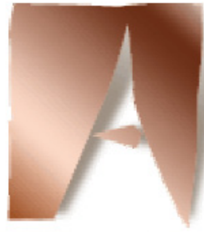


Universitat
Autònoma
de Barcelona

# AUTOMATIC DYNAMIC TUNING OF PARALLEL/DISTRIBUTED APPLICATIONS ON COMPUTATIONAL GRIDS


Computer Architecture
and Operating Systems
Departament


Thesis submitted by **Genaro Fernandes de Carvalho Costa** in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona.

Barcelona, May 29, 2009

# AUTOMATIC DYNAMIC TUNING OF PARALLEL/DISTRIBUTED APPLICATIONS ON COMPUTATIONAL GRIDS

Thesis submitted by **Genaro Fernandes de Carvalho Costa** in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona. This work has been developed in the **Computer Architecture and Operating Systems** department of the **Universitat Autònoma de Barcelona** and was advised by **Dra. Anna Morajko** and **Dr. Tomàs Margalef Burrull**.

Thesis Advisors

Anna Barbara Morajko                    Tomàs Margalef Burrull

Barcelona, July 22, 2009

*A Zequinha e Verbena*

*que me ensinaram que a vida*
*se construi com amor e dedicação.*
*E assim persigo sonhos,*
*amizades e realizaões.*

*A Tchukita,*

*essa tese é mais usa do que minha.*
*E pensar que tudo começou no*
*mar, nas estrelas e nos peixinhos.*
*Te amo.*

# Acknowledgements

First of all I would like to thank my parents Dr. Zequinha and Dona Bena for their love and teaching. They are always present with me wherever I go. Their unquestioning support and understanding motivates me and make me comfortable at trying new things. I would like to thank my fiancée Larissa Couto for her love and trust and for being on my side at all times. Without her nothing could be complete.

It is important to emphasize that this work could not have been done without many people's direct and indirect support, not just during this five-year period but throughout my entire life.

Thanks go to Emilio Luque for the opportunity of doing this work, Tomás Margalef and Ania Morajko, my advisors, for the support and key discussions. Without their persistence and continuous inquiries this work could not have been completed. Thanks also go to Josep Jorba and Eduardo Cesar for the frequent discussions that motivated this work and the CAOS group for many technical discussions over many fields.

I would especially like to thank to Eduardo Argollo for his great friendship and support, Angelo Duarte for the support and discussions about technology and Guna Alexander for not to complaining about many difficult situations! Thanks to Dolores Isabel (Lola) for her help in many ways and I would like to thank very much our CAOS group for providing the friendship and the information exchange which made this work possible. I also want to thank the PT for their infrastructure support. I give my thanks to Josemar Souza for bringing about cooperation between UAB and UCSal.

# Table of Contents

**CHAPTER 1 INTRODUCTION .......... 1**

1.1 OVERVIEW .......... 1

1.2 GOALS AND CONSTRAINTS .......... 5

1.3 THESIS ORGANIZATION .......... 7

**CHAPTER 2 COMPUTATIONAL GRIDS .......... 9**

2.1 OVERVIEW .......... 9

2.2 PARALLEL PROGRAMMING .......... 9
*2.2.1 Programming Models .......... 10*
*2.2.2 Performance Characterization .......... 12*

2.3 GRID MIDDLEWARE .......... 13
*2.3.1 Overview .......... 13*
*2.3.2 Globus Toolkit .......... 14*
*2.3.3 Condor .......... 20*

2.4 HETEROGENEOUS SCENARIOS .......... 22

2.5 MONITORING APPROACHES .......... 23
*2.5.1 Grid Monitoring Architecture – GMA .......... 23*
*2.5.2 System Monitoring .......... 26*
*2.5.3 Application Monitoring .......... 30*

**CHAPTER 3 GRID PERFORMANCE MODELS .......... 37**

3.1 RELATED WORK .......... 38

3.2 PERFORMANCE MODEL FOR DYNAMIC TUNING .......... 39
*3.2.1 Parameters Characterization .......... 43*
*3.2.2 Metrics for Grain Decomposition .......... 44*
*3.2.3 Grain Size and System Heterogeneity .......... 47*
*3.2.4 Dynamic Tuning Requirements and Process .......... 49*

3.3 TUNING IN HETEROGENEOUS SCENARIOS .......... 53

3.4 EFFECTS OF DATA ACCESS PATTERNS .......... 57

3.5 SIMULATING MASTER-WORKER IN HETEROGENEOUS SCENARIOS .......... 59

3.6 SUMMARY .......... 67

**CHAPTER 4 GMATE – GRID MONITORING ANALYSIS AND TUNING ENVIRONMENT .......... 69**

4.1 OVERVIEW .......... 69
*4.1.1 Problems .......... 70*
*4.1.2 Clock Synchronization .......... 73*

4.2 DYNAMIC TUNING ........ 76
*4.2.1 Active Harmony ........ 76*
*4.2.2 Autopilot ........ 77*
*4.2.3 MATE ........ 79*

4.3 GRID MONITORING ........ 81
*4.3.1 Design Architecture ........ 83*
*4.3.2 Process Tracking ........ 87*
*4.3.3 Monitoring Topology ........ 96*
*4.3.4 Smart Event Gathering ........ 101*

4.4 GRID PERFORMANCE ANALYSIS ........ 107
*4.4.1 Tunlet Architecture ........ 110*

4.5 GRID TUNING ........ 112
*4.5.1 Smart Tuning Actions ........ 115*
*4.5.2 Tuning in different layers ........ 120*

**CHAPTER 5 EXPERIMENTAL VALIDATION ........ 121**

5.1 INTRODUCTION ........ 121

5.2 MASTER-WORKER TUNING ON GRIDS ........ 122
*5.2.1 Framework Overview ........ 123*
*5.2.2 System Description ........ 123*
*5.2.3 Compute/Communication Dependency Analysis ........ 124*

5.3 APPLICATION CASE STUDIES ........ 126
*5.3.1 Synthetic Dynamic Master-Worker ........ 126*
*5.3.2 Matrix-Multiplication Application ........ 131*
*5.3.3 N-Body Application ........ 132*

5.4 ARCHITECTURE VALIDATION ........ 133
*5.4.1 Sensors Overhead Analysis ........ 133*

**CHAPTER 6 CONCLUSIONS AND FUTURE WORK ........ 137**

6.1 CONCLUSIONS ........ 137
*6.1.1 Process Location ........ 139*
*6.1.2 Security Polices ........ 140*
*6.1.3 Lower Communication Intrusion ........ 140*
*6.1.4 Middleware Integration ........ 141*
*6.1.5 Performance Models ........ 142*

6.2 OPEN LINES ........ 143
*6.2.1 Application Parallelism Support ........ 145*
*6.2.2 Data Type and Domain Semantic Definition ........ 147*
*6.2.3 Multi-Core Issues ........ 148*
*6.2.4 Moving to Cloud Computing ........ 148*

**BIBLIOGRAPHY ........ 151**

**APPENDIX ........ 161**

A. CLUSTERSIM – A MULTI-CLUSTER SIMULATOR ........ 161

B. GMWAT – A HIERARCHICAL MASTER/WORKER APPLICATION TEMPLATE WITH SUPPORT FOR DYNAMIC GRAIN SIZE SELECTION .......... 166

# List of Figures

FIGURE 1 – ORGANIZATION AND VIRTUAL ORGANIZATIONS IN A GRID. ........ 13
FIGURE 2 – GRID PROTOCOL ARCHITECTURE, FROM [2]. ........ 15
FIGURE 3 – GENERAL CONDOR KERNEL ARCHITECTURE WITH THE SEQUENCE OF INFORMATION FLOW ........ 21
FIGURE 4 - GRID MONITORING ARCHITECTURE COMPONENTS, FROM [8]. ........ 24
FIGURE 5 – COMPONENTS AND INTERACTION IN R-GMA, FROM [44]. ........ 27
FIGURE 6 – NWS ARCHITECTURE OVERVIEW, FROM [46]. ........ 29
FIGURE 7 – KOJAK TOOL ARCHITECTURE AS PRESENTED IN [49]. ........ 32
FIGURE 8 – COMPONENT ARCHITECTURE OF THE GRID DYNAMIC INSTRUMENTATION SERVICE, FROM [53]. ........ 33
FIGURE 9 – PARADYN OVERVIEW STRUCTURE [54]. ........ 35
FIGURE 10 – EXAMPLE OF A MASTER/WORKER STARTUP WHERE THE TIME SPENT IN EACH ROUND CORRESPONDS TO THE TIME EACH WORKER PROCESS ITS ASSIGNED GRAIN. AT TIME $T_1$ ALL WORKERS ARE PERFORMING COMPUTATIONS AND AT TIME $T_2$ MASTER NETWORK INTERFACE IS SATURATED ON BOTH INPUT AND OUTPUT COMMUNICATION CAPACITY CONSIDERING A FULL-DUPLEX NETWORK PORT [59]. ........ 40
FIGURE 11 – COMPARISON BETWEEN ATLAS AND GNU GSL EXECUTIONS OF A MATRIX MULTIPLICATION PROBLEM USING DIFFERENT GRAIN SIZES. ........ 41
FIGURE 12 – IMPACT OF GRAIN SIZE IN TOTAL APPLICATION EXECUTION TIME AND USAGE EFFICIENCY OF A FIXED RESOURCE SET SIZE OF 18 WORKERS. ........ 42
FIGURE 13 – IMPACT OF THE NUMBER OF WORKERS IN TOTAL APPLICATION EXECUTION TIME AND USAGE EFFICIENCY OF THE ASSIGNED RESOURCE SET. ........ 42
FIGURE 14 – ABSTRACTION OF THE LOAD THROUGH OPERANDS, OPERATIONS AND RESULT VALUES ........ 43
FIGURE 15 – GRAPHICAL VIEW OF THE RELATION BETWEEN DATA PARTITION AND LOAD DIVISION SCENARIO WHERE GRAIN SIZE HAS IMPACT ON COMPUTE/COMMUNICATION RATIO. ........ 44
FIGURE 16 – GRAPHICAL VIEW OF THE BEST STARTUP AND FINALIZATION PHASES. ........ 46
FIGURE 17 – IMPACT ANALYSIS OF HETEROGENEITY IN COMPUTE/COMMUNICATION TIME VALUES CONSIDERING NETWORK AND PROCESSOR CAPACITY VARIATIONS. ........ 48
FIGURE 18 – PRESENTS A GRAPHICAL VIEW OF *WHERE* AND *WHEN* MEASUREMENTS ARE GATHERED IN DIFFERENT PROCESSES. ........ 52
FIGURE 19 – GRAPHICAL VIEWS OF DIFFERENT POSSIBLE ASSIGNATIONS FROM GRID RESOURCE BROKER/META-SCHEDULER IN RESPONSE TO A REQUEST OF SEVEN COMPUTE NODES RESOURCE. ........ 55
FIGURE 20 – PSEUDO ALGORITHM OF OPTIMUM NUMBER OF WORKERS AND GRAIN SIZE TUNING. IT USES RUNTIME METRICS TO DECIDE IF AN APPLICATION NEEDS CHANGES IN THE COMPUTE/COMMUNICATION RATIO AND/OR CHANGE IN NUMBER OF WORKERS IN USE. ........ 56
FIGURE 21 – PLOT OF CACHE HIT RATIO BY NUMBER OF WORKERS CONSIDERING 256 WORK UNITS. ........ 59
FIGURE 22 – THIS PRESENTS THE SIMULATOR OUTPUT OF A MASTER/WORKER EXECUTION WITH DYNAMIC GRAIN SIZE CHANGE. BLUE BARS ARE SENDS AND GREEN ONES ARE RECEIVES. ORANGE BARS ARE TASK PROCESSING. THE NUMBERS INSIDE THE BARS ARE THE TASK NUMBER. NOTE THAT TASK 4 WAS PROCESSED USING A DIFFERENT GRAIN SIZE AND WORKERS 2 AND 3 HAVE DIFFERENT PROCESSOR POWER THAN WORKERS 0 AND 1. ........ 60
FIGURE 23 – OVERVIEW OF APPLICATION EXECUTION TIME REDUCTION BY USING DYNAMIC TUNING OF THE NUMBER OF WORKERS AND GRAIN SIZE IN ALL HETEROGENEOUS SCENARIOS AND THE NUMBER OF WORKERS ASSIGNED TO APPLICATION. ........ 62
FIGURE 24 – OVERVIEW OF RESOURCE EFFICIENCY INCREMENT AS RESULT OF DYNAMIC TUNING OF THE NUMBER OF WORKERS AND GRAIN SIZE IN ALL HETEROGENEOUS SCENARIOS AND THE NUMBER OF WORKERS ASSIGNED TO APPLICATION. ........ 64
FIGURE 25 – GAINS IN RESOURCE USAGE EFFICIENCY IN DISTINCT GROUPS OF PROCESSOR HETEROGENEITY. WE CONSIDER EFFICIENCY AS THE PERCENTAGE OF TOTAL COMPUTATION AVAILABLE PERFORMANCE USED DURING THE APPLICATION EXECUTION TIME. ........ 65
FIGURE 26 – GAINS IN TOTAL EXECUTION TIME CONSIDERING DIFFERENT GROUPS OF PROCESSOR HETEROGENEITY, AS SHOWN, ASSIGNED TO THE APPLICATION. ........ 65
FIGURE 27 – EFFICIENCY OF RESOURCE USAGE COMPARED AGAINST DIFFERENT HETEROGENEOUS PROCESSORS' DISTRIBUTION BETWEEN LAN AND WAN. ........ 66
FIGURE 28 – TOTAL EXECUTION TIME COMPARED BY HETEROGENEOUS PROCESSORS DISTRIBUTION BETWEEN LAN AND WAN. ........ 66

Figure 29 – Clock synchronization message exchange.......73
Figure 30 – Time difference between two stratum 2 sites to a client machine.......75
Figure 31 – Autopilot conceptual architecture [76] .......77
Figure 32 – Autopilot components and the iteration sequence among them. [76].......78
Figure 33 – Dynamic Monitoring, Analysis and Tuning Approach .......79
Figure 34 – MATE component architecture.......80
Figure 35 – Internal representation of the Analyzer. .......81
Figure 36 – Communication channels among GMATE components.......82
Figure 37 – Our monitoring scheme model in comparison to GMA .......84
Figure 38 – Connection between AC Wrapper instances and AC instances.......89
Figure 39 – Integration of the AC to Globus Toolkit – Initialization Phase. .......90
Figure 40 - Integration of the AC to Globus Toolkit – Runtime Phase. .......90
Figure 41 – Application detection using Grid integration. .......92
Figure 42 – Binary packaging structure. .......95
Figure 43 – Binary Packaging information block.......95
Figure 44 – Event structure representation. .......98
Figure 45 – Execution scenario which requires different gathering topology.......99
Figure 46 – Local event gathering strategy comparison. .......101
Figure 47 –Diagram for sensor concepts.......103
Figure 48 – Pseudo code of a Sensor instrumentation process. .......104
Figure 49 – Internals of the Dynamic Monitoring Library (DMTLib) and its iteration with the InstallPoints instrumented by DyninstAPI and the other GMATE components. .......108
Figure 50 – Internals of the Application Controller (AC) and its iteration with DyninstAPI and other GMATE components. .......109
Figure 51 – Internals of the Analyzer and its iteration with the ACs and other GMATE components. .......109
Figure 52 – Tunlet Interface. .......110
Figure 52 – Software modularity abstraction from [14].......114
Figure 53 – Communication/Administrative domains abstractions.......114
Figure 54 – Actuators concept. .......116
Figure 55 – Actuators internal state machine. .......117
Figure 56 – Pseudo code of an instrumentation process of a change value Actuator. .......118
Figure 57 – Pseudo code of an instrumentation process of a function call Actuator. .......119
Figure 58 – Benchmark of communications among different nodes in the constructed Grid testbed using the Intel Pallas MPI Benchmark [86]. .......123
Figure 59 – Ratio of different grain size division using GSL and ATLAS BLAS implementations. .......125
Figure 61 – Two iteration execution of synthetic Master-Worker where the master processor is mapped on cluster I. The phases A, B and C are startup, steady and finalization, respectively. .......128
Figure 62 – Two iteration of synthetic Master-Worker where the master processor is mapped on cluster P. The phases A, B and C are startup, steady and finalization, respectively.......128
Figure 62 – Histogram presenting the tasks send times from Master to Workers. .......129
Figure 63 – Comparison between execution with and without tuning grain size, considering the master mapped on cluster P. .......130
Figure 64 – Graphical representation of grain size change in the parallelization version of matrix multiplication problem. Each task can be decomposed into smaller ones with data reuse among tasks. .......131
Figure 65 – Comparison of same problem size and parallel machine configuration considering different Master-Worker processes to node mapping. .......132
Figure 66 – Graphical representation considering grain size in N-Body problem with uniform task load and input/output data size. .......133
Figure 67 – Source code from the instrumented program. .......134
Figure 68 – Model for simple sensor profile experiment. .......135
Figure 69 – Specification of the Timer Sensor from Simple Sensor. .......136
Figure 70 – Overhead of monitoring using Simple Sensors to time a function.......136
Figure 71 – Queue system used to simulate the ClusterSim Cluster Simulator .......161
Figure 72 – Screenshot for ClusterSim front end that enables the configuration of multiple grain sizes and the multiplicity of composition/decomposition.......163

FIGURE 73 – OUTPUT OF EVENTS FROM CLUSTERSIM PLOTTED USING GNUPLOT. THE NUMBERS IN THE BARS ARE THE TASK NUMBERS. THE BLUE BARS ARE SENDS AND THE GREEN BARS ARE RECEIVES. THE ORANGE BARS ARE PROCESSING EVENTS FROM WORKERS. ....... 164
FIGURE 74 – PRESENTS THE AMOUNT OF WORKERS BUSY BY TIME IN APPLICATION EXECUTION. ....... 165
FIGURE 75 – GUI FOR COMPLEX TOPOLOGY CONFIGURATION. ....... 165
FIGURE 76 – SEQUENCE DIAGRAM OF THE API EXPOSED FOR COMMUNICATION BETWEEN THE CLASSES *MPIHANDLER* AND *COMPUTEPROCESSOR*. IT ALSO PRESENTS RELEVANT TEMPLATE FUNCTION THAT CAN BE USED TO MEASURE TIME SPENT IN THE COMMUNICATION PROCESS. ....... 167
FIGURE 77 – CLASS DIAGRAM PRESENTING THE INTERNAL CONCEPTS OF THE PROCESS ROLES INSIDE THE FRAMEWORK. ....... 168
FIGURE 78 – CLASS DIAGRAM PRESENTING THE INTERNAL CONCEPTS OF THE LOAD INSIDE THE FRAMEWORK....... 169
FIGURE 79 – SEQUENCE DIAGRAM PRESENTING THE ITERATION AMONG MASTER AND WORKER PROCESSES INCLUDING THE SEQUENCE OF USER FUNCTION CALLING. ....... 170
FIGURE 80 – IT PRESENTS A MULTI-CLUSTER CONFIGURATION WITHIN A PARALLEL MACHINE WITH 8 PROCESSES. ....... 171

# List of Tables

TABLE 1 – PARAMETERS USED IN DYNAMIC TUNING WITHIN A GRID PARALLEL APPLICATION EXECUTION. .......................... 52
TABLE 2 – CLOCK SYNCHRONIZATION TIMESTAMP INFORMATION [75]. ........................................................................ 74
TABLE 3 – API USED IN PROCESS TO GATHER DATA AND SUPPORT SENSOR COMMUNICATION. ........................................ 85
TABLE 4 – SYSTEM CHARACTERISTICS FOR EACH CLUSTERS USED TO THE EXPERIMENTAL VALIDATION. ............................. 124

# Abstract

When moving to Grid Computing, parallel applications face several performance problems. The system characteristics are different in each execution and sometimes within the same execution. Remote resources share network links and in some cases, the processes share machines using per-core allocation. In such scenarios we propose to use automatic performance tuning techniques to help an application adapt itself: thus a system changes in order to overcome performance bottlenecks.

This thesis analyzes such problems of parallel application execution in Computational Grids, available tools for performance analysis and models to suit automatic dynamic tuning in such environments. From such an analysis, we propose system architecture for automatic dynamic tuning of parallel applications on computational Grids named GMATE. Its architecture includes several contributions. In cases where a Grid meta-scheduler decides application mapping, we propose two process tracking approaches that enable GMATE to locate where a Grid middleware maps application processes. One approach consists of the integration of GMATE components as Grid middleware. The other involves the need to embed a GMATE component inside application binaries. The first requires site administration privileges while the other increases the application binary which slows down application startup.

To obey organizational policies, all communications use the same application security certificates for authentication. The same communications are performed using Grid middleware API. That approach enables the monitoring and tuning process to adapt dynamically to organizational firewall restrictions and network usage policies.

To lower the communication needs of GMATE, we encapsulate part of the logic required to collect metrics and change application parameters in components that run inside the processing space. For metric collection, we create sensor components that reduce the communication by event inside the process space. Different from traditional instrumentation, sensors can postpone the metric communication and perform basic operations such as summarizations, timers, averages or threshold based metric generation. That reduces the communication requirements in cases where network bandwidth is expensive. We also encapsulate the modifications used to tune the

application in components called actuators. Actuators may be installed at some point in the program flow execution and provide synchronization and low overhead control of application variables and function executions. As sensors and actuators can communicate with each other, we can perform simple tuning within process executions without the need for communication.

As the dynamic tuning is performance model-centric, we need a performance model that can be used on heterogeneous processors and network such Grid Systems. We propose a heuristic performance model to find the maximum number of workers and best grain size of a Master-Worker execution in such systems. We assume that some classes of application may be built capable of changing grain size at runtime and that change action can modify an application's compute-communication ratio. When users request a set of resources for a parallel execution, they may receive a multi-cluster configuration. The heuristic model allows for shrinking the set of resources without decreasing the application execution time. The idea is to reach the maximum number of workers the master can use, giving high priority to the faster ones.

When we change the number of workers in a Grid environment, we should perform changes in application parameters and in the system configuration. That is an example of multi-layer tuning. To grow or shrink the number of processors, we need to interact with Grid middleware in synchronization with application process reconfiguration. To accomplish that, we have actuators that interact with the Grid services.

We presented the results of the dynamic tuning of grain size and the number of workers in Master-Worker applications on Grid systems, lowering the total application execution time while raising system efficiency. We used the implementation of Matrix-Multiplication, N-Body and synthetic workloads to try out different compute-communication ratio changes in different grain size selections.

Chapter 1

# Introduction

In this chapter, we setup the environment, its characteristics and the problems users find when facing scenarios of tuning parallel/distributed applications on computational Grids. We identify these problems and present the contributions this thesis states to accomplish concerning the user's requirements for application performance dynamic tuning in such systems. In sequence, we discuss our goals and the assumed restrictions of this work, as well as its organization and contributions.

## *1.1 Overview*

Many big problems are presented for resolution by computers in the current climate. As technology progresses, computers become more popular and their users require more processing resources to do their work. Research fields such as physics, chemistry, medicine and weather prediction are dealing with new computational challenges [1, 2]. These include high-level detailed simulations, the analysis of huge amounts of data and high processing power requirements. Computer capacity evolution grows every year as the cost per MFLOP gets cheaper [3].

An important challenge present for the HPC user community is the multiplicity of new computer systems configurations. To achieve high performance, users combine computers in many ways building new systems such as parallel machines, Beowulf Clusters, MPP's, NoW's and HNoW's [3]. The key idea is to break down the problem

into small pieces and distribute the pieces among different machines to work in parallel. By using that strategy, called parallel programming, the time required to solve the problem should be reduced. The computational time reduction achieved by parallel programming depends on a large number of properties such as system configuration, required communication between the nodes, load balance management overheads and algorithm intrinsic parallelism (ratio of serial and parallel sections) [4].

The Internet age made cooperation on many levels possible by the ease of passing on information through compute and storage resource sharing. Internet-based technologies like the web turns to be the standard of human to machine and machine to machine communication interfaces [5, 6]. With the popularity of the resources available online, new semantics of resource sharing appeared. Computational resources from different organizations started to take part in wide distributed systems currently known as Computational Grids [2, 7]. A Computational Grid is an infrastructure that allows resource sharing among different organizations. The scientific community has been expending much effort to create standard technologies for Grid construction and recommendations to get interoperability among system stakeholders [8, 9].

Currently, the most commonly adopted middleware used in Grid construction for managing and resource sharing is the Globus Toolkit [7, 10]. It contains a set of services that allow computational resources, like clusters or parallel machines, within different organizations to be operated in agreement with each organization’s usage policy. In communications, for example, the toolkit uses Internet technologies such as web services, TCP as communication abstraction and private key infrastructure and TSL to cover security requirements [2].

Many parallel programs design semantics were created to address problems of application execution on wide systems such as Computational Grids. System heterogeneity is a problem, since it can easily lead to poor performance execution due to load imbalance [11]. In Grid executions, the network heterogeneity makes it more difficult for the application developers to achieve an efficient resource use. Performance depends on many factors, such as the level of parallelism of the problem and the computation communication ratio. On Grid systems, application execution over more than one organization should deal with inter-organization communications that

generally represent high latency and low throughput [2, 7]. The developer does not have control over the compute nodes where the application executes its processes.

Application performance improvement is not an easy task. It relies on a deep understanding of applications and systems. Application domain knowledge is required, together with its parallelization and system architecture, to minimize communications by using mapping and clustering techniques. A typical application performance improvement methodology could be: execution, monitoring, analysis and code modifications [12]. The user uses the information grabbed from execution through monitoring and decides, based on analysis techniques, which changes should be made to achieve performance improvement. The analysis can be empirical, by means of attempts to reduce total execution time through improvements on heavily used functions in an execution path or by means of performance models which intend to describe the application behavior [13].

Performance is a major issue in parallel programming. When a programmer develops a parallel application she expects to reach some performance indexes. Therefore, in the last year, several efforts has sought to provide automatic performance analysis and tuning tools that guide and help programmers and users of parallel applications to reach those expected indexes. These were presented as automatic off-line performance analysis, automatic on-line performance analysis and automatic dynamic performance tuning. [14-17]

In the last ten years, grid systems have become a promising approach and have spread widely, so that many groups are developing parallel applications to run on these systems. However these systems are dynamic by nature and are composed by heterogeneous and shared resources. Therefore, the performance of parallel applications on these environments may vary dramatically depending on the particular conditions of each execution. So, automatic performance tuning tools are critically necessary to accomplish the performance expectations.

Automatic dynamic performance tuning has been used on parallel/distributed environments and now our goal is to extend the applicability of such approaches to grid environments. This application involves several aspects as there are several technical issues that must be considered:

- The main point is that when a parallel application is launched on a grid environment, the user does not have direct control of the environment and there are several decisions that are taken by different software layers, such as meta-schedulers. So, it is necessary to enable the tuning tool to have direct access to the application processes wherever they are launched [18].

- Grid systems involve different organizations with several administration domains. Therefore, the tuning tool must be able to fulfill the policies of the different organizations so that it can access the processes running on any participant organization [19].

- Since grid environments are geographically distributed environments, communication is a critical issue that can significantly affect application performance. Moreover, the tuning tool requires the collection information from different sites. This implies that information collection can be slow and can compete with and disturb application communication [7]. Therefore, it is necessary to develop collecting policies and strategies that minimize the communication requirements of the tool.

- In some cases it can be necessary to adjust the grid environment (change the number of resources assigned to a parallel application or change the type of resources). This implies that the tuning tool must be able to interact with the different middleware layers of the grid environment.

There are aspects that need to be considered related to performance tool itself and application development paradigm and architecture:

- It is necessary to analyze performance behavior to develop the performance models that can be used to steer the application. This implies determining the parameters of the application that affect its performance. So, it is necessary to know which measurements must be taken to evaluate the actual behavior of the application. That behavior should be used to develop some performance models that take these measurements and provide the optimal value of the application parameters. These

suggestions should carry out the necessary adaptation in the execution application processes to achieve better performance indexes.

- It is necessary to define the structure of the tuning tool. As has been mentioned, certain measurements must be taken from the application processes and this information must be used to evaluate a performance model. Therefore, the distribution of the collector processes, the analyzer processes and the required components involved in tuning processes must be distributed with a balance between local and global architecture.

Moreover, we discuss the goals achieved and solutions proposed to overcome these aspects in order to use automatic dynamic tuning techniques in parallel application executions on computational Grids.

## 1.2 Goals and Constraints

The thesis proposes a Grid Monitoring Analysis and Tuning Environment (GMATE) capable of the dynamic tuning of parallel/distributed applications within a Grid System. The main contributions of this work are the analysis of performance issues inside Computational Grids, the evaluation of application parameters based on performance models and the design, development and implementation of an architecture for distributed dynamic tuning enabled to Grids environments. The architecture is open so that knowledge about different performance bottlenecks can be integrated. An example to show the viability and applicability of the approach is that a performance model for tuning grain size and the number of workers in hierarchical Master-Worker applications has been developed.

The solutions adopted for the requirements described in the last section are the following:

- In cases where a Grid meta-scheduler decides application mapping, we proposed two process-tracking approaches that enable GMATE to locate where Grid middleware maps application processes. One approach consists of the integration of GMATE components as Grid middleware. The other consists of embedding GMATE components inside application binaries. The first requires site

administration privileges while the other increases the application binary which slows down application startup.

- To obey organizational policies, all communications use the same application security certificates for authentication. The same communication is performed using Grid middleware API. That approach enables the monitoring and tuning process to adapt dynamically to organizational firewall restrictions and network usage policies.

- To lower the communication needs of GMATE, we encapsulate part of the logic required to collect metrics and change application parameters in components that run inside process space. For metric collection we create sensor components that reduce the communication by event inside the process space. As they are different from traditional instrumentation, sensors can postpone metric communication and perform basic operations such as summarizations, timers, averages or threshold based metric generation. That reduces the communication requirements in cases where network bandwidth is expensive. We also encapsulate the modifications used to tune the application in components called actuators. Actuators may be installed at some point in the program flow execution and provide synchronization and low overhead controls over application variables and function executions. As sensors and actuators can communicate with each other, we are able to perform simple tuning within process execution without the need for communication with an external analysis process.

- As dynamic tuning is based on performance models, we need a performance model that could be used on heterogeneous processors and networks such Grid Systems. We propose a heuristic performance model to find the maximum number of workers and best grain size of a Master-Worker execution in such systems. We assume that the compute to communication ratio, or grain size, is decided at the point of algorithm parallelization and application development. In many cases that parameter is a factor used in application tuning. Instead of having a fixed grain size, we advocate that developers prepare the application/algorithm to work with different grain sizes. These applications may be capable of changing grain size at runtime which should be used as an action to modify application compute-communication

ratios. These actions can be used to adapt applications to different heterogeneous configurations. For example, when users request a set of resources for a parallel execution, they may receive a multi-cluster configuration. The heuristic model allows for shrinking the set of resources without decreasing application execution time. The idea is to reach the maximum number of workers, giving high priority to the faster ones.

- When we change the number of workers in Grid environment, we should perform changes in application parameters and in system configuration. That is an example of multi-layer tuning. To grow or shrink the number of processors, we need to interact with Grid middleware in synchronization with application processes reconfiguration. To accomplish that we have actuators that interact with Grid services.

- In the case of a hierarchical Master-Worker application, the tuning of the number of workers and grain size of a sub-Master may be done by a process placed in the same network domain. That decreases the tuning response time because it reduces message latency for metrics collection and tuning action communications. We present a comparison between three different approaches to analysis component distribution: centralized, hierarchical and fully distributed. In hierarchical analysis, local analysis reduces communication with global analysis components. In a fully distributed approach, each analysis component maintains its local data and a shared consensus of global analysis state required to tune the application.

We presented the results of dynamic tuning of grain size and the number of workers using our performance model in Master-Worker applications on Grid systems, lowering total application execution time while raising system efficiency. We used Matrix-Multiplication, N-Body and synthetic workloads to try out different compute-communication ratio changes in different grain size selections.

## *1.3 Thesis Organization*

In order to understand the problem of dynamic tuning in Grid environments first we have to describe Grid system characteristics. Chapter two presents assumptions about Grid environments, their requirements and also discusses some applications issues. It

analyzes some active tools used in system and application monitoring that can be used for performance analysis. In chapter three, we present our contribution with a performance model for Master-Worker dynamic tuning in computational Grids.

The proposed architecture for distributed Grid monitoring, analysis and tuning environment (GMATE) is described in detail in chapter four. This chapter presents, for each issue of automatic dynamic tuning, proposed ideas, achieved solutions, constraints and drawbacks. The model proposed in chapter three and the architecture proposed in chapter four are the main contributions of this thesis.

The experimental validation of our contribution is evaluated in simulations and a real enterprise Grid Testbed in chapter five. In chapter six, conclusions and open lines of investigation that this work acknowledges are commented on.

# Chapter 2
# Computational Grids

## 2.1 Overview

The main idea behind grid environments is "*coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations*" [2]. The resources may be distributed over the world interconnected by WAN infrastructures such as Internet. Grid environments have interconnected resources distributed under different administrative domains. These resources could be either physical or logical. Physical resource examples range from simple processing nodes to a wide parallel machine. Logical resources can be application services or middleware services. The main problem addressed by Grid computing is to facilitate cooperation between sets of users from different organizations. System architecture must be capable of the dynamic coordination of resource sharing among different institutions [1, 2, 7]. Currently, this requirement is covered by a set of software layers which abstract system heterogeneity and interoperability.

## 2.2 Parallel Programming

Computational Grids expose different software layers. These layers allow trusted resource interoperability among different administration domains. We may compare these software layers to a general operating system concept. In a Grid we have a meta-scheduler to determine application resource assignation, compute elements as compute resources and storage elements as grid file systems. From an application point of view,

these resources may be used in different roles inside an application parallelization used strategy. Moreover, that strategy relies on specific load breakdown and semantic. These semantics allows us to classify applications that run in Computational Grids into the following groups:

- Parametric Applications;
- Scientific Workflows;
- Distributed Applications.

A parametric applications class is the easiest form of parallelization semantics. Generally, the application consists of the execution of a program using different parameters or using different input data. An application finishes its execution when all component tasks finish their execution. In this class, the maximum speedup is determined by the number of executions and the minimum execution time is determined by the largest single component execution.

Scientific Workflows consist of different program execution denominated jobs where some jobs have dependences of data generated from other jobs forming a direct asynchronous graph (DAG). The communication between jobs is achieved by means of data files. The speedup of this class of application is determined by the level of parallelism found on the execution graph. The minimum execution time is derived by the sum of execution of jobs in DAG critical path.

Distributed applications consist of the collaboration of different processes during execution. That collaboration is achieved by means of inter-process communication. Current examples of communication mechanisms are GridRPC, Web Services and Message Passing.

### 2.2.1 **Programming Models**

The process to build a distributed application determines its level of parallelism and consequently its speedup. Generally, application parallelization is performed by dividing the application load to be executed among processors. Foster in [20] presents a methodology of application parallelization called 'Task/Channel' in which the process consists of:

- *Division*: the application load is divided into small pieces of code execution or tasks that communicate with each other forming a graph of execution.
- *Clustering*: due to the fact that communication may slow down dependent tasks, tasks with high data decency for other tasks (which would generate higher communication among these tasks) may be grouped.
- *Mapping*: the final step is to assign tasks to physical processors for execution.

There are many aspects that influence distributed applications executed in a Computational Grid. Lessons learned from parallel computing suggest that heterogeneity in compute and communication resources makes harder the task of performance improvement [21]. The challenge of higher application speedup depends on the efficient use of the maximum available compute resources achieving a load balance. There are many performance models that explain the behavior of parallel applications in homogeneous systems. These models generally analyze the influence of communications paradigms in application parallelizations. Examples of communication paradigms are: Master/Worker, Divide and Conquer, Pipeline and SPMD.

### Master-Worker

In the Master/Worker paradigm (MW), the application load is divided into small parts, called tasks, and the processor with the master role has the responsibility of managing the tasks distribution to be processed by processors with worker roles. The load may be divided into iterations. The master processor divides the load among workers for every iteration. Iterations may be divided in three stages: startup, steady and finalization. The startup stage is the period delimited by the iteration start and when all workers start processing or a worker finishes processing: this is what comes first. In [22] the startup phase is also called installment. The steady phase consists of the period between the end of startup and the finalization phase. The finalization phase occurs when the master has no more work units to deliver and a worker finishes its processing queue. The author in [23] provides an exhaustive analysis of the different possible cases of startup and finalization scenarios.

### Divide and Conquer

The Divide and Conquer, Pipeline and SPMD are paradigms with a high dependency on synchronization. That characteristic makes difficult its application in hetereogeneous

processors and even more in heterogeneous networks in which we easily found load imbalance. To deal with this system heterogeneity easily found in computational Grids, we need to divide dynamically and schedule work grains execution to achieve load balance. In that direction, the dynamic Master/Worker paradigm results as a basic model suited for heterogeneity and can be seen as the building block for more complex models.

### Pipeline

The pipeline programming model consists of an application functional decomposition in different stages. The load processing should be divided among processors that are responsible for each stage. The parallelism of pipeline programming is achieved in functional decomposition where different parts of the pipeline represent different functions applied over a stream of data.

### 2.2.2 Performance Characterization

An index used to measure application scalability is the *speedup*. Given a number of processors N, the maximum *speedup* an application can achieve is a function from its parallelism level. The parallel application is not complete parallel. The processors assigned to compute the load need to receive the data, at least, and send or write the results. The literature has some limit expressions of *speedup* representations. The most famous is the Amdal's Law, which rates the parts of the program that can be executed in parallel to the serial program part. We may say that the processes of data distribution and result collection can be characterized as serial parts. Equation (1) presents Amdal's Law where *S(N)* represents the maximum *speedup* of applications which have the P fraction of the load that can be performed in parallel.

$$S(N) = \frac{1}{(1 - P) - \frac{P}{N}} \quad (1)$$

Performance bottlenecks may be found in two basic resources: communications or computations. So, we may divide applications as communication- or computation-bound. In those which are communication-bound, master to workers communication

channels (network links) are busy most of the time, while workers processors wait for computation. In those which are computation-bound, network links have idle periods while processors are busy most of the time.

