *Instrumentation Order Translator* to unregister a monitoring order.

- **InstrumentationOrderTranslator Class.**
  Properties
  - `Domain *d` - representation of the analysis and tuning domain.
  - `InstrumentationOrderSender *ios` - the *Instrumentation Order Sender* instance used by this object.
  - `EventManager *em` - the *Event Manager* instance used to register instrumentation orders for monitoring whose associated events will be sent to the *Event Creator*.

  Methods
  - `HandleOrder` - called when an instrumentation order for monitoring or tuning is received. It is responsible for sending the translated order via the *Instrumentation Order Sender* and registering monitoring orders with the *Event Manager*.
  - `TranslateMonitoringOrder` - called to translate an instrumentation order for monitoring, should return a vector of monitoring orders to be applied to the analysis and tuning domain.
  - `TranslateTuningOrder` - called to translate an instrumentation order for tuning, should return a vector of tuning orders to be applied to the analysis and tuning domain.

- **InstrumentationOrderSender Class.**
  Properties
  - `NetworkProxy *np` - the *Network Proxy* instance via which this object injects instrumentation order packets into the tuning network.

  Methods
  - `SendTuningOrder` - send an instrumentation order for tuning via the *Network Proxy*.
  - `SendMonitoringOrder` - send an instrumentation order for monitoring via the *Network Proxy*.

- **EventCreator Class.**
  Properties
  - `NetworkProxy *np` - the *Network Proxy* instance via which this object injects event packets into the tuning network.

Methods

- `HandleEvent` - called by the *Event Manager* for each new event. It is responsible for sending newly created events via the *Network Proxy*.
- `NewEvent` - called to manage and store each new event, should return `true` if the complete information about the state of the analysis and tuning domain has been collected to initiate the abstraction and event creation process.
- `CreateEvent` - called to create a new event by abstracting the information from previously received events.

- **NetworkProxy Class.**
  Properties

  - `EventManager *em` - all events received are sent to this *Event Manager* instance.
  - `InstrumentationOrderTranslator *iot` - all instrumentation orders received are sent to this *Instrumentation Order Translator* instance.

  Methods

  - `SendOrder` - called to inject a packet containing instrumentation orders into the tuning network, abstracting the MRNet implementation specifics.
  - `SendEvent` - called to inject a packet containing events into the tuning network, abstracting the MRNet implementation specifics.

As well as the classes that represent *Abstractor*-ATM components, there are classes which encapsulates the information contained within instrumentation orders for monitoring and tuning, and events. Instances of these classes are the objects which are passed between the functional components of the *Abstractor*-ATM pair and, as will be described in Section 5.4.4, are also used in the ELASTIC back-end.

- **Event Class.**
  Properties

  - `Timestamp ts` - represent the time when the event was generated at the application level.
  - `int eid` - represent the id of the event corresponding to the monitoring order which provoked its generation.
  - `int tid` - represent the rank of the task where the event was generated.

  - `vector<Value> v` - contains the values of the attributes requested by the monitoring order which provoked the generation of this event.

- **Order Class.**
  Properties

  - `int tid` - represent the rank of the task where the order has to be applied.

- **MonitoringOrder Class** ***(subclass of Order)***
  Properties

  - `int eid` - represent the id of the events which will be generated as a result of this order.
  - `int action` - determines whether this order is to add new monitoring or remove existing monitoring instrumentation, 0 - add, 1 - remove.
  - `string funcName` - the name of the function to be traced in the parallel application.
  - `int place` - point of instrumentation in the function, 1 - entry of the function, 0 - exit of the function.
  - `vector<string> attrs` - names of the variables to be read at the instrumentation point.

- **TuningOrder Class** ***(subclass of Order)***[1]
  Properties

  - `int type` - represent the type of the tuning order to be applied.
    0 - empty tuning order.
    1 - `SetVariableValue`.
    2 - `ReplaceFunction`.
    3 - `InsertFunctionCall`.
    4 - `OneTimeFunctionCall`.
    5 - `RemoveFunctionCall`.
    6 - `FunctionParamChange`.
    7 - `LoadLibrary`.

[1] To perform dynamic tuning the user works with subclasses of this class containing all the information required for each different type of tuning order. The subclass definitions are beyond the scope of this section, and can be found in Appendix A.

To show the primary sequences of interaction carried out by the classes described above, we will present a number of typical scenarios where these interactions occur.

The first scenario, depicted in Figure 5.5, shows the *Performance Evaluator* generating an order for monitoring. Once the order has been constructed inside the *Performance Evaluator*, it is registered with the *Event Manager* (1). It is then passed to the *Instrumentation Order Sender* (2), which in turn injects the order into the network via the *Network Proxy* (3).

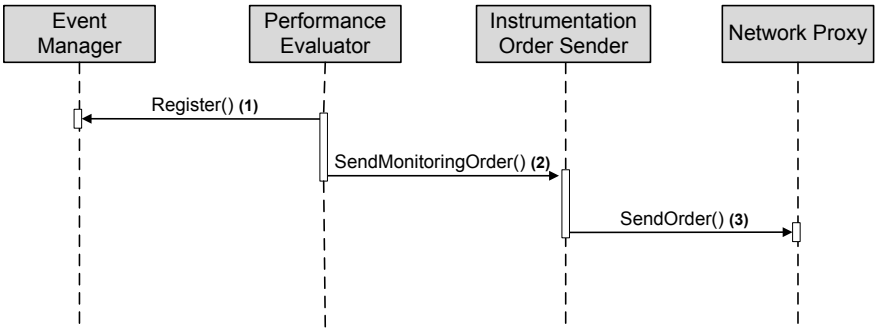


FIGURE 5.5: Sequence diagram: Generation of monitoring orders.

In Figure 5.6 the second scenario is presented. It represents the possible interactions triggered when an event is received by the *Abstractor*-ATM pair. The received event is transferred to the *Event Manager* (1), where it has two possible destinations. If the event is destined for this level in the hierarchy, then it is passed to the *Performance Evaluator* (2), or if the event is to be sent to a higher level in the hierarchy, then it is passed to the *Event Creator* (5). In the case of the *Performance Evaluator*, it is possible that the arrival of this event will invoke an analysis process resulting in a tuning order being generated. If this occurs the tuning order is sent to the *Instrumentation Order Sender* (3). Finally, the tuning order is injected into the network via the *Network Proxy* (4). In the second possible case (5), the event is sent to the *Event Creator*, which may result in a new event being generated to be sent to a higher level in the tuning network. This new event is injected into the network via the *Network Proxy* (6).

Finally, the third scenario, Figure 5.7 shows the interactions that occur when an instrumentation order for monitoring or tuning coming from the parent level is received by an *Abstractor*-ATM pair. The received order is transferred to the *Instrumentation Order Translator* (1). The orders resulting from the translation process are sent to the *Instrumentation Order Sender*, independent of whether they are orders for monitoring (2) or tuning (3). The *Instrumentation Order Sender* then injects the orders into the tuning network via the *Network Proxy* (4).

The manner in which the user exploits the capacity of this API to construct an ELASTIC Package is described in Section 5.4.6.

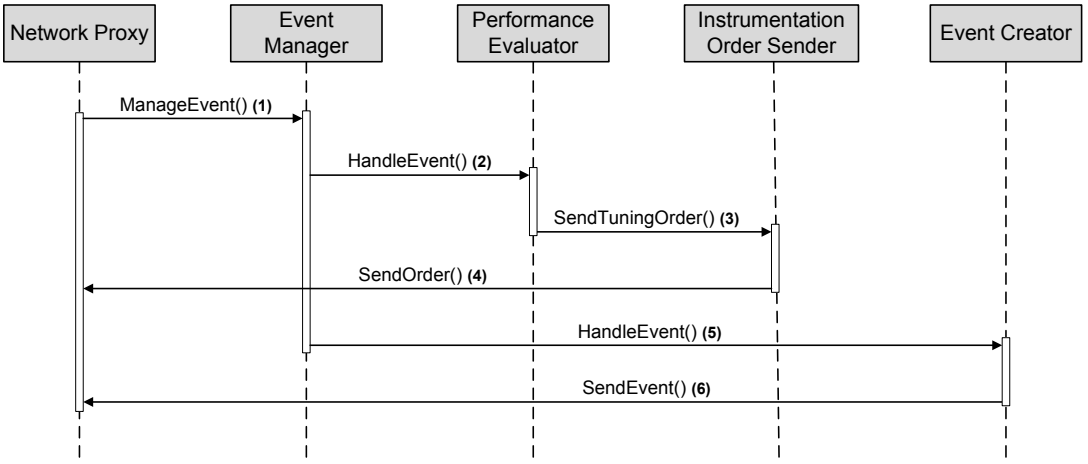


FIGURE 5.6: Sequence diagram: Event reception.

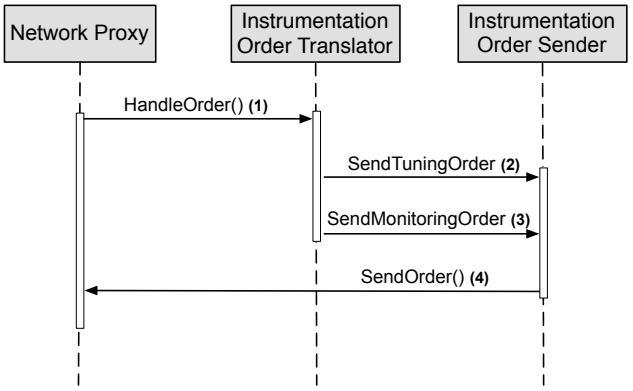


FIGURE 5.7: Sequence diagram: Translation of instrumentation orders.

5.4.4 ELASTIC Back-End

The ELASTIC back-ends are the leaf nodes of the ELASTIC tuning network. Each back-end is responsible for managing a task of the parallel application during the analysis and tuning process. Specifically, the actions of this module cover the following aspects:

- Launch and finalise the parallel application tasks.
- Load the Task Monitoring Library (TMLib) in the application task.
- Manage application instrumentation.
  - Receive instrumentation orders for monitoring or tuning from the tuning network.
  - Generate, insert and remove monitoring and tuning snippets.
- Manage events.
  - Receive events from the TMLib.
  - Inject these events into the tuning network.

The ELASTIC back-end is composed of a number of modules that cooperate to perform these actions. In Figure 5.8 the internal architecture of the ELASTIC back-end is depicted.

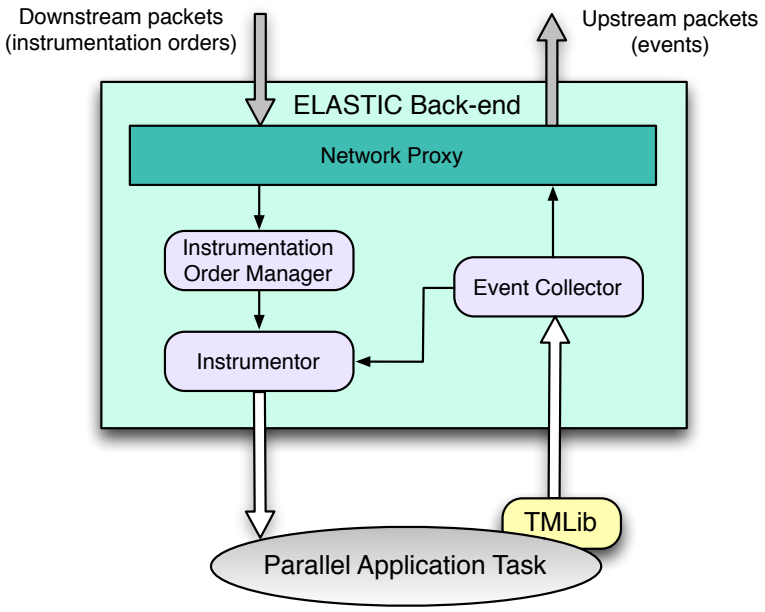


FIGURE 5.8: Internal architecture of the ELASTIC back-end.

Instrumentation Order Manager. This module receives a serialised representation of instrumentation orders from the tuning network. Here, the instrumentation order for monitoring or tuning is deserialised before being passed to the *Instrumentor*.

Instrumentor. This module is in charge of launching and terminating a single parallel application task. It must also perform the dynamic instrumentation for monitoring and tuning its application task.

When the back-end starts up, the *Instrumentor* creates the parallel application task and loads the TMLib in its memory space, using the DynInst API.

The monitoring in ELASTIC is based on the event tracing of function calls. During the course of the application execution, the *Instrumentor* receives monitoring orders from the *Abstractor*-ATM pairs of the tuning network. When these orders get to the *Instrumentor*, the DynInst library is used to dynamically insert instrumentation code (the snippet) into the parallel application task to generate events to be traced. The inserted snippet uses the TMLib to gather the information requested in the monitoring order by reading the values of global variables or function parameters at the monitoring point. These values are composed into an event and sent to the *Event Collector* as described in Section 5.4.5. It should be noted that the received monitoring orders can also request the removal of a previously inserted snippet.

Changes to improve the performance are dynamically made in the parallel application task, following instrumentation orders for tuning received, by the *Instrumentor*. The *Instrumentor* reads the information encapsulated in the tuning order and then uses the DynInst library to create a snippet which is inserted into the running application task.

Event Collector. This module facilitates the transfer of events from the TMLib to the tuning network. Events are sent to the *Event Collector* via a low-level event collection protocol based on TCP/IP. The events are then serialised for network transfer and passed to the *Network Proxy* to be injected into the tuning network.

Once the parallel application task has finished its main processing loop, a special event is sent to the *Event Collector* which then notifies the *Instrumentor* that it may terminate its task. Subsequently, the *Event Collector* sends the finalisation event to the tuning network and terminates.

Network Proxy. This module offers the same functionality as in the *Abstractor*-ATM pair, abstracting the low-level network interactions related to the sending and receiving of serialised events and instrumentation orders respectively in the form of MRNet packets.

5.4.5 Task Monitoring Library

The Task Monitoring Library (TMLib) is a library that provides support for event tracing of parallel application tasks. The TMLib is an adaptation of the Dynamic Monitoring Library created for use in the performance tool MATE [39]. The TMLib is loaded at runtime by the ELASTIC back-end into the memory address space of its associated task. As such, the TMLib is designed to be able to satisfy the types of requests for monitoring generated by the *Abstractor*-Analyzer pair.

The TMLib offers a set of functions that assist in three main parts of the monitoring process:

- **Initialisation**: The TMLib shared library is loaded into the task and a TCP/IP connection is establish with the *Event Collector*. This connection is used during the entire task execution. In this stage, it is also specified at which point in the task execution the finalisation of the TMLib occurs.
- **Event tracing**: During the execution of the parallel application, snippets are inserted into the application task by the *Instrumentor*, which call functions from the TMLib to catch events. These calls result in the values of variables or function parameters being collected and composed into events. The events are then sent

via the previously established connection to the *Event Collector* using a low-level event collection protocol. In this way, the TMLib facilitates the collection of the information requested by the *Abstractor*-ATM pairs in monitoring orders.

- **Finalisation**: When the application task reaches the predefined point, the TMLib finalisation process will be invoked. This involves sending a finalisation event to the *Event Collector*, freeing all the allocated resources and closing the TCP/IP connection to the *Event Collector*.

During the event tracing stage, the TMLib generates instances of the *Event* class detailed in Section 5.4.3. Following its properties, the event is filled out with a *timestamp* that indicates the time at which it was generated, an *event identifier* which indicates which monitoring order requested it, and the *task id* which contains the MPI rank of the parallel application task where it is generated. This is in addition to the variable values requested by the monitoring order.

5.4.6 ELASTIC Package

ELASTIC is a general purpose tuning tool which permits the resolution of a wide array of performance issues in large-scale parallel applications. To conduct performance analysis and tuning, user provided knowledge about the performance problems and possible solutions is required in the form of an ELASTIC Package.

As was detailed in Section 3.4.3, the required knowledge takes the form of a performance model, to guide the performance analysis and tuning process, and an abstraction model, to determine how to transform information as it moves between different levels in the tuning network.

The performance model is composed of a set of measurement points, a set of performance evaluation expressions, and a set of tuning actions. In ELASTIC, these three parts all reside in the *Performance Evaluator* module.

The information in the abstraction model is related to the translation of instrumentation orders for monitoring and for tuning which are received from the parent level in the hierarchy and the creation of events which abstract information about the state of the analysis and tuning domain of the *Abstractor*-ATM pair.

In order to integrate the performance and abstraction models into ELASTIC, the user must provide the codification of this knowledge in the form of the implementation of specific subclasses of the *Tuning and Abstraction API*. The classes which must be subclassed are the *Performance Evaluator* to implement the performance model, and the *Event Creator* and *Instrumentation Order Translator* to implement the abstraction model.

The implementation of these subclasses, together with some configuration information, forms an ELASTIC Package. In addition to these subclasses, the user may provide alternative implementation of other classes from the *Tuning and Abstraction API* in order to further customise the analysis and tuning process. The benefits of this plugin architecture will be discussed at the end of this section.

Performance and Abstraction Model

As previously mentioned, the user must provide a performance model in the form of a concrete implementation of a subclass of the *Performance Evaluator* class. In this subclass, three methods must be implemented:

```
vector<MonitoringOrder> PerformanceEvaluator::InitialMonitoringOrders()
```

In this method, the user must specify the initial measurement points of the performance model in terms of one or more `MonitoringOrder` objects. This set of monitoring orders form the output of this method.

```
bool PerformanceEvaluator::NewEvent(Event *e)
```

The storage of each new event should be performed in this method. The implementation must be able to recognise when all the events required to activate a performance evaluation have been received, in which case the output of this method should be `true` - otherwise `false`.

```
vector<Order> PerformanceEvaluator::EvaluatePerformance()
```

This method will be called when a call to NewEvent() results in a true response.

The user must implement the performance problem detection process, which makes use of the information contained in the previously collected events. In the case that a performance problem is detected, the appropriate `TuningOrder` objects should be created, based on the tuning points and actions of the performance model. The set of tuning orders form the output of this method, or an empty set if no tuning is necessary.

It is also possible that the monitoring process requires more or less detail about the state of the parallel application. In this case, the set of orders that are the output of this method can also contain monitoring orders.

The implementation of the abstraction model is distributed between two classes, the *Instrumentation Order Translator* and the *Event Creator*. Their respective methods, which must be overwritten, are detailed below:

```
vector<MonitoringOrder> InstrumentationOrderTranslator::
                        TranslateMonitoringOrder(MonitoringOrder *mo)
```

This method hosts the functionality necessary to convert monitoring orders from the parent level, to be applied to the level below in such a manner that the events generated as a result of these orders, satisfy the requirements of the original monitoring order. The output from this method should be a set of new `MonitoringOrder` objects which will be sent to the analysis and tuning domain.

```
vector<TuningOrder> InstrumentationOrderTranslator::
                    TranslateTuningOrder(TuningOrder *to)
```

This method contains the knowledge necessary to convert tuning orders from the parent level, to be applied to the level below. These new tuning orders, must provoke the equivalent change in the analysis and tuning domain as was requested in the original tuning order. The output from this method should be a set of new `TuningOrder` objects.

```
bool EventCreator::NewEvent(Event *e)
```

The storage of each new event should be performed in this method. The implementation must recognise when all the events required to create a new event to be sent to the parent level have been received, in which case the output of this method should be `true` - otherwise `false`.

```
vector<Event> EventCreator::CreateEvent()
```

This method will be called when a call to NewEvent() results in a true response.

To implement the abstraction model, this method should produce one or more new events which encapsulate the state of the analysis and tuning domain as if it were a single parallel application task. The information in this event should match the information which was requested by a monitoring order previously translated by the associated *Instrumentation Order Translator*. The output of this method is formed by a set of one or more `Event` objects.

Plugin Architecture

As its name implies, ELASTIC is a flexible tool that gives the user a great amount of freedom with respect to the manner in which the performance analysis and tuning process is conducted. This flexibility is due to the plugin architecture that ELASTIC Packages offer.

By basing the codification of the performance and abstraction models on the subclassing of core *Abstractor*-ATM components, the user provided code in an ELASTIC

Package is given the same access as the base classes implementing the core Abstractor-ATM pair modules. This allows a level of customisation beyond what could be provided by a "black-box" architecture with a simple API interface. Additionally, this gives the ELASTIC developers the possibility of providing generalised ELASTIC Packages, which users can customise for a specific application, without having to change the code that guides the performance analysis and tuning, by simply following the design of the subclasses.