In both startup and finalization phases, we have idle workers. In [24] we model the prediction of these phases in respect to grain size and homogeneous networks. In section 4.1.1, we analyze the impact of grain size in these phases and provide the knowledge necessary for expansion to scenarios with network and processor heterogeneity.

## *2.3 Grid Middleware*

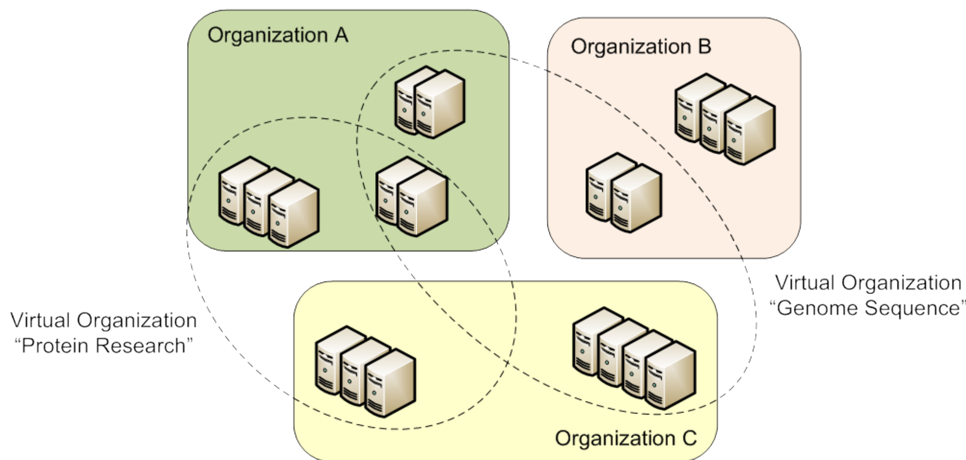



Figure 1 – Organization and virtual organizations in a Grid.

### 2.3.1 **Overview**

Part of the services needed to create a Grid are provided by middleware software. The multi-institution characteristic of the Grid requires that the cooperation should be driven by strong security constraints. The resource usage should have specific use policies and have to follow organizations policies. So, the available infrastructure can be used by different groups distributed over many organizations. The set of users in different organizations who have the same goals and objectives is called a Virtual Organization (VO). Under the VO concept, users can be identified and group policies can be applied [1, 2, 7]. Figure 1 presents an administrative view of a possible Grid scenario. Under the

presented scenario, three organizations cooperate on two distinct projects. The work group cooperation in each project constitutes different virtual organizations "Genome Sequence" and "Protein Research". Those VOs may or not share physical resources from component organizations.

Organization resource sharing is conditional: each resource owner shares the resources based on their own constraints. In other words, the resource sharing mechanisms are built on top of local policies and organization policies. The services needed to provide such services can be analyzed into layers. The common community-adopted philosophy model used to analyze the required services needed to build a Grid is the Grid protocol architecture model proposed by Foster in [2]. Based on this model, we can differentiate the services used the Grid system architecture.

### 2.3.2 **Globus Toolkit**

The architecture model adopted by most existing Grid implementations today is presented in Figure 2. That architecture allows categorization of service requirements and identification of existing technologies that can be used to fulfill requirements such resource sharing and location, security policies, identification and authorization.

The software layers to provide resource sharing and use can be divided into: application, collective, resource, connectivity and fabric as presented in Figure 2.The Fabric layer represents the available infrastructure. In order to have security in fabric services use, we need a software layer to manage inter-organization security operations. This is done by the Connectivity Layer. By using the services provided by the Connectivity Layer, we need services for resource abstraction and management, and this is done by the Resource Layer. The Collective layers manage resource groups providing Grid-wide services (that need to interact with more than one resource such as brokering and a meta-scheduler).

Each layer has specific requirements to cover and there are many working software solutions available. The requirements and implementations using the Globus Toolkit services can be analyzed through these layers [2].

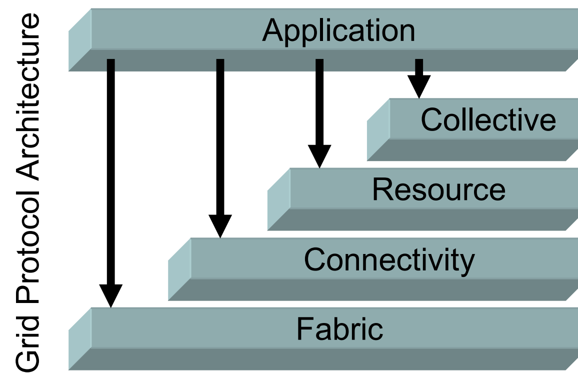



Figure 2 – Grid Protocol Architecture, from [2].

**Fabric Layer**: provides the interfaces for local control. The idea is to expose the infrastructure as resources. Examples of resources are storage systems, catalogs, network resource, computational resources and sensors. A richer Fabric interface enables more sophisticated sharing operations and makes possible the creation of high-level aggregation services such as co-scheduling and Grid-wide transaction services.

Most monitoring metrics are obtained in the Fabric Layer. Grid monitoring tools should work with the provided services to gather performance data. In some cases, the monitoring services should be exposed as Fabric Layer services to be used to construct Grid-wide monitoring services.

**Connectivity Layer**: provides the core communications and authentication protocols required for fabric services utilization. The authentication should provide services for single sign on (user identification and authorization), delegation (ability to delegate user's rights to his/her submitted program), and integration with the existing local security solutions, and also have user-based trust relationship (VPN, signed certificates, security tokens based on PKI).

The authentication, communication protection, and authorization mechanisms in current use employ the public-key Grid Security Infrastructure (GSI) protocols. GSI extends the Transport Layer Protocol (TLS) to address the presented services requirements and uses X.509-format identity certificates to handle identification and authorization based on user certificates [2, 7].

In order to fulfill these Grid security requirements, a Grid-wide monitoring tool should use the connectivity infrastructure for security and transport. In monitoring and tuning there are transmissions of events related to application behavior. These events could, for example, be used to reverse engineering applications, so they should be protected by security mechanisms such the use of cryptographics in communications.

**Resource Layer**: uses the Connectivity Layer services for the secure negotiation, initiation monitoring, control, accounting and payment of sharing service operations on individual resources [1]. The main classes of Resource Layer protocols are:

- Information Protocols: used for resource registration and localization, and resource information shared over the wider system. Examples are resource state (current status), configuration properties and running metrics. The information services are provided by the Grid Information Service, also known as Monitoring and Discovery Service (MDS), detailed in chapter 4, and are composed of:

    - Information Providers (IP);
    - Grid Index Information Services (GIIS) or Index Service;
    - Aggregator Services;
    - Trigger Services.

  The composition of the Grid is dynamic. Organizations can insert or remove resources without any control. To support that, the system should have protocols to discover and record the information about resource availability and its capabilities. Two protocols are provided to services access: The Grid Resource Information Protocol (GRIP) which is used to define a standard resource information protocol and the associated information model, and the Grid Resource Registration Protocol (GRRP), used to register resources in GIIS [9].

- Management Protocols: used to negotiate resource allocation based, for example, on a Service Level Agreement (SLA) or requirements properties. It also provides the services for resource state monitoring whilst in use and control operations such as termination (consensus about the end of resource utilization by a client). The Grid Resource Access and Management (GRAM) protocol is used for allocation of computational resources and the monitoring of resource usage [9].

From the point of view of a Grid monitoring tool, the resource layer provides essential services for resource localization and information. It also provides services for composing Grid-wide monitoring, using locally-based monitoring services registered as resources within the Resource Layer.

**Collective Layer**: handles collective resource operations and provides Grid-wide services. Those services include directory services, co-allocation, scheduling and brokering services, monitoring and diagnostics services, data replication services, Grid-enabled programming systems, workload management systems and collaboration frameworks, software discovery services, community authorization servers, community accounting and payment services, and collaboration services.

Many services are custom Grid solutions and others are adaptations of products to fulfill Collective Layer requirements. We can cite Condor-G, an adaptation of the Condor scheduling system to schedule jobs using individual resources services. Another example is the GridWay co-scheduler. The Grid-wide monitoring services are exposed in the Collective Layer which allows users to interact with a subset of active resources. For example, a request for application execution may require a number of Compute Elements to fulfill its requirements. Each Compute element may be a complex resource composed by a cluster of Compute Hosts.

A common way of using the computational power of a Grid is to spawn the processes of a massive parallel application within the available processors inside a resource. Following the Grid Protocol layers presented in Figure 2, a user can interact with a Grid Web Portal, a collaboration framework of Collective layer, and submit his batch application. That application should enter a meta-scheduler queue such as Condor-G [18, 25] or a Community Scheduler Framework (CSF), to services of the Collective Layer. The meta-scheduler negotiates to Resource Layer services in order to do resource reservation using authenticated and secure communication services provided by a Connectivity Layer. Following the Grid Protocol Stack, that request is translated into the Fabric Layer to a local cluster scheduler, such as Condor, PBS, LSF or SGE, where the application job is executed [18, 25, 26]. Depending on the services used, users do not know where their applications jobs runs. To monitor the internal structures of job processes, a monitoring tool should be able to track process submission.

Not all the machines available for job execution in the Grid are exposed as single processor resources. The resource capable of job execution is called a Compute Element (CE). A CE may be composed of a single machine or a more complex parallel architecture such as clusters, vector processing machines or even mainframes. The machine component of a CE that does not have the Grid services installed is called a Compute Node (CN).

## Large Applications

Grid systems are built to allow resource sharing between users from the same VO. The system composed by the available resources allows users to exploit more computation power. The literature presents some example cases of blood system simulations [27, 28] and weather forecasting [7]. Applications developed to have the maximum benefits of Grid systems should be concerned about the system network topology. That requires the application to be modified to have a benefits mutable network topology. Those modifications are more complex that a simply application parallelization. The Grid is a distributed system architecture generally composed by different levels of interconnected networks between its resources. Some resources are shared among applications and users. Data communication between different resources may have different throughput and latencies and typical problems of distributed systems such as load balance and synchronization bottlenecks are hard to locate and harder to solve [29]. A Grid monitoring tool can provide the required information in order to help developers solve those problems.

### *Parallel Applications*

In order to reduce application execution time, and to use the available resources, the application can divide up the work to different machines and perform it in parallel. To process the divided work, the application processes generally need to communicate in order to complete its execution. The communication semantics between processes define two main programming paradigms: Distributed Shared Memory (DSM) [30] and Message Passing (MP) [4].

### *MPICH-G2: Message Passing for the Grid*

Different to DSM, where the developer should not be concerned how and when the communication is done, Messaging Passing (MP) consists of explicit communication

primitives used in distributed process communications. If a process needs to send some data to other processes, this calls a send function and the other process in some execution point should call a receive function. There are some variants such as non-blocking, asynchronous and synchronous, and collective operations. With synchronous operations, the send operation only continues when the corresponding process calls the receive function; and in asynchronous ones, the operations do not block and there are some functions to test the operation completion. In collective operations, the MP library handles the data distribution between participants of the communication.

MP has some advantages over DSM due the fact that the developer has a great deal of control over the communications. However, the more complex the parallelization of the application, the more problems should occur, such as lost of efficiency due to load imbalance. Some process may get blocked by a receive call waiting for a send from other process which is doing something else. That problem, for example, is called *late sender* problem [31].

The Message Passing Interface (MPI) is a standard API for message passing. There are many implementations of the Grid enabled MPI as MPICH-G2, PACX-MPI, MagPIe and Stampi. [32] presents a comparative performance study of the performance of those MPI implementations on multicluster[1] environments.

With the popularity of MP-based applications, more users have concerns about performance problems in their applications. That pushes tools developers to create monitoring tools in order to help users to measure what application processes are waiting while blocked by MP calls.

### *GridMPI: Message Passing with Multi-Cluster Support*

The GridMPI is an implementation of the MPI standard 2.0 which allows multi-cluster executions. Users may use different MPI implementations for intra-cluster communications and the inter-communications are performed using sockets.

---

[1] We assume the term multicluster for a system composed for more than one cluster, interconnected by a LAN, and assume that the term Grid represents more requirements such different administrative domains and interoperability.

The advantage of GridMPI over MPICH-G2 is the capability of multi-cluster executions where the clusters have private IP addresses. MPICH-G2 requires that all machines that participate in communications to have a valid IP address. That is not a common setup issue for NoW's or Beowulf's, which generally use private networks inside the parallel machine. A multi-cluster execution follows the Interoperability MPI (IMPI) standard which defines a protocol for message and information exchange among processes executing in different clusters. This execution can use different MPI implementations and transport levels at intra-cluster message exchanges.

The GridMPI implementation provides two components to allow multi-cluster executions: an IMPI Server and an IMPI Relay. The IMPI Server is responsible for recording and informing the participants of a parallel execution of the information required to establish point-to-point communications. If all machines have public IP addresses, the IMP Server provides the information about the global process table and listening ports.

In case of clusters with private IP addresses, the GridMPI provides a component called IMPI Relay. That has the function to serve as a proxy from local machines to the 'outside' world. At startup, the IMPI Relay constructs two tables, a private table and a public table. The private table has information on how local machines are identified and the public table has all the participants that have public addresses.

### 2.3.3 **Condor**

The Condor is an environment for managing the execution of user jobs. Its philosophy consists of matching jobs to be executed to machines with capability to run job. It contributes with the Grid concept using a bottom-up strategy where Condor systems can delegate jobs to outer Condor systems, Globus based middleware and cluster batch schedulers such as PBS, SGE or LSF. The idea is that users use the same procedure to execute local jobs and to execute jobs into remote locations.

Different from conventional cluster schedulers' scheduling polices, Condor systems use the concept of matchmaking based on Classified Advertisement (ClassAds). When machines are available, they announce themselves with their properties, called 'ClassAds' in Condor jargon. When users submit a job for execution, that generates a ClassAd containing the job requirements and properties. At periodic intervals, a process

within the role of Matchmaker tries to find a best match among ClassAds machines and ClassAds jobs, evaluating a rank function specified in ClassAds jobs against properties specified in ClassAds machines.

The Condor architecture is composed of a set of processes with distinct responsibilities, as presented in Figure 3. The machines to which users submit their job execute two daemons: the '*schedd*' and '*shadow*'. The '*schedd*' is an agent responsible for queuing jobs waiting for execution. These queued jobs have their ClassAds published in the central manager process.

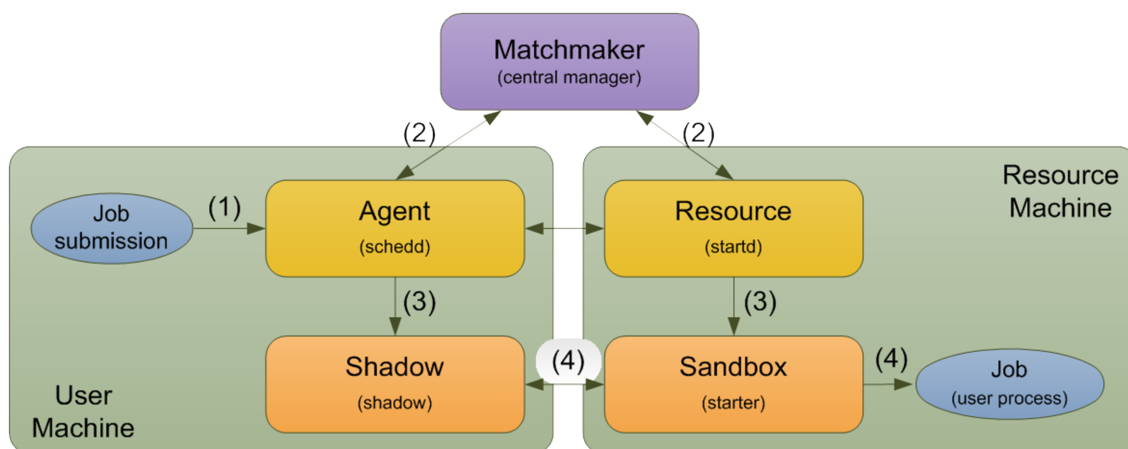



Figure 3 – General Condor Kernel Architecture with the sequence of information flow.

A successful execution use case of a user job in a Condor system consists of the following steps:

1. The user submits the job using a *ClassAd* specification.
2. The user agent and the resource publish its *ClassAd* on Matchmaker.
3. In periodic bases, the Matchmaker assigns a resource to a job. At this point, the agent creates a shadow process in the user's machine.
4. The job is created on resource machine and uses a shadow on a client machine to exchange information.

Condor software is used as a Grid resource broker in many Grid production environments as middleware package distributions such as Virtual Data Toolkit (VDT) and GLite.

## 2.4 Heterogeneous Scenarios

When users submit a parallel application to a Grid System, they may not have control of what computation resources will be assigned for application execution. Application execution requirements can be satisfied by more than one resource. For example, the user may explicitly require some physical resource in their application requirements. Indeed, to get more opportunity to execute, the application requirements can be created in order to be satisfied by more than one resource. In that case, the system evaluates its current state and decides what would be the best choice in resource assignation. The used control is a trade off between time to execution (waiting for fixed resources) and resource availability due Grid dynamic behavior. Not all resources are available all the time. In such scenarios, a user's application machine assignation is controlled by a Collective Layer (Meta-Scheduler and Resource Brokers) based on the requirements provided by users.

Parallel application execution submission to a Grid may be classified in the following scenarios of execution:

- *Homogeneous Processors and Homogeneous Network (HoPHoN)*. This scenario consists of a common cluster of dedicated processors. Examples are Ethernet based commodity clusters exposed to the Grid by a head node running on two networks. That scenario may occur when Grid meta-scheduler assigns all application processes to a single homogeneous Compute Element composed of many Compute Hosts.

- *Heterogeneous Processors and Homogeneous Network (HePHoN)*. Cluster upgrades generally lead to heterogeneous processors. In such scenarios, network characteristics remain the same in bandwidth and latency for the master/worker paradigm and the compute time for the same piece of work may differ with different groups of processors. We assume that each homogeneous part of the cluster as a processor group for analysis purposes. As with the *HoPHoN* scenario, applications may receive a heterogeneous cluster by a meta-scheduler assignation.

- *Homogeneous Processors and Heterogeneous Network (HoPHeP).* Different clusters bought at similar times, from the same supplier tend to be homogeneous. The scenario described in [33] reports that 75% of machines are homogeneous. This scenario has two variants: the master may use only one network interface for local and remote communications or may have different network interfaces for local and remote communication.

- *Heterogeneous Processors and Heterogeneous Network (HePHeN).* In such scenarios, the Grid resource broker may assign different groups of machines to different organizations. The master should deal with processor heterogeneity in local clusters and remote assigned workers. This scenario has also two variants: the master may have only one or two network interfaces as in the scenario *HoPHep*.

## 2.5 Monitoring Approaches

A Grid system can be monitored on many levels from Collective to Fabric ayers. These levels allow us to classify monitoring tools within System or Application monitoring tools. In System monitoring, the metrics are related to system states as available bandwidth, machines loads or available resources. In Application monitoring, the performance data is related to application execution as time spent on some modules, time spent on communications or cache misses related to some code region execution. In many tools implementations, the monitoring process collect measurements using sensors [34].

Sensors are software components that collect data from execution properties. For example, a sensor that collects network utilization information may be a process that interacts with some device using SNMP and feeds an information service or other consumer. The first proposed standard for monitoring architecture in Grids is the Grid Monitoring Architecture [8, 34].

### 2.5.1 Grid Monitoring Architecture – GMA

The Global Grid Forum (GGF) proposes a scalable architecture for Grid monitoring called Grid Monitoring Architecture (GMA) [8, 34]. It describes requirements for systems that collect and distribute performance information in Grid systems such as low latency, high frequency and minimal measurement overheads, security and scalability.

In this architecture, trace event data is called an event with properties such as name, timestamp and a structure that may contain other property items. The semantic concepts presented in GMA, detailed in [8], are:

- Entity: any useful network enabled resource, unique and with a defined lifetime.
- Event: a collation of values containing timestamp and type data, associated to an entity and defined by a specific structure.
- Event schema: defines the semantics of all events, consists of the event type definition catalog.
- Sensor: a process that monitors an entity and generates events. Sensors are divided into:
    - Passive: read values available about an entity, such as counters and properties.
    - Active; generate data based on benchmarks such as network probes for bandwidth and latency.

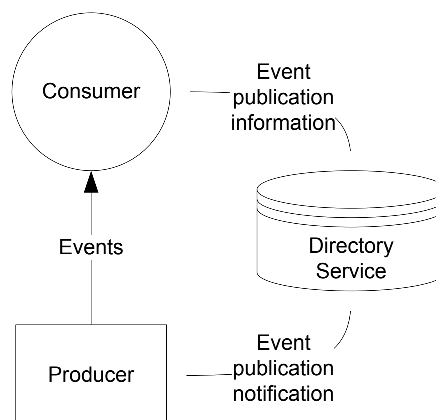



Figure 4 - Grid Monitoring Architecture Components, from [8].

The architecture is characterized by three main components: producer, consumer and directory service. The producer is the component that generates the event data and the consumer is the component that requests or accepts it, as presented in the figure below. The third component is the Directory Service which is used by the producer in order to

publish what event data is available and by the consumer in order to locate and contact the producer.

Communication between the Producer and the Consumer may interact with each other in three ways [8]:

- publish/subscribe;
- query/response;
- notification.

In publish/subscribe interaction mode, the components of the Consumer and Producer, in an initial stage, use the Directory Service to locate each other and, after that, the communication is done without accessing the Directory Service. After the initial stage, the components agree which events should be transmitted and this characterizes the event subscription by the Consumer which is published by the Producer. In query/response iteration mode, the Consumer locates the producer similarly to the initial stage of publish/subscribe and sends a request with one or more events data query. The Producer later responds with the requested data. In notification iteration mode, the Consumer configures the registry to event notification information. When event data is generated by the Producer, it is sent to all registered Consumers by notification. The event data is produced by sensors controlled by the Producers [8, 34].

The major characteristic of GMA architecture is the direct communication between Producer and Consumer. That allows configurations where Consumer/Producer components act as a proxy Consumer/Producer. Proxy-based configurations could allow event filtering and transformation, the ability of data rewind and also cache behavior. There are many systems such SCALEA-G [35] and R-GMA [36] that are built on top of GMA architectures, although, the GMA architecture document does not specify the interfaces used for communication between the components, which could be used for monitoring system interoperability [34]. The GMA provides the base information which can be used to classify the current implementation of monitoring tools. A good analysis of current available tools in contrast to GMA architecture is provided by [34].

### 2.5.2 System Monitoring

A Grid system can be analyzed from many points of view. In a top down analysis, the groups of resources sharing a VO may have some general properties such as available compute elements, available storage information or even accounting information. Other properties represent information from a specific organization inside the VO. Some organizations may limit bandwidth utilization on a specific project, for example.

On other level, within the organization we may have clusters such as CEs. Each cluster has specific properties concerning what should be interesting for users. The same semantics can be used in the case of a CH inside the cluster CE. The machines have properties which can be monitored and serve users' needs.

Most of the Grid monitoring tools deal with monitoring of the environment where the application runs [37-42]. These tools rely on fabric level services in core Grid concept and protocols to fulfill a user's requirements. The construction of monitoring tools for Grid applications should take into consideration the proposed architecture design points used in the construction of Grid system monitoring tools.

The *gmond* component collects from 28 to 37 different metrics depending on the operating system. The data is sent in the multicast and unicast over TCP or UDP and it is packaged in a XDR representation. The information can be collected from the *gmond* daemon by listening to the multicast channel, configured by *unicast* or by pull mode, by making a direct connection through the *gmond* daemon. There is no mechanism for event selection or filtering [34, 37].

## R-GMA

The Relational Grid Monitoring Architecture (R-GMA) [34, 36, 39] is a distributed monitoring system compliant with GMA, based on a relational database system. It specifies a data model, a query language, and the functionality of a directory service. The data is distributed over the system. Users access the data using a global schema, without knowing where the data is. The query language is a fragment of SQL.

For monitoring information created by the producer components, the R-GMA has two types of data: static and stream, although the current implementation provides stream producers. The static information is provided as stream data. The internal data

representation is known as a GLUE schema [43]. The idea is to use the background of database management systems (DBMS) to help clients get information data. For many years, DBMS components such as query optimizers and data distribution have been active research areas. The data request and transport is done by Java Servlets as presented in Figure 5.

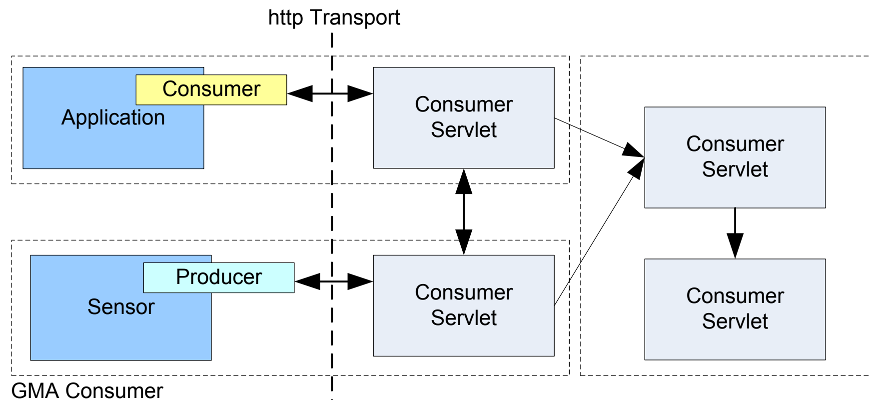



Figure 5 – Components and interaction in R-GMA, from [44].

One drawback in this implementation is that the communication between producer and consumer is done using server side components. That strategy has the overheads of two steps of data delivery, the producer and the consumer servlets [44].

## MonALISA

The Monitoring Agents in A Large Integrated Services Architecture (MonALISA) [38, 45] is a monitoring system built on a Dynamic Distributed Services Architecture (DDSA) framework. The agents can collect data from any SNMP agents such CPU and memory utilization from execution nodes, network link states and utilization statistics from routers, switches and other devices. It allows integration with Ganglia and other tools. That framework uses JINI for components communications.

The monitoring agents in MonALISA register themselves using a group JINI Lookup and Discovery Service (LUS). The registration is based on a lease mechanism which ensures the notification of clients in case the service fails. The clients connect directly to

selected agents in order to receive monitoring information. The agent management is done using RMI over SSL.

The MonALISA provides some features for lower-event communication. Users can get real-time or historical data based on a regular expression mechanism called a predicate mechanism. Communication between clients and agents is done using web services. Users can choose to use Agent Filters to get their information. Agent Filters are Java dynamic modules that can be deployed to any MonaLISA service in order to preprocess event data locally. Users may choose to receive information only when some trigger alarm condition occurs by using Alarm Agents. Similar to Agent Filters, it can be loaded and configured for information delivery based on logical expressions. An example scenario is when a client wants to receive the network utilization information only when the value is greater than 80% [38, 45].

### NWS

The Network Weather Service (NWS) [46] is a forecast system that provides prediction values for historical series. In a distributed system, the NWS can periodically monitor parameters from a network to available computational resources. The prediction values are derived periodically. There is a prototype implementation for Globus Grid Information System (GIS) architecture.

The idea is to provide forecast values after successive measurements as a time series. The process is done using different forecasting techniques. The supported forecasting methods supported and implemented as predictors are:

- mean-based methods, which provide estimate values of the sample mean;
- median-based methods, which use a median estimator;
- autoregressive methods.

The NWS keeps track of the predicted values for all the predictors and chooses the best forecasting method for the resource properties based on the cumulative error measure. The NWS package comes with CPU and network utilization sensors. The data extraction by users or applications can be done by web CGI or by a reporting API [46].

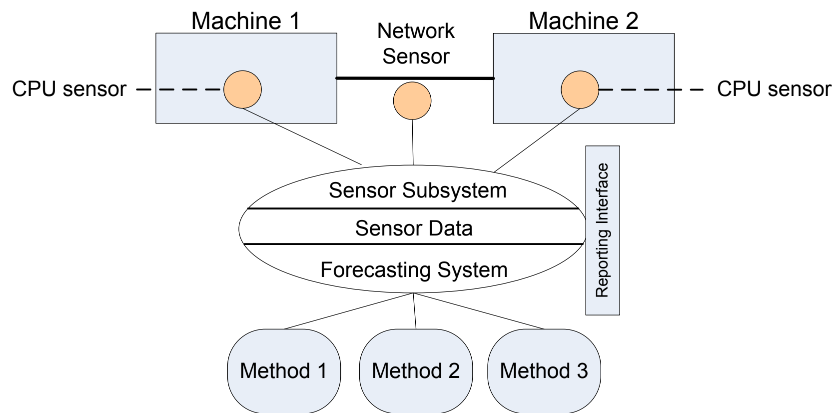



Figure 6 – NWS architecture overview, from [46].

## MDS

The component of Globus Toolkit which provides information services is the Monitoring and Discovery Service (MDS). This service centralizes all resources information. The MDS version 2 (MDS2) has three main components: the Grid Index Information Service (GIIS), the Grid Resource Information Service (GRIS) and the Information Providers (IPs). These components provide soft-state registration and enquiry protocols. The registration protocol allows MDS2 clients to publish information in the MDS2 database using the GIIS and the enquiry protocol allows MDS2 clients to request information from an information provider using the GRIS. The GIIS in MDS2 is implemented using LDAP services [7, 9, 10].

In MDS version 4, available in Globus Toolkit version 4, the access to the MDS services is based on WSRF presented on section 2.3.2 and it is called WS-MDS. The data information is provided by the information sources, which are communication interfaces implemented by Grid resources. The data extraction from information sources can be done by pooling or by the subscription/notification mode. The main services in WS-MDS are Aggregator, Index and Trigger Services [7, 10].

An aggregator Service is a kind of service that collects data from information sources and carries out a process with it. This is based on a framework called Aggregator Framework which is the base for the services under WS-MDS such as Trigger and Index Services. The Index Service provides the interface for explicit Grid resource

registration. The Trigger service allows some action to be performed in response to data changing within the WS-MDS such executing commands.

### 2.5.3 Application Monitoring

## Static Instrumentation

The application static instrumentation has always been done by developers [12]. The most primitive use of static instrumentation is simple screen print commands placed before and after some code region in order to verify the time spent in that region. The data generally is achieved during execution and analyzed after application execution. In order to help developers in such processes, many tools analyze the source code and insert function calls delimiting the interested code regions. Generally, this is done as a step of the code compilation process. The inserted function calls can generate trace events or can record profile information, depending on user needs. [14, 31]

***TAU***

The Tuning and Analyzes Utilities (TAU) [47] is a set of tools developed to help developers improve application performance. The tool provides a wide range of instrumentation types, performance data gathering, traces file format conversion programs and also includes two visualization programs. The supported instrumentation types are:

- Source code: handles an extensive list of languages, preprocessing the source code in order to insert instrumentation.
- Object code: provided by a modified compiler, which inserts the instrumentation on generated binary code after an optimization phase.
- Library wrapper: provides wrapper library for MPI, allowing measurements at library use level.
- Binary code: uses DyninstAPI to insert calls to TAU components on running binary code.
- Software Component: allows trace of component interface use by generating a proxy component with instrumentation included.

- Virtual machineL uses the Java Virtual Machine Profiler Interface (JVMPI) to register TAU components as a profiler agent in order to receive instrumentation information from function calls.

TAU tools can collect performance data in many configured forms:

- They can profile for region execution, recording the time spent on the delimited code region.
- They can record single events, to provide the number of events that have occurred or, they can record full traces with the begin/end events for selected regions.

The collected data is archived on a file; however, it has an interface which can be used to program an agent inside TAU to send the event information over a network. Even though it uses DyninstAPI [48] for dynamic instrumentation, the behavior is of a post-mortem instrumentation tool, the there is no action or analysis during application execution [42, 47].

***KOJAK***

The KOJAK project (Kit for Objective Judgment and Knowledge based Detection of Performance Bottlenecks) is a set of tools for automatic performance analysis for parallel programs to be used in application development. The performance data is collected by static instrumentation on compile phase and stored in proprietary trace format called EPILOG (Event Processing, Investigating and LOGing) [49].

The automatic performance analysis is done by recognition of inefficiencies patterns in collected measurements expressed in EARL (Event Analysis and Recognition Language). The analysis presents the performance data as a three-dimensional view correlating kinds of behavior, problems within binary/source code and runtime locations within processes and threads using the EXPERT (Extensible Performance Tool) analyzer [49].

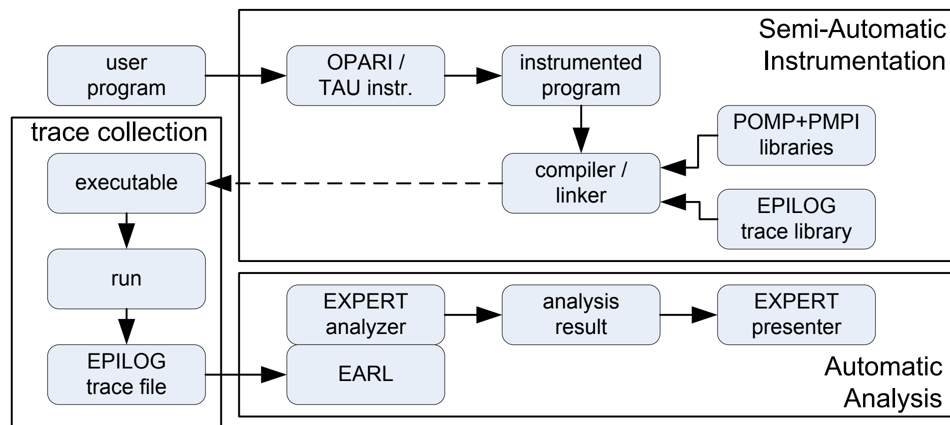



Figure 7 – KOJAK tool architecture as presented in [49].

## Dynamic Instrumentation

The post mortem method of application performance improvement is intended to change the code to have instrumentation, execute the application, and use the generated execution information to make changes in the code. However, that cycle consumes a significant amount of time because the application compilation time may be long. In this case, the approach of a change in the application binary during its execution can speed up the instrumentation process, cutting out the code change and compilation phases. There are some binary instrumentation tools such GNU bfd [50], EEL [51], DPCL [52] and DyninstAPI [48]. From those tools, the DyninstAPI library provides an extensive API which allows process attachment, binary parsing, management and modifications services.

When DyninstAPI attaches to a process, it parses the binary information and builds the necessary structures to allow process modification. The management services are: process stop, continue and terminate. The modifications services include variable creation, code sequence insertion, function replacement and dynamic linked library load. These services allow the dynamic instrumentation on a running process without a need for its source code [48].

### *SCALEA-G*

The SCALEA-G [31, 35] is a platform for performance monitoring and the analysis of Grid environments. It provides components such visualization programss, performance analyzers and instrumentation services. The instrumentation needed by monitoring is

done by the Dynamic Instrumentation Service which uses DyninstAPI for instrument Grid Applications. It follows the GMA model and uses OGSA in communications. The dynamic instrumentation service is composed of main components situated on different locations [31]:

- Instrumentation Service: controls the instrumentation processes.
- Instrumentation Mediator: controls the client side service communication and it runs on user identity proxy PKI certificates. It provides the transparency abstraction in terms of PKI certificates used in communication with the Instrumentation Forwarding Service.
- Mutator Service: a process that is executed on the same machine node where the application process executes. It has the responsibility of performing the dynamic instrumentation. It is based on the concept of application sensors inserted into an application processes.
- Instrumentation Forwarding Service: controls the instrumentation exposing a Grid web service for that purpose. It runs on service identity and provides the communication between the Instrumentation Mediator and the Mutator Service.

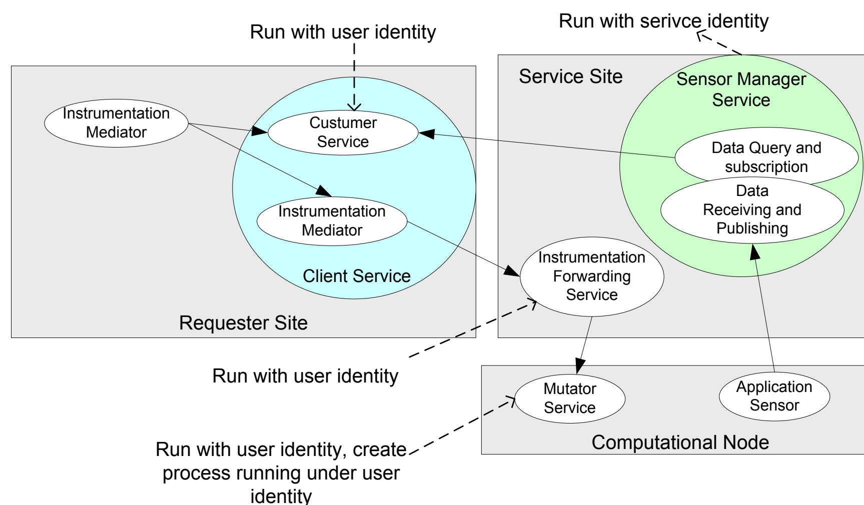



Figure 8 – Component architecture of the Grid dynamic instrumentation service, from [53].

All the communication within the components of dynamic instrumentation service is based on XML. This is an advantage for the point of view of tool integration, but there

are data and processing costs. The XML representation of data consumes more space than binary representation. In [31], some ideas such as data compression are presented in order to lower network bandwidth. The application sensors that provide profiling-based measurements and the profile data communication may be done in two ways: pull mode or push mode. In pull mode, the profile data is stored in a shared memory and the client gets it by request to the Instrumentation Mediator. In push mode, the update is done by buffer overflow trigger events.