5.4.7 MRNet in ELASTIC

MRNet is a framework that establishes a hierarchical network of processes as communication substrate between a front-end and a number of back-ends. In the case of ELASTIC, the hierarchical network connects the ELASTIC front-end with the ELASTIC back-ends via the *Abstractor*-ATM pairs located at the internal nodes of this communication network.

Now, we will present how ELASTIC makes use of the capabilities provided by the MRNet framework in order to perform analysis and tuning on large-scale parallel applications.

To begin, the EFE uses the MRNet API to create and connect the internal processes of our tuning network. The tuning network topology that must be created is determined by a topology file that specifies the hosts on which the internal processes of the network should be located. Then, the EFE waits for the connection of the back-ends. To connect to the created network, back-ends require information such as process host name and port number, that is provided by the EFE via the file system. This mode of MRNet network instantiation is called *back-end attach mode* and it is suitable for tools that require their back-ends to create, monitor, and control other processes, as in our case.

Once the back-ends have attached to the network, the EFE uses two streams, one for control and one for performance analysis and tuning that will be used for communicating between the EFE, EBEs and the internal processes of the network, the *Abstractor*-ATM pairs.

The *control stream* is used to send metadata about the state of the tuning network at the beginning and end of the operation of ELASTIC. At the beginning, the EFE ensures that all EBEs work correctly and are connected to the network, and then the EFE initiates the performance analysis and tuning process. At the end, EFE checks, before deleting the tuning network, the all EBEs have finished the performance analysis and tuning over their task.

The *performance analysis and tuning stream* is used to send the performance analysis and tuning data (monitoring orders, events and tuning orders) between the EBEs and the internal nodes of the hierarchy where the *Abstractor*-ATM pairs are hosted.

In order to locate each *Abstractor*-ATM in an internal node, ELASTIC uses two MRNet filters. Both filters are bound to the *performance analysis and tuning stream*. One of these filters receives the upstream packets, while the other receives the downstream packets, they are named the *Upstream Filter* (UF) and the *Downstream Filter* (DF) respectively. It should be remembered that in the ELASTIC tuning network, upstream packets contain events and downstream packets contain instrumentation orders for monitoring or tuning.

To implement the *Abstractor*-ATM pair, its functionality must be divided between these two filters.

The UF accommodates the components of the *Abstractor*-ATM pair which are directly or indirectly involved in the reception and processing of events. These components are the *Event Manager*, the *Event Creator*, the *Performance Evaluator*, and the *Instrumentation Order Sender*. Additionally, to manage the connection with MRNet, the UF contains a *Network Proxy*.

To complete the functionality of the *Abstractor*-ATM pair, the DF hosts the components whose functionality is related to the translation of instrumentation orders. These components are the *Instrumentation Order Translator* and the *Instrumentation Order Sender*, as well as a *Network Proxy*.

Figure 5.9 shows how the components of the *Abstractor*-ATM pair are split between the UF and the DF.

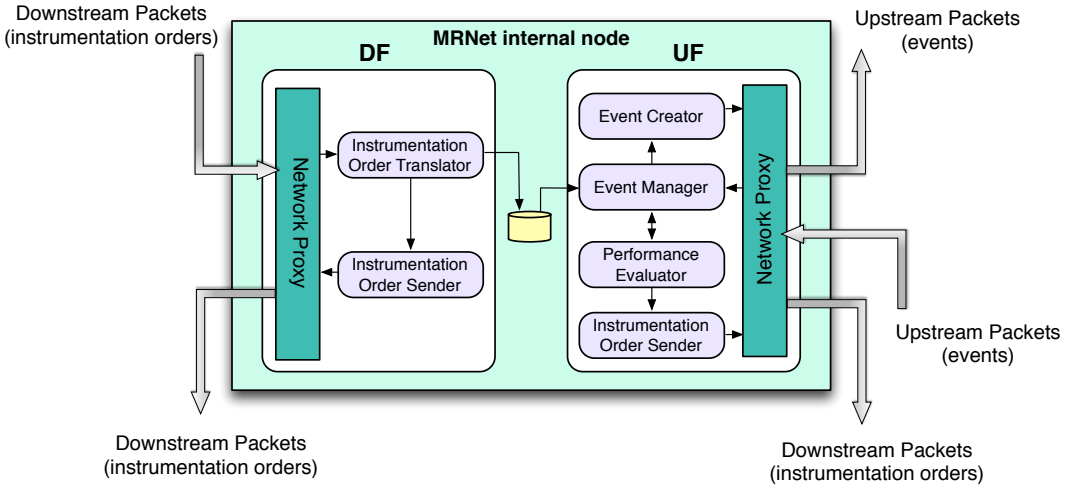


FIGURE 5.9: *Abstractor*-ATM pair as MRNet filters.

The operation of the UF includes the creation of new events and the performance evaluation process. As a result of the creation of new events, upstream packets are emitted to feed the UF of the parent level. On the other hand, the performance evaluation process may result in the generation of instrumentation orders for monitoring or tuning, which are sent as downstream packets. All downstream packets received by the DF will be transformed into one or more new downstream packets, containing instrumentation orders for monitoring or tuning. The instrumentation orders sent by both the UF and the DF feed DFs in the child level or will be received directly by ELASTIC back-ends.

The UF and DF share a common storage space. This storage is used for communication between the *Instrumentation Order Translator* and the *Event Manager* components which reside in different filters.

When the required infrastructure for communication and analysis has been created, the EFE starts to communicate with the EBEs following the pattern shown in Figure 5.10.

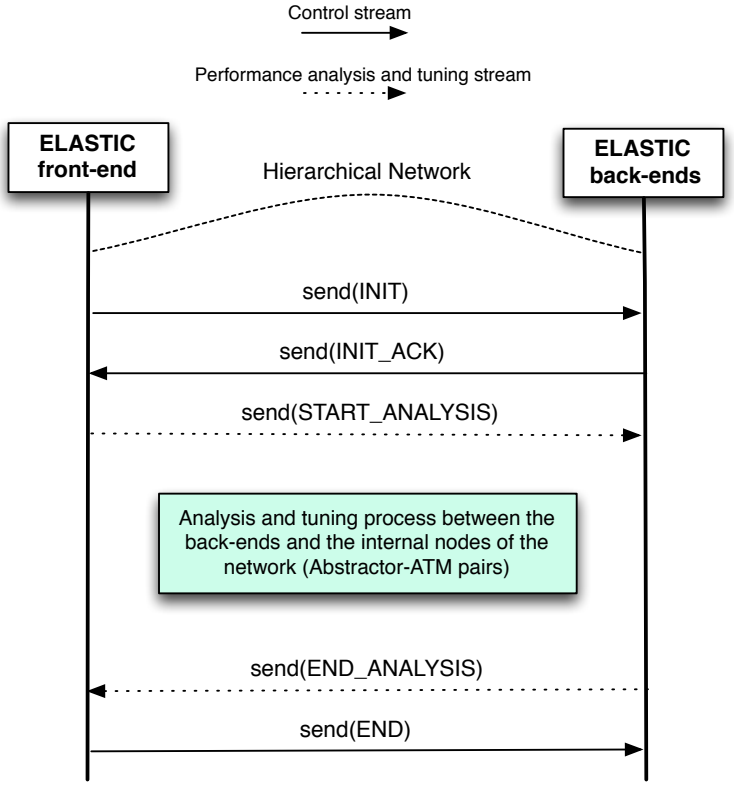


FIGURE 5.10: Communication protocol between the front-end and the back-ends in ELASTIC.

The EFE initialises the *control stream*, sending an `INIT` broadcast packet to the EBEs. EBEs launch the parallel application, and respond with an `INIT_ACK` packet letting the EFE know that they are ready to start the analysis and tuning process. Then, the EFE initialises the *performance analysis and tuning stream* sending a `START_ANALYSIS`

packet to the EBEs. This `START_ANALYSIS` packet is "bounced" back upstream so that it passes through all the DFs and UFs of the tuning network, activating the beginning of the analysis and tuning process. From this moment, the EFE stops receiving data from EBEs, because performance information about the application only travels through the internal processes of the network where *Abstractor*-ATM pairs are hosted by filters, carrying out the performance analysis and tuning process on the parallel application tasks.

When the parallel application finishes its execution, each EBE informs the EFE, sending a `FINISH_ANALYSIS` packet using the data stream. Then, EFE broadcasts an `END` packet via the *control stream* to the EBEs and proceeds to destroy the network.

5.5 Conclusions

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. In this chapter, this objective is met in the form of the implementation of the proposed model for hierarchical dynamic tuning. The result is the tool for large-scale tuning called ELASTIC.

ELASTIC follows the closed tuning loop of 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, the nodes of the tuning network where automatic performance analysis is conducted. After detecting a performance problem, tuning orders are sent to the ELASTIC back-ends to be inserted into the application task at runtime. These three phases comprise the dynamic tuning in ELASTIC.

The two main technologies employed by ELASTIC are DynInst and MRNet. The dynamic instrumentation required for the insertion of monitoring code and tuning orders into the parallel application is conducted using the DynInst library. The hierarchical communication of the proposed model is established in ELASTIC using the MRNet framework.

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. This plugin architecture gives ELASTIC the flexibility to tackle a wide range of performance problems.

6

Experimental Evaluation

"Esperando el nudo se deshace y la fruta madura."

– Federico García Lorca

In this chapter, we present the experimental evaluation conducted in order to verify that our model, presented in Chapter 3 and implemented in ELASTIC, enables dynamic tuning of large-scale parallel applications.

6.1 Introduction

Throughout this thesis, a model for hierarchical dynamic tuning has been presented and its implementation in the tool ELASTIC has been detailed. Therefore, this chapter presents an experimental evaluation with the aim of demonstrating that the proposed model implemented in ELASTIC enables dynamic tuning of 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. The first part of this evaluation is designed to test the scalability, achieved by adapting the tuning network to the application size, of the model for hierarchical dynamic tuning. It will show that ELASTIC is capable of managing the volume of connections and information necessary to effectively tune large scale parallel applications. This will also validate the scalability experimentation conducted using the prototype in Section 4.4. The second part of the evaluation attempts to show that when operating at large scales, ELASTIC is able to achieve performance improvements in the analysed parallel application.

Two parallel applications have been used in the evaluation. The first is a synthetic SPMD (Single Programme Multiple Data) parallel application into which various load imbalance patterns have been introduced. These load imbalances must be solved by ELASTIC using local migrations between neighbouring tasks. The second application is an agent based simulation which also follows an SPMD paradigm. This application simulates an epidemic model, which due to the dynamic nature of the agents (reproduction and death), presents load imbalance problems. In this case, the migration of agents to balance the load follows an all-to-all pattern.

These applications have been chosen because, of all the currently employed parallel programming paradigms, SPMD is that which is most likely to scale to tens of thousands of processes [53]. Similarly, in current SPMD applications load imbalance is common and one of the main causes of inefficient use of parallel machine resources. Furthermore, load balancing is a problem which cannot truly be resolved without a global view of the load state of the application, differing from locally resolvable problems such as inefficient use of cache memory. As such, load balancing exploits the potential of the proposed model in this work because it is a *decomposable* and *abstractable* problem.

For each one of these parallel applications, an ELASTIC Package which is capable of resolving the inefficiencies related to the load imbalance has been integrated into ELASTIC. Each of these packages takes into account the characteristics of the underlying

parallel application and implements a strategy to conduct dynamic tuning, improving the performance.

The details of the experiments performed with the synthetic SPMD application, and their results, are presented in Section 6.2. Then, in Section 6.3, the use of ELASTIC with the agent based application is detailed. A study of the overhead caused by ELASTIC is presented in Section 6.4. Finally, Section 6.5 presents conclusions and some final remarks arising from the experimentation.

Execution Environment

The experimental evaluation was executed on the supercomputer SuperMUC at Leibniz Supercomputing Centre. As of the June 2013 Top500 list of supercomputers, SuperMUC ranks in ninth place [59]. SuperMUC is composed of 9 400 compute nodes, with a total of 155 656 processor cores and over 300 Terabytes of main memory between them.

The experiments were conducted using *thin* node islands. Each island is composed of 512 nodes interconnected by Infiniband FDR10. The nodes have 2 8-core 2.7GHz Intel Xeon processors and 32GB main memory, running SuSe Linux. During the experiments, up to 3 islands were used for each execution.

In the experimental tests presented in this section, each ELASTIC back-end is allocated in one core, whereas each *Abstractor*-ATM pair uses four cores. This is because each of the internal nodes of MRNet is multi-threaded and manages two threads for every directly connected node (children and parent) in the hierarchy [7].

6.2 Synthetic Application

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

The tasks that make up the application are logically arranged in a square two dimensional grid. As such, each task has up to four neighbouring tasks, with those tasks on the edge of the grid having fewer. Each task is only able to communicate directly with its neighbouring tasks. Figure 6.1 shows a representation of the application grid, with the communication lines between 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.

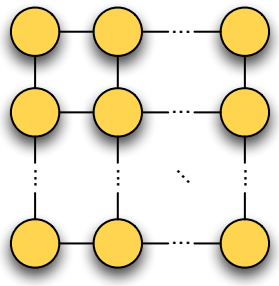


FIGURE 6.1: Logical layout of the synthetic application.

In the communication phase, each task exchanges messages with all of its neighbouring tasks. Each message is of a fixed size.

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. This situation is common in many scientific applications, such as meteorological simulations, where a larger number of tasks are used to increase the granularity of the simulation, rather than reduce execution time. In our experimental evaluation, each task started the execution with 20 work units.

A synthetic parallel application was chosen to perform these experiments as it provides the possibility of testing different configurations in the most controlled manner.

6.2.1 Performance Problem

A common performance issue in SPMD applications is the imbalance of workload between individual tasks. This causes inefficiency in the application performance, due to the tasks with less workload having to wait for overloaded tasks to complete the iteration. This situation is illustrated in Figure 6.2.

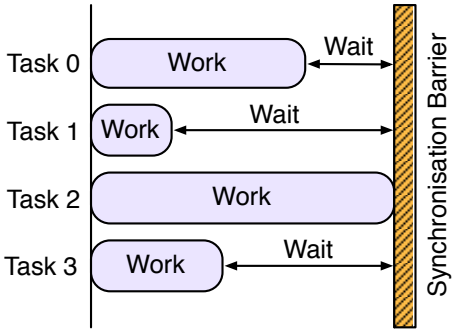


FIGURE 6.2: Load imbalance problem.

Since we are working with the synthetic application, the load pattern can be controlled. To demonstrate the dynamic tuning capabilities of ELASTIC, load imbalance was introduced into the synthetic application at runtime. In all cases, the load imbalance

introduced was localised to one or more logical areas of the application grid, forming "hotspots" of additional load.

Two different load imbalance scenarios have been employed: single localised hotspot and multiple hotspots.

Single Localised Hotspot.

This scenario is characterised by presenting a single area of additional workload located at the centre of the application grid. This area of additional workload is introduced at a specific iteration of the synthetic application execution.

The hotspot contains three concentric areas of incremental additional load. The widest area has a diameter of approximately 40% of the width of the application grid. The additional load introduced is from 20 work units at the outer edge of the hotspot to 60 work units at the inner most point. Counting the original 20 work units, each of these inner tasks will have a total of 80 work units. In total, the introduced load accounts for an additional 10% of the initial application workload for all application sizes.

Figure 6.3 depicts an example of a single localised hotspot in a synthetic application made up of 1 024 tasks in a 32×32 grid. This figure is a **heatmap**, the space represents the logical grid of application tasks. The colour assigned to each task represents its number of work units.

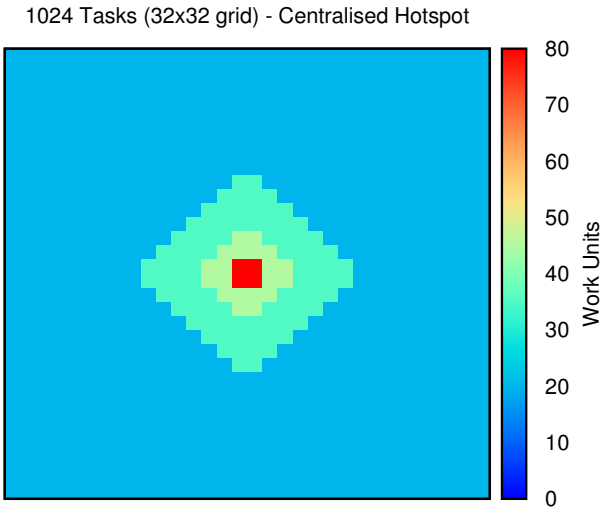


FIGURE 6.3: Centralised hotspot in a 1 024 task synthetic application.

Multiple Hotspots.

In this scenario the additional load introduced is distributed amongst a number of smaller hotspots. To demonstrate the usefulness of dynamic tuning, 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, however this placement does ensure that the area covered by hotspots from the same group does not overlap. The three groups are laid out so that the first two groups do not overlap, but the second and third groups do. This situation presents different levels of imbalance throughout the application execution.

Each hotspot is formed of two concentric areas of incremental additional load. The hotspots in the first and third groups add up to 20 work units, while those of the second group add up to 40 units. If no load balancing takes place, the maximum number of work units for a single task after each of the three injections of additional load will be 40, 60 and 80 respectively.

An example layout of the three groups (a, b, c) is presented in Figure 6.4 for a synthetic application composed of 1 024 tasks in a 32×32 grid.

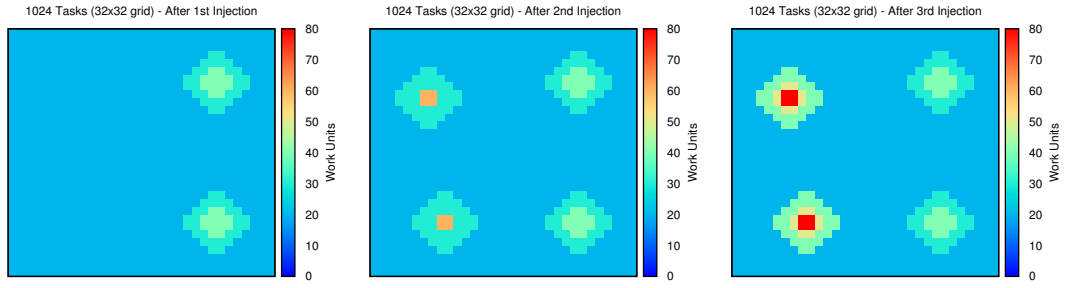


FIGURE 6.4: Multiple hotspots in a 1 024 task synthetic application.

6.2.2 ELASTIC Package

To solve the problems related to load imbalance in the synthetic application an ELASTIC Package has been implemented and integrated into ELASTIC. The same ELASTIC Package is used in all the *Abstractor*-ATM pairs.

This ELASTIC Package attempts to balance the load 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. In the remainder of this section, the specific details of the functionality implemented by this package are presented.

This ELASTIC Package uses the Asynchronous Policy of the tuning network presented in Section 3.4.4 along with the Tuning Coherence Mechanism proposed in the same section.

To structure the content of this ELASTIC Package, it will be presented divided into its two conceptual parts, the performance model and the abstraction model.

Performance Model

The performance model provides the knowledge required to guide the dynamic analysis and tuning process carried out by ELASTIC.

As it was presented in Section 3.3.1, a performance model for dynamic tuning is defined in terms of measurement points (parameters that have to be measured at specific points in the application to collect information about its performance), a set of expressions to detect performance problems and give solutions to overcome them, and tuning points, actions and a synchronisation method (where to insert changes, what kind of changes to insert, and when to insert these changes).

Measurement Points. To detect load imbalance, the performance model requires, for a given iteration, the number of work units of each application task over which it is performing load balancing (i.e. the set of tasks that comprises an analysis and tuning domain). This information is provided by a variable in the application called `work_units`. It is also necessary to identify the task from which this information comes and the iteration in which it was collected. For this, we use the task's MPI rank and the variable `iteration_id`.

The values of `work_units` and `iteration_id` are collected at the beginning of the function which constitutes the computation phase, `work()`. The MPI rank is automatically included in the event generated. Following the *MonitoringOrder* class specification, detailed in Section 5.4.3, the contents of the monitoring order generated are shown in Figure 6.5.