### *Paradyn*

Paradyn [54] is a performance analysis tool which allows for dynamic interactive analysis of performance data generated by application processes. DyninstAPI was built as a component of Paradyn. The goal of Paradyn is to lower the instrumentation overhead by instrumentation insert and remove on-demand, based on performance analysis need. In this kind of approach, Paradyn allows us to measure top function calls and do a top down search for function time consumption. The tool has two processing kinds, the Paradyn daemon and the Paradyn application. The main components of the Paradyn application are:

- Performance Consultant: responsible for analysis of the performance data and requests for needed instrumentation.
- Visualization Manager: handles the graphical display representing the performance information data.
- Data Manager: handles the communication of the Paradyn daemon.
- User Interface Manager: handles user interface commands.

The Paradyn daemon is executed on each machine where the application processes are running. The main components of the Paradyn daemon are:

- Metric Manager: responsible for storing the metrics data.
- Instrumentation Manager: has the responsibility for generating the instrumentation code and inserting the binary code into an application process. It uses, through DyninstAPI, an library for application binary code patching [48].

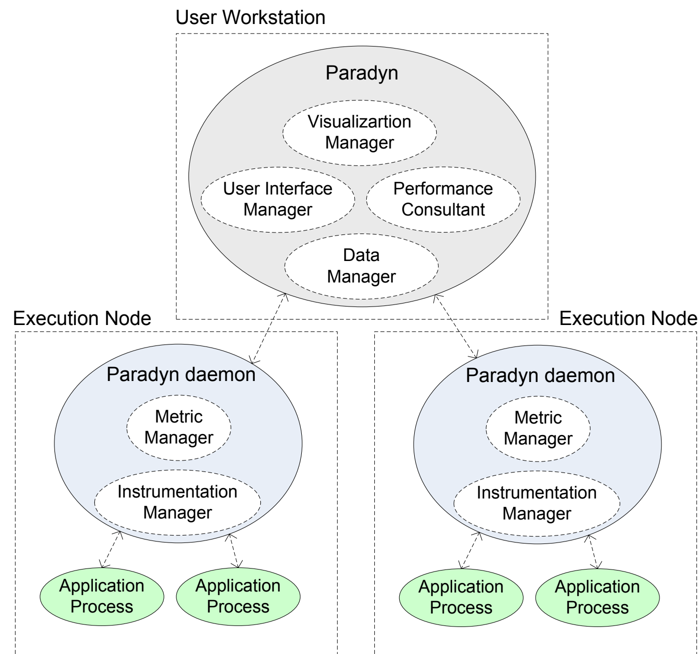



Figure 9 – Paradyn overview structure [54].

We will not cover the analysis features of Paradyn here due to our focus on the monitoring stage only. From the point of view of instrumentation and monitoring, the Paradyn tool uses DyninstAPI for process instrumentation. The instrumentation can be placed in a procedures entry, procedures exit and individual call statements. The metric manager uses six primitives in order to collect metrics: set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. In these instrumentation points, the primitives are inserted steered by the performance consultant. The values of counters and timers are collected periodically [54-56].

Chapter 3

# Grid Performance Models

Performance is a major issue in parallel programming. When a programmer develops a parallel application she expects to reach some performance indexes. In many cases the situation is even more critical, when the system presents dynamic changes (for example, load sharing changes) or application behavior varies during its execution due to data evolution. Therefore, over recent years, several efforts has been undertaken to provide automatic dynamic tuning tools that help users of parallel applications to reach those expected indexes.

The automatic dynamic tuning process consists of collecting measurements, evaluating current execution states based on a performance models and applying parameter changes in order to improve performance indexes. A dynamic tuning tool (MATE: Monitoring, Automatic and Tuning Environment) is presented in [57]. It carries out parallel application dynamic tuning on local clusters.

Computational Grids aim to provide the sharing of a large number of computational resources within different administration domains [2]. The Grid system can be used to tackle a high number of users, running many different classes of application, and/or to solve large problems. However, these systems are heterogenic and dynamic in nature and the situation described above is emphasized dramatically. Thus, the automatic dynamic tuning approach appears as an indispensable necessity in order to accomplish performance expectations.

In [58] the required changes in MATE to collect measurements on Computational Grids are presented. This chapter focuses on performance model development to enable dynamic tuning of parallel applications in Computational Grids.

A well-known problem in parallel programming is the load imbalance in master/worker applications. In these applications, an efficient execution depends on the balance between communication (data volume vs. networks bandwidth) and computation (task complexity vs. processors performance) [59]. The load balancing may be done statically, prior to application execution, or dynamically, using an application level schedule [60]. When the dynamic approach is used, finer grains workload divisions facilitate the computation load balancing, but increase the total communication volume [61]. Therefore, it is necessary to reach a trade-off that depends on the current conditions of the system, which change over time.

In this work, we aim to reduce execution time without losing efficiency. We consider efficiency to be the ratio between busy time and execution time of the processors assigned to an application. Dynamic tuning of granularity and the number of workers at runtime helps to reduce applications' execution time and improves execution efficiency in different scenarios of network and processor heterogeneity. We extend the work presented in [59], which provides a model for evaluation and tuning of multi-cluster post-mortem applications execution, to the Grid environments by providing a heuristic to tune dynamically the execution grain size and number of workers.

## 3.1 Related Work

There are many resources in the literature which address load balance in heterogeneous computing. We can categorize related work into approaches that suit parallel processing and approaches that take into account distributed computing. Parallel related performance models [61] use message latency as basic parameters to explain application behavior. Distributed computing performance models consider that bandwidth has more influence in application performance and it is considered to be a speedup limiting factor [59].

A comprehensive survey of load divisions strategies is provided in [62]. Options range from linear optimization considering heterogeneous scenarios and analytical multi-cluster analysis using multi level queue systems [63], to different scheduling strategies

assuming that master processes compare to meta-schedulers as workers compare to processing nodes [60, 64].

Javadi in [63] presents a greedy strategy for work distribution in which the priority in worker selection relates to processor speed. Indeed, these models do not take into account that changes in grain size impact on compute/communication ratios. Argollo in [59] states that the grain size tuning is done in application development processes. Machines on different networks in a multi-cluster system (local and remote cluster) should use different grain sizes. We differ by using a dynamic analysis of execution measurements, a heuristic resource selection and we advocate that applications may have different grain size selections within their execution due to temporal system heterogeneity.

Cesar in [61] provides a performance model for the optimal number of workers in parallel master/worker applications using finer grains considering dynamic tuning, but does not consider the variation in the total volume of communication in response to grain size variation. Morajko in [65] presents a factoring-based strategy for load division well suited for dynamic tuning. However this work, as well as [61], assumes that the communication volume does not change as a result of different load partition.

## *3.2 Performance Model for Dynamic Tuning*

Grid Systems are dynamic and users do not have control over the performance of available resources. Network channels and processors may be used in shared modes which result in variations in the available capacity over time. This aspect has a direct impact on the application performance indexes [7]. When users submit a job to be executed in a Grid environment, they do not know if the assigned resources meet their expectations. In some cases, the application cannot scale to all resources due to a communication boundary, and in others, the total execution time is limited by the assigned capacity of processor power. In a system such as a Grid, it is possible to have both problems within the same application version in different executions. Therefore our main goal is to minimize application execution time while increasing the efficiency of resource usage.

Application speedup in master/worker paradigms is limited by three constrains: total time to send operands (input data) from the master to workers through communication

channels, time to compute tasks in processors and time to receive result values (output data). In case of the master/worker paradigm, communication and computation may be overlapped to reach maximum system efficiency expressed as the occupation of workers' processors and master network interface capacity as presented in Figure 10 [59].

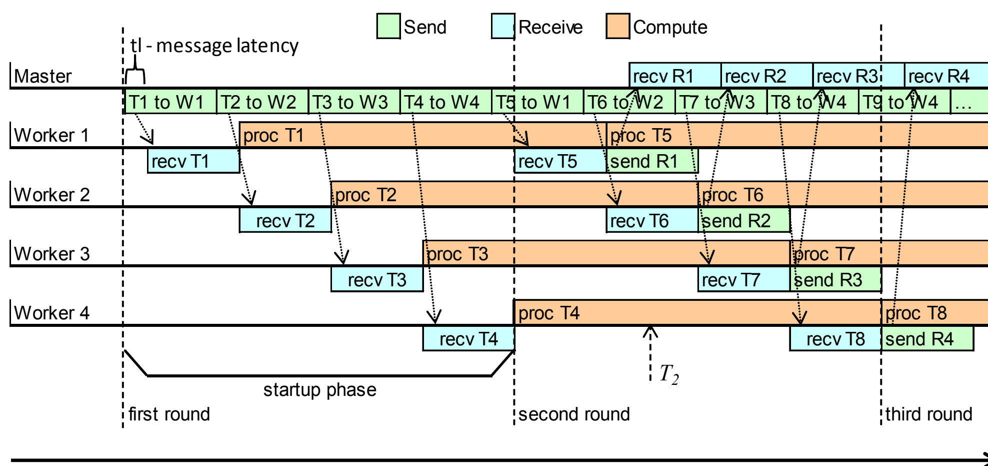



Figure 10 – Example of a master/worker startup where the time spent in each round corresponds to the time each worker process its assigned grain. At time $T_1$ all workers are performing computations and at time $T_2$ master network interface is saturated on both input and output communication capacity considering a full-duplex network port [59].

Consider a *round* as a sequence of grain distribution for all workers. Communication/computation overlap should be obtained in two initial *rounds* of task distribution. Subsequent data assignment follows a heuristic to feed faster processors first while maintain maximum execution queue of two tasks. All workers should have a work unit ready to be processed. Figure 10 illustrates that distribution pattern of a homogeneous environment. The same distribution pattern works on heterogeneous processor scenarios. Task scheduling is not addressed in this work. We focus on the impact of changes in the number of workers and application grain size.

When working with different grain sizes, applications may suffer processor cache interference. We assume that users may be shielded from such effects through using high performance processing kernels. Figure 11 present the total execution time of a

matrix multiplication using different sizes of grains and different multiplication kernels, ATLAS and GNU GSL. The standard deviation of ATLAS executions is 1.95% and GNU GSL is 2.26%.

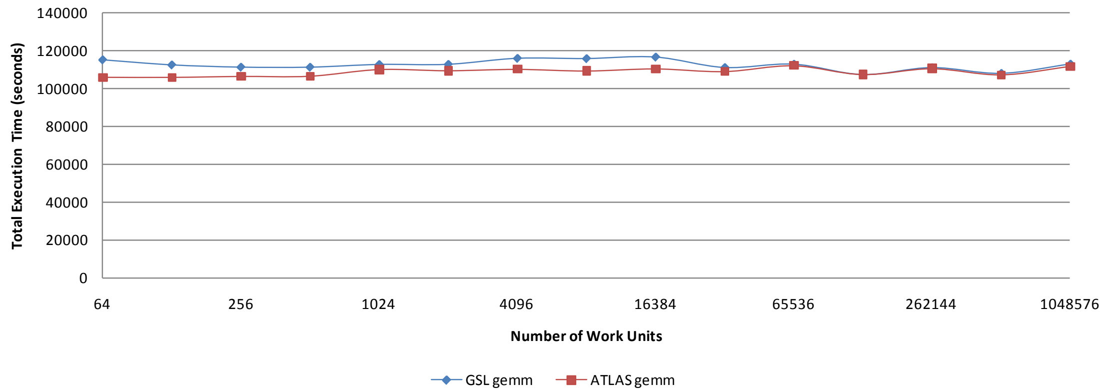



Figure 11 – Comparison between ATLAS and GNU GSL executions of a matrix multiplication problem using different grain sizes.

As defined in [59], application execution may be divided into three phases: startup, steady and finalization. In the startup phase master distributes work units to all workers. The steady phase starts when all workers are in a busy state and finishes when a worker becomes idle and the master does not have any work to assign. After a steady phase there is a finalization phase where master waits for remaining processing to be finished. We consider that the master's role is to manage workload distribution and results collection in order not to have a application bottleneck.

In order to deal with system heterogeneity, we focus on application load balance, applying dynamic tuning of application grain size and the number of processors. As a case study, we choose the master/worker programming paradigm. In this paradigm the change of application compute/communication ratio may balance the total execution time and efficiency. These behaviors can be seen in Figure 12 and Figure 13. Figure 12 plots different execution times for a master/worker application of varying grain size and Figure 13 plots executions varying the number of workers.

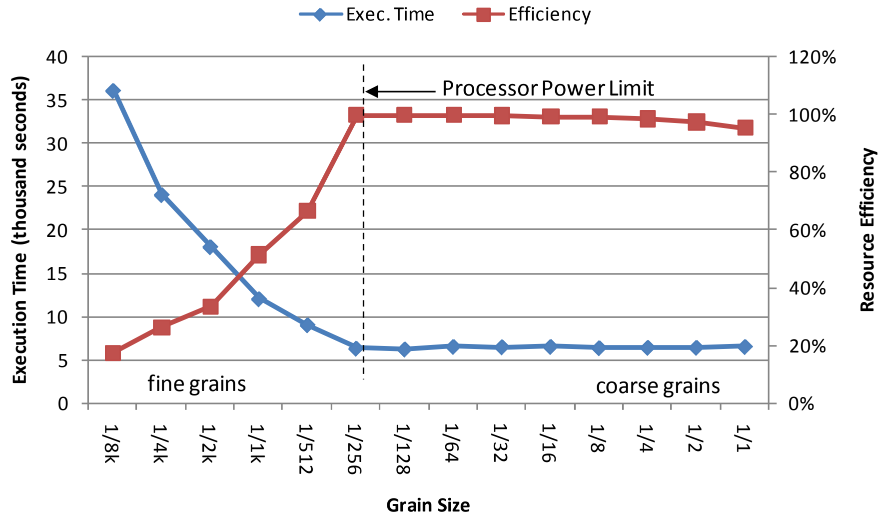



Figure 12 – Impact of grain size in total application execution time and usage efficiency of a fixed resource set size of 18 workers.

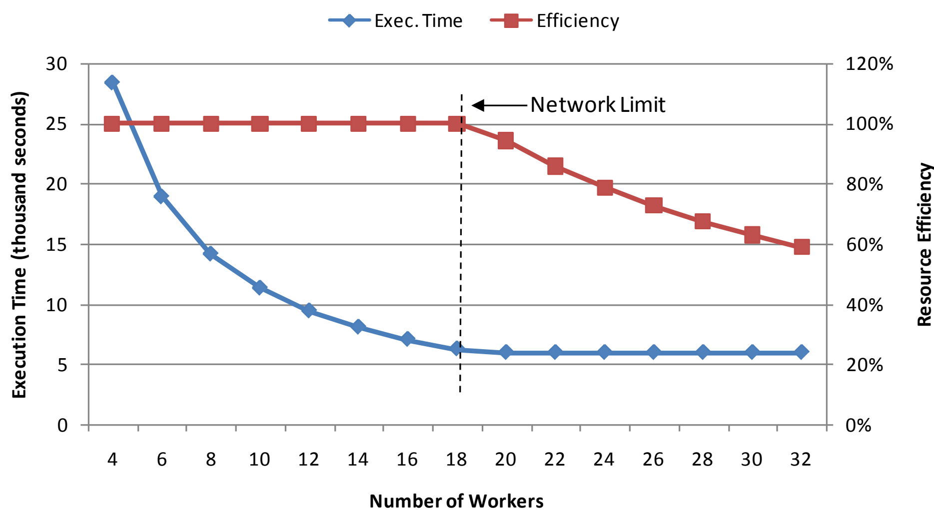



Figure 13 – Impact of the number of workers in total application execution time and usage efficiency of the assigned resource set.

In Figure 12, the network bandwidth limits the processor usage when using fine grains from 1/8k to 1/256. Using these grains makes the application communication-bound. When using coarse grains, an application becomes computation bound. This point shows that the coarser the grains, the more efficiency an application losses due to the impact of initial task distribution and final result collection.

In Figure 13, the network bandwidth and selected grain size limits the maximum number of workers to 18. If an application receives more workers than that, it will lower the resource usage efficiency without decreasing execution time.

### 3.2.1 Parameters Characterization

The characterization of the load is done by the analysis of which parameters have more impact in its composition. Some problems specify their load by the number of data items or the number of processing steps over some data. In order to find out how to change the compute/communication ratio, we need to find out which parameters are relevant to compute and communication and how these can be used to change the ratio. In the following, we use the abstraction that load consist in the transformation of input to output data as presented in Figure 14.

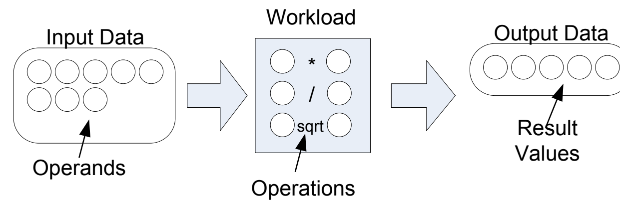



Figure 14 – Abstraction of the load through operands, operations and result values.

Suppose we have a problem with a workload $W$, characterized by $C$ operations performed over $Vi$ input data operands and that generates $Vo$ output result values. In such model, a machine which can process $S$ operations per second would take $C/S$ seconds to complete the workload computation. For some problems, the workload can be divided in ($wu$) equal slice parts we call tasks that: $wu = \frac{C}{gi}$ and $W = \sum_{k=1}^{wu} C_k^{gi}$, where $gi$ represents the grain size and $C_k^g$ represents a task unit with index $k$ using grain size $gi$ in load division.

Consider that $vi^{gi}$ is the volume of input data operands required to compute a task $C_k^{gi}$ and $vo^{gi}$ is the data volume generated in output data of result values of such computation while using grain size $gi$. The total volume of data needed to be transferred for remote execution is then $Vi^{gi} = \sum_{k=1}^{wu} vi_k^{gi}$ and the volume of generated results is

$Vo^{gi} = \sum_{k=1}^{wu} vo_k^{gi}$. During execution, $Vi^{gi}$ and $Vo^{gi}$ should be transmitted between the master and workers. Given the network characteristics, the lower bound limit of execution time is $\max(Vo^{gi}, Vi^{gi}) * \lambda$, where $\lambda$ is the average network bandwidth, considering overlap of input and output communications.

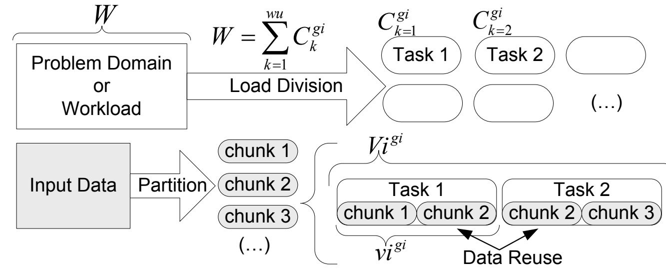



Figure 15 – Graphical view of the relation between data partition and load division scenario where grain size has impact on compute/communication ratio.

We work with loads that have data reuse among different tasks. That allows us to control the total communication volume using the grain size $gi$. The selection of different values for $gi$ may result in different values for $Vi^{gi}$ and/or $Vo^{gi}$ sizes, as presented on Figure 15.

A general rule of grain size selection impact on application execution is presented in equation (2). Data reuse among work blocks generates scenarios that smaller grain size produces more total communication volume than bigger grains. The higher is $gi$ value, the finer are the grains and, the lower is $gi$ value, the coarser are the grains distributed by the master.

$$\forall gi' < gi'' \Longrightarrow Vi^{gi'} \geq Vi^{gi''} \wedge Vo^{gi'} \geq Vo^{gi''} \quad (2)$$

### 3.2.2 **Metrics for Grain Decomposition**

Let $Tc$ be the total execution time required to process the $C$ application basic operations. In case where an application is communication-bound, when $(Vi * \lambda > Tc) \vee (Vo *$

$\lambda > Tc$), the communication between master and workers saturates the network interface bandwidth and limits the number of workers that can be used without efficiency decrease.

The granularity may be changed to improve the computation/communication ratio in order to increment the amount of workers while keeping a high efficiency. If all workers are performing computation and the master communication interface is saturated then it is not possible to change the grain size. In this case, a system is in full utilization and the application reaches its scalability limit.

We divide the work unit delivery into *rounds*. In a *round*, the master delivers one work unit to every worker. To achieve full network bandwidth utilization in master networks, the *round* time should match the time needed by the faster worker to process a work unit. That limits of the maximum number of workers given a grain size $gi$, named as ${N_{opt}}^{gi}$, are defined by equation (3). In such cases, the maximum number of workers is the time spent to process a work unit, $Tc^{gi}$, divided by the maximum between time spent to transmit input values $vi^{gi}$ and time to receive output values $vo^{gi}$, in a network with message latency $tl$ and inverse average bandwidth $\lambda$, as detailed in [59].

$${N_{opt}}^{gi} = \frac{Tc^{gi}}{\max(vi^{gi}, vo^{gi}) * \lambda} \tag{3}$$

$${T_{startup}}^{gi} = tl + \left(vi^{gi} * \lambda\right) * \left(\frac{N_w + 1}{2}\right) \tag{4}$$

$${T_{finalization}}^{gi} = tl + \left(vo^{gi} * \lambda\right) * \left(\frac{N_w + 1}{2}\right) \tag{5}$$

Equations (4) and (5), presents the behavior of the startup and finalization stages' length in an execution in a system with $N_w$ workers within a best execution scenario as detailed in [59]. In those stages, system efficiency is directly related to grain size selection. Bigger grains result in higher $vi$ and $vo$ which results in higher $Tstartup$ and $Tfinalization$ stage time values. The execution time from those stages has a direct

influence on overall system efficiency. An example of that influence is presented in Figure 12 where the efficiency decreases as the grain size gets finer.

This observation suggests the use of smaller grains, which represent a higher number of tasks to be distributed, as a solution to load balancing [20] and consequently shorter $Tfinalization$ time.

If we discard message latency, the efficiency of startup and finalization phases is detailed in equations (6) and (7). Equation (7) presents the combined efficiency in both phases considering the best execution scenario.

$$Eff_{startup}^{gi} = Eff_{finalization}^{gi} = \left(\frac{N_w^{gi} - 1}{2\ x\ N_w^{gi}}\right) \tag{6}$$

$$\lim_{N_w^{gi} \to \infty} \left(\frac{N_w^{gi} - 1}{2\ x\ N_w^{gi}}\right) = 50\% \tag{7}$$

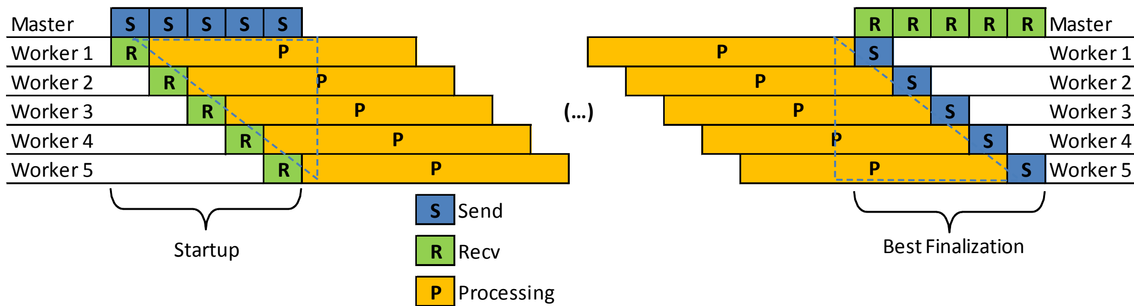



Figure 16 – Graphical view of the best startup and finalization phases.

There are some considerations that should be taken in account when the biggest grain is used. First, due to bigger grain size, all execution time is spent on both startup and finalization phases. As the best case of startup and finalization efficiency is 50%, we have processors idle in a system for about 50% of the execution time [23]. The idea is that the startup draws an upper triangle of busy processors and the finalization draws a lower triangle of busy processors. The right value depends on a ratio between the time spent to process a grain $wu^{gi}$ and the time spent in startup plus finalization. Second, the

application does not know prior its execution what network characteristics will be available at runtime in order to determine its processors' needs.

During runtime execution it is possible to verify if, given a set of resources, the application can scale and use them in an efficient manner or whether some resources can be released without harming the total execution time, while there is a rise in total efficiency. Considering a dynamic Master-Worker execution in scenarios of the heterogeneity of network links and processor speed, we can achieve the lowest execution time by tuning the grain size in order to allow the utilization of the maximum number of assigned workers and the number of workers to be released from unnecessary assigned compute elements without any penalization in execution time.

To have the benefits of dynamic grain size tuning, the application should support dynamic grain size change. This support consists of working with coarser tasks if values of $gi$ decrease and work with finer tasks if $gi$ increases. The $\lambda * vi^{gi}$, $\lambda * vo^{gi}$ and $Tc^{gi}$ averages should be measured continuously during application execution. These values are sensible to bandwidth/network latency variations and effects derived from process execution in shared environments.

Master-Worker applications with data reuse among different tasks should scale better using SPMD paradigms which explore better data locality. However, load balance is hard to achieve on heterogeneous scenarios using such paradigms. Some processors in a shared environment, commonly found in Computational Grids, slow down all application processes due to communication synchronization. Master-Worker paradigms also apply in scenarios where application input data comes from one place and under a low WAN bandwidth. The data required to process the problem comes from one site or storage element. This data is accessed from the master processor at one site. In cases under low WAN bandwidth, the data transmission should be as well managed as the computation because it has a higher impact on the total communication time.

### 3.2.3 Grain Size and System Heterogeneity

When communication and computation overlap in a Master-Worker application, each worker may be analyzed as a three stage pipeline composed of: task transmission, task processing and results transmission. For task of grain size $gi$, those stage times are $\lambda * vi^{gi}$, $\lambda * vo^{gi}$ and $Tc^{gi}$. The task round trip time determines how fast the processor is

from the master point of view. The lower execution time is obtained when the master uses the faster processors [62].

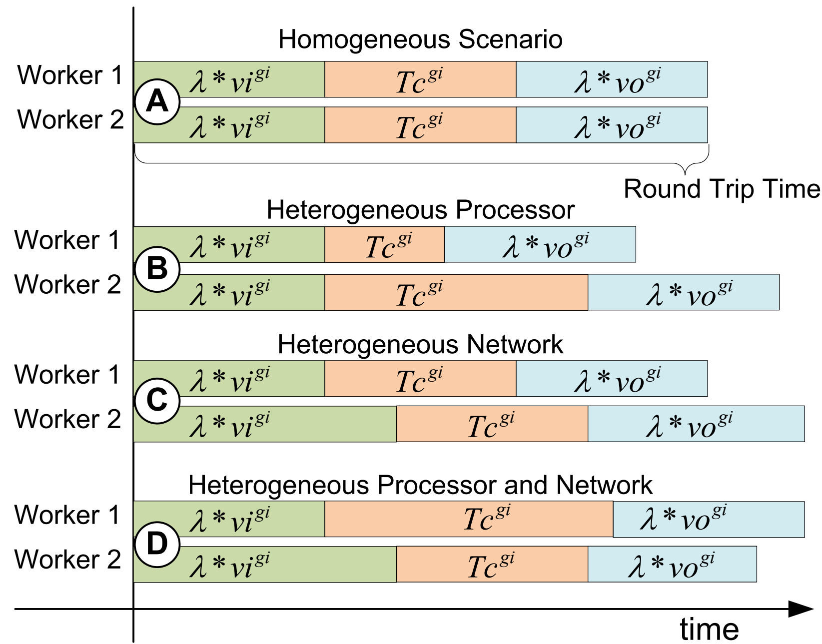



Figure 17 – Impact analysis of heterogeneity in compute/communication time values considering network and processor capacity variations.

Changing $gi$ value at runtime allows the tuning tool to find out if the system is communication-bound or computation-bound, given a set of resources assigned to an application. By tuning $gi$ it is possible to get better compute/communication ratios in heterogeneous systems.

In Figure 17 we present the distinct heterogeneous cases. In case A, the master perceives all workers as working at the same speed. Considering homogeneous networks, round trip time is limited by processor power capacity in case B. In case C, lower communication time reduces the round trip time of worker 1. In such cases, worker 1 is considered faster than worker 2.

The case D presents a particular sample of two workers with different processor powers and distinct network accessibility. This case illustrates a worker speed from the master point of view and is related directed to processor power and network capacity.

To lower execution time in cases of processor heterogeneity in master-worker applications, the master must first choose faster workers for task assignation to get the lowest execution time. However, considering master process execution on a one port machine[2], communication time should be considered as serial due the network bandwidth limit. Working with coarse grains allows a decrease in the communication volume and increases the maximum number of workers that the master can feed.

Concurrent task communications increase $\lambda * vi^{gi}$ and $\lambda * vo^{gi}$. In cases of network heterogeneity, the use of communication managers[3] isolates task delivery throttling caused by communications within slow networks [59]. In application parallelization supports task composition/decomposition, it is possible to configure a hierarchical master-worker as presented in [23]. With such capability, in a multi-cluster configuration, remote clusters receive coarser tasks and decompose them into finer ones for local processing. The remote decomposition/composition lowers the time to process coarser tasks on remote clusters and reduces the required communication volume.

### 3.2.4 **Dynamic Tuning Requirements and Process**

Dynamic tuning techniques consist of three main phases: monitoring, performance analysis and program/system modification [57]. The technique is centered on performance models. During tuning processes, performance models specify what should be measured by applications and systems, and suggest modifications which can be applied to applications in order to obtain better performance indexes.

Basically, the performance models explain *why* an application has some performance problem based on *what* is measured and *when* and *how* it should be measured. As a result of that process, it produces *what* should be changed, and *when* and *how* this should be done. All processes are performed without user intervention.

---

[2] That is a very common scenario. In that case, the machine where the master processor executes has only one network card. Overlapped send operations competes to network bandwidth.

[3] Communication managers are just processes with the role to proxy communication operations from network with different characteristics. The idea behind that model is to avoid blocking due to transmission through low bandwidth links.

Our group had a tool called MATE, to perform dynamic tuning of parallel/distributed applications in clusters [57] and it was used in the tuning of master-worker applications in such systems [66] using the performance models developed in [61]. Previous work has presented that the more dynamic is the system, the more the benefits achieved from dynamic tuning [14].

Inside the tuning tool, each performance model is encapsulated in a component called *tunlet*. During a monitoring phase, the *tunlet* interacts with its container to command what should be instrumented in application processes. Such instrumentation produces measurements that are transferred to the *tunlets*.

During an analysis phase, the internal logic of *tunlet* evaluates metrics based on a coded performance model. During modification phase, such logic decides what should be modified in order to raise performance indexes.

When applying the dynamic tuning technique based on models over Grids Systems, some aspects should be considered:

- The tool must be running on the machine in which application processes are running and should be able to communicate with the *tunlet* container. In Grid Systems where the assignation of machines to the application is controlled by Meta-Schedulers/Resource Brokers, users do not have control over where an application runs. The tool modifications required to address such a problem is presented in [58].

- Some measurements and modifications should be performed in different [2] Grid software layers. For example, to change the number of resources assigned to an application, the tool should make a request to the collective layer (Meta-Schedulers/Resource Brokers) and modify application processes in order to use obtained resources.

- The monitoring message amount may interfere in application communications, which require a reduction of data produced by instrumentation and modification commands.

Once these aspects are supported, the tuning process is steered by the equation (3) to find what should be the number of workers and the heuristic selection to change the grain size used by that application. Equations (4) and (5) are used to calculate the startup and finalization phases and their impact on the predicted execution time.

To calculate the required metrics specified in section 3.2.2, it is necessary to collect measurements as detailed in Table 1. Figure 18 associates such measurements to its instrumentation locations in master and worker processes. At moment A, the *tunlet* can obtain the compute/communication ratio in respect to $\lambda * vi^{gi}$. At moment B, the same can be done in respect to $\lambda * vo^{gi}$. The other parameters required for tuning may be calculated in sequence.

Using the measurements described in Table 1, the network latency value $tl$ may be obtained by the difference of $tms1$ and start point of $twr$. The inverse bandwidth $\lambda$ from upstream and downstream communications can be obtained from $vo$ divided by average $tmr$ and $vi$ by $twr$ average respectively.

By using equation (3), the tuning engine may verify what should be the optimum number of workers, notated as $Nop$, for runtime $gi$ value. The *tunlet* may suggest changes of the number of workers to $Nop$ or the $gi$ using the following heuristic.

The number of workers analysis consists of a continuous evaluation of equation (3) during runtime and allows for the following tuning actions:

- If $Nop > nw$, workers processors are running under maximum efficiency but master network interfaces may accept more workers. This may configure a heterogeneous network and/or heterogeneous processor scenario which will be analyzed in a following section. In a case when it is not possible to add more workers, $gi$ should be increased to reduce possible load balance problems.

- If $Nop < nw$, a system is running bellow maximum efficiency. In this case, the required tuning action is to decrement $gi$ value in order to use more workers if $g > 0$ and change the value of $nw$ to $Nop$, if $gi = 0$.

- If $Nop = nw$, a system is at maximum efficiency and its lowest execution time with the assigned set of resources.

Table 1 – Parameters used in dynamic tuning within a Grid parallel application execution.

| Id | Description | Location | Semantic |
|---|---|---|---|
| wu | Total number of work units | Master Binary | read |
| wp | Work units waiting to be processed | Master Binary | read |
| gi | Suggested work class (grain size) | Master Binary | read/write |
| vo | Bytes received as output values | Master Binary | read |
| vi | Bytes sent as a work unit | Master Binary | read |
| nw | Amount of available workers | Master Binary | read/write |
| tms1 | Time when master starts a task send of a work unit | Master Binary | read |
| tmr | Time between start and end in output result receive | Master Binary | read |
| twr | Time between start and end in work unit receive | Worker Binary | read |
| tc | Computation time of a work unit | Worker Binary | read |
| tws1 | Time when worker starts send an output result | Worker Binary | read |
| pinfo | Processor Information (frequency and architecture) | Worker Machine | read |

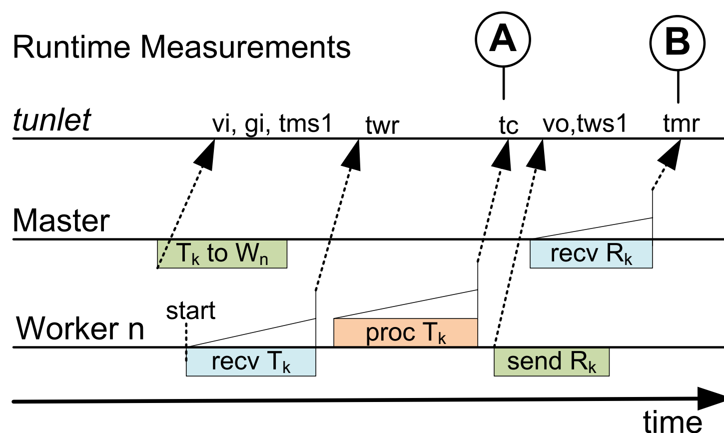



Figure 18 – Presents a graphical view of *where* and *when* measurements are gathered in different processes.

There is no sense in incrementing $gi$ value when a system is running below maximum efficiency because this would be likely to minimize the computation/communication ratio. A simple restriction to $gi$ values would be the required work units of coarser $gi$,

that $\sum vi^{\min(gi)} + \sum vo^{\min(gi)}$ fits in the available memory information provided by measurement $pinfo$.

To allow the dynamic tuning of grain sizes, such parameters should be exposed as a program variable and should support changes at runtime or exposes some function call that changes the grain size in response to a call. With this capability, the application can be tuned by an external tool using the library DyninstAPI [48]. All collected measurements should be gathered and analyzed by the tuning engine. A complete architecture example for clusters is presented in [14] and its adaptations to the Grid System is presented in Chapter 4 and published in [58].

## 3.3 Tuning in Heterogeneous Scenarios

The evaluation of equation (3) fits the scenario where processors are homogeneous. In the case of heterogeneous scenarios, the analysis of the maximum number of workers is obtained by a network bandwidth allocation heuristic.

First, we need to examine the different scenarios a master-worker application encounters when running in a computational Grid. When users submit their application for execution, they specify the application requirements in job description language. Based on number the of machines present in such requirements, the Grid resource broker may assign different resource groups to match job needs which configures different levels of heterogeneity for processors and network links. For example, suppose a scenario where a user has a parallel job request submitted to a Grid, specifying a requirement for seven processors. If there are resources available, the Meta Scheduler can assign one Compute Element (CE) containing all requested Compute Hosts (CHs). However, different scenarios may occur where requested CHs are scattered among more than one CE.

The following taxonomy classifies those different heterogeneity scenarios, from the master processor point of view:

- *Homogeneous Processors and Homogeneous Network (HoPHoN).* This scenario consists of a common cluster of dedicated processors. Examples are Ethernet based commodity clusters exposed to the Grid by a head node with two network interfaces. Such scenarios may occur when Grid meta-schedulers assign all

application processes to a single homogeneous CE composed of many CHs or when the master processor is mapped to a CE with all workers in a different CE.

- *Heterogeneous Processors and Homogeneous Network (HePHoN).* In this scenario, network characteristics remain the same in bandwidth and latency for the master/worker paradigm and the compute time for the same piece of work may differ with different group of processors. As with a *HoPHoN* scenario, applications may receive a heterogeneous cluster by means of a meta-scheduler assignation.

- *Homogeneous Processors and Heterogeneous Network (HoPHeN).* The scenario described in [33] reports that 75% of machines are homogeneous. This scenario has two variants: the master may use only one network interface for local and remote communications or may have different network interfaces for local and remote communication.

- *Heterogeneous Processors and Heterogeneous Network (HePHeN).* In such scenarios the Grid resource broker may assign different groups of machines distributed over different organizations. The master should deal with processor heterogeneity in local clusters and in remote assigned workers. This scenario has also two variants: the master may have only one or have more than one network interface as in the *HoPHeN* scenario.

Those scenarios presented in Figure 19, represent different levels heterogeneity caused by the dynamic behavior of Grid Systems. Some homogeneous scenarios may become heterogeneous during application execution. There are many examples of multi-core and SMP machines inside a CE that are exported to a resource level as multiple CH. That may temporally slow down concurrent processes if different cores/processors share machine memory.

Due to continuous application monitoring, it is possible to adjust application processes during execution using the following heuristic analysis to support scenario changes:

- If scenario matches a *HoPHoN*, then the analysis presented in section 3.2.4 is sufficient to choose the optimum grain size and number of workers.

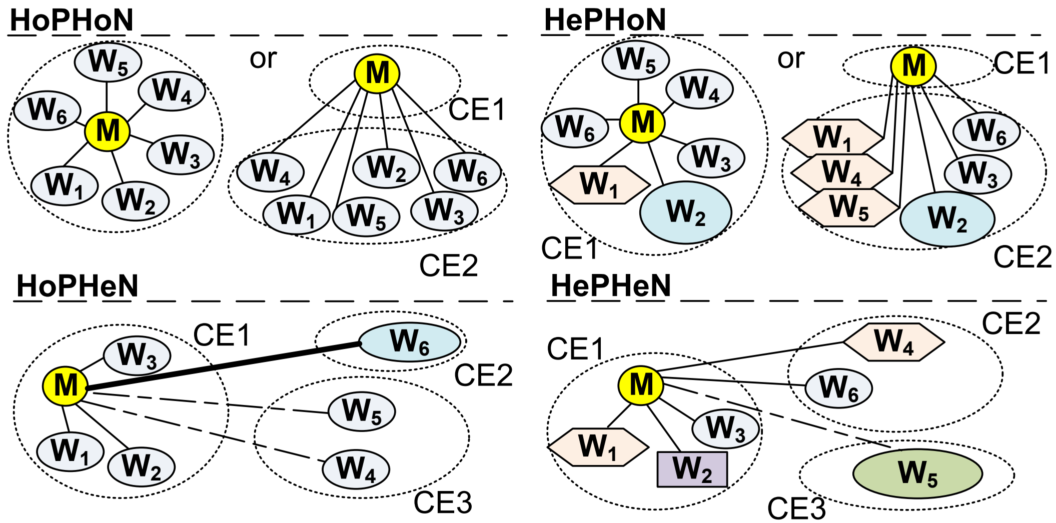



Figure 19 – Graphical views of different possible assignations from Grid Resource Broker/Meta-Scheduler in response to a request of seven compute nodes resource.

- In the case of *HePHoN*, the proposed strategy used to deal with processor speed heterogeneity is an adaptation of the methodology of multi-cluster execution tuning proposed in [59] to each group of homogenous processors as if they were remote processors. Consider $Wn$ as the total number of workers composed by $Wgn$ groups of workers with $Wg$ processors classified by speed where $Wn = \sum_{k=1}^{Wgn} Wg$. Let $pp_k$ be the ratio between the slowest processor and a sample of $Wg_k$ and $k$ to index groups from faster to slower ones.

  In such structures, the heuristic to tune *HePHoN* using the same measurements nomenclature from section 3.2.4 is presented in the pseudo algorithm in Figure 20. The goal of the heuristic presented in such a pseudo algorithm is to allocate available master output/input bandwidth to faster workers. The allocation of master processor input/output network bandwidth is based on the ratio between different task processing time measurements among heterogeneous workers.

- Same heuristics can be used in the case of *HoPHeN* with some modifications. First, local *Wg* groups are taken into account before remote ones. For the variation of the scenarios, where the master has two or more network interfaces, a remote group of workers should compete for the external interface. That is noted on runtime

measurements and the heuristic reacts to this shared condition to balance the computation/communication ratio.

```
# Variables Description
# slot      - Allocated network busy time
# nw        - Allocated number of workers
# Wg(k)     - Homogeneous set of workers, index k
# Nopi      - Calculated optimal number of workers
# gs        - Grain size to set
# Tc(k)     - Moving average of compute time, index k
# Nopt      - Tuned optimal number of workers


let slot = 0 and nw = 0
for each Wg(k) from k=0 to Wgn-1
   # measure required parameters
   let Nopt = calc(Nopt for Wg(k))
   if Nopt <= Wg(k) then
      if gs < max(gi) then
         # tune grain size
         let gs = gs + 1
      else
         # network saturated
         let Nopt = Nopi
      end if
      exit for
   else
      let Tc(k) = (Tc(0)-slot)*(Tc(k)/Tc(0))
      let Nopi = calc(Nopt in Tc(k))
      if Nopi < Wg(k) then
         # network saturated
         let Nopt = Nopi
         exit for
      else
         let Nw = Nw + Wg(k)
         let slot = slot + Wg(k)*(Tc(k)/Nopi))
      end if
   end if
end for
```

Figure 20 – Pseudo algorithm of optimum number of workers and grain size tuning. It uses runtime metrics to decide if an application needs changes in the compute/communication ratio and/or change in number of workers in use.

- In a *HoPHeN* scenario, the task round trip time is used to sort the list of workers to help finding which processors should be selected. This approach consists in allocating the bandwidth to faster processors until a match in the *round* time. That is

done using the *slot* variable in the pseudo-code presented in Figure 20. It is done for master communication ports when the master has more than one interface. In the case of one network interface, the available network bandwidth (which may be used to transmit work units to remote workers) could be obtained by the difference between $Tc(0)$ and the *slot* time allocated to the transmission to local workers.

The initial grain size selection is defined to have the number of tasks with one order of magnitude higher than the number of workers according to the recommendation from [20]. As soon as worker finishes one task, the tool can start to evaluate if the grain size selected needs to be changed or not. As the analysis continues, it may need many rounds to reach the best grain selection depending on how good the first estimation was.

The selection of finer grain at a premature point in the tuning process helps with an initial evaluation of network and processor characteristics. Coarse grains however, have better computation/communication ratios and consume less network resources. Indeed, as presented by equations (4) and (5), coarse grains have a large startup time, which lowers resource usage efficiency. Iterative applications have such penalization risks only on first iteration. Following iterations should use the more recently tuned iteration grain size and value for the number of workers.

## 3.4 Effects of Data Access Patterns

The operand reutilization among different grains suggests that a parallelization strategy should use a paradigm other than a Master-Worker in order to reduce communications such as SPMD or pipeline. The problem is how to deal with heterogeneity in such paradigms. Pipeline load balance is hard to achieve using homogeneous networks and a processor and should be even harder in heterogeneous scenarios.

When we parallelize an application using the Master-Worker paradigm and add the support for grain size change, we play with the total volume of the communications to change the compute/communication ratio. In cases of very slow network bandwidth, this introduced redundancy may be too costly. One alternative to overcome this problem is to introduce a cache of operands on master to worker communications. This reduces the volume of communications and allows for better application scalability. However, this scheme requires an increase in the complexity of the task level scheduler. In such

cases, the task scheduler deals with task affinity in order to have the maximum benefits from operand caches.

The basic idea of an operand cache is to structure tasks in data chunks and label the redundant data chunks. All operand caches are orchestrated by a master. In other words, master processors knows what data is cached by clients and has a consensus about whether a partial task content can be reconstructed on the assigned worker processor. The task scheduling can use a greedy strategy and operand cache hit is a metric that can be measured on runtime. The operand cache should be done in the memory, or on a local disk in cases where the local disk is faster than the network bandwidth.

Consider an index of chunk cache hit $cch_{nw}^{gi}$ which represents the percentage of work reutilization by an application using grain size $gi$ and number of workers $nw$. From a performance model point of view, each problem mapped on Master-Worker paradigms may have different grain size options, and because of that, different cache hit ratios. By using an operand cache strategy, we have better scalability of applications with shared operands among different tasks when implemented using a Master-Worker paradigm by lowering the network bandwidth requirements.

Let's use an example to illustrate the use of the operand cache in Master-Worker paradigms. Consider a matrix multiplication A(M,K) by B(K,N), resulting in a matrix C(M,N). Fist we need to choose a parallelization strategy to map to a Master-Worker. One of the approaches is to divide matrixes A using first dimension by $i$ and B using second dimension by $j$ generating $i \times j$ grains to be scheduled, transferred and processed by worker processors. That strategy allows for work break-down which results in a change of the total volume needed to be transferred to workers, as explained in section 3.2.2.

A simple greedy heuristic for grain scheduling is, given a time to send a grain to a worker $wi$ the scheduler, to choose the task that has more probability of being cached by worker $wi$. Let $i^{wi}$ be the number of blocks from $i$ where $i^{wi} \subseteq i$ and $j^{wi}$ are the number of blocks from $j$ where $j^{wi} \subseteq j$ assigned to worker $wi$. For maximum cache hit, $i^{wi} \times j^{wi}$ should be maximized, so, $i^{wi} \cong j^{wi}$ .

For example, if the worker 1 receives 4 work units, it might have different cache hit values depending on scheduling options. In that case the following $sched(i^{wi}, j^{wi}) = \{(4,1)\ (2,2)\ (4,1)\}$ satisfies 4 work units assignation. Indeed, cases $\{(4,1)\ (1,4)\}$ have cache hit 3 blocks in 8 transmissions, $cch^{gi}_{nw} = 0.38$ while case $\{(4,4)\}$ have cache hit of 4 blocks in same 8 transmissions, $cch^{gi}_{nw} = 0.5$.

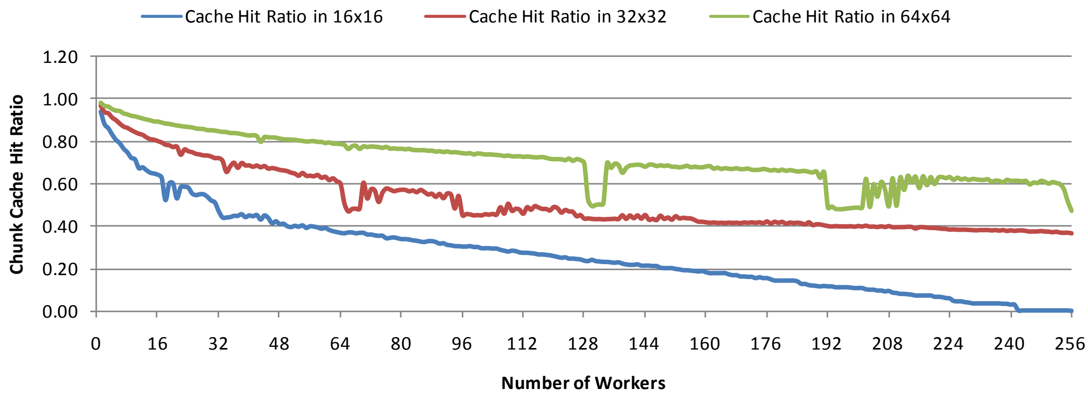



Figure 21 – Plot of cache hit ratio by number of workers considering 256 work units.

Figure 21 presents the cache hit ratio obtained in a matrix multiplication problem using different division strategy. The divisions using 16 by 16, 32 by 32 and 64 by 64 blocks result in 256, 1024 and 4096 tasks, respectively. The cache hits in that case depend on the number of workers as the problem chosen grain size. Finer grains allow better load balance in task distributions and better data reutilization in worker processing nodes. The authors in [67, 68] provide an extensive evaluation of different strategies for scheduling Matrix-Multiplication within homogeneous and heterogeneous scenarios.

## 3.5 *Simulating Master-Worker in Heterogeneous Scenarios*

The main goal of our experimental scenarios is to inspect the reduction of total application execution time while increasing efficiency using dynamic tuning of the number of workers and grain size selection in a master/worker application. To accomplish this goal, a master/worker discrete events simulator was built based on

SMPL [69], with the support of a custom heterogeneous group of workers connected by LAN or WAN link.

Based on real parameters, the simulator can predict the behavior of a dynamic master/worker application with grain size and number of workers change support. The scheduling policy used is based on a minimal queue. The master scheduler sends tasks to the faster worker with a minimal queue. Each worker has a queue of two tasks at maximum. The idea is to have one task in a processing state and another in a receiving state.

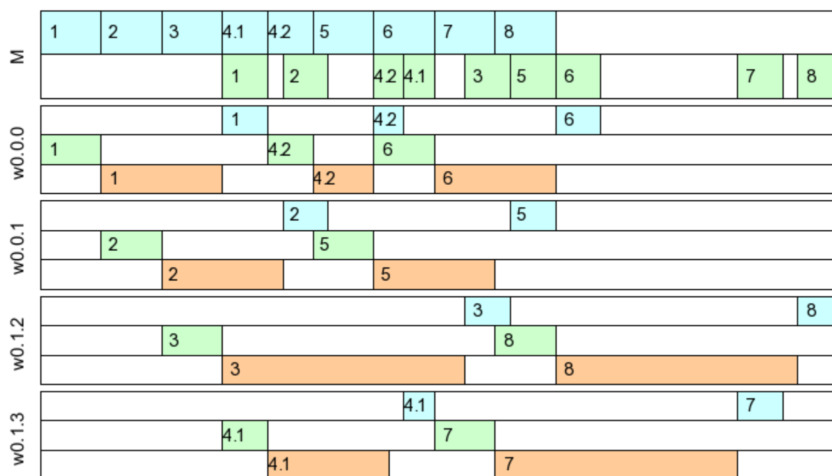



Figure 22 – This presents the simulator output of a master/worker execution with dynamic grain size change. Blue bars are sends and green ones are receives. Orange bars are task processing. The numbers inside the bars are the task number. Note that task 4 was processed using a different grain size and workers 2 and 3 have different processor power than workers 0 and 1.

The task distribution process tries to overlap computation and communication in order to decrease worker idle times. The simulation engine was validated using the performance model of multi-cluster executions from [59], with parallel applications using MPICH-G2 [70] based on an application template which supports dynamic change of grain size and number of workers. The simulator and the application template share the same task schedule strategy code. It allows for visual event debugging where task transmission, execution and response transmission are presented as a Gantt chart, as can be seen in Figure 22.

To support our goal we analyzed the gains/losses in execution time and efficiency, considering the tuning of grain size and number of workers in the different mapping scenarios presented in section 3.3. The mapping of processors in different CEs is done by Grid Meta-Schedulers/Resource Brokers at application startup. Thus, users cannot choose the initial best grain size because the processors and network characteristics may vary along with execution. We consider that if the dynamic tuning technique shows gains in all those scenarios, whatever scenario the application encounters, it will have the benefits from dynamic tuning.

The grain size range was selected to allow for the modification of application runtime behavior from computation-bound to communication-bound. In the configured scenarios, we have the same simulation, with the different cases varying only the grain size: these cases configure computation-bound and communication-bound executions. Coarse grains cause computation-bound executions and finer grains configure communication-bound ones. The range is wide enough to cover the processor power and network characteristics variations considering the presented parameters. The results presented cover executions considering all grain size selections as startup parameters to compare with the same basic case using dynamic tuning.

The application workload was a matrix multiplication having $A(32k, 8k)\ x\ B(8k, 32k) = C(32k, 32k)$ single precision elements. The mean and deviation of task processing time was obtained by a real application execution. The reference machine for the measurements of task execution times was an Intel Pentium IV 2.8Gz class processor with 512Mb of memory.

The LAN input parameters were obtained through measurements sampled by a simple MPI application using MPICH within a Fast Ethernet switch. The WAN link profile metrics where obtained using the *iperf* tool between machines of UAB *aogrdini* and UOC *dpcsgrd* machines.

The grain size[4] $gi = 0$ was set to $A^0(4k, 8k)\ x\ B^0(8k, 4k) = C^0(4k, 4k)$ operands, which corresponds to a maximum grain size value which the reference machine

[4] Note that we choose to have the maximum grain size defined by the value zero. The grain size value should be perceived as a denominator of a coarser task workload.

configuration could process without swap use. We sampled grain sizes from $(4k, 4k)$ to $(512, 512)$ elements of $C^g$ block of result matrix using ATLAS algebra kernel package, which isolates us from cache effects. The strategy for obtaining the different grain size used was a recursive alternate dimension division. For example, $gi = 0$ have $C^0(4k, 4k)$, $gi = 1$ have $C^1(2k, 4k)$, $gi = 2$ have $C^2(2k, 2k)$ and so on. Such a strategy divides the number of operations, and consequently the execution time, by almost two. A detailed explanation of that division is provided on section 5.3.2.

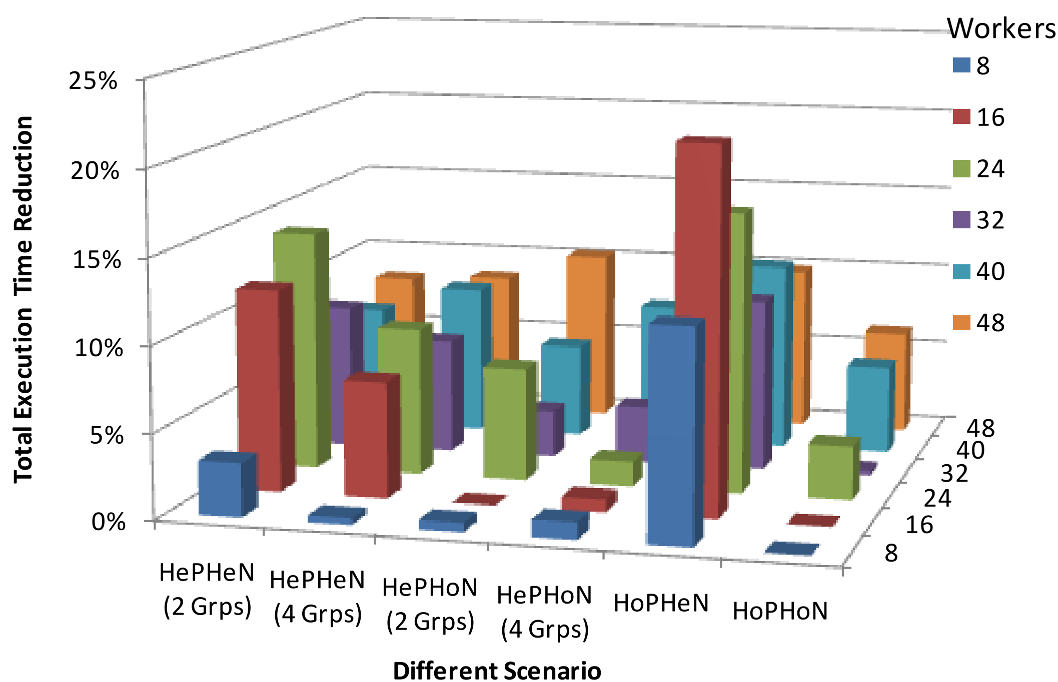



Figure 23 – Overview of application execution time reduction by using dynamic tuning of the number of workers and grain size in all heterogeneous scenarios and the number of workers assigned to application.

The scenarios presented in section III were composed as follows. Simulations were performed using different job requirements of 8, 16, 24, 32, 40 and 48 nodes.

In order to test processor heterogeneity each job requirement was also simulated with 2 and 4 CE composed by heterogeneous set of workers. The processor speed of these sets was normalized to reference processors resulting in (0.8, 1.3) relative processor power for simulations with 2 groups, and (0.8, 1, 1.3, 1.6) for simulation with 4 groups.

The simulation also uses as a parameter the standard deviation for all model input values. The tuning process was also implemented inside the simulator considering the transmission of the measurements to a different host for analysis. Each experiment was repeated using the 16 seeds for random number generation. These seeds are provided by SMPL as streams. All these cases were executed with or without dynamic tuning.

The cases without dynamic tuning were executed with all possible sixteen values for grain size. In cases of heterogeneous processors in heterogeneous networks (*HePHeN*), three extra scenarios were generated: faster processors placed locally in LAN, distributed equally and placed on WAN. With these parameter sweep configurations, a total of 65536 simulations were executed.

Figure 23 presents a general view of the application execution time reduction while using dynamic tuning techniques in all presented heterogeneous scenarios and number of workers. Same characterizations can be seen in Figure 24 from the point of view of resource efficiency improvement. As can be seen, major benefits from efficiency are found in scenarios with a heterogeneous network.

The maximum execution time reductions are 21.4% and 16.5% in *HoPHeN* scenarios with 16 and 24 workers respectively. In the *HoPHoN* scenario, the task distribution strategy used is near optimal for the small number of workers, as seen in classes of 8 and 16 workers. The best efficiency gains are 19.7% and 19.0% presented in *HePHeN* scenarios with 4 groups of heterogeneous processors in classes of 24 and 40 workers, respectively. Higher resource usage efficiency improvements were found in scenarios with heterogeneous networks *HePHeN* (2 Groups), *HePHeN* (4 Groups) and *HoPHeN*.

Figure 25 presents a flattened view of efficiency gains in all experiments with the different number of workers as single values in the tested scenarios, considering different groups of heterogeneity. The same behavior from the point of view of execution time is presented in Figure 26.

Considering all job sizes, the *HePHeN* scenario with 2 and 4 groups of heterogeneity presents reductions of 8.1% and 6.7% in total execution time and the *HePHoN* scenario show reductions of 2.7% and 2.3% respectively. *HoPHeN* scenarios show reductions of 14.2% and scenario *HoPHoN* present reduction of 1.4%. The proposed tuning strategy

lowers total execution time by 6.8% while raising resource usage efficiency by 10.2% considering all cases.

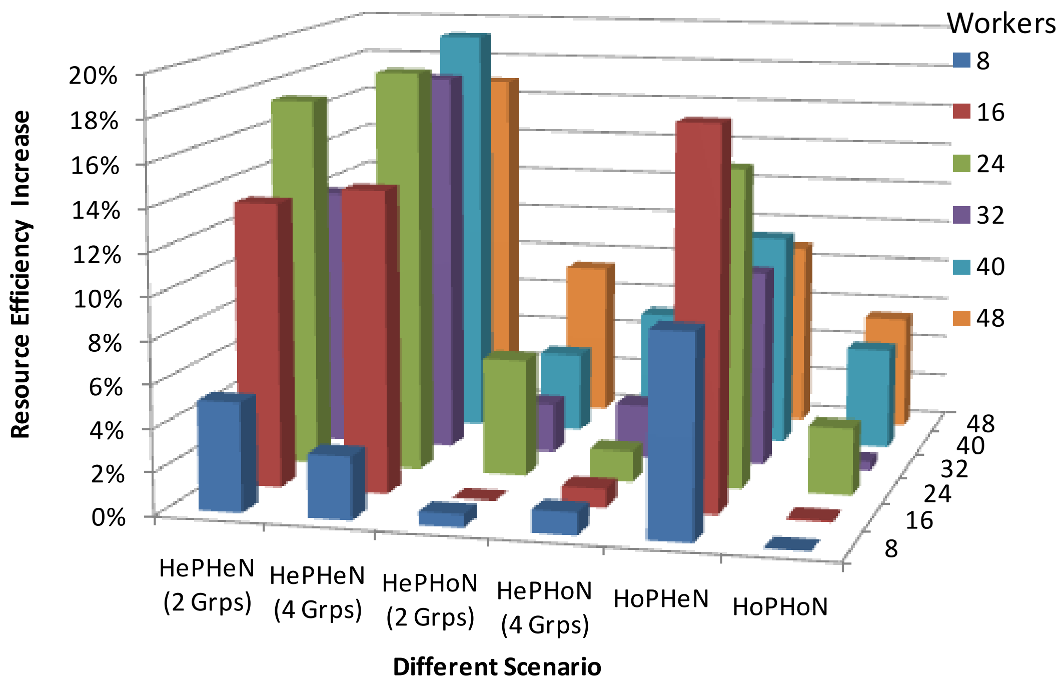
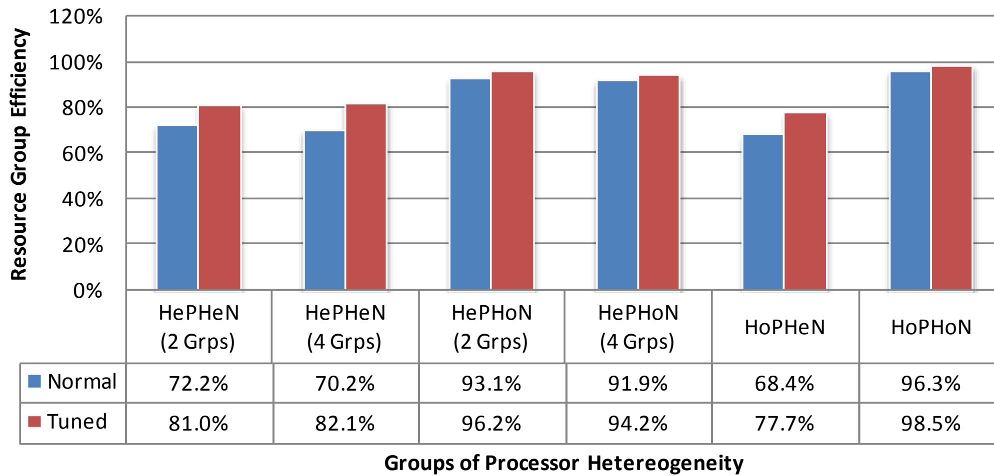



Figure 24 – Overview of resource efficiency increment as result of dynamic tuning of the number of workers and grain size in all heterogeneous scenarios and the number of workers assigned to application.

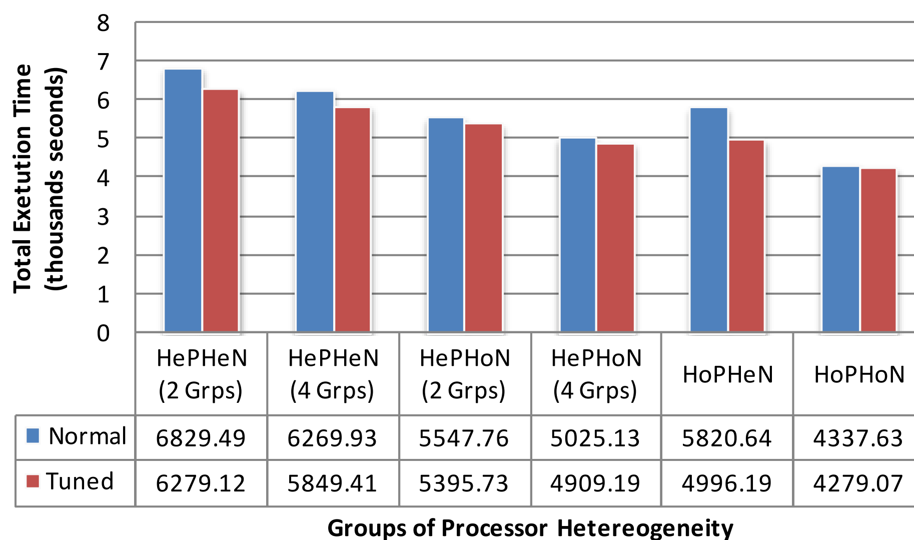
The gains are higher in scenarios with heterogeneous networks because the original application uses a dynamic task assignation on demand. This scheme assigns more tasks to faster processors which results in a better load balance. However, there were cases in which the tuned application presents no gain in execution time. In such cases, the applications were using the suggested optimum grain size and number of workers.

We have more benefits from dynamic tuning when the number of heterogeneous groups increases. In real world scenarios, when executed in shared environments, applications may face changes in their resource processing capacity over time. Such behavior increases the groups of heterogeneity and makes more important the use of dynamic tuning for grain size and the number of workers in these applications.




| | HePHeN (2 Grps) | HePHeN (4 Grps) | HePHoN (2 Grps) | HePHoN (4 Grps) | HoPHeN | HoPHoN |
|---|---|---|---|---|---|---|
| Normal | 72.2% | 70.2% | 93.1% | 91.9% | 68.4% | 96.3% |
| Tuned | 81.0% | 82.1% | 96.2% | 94.2% | 77.7% | 98.5% |

Figure 25 – Gains in resource usage efficiency in distinct groups of processor heterogeneity. We consider efficiency as the percentage of total computation available performance used during the application execution time.

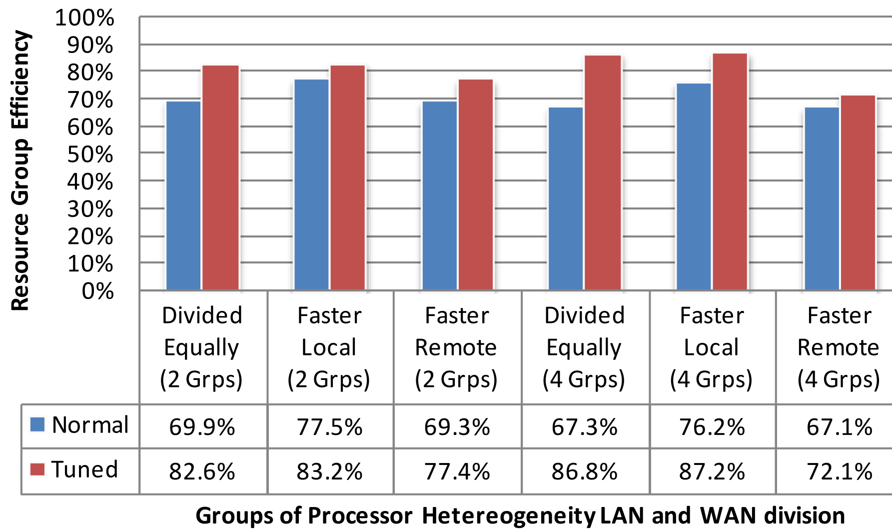



| | HePHeN (2 Grps) | HePHeN (4 Grps) | HePHoN (2 Grps) | HePHoN (4 Grps) | HoPHeN | HoPHoN |
|---|---|---|---|---|---|---|
| Normal | 6829.49 | 6269.93 | 5547.76 | 5025.13 | 5820.64 | 4337.63 |
| Tuned | 6279.12 | 5849.41 | 5395.73 | 4909.19 | 4996.19 | 4279.07 |

Figure 26 – Gains in total execution time considering different groups of processor heterogeneity, as shown, assigned to the application.

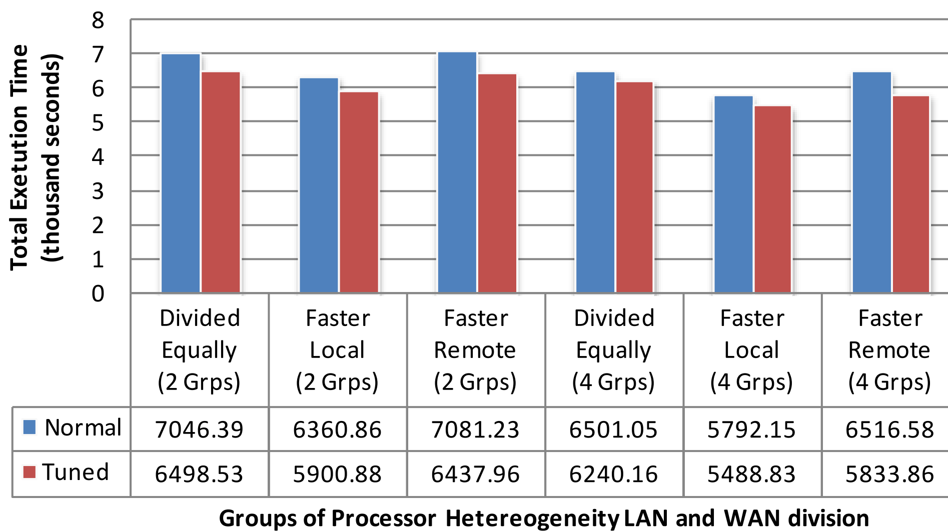
Considering only the *HePHeN* scenario, it is possible to have three different instances: faster processors on local networks, faster processors on wide area networks or close to equally distributed. Figure 27 and Figure 28 are flattening aggregations of gains in efficiency and total execution time respectively considering only these different cases of

*HePHeN* scenarios. The result values shows gains in all cases, from 4% to 14.2% in execution time and 7.3% to 29% in efficiency improvement, respectively.




| | Divided Equally (2 Grps) | Faster Local (2 Grps) | Faster Remote (2 Grps) | Divided Equally (4 Grps) | Faster Local (4 Grps) | Faster Remote (4 Grps) |
|---|---|---|---|---|---|---|
| Normal | 69.9% | 77.5% | 69.3% | 67.3% | 76.2% | 67.1% |
| Tuned | 82.6% | 83.2% | 77.4% | 86.8% | 87.2% | 72.1% |

Figure 27 – Efficiency of resource usage compared against different heterogeneous processors' distribution between LAN and WAN.




| | Divided Equally (2 Grps) | Faster Local (2 Grps) | Faster Remote (2 Grps) | Divided Equally (4 Grps) | Faster Local (4 Grps) | Faster Remote (4 Grps) |
|---|---|---|---|---|---|---|
| Normal | 7046.39 | 6360.86 | 7081.23 | 6501.05 | 5792.15 | 6516.58 |
| Tuned | 6498.53 | 5900.88 | 6437.96 | 6240.16 | 5488.83 | 5833.86 |

Figure 28 – Total execution time compared by heterogeneous processors distribution between LAN and WAN.

The more heterogeneous is the scenario, like mixing network and processor heterogeneity, the greater the benefits from dynamic tuning of grain size and number of workers. As shown in Figure 28, scenarios with more groups of heterogeneity present higher reductions on total execution time when tuned.

## 3.6 Summary

In Computational Grids, the main problem for achieving high performance is to scale applications, considering the high level of heterogeneity in network links and compute processors. To deal with that, applications should be developed to have good compute-communication relations or grain size. Indeed, different scenarios require different grain sizes for optimal execution. That motivates the architecture of applications with dynamic grain size support. Applications with such architectures can support changes from being computation- to communication-bound by runtime reconfiguration in response to tuning.

We proposed a heuristic to tune the grain size and the number of workers using the models of multi-cluster tuning found in [23]. Our main contributions are:

- Evaluation of assigned workers in heterogeneous groups;
- Infrastructure for dynamic grain size change support of master-worker applications;
- Heuristic for combined grain size tuning considering heterogeneous workers and placed in heterogeneous network links.

In this chapter we proposed a heuristic to change dynamically the application compute/communication ratio in order to reduce execution time while maintaining resource usage efficiency at certain levels. We contrasted our approach, considering four main scenarios of applications execution in Computational Grids. These scenarios combine different homogeneous and heterogeneous characteristics for processors and networks. A simulator for master-worker applications was built in order to test the dynamic tuning of these exhaustive scenarios. These scenarios are commonly found in Grid Systems environments.

The proposed heuristic for dynamic tuning provides significant reductions in overall time, considering all scenarios while rising overall efficiency. The obtained results indicate that it is possible to take advantage of dynamic tuning techniques to adjust

applications to execute within Computational Grids. Scenarios with heterogeneous network and heterogeneous processors obtained the best results in execution time decrease and efficiency increase. The best gains were obtained in scenarios with higher levels of heterogeneity.

# Chapter 4

# GMATE – Grid Monitoring Analysis and Tuning Environment

## *4.1 Overview*

The tuning based on performance analysis use with application runtime measurements feeds some performance models as model parameters and guides the tuning actions. As presented in Chapter 3, we can have performance models for different layers which cover restricted sets of parameters. As presented in Chapter 2, Grid Environments display heterogeneity in many properties which directly influence application performance. The major properties are machine and network heterogeneity. Note that a machine is a complex property composed of other important sub properties such as processor architecture, processor speed, cache size, memory size, disk space and speed. Each property can have its own location which can be measured in order to be analyzed.

The network parameter influences the communication between machines. In cluster system architectures, the network parameter is fixed or limited to an architecture. Each machine within a cluster expects the same network bandwidth and latency in communications through other machines in the same cluster. The network heterogeneity

is complex due to the interconnected network used for Grid communications. Communications between machines inside a cluster have different latencies and bandwidth properties than communications between machines outside the cluster but within the organization. Communications between machines located in distinct organizations have different communication properties, generally low bandwidth and high latency. Examples scenarios are provided by [71, 72] which present inter-cluster communications through the Internet.

All the heterogeneous properties of Grid Systems contribute to the complication of the performance analysis and application performance improvement. Differences in processor speed, for example, should generate load imbalance problems, different network properties should change the relation data transfer time to processing time, generating barrier contentions and idle time due to different data transfer delays. It is hard to balance the application to overcome such problems because most parameters are known only at runtime. When the user submits her application through the Grid Protocol Stack, a resource broker can select a different group of machines for an application's execution for each submission. In such scenarios, to monitor and tune the application is to deal with application process locations within the Grid and cross organization process and analysis of data gathering and tuning actions in response to that analysis.

Another important aspect of Grid Systems is that most heterogeneous properties are dynamic. Shared machines may have different processor speeds or available memory measurements, due to external loads. The network property measurements have the same behavior. Latencies and bandwidth capacity can dynamically fluctuate in response to network congestion and shared utilization. In the following, we discuss in more detail some of the main problems of Grid systems in respect to monitoring tools and then we propose an architecture infrastructure for monitoring applications in such systems.

### 4.1.1 Problems

From the point of view of the application performance, heterogeneity is the characteristic which most influences application execution. The literature has an extensive analysis of load balance problems that have been studied over a long period. The idea is that system heterogeneity is harmful for application developers [11].

In Computational Grid environments, system heterogeneity is a fact. There are some reports about homogeneous Grids in [73], although, with technological advances, and the multi-institutional characteristic of the Grid, it is hard to do system upgrades. Most Grids uses *commodity of the shelf* (COTS) machines as resources and COTS components today are not on sale due to the presence of newer, faster and cheaper products.

Despite system heterogeneity problems, another characteristic that affects application construction is the fact that Grid resources belong to more than one organization. Each organization has its own policies. In the case of security policies, the application should satisfy the requirements of identification, authorization of service usage and transport level operations. In the case of execution policies, problems of accounting and resource usage limits may affect application execution. At the fabric layer, for example, the policy applied to a user may command the operating system to restrict memory, disk quotas and other low level resources to the participant of some VO.

### Administrative Domains

Computational Grids differ from conventional distributed systems in their administration requirements. As mentioned before, Grids belong to more than one organization and each organization imposes its own runtime and execution policy. In such environments the application should satisfy all policies of the sites where it runs. These policies' appliances are handled by middleware. The middleware may apply security policies using PKI certificates or existing policy systems. At a fabric level, for example, the execution policies may be enforced by local schedulers such as Condor [25] or PBS [26]. Some resources can have a maximum execution time or network bandwidth restrictions.