Only one monitoring order is necessary to gather the parameter required to evaluate the performance expressions. This monitoring order is sent by the ATMs to all the tasks

eid	action	funcName	place	attrs
1	0	work	1	[work_units, iteration_id]

FIGURE 6.5: Monitoring order to collect the required information from measurement points in the synthetic application.

in their analysis and tuning domains at the beginning of the application execution, and then inserted into the application tasks. As a consequence of this monitoring order, each task will generate one event per iteration containing the requested information.

Performance Expressions. In this ELASTIC Package the performance expressions take the form of an algorithm for load balancing. As previously mentioned, the input of the algorithm is, for a given iteration, the number of work units in each task of the analysis and tuning domain. The output is the number of work units (if any) that each task should send to each of its neighbours.

When the *Performance Evaluator* in an ATM has received an event from each task, for a given iteration, performance analysis is activated. The load balancing algorithm is only executed if the imbalance in the analysis and tuning domain, in terms of work units per task, is greater than a specific threshold. The imbalance is calculated as the difference between the maximum and minimum number of work units per task, divided by the average of work units per task. In this ELASTIC Package the threshold was set to 5%.

Each task object t in the algorithm contains a number of properties:

- $work$ - the number of work units in this task
- $rank$ - the rank of this task
- $to\_send[4]$ - a map defining the number of work units to be sent to each neighbouring task
- $to\_receive[4]$ - a map defining the number of work units to be received from each neighbouring task
- $available\_work$ - the number of work units after subtracting units to be sent $(t.work - \sum t.to\_send[])$
- $next\_iter\_work$ - the number of work units after adding units to be received and subtracting units to be sent $(t.work + \sum t.to\_receive[] - \sum t.to\_send[])$

Pseudocode describing the operation of the algorithm is presented in Algorithm 6.4.

The algorithm looks at each row and each column of tasks in the analysis and tuning domain individually. The row or column is balanced by first finding the ideal number of work units based only on the tasks in that row or column. Then, the most

Algorithm 6.4 Pseudocode to load balance the synthetic application.

Input: $tasks[N]$
Input: $size$ // the size of the analysis and tuning domain ($size.width * size.height = N$)
Input: $threshold$ // the imbalance threshold
Output: $tasks[N]$ // array of tasks with *to_send* and *to_receive* updated

```
max_work ← maximum number of work units for a task in tasks[]
min_work ← minimum number of work units for a task in tasks[]
ideal_work ← average number of work units for tasks in tasks[]

if (max_work − min_work)/ideal_work > threshold then

  // Balance individual columns
  for i = 0 → size.width − 1 do
    column[size.height] ← i^th column of domain
    k ← index of task in column[] with greatest number of work units
    ideal ← average number of work units for tasks in columns[]
    lower_col[k + 1] ← column[] elements 0 up to k
    calculate_vector_balance(lower_col, ideal)
    upper_col[size.width − k] ← column[] elements (size.height − 1) down to k
    calculate_vector_balance(upper_col, ideal)
  end for

  // Balance individual rows
  for i = 0 → size.height − 1 do
    row[size.width] ← i^th row of domain
    k ← index of task in row[] with greatest number of work units
    ideal ← average number of work units for tasks in row[]
    lower_row[k + 1] ← row[] elements 0 up to k
    calculate_vector_balance(lower_row, ideal)
    upper_row[size.width − k] ← row[] elements (size.height − 1) down to k
    calculate_vector_balance(upper_row, ideal)
  end for

end if
```

overloaded task is found, and the row or column is divided around that task. Each part of the row or column is called a task vector, and passed separately to the function *calculate_vector_balance*(). Figure 6.6 shows an example of this behaviour for a single row composed of six tasks.

Pseudocode describing *calculate_vector_balance*() is given in Algorithm 6.5. It should be noted that both vectors include the most overloaded task.

Each vector is balanced by starting at the opposite end to where the overloaded tasks is located. Each task is checked in turn, and if the task is underloaded, work units are moved from the next task in the vector. This process is repeated until the overloaded task is reached. The migration information is stored in the task objects, in the properties *to_send* and *to_receive*.

By balancing each row and column separately, a migration scheme to improve the imbalance in the entire analysis and tuning domain is achieved. It is important to recognise, that due to the local-only communication pattern to which this algorithm is

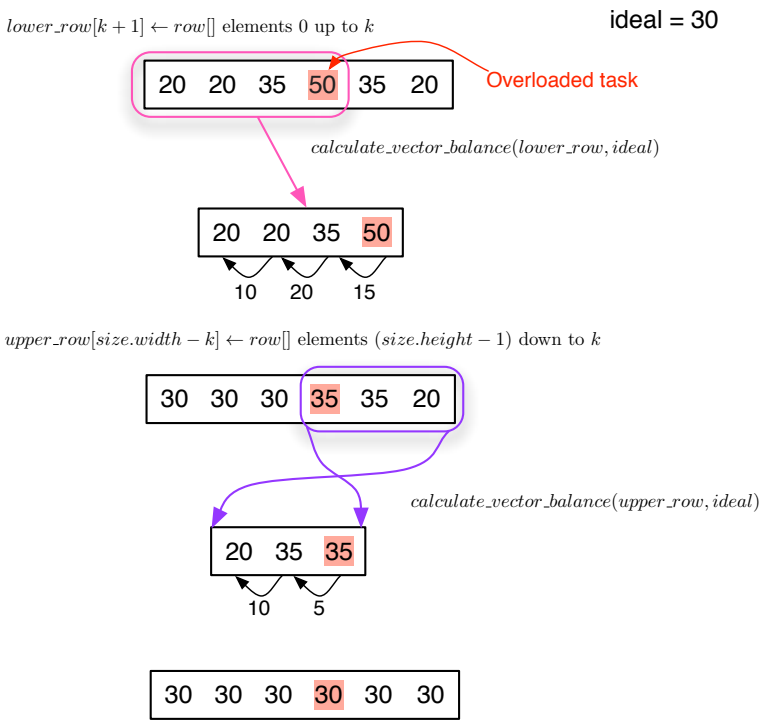


FIGURE 6.6: Single row load balance example.

Algorithm 6.5 Pseudocode for *calculate_vector_balance*() function.

Input: *task_vector*[*M*] // array of tasks from a column or row of size *M*
Input: *ideal* // the ideal number of work units in the given row or column
Output: *task_vector*[*M*] // array of tasks with *to_send* and *to_receive* updated

```
for j = 0 → M − 2 do
  if task_vector[j].next_iter_work < ideal then
    to_move ← min{ideal − task_vector[j].next_iter_work, task_vector[j+1].available_work}
    task_vector[j].to_receive[task_vector[j + 1]] ← to_move
    task_vector[j + 1].to_send[task_vector[j]] ← to_move
  end if
end for
```

subjected, it is often not possible to balance the load perfectly in one execution of the load balance algorithm. However the imbalance state of the analysis and tuning domain will be improved.

Tuning Points, Tuning Actions and Synchronisation Method. In order to move load between tasks a migration function has been inserted into the execution flow of the synthetic application after the communication phase. This migration phase is activated every three iterations in each application task, independent of whether there is anything to be migrated. This is because a task does not have prior knowledge about whether it is going to receive work units from one of its neighbours or not.

The maximum migration frequency possible is every 2 iterations because a full iteration is required to collect information about the state the application after each

migration. The migration frequency of every 3 iterations was chosen to strike a balance between responsiveness and minimising the intrusion generated by the tuning process.

To activate the migration of work units from one task to another, the sending task must be instrumented. In this instrumentation, the tuning points are the variables that represent the number of work units to be sent to each neighbouring task. These variables are located in the migration function and are called `send_north`, `send_south`, `send_east`, and `send_west`. The names correspond to the neighbouring tasks located at each cardinal point. The tuning action to be performed is setting the value of each of these variables with the value resulting from the load balancing algorithm. This action occurs at the beginning of the migration phase (synchronisation).

After the performance evaluation and in the case that load imbalance is detected, an ATM creates and sends tuning orders for each task in its domain. A tuning order is an instruction to set the value of a single variable, one of `send_north`, `send_south`, `send_east`, or `send_west`. So, each task will be sent up to 4 tuning orders.

Abstraction Model

The abstraction model determines how the information generated during the analysis and tuning process is transferred between the *Abstractor*-ATM pairs at different levels in the hierarchy, as well as how the application is divided into domains.

This section describes a) how the decomposition is performed over the synthetic application, b) how events that encapsulate the information which represents the state of an analysis and tuning domain are generated, and c) how monitoring and tuning orders are translated as they flow down through the hierarchy.

Division Scheme. The abstraction model defines how the application is divided into domains to perform a hierarchically distributed analysis and tuning process.

In the case of the synthetic application, an "abstractable" decomposition can be obtained by ensuring that the original application is divided into domains, each of which is composed of contiguous blocks of tasks, as shown in Figure 6.7.

These domains are the analysis and tuning domains of ATMs at the base level of the hierarchy. So, these ATMs, following the communication pattern of the application, are able to balance the load within their own domain. Following the abstraction concept of the model proposed in this work, the *Abstractor*-ATM pairs in the base level form a virtual synthetic application. As shown in Figure 6.7, this virtual application may then be further subdivided to form the analysis and tuning domains of the *Abstractor*-ATM pairs at the parent level in the hierarchy. When an ATM at this parent level decides to

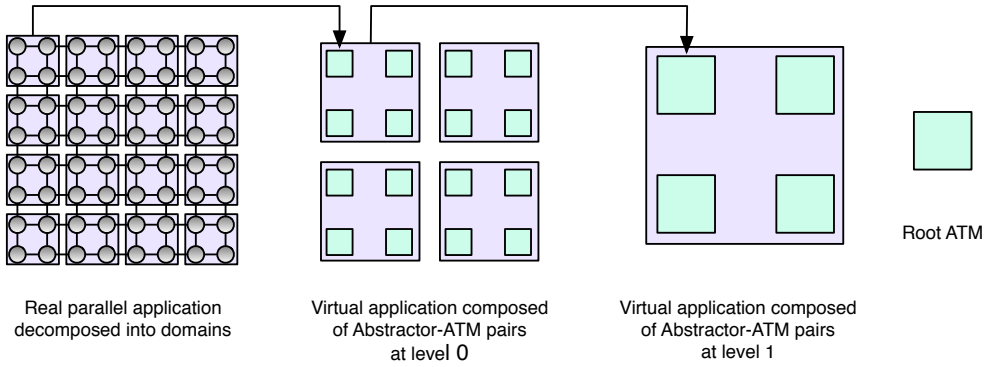


FIGURE 6.7: Parallel application decomposition into "abstractable" domains.

move load in its analysis and tuning domain, this decision is applied at the level of the original application, as movement of load, between the application tasks which form the borders between domains.

To maintain the homogeneousness of the original application, each domain in the same level will have the same dimensions. This division scheme also produces domains which are *analysable and tunable*. In all the experiments with the synthetic application in this section, the application and the domains are square.

Monitoring Order Translation. The information required by the performance model is the total number of work units in each application task and the rank of the task. As the same ELASTIC Package is used at all levels in the hierarchy, the same monitoring orders are generated at all levels. Therefore, when an *Abstractor* receives the monitoring order from its parent ATM, it simply registers that order with the Event Manager module. In this way, the *Abstractor* will also received the events generated as a consequence of the monitoring order sent by its associated ATM.

Event Creation. The *Abstractor* must be able to satisfy the monitoring order received from its parent ATM and provide the number of work units and the iteration when this information was collected.

This abstraction model considers the number of work units of an analysis and tuning domain (which will be represented as a virtual task) to be the sum of the work units of each of the tasks within it. To calculate this value, an *Event Creator* has to received one event from each task in the domain for a give iteration.

In Figure 6.8, an example of the event creation process is shown. Supposing an analysis and tuning domain composed of 4 tasks, when the *Event Creator* has received 4 events for the same iteration, it can create a new event. The 4 received events and the new event are represented in terms of the *Event* class detailed in Section 5.4.3.

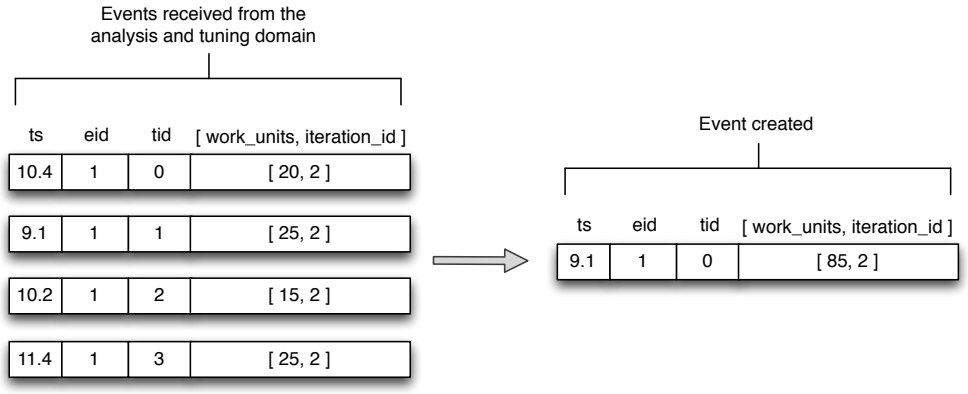


FIGURE 6.8: Event creation example for the synthetic application.

The timestamp of the new event, ***ts***, is the minimum of the received events because the information in the new event should be considered as old as the oldest event from which it is created. The event ID, ***eid***, is the same, and it should be noted that it corresponds with the ID of the monitoring order that requested the information. The rank, ***tid***, is the rank of the *Abstractor*-ATM pair where the *Event Creator* is located. This value is assigned at the beginning of the execution of ELASTIC, in this example it is 0. Finally, the value of work units is the sum of this variable in each of the received events and the iteration is the same.

Tuning Order Translation. Tuning orders are generated by ATMs after evaluating the performance model in order to improve the parallel application's performance. When an *Abstractor* receives a tuning order from its parent ATM, it must be translated to be applied to the analysis and tuning domain of its associated ATM. The translated tuning orders must provoke the same change in the application at the level below (be it the original parallel application or another virtual application) as was requested by the tuning order received at this level.

This abstraction model implements this translation according to the communication pattern of the synthetic application. This means that an order to migrate work units between two *Abstractor*-ATM pairs is translated so that the migration occurs between the tasks of their respective analysis and tuning domains which share a common border.

To clarify this behaviour, Figure 6.10 shows an example where an order from level $i+1$ is received, by a given *Abstractor* **AB** at level i, to move 200 work units to the neighbouring *Abstractor*-ATM pair to the *north*, **N** (a). **AB** actually represents a 4×4 grid of tasks at level $i-1$ (the analysis and tuning domain of its associated ATM). The 200 work units to be moved are evenly divided, so that each task on the *north* border of the domain (at level $i-1$) receives a tuning order directing it to move 50 work units to its *north* neighbouring task (b). If level i is not the base level in the tuning network, then

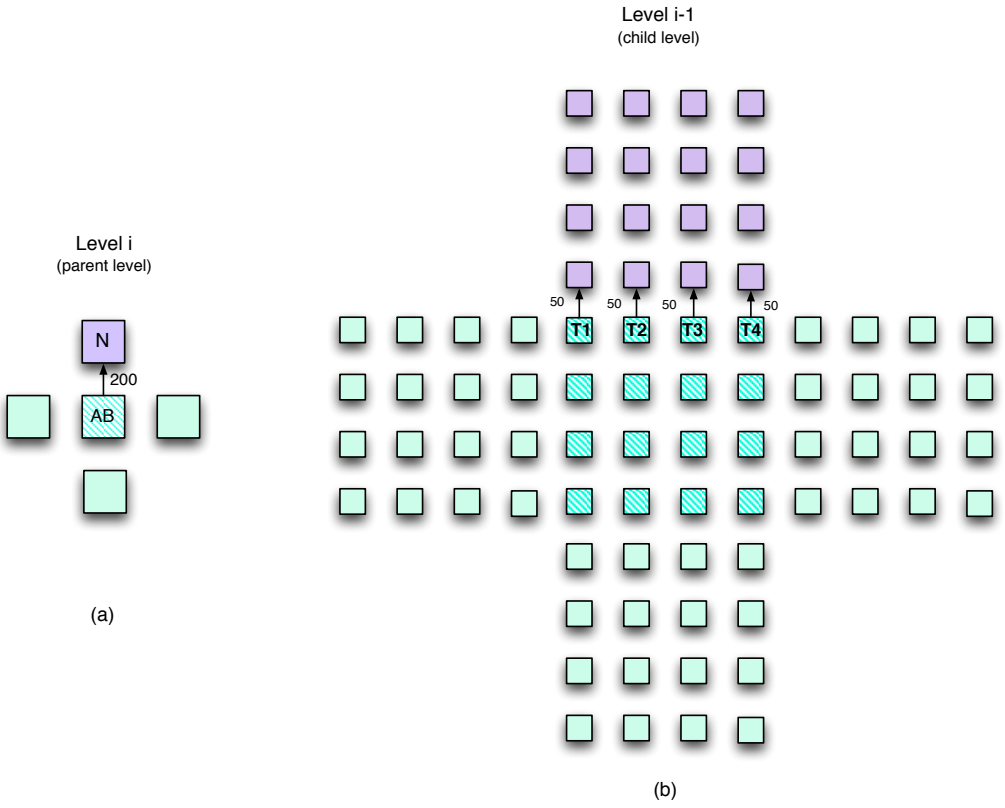


FIGURE 6.9: Tuning order translation diagram for the synthetic application.

these tasks are actually also *Abstractor*-ATM pairs, and the tuning order translation process will be repeated in each one.

So, in this example, the tuning order to set the value of the variable `send_north` to 200 is translated into 4 tuning orders which are sent to the tasks T1, T2, T3 and T4, located on the northern border of the analysis and tuning domain of **AB**. This translation in terms of the specification of the *TuningOrder* class is shown in Figure 6.10. The definition of *SetVariableValueTuningOrder* subclass used can be found in Appendix A.

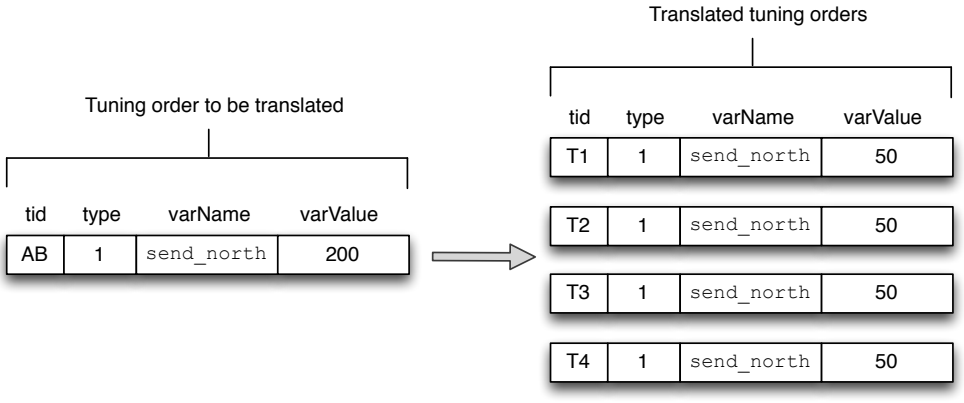


FIGURE 6.10: Example of *SetVariableValueTuningOrder* translation for the synthetic application.

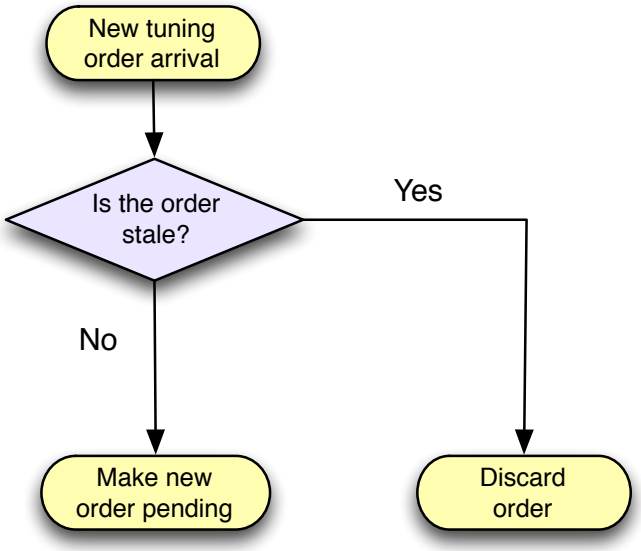


FIGURE 6.11: Simplified coherence mechanism.