From the point of view of monitoring and tuning Grid applications, the multi-institutional characteristic of the Grid has its own security requirements. In most cases, the communication required for the tuning process can be executed under a user identity. In this case, the generated performance data and tuning action messages' transport should use the user certificate to ensure authorization and private data transfer. The GSI mechanism, explained in Chapter 2, ensures that the tuning tools have the same security rights as the application. The tuning tool may use the proxy user certificate delegated by execution to secure communication channels.

## Heterogeneous Structure

In Grid applications, performance goals depend on many aspects. The application itself, at an algorithm level, may not scale well or have more sequential than parallel codes. In the case of parallel parts, the relation between computation and communication time may limit the application speedup [4].

One of the main differences between a cluster system and a Grid system is the network heterogeneity characteristic. In Cluster systems, there are many studies of load balance in order to seek high efficiency and lower execution times. With network heterogeneity, some models were proposed to have load balance on the Internet [71, 72].

When a tuning tool steers the execution of a parallel application over a heterogeneous structure, it consumes network and processing resources that in some cases may compete with the application itself. To overcome such scenarios, the tuning tool should balance its resource usage to lower network and processor utilization. The idea is the tuning tool should be parallelized itself and work in a distributed manner.

## Event Ordering

In cases of performance analysis of parallel applications, event ordering and time synchronization is crucial for bottleneck detection. Event ordering is an old problem for distributed systems, studied as a causal ordering issue [74]. In our case, if we want to measure the time a process spent in a blocked message passing receive, we need to correlate that receive to a message passing send on other processes. That can be done by event ordering over the sender and receiver processes. Such kinds of event ordering allows for a deeper analysis of such causal issues.

In MATE, covered in detail on section 4.2.3, the event ordering is done by event time ocurrence. The tool has a syncronization phase executed at the beginning of the tuning session and it generates the event timestamp based on the time offset among the machines. There is no re-synchronization phase for long-runing applications.

In Computational Grids, due to network hereogeity characteristics, the clock syncronization between machines is hard to maintain. We face problems in the tuning process due to incorrect event ordering. For example, for small communications, we

may receive the response event trace before the corresponding send. To help the tuning process, we need mechanisms to order events from different processes.

When state event ordering, the event collection can easily become the botleneck. If we receive an event caused by another event, we may wait for the cause event, blocking the event stream. To overcome such scenarios, the tuning tool should treat the event stream and event processing asynchronously.

### 4.1.2 Clock Synchronization

If monitoring is used for performance analysis, performance analysts may want to correlate events from different process within an application execution. That analysis is based on time differences between events from different processes on different machines with their own clock. In order to provide event synchronization in application monitoring processes, we can use two approaches:

- Based on the tool;
- Based on the system.

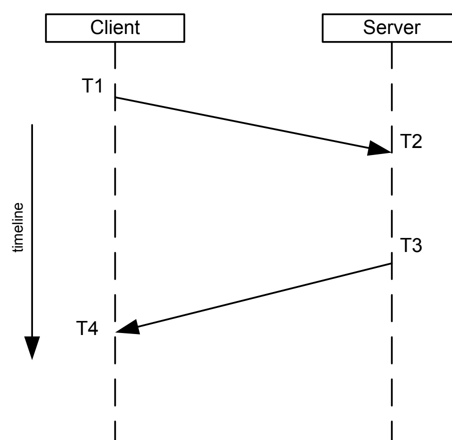



Figure 29 – Clock synchronization message exchange.

In clock synchronization based on the tool, the synchronization process is done by the estimation of communication messages' roundtrip time plus clock difference. Two machines can exchange timestamp messages and estimate the clock difference between then. A simple algorithm is presented in Figure 29 [75]. The client calculates the

roundtrip delay *d* and local clock offset *t* relative to the server process, by sending a message with client timestamp clock information T1. The server receives the message on server time T2 and generates a reply message containing the T1 and T2 values and then sends the message timestamp T3. The client receives the message at time T4 and calculates the clock difference. Table 2 presents these timestamps descriptions and equation (8) relates them to obtain the delay and offset.

$$d = (T4 - T1) - (T2 - T3) \qquad t = \frac{((T2 - T1) + (T3 - T4))}{2} \tag{8}$$

Table 2 – Clock synchronization timestamp information [75].

| ID | Timestamp Name | When Generated |
|---|---|---|
| T1 | Originate Timestamp | Time request sent by client |
| T2 | Receive Timestamp | Time request received by server |
| T3 | Transmit Timestamp | Time reply send by server |
| T4 | Destination Timestamp | Time reply received by client |

The client can repeat the transmission many times in order to get the average roundtrip time. Note that there are some drawbacks for this operation in Grid environments. In these systems, network latency can have a high variance because it is a shared resource and the hop[5] path may change, due to the network load balance route schemes. In some senses, machines distributed over the WAN should use some third part synchronization source to overcome network latency variation problems. Another point is that machine clocks are not synchronized. A simple measurement from different between and Internet exposed server and two stratum 2 servers is presented in Figure 30. Note that the clock difference among the machines varies over time. Within same day we have variations

The term 'hop' uses the same concept as TCP/IP networks. It consists of each network element having the information it has to cross to reach the destination.

from +115ms to -110ms to Site A and Site B have maximum of 32ms and minimum of -33ms.

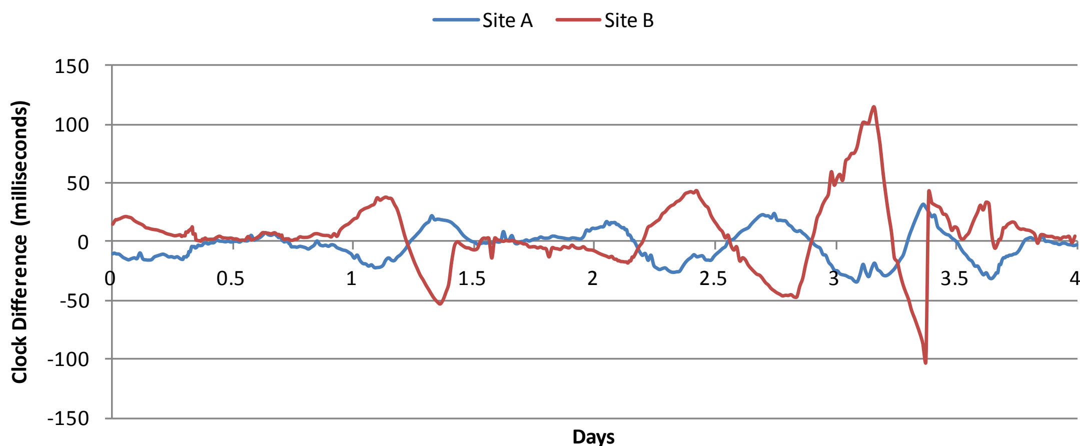



Figure 30 – Time difference between two stratum 2 sites to a client machine.

Note that the presented time variation graphed on Figure 30 uses the same site time offset to synchronize the machine clocks. We state that such variation should not have high influence on the performance models used in tuning. If the application needs more time synchronism than 100ms, it should rely on causal relation among the instrumented events. For a Master-Worker application, for example, it is possible to inspect the time difference from the master to worker processes by analysis of the sampled events of task sending by master and receiving by worker and result sending by worker and receiving by master. That could also be applied in small message communication in any communication paradigm.

The idea of using a third party machine as a time synchronization source is handled by a system synchronization approach. In this approach, the machine should synchronize its clock using a trusted and precision time-source. The service infrastructure for time synchronization is standard and its services are available on the Internet, called Network Time Protocol (NTP) servers. By now, most operating systems come with a NTP client. By using a close NTP server (e.g., one with low latency or controlled latency between server and client), clients distributed over the world should be synchronized.

The idea is that the NTP servers should be synchronized by high precision time-source hardware. Other NTP servers may synchronize with those servers in a hierarchical structure. The distance from machines running the NTP time daemon to an external source of Coordinated Universal Time (UTC) is called a "stratum". Stratum 1 servers can connect directly to an external source of UTC, such as a radio clock synchronized to a standard time signal broadcast. In general, the stratum value determines the number of hops, minus one from a stratum 1 server [75].

To overcome the clock speed difference, the NTP client can detect and make the correction by software, inside the OS. The NTP daemon that comes in most LINUX distributions uses the kernel system call *adjtime* to configure such tune parameters. Currently, it is easy to find stratum 2 or 3 servers to be synchronized with. That gives clock synchronization with an error of less than 50ms in our test environment, as described in section 5.2.2 (we used the stratum 2 servers with less variability). Note that the machines inside a cluster have their clocks synchronized with an error less than 10ms, enough for event ordering and synchronization. Measurement of events using such time resolution modifies the application behavior caused by overhead effects and so its performance analysis.

## 4.2 Dynamic Tuning

With different system configurations, the application should have different performance indexes. When these systems are used in shared mode, applications perceive more system heterogeneity. In such scenarios, applications may suffer performance problems due the difficulty in adapting to the different system characteristics. The dynamic tuning technique can help the application adaptation to overcome these problems. In the following sections we analyze some state of the art tools related to dynamic tuning.

### 4.2.1 **Active Harmony**

Active Harmony consists of a software library that developers can use to prepare an application for adaptation in different systems. That adaptation is done by automatic parameter search and evaluation. The programmer uses the library to expose some tuning factors and runtime metrics. During application execution, Active Harmony explores the tuning factor parameter variations and verifies the gains in runtime metrics.

The current version supports two types of parameter search: exhaustive search and a Nelder-Mead simplex algorithm. [15]

To deal with parallel applications, Active Harmony has a Client/Server model. The client consists of the linked library in application binary. Within a server, the developer may specify local and global variables. These variables are updated explicitly by API calls.

The difference between Active Harmony and our work is the philosophy that drives the parameter search. We assume that complex systems with high levels of heterogeneity may generate a huge parameter search space. In such scenarios, a parameter search guided by performance models should have faster responses and should avoid local sub-optimal configurations.

### 4.2.2 Autopilot

Autopilot [76, 77] is a software toolkit used for the performance monitoring and analysis of distributed applications and infrastructure data. The instrumentation is done by the developer in source code and the performance data collection is selective by means of a pattern classification scheme. The goal of the tool is to drive application execution based on sensor information data and decision procedures. The main components are sensors, decision procedures and policy actuators, as presented on Figure 31.

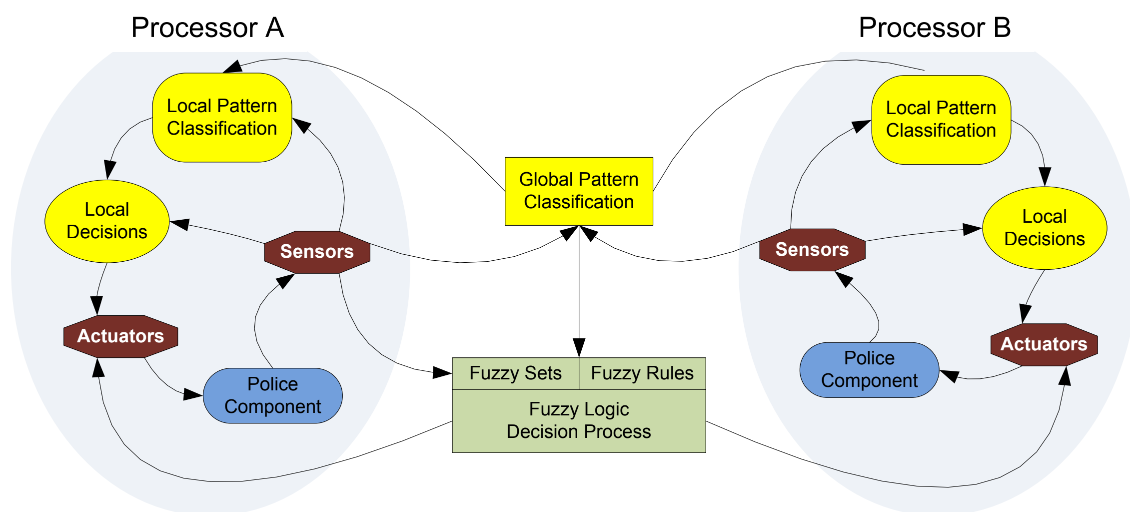



Figure 31 – Autopilot conceptual architecture [76]

In contrast with the GMA model, detailed in section 2.5.1, the registry role is done by the Autopilot manager, acting as a name server. The producer role is played by application processes instrumented with Autopilot sensors and the consumer role is played by the clients. In Autopilot, sensors have a set of associated properties defined at the moment of sensor creation. These properties include sensor name, type, identifier, IP address and custom user-defined pairs of attribute/value. Clients use properties to query the Autopilot manager for distributed sensors over the system.

The sensors are associated with a process and a set of system variables used as a data source. The sensors can be used in two programming models: threaded or non-threaded. When a threaded sensor is activated by a client, it starts to monitor periodically and transmit to the client. The non-threaded usage of sensors depends on an application's explicit function calls to perform data delivery. Sensors uses a NEXUS component of Globus to send the collected data to clients [76, 77].

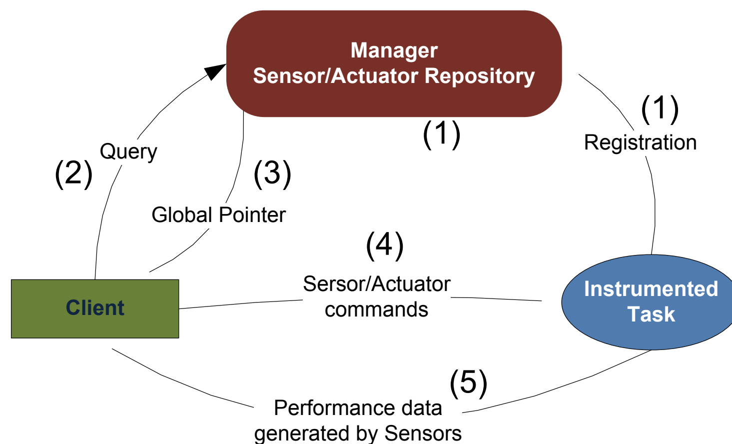



Figure 32 – Autopilot components and the iteration sequence among them. [76]

To allow a dynamic change of variables, Autopilot provides software components called actuators. Clients can interact with actuators in order to change variables or to invoke application level functions. Actuators, similar to sensors, can be located by querying the Autopilot manager. Figure 32 present the iteration among these components. Instrumented tasks when running register themselves in the Autopilot manager. The client task, which performs the global pattern classification, connects to

the Autopilot manager to query what instrumented tasks are active. It receives a global pointer from where it is possible to command sensors and actuators coded on the instrumented tasks. When activated, sensors generate performance data that is used by a task client for performance analysis. The analysis is done by continuous evaluation of fuzzy logic rules which drive what commands should be performed by actuators on instrumented tasks.

### 4.2.3 MATE

The Monitoring, Analysis and Tuning Environment (MATE) [14, 57] consists of an execution environment that permits dynamic tuning of applications without the need for code modification, compilation or linkage. The environment is based on DyninstAPI [48]. The idea is that the developer does not need to be an expert to tune her application. Internally, the tool has knowledge of the performance bottlenecks problems, and how to detect and solve them. With this information, MATE introduces instrumentations and modifications in the application binary to optimize its execution. That optimization process is done without user interaction.

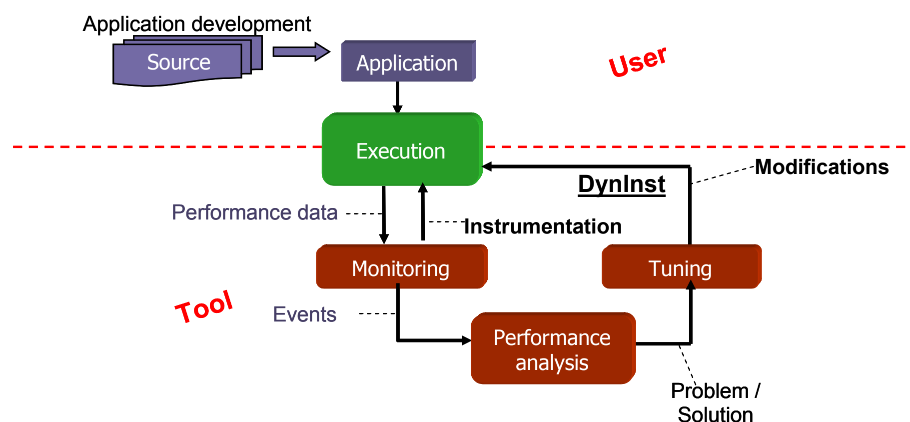



Figure 33 – Dynamic Monitoring, Analysis and Tuning Approach

Figure 33 shows the work done by the user and the tool. MATE inserts the instrumentation needed by the performance analysis within the running application. The instrumentation inserted by the monitoring process generates performance data that is collected and represented by trace events. These event traces are analyzed using some

performance models to verify the existence of bottlenecks. For example, the instrumentation can measure the size of messages sent and received, and buffer sizes: a performance model could relate message sizes to optimal buffer size and the modification could be a change of the buffer size in process binary.

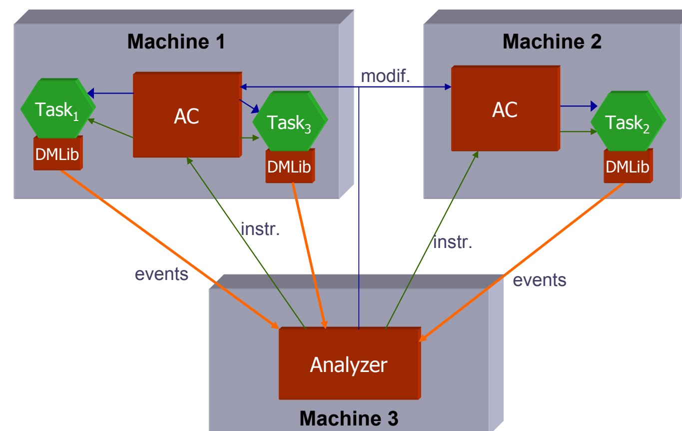



Figure 34 – MATE component architecture.

As presented in Figure 34, MATE has two main components: (i) the Application Controller (AC), and (ii) the Analyzer. The AC is the component which interacts with the application process, inserting instrumentation code and doing dynamic tuning modifications [14, 57]. The AC uses DyninstAPI to attach to the application process and load within the application process space in a MATE dynamic library called *DMLib*. This library helps the instrumentation and tuning process. In the current architecture, it is required to run just one AC process instance per processor node. One AC process can monitor and tune many application job processes on the same machine. The Analyzer consists of a software container of tuning components called tunlets. It has the responsibility of coordinating the tune session in cooperation with the AC components distributed over the system. Each tunlet can encapsulate the logic of what should be measured, how data can be interpreted by a performance model and what can be changed to achieve a better execution time or better resource utilization [14, 57].

The Analyzer interacts with the tunlets through the Dynamic Tunlet Application Programming Interface (DTAPI) as presented in Figure 35. That API allows for the tunlet receiving callback events such as application startup, job process startup and the

event trace generated by instrumentation. Within these callback events, the tunlet can request application process instrumentation or application modification. All instrumentation requests generated by the tunlets are forwarded to the AC. The AC receives the requests, instruments the application and forwards the trace back to the Analyzer. By DTAPI, these trace events are dispatched to the desired tunlets. The tunlet decides, based on its performance model, what should be changed to tune the application and requests the Analyzer application to make changes. These requests are forwarded to the AC and the appropriate changes are made in the application [14, 57]. Some cases of MATE are presented in [66, 78].

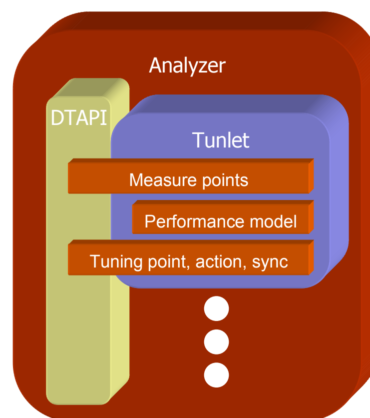



Figure 35 – Internal representation of the Analyzer.

## 4.3 Grid Monitoring

Once we saw the state of the art in automatic performance analysis and tuning, we detailed our architecture for its appliance on computational Grids. First we started the monitoring stage of the automatic performance tuning. Tuning requires sense application behavior through measurements obtained by monitoring. Monitoring parallel applications in computational Grids using a dynamic instrumentation approach needs some requirements to work well. The first requirement is a technique to insert the instrumentation into the running process. We presented some alternatives for dynamic instrumentation in section 2.5. In our tool, the dynamic instrumentation is done by a DyninstAPI library. By using DyninstAPI concepts [48], we treat application processes

as *mutatees* and one of our processes as the *mutator*. Using this scheme, we need a process running on each machine where the application processes run.

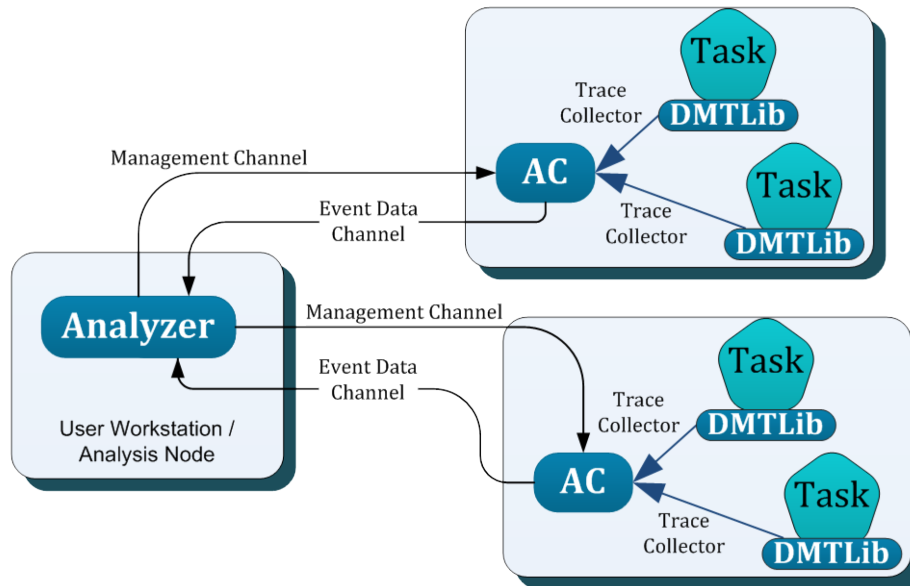



Figure 36 – Communication channels among GMATE components.

The process which has the responsibility for controlling the application process is called, in our architecture, an Application Controller (AC). The AC component is the processes which uses DyninstAPI for the instrumentation of application processes. As the execution of AC and application job processes are done on different machines that are used to run the program which consumes the produced event data, we have the client component. In our architecture, the client is the Analyzer component. The client process establishes communication channels to the AC and its client. The communication among presented components is done by three communication channels, as presented in Figure 36:

- Management Channel: used to transport management commands between the AC and the Analyzer. For this channel the AC can be controlled for instrumentation of the application by installing process sensors, to continue or pause application executions and establish new management and event channels.

- Event Channel: used just for transfer data collected from the sensors to the Analyzer. Currently it is used as an upstream data transfer channel.

- Trace Collector: the communication scheme for collecting information from the running process. This channel is distinct from the Event Channel because overheads generated due to the data transmission should be minimal. That is discussed in the section on monitoring topology below.

To monitor an application execution in a Grid environment, we need to use the services provided by the middleware infrastructures in order to comply with site usage and security policies. In a Grid environment, an AC should fulfill some requirements such as:

- *Application Process Tracking*: depending on the software layers installed over the Grid, process tracking may not be a simple issue. The location of process execution may be known only during execution. We propose two main approaches to solve this problem [79], a System Service Approach and a Binary Service Approach. Both topics are covered later on in this chapter.

- *Locate the Analyzer*: once the application process is located, the next step is to establish communications between the AC and the Analyzer. This can be done by using Grid Information Services such as MDS or can be passed as a configuration parameter to the AC process. In order to fulfill Grid security requirements, the message exchange among tool components should use the middleware infrastructure to ensure data privacy [79].

### 4.3.1 **Design Architecture**

Our monitoring tool has the same components presented in MATE [14, 57]. The coarser components are the Application Controller (AC), the Analyzer and the Dynamic Monitoring and Tuning Library (DMTLib). The Analyzer corresponds to the Analyzer component in MATE. As presented before, an AC component controls the application using the DyninstAPI library [48]. This library facilitates the modification of binary applications without the need for source codes. Using that library, the AC process is capable of dynamically changing the application job process. These modifications are

function calls to the functions from the dynamic loaded library in order to allow communication between the running application program and the AC.

Our tool follows the GMA model, as presented in Figure 37. We use the concept of a sensor to collect event data from the application. Sensors are objects which have a internal memory state and a defined protocol for communication. The AC installs sensors into the application job process by inserting a sequence of function calls on an execution point of the application. That is done using DyninstAPI. By doing that, when the application execution reaches the changed execution point, the sensor receives the information and acts in response to it. A more detailed form of sensor structure and internal processing is presented later in this chapter.

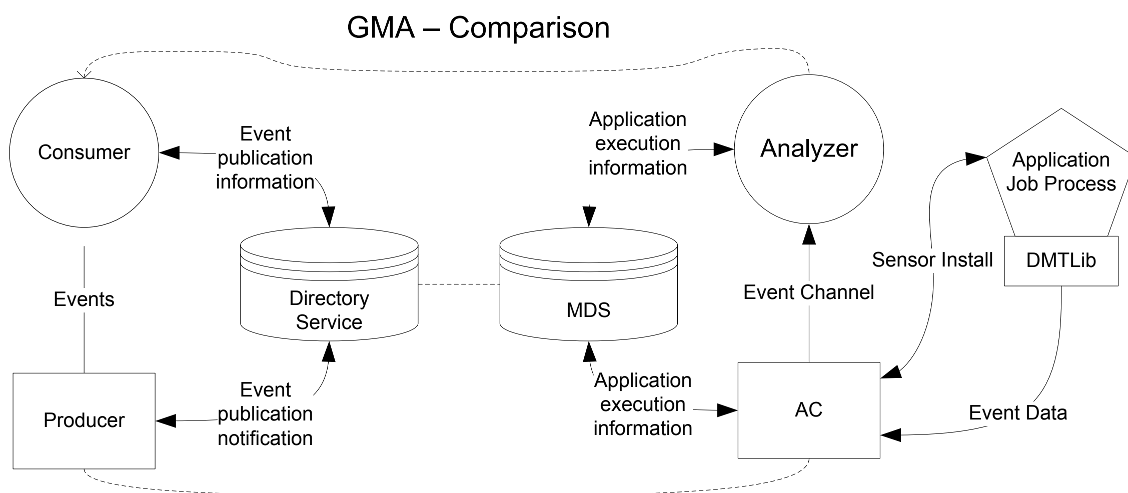



Figure 37 – Our monitoring scheme model in comparison to GMA

All the sensor logic is implemented by a dynamic linked library called DMTLib. When the AC starts to control the application job process, the first step is to load the DMTLib inside the process memory space. DMTLib establishes the communication protocol between the running process and installed sensors by exposing general API functions. The instrumentation is done by inserting function calls using DyninstAPI in the selected execution points. These function calls are implemented by DMTLib binary and allow the instrumented execution points to activate the sensor components. Table 3 presents

the API available and used within the instrumentation points and details its functionality description.

The instrumented API calls can generate trace data which are transmitted in binary form to the AC. When the AC receives the data, it decides, based on configuration, to store locally or to forward it in the Event Data Channels. Moreover, to allow analysis of the complete application execution behavior, we merge the collected event trace data. That merge process is done using the event timestamp information.

Table 3 – API used in process to gather data and support sensor communication.

| API Function Name | Description |
|---|---|
| DMTLib_addSensor (SensorConfig) | Allocates the sensor inside process memory space. The sensor configuration data contains all the information necessary to build the sensor. |
| DMTLib_reset (EventId) | Informs the sensor about the beginning of data transfer. When the sensor receives this call, it waits for a sequence of DMTLib_setProperty function calls. |
| DMTLib_trigger (EventId) | Informs the sensor about the end of data transfer. In this point the sensor can send the trace data event to the AC or can do other computation. |
| DMTLib_setProperty (SensorId, propertyData) | Used to send runtime measurement data value to the sensor. |
| DMTLib_sendValue (FromSensor, Index, ToSensor, Indwx) | Allows instrumentation of value exchange between different sensors. Realizes the inter sensor communication protocol. |
| DMTLib_bindSensors (FromSensor, ToSensor) | Configure FromSensor to propagate trigger event to sensor ToSensor. Allows trigger event subscription between sensors. Realizes the inter sensor communication protocol. |

Each application job process produces one set of events. On our architecture, the generated events can be merged *offline* or *online*. In *online mode*, the AC and the Analyzer establishes one Event Channel based on the Grid transport services, the

Globus Extensible Input Output System or Globus-XIO [80]. These services are accessed through a common API for data transport. In such an API, it is possible to configure a stack of protocols and use it in a transparent manner. In such configurations, the Analyzer can receive all application event information while the event occurs. In *offline mode*, the Analyzer configures the AC for storing on disk the event trace data and the generated file is transmitted at the end of an application execution.

As we saw in chapter 3, in order to improve performance, the application processes should be mapped to the selected execution nodes to overcome network bottlenecks. The application can have a process group which has more inter-process communication than another. In that case, that process group should be placed on a machine group with high bandwidth and a low latency network such as a cluster. Other processes that do not need much communication can be placed on 'far' machines over the Grid, with the overhead of high latency and low bandwidth network. The monitoring tool should be able to overcome the problem of different network properties such as latency and bandwidth by a change in its topology of communication. If the event collection generates more events that the available bandwidth of the network between the AC and an Analyzer, the monitoring process slows down the application execution. To overcome that problem, our monitoring tool provides a mechanism for the use of different network paths. The monitoring topology is covered later in this chapter in section 4.3.3.

The communication between the AC and the Analyzer is achieved by two channels based on a Grid middleware infrastructure, Management Channels and Event Channels. The Management channel is used to send control commands from the Analyzer to the AC and to send application notification status changes from the AC to the Analyzer. We distinguish the following control commands transferred in management channels:

- Application execution mode change: allows start, stop, continue and terminate the application job processes.
- Sensor configuration: allows sensor install, remove, enable and disable.
- Topology construction: allows the creation of new management or event channels to other destinations. This allows the construction of any topology over the Grid.

The Event Channels are used to transfer event data from the AC and the Analyzer. These Globus-XIO channels connect the two components and the AC has the capability to receive connections from other AC's and forward event data to other established event channels. In order to lower the network bandwidth used for event transmission, the event data transmitted over event channels should be minimal or encoded to be small. The concept is that on each endpoint of an Event Channel we have the same configured sensors. The sensors configured on an Analyzer are used to decode the information received by the Event Channel. The topology of Event Channel connections can be changed using control commands and may differ from the topology interconnection used for management.

### 4.3.2 **Process Tracking**

Due to the distribution of resources within the Grid, following the process execution may not be an easy task. The Compute Elements on the Grid may be controlled by cluster local schedulers and the job distribution may be done using broker services. The resource allocation for the application should fit application requirements. These requirements may not fix the execution machine of the application. The application has the information about the allocated resources on execution time.

One execution scenario that presents this kind of problem may be a Grid environment which uses Condor-G [18] as Grid-wide job scheduler. In such environments, the job description specified on the submit command file should not force the target resource to take advantage of the Grid capabilities. In this case, Condor-G will search for the available resources which satisfy the application requirements, such as operating system type, free memory and disk amount.

In order to track application job processes over the Grid, we need to get information about the process startup on the resources. We assume a Globus Toolkit as the middleware which provides the software layers required for Grid construction. On this middleware, the information about where the processes are executing is not available on MDS. Once the application process is started, we need to have our monitoring process running on the allocated CN resource in order to instrument and monitor it. If the execution information were available on Grid information services such MDS [9], we also could not run our monitoring process on the allocated node using the current

scheme available for job submission because the job submission process does the resources allocation in exclusive mode.

Another problem is the execution of SPMD applications. The submission of Grid MPI applications may distribute only the application binary. This occurs, for example, on simple submissions using a MPICH-G2 *'mpirun'* command [70]. The *'mpirun'* command in MPICH-G2 generates the job submission file and binary transmission in a transparent way. In that case, we need to be able to follow and control the process execution in a transparent way.

The lowest software layer, where the application jobs execute, is the operating system. In order to support any software layers that exist between the user and the machine, we found two approaches which can be used to steer the application execution. The first is when the operating system starts the application in response to a high level request and the other is the application execution itself. We can, using these events, plug in a daemon to the operating system, in order to detect the application startup, or we can put a code in the application and fake the submission system, identifying the execution resources. These two different approaches are called *System Service* Approach and *Binary Packaging* Approach respectively [79].

## System Service Approach

The Grid infrastructure evolution indicates the emergence of Service Oriented Architecture (SOA) semantics in Grid computing. In such semantics, the system infrastructure and application components should be accessed through services. The Globus Toolkit follows the SOA philosophy and its services interfaces are now, in version 4 of Globus Toolkit, accessed through the Web Services Resources Framework (WSRF) as we discussed in chapter 2. In a simple manner, the Grid services are provided by resources and the access to these resources is achieved by web service calls conforming to WSRF standards.

In this approach, the concept is to have monitoring and tuning (application change) services pre-installed on machines in the Grid, similar to SCALEA-G tools [31, 35]. This mainly consists of having the AC as a system daemon running on the processor nodes waiting for monitoring/tuning sessions. This approach requires administration privileges because it must be capable of changing its security context to instrument

processes from different users. The key idea is to enable a machine with dynamic tuning services that can be used by any registered application.

With the AC service daemon running on each machine of a cluster, this cluster is ready to undertake the monitoring/tuning sessions by user request. In clusters exposed as a CE, it is common to have Globus services installed only on the machine that controls the cluster. The integration is done by WS built on top of WSRF that handle the control of the AC daemons running on CN elements of the CE. Those services expose two resources: the AC Wrapper (ACWS) and Application Session. The AC Wrapper handles the integration through the Globus Index Service and exposes the services of each AC. Because the ACWS is an exposed grid resource capable of indexing, the resource properties are indexed by the MDS. The MDS propagates the collected information to other configured MDS services that belong to the VO information hierarchy. Figure 38 presents the relation of the AC Wrapper and the AC instances. The integration actions to the Grid middleware are presented in Figure 39 and Figure 40.

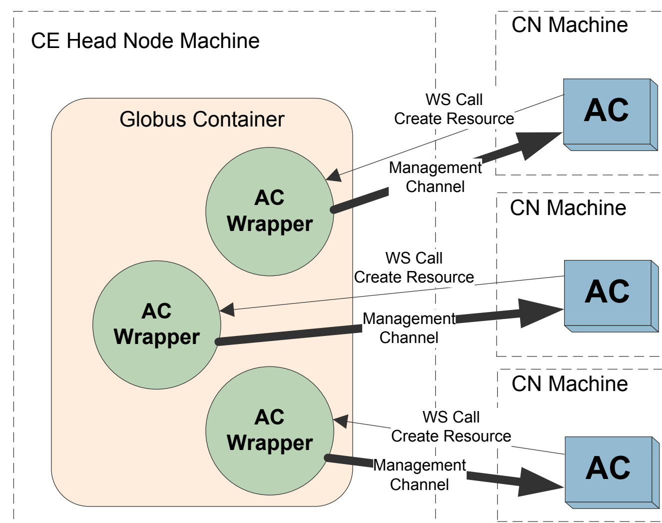
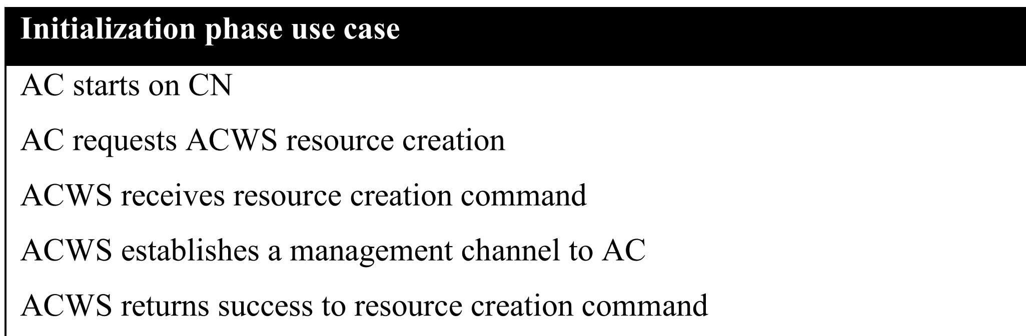



Figure 38 – Connection between AC Wrapper instances and AC instances.

When the AC daemon starts, it creates an AC Wrapper instance inside the Globus container. In this process, the AC Wrapper establishes a management channel to the AC daemon. After that, all the communication between the AC Wrapper and the AC is done using the established management channel. After initialization, the AC Wrapper

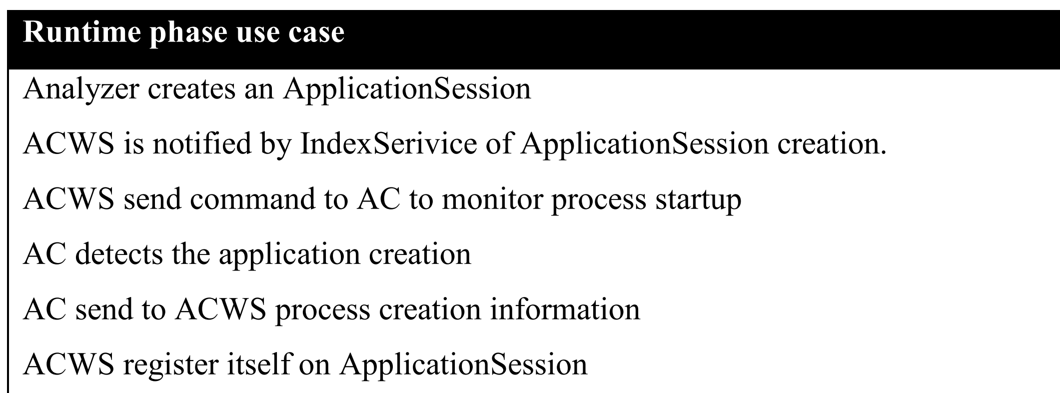
subscribes modifications to the '*GMATE/ApplicationSession*' resource group in MDS Index Service by using the Notification Service provided by the WSRF.