According to the characteristics of the migration process and the abstraction mechanism, an individual task (virtual or real) cannot receive contradictory orders, from different levels in the tuning network, directing it to migrate work units in the same direction. Using the previous example, the task **T3** can receive orders to migrate work east, south or west from the the ATM located at level i in the hierarchy, and orders to migrate work north which originate at level $i+1$ and are then translated at level i. Because orders cannot be contradictory, the coherence mechanism described in Section 3.4.4 can be simplified, and so the ELASTIC Package follows the decision flow shown in Figure 6.11

6.2.3 Tuning Network Topology

Throughout the experimental evaluation with the synthetic application, tests were executed using application configurations composed of different numberst of tasks. For each application size, the topology of ELASTIC's tuning network was calculated, following Algorithm 3.1 depicted in Section 3.5.1, so that the *Abstractor*-ATM pairs would not become a bottleneck.

In order to calculate these topologies, certain values had to be measured while others could be calculated directly from the configuration of the synthetic application and the analysis and tuning process. The values of all the required variables can be found in Table 6.1

TABLE 6.1: Values of variables required to calculate tuning network topology for the synthetic application.

Variable	Value	Description
E_a	1	# events from each child node required for analysis
E_c	1	# events from each child node required for event creation
$T_a(N)$	$0.004 \cdot N^2$ ms	analysis time (quadratic)
T_m	0.02 ms	management time
T_c	0.2 ms	event creation time
T_t	0.3 ms	instrumentation order translation time
f_e	2.5 events/s	frequency of event generation in the parallel application
f_{rc}	2.5 batches/s	frequency of event reception from each child node
f_{rp}	2.5 orders/s	frequency of tuning order reception from the parent ATM

Parameters Calculation

As was explained in Section 6.2.2, the analysis process in the ATMs requires one event from each child node to be activated, and from each of these "event batches", a new event is created to be sent to the parent ATM. For these reasons, $\boldsymbol{E_a}$ and $\boldsymbol{E_c}$ respectively take the value 1.

In the synthetic application, each work unit represents an amount of work that must be performed in each iteration. Since the iterations are synchronised by barriers, the iteration time for the application as a whole is equal to that of its slowest task. As such, the minimum iteration time is when the application is fully balanced.

In our evaluation, the imbalance introduced into the application never reduced the total number of work units in the application from its starting amount, which is 20 work units per task. In order to ensure that the topology does not become saturated at any point in the execution, it is this minimum which we must use to calculate it. A single work unit represents approximately $20ms$, which means that the minimum computation time for the application in each iteration is approximately $400ms$.

As the synthetic application represents a computation bound application, the time required for communication between tasks can be disregarded for the purpose of these calculations, and we will consider the iteration computation time to be equivalent to the total iteration time. As such, $\boldsymbol{f_e} = 1/400ms = 2.5$ events per second.

Since the rate at which events are created (E_c) is 1, the frequency with which an event is received from each child node will be the same at all levels in the tuning network, and is equal to the frequency with which events are generated at the application level. So, $\boldsymbol{f_{rc}} = f_e$.

In this load balance experimental evaluation, it is expected that there will be times when a tuning order is created following every analysis process performed, which will lead to a frequency of tuning order generation that is equal to that of the reception

of event batches. For this reason, the frequency with which tuning orders are received from the parent ATM is considered to also be the same as the frequency that events are generated in the application, i.e. $\boldsymbol{f_{rp}} = f_e$.

The remaining variables represent the time required to perform various tasks from the analysis and tuning process, $\boldsymbol{T_m}$, $\boldsymbol{T_c}$, $\boldsymbol{T_t}$ and $\boldsymbol{T_a(N)}$. In this case, the values have been obtained through prior evaluation of the analysis process. In the case of the analysis time, it was found that the level of imbalance in the analysis and tuning network affected the time required to perform analysis, but only up to a certain level (where migration had to be considered for every task in the domain). This maximum time is the value used for $\boldsymbol{T_a(N)}$, which is a quadratic function in the number of nodes in the analysis and tuning domain.

Topology Calculation

Using the previous values, we can calculate the most efficient topology for ELASTIC's tuning network. While the topology will be different for different application sizes, the number of application tasks that a base level ATM can support will remain the same, although the actual analysis and tuning domains of these ATMs may be changed for different application sizes.

To calculate the maximum domain size for the base level ATMs, the first iteration of the loop in Algorithm 3.1 must be followed, for $i = 0$, and the following expression must be solved for N_{max}. The analysis frequency is equal to frequency with which events are generated in the application, $f_e = f_a = 2.5$ analyses / second.

$$N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp} = \frac{1}{f_a}$$

$$\left(\frac{0.004}{1000}\right) N_{max}^2 + \left(\frac{0.02}{1000}\right) N_{max} + \left(\frac{0.2}{1000} + \frac{0.3}{1000} \cdot 2.5 - \frac{1}{2.5}\right) = 0$$

Using the quadratic formula we can solve for N_{max}, which gives,

$$N_{max} \approx 314$$

This indicates that each ATM at the base level of the tuning network is capable of supporting an analysis and tuning domain of up to 314 application tasks.

Because the ELASTIC Package used to perform load balancing on the synthetic application is the same at all levels of the hierarchy, and because a new event is created and sent to the parent ATM for every event batch received ($E_c = 1$), this expression

is the same for all levels in the tuning network. As such, all ATMs, irrespective of the level at which they are situated, are able to support an analysis and tuning domain of up to 314 child nodes without becoming saturated.

In order to evaluate the scalability and effectiveness of the proposed model for dynamic tuning as it is implemented by ELASTIC, the experiments presented have been run for applications composed of different numbers of tasks.

In the case of the local communication limitations present in the synthetic application, load balancing performance becomes dependent on the geometry of the application. Balancing the load of a 12×12 grid is not equivalent to doing the same over a 16×9 grid, even though both are composed of 144 application tasks.

In order to make experiments over different numbers of application tasks as comparable as possible, it is desirable that all applications have proportional geometry. To extend this proportionality to the analysis and tuning domains as well, all the application grids as well as the analysis and tuning domains should have square geometry, that is, the grid of application tasks has equal height and width. Additionally, to aid comparability, the width of the application should be divisible by the width of each analysis and tuning domain.

The application sizes have been chosen to accommodate analysis and tuning domains of 16×16 application tasks. This gives 256 application tasks, a square number below N_{max}, the maximum number of tasks that a single ATM can support. The application sizes and their associated tuning network topologies are presented in Table 6.2.

TABLE 6.2: 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

The 256 task application is analysed and tuned by a centralised tuning network, while the remainder of the application sizes are analysed and tuned by tuning networks made up of two levels. A single ATM at the root level controlling between 4 and 64 ATMs at the base level.

6.2.4 Scalability Evaluation

The first evaluation to be performed has been designed to test the scalability of ELASTIC.

This experimentation will be used to verify the results obtained in Section 4.4 and demonstrate that the proposed hierarchical model provides a scalable environment which offers the capabilities necessary to perform dynamic tuning over parallel applications composed of tens of thousands of tasks.

As in Section 4.4, the global decision time will be measured for parallel applications of different sizes. The results to be presented come from the same experiments used to demonstrate the effectiveness of ELASTIC in the sections that follow. Specifically, we have used the experiments featuring multiple hotspots of imbalance.

The most influential component of the decision time is the analysis time. As it was mentioned, the analysis time of the load balancing algorithm implemented in the employed ELASTIC Package is variable, depending on the state of imbalance in the analysis and tuning domain. In order to obtain results comparable with those presented in Section 4.4, where the analysis time did not vary during the execution, the maximum decision time during an execution is used. We believe that this also presents a stronger case for the scalability of our model.

Using the application sizes and their associated ELASTIC topologies presented in Table 6.2, Figure 6.12 presents the maximum decision time for each level in ELASTIC's tuning network for each application size.

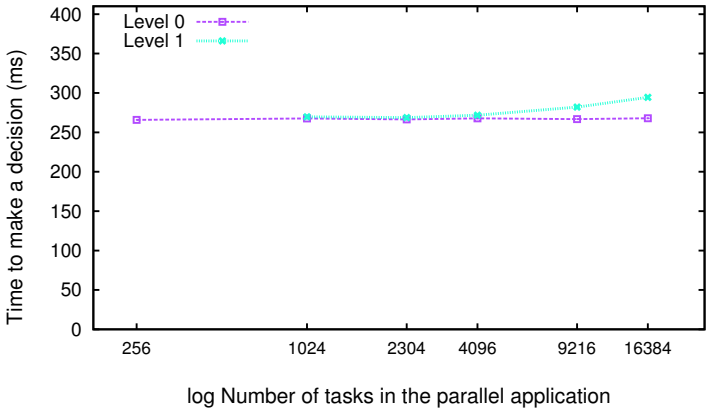


FIGURE 6.12: ELASTIC's global decision time tuning a synthetic application.

Because the load balancing performance model used to analyse and tune the synthetic application presents a quadratic analysis time with respect to the number of tasks in the analysis and tuning domain, the global decision time is seen to grow accordingly for the experiments from 1 024 to 16 384 application tasks. This follows the same pattern

that was observed during the experimentation in Section 4.4, offering validation that the global decision time increases logarithmically with the number of tasks in the parallel application.

6.2.5 Effectiveness Evaluation

Having shown that the proposed model for hierarchical tuning is able to scale to manage applications composed of many thousands of tasks, it remains to show that this model is able to effectively tune a parallel application. In this section, the results of how ELASTIC tunes the synthetic application will be presented. Two separate load imbalance scenarios will be tuned, as detailed previously in this section: single localised hotspot and multiple hotspots.

Following the performance and abstraction models, implemented in the ELASTIC Package and described in Section 6.2.2, each analysis module at the base of the hierarchy balances the workload of its analysis and tuning domain. The root analysis module, whose domain is composed of the base level analysis modules, balances the workload between the domains of its child analysis modules. The workload migration decisions made at both levels balance the load of the entire application.

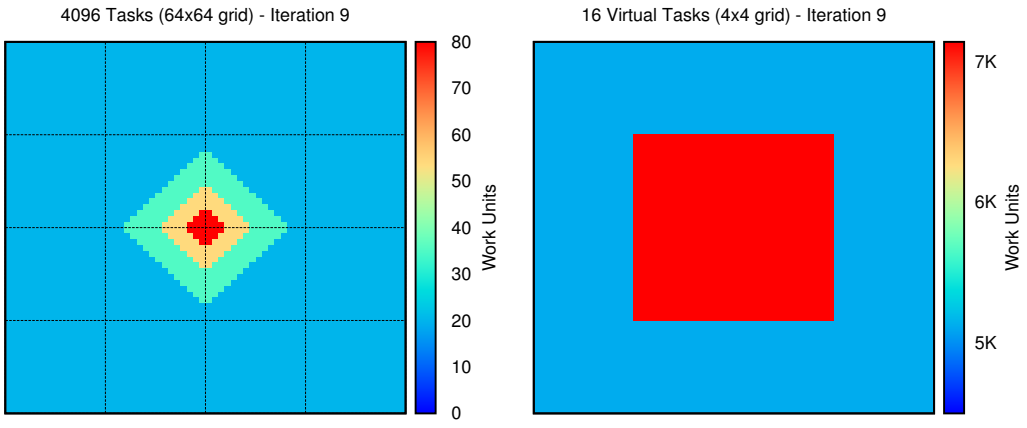
Centralised Hotspot

To begin this section, we will show how ELASTIC employs hierarchical performance analysis and tuning in order to distribute additional load which is dynamically introduced at the centre of the application grid.

Using the synthetic application composed of 4096 tasks, we will use a series of *heatmaps* to show the state of the load imbalance in the synthetic application and how it changes under the dynamic tuning performed by ELASTIC.

In Figure 6.13 (a) the load state of the application is depicted in the iteration when the imbalance is first introduced.

The analysis and tuning domain of each *Abstractor*-ATM pair at the base level of the tuning network, composed of 256 tasks, is delimited by the black lines. In turn, the root ATM controls the virtual application composed of these *Abstractor*-ATM pairs. The view that the root ATM has of the application is show in Figure 6.13 (b), where each square represents an abstracted analysis and tuning domain at the base level as a virtual application task. The work units in each virtual task, following the abstraction model, are equal to the sum of the work units of all the tasks in the domain that it represents.



(a) (b)

FIGURE 6.13: Initial centralised load imbalance in (a) the synthetic application and (b) the virtual application.

As can be seen the load imbalance takes the form of a centralised hotspot of additional load. Once ELASTIC detects this imbalance situation, the ATMs will send tuning orders to their respective analysis and tuning domains in order to correct the imbalance.

As was previously mentioned, load migration between neighbouring tasks occurs every three iterations. Therefore, since no changes can be observed between migrations, we are only interested in the state of the application every three iterations as ELASTIC performs dynamic tuning. In Figure 6.14 the heatmaps obtained from the dynamic tuning process are shown, up until the point where the state of the application becomes stable and no further tuning is performed.

The transitions between heatmaps show the effects of how ELASTIC dynamically balances the load of the application. Throughout the sequence of heatmaps it can be seen how the imbalance situation detected by the root ATM is corrected by moving load between the tasks located at the borders of the domains. Due to the abstraction mechanism implemented in the ELASTIC Package, the tuning orders generated by the root ATM cause destabilisation within the individual domains that comprise the real application. The imbalance introduced by the actions of the root ATM will result in the tasks of the domain borders being over or underloaded. It is observed that when the base level *Abstractor*-ATM pairs discover this imbalance, new tuning orders are sent to correct the situation, while the root ATM continues performing its analysis and tuning process. This behaviour reflects the pattern of level by level stabilisation described in Section 3.6.

The progression of the load balancing process shown in the heatmaps reflects how ELASTIC's combination of operations performed by all the *Abstractor*-ATM pairs in the

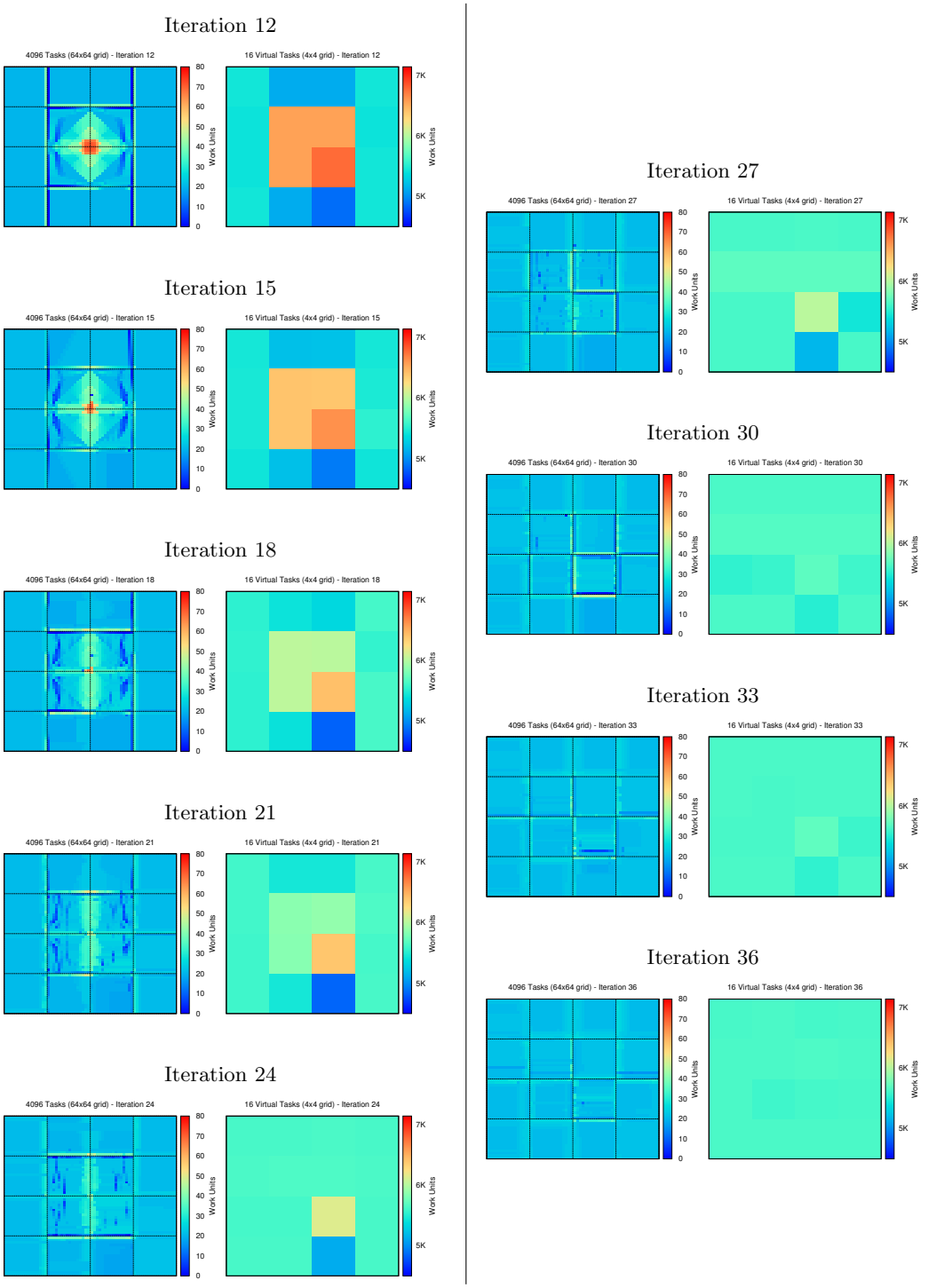


FIGURE 6.14: Heatmaps obtained during the synthetic application execution.

tuning network is able to produce a performance improvement which takes into account a complete vision of the parallel application. This is achieved by the two main processes of the proposed model for large-scale tuning. The decomposition of the application into domains allows local load balancing, and the abstraction mechanism between levels in the tuning network permits the load imbalance that exists between domains to be rectified.

Figure 6.15 presents the application iteration time for the executions of synthetic applications ranging from 256 to 16 384 tasks. Each graph shows the evolution of the application performance with and without ELASTIC performing analysis and tuning. The third line shows the theoretical ideal iteration time if the load in the application were perfectly balanced.

The patterns reflected by the iteration time of the applications without ELASTIC present show how the introduced load imbalance affects the application performance. Since the iteration time is dependent on the computation time of the task with the maximum number of work units in the synthetic application, this pattern remains the same for all application sizes.

When the synthetic application is tuned by ELASTIC, the behaviour of the iteration time for all application sizes follows a similar pattern. Once the introduced imbalance is detected by ELASTIC, a period characterised by tuning operations takes place. After which the load imbalance reaches a level where ELASTIC decides no further tuning is necessary. It is this period of load migration where we will focus our attention.

As the size of the application increases it takes longer for the load balancing algorithm distributed hierarchically to spread the centralised additional load throughout the rest of the application grid. For this reason, the period of load migration lasts more iterations for larger applications before a stable state is reached. The peaks of additional imbalance that can be seen in the iteration times of larger application sizes are due to movement of load between domains as we previously explained in the description of the heatmaps.

Even though this kind of localised performance problem presents a challenge for ELASTIC's hierarchical tuning network as the application size grows, it is capable of resolving the load imbalance effectively. It should be noted that, as shown in the graphs, the number of iterations required to distribute the load throughout the application does not increase proportionally with the size of the application, highlighting the capabilities of the hierarchical tuning process.