When a user starts a monitoring session, he/she creates an ApplicationSession resource containing information about the application submission. When the application is registered as an ApplicationSession resource, the associated information is collected by MDS and the AC Wrapper service receives the notification of the resource creation.



| **Initialization phase use case** |
|---|
| AC starts on CN |
| AC requests ACWS resource creation |
| ACWS receives resource creation command |
| ACWS establishes a management channel to AC |
| ACWS returns success to resource creation command |

Figure 39 – Integration of the AC to Globus Toolkit – Initialization Phase.



| **Runtime phase use case** |
|---|
| Analyzer creates an ApplicationSession |
| ACWS is notified by IndexSerivice of ApplicationSession creation. |
| ACWS send command to AC to monitor process startup |
| AC detects the application creation |
| AC send to ACWS process creation information |
| ACWS register itself on ApplicationSession |

Figure 40 - Integration of the AC to Globus Toolkit – Runtime Phase.

The AC Wrapper uses the ApplicationSession services to get the application information for local detection and transmits it to the AC daemon. In response to that, the AC daemon starts a time-limited detection of application startup. In case of success, it transmits this information to the AC Wrapper and the AC Wrapper resource registers itself in the ApplicationResource. The sequenced use case of the initialization phase and the runtime phase are provided in Figure 39 and Figure 40 respectively. The connection

between the AC daemon and the AC Wrapper service runs under the security context of Host Identity, so it requires the running machine to have a valid host PKI certificate.

To start application monitoring, the Analyzer should register the application by creating a resource ApplicationSession. The resource information is collected by MDS. The resource ApplicationSession represents an application execution session that should be monitored. This component exists to provide to running AC processes the information necessary to start the monitoring session. This information includes:

- Application program name: used for application identification;
- Grid contact endpoint: used on AC management channels creation;
- Startup detection time: used by AC services for timeout detection;
- Monitoring control information: used by AC to choose the authentication method of the application;
- Environment Variable Detection: includes a name/value pair used for detection;
- Binary Detection: includes the application binary size and MD5 hash number of the application program used for application authentication.

The main concept is that when the application is registered, all the AC services available on the Grid will be made aware of the application registration by the notification service. At this phase, all AC services start to monitor their local machine scheduler in order to find some process name that matches the registered information. If startup detection times exceed the value provided on MDS, the AC stops searching for application detection and assumes that the application programs were detected on other machines. That corresponds to the steps 1, 2 and 3 on Figure 41.

The local process detection by AC daemon can operate in two modes: *pooling mode* or *pull mode*. In pooling mode, it monitors changes in '/proc' file systems to detect application process startups. In pull mode, the AC service instruments the cluster batch scheduler process such as PBS or Condor daemons with DyninstAPI and waits for the callback event generated by an exec system call [48]. This makes known the exact time of application startup and allows the instrumentation from the beginning of the application.

With *pooling mode* detection, when the operation system starts a new process, the '/proc' file system changes and the AC service reads its content looking for the

registered application program name received by the Grid integration. In *pull mode* detection, the AC service checks for process startup on each callback generated by the instrumentation of the batch scheduler instrumentation. In both modes, if it finds a process name which matches the registered application program name, the AC service authenticates the application. The authentication depends on the monitoring control information provided in the MDS. On that information, the Analyzer specifies how the authentication can be done: by application environment variable detection or binary detection. Figure 41 presents the step sequence of application processes detection using the Grid Services.

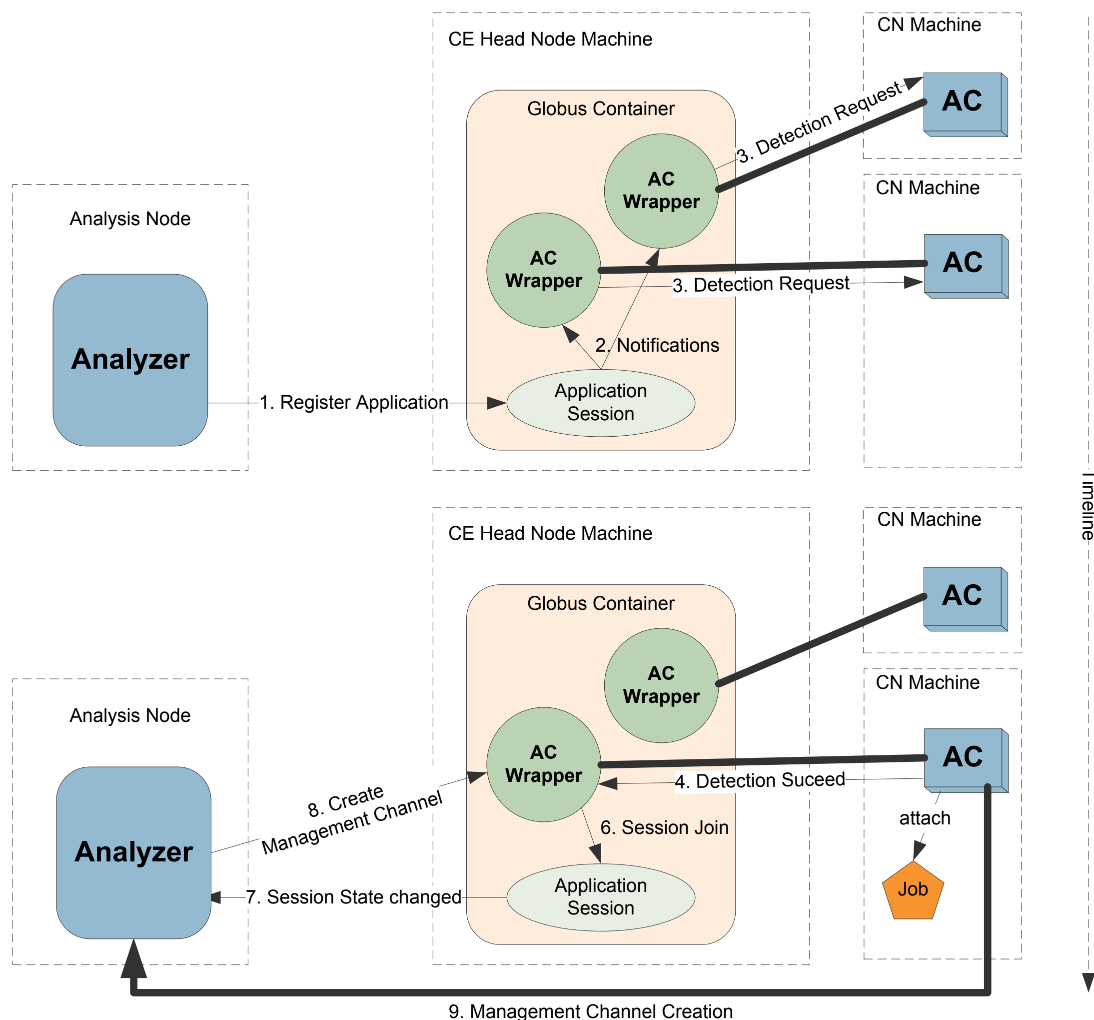



Figure 41 – Application detection using Grid integration.

The authentication can be done using application processes environment variables. The AC daemon looks for known environment variables registered in the application session information. If the process has the environment variable, the AC service assumes the process is authentic and starts the monitoring/tuning session.

In cases of an authentication based on binary detection, the AC service checks for the binary size matches to be provided on registration. If it matches, the AC service attaches to the found process using DyninstAPI and perform a MD5 hash on a process binary file in order to complete the authentication. Using this approach, we stop only programs that have the same name as the registered application process and authenticate the binary using the MD5 hash. In this scenario, we suppose the application program binary is unique for that execution.

To use the AC service configured to *'pull mode'* detection, it is necessary to having high levels of privileges in order to allow the attach operation to the batch scheduler process and follow the batch scheduler process to a user's security identity. In this mode, we performed tests over the OpenPBS [5] as proof of the concept. In each of the presented models of execution, the target execution machine should support DyninstAPI [7].

## Binary Packaging Approach

The idea of an application plug-in approach is to track down application processes using the same binary distribution and execution used by the application. This approach has as advantages over the System Service Approach because the user has total control over the software requirements of the tracking process. This approach allows transparent execution, for example, of MPICH-G2 compiled SPMD programs over Grid Systems using *'mpirun'* submission script. The tool is packaged inside application binary.

When a user submits an MPICH-G2-based application for execution on a Grid System using the *'mpirun'* submission script it generates the resource specification needed and lookup on Globus MDS in order to locate the hosts. If the lookup finds the required host resources, the script uses the GASS to deliver the application binaries to selected hosts. In sequence, the script uses the DUROC services [81] of Globus to coordinate the application process startup and authentication, using GRAM services.

In such scenarios, the application binary is the startup entry point. To follow binary delivery and have total control of the application process startup we cheat the execution system. The problem of process tracking in such scenarios can be broken in two:

- Follow the application binary;
- Control the process startup entry point.

In submissions using Condor tools or even Globus *globusrun* submission commands, it is possible to specify which files should be delivered and which application is the execution entry point. In cases of *'mpirun'* use, the user can not specify the parameters. In such scenarios, the process tracking should fake the binary delivery mechanism and the application startup to get the monitoring tool binary transmitted to selected machines and to get the AC program started as a first application.

To get the AC program delivered together with the application process binary, we merge both binaries, the AC program, the application program and all configuration data needed for the monitoring process. This process is called Binary Packaging and, as a result, we have a single executable file containing:

- Glue program: a process that knows how to unpack the programs and libraries;
- AC program: our dynamic monitoring module which controls the application;
- DMTLib: our dynamic linked library which is loaded into application process space to help process instrumentation and event trace collection;
- DyninstAPI Libraries: optional libraries that can be used in case the target execution machines do not have DyninstAPI installed.

That binary packaging process can be done by the developer or even by the application users before the execution as a preparation step for execution or a post compilation step. That packs all required files into one using the structure presented in Figure 42. Using this approach, the AC program manages to follow the application program on any machine selected for execution over the Grid.

For controlling the application startup entry point, we need to change the application startup process. The packaging process puts the Glue program as the first program in the composed binary. By that composition, at runtime, the first executed code is the one that

is loaded on memory. The key idea is to have the Glue program as an independent program which should care about the application startup process.

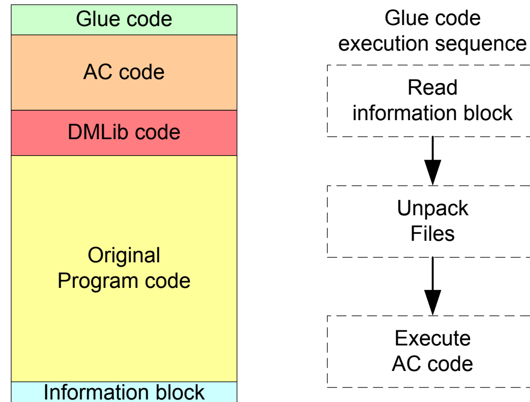



Figure 42 – Binary packaging structure.

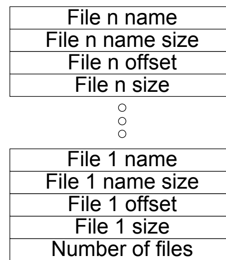



Figure 43 – Binary Packaging information block

In the startup phase, the Glue program locates the application and AC programs within the composed binary, extracts the packaged files, and does some checkups, initializations and passes execution control to AC programs. To extract the packaged files, the Glue program locates at the end of its execution some information included in the packaging process. This information block, called a “Glue record”, contains the number of packaged files, and for each file, an information block containing the name, offset and size of the packaged file as presented in Figure 43. This information is included at the end of the binary in an inverse order to allow easy location of each information block.

After the file extraction, the Glue code tries to detect if the machine has DyninstAPI installed by checking the environment variables and trying to load the DyninstAPI runtime library. If it cannot find the libraries or the library load fails, and the binary packaging has the DyninstAPI binaries included, the Glue program reconfigures the environment variables and tries to load the library again. If it succeeds, the Glue program starts the AC program by passing the name of the extracted application program and the received command line parameters. If the Glue program cannot start DyninstAPI, the instrumentation of the process is not possible on the running machine, so, the application process is started using the command line parameters and the instrumentation of the whole application will became partial. In a partial instrumentation, some processes do not generate monitoring information. The Glue program acts as a wrapper process to an AC execution configured to process startup using the runtime properties of target system.

In cases where we have a machine which supports dynamic instrumentation, when AC executes, it starts the application as a controlled process using the DyninstAPI library services and starts the monitoring process described above.

Using the Binary Packaging approach, our tool has total control of the application execution. It enables applications for monitoring in scenarios in which application program submission for execution cannot be detailed as scheduler submission scripts such as a Condor submission script or *globusrun* RSL files. Generally, in scheduler submission scripts such as in Condor submission scripts and RSL files, the user can specify which binaries should be transferred to the CN before execution. In that scenario, the packaging is not necessary because the user can configure the script to send all required files to the CN and configure the script to have the AC program as the startup program.

### 4.3.3 **Monitoring Topology**

The capability of running wide applications is a well known Grid goal. With resource sharing, new computational challenges can be addressed by the available resources. The application should scale in order to cover these challenges. On the other hand, the system communication configuration is not homogeneous. Grid systems may have multiple networks interconnected by different technologies and different characteristics of latency and bandwidth. Some networks are dedicated; others are shared and chaotic

such as the Internet. Grid systems generally are geographically distributed and, due to available interconnection network communications, can easily become the application bottleneck. Some example scenarios can be found in [71, 72]. The network throughput determines the amount of data that some sites can produce. In scenarios with network congestion, different topology can take advantage of network its utilization within the application mapping to avoid congestion hostpots.

Nevertheless, developers construct applications communications by clustering processes that have communication within domains with better network services. One strategy is to maximize the data locality in order to minimize the communication. If the communication is necessary, the processes should be mapped on CEs with good interconnection network properties. The same strategy may be used in monitoring tools. In order to handle the problem of trace event data transmission, monitoring/tuning tools should be able to reconfigure the logic communication topology. Each machine that has an application process job with instrumentation that generates a stream of event trace data during job execution. If the monitoring process is used for a postmortem analysis, for example, the event data can be stored on machines until the execution end. That allows, for example, the possibility of good compression of the produced data and it is the solution that has the best lower bandwidth requirements.

In cases of online utilization of the produced event trace data, the monitoring tool should transfer the event data stream to a front end tool or other online analysis tool. The event data stream may be configured by the user to be merged online and written on a stable Grid storage service. The event trace data can be analyzed as a soft real time system. The monitoring client should have specific time limits which determine if an event is useful or not. The worst case is the reception of events on the application execution end. The best case is determined by the network latency between the client and the monitored application process.

Both ACs and Analyzer components of GMATE are capable of input and output Event Channels. The user can configure different topologies and the tunlet have control to establish new channels and close old ones within an application tuning session.

## Event Routing

The event information data is small compared to other structure types such as an IP packet structure. One very simple event trace containing timestamp values and event identification fits on 16 bytes. If the event contains more than 4 function parameters of integer types, it fits on 28 bytes. In our architecture, we support events of different sizes and the events may be generated by different sensors. Figure 44 presents our complete event structure representation. The header used for routing have two integers (64 bits). The event header consists of one sensor identification (16 bits), a packet size (32 bits) and event identification (16 bits). The event structure may have unlimited data items which follows the event header and it is coded and decoded by sensor components. The monitoring tool uses this binary representation form of events in transmission to cover the lower bandwidth requirements in comparison with text-based transmissions.

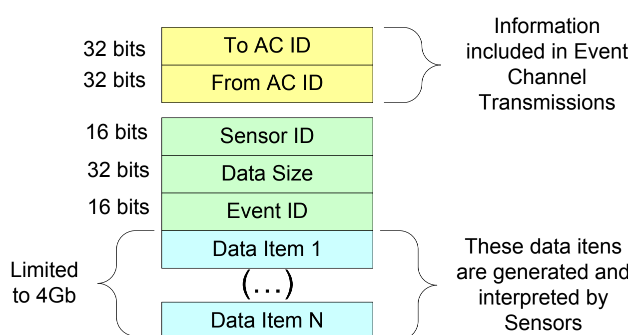



Figure 44 – Event structure representation.

If the monitoring tool sends the event at the exact time of the generation, each network packet will contain only one event. In that case, the transmission consumes more bandwidth due to the relation between the event data and network address information. On TCP/IP networks, the packet header that handles IP and TCP addressing data is at least 32 bytes. On 28 bytes event traces, 32 bytes of header corresponds to approximately 79% of overhead per packet. To overcome that problem, monitoring tools should relax the real-time properties and aggregate events when it is possible in order to utilize the network Maximum Transfer Unit (MTU). On networks with a MTU of 1500 bytes such as the Ethernet, the cost of sending a 32 bytes event trace is the same

as sending 46 events of 32 bytes each. A TCP protocol uses the Nagle's algorithm to handle this problem. The time limit in Nagle's algorithm is 200ms [82]. If the monitoring client support more than that time limit, more events could be grouped in one transmission packet.

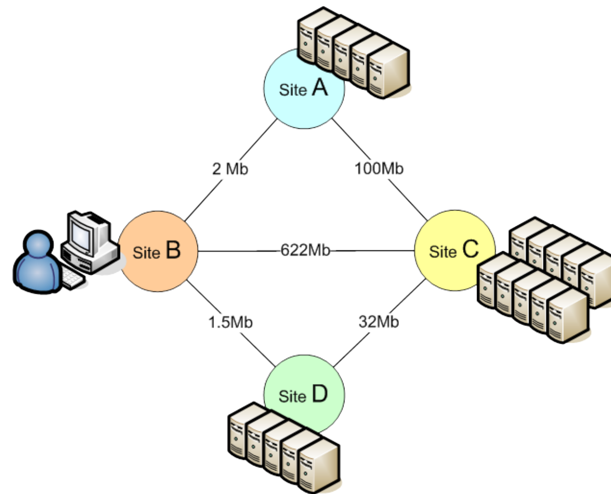



Figure 45 – Execution scenario which requires different gathering topology.

The total time from the event generation and its storage or utilization depends on the communication path used in the transmission. In order to have the benefits of heterogeneous network topologies, the monitoring tool may be configured to transmit the event data stream by a path different from the path used by common network routing. Figure 45 presents a scenario where the monitoring process should route events to the faster channel. The user located on site B submits an application for execution on sites A, B and C. If that application is instrumented to online performance analysis, the event stream generated from sites A and D will use common network routing mechanisms and should get only 3.5Mb of network bandwidth (2MB from site A plus 1.5MB from site D). That topology is called the *direct topology*. An *indirect topology* could be constructed to have the benefit of the high interconnection network between sites B and C. The idea is to merge the generated event stream from sites A and D within the C generated event data stream. That strategy allows the monitoring process to aggregate more events per packet, which lowers the overheads of single event transmissions and allows users to take advantage of Grid network heterogeneity. The

routing information is based on configuration information. That configuration may be static, by configuration file or dynamic, commanded by a tunlet.

The proposed architecture allows for the construction of any topology of data gathering. Any Application Controller is capable of event data routing and can be configured to establish event stream connections and route its events through those connections. This topology allows event ordering inside the Application Controller which reduces processing time from the monitoring client. Indeed, those topologies can increase the total latency for event data transmission because event trace data should pass to more than one hop. Although the influence of such collateral effects should be a trade off among network utilization and total transmission time.

## Scalability Issues

In Grid environments, there are many different network communication architectures. The event trace data path from generation to the analysis of the monitoring client has many steps. Those steps influence total overhead costs and determine system scalability. In a Grid environment, the event trace data passes from a local machine, a local cluster, or other machines, depending on the selected topology, and reaches the user workstation or some storage service. In first step, SMP machines can execute more than one application process job. In the presented architecture, each machine should have only one AC controlling the instrumentation and event data collection. The data transmission among processes may be done by using domain or TCP/IP sockets, shared memory or message queues. From these mechanisms, shared memory requires inter process synchronization, and message queues have a performance lower than domain sockets [83]. In some experiments using IPC mechanisms on UNIX systems, domain sockets have bandwidth values approximately 96.57% higher than TCP/IP sockets on stream transmission. Figure 46 presents the values from a Pentium IV 2.8 MHz machine.

The use of domain sockets provides better event trace collection bandwidths. In a local cluster, machines from the same cluster can have the benefits of merging the event trace data stream generated by each machine into one before sending them to lower networks. That helps solve event routing issues presented in the previous section. In the case of Massive Parallel Processor (MPP) machines, the use of tree topologies inside the machine can help to overcome network hotspot problems in network interfaces. This technique is presented in recent version of the Paradyn tool in order to provide

scalability by using a Tree Based Overlay Network (TBON) topology [84]. Paradyn also reduces the event amount in each node by event values summarization. This is possible because the monitoring client generally need aggregation functions over the event values.

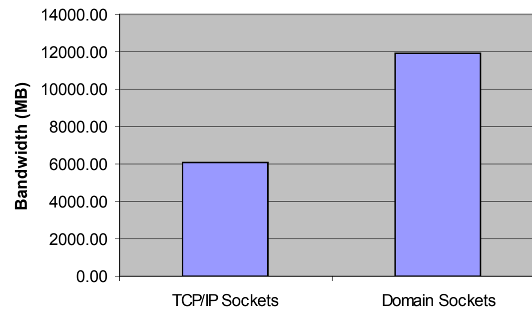



Figure 46 – Local event gathering strategy comparison.

### 4.3.4 **Smart Event Gathering**

Application programs need to be instrumented in order to generate event trace data. In dynamic instrumentation, this is done by changing binary codes and inserting calls to code sequences. In DyninstAPI terminology, the places where these modifications can be inserted are called execution points. A simple instrumentation capable of measuring a function execution time can be achieved by inserting the instrumentation in the function entry point to record the current timestamp value and inserting the instrumentation in the function exit point to calculate the difference between the recorded value and the current timestamp value. DyninstAPI allows such program binary patching programming capabilities. The overhead of the introduced code is proportional to its execution in relation to the original program code execution. In that sense, instrumentation of very small and greatly used functions may have high overheads and event tracing is not recommended in such situations [85].

Event tracing over processes consists of the execution of the code instrumented in a binary program and the transmission of the collected data to a monitoring tool program. In our architecture the instrumentation consists of a sequence of function calls that

record the program's state such as the parameters of functions or variables and sends the event trace data through domain sockets to the AC process. In such architectures, the overheads can be divided into the overhead of the instrumentation of runtime data collection and the overhead of data sending to the AC process. The overhead of event transmission depends on socket buffer sizes and the amount of data transmitted.

In our architecture, to lower the need for data transmission, we propose a more sophisticated component called sensors to be inserted at points inside the application process. The idea is to reduce and postpone the data transmission as much as possible by doing some simple logic and arithmetic operations inside the process space. Nevertheless, that kind of improvement affects the monitoring use of the generated event data trace because the received events can contain more information than simple trace data associated to a point of execution. The sensor concept is not new. The Autopilot tool [76, 77] uses this concept to collect data and preprocess data before sending it to its clients. We propose a more sophisticated use of sensors by inserting some codes to provide smart event gathering.

### InstallPoints

Once we know what should be measured, the other parameter to consider is where it should be measured. That also applies to the tuning part. The changes we make in application binary should be located in program execution and somehow synchronized. Similar to the Sensor concept, we encapsulate the logic of locating such points in program execution as InstallPoint.

The current prototype of GMATE implements the following InstallPoint types for placing Sensors and Actuators:

- Function Entry/Exit: locates execution address of functions by querying DyninstAPI using a pattern rule and generates install points on the entry or exit of first found points.
- Multi-Function Entry/Exit: same as the above, but installs the Sensor or Actuator at the entry or exit of all found functions.
- Binary Location: corresponds to an execution address in a program loaded binary. These points may be probed by a debugger tool such as GDB.

- File and Line Number Location Entry/Exit: an InstallPoint using the name of a source code file and a line number. Can be installed before the execution of the line or after its execution. Needs to use debug symbols.

All the presented InstallPoints are built over the capabilities of DyninstAPI for binary code patching.

## Simple Sensors

To lower instrumentation overheads resulting from event data transmission, some event consolidation should be done inside the process space. Simple consolidations such as sum, average and differences do not represent much computation overhead and can save some transmission operations. Sensors are software components that can be installed inside running processes at one or more points and have the logic to decide what to collect and when to generate the event data transmission.

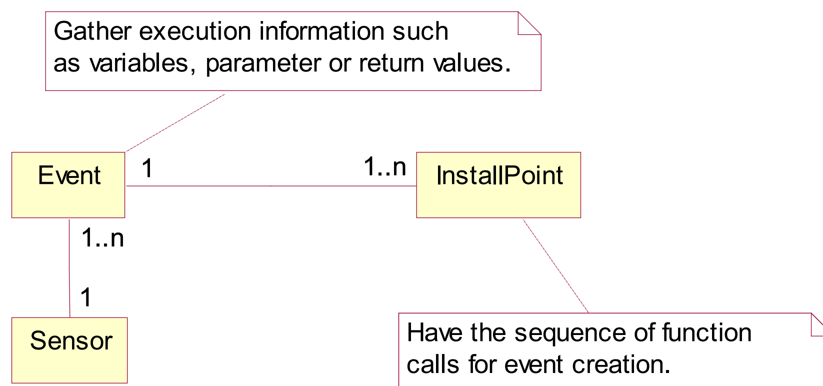



Figure 47 –Diagram for sensor concepts.

The simplest sensor is a software component that is activated at the installed points. On activation, the instrumentation placed by the installation process collects the running process information and sends the data through domain sockets to the AC process. These can be used by traditional event tracing. Sensors can be installed at any InstallPoint. In our architecture, an *event* consists of a data record that can be generated in one or more InstallPoints. Figure 47 presents a concept diagram associating sensors, install points and events. The idea is to use a same sensor to generate the same events

from different functions. That can be achieved by installing the sensor at different install points. Event identification is how the sensor binds to different installation points.

The concept of an event allows the sensors to identify different installation points and allows sensors to handle more than one tem of event information. At runtime, the event generation is represented by the instrumentation code used to call the sensors. Each event may contain information about program execution such as parameters of function or variables. Sensor components handle all the information required for event collection on installation points within user applications. Figure 48 presents a pseudo code of a sensor that handles multiple events installed on more than one installation point.

| **Sensor Installation** |
| --- |
| Execution of function **DMTLib_addSensor**(SensorConfig) |
| For each Event Mapped InstallPoint *ip* on Sensor *s* |
|     Get binary location usin Installation Point *ip* for Event *e* |
|     For each Installation Point |
|         Instrument call **DMTLib_sensorReset**(s.sensor ID, *e*.event ID) |
|         For each Event Data Property from *e* |
|                 Instrument call **DMTLib_setProperty**(s.sensor ID, measurementValue) |
|         Loop For |
|         Instrument call **DMTLib_sensorTrigger** (*s*.sensor ID, *e*.event ID) |
|     Loop For |
| Loop For |

Figure 48 – Pseudo code of a Sensor instrumentation process.

## Function Timer Sensors

Most event tracing is used to measure function execution times by computing the difference from function exit time minus function entry time. That calculation generally is done by the monitoring client. By bringing that computation to sensors, we save time on event trace generation because we can generate one event trace instead of two. That is called “Function Time Sensors” in our architecture.

Function Timer Sensors are sensors that recognize two specific events: region entry and region exit identified by installation points. Each event can be configured to sample values from parameters of function and variables and store it in the sensor. When the instrumentation triggers a region exit event, all the collection of stored event data are sent into the Event Channels. That allows, for example, to timer a complex code region in different slices and generates only one event trace.

### Other Sensors Types

By including the collection of logic inside the process space, it makes it possible that there are other ways of overhead reduction using more elaborated sensors. In that sense, sensors can be classified by the source of trace generation:

- **Trace data generated by event**: this is the case of function time sensors where the trace is generated after the function end event. In those sensors, the transmission of the trace event data is deterministic and identified by a mapped installation point.

- **Trace data generated by trigger condition**: in this case with sensors, the event generation is not deterministic and depends on the collected data and internal sensor logic. The sensor may use a trigger condition to control the generation or the trace event data. In such situations, the trace is generated in cases where a condition is evaluated to true.

- **No trace data generation**: the idea of these sensors is to provide information to be used by other sensors. In our architecture these sensors are called *'state sensors'*.

Our architecture proposes some example sensors for both trace data generation types:

- **Function Average Sensors**: this is a *'trace data generated by event type'*. The instrumentation is used to produce the average of a number of function executions. For example, a performance model can include a parameter as the mean of execution time of send and receive operations. In that scenario, this kind of sensor can be useful by recording fixed circular list of timestamp differences and generating events in a different frequency than a function time sensor. It can be configured to record a fixed amount of executions or time period threshold and to generate the trace data with the moving average execution a fixed number of

execution occurrences. The event generation can be disabled for use as *'state sensors'*. For example, a sensor could generate the moving average of the last 16 send operation durations.

- **Jitter Sensors**: this sensor type has the logic needed for calculating the difference between the different timestamps of different install points. That can help users in measuring program iterations or sequences of function calls. It can operate as trace data generated by events trigged on a configured number of samples collected or as *'state sensors'*.

- **Threshold Sensor**: this is a trace data generated by trigger condition sensor type. The idea is that the collected event data items should be sent as event trace data only when the value measured is below or above some threshold value. It can operate in '*lower than*' or '*greater than*' mode. The value evaluated in the install points can be read from variables, parameters of functions or other sensors using the inter-sensor communication protocol.

By using sensor communication, an Analyzer can configure sophisticated trace data event collection configurations. Sensor values can feed other sensors and the event data transmission can be dramatically reduced. Complex state machines (composed by many interconnected sensors) can be built using the trigger event subscription and sensor communication operation.

The idea is that the event data is generated only when it is really strictly necessary. The client of the monitoring, the Analyzer in our architecture, may use the provided sensor components to restrict the generation of event trace data. The sensor logic can build inside process spaces as a part of the logic that would be performed in the monitoring client. By using elaborated sensor types, much of the logic of the monitoring tool client may be included in the process space, reducing the communication between the monitoring engine and the tool which uses it. However, there will be a tradeoff between processing event data and data transmission overheads. In Grid environments where network bandwidth can easily become the application execution bottleneck, that strategy will have more importance.

## *4.4 Grid Performance Analysis*

The monitoring interface provides all event trace data that can be used to sense application behavior. Once we understand application behavior we can choose which performance indexes should be evaluated as objective functions and as parameters having an impact on such functions. To support such capabilities, GMATE inherits the concept from MATE, presented in section 4.2.3, and adds new features to help the tuning not generate application performance problems. The major characteristics are:

- The Analyzer is written in pure Java;
- The tunlets are compiled and loaded dynamically;
- The instrumentation installs Sensors and Actuators in application binary are in places specified by InstallPoint components;
- All communication between the Analyzer and AC is done using the Grid middleware;
- Asynchronous tuning as follows:
    - The event trace reception has its own thread;
    - The tunlet callback is multi-threaded;
    - The tunlet callback uses a different thread to the event trace reception;
    - The instrumentations are non-blocking operations;
    - The modifications on process binary are non-blocking operations.
- Have support for event routing;
- Have support for declarative Sensors, Actuator and InstallPoints;
- Independence of a message passing library;
- The DMLib is replaced by the DMTLib that also handles the tuning part;
- The event trace and action data are exchanged between a DMTLib dynamic library and the Application Controller;
- The tunlet API was extended to support callbacks in response to actions executed in an application program;
- Support for event ordering per application process and global issue;
- Provide statistics for event callback execution time for tunlet performance evaluation;
- Include in container statistical packages that facilitate tunlet development;

- The actuators change the process on the fly and do not require a process to stop;
- Enhanced InstallPoint types such as source files and the line-based and binary address-based;
- Smart monitoring with sensors to lower event trace generation

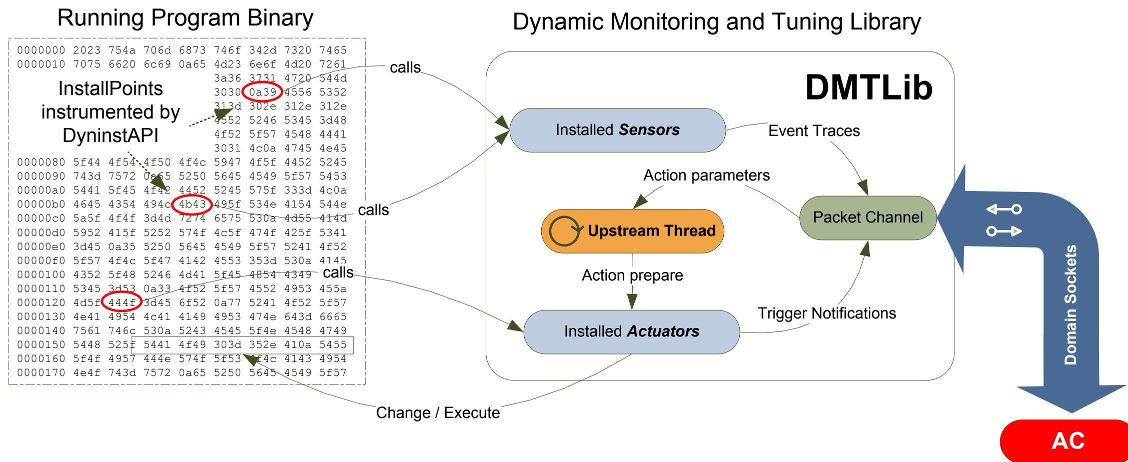



Figure 49 – Internals of the Dynamic Monitoring Library (DMTLib) and its iteration with the InstallPoints instrumented by DyninstAPI and the other GMATE components.

Figure 49 presents the internals of the DMTLib in relation to the DyninstAPI and the Application Controller. The library is a placeholder for the sensors and actuators installed in program binary. The program is required to pause only when sensors and actuators are installed. All the monitoring and changes are performed without the overheads caused by program pause.

When initialized, the DMTLib starts a thread that is responsible for the collection of Action parameters required for Actuator preparation. The detailed Actuator internals will be presented in section 4.5. All the communication between the DMTLib and the outside world are done through the AC.

Figure 50 presents an internal view of the Application Controller (AC) component. The AC is connected to the DMTLib using Domain Sockets and to other components by Event and Management Channels. The AC has the capability to install Sensors and Actuators using the DyninstAPI and the DMTLib functions as detailed in sections 4.3 and 4.5.

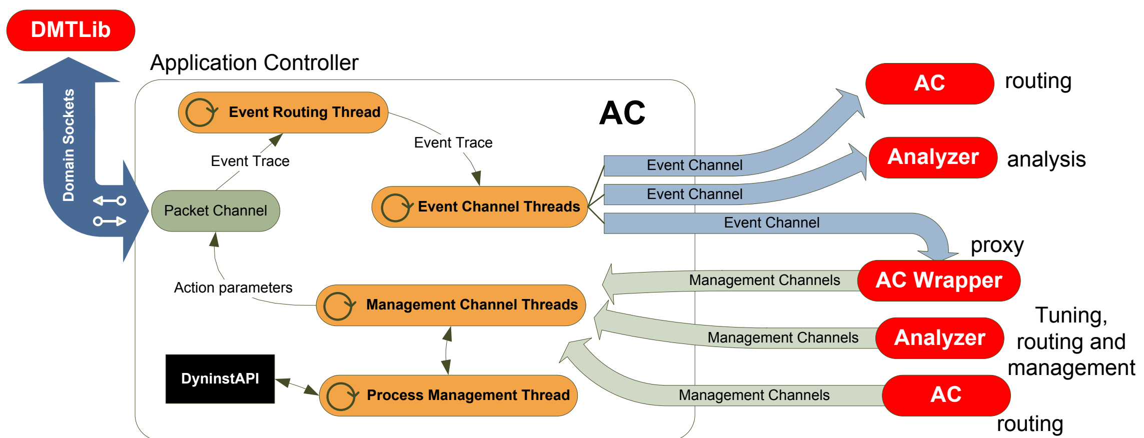



Figure 50 – Internals of the Application Controller (AC) and its iteration with DyninstAPI and other GMATE components.

All generated trace data are transmitted using Event Channels and can be used for routing, using troughs other than AC, analysis and proxy. The management channels are used for proxy, routing and tuning. The Action parameters are transmitted over the management channels from an Analyzer to the AC and later to the DMTLib.

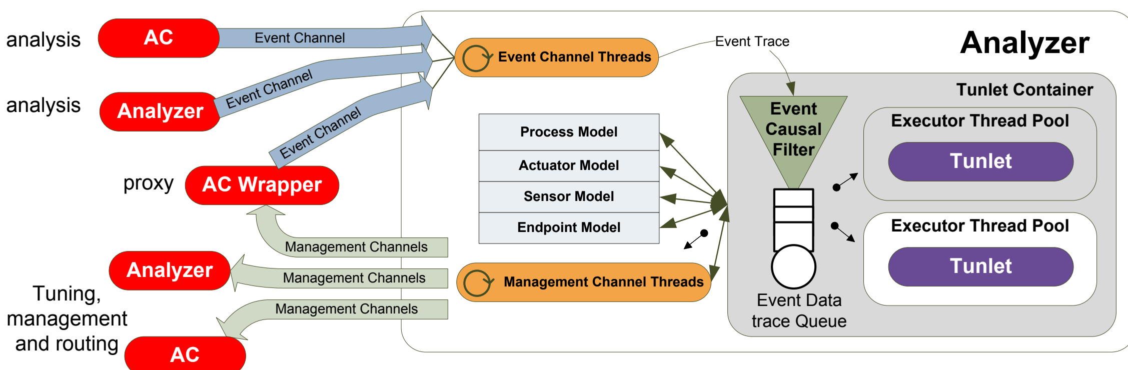



Figure 51 – Internals of the Analyzer and its iteration with the ACs and other GMATE components.