The effects of the load balance performed by ELASTIC finally result in an improvement of the application performance, which is reflected in a reduction in the execution time of the synthetic parallel application. Figure 6.16 shows the execution times for each of the experiments presented in this section. The percentage presented above the second

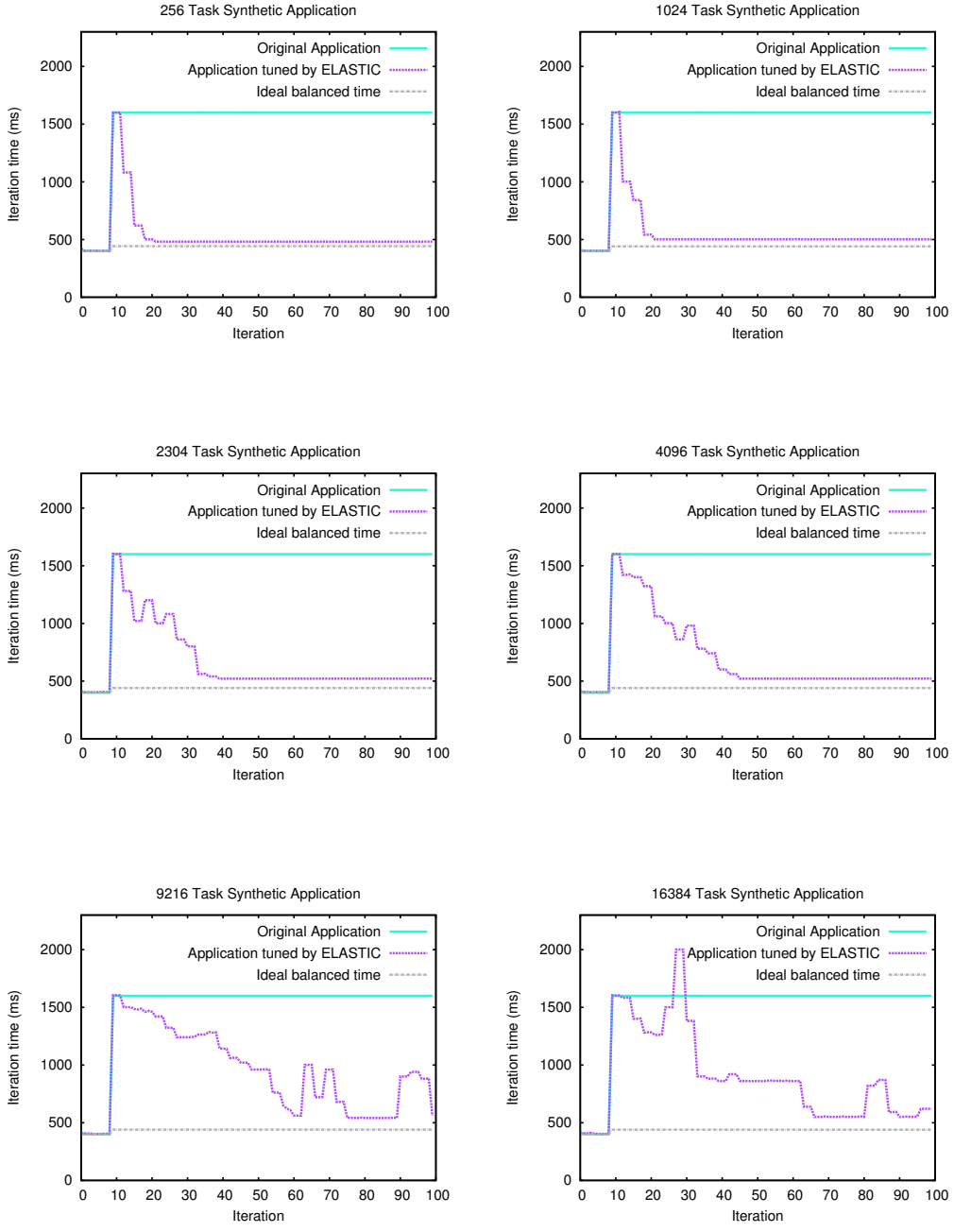


FIGURE 6.15: Iteration times for centralised hotspot imbalance for each parallel application size.

bar is the reduction in the execution time when the application is tuned by ELASTIC compared to the original application.

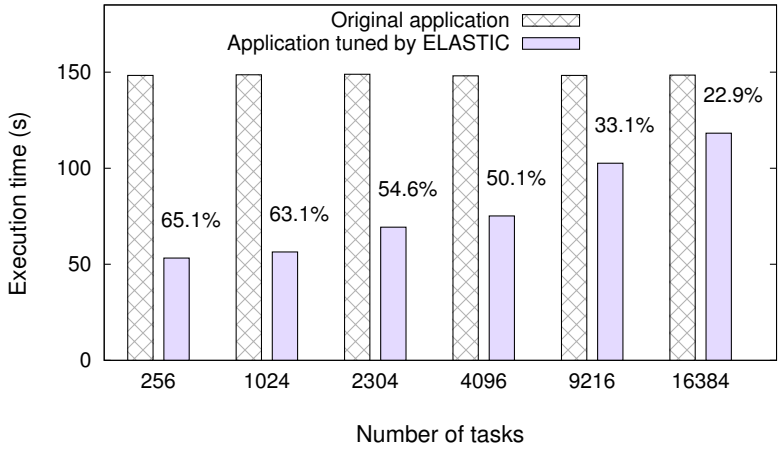


FIGURE 6.16: Execution time of the synthetic application with a single centralised hotspot.

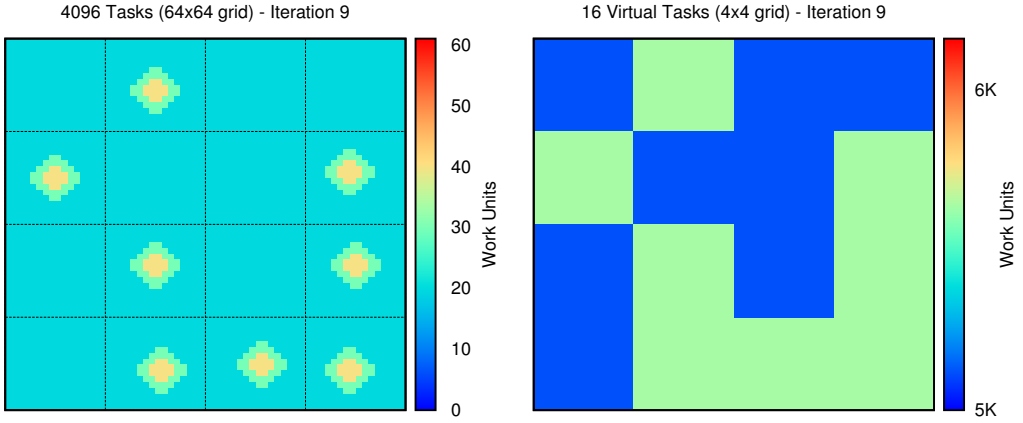
As can be seen, ELASTIC reduced the execution time of the parallel application for all sizes tested. This shows that our approach is able to manage the data generated during the monitoring process and use its tuning network to effectively detect the induced load imbalance and correct it in the synthetic application running on many thousands of processors.

Multiple Hotspots

In this scenario, ELASTIC faces a pattern of load imbalance characterised by multiple hotspots of load, distributed throughout the application grid, introduced at various moments of the synthetic application execution.

In Figure 6.17 the first introduction of imbalance in the case of the synthetic application composed of 4 096 tasks is presented. The smaller hotspots that are introduced can be observed in Figure 6.17 (a). Different to the centralised hotspot in the previous section, these hotspots appear as randomised imbalance from the point of view of the root ATM, shown in Figure 6.17 (b). This pattern repeats for the subsequent two injections of load imbalance.

In this scenario, the load state of the synthetic application changes due to migrations and due to the three injections of additional load. In Figure 6.18 the heatmaps show the process of load balancing resulting from the first injection of additional load. As can be seen, ELASTIC achieves a load balanced state after 3 migrations. Once again, the root



(a) (b)

FIGURE 6.17: First introduction of multiple hotspots in (a) the synthetic application and (b) the virtual application.

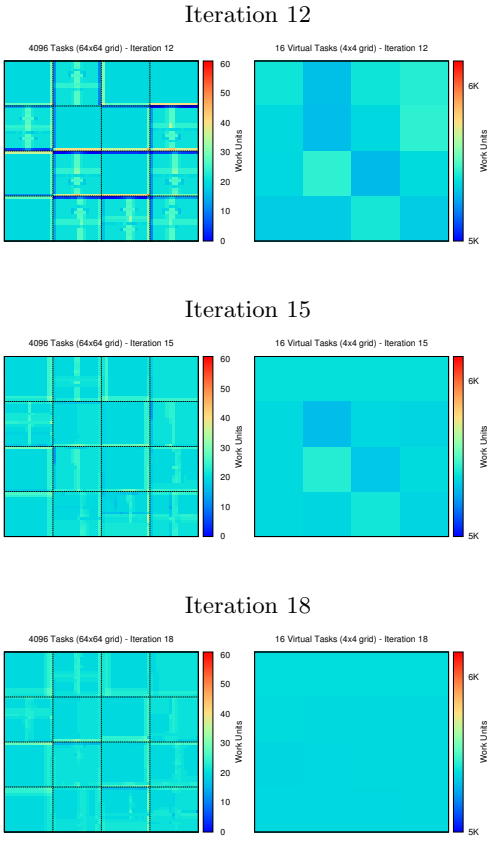


FIGURE 6.18: Heatmaps obtained after the first injection of additional load.

ATM balances the load between the domains. Meanwhile, the base level *Abstractor*-ATM pairs resolve the injected load imbalance as well as that provoked by the tuning actions of the root ATM.

Figure 6.19 depicts the heatmaps that reflect the load situation after the second injection of additional load. This includes the third injection, which is timed so that ELASTIC would not have had enough time to resolve the problems derived from the second injection of load. This situation demonstrates how ELASTIC is capable of reacting to dynamic changes throughout the application execution.

As in the previous scenario, Figure 6.20 presents individual graphs for each synthetic application size, showing the iteration time throughout the execution with and without ELASTIC performing analysis and tuning. The third line represents the theoretical ideal iteration time, if the application were perfectly balanced.

The behaviour of the synthetic application without ELASTIC shows the three distinct rises in the iteration execution time as the additional load is injected.

In all the cases where ELASTIC is tuning the synthetic application, the injection of additional load is countered by the dynamic tuning performed at each level in the tuning network. In these executions, the same general load pattern can be observed. In the case of 9 216 and 16 384 tasks, there is one last peak of instability before obtaining a load balanced state, which occurs after the final injection of additional load. This is caused by a migration of data from multiple domains to a single domain, ordered by the root ATM. This situation becomes more probable as the number of domains into which the application is decomposed increases.

The peaks of instability are caused by the naïve manner in which the performance and abstraction models operate with respect to the migration of work units between domains. Migrations between domains are performed with no specific knowledge of the state of the tasks which are located at the borders. Although the domain as a whole is underloaded, migrating large quantities of work units will cause these border tasks to become temporarily overloaded.

As can be seen from the iteration times for this type of imbalance, ELASTIC achieves a final balanced state at roughly the same time for each application size. This occurs between iterations 65 and 75. This differs from the centralised hotspot presented in the previous section, which required more iterations to balance the load as the size of the application increases. This difference highlights the strength of ELASTIC's hierarchical tuning network when resolving partially distributed performance problems.

The graph in Figure 6.21 shows how the difference can be appreciated in the execution times for the synthetic application with multiple distribution hotspots injected,

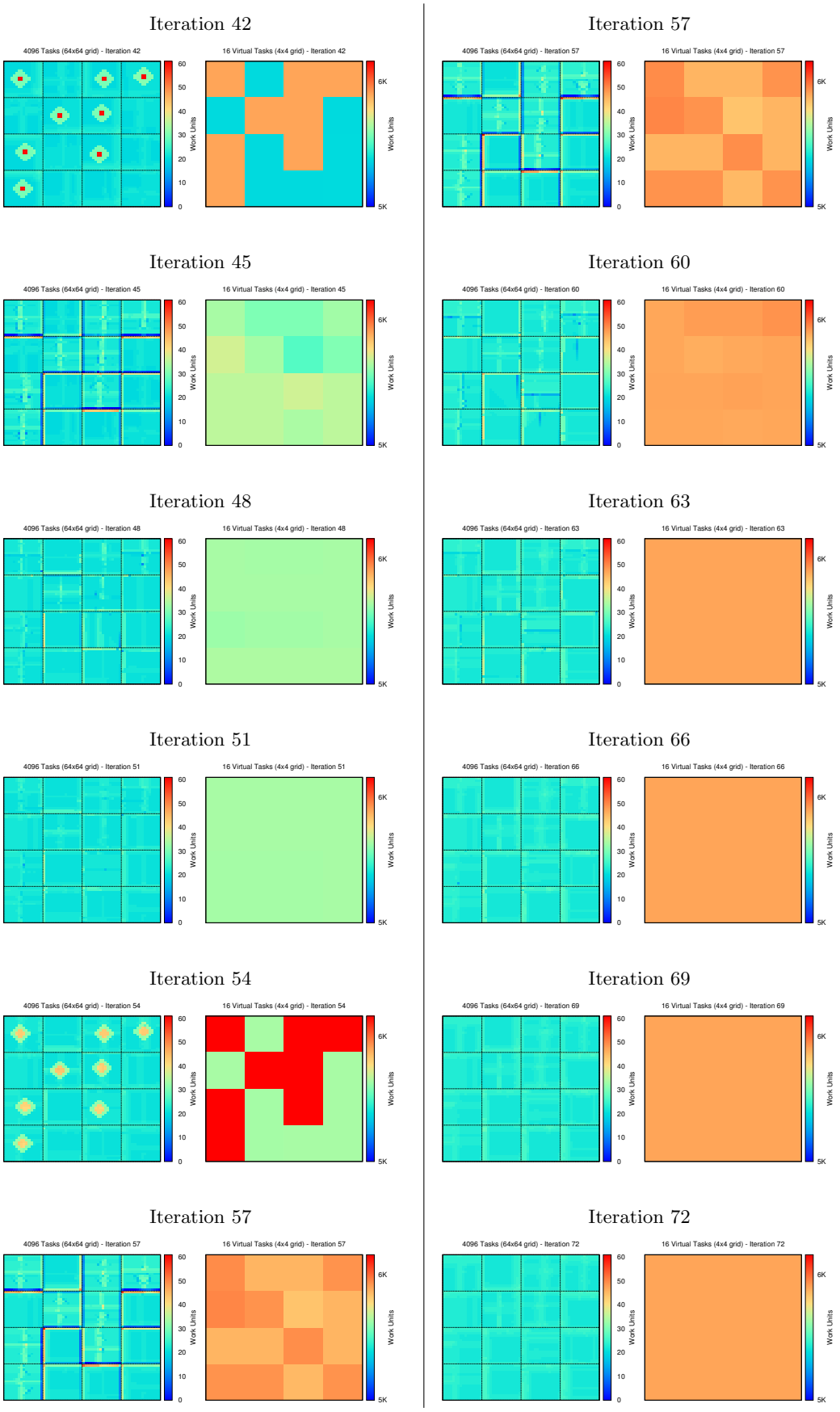


FIGURE 6.19: Heatmaps obtained after the second and third injection of additional load.

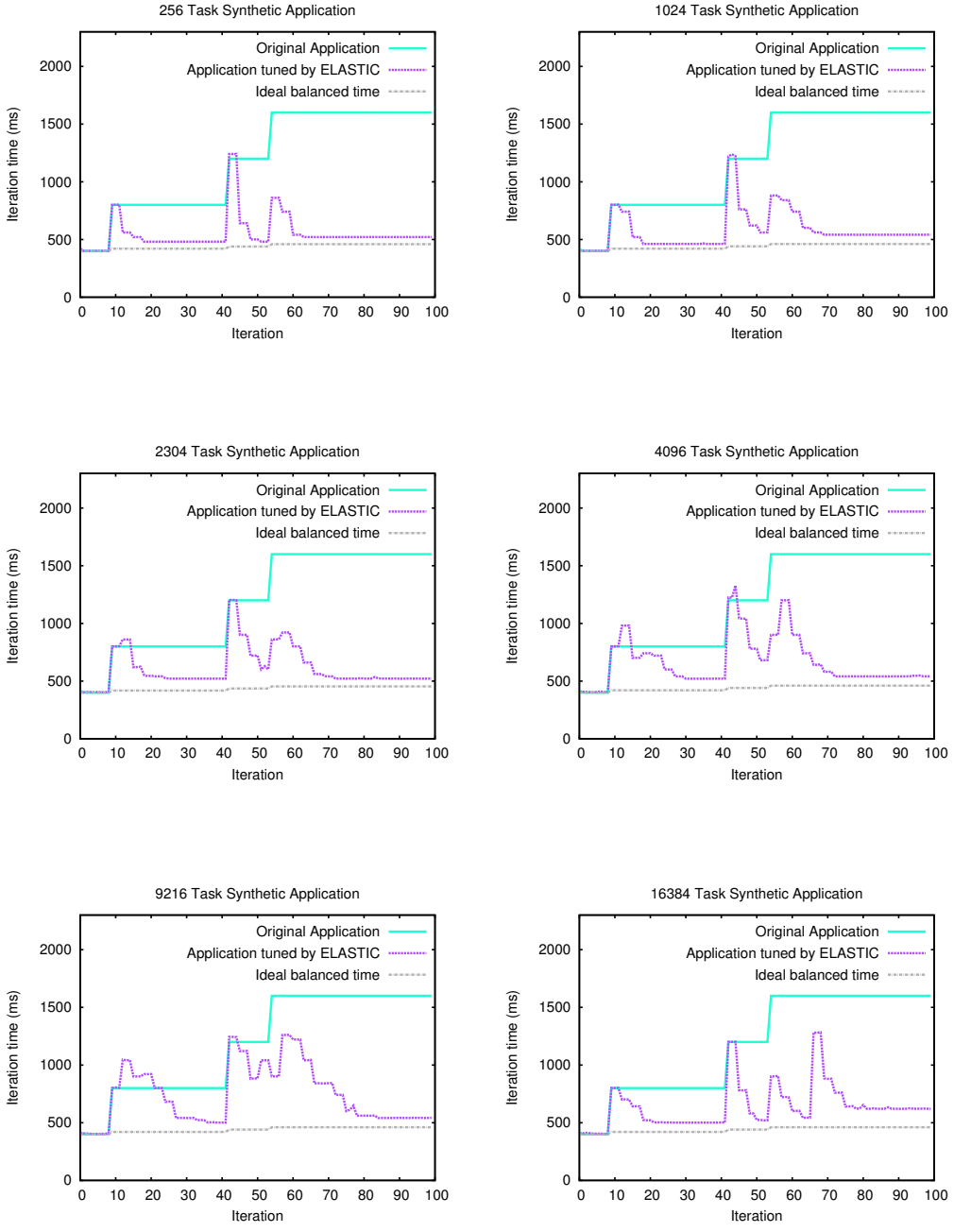


FIGURE 6.20: Iteration times for multiple hotspots of imbalance for each parallel application size.

for the original application and for the application tuned by ELASTIC. The percentage presented depicts the reduction in the execution time when the application is tuned by ELASTIC compared to the original application.

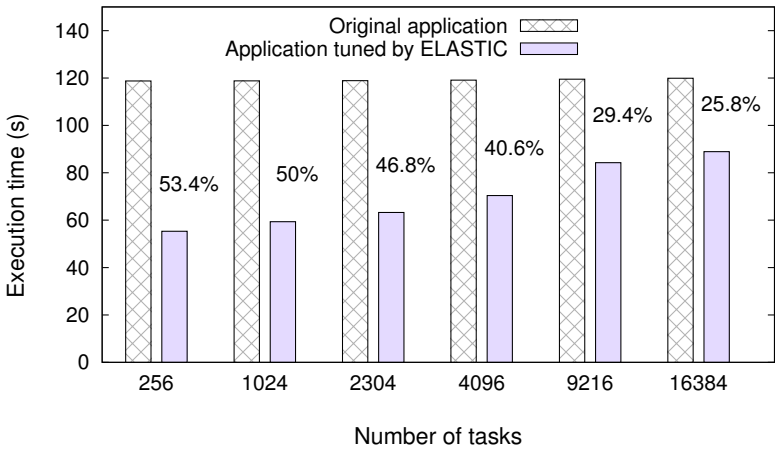


FIGURE 6.21: Execution time of the synthetic application with multiple distributed hotspots.

It can be seen that the reduction in the performance improvement gained with the size of the synthetic application is not as pronounced in this 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.

From these results, it can be observed how in this case ELASTIC is also capable of improving the performance of the synthetic application when the load dynamically changes throughout the execution.

6.3 Agent-Based Application

In this section, the model for hierarchical dynamic tuning proposed in this thesis and implemented in ELASTIC is applied to a real parallel application. The application is a large-scale agent-based simulation of a SIR epidemic model, named for the primary agent states: Susceptible (S), Infectious (I), and Recovered (R) and follows an SPMD paradigm.

The simulation describes how an epidemic spreads throughout a population. In this model, agents represent members of a population divided into three groups:

- Susceptible. These agents can contract the disease, but are not yet infected or immune.

- Infectious. These agents have been infected, and may pass the disease on to others.
- Recovered. These agents have recovered from the disease and are now immune.

During the simulation, the age of each agent is known, which is used to trigger reproduction and natural death in the population. As such, new agents are created as a consequence of birth and agents are removed due to death of natural causes or from the disease. The simulation takes places in a 2D toroidal grid.

This SIR model is implemented using the agent-based simulation tool FLAME (Flexible Large-scale Agent Modelling Environment) [25]. FLAME is an agent-based applications generator. Through model X-Machine Markup Language specification files and the implementation of the agent functions, FLAME automatically generates the simulation code in C.

FLAME enables parallel execution of agent-based simulations via MPI following an SPMD paradigm. Each task executes the same code, but operates over a different set of agents. At the beginning of the application execution, agents are assigned to parallel application tasks following a round-robin scheme.

In each iteration the parallel application tasks communicate in order to exchange the information that they will require to simulate the following iteration. The application follows an "any-to-any" communication pattern, wherein each task may exchange information with any other task in each iteration.

The communication between tasks arises from the interactions amongst the agents that they host. Agents have a geographic location in the simulation space. An agent only interacts with other agents that are geographically near. Therefore, tasks that contains agents which are near to one another will need to exchange information.

6.3.1 Performance Problem

During the course of the simulation, agents are created through births and they are eliminated because of death. When a new agent is created, it is allocated to the same task as its parent agent. The addition of agents to some tasks and elimination from others can lead to computation imbalance between tasks in the parallel application. Additionally, the computation time required to process each agent is not uniform. Due to the complex interactions between agents, some of them require more computation than others in each iteration.

Tasks with more computation to perform in each iteration will slow down the execution of the entire application as tasks with less computation must wait to perform inter-iteration synchronisation. This situation results in a similar performance problem

to that induced in the synthetic application. Furthermore, because the simulation is stochastic this imbalance will occur differently from one execution to the next. This makes this application suitable to apply dynamic tuning.

Figure 6.22 shows the level of imbalance, in terms of the computation time per iteration, present in an example execution using 1 024 parallel application tasks. It can be observed that the imbalance increases during the application execution. This is a tendency of the performance problem present in this application, where overloaded tasks tend to become more overloaded as the simulation progresses.

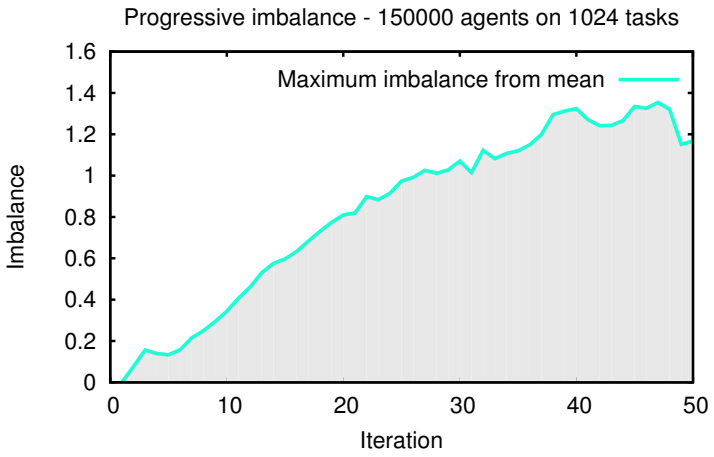


FIGURE 6.22: Increasing level of imbalance in the agent-based application executed on 1 024 tasks.

6.3.2 ELASTIC Package

The load imbalance problem in the agent-based application has been tackled employing ELASTIC with a specific ELASTIC Package. In this experimentation, the *Abstractor*-ATM pairs at all levels in the tuning network use the same ELASTIC Package.

In this instance, the ELASTIC Package is based on an algorithm for balancing the load in agent-based SPMD applications which is presented in [29]. This algorithm uses information about the number of agents and the computation time of each task in the application, to determine how to move agents between tasks in order to improve the application's performance.

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. For this reason, migration can be performed between any tasks in the application, and is not restricted in the same manner as was the case for the synthetic application.

The ELASTIC Package described in this section, embeds the functionality of this algorithm, which it performs over its analysis and tuning domain.

Performance Model

To guide the analysis and tuning process of the agent-based application, the performance model required has been defined based on the agent-based load balance algorithm previously introduced. In this section, it is detailed in terms of the measurement points, performance expressions, and tuning points, tuning actions and synchronisation method.

Measurement Points. The measurement points of the performance model are the inputs of the load balance algorithm. These inputs are the number of agents and the total computation time for each task in a given iteration. This information is contained in two variables of the application, `agent_count` and `iter_work`. The current iteration is also collected from the variable `iteration_id`.

There are four computation phases in each iteration. Therefore, the computation time per iteration is calculated as the sum of these phases, and will be available to be collected at the end of the final computation phase of each iteration, `phase_4()`. To reduce the number of events generated in the parallel application, the number of agents is also collected at this point.

The contents of the monitoring order generated is shown in Figure 6.23, following the *MonitoringOrder* class specification, detailed in Section 5.4.3.

eid	action	funcName	place	attrs
1	0	phase_4	0	[agent_count, iter_work, iteration_id]

FIGURE 6.23: Monitoring order to collect the required information from measurement points in the agent-based application.

A single monitoring order is sufficient to collect these three parameters, and therefore a single event containing the requested information is generated by each task in each iteration.

Performance Expressions. The performance expressions in this ELASTIC Package correspond with a set of rules and the evaluation of the load balance algorithm schema presented in [29].

The algorithm decides how to balance the load within an analysis and tuning domain by using the computation time and number of agents of each task in the domain, collected from the measurement points. An acceptable imbalance threshold is defined by the user, as the percentage deviation from the mean computation time. This imbalance threshold was set to $\pm 10\%$.

After each iteration, when the *Performance Evaluator* has received an event containing the required information from each task in the domain, the performance analysis

process is activated. If the imbalance detected is over the defined threshold, the load balancing algorithm is executed.

The algorithm operates by assigning agents to be migrated from the most overloaded tasks to the most underloaded tasks successively, until the projected load in all tasks is within the designated threshold. These migrations can be between any two tasks in the analysis and tuning domain. Each load balance algorithm execution provides a migration scheme that achieves a balanced state in the analysis and tuning domain.

In Algorithm 6.6, pseudocode describing the operation of the algorithm is presented.

Algorithm 6.6 Load balance scheme for agent-based application.

Input: N // Number of tasks in the analysis and tuning domain
Input: $iter\_work[N]$ // Computation time for each task
Input: $agent\_count[N]$ // Number of agents for each task
Output: $migration[N][N]$ // Agents to migrate from each task o to each task u. Initialised to 0.

```
avg_time ← ∑ iter_work[]/N
tolerance ← threshold * avg_time
proj_work[] ← iter_work[] // Create a copy of iter_work

if max_{0≤i≤N} {|iter_work[i] − avg_work|} > tolerance then
   centre ← least overloaded task
   o ← index of most overloaded task

   while proj_work[o] − avg_time > tolerance do
      exceeded_time ← proj_work[o] − avg_time
      time_per_agent ← iter_work[o]/agent_count[o]
      u ← index of most underloaded task

      while avg_time − proj_work[u] > tolerance do
         required_time ← avg_time − proj_work[o]
         migration[o][u] ← to_migrate(exceeded_time, required_time,
            time_per_agent)
         proj_work[o] ← proj_work[o] − (migration[o][u] * time_per_agent)
         proj_work[u] ← proj_work[u] + (migration[o][u] * time_per_agent)
      end while

      o ← index of most overloaded task
   end while

end if
```

The algorithm selects the most overloaded task, $task_o$, and the most underloaded task, $task_u$. A number of agents are selected to be migrated from $task_o$ to $task_u$ such that the projected load of one or both of these tasks is situated inside the threshold. Considering the projected load after the proposed migration, the new most underloaded task is designated $task_u$. If the projected load of $task_o$ remains above the threshold, then further agents are designated to be migrated to the new $task_u$, otherwise a new $task_o$ is selected and the algorithm continues.

Since the computation time of each agent is not fixed, to calculate the number of agents to migrate, the average computation time per agent in the overloaded task is

considered. It is this average time per agent that is used to calculate the projected computation time in both the sending and receiving tasks in a proposed migration. So, the number of agents to migrate from $task_o$ to $task_u$ is calculated in Algorithm 6.7.

Algorithm 6.7 Pseudocode for *to_migrate*() function.

Input: *exceeded_time*
Input: *required_time*
Input: *time_per_agent*
Output: *agents_to_migrate*

```
if exceeded_time < required_time then
  agents_to_migrate ← exceeded_time/time_per_agent
else
  agents_to_migrate ← required_time/time_per_agent
end if
```

Tuning Points, Tuning Actions and Synchronisation Method. The tuning operation in this case, consists of moving agents between parallel application tasks as directed by the performance expressions.

Currently, simulations generated by FLAME do not include the mechanism required to migrate agents from one task to another in order to distribute the computation equally between parallel application tasks. As such, additional code to perform the migration must be introduced dynamically into the application. This migration method is also taken from the work in [29].

The migration is performed in two phases. In the first phase a collective communication is conducted amongst all tasks in the domain in order to establish how many agents each task will receive from each other task in the same domain. Then, point-to-point communication is established to send the actual agent information between tasks. This migration operation takes place every 2 iterations, however if there are no agents to send, then only the first phase of collective communication is performed.

The tuning point in each task consists of an array, `intradomain_migrate`, that represents the number of agents to migrate to each task in the same domain. A second array `interdomain_migrate` is another tuning point, which represents the number of agents to migrate to the corresponding tasks of other domains. The tuning action associated with these variables corresponds to dynamically setting their contents with the result of the evaluation of the performance expression. This action takes place at the beginning of the migration process (synchronisation).

Abstraction Model

In this section, the abstraction model associated with the agent-based load balancing performance model is described. This includes how the application is decomposed into

domains as well as the event creation and instrumentation order translation processes.

Decomposition Scheme. When tuning the agent-based application, there is no concept of application task nearness because the migration follows an "any-to-any" pattern. Because the concept of contiguous blocks of tasks has no meaning, the decomposition of the application into domains does not have any restriction with respect to which tasks should be placed in the same domain. So, an ATM at the base of the hierarchy will be able to treat any grouping of application tasks as an *analysable and tunable* domain.

In order for the domains to be "abstractable", the only requirement placed on the decomposition process is that all the domains contain the same number of application tasks. As such the domains are homogeneous, in the same way that the underlying processors are homogeneous.

Figure 6.24 illustrates an example of an agent-based application composed of 16 tasks. Although the tasks have been placed in a grid, the distribution is not relevant. The domains could be formed of (a) blocks of tasks, (b) rows of tasks, or (c) other configurations. These three examples are functionally equivalent as the groups contain the same number of tasks.

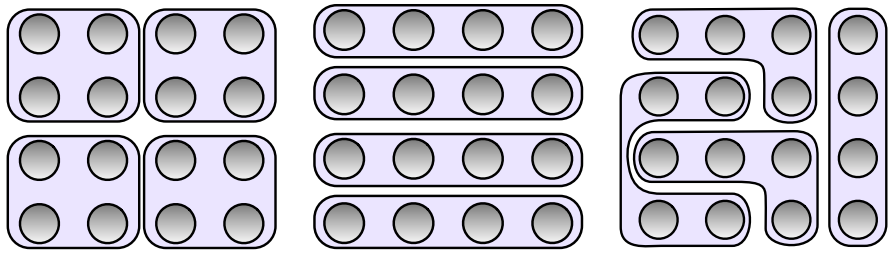


FIGURE 6.24: Examples of valid decomposition schemes for the agent-based application.

Monitoring Order Translation. Since the same ELASTIC Package is used at all levels in the hierarchy, all ATMs generate the same monitoring order requesting the same information. As such, when an *Abstractor* receives a monitoring order coming from its parent ATM, it only needs to register that order with the Event Manager module. This is the same behaviour presented in the abstraction model in the synthetic application case.

Event Creation. In order to present itself as a virtual application task, the *Abstractor* must provide its parent ATM with the information that was requested by the previously received monitoring order. In this case, the required information is the number of agents, the computation time, and the ID of the iteration for which the information has been collected.

To create a new event, an *Event Creator* requires one event from each task in the domain. The number of agents is calculated as the sum of all the agents hosted in tasks in the analysis and tuning domain of the *Abstractor's* associated ATM. The mean of the computation time of all tasks is used as the virtual task's computation time. Similar to the case for the synthetic application, a rank is assigned to each *Abstractor*-ATM pair at the beginning of ELASTIC's execution.

An example of the event creation process is given in Figure 6.25. An analysis and tuning domain with 4 tasks is considered. Once the *Event Creator* receives 4 events for the same iteration, a new event can be created. In this figure the events are shown in terms of the *Event* class detailed in Section 5.4.3.

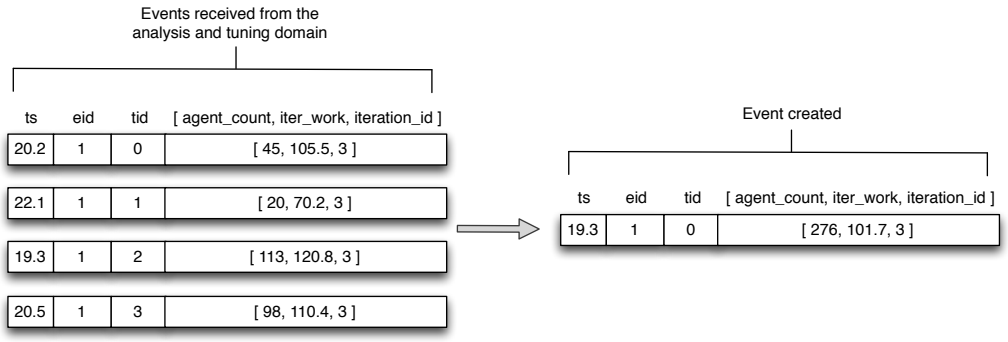


FIGURE 6.25: Event creation example for the agent-based application.

As in the case of the synthetic application, the timestamp of the new event, ***ts***, is the minimum of the received events. The event ID, ***eid***, corresponds to the ID of the associated monitoring order that requested the information contained in the event, and the rank, ***tid***, is that of the *Abstractor*-ATM pair, which is 0 in this example.

Tuning Order Translation. In this ELASTIC Package, the tuning orders sent by ATMs contain the number of agents to migrate between tasks of its analysis and tuning domain. When these tasks belong to a virtual application, these orders are received by the *Abstractor* representing this virtual task and translated to be applied to the analysis and tuning domain of its associated ATM.

The abstraction model must translate orders to migrate agents from one *Abstractor*-ATM pair to another into orders that affect the domains that these *Abstractors* represent. This translation is performed in such a way that every task in the sending domain will send a number of agents to a distinct task in the receiving domain, i.e. no two sending tasks migrate agents to the same receiving task and vice versa. The agents to be sent are divided evenly between all the tasks in the sending domain, and so all the tasks in the receiving domain will receive the same number of agents. The tuning order translation is illustrated with an example in Figure 6.26.

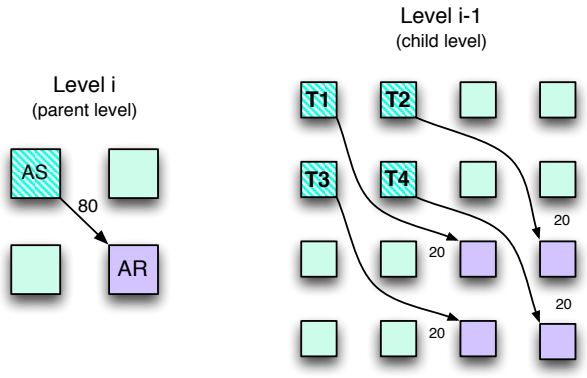


FIGURE 6.26: Tuning order translation diagram for the agent-based application.

In this example, a tuning order from level $i+1$ directing an *Abstractor* **AS** at level i to migrate 80 agents to *Abstractor* **AR**. This order is translated into a tuning order for each of the 4 tasks (T1, T2, T3, T4), at level $i-1$, in the domain represented by **AS**. Each new tuning order directs a task in this domain to migrate 20 agents to its corresponding task in the domain represented by **AR**. If level i is not the base level in the tuning network, then the tasks at level $i-1$ are actually also *Abstractor*-ATM pairs, and the tuning order translation process will be repeated in each one.

In Figure 6.27, this example of tuning order translation is presented in terms of the *SetVariableValueTuningOrder* class, defined in Appendix A.

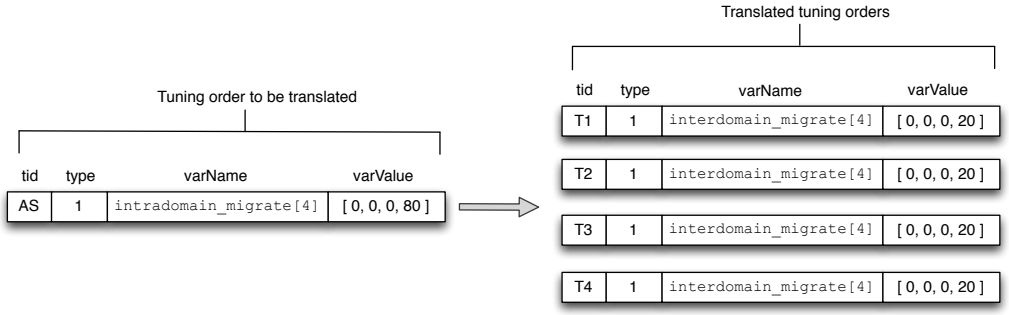


FIGURE 6.27: Example of *SetVariableValueTuningOrder* translation for the agent-based application.

As **AS** and **AR** are in the same analysis and tuning domain, the order received by **AS** is to set the value of the array `intradomain_migrate`, which specifies intra-domain migrations. However, due to the abstraction mechanism, the translated orders are to set the value of the array `interdomain_migrate`, which implies migrations between tasks of different domains.

Since there are 4 virtual tasks at level i and 4 domains at level $i-1$, the size of both `intradomain_migrate` and `interdomain_migrate` is 4. Additionally, the rank of a virtual task and the domain that it represents will be the same. For this reason, the

agents to migrate to **AR** in `intradomain_migrate` are in the same position of this array as those to be migrated to the domain that it represents in `interdomain_migrate`.

It should be noted that only the receiving domain, and not the receiving task within it, needs to be specified. This is because the pairs of tasks that communication between two domains are predefined in the migration code.

This abstraction model uses the same simplified coherence mechanism that was presented in Section 6.2.2. This is because inter-domain migrations involve interactions with different tasks rather than intra-domain migrations.

6.3.3 Tuning Network Topology

Tests using the agent-based application were executed using configurations composed of different numbers of tasks. Therefore, for each application size the topology of ELASTIC's tuning networks was calculated. This calculation was done following Algorithm 3.1 explained in Section 3.5.1, that permits the creation of topologies composed of ATMs that are not saturated.

The calculation of the topologies is based on parameters that characterise the analysis and tuning process. The values of these parameters, which have been obtained from specific measurements or from the configuration of the agent-based application, are depicted in Table 6.3.

TABLE 6.3: Values of variables required to calculate tuning network topology for the agent-based application.

Variable	Value	Description
E_a	1	# events from each child node required for analysis
E_c	1	# events from each child node required for event creation
$T_a(N)$	0.0003 ms	analysis time (quadratic)
T_m	0.02 ms	management time
T_c	0.2 ms	event creation time
T_t	0.3 ms	instrumentation order translation time
f_e	- events/s	frequency of event generation in the parallel application
f_{rc}	f_e batches/s	frequency of event reception from each child node
f_{rp}	f_e orders/s	frequency of tuning order reception from the parent ATM

Parameters Calculation

The load balancing process is activated in an ATM when it receives one event from each task in its analysis and tuning domain, so $\boldsymbol{E_a}$ takes the value 1. Using an event from each task in the domain, a new event is created and sent to the parent ATM, as such $\boldsymbol{E_c}$ is equal to 1.

The computation time of a single iteration depends on the agents residing in a task and the complexity of their interactions in that iteration. This complexity is influenced by the number of agents in the entire simulation. For this reason, the iteration time, and as such the frequency of event generation, is different for each simulation scenario. In this experimental evaluation, a simulation scenario exists for each parallel application size. The information that describes these scenarios can be found in the following section.