Figure 51 presents an internal view of the Analyzer component. As with the AC, the Analyzer supports Event and Management Channels. It uses the Management Channel to request Event Channel establishment, and Sensors and Actuators instal on

InstallPoints. When an AC connects to a Management Channel on the Analyzer, it corresponds to an Endpoint connection in the Analyzer model. The Analyzer maintains a list of connected Endpoints[6], registers application processes and creates Actuators and Sensors.

When a process is located, the AC is registered on the Analyzer and initiates the commands required by the installed tunlets. The analysis is done inside the tunlet. That is done by following some heuristic targeting performance improvements. These heuristics can range from simple index functions to data mining procedures, decision trees and stochastic methods such as simulation or complex analytical analysis.

### 4.4.1 Tunlet Architecture

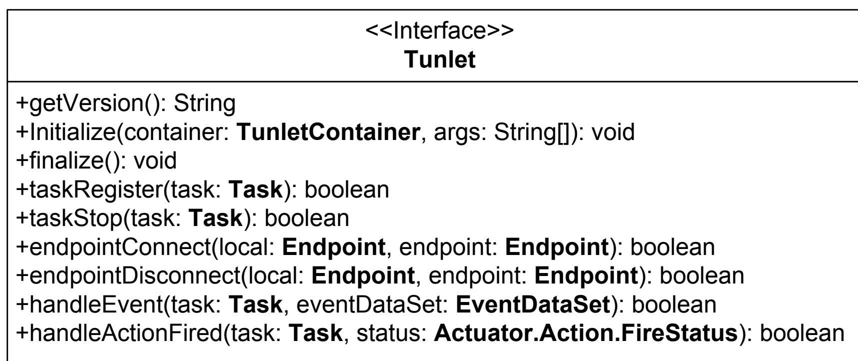



Figure 52 – Tunlet Interface.

The tunlet are any class write in Java that implements the interface presented on Figure 52. The interface allows the Analyzer to notify the tunlet about process, endpoint and event management. Each application process is represented by an instance of object Task. The endpoints are any Analyzer or AC that receives Event Data Channels or Management Channels. That interface allows, for example, for notifications when the Analyzer receives some connection from an AC or when an AC receives a connection

[6] We name endpoint any component that may generate or receive Management and Event Data Channels in our architecture.

from other AC, in case of event routing scenarios. The TunletContainer is an interface that allows the tunlet to interact with the Analyzer.

Within the presented methods, the tunlet may interact with the Task, requesting Sensor and Actuator installation, query process state, variables, available instrumentation points and query binary composition. The frequently called methods and the 'handleEvent' and the 'handleActionFired' that are called in response from some sensor event trigger or some actuator action fired on some application process respectively.

The helper classes/interfaces implemented by the Analyzer are:

- TunletContainer Services
  - It maintains the *SystemModel* instance.
  - It maintains the *CausalControl* instance per tunlet.
- SystemModel Services
  - It maintains the list of Sensors, Actuators and InstallPoints.
  - It maintains the list of EventInstall (where relates events from Sensors to InstallPoints) and ActionInstall (where relates actions from Actuators to InstallPoints).
  - It maintains the list of running Tunlets and registered Tasks.
  - It maintains the configurations of the engine.
- CausalControl Services
  - Configures the local and global event filter.
- Task Services
  - Sensor event install, enable, disable and remove.
  - Actuator action install, enable, disable, remove and fire.
  - Process start, stop and terminate.
  - Binary structure.
- Sensor.EventDataSet Services
  - It maintains a list of Sensor.Event.EventData instances.
- Sensor.Event.EventData Services
  - It maintains the measurement values collected in application process.
  - It knows when the event trace occurred.
- Actuator.Action.FireStatus Services
  - It knows the status of the change action. It allows, for example, for error detection.

The CausalControl is a mechanism that filters the events a tunlet receives based on event metadata configuration. Each event may have two causal filters, one local and one

global. A causal filter is a simple an integer. The CausalControl maintains for each process a local causal filter and one global causal filter. If the value configured on the event is greater than the actual value of the CausalContol, the event is queue until the CausalControl updates its values and the event configuration satisfies the filter. It allows specifying in declaration form, that the tunlet receives only the events it is prepared to deal with.

## *4.5 Grid Tuning*

The main distinction between computational Grids and cluster computing is the high level of heterogeneity in networks and processors. In addition, these systems are generally geographically spaced and their components distributed among different administration domains. As covered in monitoring session, the problem of security needs due to multiple administrative domains should be left to Grid Middleware.

As with monitoring, the tuning module uses communication channels to perform application changes as a result of performance model evaluation. When applying MATE concepts to Computational Grids, we should deal with communication restrictions that occur with communications to remote sites. The tuning process can use the same concepts applied to monitoring to lower the intrusion in application execution due to execution blocking. Those execution blocking occurs when the tuning is synchronized to the application execution in the event trace generation or in the application changing process.

Another important aspect in application tuning in Computational Grids is that changes are performed in different conceptual layers. For example, the tuning of a number of workers in cluster computing consists of application modification, while the same changes in Computational Grids may require the addition to that interaction of collective layers such as meta-schedulers and/or resource broker services. Any additional resource should be obtained by the iteration with those collective services.

In fact, the original implementation of MATE has a tight integration with PVM process controls in order to deal with process creation and management. Same concepts were applied to the Grid, although that problem was broken down in two parts: First, we needed to find where the application processes starts its execution. Second, if an application supports changes to a number of processes, we considered how to grow or

shrink parallel virtual machines. The first part was covered by monitoring requirements. The second part was covered by tuning processes.

The main aspects involved in the tuning process are *where*, *how* and *when* to allow changes in applications. That classification allows us to trace a correlation between MATE concepts in cluster computing and their appliance to Grids. The '*where'* aspect consists of the issue of which element receives the change actions. The change in the number of workers in an application may require interface with meta-schedulers and/or resource brokers in combination with application binary modifications. The *'how'* aspect resides in what functionality is required to interface with elements identified by the '*where'* aspect. In a case of cluster computing, we had used runtime binary modifications such as changes in variables' values and function replacements. In Grids, we need additional services such as petitions to collective services as a collection of group metrics abstractions using MDS [9] or NWS [46]. The '*when'* aspect consists of the synchronization required to perform the changes. Some applications may only support variable changes at some time in its execution. Those changes require being synchronized among different application processes and being in synchronization with some service requisition.

In fact, an execution of a parallel application in a Grid can be seen through different prisms as in the case of clusters: *software modularity*, *communication/administrative domains*, and *isolation abstractions*. By "software modularity" we mean the classification proposed in [14] and illustrated by Figure 53. The idea is that if we propose dynamic changes in common parts such as the Framework and Library code we may benefit all applications that use these software modules, while modifications applied to application codes benefit only the application. Changes in an Operating System Kernel is also considered, but changes at that level are generally considered as system tuning, which are out of the scope of this thesis work.

Within Computational Grids we may divide tuning processes as based on communication and/or administrative domains, as illustrated by Figure 54, and isolation layers, as illustrated by an 'Hourglass model' presented in Figure 2. There are changes that may be performed in distributed system abstractions such as interface with collective services. There were some changes that may be done in Virtual Organization abstractions such as economic costs or limits regarding VO polices.

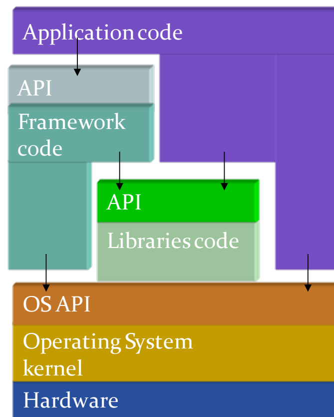



Figure 53 – Software modularity abstraction from [14].

At the Organization level, we may have resources usage police which may also have economic issues. In cases of CE and CH abstractions, we should have the same concepts as cluster computing. In addition, we should have iteration with Fabric services such as batch schedulers. In CH and node cores, we have the application process instances that should receive binary modifications considering the software modularity abstractions.

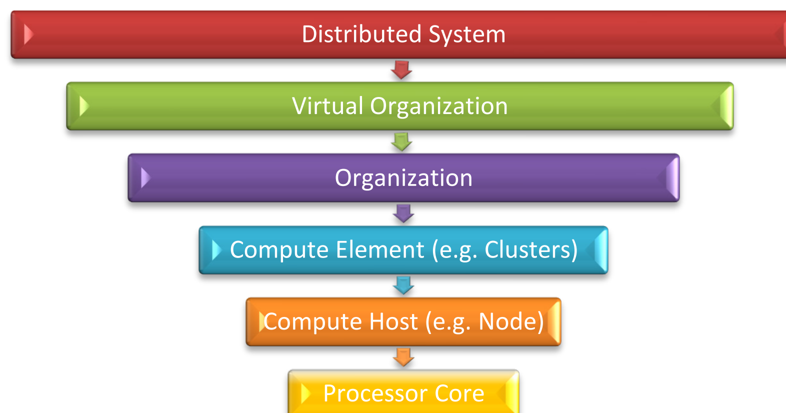



Figure 54 – Communication/Administrative domains abstractions.

In order to facilitate the tuning process, we need an abstraction that isolates different modifications from different layers throughout an interface. For that direction, we

borrow the concept of software Actuators presented in [76] and adapted to MATE concepts. The idea is to have a component that can be 'plugged' in different places covering the *'where'* concept and hide some functionality logic to handle the *'how'* concept and identify conditions for the synchronization required by the *'when'* concept.

### 4.5.1 **Smart Tuning Actions**

Tuning actions consist in the appliance of changes which affect application execution behavior. This process could be analyzed as a piece of code which is based on an internal state interacting with some element which corresponds to the performance of the change. In cluster computing implementation of MATE [14], the modifications were performed by the Application Controller (AC) in response to Analyzer requests. That implementation uses DynInstAPI breakpoints to stop applications at the synchronization points and performs the desired change when the execution reaches that point.

One interesting behavior in changes performed by dynamic tuning is that applications are constantly evaluated during execution. In some senses, the modifications are generally performed at same points using different state data. For example, the tuning of a number of workers generally consists of a variable value change or a function call execution using different parameters. By perform smart tuning, we mean to install in just one instrumentation operation a piece of software that makes the application capable to change its behavior by receiving some action instruction from an AC.

We have found some benefits using that approach. First, the instrumentation is done just once for each Actuator object instance, which reduces process stops and the overhead of tuning. Second, the modified binary, after instrumentation, can be executed without needing to be attached to an AC. That has special benefits for applications which frequently use signals. When processes receive signals and they are attached to another process, the parent process receives the signal and has the responsibility of continuing child processes. That generates a lot of intrusion into application processes that make intensive use of signals while also being controlled by DyninstAPI. We found also that domain sockets are the communication scheme that has less overhead in action reception.

The difference between tuning processes and monitoring concerns which thread executes the operation. In a case of monitoring, trace data is transmitted using

application threads, while tuning requires a different thread to wait for change requests, inside application process. We, in that case, are assuming that the system has threading support.

The philosophy of tuning has the same principles of monitoring. Due to communication costs, tuning processes should be implemented taking into account the low overhead in process execution. At that point, part of the logic required to change application configurations in order to tune may be encapsulated in components inside application processes.

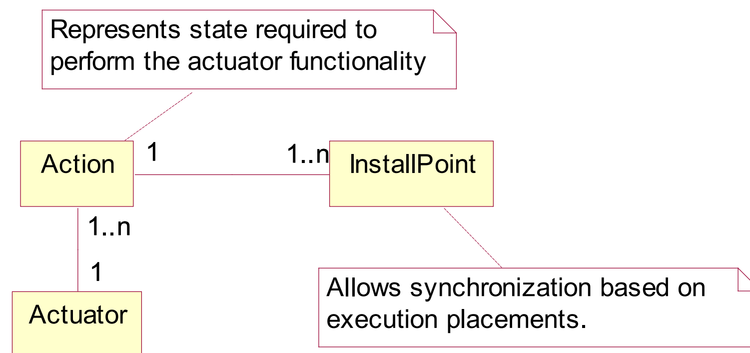



Figure 55 – Actuators concept.

## Simple Actuators

An Actuator is an element of software that may perform one or more actions. These actions represent the knowledge component needed to execute the change. These actions may be installed in one or more InstallPoints. Take, for example, the action of changing an integer variable value on some function entry. In that case, the action is an integer that represents a new value and the InstallPoint is the function entry. Figure 55 presents a diagram correlating those concepts.

Basically, after an actuator is installed, it waits for a command from an AC containing some action value collection. When an actuator receives the action state, it becomes '*armed*'. When an application passes on some install point related to an action, the Actuator *'fires'* an action which performs the change. In fact, actuator components work

on an internal state machine that interacts with an AC. Figure 56 presents the internal Actuator state machine. When an Actuator is installed, the initial state is Active. When an AC sends a change request, the required data is stored inside an Actuator component and its internal state become Armed. When a program execution reaches some InstallPoint, the Actuator is triggered and the change is performed.

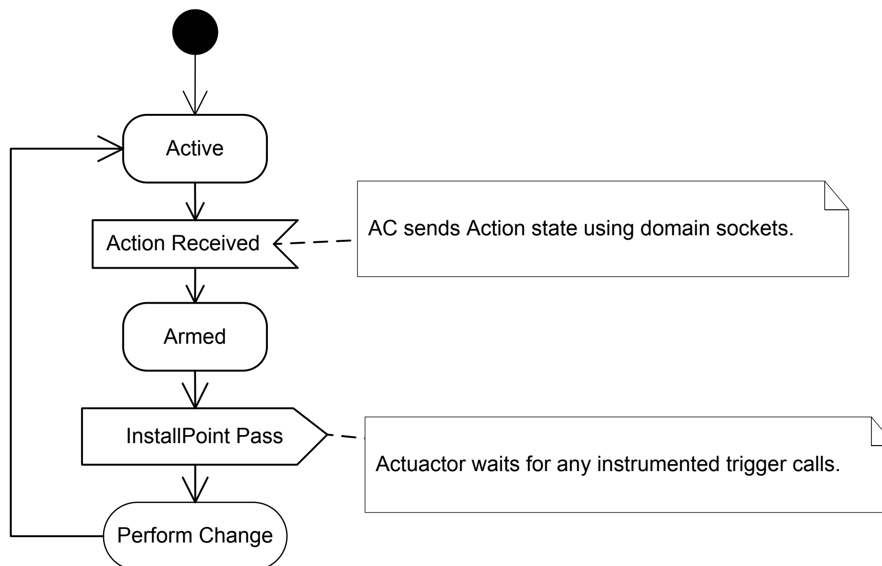



Figure 56 – Actuators internal state machine.

By using the runtime scheme illustrated in Figure 56, the change does not request program stop as with MATE implementations and maintains the same synchronization properties. For example, if we install an Actuator for changing the value of one variable, when the Analyzer request the value to change, it commands the AC to send the new value using domain sockets to the application process. Inside the application process, our architecture has a thread waiting for such commands, called the *'upstream thread'*. When the *'upstream thread'* receives the new variable value, it arms the Actuator with such value. When the program passes in an InstallPoint where the Actuator is installed, the variable value is changed automatically by the Actuator.

In some situations, Actuators may be configured always to become 'Armed'. That means it is able to perform its logic all times when triggered. That functionality is useful for some Actuators, as explained in following sections.

| **Actuator Installation** |
|---|
| Execution of function **DMTLib_addActuator**(ActuactorConfig) |
| For each Action Mapped Installation Point on actuator *c* |
|     Get Installation Points for Action *a* |
|     For each Installation Point *ip* |
|         Instrument call **DMTLib_actuactorReset** (*c*.actuactor ID, *a*.action ID) |
|         For each property from Action *a* |
|                 Get pointer *ptr* to value to be changed. |
|                 Instrument call **DMTLib_setProperty** (*c*.actuactor ID, *a*.action ID, *ptr*) |
|         Loop For |
|         Instrument call **DMTLib_actuactorTrigger** (*c*.actuactor ID, *a*.action ID) |
|     Loop For |
| Loop For |

Figure 57 – Pseudo code of an instrumentation process of a change value Actuator.

Figure 57 presents the pseudo-code of the Actuator installation with the capability of variable and parameter change. When the application execution reaches these instrumented function calls, the Actuator internal code may change the values passed by reference if it is in *'armed'* state.

### Value Change Actuators

Generally, application tuning is done by configuration change. That configuration generally consists of variable values that an application program uses to change its behavior during execution. When an Actuator is installed into a program space, the Action has its items associated with the references of variables. When an Actuator is armed, it stores the new variable values in a temporal buffer and replaces the old values when an Actuator is triggered.

| **Actuator Installation** |
|---|
| Execution of function **DMTLib_addActuator**(ActuactorConfig) |
| For each Action Mapped Installation Point on actuator *c* |
|     Get Installation Points for Action *a* |
|     For each Installation Point *ip* |
|         Instrument call **DMTLib_actuactorReset** (*c*.actuactor ID, *a*.action ID) |
|         For each property from Action *a* |
|                 Create a variable *var* on process heap. |
|                 Get the reference *ref* of *var*. |
|                 Instrument call **DMTLib_setProperty** (*c*.actuactor ID, *a*.action ID, *ref*) |
|         Loop For |
|         Create a variable *callFunction* on process heap. |
|         Get the reference *refCF* to *callFuntion*. |
|         Instrument call **DMTLib_getCallAction** (*c*.actuactor ID, *a*.action ID, *refCF*) |
|         Instrument 'if (callFunction!=0) call a →functionName using all allocated *vars*' |
|         Instrument call **DMTLib_actuactorTrigger** (*c*.actuactor ID, *a*.action ID) |
|     Loop For |
| Loop For |

Figure 58 – Pseudo code of an instrumentation process of a function call Actuator.

## Function Execution Actuators

Sometimes, the modification of program configurations should be done by function calls. One example is the configuration of the buffer size of sockets. That can only be done by a system call requesting a change of socket buffer size. For that situation we build an Actuator that represents a function. The Action property items, or parameters, are mapped to the function arguments using memory allocated in process heap.

The configured function is called when an Actuator is triggered by an install point and it is in *'armed'* state. That is controlled an allocated variable in heap space. The instrumentation detail for such Actuator instrumentation is presented on Figure 58.

### Other Actuator Types

*Value Map Actuators*: consist of an Actuator that has auto-armed actions that serve to change the values of variables or parameters. That can be used, for example, to generate different process mapping to MPI executions. At program startup, the tuning tool may install one actuator of that type to swap some process ranks. That may be useful for placing the master processor of a Master-Worker application in any rank in a transparent manner. The installation of this actuator type is the same of the value change actuators.

*Web Service Actuator*: consists of an Actuator that may be used to perform a web service call. That is useful in cases of changing the number of processors of an application which can be done by interacting with collective layers using web service calls. Using the Globus middleware, for example, we can request more resources using a web service call.

### 4.5.2 **Tuning in different layers**

Another problem when tuning applications in computational grids is to find out which application parts are addressed to Grid Middleware. When a parallel application executes on a Computational Grid, a number of processes may be a requirement but is a runtime value that is controlled by a meta-scheduler. As a consequence, the application may not use more machines unless as meta-schedulers for new ones.

When we talk about changing the number of workers in a Master-Worker application, the changes must be done at many levels. The tool should, at a Compute Host abstraction, change the process binary and, at a Compute Element and/or Collective Layer, request more machines for a system.

In some senses, we may classify the changes in the distinct abstraction layers presented in Computational Grids. If changes grow or shrink an application's set of resources, it should interact with Collective services, if not only Fabric changes are sufficient.

Chapter 5

# Experimental Validation

## *5.1 Introduction*

In Chapter 2 we explained the properties that characterize a computational Grid and the implications for parallel executions within that environment. We had shown that after the application starts its execution, it has to deal with a multi-cluster environment. In the same chapter, we presented some active state of the art tools used to generate performance data for analysis. We use some ideas in our architecture detailed in chapter four and for the application of dynamic tuning in Grids we proposed the model detailed in Chapter 3. In the same chapter, we presented simulated results for the analytical model. This chapter presents the validation of the performance model detailed in chapter three and its evaluation in a controlled enterprise Grid environment testbed.

The main characteristics of a representative computational Grid testbed are a collection of computer nodes grouped in clusters interconnected using a network with different properties to an intra-cluster network. That should represent distinct CE's distributed among different geographic locations characterizing network heterogeneity. These clusters should have machines with different processors and memory characteristics to ensure CE heterogeneity.

Our main goal in this chapter is to highlight some case scenarios with real application tuning using our model and architecture. That should provide a comprehensive set of examples for its applicability using the provided ideas and techniques. We present some

cases already exercised in Chapter 3 using simulations to illustrate real world scenarios. The evaluation should cover applications with different compute/communication characteristics. As we state, those applications should allow different grain sizes change during execution: the provided workload should sample distinct examples of ratios from different strategies for grain size partition.

Our tuning model and architecture should help Master-Worker applications execute over a Grid Test-bed to adapt to system heterogeneity. From that, the first parameters for the experiments are intra- and inter-cluster network latency, bandwidth and processor speed, architecture and memory. Another important parameter is the mapping of these resources to the application. Different application executions should get different system configurations and a different process to parallel machine mapping. From the parallel machine view, we have the parameters' initial number of workers and grain size selection.

From the provided parameters, the parallel machine mapping, initial number of workers and initial grain size selection are evaluated by means of different factors. The executions should be ranked using total execution time and processor efficiency. Due to the current limitation of the available implementation of messages passing from Computational Grids, we cannot increase the parallel machine size using dynamic process spawn. Indeed, that characteristic should be available soon and does not affect the provided experiments as proof of the concept.

Our model requires the parallel application to be developed using the Master-Worker paradigm and supports dynamic grain size change during its execution. To facilitate application coding with these properties, we developed a template to abstract the communication logic and help the application work out when it should change the grain size and how it should be done. We use some examples of applications developed using this template as the experiment's workload.

## 5.2 Master-Worker Tuning on Grids

In following sections we detail how we facilitate the development of applications with runtime grain size change support. To apply the model presented in chapter three, there are some requirements. The application should have a stable average task size in its properties of time to compute and time to transfer input data from master to workers and

output from workers back to the master. We analyzed the one port master model. As input and output communication from a master to workers shares the same port (considering full duplex communications), we serialized the communications. Serialized communications within Master-Worker programming model results in lower startup and finalization times, as described in chapter three.

### 5.2.1 Framework Overview

To abstract the Master-Worker paradigm we developed a Generic Master-Worker Application Template (GMWAT) which consists of a set of C++ classes created to hide communication logic from the application. Using that template, the developer should implement a simple API to create tasks, compute tasks and write results. In addition to that, we added the API to split tasks and merged results to allow grain size change. When the template asks the application to split a task, it expects to have different communication requirements from the resulting tasks and the original task. The detailed framework implementation is provided in Appendix B.

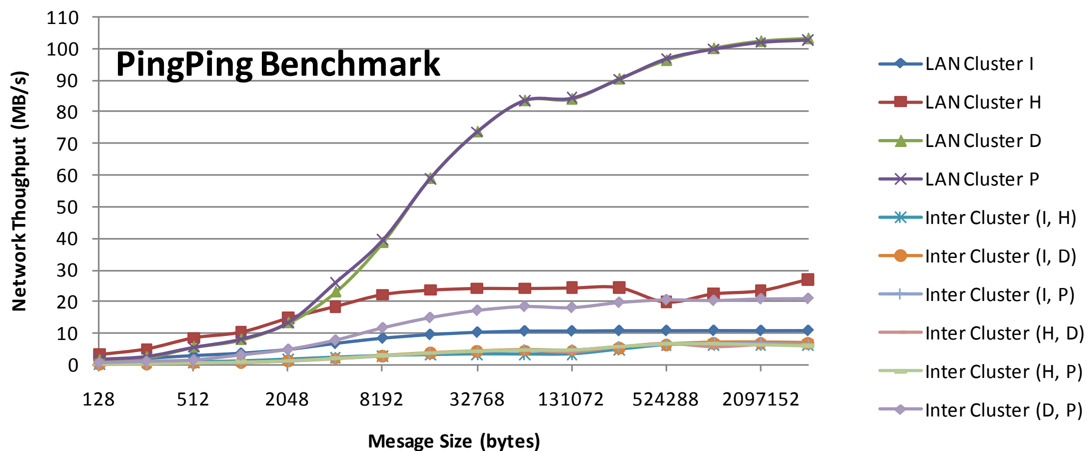



Figure 59 – Benchmark of communications among different nodes in the constructed Grid testbed using the Intel Pallas MPI Benchmark [86].

### 5.2.2 System Description

The enterprise Grid testbed consist of a set of 52 compute hosts spread over four computer elements represented by clusters with different network and processor

characteristics. Figure 59 presents the different network throughput for intra-cluster communications (LAN) and inter cluster (WAN), and the following Table 4 presents the machine characteristic as processor architecture, memory specifications and operating system versions. To facilitate the identification of different multi-cluster configurations we labeled the clusters as I, H, D and P.

Table 4 – System characteristics for each clusters used to the experimental validation.

| id | frontend | N | Processor, Memory and Cache | Network | Bogomips |
|---|---|---|---|---|---|
| **I** | aogrdini | 6 | Intel(R) Pentium(R) 4 CPU 2.80GHz, 512MB RAM, 512Kb L2 cache | 100Mb Fast Ethernet | ±5550 |
| **H** | aohyper | 8 | AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ (2GHz), 2Gb RAM, 512Kb L2 cache | Gigabit Ethernet | ±4000 |
| **D** | aoclsd | 8 | Intel(R) Pentium(R) 4 CPU 3.00GHz, 1Gb RAM, 1024Kb L2 cache | Gigabit Ethernet | ±6010 |
| **P** | aoclsp | 32 | Intel(R) Pentium(R) 4 CPU 3.00GHz, 1Gb RAM 1024K L2 cache | Gigabit Ethernet | ±6010 |

### 5.2.3 Compute/Communication Dependency Analysis

Our model states that if there exists data reutilization among different tasks, we can change the compute/communication ratio by selecting different grain size alternatives in a stateless Master-Worker execution. As we needed lower variance in task execution times (i.e., the compute should not be data dependent), we need to be able to divide a task load in a uniform way. For example, a matrix multiplication problem can use kernels as BLAS implementations to avoid cache influence in task load division. Figure 60 presents different task load divisions for matrix multiplication using the GNU Scientific Library (GSL) and the Automatic Tuned Linear Algebra System (ATLAS) implementations of BLAS API. As we can see, different grain sizes provide an almost linear compute time.

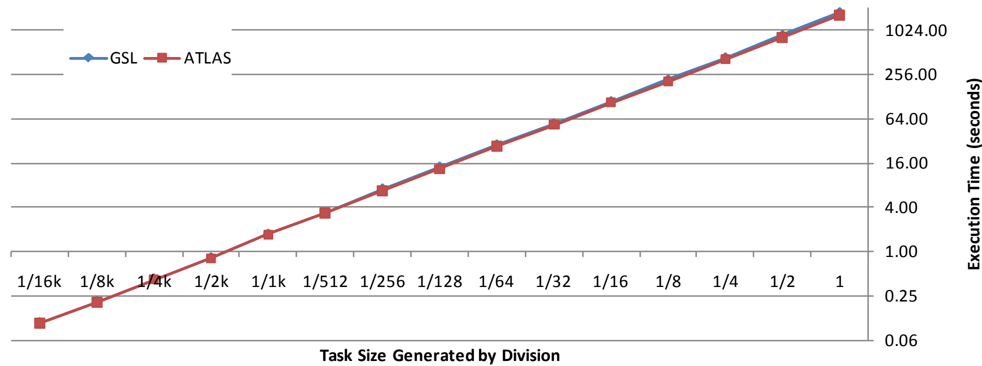



Figure 60 – Ratio of different grain size division using GSL and ATLAS BLAS implementations.

We found that using the strategy task division at runtime facilitates the implementation of grain size change support in parallel application development. If we have data reuse among these task divisions, different network and processor speeds may characterize certain grain sizes as a compute- or communication-bound application.

As presented in Chapter 3, the parallel programming paradigms help developers to divide a load among processors. In that process, we do not require that the grain size selection follows certain functions (i.e., making task computation time be function of grain size). Our model only requires that application support increases and decreases a compute/communication ratio as a function of a numeric value named as grain size. Different divisions may compose different computations and communications as a function of grain value.

In a steady state the pipeline composed by task transmission, execution and result transmission limits the number of workers. We expect that the grain size increase has different impacts on task and/or result transmissions from the impact of task execution. We consider that tasks are composed of data chunks. We may have, for example, the following scenarios of data reutilization:

- Task division shares a data chunk. This scenario could be applied in forest fire simulations where a task can be represented as a map and a set of different simulation configurations. A sub task shares the full map from parent task but has a subset of the simulation configuration. As a result, we may have different simulations combined in one map. The master role in such applications may

recombine the maps from results obtained from the sub tasks in the expected result of the parent task or may assign this work in other phases to workers. We name that division strategy as **Fixed Shared Data Chunk (FSDC)**. Another application that may be parallelized using the same heuristic is heat propagation simulation. Each task reuses the border elements from other tasks for computation as detailed in section 5.3.1.

- Task load is the product of chunks. In this scenario we classify the matrix multiplication parallelization using row and column set division. A row set participates in all tasks for each column set to generate the result matrix. We detail such implementation in section 5.3.2. Another example is the NBody simulation in which a group of bodies should be reused among tasks. That implementation is detailed in section 5.3.3. We name this strategy as **Product Data Chunk (PDC)**, where computation grows exponentially from smaller to coarse grain tasks. We also found that the popular implementation of a protein alignment search tool, BLAST, has the same properties. The parallel version has same query data reused with different database fragments [87]. In such cases, the task load is the product of query subdivision and database subdivision.

In sequence, we present the analysis of grain size tuning in some example applications with Fixed Shared Data Chunk and Product Data Chunk division strategies executed in the detailed Grid Testbed. We assume that the master process mapping is a batch scheduler attribution which we will not control.

## *5.3 Application Case Studies*

### 5.3.1 **Synthetic Dynamic Master-Worker**

The following analysis is performed on a synthetic master-worker application that mimics both Fixed Shared Data Chunk (FSDC) and Product Data Chunk (PDC) division strategies. The number of different scenarios of multi-cluster configuration we may construct is large, so we present limited cases, where a master process is mapped onto a higher speed processor/network cluster (P) and slower speed processor/network cluster (I).

The FSDC load type consists of a number of tasks to be executed in a number of configured iterations. Each task is composed by a number of data chunks. The data are generated using a pattern which allows the worker to validate whether the data is transmitted correctly. Each task can be decomposed by the application into two tasks containing the first data chunk and splitting the other data chunks into two. The compute interval for each task is calculated using the product of the chunks that compose the task.

The Testbed described has Compute Hosts (CH) with a private IP address. The communication between machines from different clusters requires that the used MPI implementation supports that kind of infrastructure. The follow executions use the GridMPI that supports the IMPI Standard for interoperability among MPI executions to form a single parallel machine. In addition to that, the GridMPI implementations provide messages relays to proxy messages passing between private networks as a message passing proxy, as described in section 2.3.

The following experiments assume that we have assigned a multi-cluster by a resource broker. The application maps the MPI rank using the sequence of clusters. The startup process of the MPI parallel virtual machine consists of the following steps that are explained in detail in section 2.3:

1. Start the IMPI-server and get the connection contact information;
2. For each cluster with private address
    a. Start IMPI-relay with the IMPI-server connection contact information and get the IMPI-relay connection contact information. In Globus toolkit-based implementations, this process is started with the *fork* type in GRAM job submission;
    b. Start the MPI processes using the '*mpirun*' command using the IMPI-relay connection contact information.

Each process group started in step (2) contacts the IMPI-relay in each cluster and provides the information about the local processes. The IMPI-relay contacts the IMPI-server and exchanges local information with the other processes participating in the execution. After that, all processes are capable of communicating with each other, using the IMPI-relay as necessary. That allows for executions using the Internet, for example,

using a configured TCP/IP port for inter-cluster communications. From the application perspective, the processes show no differences between them, and all are identified as a numeric rank in a MPI_COMM_WORLD communicator.

## Fixed Shared Data Chunk (FSDC)

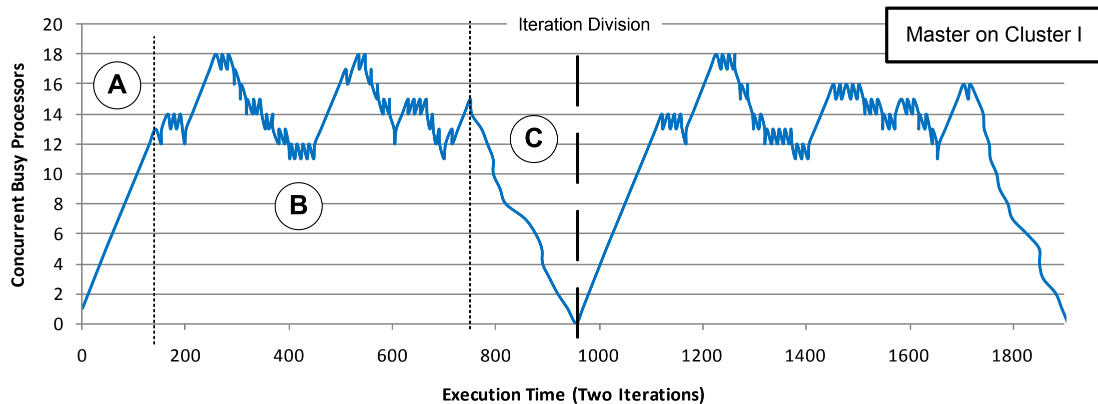



Figure 61 – Two iteration execution of synthetic Master-Worker where the master processor is mapped on cluster I. The phases A, B and C are startup, steady and finalization, respectively.

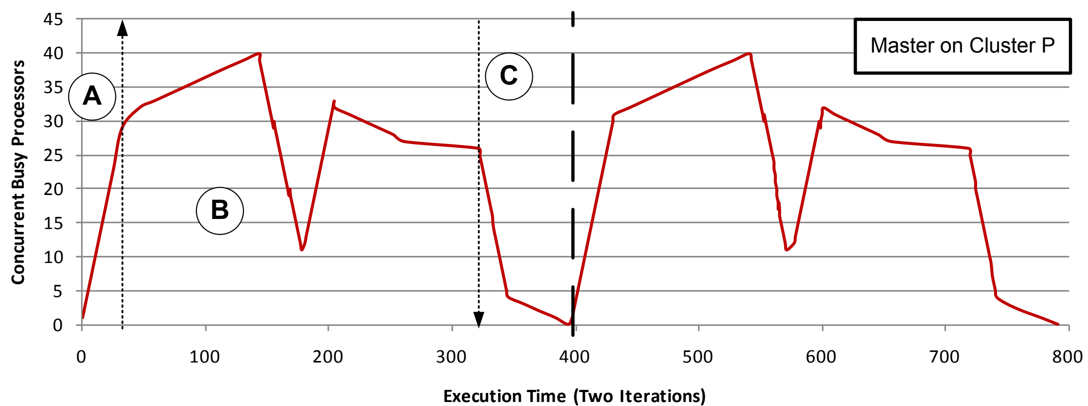



Figure 62 – Two iteration of synthetic Master-Worker where the master processor is mapped on cluster P. The phases A, B and C are startup, steady and finalization, respectively.

Figure 61 and Figure 63 present the execution of two iterations of FSDC data reuse with maximum grain size selection for clusters I and P. Note that due to higher network

speed properties in cluster P in Figure 62, the startup and finalization phases (A and C) are shorter than Figure 61.

As the limitation of processing is that the master is blocked by communications to slow workers, we can analyze the Master to Worker task communication times. Figure 63 provides a histogram of these communication times where we can get a clear differentiation between local and remote communications.

Figure 63 – Histogram presenting the tasks send times from Master to Workers.

In the tuned executions, as we control the environment, the '*mpirun*' startup process executes the Application Controller (AC) component of GMATE. The AC connects to the Analyzer and parses the application process binary, loads the Dynamic Monitoring and Tuning Library (DMTLib) inside the process memory space and initializes the data structures to manage the actuators and sensors installed. After the binary parse, the AC registers the process execution in the Analyzer using the configured management and event channels. In the Analyzer container, the process registration locates which tunlet is responsible for its tuning process. If it is the first process to register, the tunlet is initialized and receives the process registration event.

Since we are working with the same binary for all process, we installed all the required sensors to gather the measurements needed for the tuning process and command the application to continue its execution. The tunlet instance repeats that procedure for all application processes registrations. When the application executes, the sensors send collected event data to the Analyzer that passes it concurrently to the tunlet. Depending

on the Analyzer location, events from workers may arrive sooner than events from the master. We use the causal order to evaluate if event data should be processed or should wait for a preceding event. We leave that responsibility up to the tunlet since the Analyzer does not have semantic information to provide event ordering.

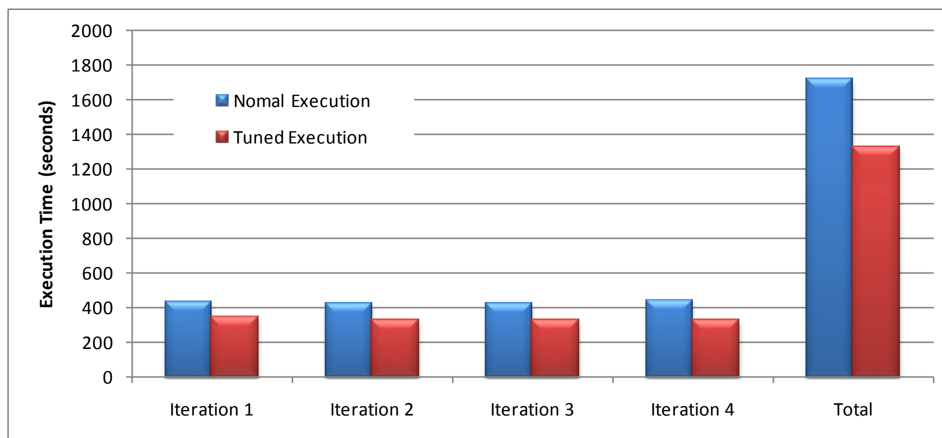



Figure 64 – Comparison between execution with and without tuning grain size, considering the master mapped on cluster P.