To calculate the event generation frequency, $\boldsymbol{f_e}$, it was necessary to execute the agent-based application and measure iteration times to gain an idea of the minimum iteration time for a task in a balanced execution. A good estimation of the minimum iteration time can be made by calculating the average time required to process an agent across the entire application and multiplying this by the average number of agents per task (a balanced situation). This calculation is performed for each iteration, and the minimum is chosen because it represents the maximum event generation frequency. This value is given for each application size in Table 6.4.

TABLE 6.4: Event generation frequency for each agent-based parallel application size.

Application Size	$\boldsymbol{f_e}$
256	4.7 events/s
512	3.1 events/s
1 024	2.3 events/s
2 048	1.4 events/s

As the event creation rate E_c is equal to 1, the frequency with which events are received from the child tasks in each level of the tuning network is equal to the event generation frequency in the application, i.e. $\boldsymbol{f_{rc}} = f_e$. The analysis rate E_a of 1, has the same effect on the frequency with which orders are received from the parent level, and so $\boldsymbol{f_{rp}} = f_e$. Therefore, the values of $\boldsymbol{f_{rc}}$ and $\boldsymbol{f_{rp}}$ change for each scenario as shown in Table 6.4.

The values of the variables which represent the time required to perform tasks in the *Abstractor*-ATM pair, $\boldsymbol{T_m}$, $\boldsymbol{T_c}$, $\boldsymbol{T_t}$ and $\boldsymbol{T_a(N)}$, have been measured by executing the agent-based application with different numbers of tasks and a two level tuning network operating over it. The tuning network requires two levels to perform these measurements, so that the time of the inter-level operations (event creation and order translation) can be obtained. The value of analysis time, $\boldsymbol{T_a(N)}$ is quadratic in the number of tasks in the analysis and tuning domain of an ATM.

Topology Calculation

The values of these variables are employed to calculate the topology with the minimum number of non-saturated ATMs for ELASTIC's tuning network. Because the same

ELASTIC Package is used at each level and event creation rate is 1, the same values can be used to calculate the maximum number of tasks that an ATM at any level in the hierarchy can support without becoming saturated. Following the first iteration of Algorithm 3.1 in Section 3.5.1, the following expression is used to calculate this maximum value N_{max}. This value has to be calculated for each different f_e using the formula below:

$$N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_{rp} = \frac{1}{f_a}$$

$$N_{max} \cdot E_a \cdot T_m + T_a(N_{max}) + \frac{E_a}{E_c} \cdot T_c + T_t \cdot f_e = \frac{E_a}{f_e}$$

$$\left(\frac{0.0003}{1000}\right) N_{max}^2 + \left(\frac{0.02}{1000}\right) N_{max} + \left(\frac{0.2}{1000} + \frac{0.3}{1000} \cdot f_e - \frac{1}{f_e}\right) = 0$$

Using the quadratic formula we can solve N_{max} for each application size and its f_e, the results are shown in Table 6.5.

TABLE 6.5: N_{max} for each agent-based parallel application size.

Application Size	N_{max}
256	792
512	990
1 024	1159
2 048	1501

So, the base level *Abstractor*-ATM pairs are able to support analysis and tuning domains of up to the calculated number of tasks, N_{max}, without becoming saturated.

These values for N_{max} would give centralised tuning network topologies for the first three scenarios. However, constraints in the underlying execution environment do not permit many more than 512 children per *Abstractor*-ATM pair. This is because MRNet spawns two threads for each connection, which can cause performance issues when the number of threads per available core grows too large. This limit can depend on the event reception frequency as well as thread support in the operating system and hardware.

In this experimental evaluation, the number of children per *Abstractor*-ATM pair has been limited to 512. Because the event creation rate is 1, N_{max} for the level 1 ATM will also be set to 512. Therefore, the topology for the application composed of 1 024 and 2 048 tasks will be composed of two levels.

The application sizes and the topology of the tuning network used to analyse and tune the application are given in Table 6.6.

The 256 task and 512 task applications are analysed and tuned by a centralised tuning network, on the other hand the applications composed of 1 024 and 2 048 tasks

Table 6.6: ELASTIC tuning network topologies over the agent-based application.

Number of Application Tasks	Level 0 Number of ATMs	Level 1 Number of ATMs
256	1	-
512	1	-
1 024	2	1
2 048	4	1

are analysed and tuned by two level tuning networks, with the root ATM controlling 2 and 4 base level *Abstractor*-ATM pairs respectively.

6.3.4 Effectiveness Evaluation

In this section we test the effectiveness of our model for hierarchical tuning using it to improve the performance of the agent-based parallel application.

The simulations were performed with a set of agent parameters defined in Table 6.7.

Table 6.7: Agent parameter configuration for all simulations.

Agent Parameter	Value
Starting number of infectious agents	10
Agent lifespan (iterations)	100
Average number of offspring	4
Probability of disease transmission	0.6
Probability of recovery	0.5
Disease duration (iterations)	20

For each size of the parallel application the number of agents in the simulation was scaled accordingly. Furthermore, the simulated space size was increased to avoid simulations characterised by a high infection rate and subsequent mass agent die-off. The details of these scenarios are shown in Table 6.8. The experiments were performed during 50 simulation iterations.

Table 6.8: Simulation scenarios for each size of the agent-based application.

Number of Application Tasks	Number of Agents	Simulated Space Size
256	37 500	1020×1020
512	75 000	1440×1440
1 024	150 000	1800×1800
2 048	300 000	2240×2240

In the available execution environment, issues were encountered that prevented the execution of applications composed of more than 2 048 tasks. Additionally, we were

unable to find reference in the literature of successful experimentation using FLAME based simulations at this scale.

Each of these scenarios were executed with and without ELASTIC performing dynamic tuning over the application using the ELASTIC Package detailed in Section 6.3.2.

Figure 6.28 shows the computation time for each iteration for each of the executed scenarios. This computation time for each task includes only the time required for the computation phases, and does not include communication between tasks to synchronise the simulation state or migration time to balance the load. Specifically, the graphs show the maximum computation time of all tasks, which dictates the iteration time for the application as a whole.

Each graph displays three computation times, the original agent-based application, the application tuned by ELASTIC and a perfectly balanced "ideal" computation time.

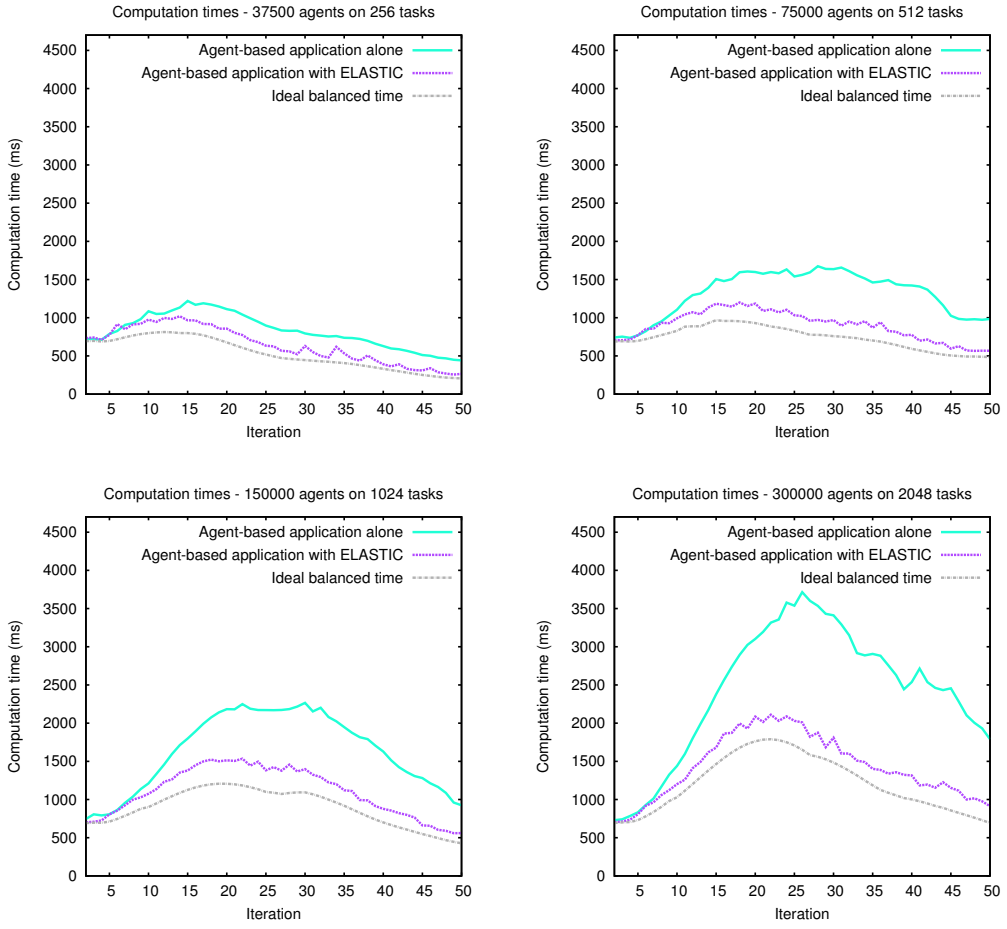


FIGURE 6.28: Computation times for the agent-based application with and without ELASTIC.

As the graphs in Figure 6.28 show, the original application presents serious load imbalance issues, as is evident from the difference between the application computation

time and the ideal. Moreover, this imbalance increases with larger sized applications. On the other hand, the computation time for the application tuned by ELASTIC remains much closer to the ideal time. This highlights the effective operation of ELASTIC to dynamically balance the load between the tasks of the agent-based application.

In this application, ELASTIC does not provoke peaks of instability due to movement of load between domains. This is because of the strategy used to move the load between domains, in which there is no concept of distinct borders between domains. Since every task in a domain will receive an equal number of agents, global improvements do not provoke local performance degradation. This behaviour presents an ideal situation in which to take advantage of the hierarchical tuning conducted by ELASTIC.

The benefit of balancing the load and reducing the computation time can also be appreciated in the total application execution time, shown in Figure 6.29. This figure presents the comparative execution times for the different simulation scenarios and the percentage that expresses the reduction in the execution time when the application is tuned by ELASTIC compared to the original application. The execution time shown includes all aspects of the simulation, both the computation and the inter-task communication, as well as the time required for the migration of agents in the case of the application tuned by ELASTIC. The times are given for the original application, and for the application tuned by ELASTIC.

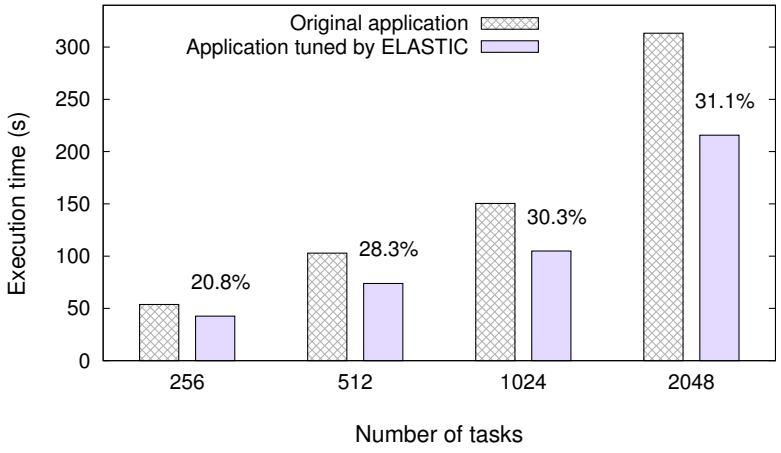


FIGURE 6.29: Execution times of the agent-based application with and without ELASTIC.

The same behaviour that was seen in the computation time graphs is repeated here. Although, the scalability of the original agent-based application is limited, ELASTIC is able to provide considerable performance improvements for all simulation scenarios. In fact, the reduction in execution time as a percentage of the original execution time increases as the application grows. This is due to the increasing imbalance in larger size applications which was apparent in Figure 6.28.

In this experimental evaluation, it has again been demonstrated that ELASTIC is not only able to scale with the size of the application, but is also able to provide effective dynamic tuning via the employed ELASTIC Package.

6.4 ELASTIC Overhead

An important consideration for any dynamic tuning tool is the amount of overhead that it introduces into the application being analysed and tuned. This intrusion should be limited so that the benefits of improving the parallel application's performance are not outweighed by the overhead.

The overhead in ELASTIC can be divided into three categories:

- Instrumentation overhead. This intrusion is related to the **modification** of the running application via the DynInst API. It consists of inserting or removing code snippets to generate events, and the modifications that fulfil the requirements of the tuning orders.
- Event tracing overhead. The generation of events implies the **execution** of code from the TMLib in order to collect information from the monitoring points and transmit it to the *Event Collector*.
- Concurrent BE execution. Each BE runs on the same core as the application task that it controls. The BE and the application task must **contend** for resources.

Additionally, overhead may be introduced due to the execution of user-generated snippets that are required by the tuning process. In this situation, the overhead does not come from ELASTIC, but from the specific tuning strategy being employed. For example, the migration phase in each of the applications presented in this chapter can be considered overhead of this type.

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 executing the migration phase. 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 by ELASTIC.

Table 6.9 shows the execution times for each application size executed with ELASTIC (but without migrations) and without ELASTIC. The overhead is given as a percentage of the execution time without ELASTIC.

TABLE 6.9: ELASTIC overhead measured in the synthetic application.

Number of Application Tasks	With ELASTIC (seconds)	Without ELASTIC (seconds)	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

As can be seen, 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.

Due to the small amount of overhead measured it has not been possible to determine how much influence each overhead category has in this total intrusion. For this reason, the origin of the slight increase in the overhead with the size of the parallel application is undetermined. However, it is believed that this is related to the number of parallel application tasks with the maximum number of work units amongst all other tasks.

If the intrusion in these tasks, caused by the instrumentation of tuning orders, falls within the work phase, then it could delay the completion of this phase, thereby increasing the iteration time of the entire application. If this occurs in a task which has fewer than the maximum number of work units, the iteration time of the application will not be affected, as this task will have to wait for tasks with more work. In the centralised hotspot scenario, larger applications have more tasks with the maximum number of work units. So, there is a greater probability that an intrusion of this kind will delay each iteration as the size of the parallel application grows.

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.

It was not possible to isolate the overhead introduced into the agent-based application. The execution time in the scenarios presented varied by approximately 5% between runs, which can be attributed to the complex inter-task communications performed by this application. It was found that there was no distinguishable difference between executions with ELASTIC (but without performing migrations) and the original application. The overhead remains unknown, but must be significantly less than the 5% variance between executions.

6.5 Discussion

In this chapter we have presented experimental evaluations using ELASTIC, an environment that implements the hierarchical tuning model proposed in this thesis. The evaluations have been carried out using a synthetic SPMD application as well as a real agent-based parallel application.

The experimental evaluation demonstrates that ELASTIC is able to scale to dynamically tune large-scale parallel applications, achieving performance improvements. As its name suggests, ELASTIC's architecture can adapt to the size of the parallel application being analysed, as well as the requirements of the ELASTIC Package used to conduct performance analysis.

The quality of the results achieved using ELASTIC depends on the intelligence integrated into the ELASTIC Packages. However, it is ELASTIC which permits the employment of this intelligence to tune large-scale parallel applications.

Two performance problems related to load imbalance have been addressed. However, the resolution of other kinds of performance problems using ELASTIC is completely viable, owing to its plugin architecture.

The overhead that ELASTIC introduces into the application in order to perform dynamic tuning has been evaluated, and shown to be minimal compared to the execution time of the applications studied.

To conclude, the results highlight the viability of using the proposed model for hierarchical tuning and, by extension, ELASTIC in the area of large-scale dynamic tuning.

7

Conclusions

"Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop."

– Lewis Carroll, *Alice in Wonderland*

This chapter presents the experiences gained and conclusions derived from this thesis. We also describe the viable open lines that can be considered in the future in order to continue evolving in the area of large-scale dynamic tuning of parallel applications.

7.1 Conclusions

The defining aspect of this thesis is that it provides a conceptually simple, yet powerful approach to apply dynamic tuning to large-scale parallel applications. This approach employs a decentralised scheme for performance analysis and tuning, which presents inherently scalable qualities that a centralised approach will never be able to provide.

The spectacular growth that hardware has experienced in recent years has led to two principal challenges in the performance analysis and tuning area. Firstly, performance tools must be developed that are capable of analysing and correcting performance problems at execution time, and secondly, these tools must be able to match the scalability of the parallel applications they propose to analyse, potentially, up to tens of thousands of processes.

Currently, there are no approaches which meet both these challenges and provide scalable dynamic tuning of parallel applications. The work developed in this thesis is focused on filling this void.

We began our work by researching well-known available dynamic tuning tools. An important aspect of this study was to determine where the scalability limits reside in these tools and what methods, if any, they had used to attempt to overcome them. The focus of our investigation was then turned to existing tools that conduct performance analysis (without tuning at runtime), which are able to operate in large-scale contexts. In this field, we found approaches often presented a decentralised design. This provided us with inspiration on how to face the challenge of scalability in our own area, dynamic tuning.

Considering the lessons learnt and with our objectives in mind, we created a model which consists of an efficient decentralised, but coordinated, approach to automatically and dynamically analyse and tune large-scale parallel applications. In this model, parallel applications which are too large to be analysed and tuned in a centralised manner are decomposed into disjoint subsets of tasks, that can be operated upon individually.

To view the application as a whole an abstraction mechanism was devised, which permits the representation of each subset as a single "task", which together form a virtual parallel application. For large parallel applications, the decomposition and abstraction is repeated until the size of the virtual application is such that it can be managed in a centralised manner. This gives rise to the hierarchical structure of the proposed model. The virtual application at the top level in the hierarchy enables analysis using a coarse global view of the application state. At lower levels in the hierarchy, analysis is restricted to subsets of the application, but using more detailed information about the application state.

In order to offer effective dynamic tuning, we decided to follow a collaborative approach rather than attempt *blind* dynamic tuning. This collaborative approach requires the integration of knowledge into the model, to guide the performance analysis and tuning and to define the abstraction process. We codified the required knowledge in the form of a *performance model* and an *abstraction model*.

As the next step towards developing a dynamic tuning tool, we translated the model into a hierarchical tuning network. The nodes of the network are analysis modules, that represent the model's virtual parallel application tasks and conduct the distributed performance analysis and tuning. The hierarchical tuning network has been defined in such a way that its topology (the number of levels and the number of analysis modules per level) can be adapted to the complexity of the performance and abstraction models and the number of tasks in the parallel application being tuned.

To accompany this adaptability, we also developed a method that addresses the challenge of calculating network topologies that are composed of the minimum number of analysis modules. Since dynamic tuning tools are active during the application execution, they need additional resources to reduce their influence on the tuned application. In a time where the use of resources, and the associated energy costs, are carefully controlled, providing a method that determines how many additional resources are necessary is more valuable then ever.

Having completed the hierarchical tuning network design, we focused on the scalability of the proposed model. We carried out a scalability study using a simulation environment which implements the tuning network's hierarchical communication and simulates the analysis, tuning and abstraction processes. To validate the scalability we measured the global decision time of the tuning network. The global decision time is the time required for the root node to detect a performance problem in the application being analysed. In the scalability study we revealed that the global decision time presents a logarithmic growth with the size of the parallel application, which demonstrates the scalability of the proposed model for dynamic tuning.

At this point, we wished to prove that the proposed model, when it takes the form of a tool, is able to provide effective dynamic tuning over large parallel applications. Therefore, our development concluded with the implementation of ELASTIC, a tool for large-scale dynamic tuning. ELASTIC offers dynamic tuning through monitoring (based on event tracing), performance analysis, and dynamic modifications. These three operations are performed automatically and continuously during the application execution. To provide monitoring and tuning at execution time ELASTIC uses dynamic instrumentation.

In order to guide the dynamic tuning in ELASTIC, we gave it a plugin architecture. The plugins take the form of ELASTIC Packages, which are a set of code and configuration that allow a tuning strategy, specified in terms of performance and abstraction models, to be employed to improve the performance of a parallel application.

Finally, we concluded this work with an experimental evaluation of ELASTIC and through it the proposed model for large-scale dynamic tuning. The tests were performed over a synthetic application composed of up to 16 384 tasks and a real parallel application composed of up to 2 048 tasks. The synthetic application had load imbalance introduced in order to evaluate ELASTIC's reaction to different scenarios. The real application is an agent-based simulation which suffers from load imbalance due to the agent life cycle. For each case, an ELASTIC Package was developed to resolve a load imbalance performance problem affecting the application. In both cases, ELASTIC is shown to not only scale to meet the demands of dynamic tuning over thousands of processes, but also effectively improve the performance of the applications tested.

The results of the experimental evaluation have been very encouraging and indicate the potential of ELASTIC as a tool for parallel application performance improvement. Consequently, it is shown that the main contribution of this thesis, the model for hierarchical dynamic tuning, is a viable solution to the problem of performing dynamic tuning over large-scale parallel applications.

7.2 Future Work

The work presented in this thesis will allow for further investigation into specific dynamic tuning techniques for large-scale parallel applications.

The most directly related extension of this work is the creation of additional ELASTIC Packages which model different performance issues. Thanks to the plugin architecture which ELASTIC provides, a valuable resource could be created in the form of generalised packages which solve a given performance problem, and which would require only small adaptations in order to be applied to a specific parallel application which exhibits this problem.

This research could also be used to further validate the proposed model for hierarchical dynamic tuning with applications based on distinct parallel programming paradigms, which present different performance problems to be resolved.

The AutoTune project [36] also offers exciting possibilities for ELASTIC Package creation. The extensive search capabilities of AutoTune identify the aspects of an application which most heavily influence its performance. This knowledge could then be

directly employed to construct the performance model required by an ELASTIC Package.

An open line of research arises from the calculation of tuning network topologies. It would be beneficial to generalise the proposed method to calculate topologies for other approaches that perform online automatic or dynamic performance analysis in large-scale contexts. This would allow resource usage tradeoffs to be compared between tools and assist users in selecting the appropriate configuration given their individual requirements.

This proposed method could calculate efficient tuning network topologies that include criteria other than minimal usage of resources. Topologies chosen to balance multiples objectives could be considered. For example, a maximum time to initiate a tuning action could be specified in conjunction with criteria to reduce the energy consumption of the tuning network resources. In such a situation, finding a good compromise between these constraints when calculating the topology would provide an effective, but energy efficient tuning environment.

7.3 List of Publications and Grants

The work and motivation for this thesis have been published in the following papers:

1. **A. Morajko, A. Martínez, E. César and J. Sorribes.** ***MATE: Towards Scalable Automated and Dynamic Performance Tuning Environment*** **in Proceedings PARA 2010: State of the Art in Scientific and Parallel Computing, Reykjavik, 2010.** [42]

   This work presents a series of ideas that attempt to discover new possibilities to overcome the scalability barriers present in the centralised dynamic tuning tool MATE.

2. **A. Martínez, A. Morajko, E. César and J. Sorribes.** ***Dynamically Tuning Master/Worker Applications with MATE*** **in XXII Jornadas de Paralelismo 2011, pp. 609-614.** [30]

   This paper focuses on a methodology to be followed to perform dynamic tuning on master-worker applications using MATE.

3. **A. Martínez, A. Sikora, E. César and J. Sorribes.** ***Tuning Master/-Worker Applications: A Practical Use Case with MATE*** **accepted in International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA 2013.**

This work presents the possible benefits that can be achieved when tuning master-worker applications using a tool such as MATE. Specifically, MATE acts following a performance model which determines the optimal number of workers and work division size.

4. **A. Martínez, A. Sikora, E. César and J. Sorribes. *Hierarchical MATE's Approach for Dynamic Performance Tuning of Large-Scale Parallel Applications* in Proceeding of the EEE International Performance Computing and Communication Conference, pp. 191-192, 2012.** [31]

   In this paper, the initial design for an approach to perform hierarchical dynamic tuning in terms of MATE's structure is presented.

5. **A. Martínez, A. Sikora, E. César and J. Sorribes. *How to Scale Dynamic Tuning to Large-Scale Applications* in International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), pp. 355-364, 2013.** [32]

   This paper presents the complete model for hierarchical dynamic tuning. A prototype of what would become ELASTIC is presented in order to validate the scalability of the proposed model. In addition, concepts relating to the calculation of efficient tuning network topologies are discussed.

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 International Workshop on Automatic Performance Tuning (IWAPT) - ICCS, pp. 1352-1361, 2013.** [33]

   In this work, the full method to determine tuning networks composed of the minimum number of resources is described. An experimental evaluation is also presented which shows how different analysis and tuning patterns influence the selected topology.

In addition, the following grants were received during the course of this work:

- Project: *Scalable and Dynamic Performance Tuning for Large-Scale Parallel Applications*. Access to the supercomputer MareNostrum via the Spanish Supercomputing Network (*Red Española de Supercomputación*). We gained access to 30 000 CPU hours from March to May, 2012.
- Project: *Scalable and Dynamic Performance Tuning for Large-Scale Applications*. Access to the supercomputer JUGENE via the PRACE Preparatory project. We gained 250 000 CPU hours from April to November, 2012.
- Project: *Scalable and Dynamic Performance Tuning for Large-Scale Applications*. Access to the resources of the High Performance Computing Center of Stuttgart

(Germany) via the HPC-Europa Transnational Access visit. Part of this grant involved a research stay at Technische Universität München from September to November, 2012.

By invitation, part of the work included in this thesis was presented at the Paradyn/HTCondor Week 2013. An invitation has also been accepted to present this work at the Dagstuhl Seminar on *Automatic Application Tuning for HPC Architectures*.

Acknowledgements

We thankfully acknowledge the computer resources, technical expertise and assistance provided by the Red Española de Supercomputación and Leibniz Supercomputing Centre.

Bibliography

[1] C. Armstrong, R. W. Ford, J. R. Gurd, M. Luján, K. R. Mayes, and G. D. Riley. Performance Control of Scientific Coupled Models in Grid Environments. *Concurrency and Computation: Practice and Experience*, 17(2-4):259–295, Feb. 2005. ISSN 1532-0626. doi: 10.1002/cpe.v17:2/4.

[2] D. Arnold, D. Ahn, B. De Supinski, G. Lee, B. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Proceeding of 21th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007. doi: 10.1109/IPDPS.2007.370254.

[3] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003. ISSN 0272-1732. doi: 10.1109/MM.2003.1196112.

[4] S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: An Online-based Distributed Performance Analysis Tool. In *Parallel Tools Workshop*, pages 1–16, 2009. doi: 10.1007/978-3-642-11261-4_1.

[5] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. In *Proceeding of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, Sept. 2010. ISBN 978-1-4244-7913-9. doi: 10.1109/ICPP.2010.18. Best Paper Award.

[6] M. Brim and B. Miller. Group File Operations for Scalable Tools and Middleware. In *Proceeding 16th International Conference on High Performance Computing, HiPC*, pages 69–78, 2009. doi: 10.1109/HIPC.2009.5433223.

[7] 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.

[8] H. Brunst, A. D. Malony, S. Shende, and R. Bell. Online Remote Trace Analysis of Parallel Applications on High-Performance Clusters. In *Proceedings of*

the 5th International Symposium on High Performance Computing, volume 2858 of *Lecture Notes in Computer Science*, pages 440–449. Springer, 2003. doi: 10.1007/978-3-540-39707-6_39.

[9] H. Brunst, W. E. Nagel, and A. D. Malony. A Distributed Performance Analysis Architecture for Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, Cluster 2003, pages 73–83. EEE Computer Society, 2003. doi: 10.1109/CLUSTR.2003.1253301.

[10] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000. ISSN 1094-3420. doi: 10.1177/109434200001400404.

[11] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple Page Size Modeling and Optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 339–349. IEEE Computer Society, 2005. ISBN 0-7695-2429-X. doi: 10.1109/PACT.2005.32.

[12] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and Environment Monitoring for Continuous Program Ooptimization. *IBM J. Res. Dev.*, 50(2/3):239–248, Mar. 2006. ISSN 0018-8646. doi: 10.1147/rd.502.0239.

[13] P. Caymes-Scutari. *Extending the Usability of a Dynamic Tuning Environment.* PhD thesis, Universitat Autnoma de Barcelona, 2007.

[14] E. César, A. Moreno, J. Sorribes, and E. Luque. Modeling Master/Worker Applications for Automatic Performance Tuning. *Parallel Computing*, 32(7):568–589, Sept. 2006. ISSN 0167-8191. doi: 10.1016/j.parco.2006.06.005.

[15] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy.* PhD thesis, University of Southern California, 2007.

[16] K. Chen, K. R. Mayes, and J. R. Gurd. Autonomous Performance Control of Distributed Applications in a Heterogeneous Environment. In *Proceedings of the 1st international conference on Autonomic computing and communication systems*, Autonomics '07, pages 14:1–14:5, Rome, Italy, 2007. ISBN 978-963-9799-09-7. doi: 10.1145/1365562.1365581.

[17] I.-H. Chung. *Towards Automatic Performance Tuning.* PhD thesis, University of Maryland, 2004.

[18] W. Dorland, F. Jenko, M. Kotschenreuther, and B. N. Rogerszle. Electron Temperature Gradient Turbulence. *Physical Review Letters*, 85, Dec. 2000.

[19] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Further Improving the Scalability of the Scalasca Toolset. In *Proceeding of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 2010. doi: 10.1007/978-3-642-28145-7_45.

[20] J. Guevara, E. César, J. Sorribes, A. Moreno, T. Margalef, and E. Luque. A Performance Tuning Strategy for Complex Parallel Application. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP*, pages 103–110, 2010. doi: 10.1109/PDP.2010.34.

[21] J. K. Hollingsworth and B. P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *Proceedings of the 7th international conference on Supercomputing*, ICS '93, pages 185–194. ACM, 1993. ISBN 0-89791-600-X. doi: 10.1145/165939.165969.

[22] K. A. Huck, A. D. Malony, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 473–482. IEEE Computer Society, 2005. ISBN 0-7695-2380-3. doi: 10.1109/ICPP.2005.29.

[23] M. Hussein, K. Mayes, M. Luján, and J. Gurd. Adaptive Performance Control for Distributed Scientific Coupled Models. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 274–283. ACM, 2007. ISBN 978-1-59593-768-1. doi: 10.1145/1274971.1275009.

[24] R. Jain. The Art of Computer Systems Performance Analysis. *SIGMETRICS Performance Evaluation Review*, 18(3):21–22, 1990.

[25] M. Kiran, P. Richmond, M. Holcombe, L. Shawn Chin, D. Worth, and C. Greenough. FLAME: Simulating Large Populations of Agents on Parallel Hardware Architectures. In *Proceeding of 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1633–1636, 2010. doi: 10.1145/1838206.1838517.

[26] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In *Parallel Tools Workshop*, pages 139–155, 2008. doi: 10.1007/978-3-540-68564-7_9.

[27] C. W. Lee, A. D. Malony, and A. Morris. TAUmon: Scalable Online Performance Data Analysis in TAU. In *Proceeding of International Euro-Par Conference Workshops*, pages 493–499, 2010. doi: 10.1007/978-3-642-21878-1_61.

[28] A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon. Performance Analysis and Grid Computing. chapter Advances in the TAU performance system, pages 129–144. Kluwer Academic Publishers, 2004. ISBN 1-4020-7693-2.

[29] C. Márquez, E. César, and J. Sorribes. A Load Balancing Schema for Agent-based SPMD Applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Accepted*, 2013.

[30] A. Martínez, A. Morajko, E. César, and J. Sorribes. Dynamically Tuning Master/Worker Applications with MATE. In *XXII Jornadas de Paralelismo*, pages 609–614, 2011.

[31] A. Martínez, A. Sikora, E. César, and J. Sorribes. Hierarchical MATE's Approach for Dynamic Performance Tuning of Large-Scale Parallel Applications. In *Proceedings of IEEE International Performance Computing and Communications Conference, IPCCC*, pages 191–192, 2012. doi: 10.1109/PCCC.2012.6407696.

[32] 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*, pages 355–364, 2013. doi: 10.1109/IPDPSW.2013.31.

[33] 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*, pages 1352–1361, 2013. doi: 10.1016/j.procs.2013.05.302.

[34] 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):1975–1986, 2005. doi: 10.1098/rsta.2005.1607.

[35] K. Mayes, M. Elliot, A. Manning, D. Haglin, and J. Gurd. A Distributed Search Infrastructure for Statistical Disclosure Control on a Grid. In *Proceedings of the Second International Conference on e-Social Science*, June 2006.

[36] 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)*, volume 7782, pages 328–342, 2012. doi: 10.1007/978-3-642-36803-5_24.

[37] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel

Performance Measurement Tool. *Computer*, 28(11):37–46, Nov. 1995. ISSN 0018-9162. doi: 10.1109/2.471178.

[38] B. Mohr, B. Wylie, and F. Wolf. Performance Measurement and Analysis Tools for Extremely Scalable Systems. *Concurrency and Computation: Practice and Experience*, 22:2212–2229, 2010. doi: 10.1002/cpe.1585.

[39] A. Morajko. *Dynamic Tuning of Parallel/Distributed Applications*. PhD thesis, Universitat Autnoma de Barcelona, 2003.

[40] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: Monitoring, Analysis and Tuning Environment for Parallel/Distributed Applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2005. ISSN 1532-0626. doi: 10.1002/cpe.v19:11.

[41] A. Morajko, T. Margalef, and E. Luque. Design and Implementation of a Dynamic Tuning Environment. *Journal of Parallel and Distributing Computing*, 67:474–490, 2007. ISSN 0743-7315. doi: 10.1016/j.jpdc.2007.01.001.

[42] A. Morajko, A. Martínez, E. César, T. Margalef, and J. Sorribes. MATE: Toward Scalable Automated and Dynamic Performance Tuning Environment. In *PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable Tools for High Performance Computing*, pages 430–440, 2010. doi: 10.1007/978-3-642-28145-7_42.

[43] A. Moreno, E. César, A. Guevara, J. Sorribes, and T. Margalef. Load Balancing in Homogeneous Pipeline based Applications. *Parallel Computing*, 38(3):125–139, Mar. 2012. ISSN 0167-8191. doi: 10.1016/j.parco.2011.11.001.

[44] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[45] A. Nataraj, M. J. Sottile, A. Morris, A. D. Malony, and S. Shende. TAUover-Supermon: Low-Overhead Online Parallel Performance Monitoring. In *Proceeding of 13th International Euro-Par Conference*, pages 85–96, 2007. doi: 10.1007/978-3-540-74466-5_11.

[46] A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, Apr. 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:6.

[47] C. Oehmen and J. Nieplocha. ScalaBLASt: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Transactions*

on Parallel Distributed Systems, 17(8):740–749, Aug. 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.112.

[48] R. L. Ribler, H. Simitci, and D. A. Reed. The Autopilot Performance-Directed Adaptive Control System. *Future Generation Computer Systems*, 18(1):175–187, 2001. doi: 10.1016/S0167-739X(01)00051-6.

[49] P. C. Roth. *Scalable On-line Automated Performance Diagnosis*. PhD thesis, University of Wisconsin-Madison, 2005.

[50] P. C. Roth and B. P. Miller. On-line Automated Performance Diagnosis on Thousands of Processes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 69–80, New York, USA, 2006. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122984.

[51] 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*, page 21, 2003. ISBN 1-58113-695-1. doi: 10.1145/1048935.1050172.

[52] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006. ISSN 1094-3420. doi: 10.1177/1094342006064482.

[53] L. Silva and R. Buyya. *Parallel Programming Models and Paradigms*. Monash University, Melbourne, Australia, 1998.

[54] R. Smith and P. Gent. Reference Manual for the Parallel Ocean Program (POP). In *Tech. Rep. LAUR-02-2484, Los Alamos National Laboratory*, 2002.

[55] M. J. Sottile and R. G. Minnich. Supermon: A High-Speed Cluster Monitoring System. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 39–46. IEEE Computer Society, 2002. ISBN 0-7695-1745-5. doi: 10.1109/CLUSTR.2002.1137727.

[56] 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*, pages 1–11, 2003. doi: 10.1145/762761.762771.

[57] A. Tiwari and J. K. Hollingsworth. Online Adaptive Code Generation and Tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 879–892. IEEE Computer Society, 2011. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.86.

[58] A. Tiwari, V. Tabatabaee, and J. K. Hollingsworth. Tuning Parallel Applications in Parallel. *Parallel Computing*, 35(8-9):475 – 492, 2009. ISSN 0167-8191. doi: 10.1016/j.parco.2009.07.001.

[59] Top 500 List. In *http://www.top500.org/*, Jun 2013.

[60] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, and M. Hauswirth. Performance and Environment Monitoring for Whole-System Characterization and Optimization. In *Proceedings of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC), Yorktown Heights*, pages 15–24, 2004.

[61] B. J. N. Wylie, D. Böhme, W. Frings, M. Geimer, B. Mohr, Z. Szebenyi, D. Becker, M.-A. Hermanns, and F. Wolf. Scalable Performance Analysis of Large-Scale Parallel Applications on Cray XT Systems with Scalasca. In *Proceeding of 52nd Cray User Group Meeting, Edinburgh, Scotland*. Cray User Group Incorporated, May 2010.

[62] B. J. N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, and F. Wolf. Large-Scale Performance Analysis of Sweep3D with the Scalasca Toolset. *Parallel Processing Letters*, 20(4):397–414, Dec. 2010. doi: 10.1142/S0129626410000314.

A

Tuning Order Subclasses

The TuningOrder base class specifies a type property which should be set to indicate one of the seven available types of tuning orders. The properties which are available for each of these subclasses are defined in this appendix.

For the sake of completeness, the TuningOrder class definition is repeated below, followed by the definitions for each of the subclasses..

- **TuningOrder Class** ***(subclass of Order)***

  Properties

  - `int type` - represent the type of the tuning order to be applied.

    0 - empty tuning order.

    1 - `SetVariableValue`.

    2 - `ReplaceFunction`.

    3 - `InsertFunctionCall`.

    4 - `OneTimeFunctionCall`.

    5 - `RemoveFunctionCall`.

    6 - `FunctionParamChange`.

    7 - `LoadLibrary`.

- **SetVariableValueTuningOrder Class** ***(subclass of TuningOrder)***

  Changes the value of a global variable in the application process.

  Properties

  - `string varName` - the name of the variable to change the value of.
  - `string varValue` - the new value to assign to this variable.

- **ReplaceFunctionTuningOrder Class** ***(subclass of TuningOrder)***

  Replaces all calls to the function named `oldFunc` with calls to the function named `newFunc`.

  Properties

  - `string oldFunc` - the name of the function to be replaced.
  - `string newFunc` - the name of the new function where calls will be directed.

- **InsertFunctionCallTuningOrder Class** ***(subclass of TuningOrder)***

  Inserts a call to the function named `funcName` with parameters `attrs`. The call is inserted into the function named `destFunc` at either the entry or the exit.

  Properties

  - `string funcName` - the name of the function to be called.
  - `vector<string> attrs` - the parameters to call the function with.
  - `string destFunc` - the name of the function where the call is to be inserted.
  - `int place` - point of instrumentation in `destFunc`, 1 - entry of the function, 0 - exit of the function.

- **OneTimeFunctionCallTuningOrder Class** ***(subclass of TuningOrder)***

  Inserts a function call in exactly the same way as the InsertFunctionCallTuningOrder class, but the inserted function call is automatically removed after it is called for the first time.

  Properties

  - `string funcName` - the name of the function to be called.
  - `vector<string> attrs` - the parameters to call the function with.
  - `string destFunc` - the name of the function where the call is to be inserted.
  - `int place` - point of instrumentation in `destFunc`, 1 - entry of the function, 0 - exit of the function.

- **RemoveFunctionCallTuningOrder Class** ***(subclass of TuningOrder)***

  Removes all calls to the function named `funcName` from within the function `callerFunc`.

  Properties

  - `string funcName` - the name of the function to which calls are to be removed.
  - `string callerFunc` - the name of the function from which the calls are to be removed.

- **FunctionParamChangeTuningOrder Class** ***(subclass of TuningOrder)***

  Changes the value of a single parameter on the function named `funcName`. The parameter with index `paramIdx` is assigned `newValue` before the body of the function is called. If `requiredOldValue` is set, then the parameter must have previously had this value for the change to take place.

  Properties

  - `string funcName` - the name of the function to change the parameters of.
  - `int paramIdx` - the index of the parameter to changed in the function.
  - `string newValue` - the new value to call this function with.
  - `string requiredOldValue` - if this parameter is set, the value will only be changed if the parameter had this value.

- **LoadLibraryTuningOrder Class** ***(subclass of TuningOrder)***

  Loads the dynamic library located at `libraryPath` into the application task's memory space.

  Properties

  - `string libraryPath` - the file system path of the library to be loaded.