The first event received in the application processes is the configuration of the template instrumented with a simple sensor installed in the execution of function *'user_config'*. We collect from that event trace the application process role assigned by the template, the number of work units per iteration and the number of iteration from the problem size. When the tunlet receives that information from the master process the tunlet side creates the master related data structures.

The decrease of concurrent busy processors in the middle of a steady phase (B) in Figure 62 is a result of a master blocked in its communications due to slower inter-cluster communication.

Figure 64 presents 28% of total execution time reduction. In the first iteration, it reduces by 24% and the following by about 28%. The tunlet was configured to start with the smallest grain size and balance to get the best grain size. Note that this is different from the model proposed in [66]: within the first iteration we have gains.

### 5.3.2 **Matrix-Multiplication Application**

The Matrix-Multiplication problem has a different compute/communication function of grain size. Suppose, for example, that we have to multiply two matrices as presented in Figure 65. We can, for example, have only one task where we transmit the two matrices and receive the matrix result. We can call that the maximum grain size that the problem supports and this is one task containing the complete problem.

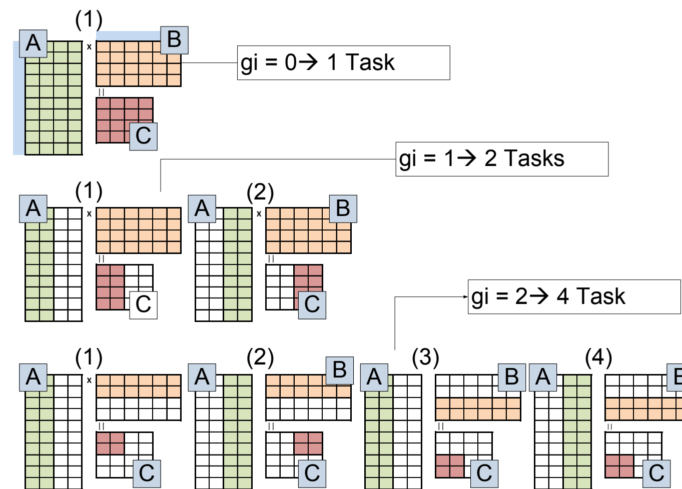
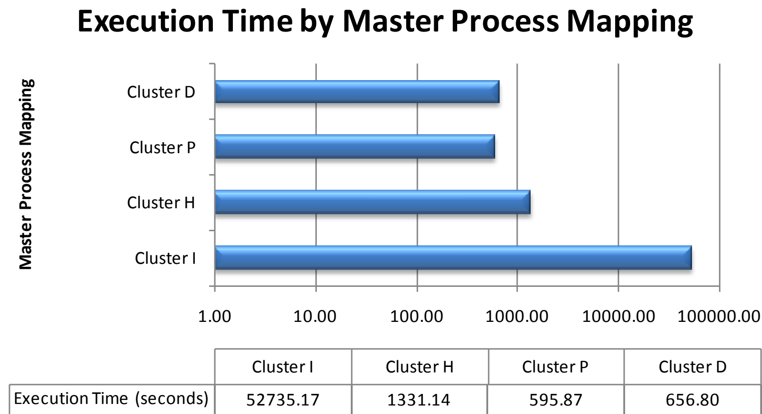



Figure 65 – Graphical representation of grain size change in the parallelization version of matrix multiplication problem. Each task can be decomposed into smaller ones with data reuse among tasks.

Suppose that we use a finer strategy. We can choose to divide each task into finer ones using a strategy of alternate dimension division. In the example presented in Figure 65, a task with grain size *gi = 0* (maximum grain size) can generate two tasks of grain size *gi = 1*, and these two tasks of grain size *gi = 2* can be decomposed into four tasks of *gi = 3,* and so on. That strategy allows that we may change the grain size of the tasks sent to workers by decomposing coarse grains.

When executing the Master-Worker application within the scenario detailed in section 5.2.2 we may have different process mapping, considering the different localization of the master process. Figure 66 presents the impact of the master process mapping given the same resource set.




| | Cluster I | Cluster H | Cluster P | Cluster D |
|---|---|---|---|---|
| Execution Time (seconds) | 52735.17 | 1331.14 | 595.87 | 656.80 |

Figure 66 – Comparison of same problem size and parallel machine configuration considering different Master-Worker processes to node mapping.

Note that the application becomes communication-bound in the executions where the master process is mapped in cluster I because it is the cluster with less network performance. We monitored all machines and ensured that the application had no memory swap use which could cause such execution time variations.

### 5.3.3 N-Body Application

The parallelization of n-body is done by load division. The most time-consuming operation in an n-body problem is the calculation of distances between bodies in space. Taking that in account, let it be a task to be processed composed of the permutation of block divisions of total body counts, called segments. Each segment is present in the same amount of tasks generated by permutation. See a complete example from a set of 12 bodies divided in 3 segments of 4 bodies each on Figure 67. Note that the operation among bodies from same segment is fragmented in the permutations. Using such data partition schedule we have uniform input and output task size and number of operations performed in task computation.

To have a uniform time among task executions, the calculation of distances from a block to itself should be divided into permutation participations. If $d$ is number of segments, the number of generated permutations are $(d^2-d)/2$. Each segment participates in $d-1$ tasks, so the distance between bodies of the same segment should be divided equally among these $d-1$ tasks in which this segment participates.

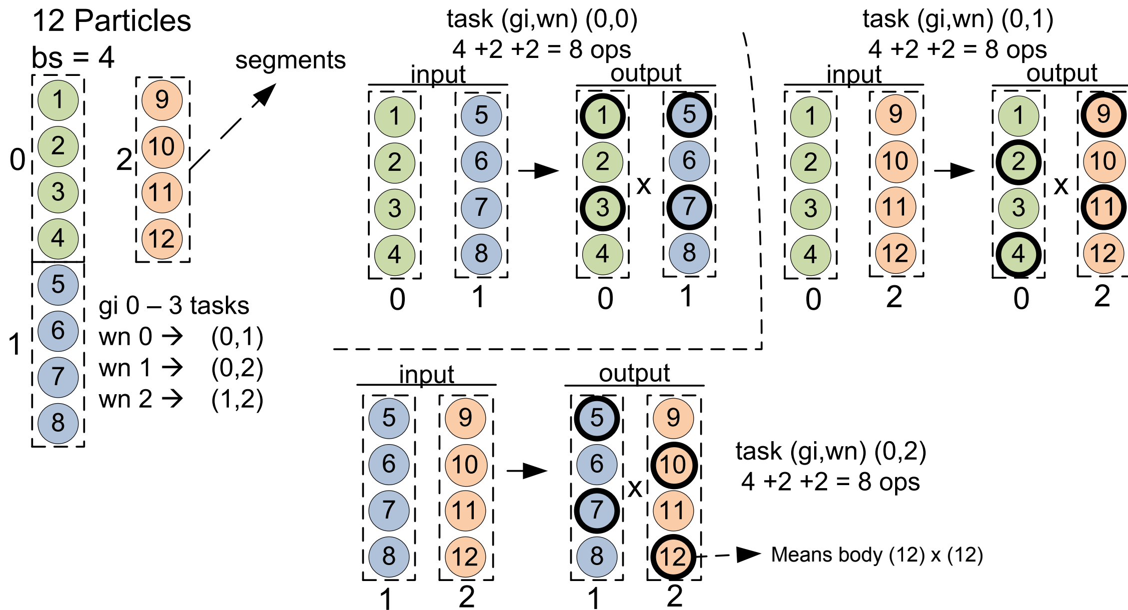



Figure 67 – Graphical representation considering grain size in N-Body problem with uniform task load and input/output data size.

Take, for other example, the following instance: a set of bodies divided into *d=4* segments *{A,B,C,D}*. The generated tasks should be *{(A,B), (A,C), (A,D), (B,C), (B,D), (C,D)}*. The task *(A,B)* should calculate the distances between the bodies from segment A to segment *B* and one third of the calculations for distances from bodies from segment *A* to itself and the same from bodies from segment *B*. This strategy divided the amount of operations between tasks to nearly equal.

## 5.4 Architecture Validation

When centralizing the analysis of collected events, the machine where the Analyzer executes receives all events from the Application Controllers. That may be a problem if the Analyzer is mapped to a machine with low bandwidth and high message latency from these Application Controllers. In following experiments we analyze the performance of the monitoring and tuning phases.

### 5.4.1 Sensors Overhead Analysis

The tuning process requires the monitoring of application behavior using sensors to acquire the runtime metrics and parameters. This data is transmitted back to the

Analyzer through Event Channels, as described in section 4.3.2. The path of the data from the collection to processing is: sampled by binary instrumentation, transmission to the AC via Domain Sockets, transmission to the Analyzer through Globus XIO or socket communications and delivered to the tunlet.

When the produced sampling rate is higher than any step in that path, we introduce an overhead in application execution. That overhead is higher when the analyzer receives the communication over a slow WAN link. The goal of this experiment is to verify, given LAN and WAN networks, how great is the overhead of event collection using the Sensors.

```
void pfunc(int period) {
          usleep(period);
}

(...)

t1 = sampleTime();
for(i=0;i<count;i++)
          pfunc(period);
t2 = sampleTime();
```

Figure 68 – Source code from the instrumented program.

The load is generated by a simple program that calls a function named *'pfunc'* a thousand times and exits. That function has just an *'usleep'* call as presented in Figure 68. The idea is to have a different frequency of event generation. The load is characterized by the parameter *'period'* which represents the value used in the *'usleep'* function call. We expect that, if the given generation rate is higher than the network transmission, the total program execution time should be higher than the program without instrumentation. We use clusters I and D detailed in section 5.2.2.

We created a simple tunlet that does nothing with the collected data and a sensor declaration with two events installed in the entry and exit of the program code. We used

the following sensor/actuator xml model file presented in Figure 69 and the model from the Timer Sensor is presented in Figure 70.

```
<?xml version="1.0" encoding="ISO-8859-1"
standalone="no"?>
<GMATEModel>
<installPoints>
    <installPoint id="1"
     localRule="pfunc" pointType="functionEntry"/>
    <installPoint id="2"
        localRule="pfunc" pointType="functionExit"/>
</installPoints>
<sensors>
    <sensor id="1001" type="simpleSensor">
        <event id="1" logInfo="pfunc" />
        <event id="2" logInfo="pfunc" />
    </sensor>
</sensors>
<eventInstalls>
    <eventInstall sensorId="1001" eventId="1"
        installPointId="1" installPattern="." />
    <eventInstall sensorId="1001" eventId="2"
        installPointId="2" installPattern="." />
</eventInstalls>
<actuators />
<actionInstalls />
<tunlets>
    <tunlet type="file">SimpleSensorProfile.java</tunlet>
</tunlets>
<startup>
    <managementChannel mode="listen" port="41007"/>
    <eventChannel mode="listen" port="41008"/>
</startup>
</GMATEModel>
```

Figure 69 – Model for simple sensor profile experiment.

Figure 71 presents the overhead of the Simple Sensor monitoring with the Analyzer placed in the 'I' cluster and the program in the 'D' cluster. The execution where the Analyzer is placed in same cluster gives the mean execution with and without instrumentation which are statistically equal (mean values ranged by standard deviation overlaps).

```
<sensor id="1001" type="timerSensor">
          <event id="1" logInfo="pfunc" />
          <event id="2" logInfo="pfunc" trigger="yes" />
</sensor>
```

Figure 70 – Specification of the Timer Sensor from Simple Sensor.

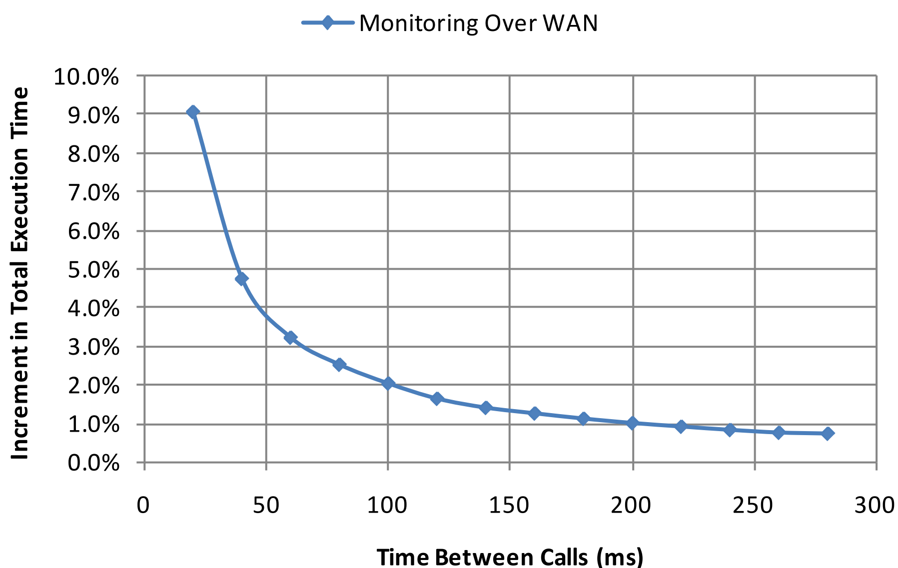



Figure 71 – Overhead of monitoring using Simple Sensors to time a function.

Chapter 6

# Conclusions and Future Work

This chapter presents the findings and conclusions obtained from our work for each problem statement and our proposed contributions. The chapter also highlights the possible open lines that can be pursued as research topics on dynamic and automatic tuning of applications and heterogeneous systems.

## *6.1 Conclusions*

With the emerging of Grid Computing, the use of HPC came closer to large numbers of users distributed around the world. That facilitates collaboration among organizations within research projects, providing shared resources and higher common computational capabilities. These resources include clusters or workstations, MPP systems, and supercomputers located in many university departments or industry organizations interconnected by the Internet or private networks. The Grid concept abstracts these resources as computational elements which can be used by applications to solve complex and increasing problems that demand more computational power than that available within local resources as single clusters.

Indeed, the dynamic behavior and heterogeneity characteristics of the Grid make application tuning difficult due to the lack of system information. Some properties are hard to predict, such as network bandwidth or the topology of the resource set that will be assigned to an application execution. The dynamic behavior of Grid environments reinforces the need for dynamic tuning tools since the user has less control over the

application target execution hosts. The heterogeneous character of Grid systems can generate different performance problems from those found in the same execution in controlled environment such as clusters. The number of parameters can influence the execution time increase and most of them cannot be controlled, such as available Internet bandwidth and latency (inter-cluster communications over shared networks).

This thesis presented contributions to the applicability problems for automatic dynamic performance tuning in parallel/distributed applications in computational Grids. We expect this research to provide technical and conceptual solutions that help the support of dynamic tuning in systems with higher levels of heterogeneity, such as Computational Grids. The first problem encountered was how to deal with system transparency without replacing or reinventing the system services. In advance, we noticed that the network heterogeneity of Grid systems boosted the overhead results from the monitoring and tuning phases for automatic tuning. Many Grid systems use the Internet as an inter-sites communication backbone. The Internet is well known as a shared environment where we do not have control over the resources used.

Another interesting aspect is how application development was changed with the use of these systems. To achieve parallelism, users used different mechanisms such as functional decomposition in services, data decomposition in parametric executions, scientific workflows, wide area job execution pipelines and wide application parallelism using message passing. Each of them has benefits and drawbacks: we choose to work with wide area application parallelism due to the performance analysis needs of such applications. It is difficult to measure and analyze shared resources located in different administrative domains and geographically distributed.

We chose the Master-Worker programming paradigm and analyzed the impact of the uses of more than one virtual communication channel and communication managers to hide network heterogeneity of application execution considering these environments. During application behavior analysis, we found that applications may have benefits in the reduction of execution time if it could deal with dynamic selection of different grain sizes. In the literature, the granularity was seen as a choice of application/algorithm design and it is fixed at the development time. After that, an application has some fixed compute to communication ratios which limits its scalability in systems with different characteristics.

To get better compute to communication ratios, we can play with two parameters, the compute time and the communication time. There is nothing to be gained in increasing the compute time because that would increase the total application execution time. Indeed, the communication time, if it is overlapped with the compute time, could give the possibility of exploring a different communication volume. We state that certain types of applications, when parallelized using the Master-Worker paradigm, generate data reuse among different tasks. That changes the total data communication volume required for distributing the load considering different grain sizes. The variation of that communication volume, when overlapped with compute, affects the total execution time. The grain division strategy may suit a different resource set topology assigned to the application.

We also had to adapt the architecture used in automatic tuning to overcome the intrusion into application execution and possible bottlenecks resulting from tuning applications in wide area network (WAN) executions. We found that using decoupled processes for collecting, analyzing and tuning could reduce the overhead impact in the application processes while maintaining the benefits of the application adaptation to system characteristics. We break many synchronization issues of the tuning process in order to expect architecture scalability with low overheads.

With these previous problems and findings we can divide the contribution into more details and results in following sections.

### 6.1.1 **Process Location**

The main idea behind a Grid system is transparency. The user should launch her application and the system uses its requirements to find a place to execute its processes. The user shall not have direct control over those mechanisms because the resources exposed by the system are in continuous change. Different executions of an application generally receive different sets of resources based on decisions that are taken by different software layers, such as a meta-scheduler.

We presented two approaches for process location within computational Grids that can be used by an automatic tuning environment tool: one that uses the Grid information services which require administrative privileges and another that uses user level access. We found that the information systems provided in current implementation middleware

could slow down the application process startup. The information propagation among the system may generate inconsistent global system views from the location process. Indeed, it is the only solution that works in environments where the CH does not have network access to outside world. Within the used level approach, the application processes startup is guided by the Application Controllers which call back the Analyzer establishing the management and event channels required for the automatic tuning process. The ideas and results of that architecture were published in:

> G. Costa, A. Morajko, T. Margalef, E., *"Process Tracking for Dynamic Tuning Applications on the Grid"*, Journal of Computer Science & Technology (2007), vol. 7, no. 1.

### 6.1.2 **Security Polices**

The most predominant characteristic of Grid Computing is the different administrative domains. The process execution should follow strict different local and organization policies. That includes which TCP ports are open, the sequence of communication establishments, the communication privacy identification and authorization [19].

We assume that the processes required to perform the application instrumentation and tuning should share the same security infrastructure as the application. That includes in the proxy PKI certifications the application exports on execution. These certificates allow us to identify and to authorize communications among processes to services and processes from different CEs and CHs. The implementation details and ideas are also published with the process location results.

### 6.1.3 **Lower Communication Intrusion**

Different from MATE implementation, GMATE uses an asynchronous mode form in communication between analyzer and application controllers. That reduces the total time of data transmission and possible overheads due to communication blocking. Since grid environments are geographically distributed environments, communication is a critical issue that can significantly affect application performance.

We assume that the conventional event monitoring process generates too much raw performance data. With limited available bandwidth to perform online collection, we choose to move part of the analysis to where each process executes. The idea is to pre-

process and filter the data as much as possible by doing some analysis and aggregation at the moment that the performance data is collected. Instead of inspecting the execution behavior with event collection, we created a component that handles some simple processing mechanism inside the application process in order to allow performance event data preprocessing. We called that “smart sensors”. With a smart sensor concept, we lowered the total number of messages required to gather performance data used to feed the performance model analysis.

Another contribution concerns how the tuning actions are performed. Instead of stopping application processes from acting over their state variables and/or functions such as MATE, we chose to instrument trigged controlled components called Actuators to control these variables and call functions whenever necessary. That reduces the time we needed to change process parameters, thus expecting better performance improvements where we lower the intrusion from the tuning process.

With the concept of smart Sensors and Actuators we do not need too much synchronization between the Analyzer and the Application Controller because we do not stop the application from collecting performance data and perform tuning actions. That lowers the instrumentation phase with the possibility of a batch installation of Sensors and Actuators and not generating signals to processes to stop their execution in case of some variable change or some function execution in response to a performance model implemented inside a tunlet instance. That reduces the overhead to less than 0.5% on GMATE compared with 2% to 5% of overhead generated by MATE.

### 6.1.4 **Middleware Integration**

We detail a strategy of middleware integration that allows the tool to use system services in order to help the tuning process. That was the first initiative to move MATE concepts to build a new tool called GMATE. We used the Globus XIO communication layer provided by the Globus Middleware to ensure firewall passing and security policy conformance. The communication messages among application controllers and analyzer were routed in different and independent channels. The management and Actuator activation messages are transmitted in management channels and the measurements generated by the Sensors are transmitted in event channels. The AC can be integrated to the Globus container using a proxy component which exposes AC managed commands as web services. Within that interface, the Analyzer may request the creation of required

management and event channels. The details of process location and tool integration in the Globus Toolkit middleware were published on:

> G. Costa, A. Morajko, T. Margalef, E., *"Automatic Tuning in Computational Grids"*, Applied Parallel Computing. State of the Art in Scientific Computing, Lecture Notes in Computer Science, vol. 4699/2008, pp. 381-389, ISBN 978-3-540-75754-2, Umeå, Sweden, Jun. 18-21, 2006

### 6.1.5 Performance Models

We proposed a performance model that allows for the tuning of grain size within applications over heterogeneous resources found in multi-cluster executions within Grid systems. The model is based on a balance of the compute – communication ratio of application load division in tasks in order to use all provided resources with minimal startup and finalization time on master-worker executions. The analytical heuristic model and exhaustive simulation results were published in:

> G. Costa, J. Jorba, A. Morajko, T. Margalef, E. Luque, *"Performance models for dynamic tuning of parallel applications on Computational Grids"*, IEEE International Conference on Cluster Computing, vol., no., pp.376-385, Sept. 29 2008-Oct. 1 2008, ISBN 978-1-4244-2639-3. Tsukuba, Japan, Sept. 29 -Oct. 1, 2008.

We use the benefits of the load balance provided by the Master-Worker executions considering that the available workers can be grouped in different communication domains. The analysis of application execution within such scenarios shows that we can decompose in parallel Master-Workers overlapped sharing of the master and its communication capabilities.

We designed an application template to help with application development using the paradigm of multiple communication channels and independent task distribution among available master communication channels. We included the necessary hooks for dynamic binary instrumentation and we shielded the concurrent parameter changes from outside processes with the necessary synchronization mechanisms. The experiments with different multi threaded MPI implementations show that we can expect compute to communication overlap and task received communication to result in send communication overlap.

The proposed models were implemented in a tunlet capable of instrumenting the application template to create communication channels, assign workers to communication channels, change grain size and shutdown worker processes. We used that tunlet to adapt the applications developed with the application template to execute using four different clusters with different processor and network characteristics, lowering application execution time and shutting down the workers that the master was not capable of using, which raised the measured efficiency in resource use.

## *6.2 Open Lines*

Due the dynamic characteristic of Grid computing, applications need to adapt themselves to different system configurations over time. This can be driven by tools that have knowledge of the properties of the system and about the application. Those tools can dynamically monitor the application and the system and select actions that can be made online in order to get performance improvements.

Due to network influence on the monitoring overheads, the analysis process called the Analyzer may have different configurations. It can be done in a centralized, hierarchical and complete distributed approach. In the centralized approach, the Analyzer process responsible for collecting the information and doing the online analysis is placed close to the resources in order to minimize network bandwidth influence. The hierarchical approach can be used to reduce the necessity of data transmission among 'far' monitored resources.

The idea is to have the local Analyzers placed 'close' to analyzed resources in order to preprocess and reduce the performance data produced by those resources. In response to that, local Analyzers generate abstract events representing collected process data and send it to a central analysis process. This case can reduce dramatically the network requirements needed in communications. In such schemes, local Analyzers transmit to central sites only small event abstractions required to compute global model states. In the complete distributed approach, each Analyzer instance cooperates with others using abstract events which represent the performance data of the controlled resources. For example, in scenarios where we could have an SPMD application running among many clusters, we may have one Analyzer in each cluster tuning grain size and load distribution parameters. Following the SPMD heuristic, this Analyzer should only

inform communication partners about local cluster states and the global application model should have partial updates with that information in a distributed shared view. We can use a consensus to ensure view consistency.

The current analyzer implementation is developed in Java which can be adapted easily to allow a tunlet migration capability. We can, for example, have a scenario in which a tunlet acts like a mobile agent and the Analyzer as an agency. In such scenarios, the tunlet may decide to migrate to a different tunlet container expecting to lower event collection and provide a faster response time for system changes.

Different approaches of the Analyzer process distribution may require different models which can be broken into partial models to be distributed among different Analyzers. We expect that some performance models can be evaluated and tuning suggestions may be applied inside process spaces, by the sensor model, some performance model tuning suggestions can be applied on cluster level, by local Analyzers, and other model tuning suggestions can be applied at a Grid level or, in Grid terminology, Collective layer. The idea is to minimize event transmission among different levels of analysis lowering the intrusion overhead as much as possible.

Other improvements that could be useful would be to have soft programmable sensors and actuators. The idea is the tuning tool could complete customized sensor and actuator behavior through a script language. That script language could directly program the sensor and actuator code using DyninstAPI binary programming capabilities [48]. In such scenarios, tools could program sensors and actuators logic cores inside process spaces in order to monitor and change process behavior based on measurements taken during the process execution. This could allow for the creation of tuning scenarios completely inside process spaces, or in-process automatic tuning.

Concerning the topology aspects of data gathering, the monitoring process can use information about Service Level Agreement (SLA) contracts to control the pace of the event data stream generation. By means of those contracts, events can be packaged in compressed information frames, and could lower the network requirements of event data transmission.

Some interesting complementary tools that help monitoring and analysis in Grid environments could be event storage services. Different from current Grid storage

services, these services could handle a decentralized and synchronized stable storage for event data. In a postmortem analysis, users could query those storages for the events they are interested in and receive ordered multiplexed event data. That could allow for analysis of wide applications such complex Grid scientific workflows.

### 6.2.1 **Application Parallelism Support**

One recognized problem regarding parallel application development is the complexity of the performance engineering. The application developers are generally field-specific specialists, not high performance computer specialists. When they involve performance specialists, the application and its embedded algorithms are generally coded and running in a prototype implementation which should be analyzed in order to obtain performance improvements. The problem we see in such processes is the analysis of source code for potential parallelisms is much harder than at a conceptual level. When an algorithm is developed, its conceptual problem is mapped in data and function structures. Let us take the n-body parallelization strategy used in section 5.3.3. Without breaking down the analysis of the forces of each body in a composition of single steps (calculus of the force influence for the other bodies over the body under analysis), it was not possible to generate different grain sizes with uniform load division and data division among tasks. We acknowledge that there are many different algorithms for the n-body problem considering SPMD approaches, but we advocate the use of simple load balance mechanisms as master-worker or/and pipeline strategies to deal with complex system configuration which involve a hetereogeneous network and processor.

There are many initiatives to declare the parallelism semantic during application development. These semantic facilitates prototype applications with the use of Application Templates [60], language preprocessing directives [88-90], automatic parallel code generation [91] and different languages semantics [92, 93] where developers have helped with problem parallelization decomposition. Indeed, we are far away from semantics that work well with the high degree difference in the systems the applications should run within. That problem grows when the number of cores increase on many-core systems with many memory cache strategies and group core specializations. Different approaches for these parallel application development semantics have as a common goal how to help developers specify the application in a way such that potential parallelism can be exploited. We can classify the ideas in

functional based and data based parallelism semantics, but there are no semantics of how to specify the load division, considering for example, the parallelization Task/Channel methodology presented by Foster. If the developer can specify in semantic terms the units used to generate application grains we may have programs capable of adapting themselves over these complex system architectures with the tool support to tune performance.

As with the problem of mapping, the problem of task grouping is that its complexity is also NP complete. The trends are that we have more elements in system architecture, and, without application composition malleability, its adaptation within large systems will be even harder. Within Grids, application developers experience some of the solution when breaking applications in workflows with a thousand job tasks. These job tasks are scheduled independently and, by having the benefit of High Throughput Computing, we lower an application's total execution time.

We see a promising research field in bringing these asynchronous task processing queues, represented by CE's schedulers, to application task's composition/decomposition, guided by the queues semantics, in HPC services provided by the different systems. The use of queue theory in the development of parallel applications allows for the construction of asynchronous out of order schemes for task execution. The message passing API provides some facilities that can be used to queue identification (message tags), what is missing is the semantic definition for processors that consume those queues and the semantics for processors to queue routing mechanisms.

With the task queues consumed by processors and the capability of task splitting and merging semantics explicated by the application developers, we could have the possibility or better performance prediction, system adaptability, and application state definition that could help dependability, and give easier load balance in heterogeneous architectures. We should note, indeed, that programming applications using such paradigms may require a mind-shift, of the sort that is hard to achieve nowadays. The standards that survive are the most used and, in some cases, not the ones with better characteristics/results.

### 6.2.2 Data Type and Domain Semantic Definition

When we try to develop applications with different grain sizes and get the process of working with those grain sizes set up in an independently manner, we did not find support in programming languages. The mechanisms provided by a type definition are static and without a type to type conversion process. It is common knowledge among developers that a different data memory layout produces different performance indexes. Most of the difference between the serial and parallel codes of NAS kernels is the composition and decomposition of data types used to perform inter-process communications and better memory layouts for intensive computes. That is a trade off between simplicity and overheads. It is hard to get lower overheads in message passing considering the complex types composed of a high number or non contiguous memory segments. We found that, if we have a programming interface or language semantics to specify how data types are composed, and how to identify instances of its compositions, we could have systems with message passing implementations that may balance the network load through caching known data.

Consider, for example, that we could have a data type definition semantic that allows specifications in which different instances from some variables share common data. The message passing could reduce the amount of data needed to be transferred, by acknowledging the existence of the shared part within message composition. From the application developer perspective, all codes required to pack and unpack those variable in messages to be transmitted would be simplified. The semantics definition could be added to the standard message passing its benefits to facilitate application development.

A more complex semantic could be found in type composition. Imagine a scenario where we have three processors where two processors send messages that are assembled in a complex message received by a third one. If we have some mechanisms for data type definition which allow that the third process received a composed type, we could simplify the application development to data domain engineering. The lesser code could reduce error incidence and the message dependencies specification could help, or hide, the problem of causal dependency among messages during application development. We may say that, having explicitly support for message dependency, we could have lower application synchronization points. With less synchronization requirements, we have less load unbalance.

### 6.2.3 Multi-Core Issues

With new many-core systems, the application development semantics using shared memory and distributed shared memory receive revived attention. One problem in using these semantics in programming is to detect locality at runtime. When two or more processes have their load centered over shared variables, we have serious performance problems. For example, if two programs update different variables within the same cache line, each update instruction invalidates the other process cache. To overcome that, developers use some synchronization steps to reduce process to process locality interference, creating working regions. The problem is to balance these region sizes, when the application executes over machines with different architectures. In some scenarios, the processes should shrink or expand their locality by regions using different region sizes in order to balance the load among the available cores, lowering execution time.

If applications could have hooks where they could be used to guide the regions' shrinkage and expansion, the multi-threading programming could have more benefits than the dynamic tuning. The parallel execution environment could detect how well these regions block the execution threads, how well the jump prediction and cache hit statistics are and play with the region size to get better application adaptation to the execution environment. In certain ways, the region sizes could be the grain size options within shared memory systems.

We should not forget to mention that the multi-core system can be analyzed as a cluster on a chip, having the same performance problems and tuning possibilities and benefits as the same solutions proposed for NOWs and Multi-Cluster studies. Roughly, machines are cores, heterogeneous in some scenarios, interconnected by a network (Network On a Chip – NOC), heterogeneous in other scenarios. We can, for example, analyze a cluster of multi-core processors as a muti-cluster system, which can have the benefits of channel selection, grain composition/decomposition, channel based load distribution analysis and compute/communication overlap.

### 6.2.4 Moving to Cloud Computing

Grid technologies were the state-of-the-art model of systems for the last ten years. Nowadays, the Grid research products are used in production environments most in

government and academic areas. Indeed, application development could use an economic model based on commerce requirements. The lack of these economy indexed performance models could be seen in different industry initiative to sell on demand computing. Each seller has his service characteristics, most uses virtual machines and network use to compose products for users. These companies sell these products as Cloud Computing. The idea is that the client may buy virtual machines per hour of use and network bandwidth. When uses HPC applications and such systems, we could maximize the cost benefit of application execution, reducing its execution time, wide area network and adapt application execution within a multiple of time segments.

Imagine the following scenario. A user has an HPC application to execute on a Cloud Computing system at minimal cost possible. The cost of application execution is function of the number of virtual machines that it uses and the execution time and the network bandwidth exchange if those machines are located in different sites. Suppose that the virtual machine cost is sold per hour of use, without hour fragmentation. That allows us, for example in case of dynamic tuning, to tune the application execution time using different grain size in order to fit the machine utilization and execution time to multiple of one hour to get maximum cost efficiency. If the application, for example executes for ten hours and five minutes with ten machines have less cost if it executes for eleven hours and nine machines.

Our dynamic tuning environment GMATE could be used to interact to the Cloud system using some economic based tunlet that drives the activation and deactivation of virtual machines within an application execution exploring the performance and economic factor. It could be analyzed if a remote cluster of virtual machines cloud be activated considering the cost per task/results communications and its impact in total application execution time and cost. The uses of dynamic tuning have most benefits on those cases where runtime system conditions are the parameter that drives the application performance behavior.

# Bibliography


[1] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The Physiology of the Grid," in *Grid Computing - Making the Global Infrastructure a Reality*: John Wiley & Sons, Ltd, 2003, pp. 217-249.

[2] I. T. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid - Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications,* vol. 15, 2001.

[3] J. L. Hennessy, D. A. Patterson, and D. A. Patterson, *Computer architecture : a quantitative approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[4] M. J. Quinn, *Parallel programming in C with MPI and OpenMP*. Boston, USA: McGraw-Hill Higher Education, 2004.

[5] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, "Simple Object Access Protocol (SOAP)," W3C Recommendation24 June 2003.

[6] I. Foster and S. Tuecke, "Describing the elephant: the different faces of IT as service," *Queue,* vol. 3, pp. 26-29, 2005.

[7] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2 ed. San Francisco: Morgan Kauffman, 2003.

[8] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski, "A Grid Monitoring Architecture," GGF Performance Working Group, 2002.

[9] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," in *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, 2001, pp. 181--194.

[10] I. T. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *NPC*, 2005, pp. 2-13.

[11] M. Cierniak, M. J. Zaki, and W. Li, "Compile-Time Scheduling Algorithms for a Heterogeneous Network of Workstations," *The Computer Journal,* vol. 40, pp. 356-372, 1997.

[12] R. Jain, *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. New York: Wiley, 1991.

[13] J. K. Hollingsworth, J. Lumpp, and B. P. Miller, "Techniques for Performance Measurement of Parallel Programs," *Parallel Computers: Theory and Practice,* 1995.

[14] A. Morajko, "Dynamic Tuning of Parallel/Distributed Applications," in *Departament d'Arquitectura de Computadors i Sistemes Operatius*. vol. Phd Barcelona: Universitat Autonoma de Barcelona, 2004.

[15] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," *SC'02,* November 2002.

[16] M. Gerndt, K. Furlinger, and E. Kereku, "Periscope: Advanced techniques for performance analysis," in *Proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005)*, 2005, pp. 15–26.

[17] M. Gerndt, "Automatic performance analysis tools for the Grid," *Concurrency and Computation: Practice and Experience,* vol. 17, pp. 99-115, 2005.

[18] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," in *Cluster Computing*, 2002, pp. 237-246.

[19] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang, "Virtual Clusters for Grid Communities," 2006, pp. 513-520.

[20] I. Foster, *Designing and building parallel programs : concepts and tools for parallel software engineering*. Reading, Mass.: Addison-Wesley, 1995.

[21] L. Colombet and L. Desbat, "Speedup and efficiency of large-size applications on heterogeneous networks," *Theoretical Computer Science,* vol. 196, pp. 31--44, 1998.

[22] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems*: IEEE Computer Society Press, 1996.

[23] E. Argollo, A. Gaudiani, D. Rexachs, and E. Luque, "Tuning Application in a Multi-cluster Environment," in *Euro-Par 2006 Parallel Processing*, 2006, pp. 78-88.

[24] G. Costa, J. Jorba, A. Morajko, T. Margalef, and E. Luque, "Performance models for dynamic tuning of parallel applications on Computational Grids," in *Cluster Computing, 2008 IEEE International Conference on*, 2008, pp. 376-385.

[25] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *Concurrency and Computation: Practice and Experience,* vol. 17, pp. 323--356, 2005.

[26] "Portable Batch System Administrator Guide," Veridian Systems PBS Products Dept. 2672 Bayshore Parkway, Suite 810 Mountain View, CA 94043: Veridian Information Solutions, Inc., 2000.

[27] S. Dong, G. E. Karniadakes, and N. T. Karonis, "Cross-site computations on the TeraGrid," *Computing in Science \& Engineering [see also IEEE Computational Science and Engineering],* vol. 7, pp. 14--23, 2005.

[28] K. Yurkewicz, "Grid Gets the Blood Flowing," *Science Grid,* March 29 2006.

[29] B. Lowekamp, B. Tierney, L. Cottrell, R. H. Jones, T. Kielmann, and M. Swany, "A Hierarchy of Network Performance Characteristics for Grid Applications and Services," GFD-R-P.023 (Proposed Recommendation), 2004.

[30] J. Proti*c, M. Tomaševic, and V. Milutinovi*c, *Distributed shared memory : concepts and systems*. Los Alamitos, Calif.: IEEE Computer Society Press, 1998.

[31] H.-L. Truong, "Novel Techniques and Methods for Performance Measurement, Analysis and Monitoring of Cluster and Grid Applications," University of Innsbruck, 2005.

[32] M. Muller, M. Hess, and E. Gabriel, "Grid enabled MPI solutions for clusters," in *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, 2003, pp. 18--25.

[33] S. Genaud, M. Grunberg, and C. Mongenet, "Experiments in running a scientific MPI application on Grid’5000," *4th High Performance Grid Computing International Workshop, IPDPS conference proceedings*.

[34] S. Zanikolas and R. Sakellariou, "A taxonomy of grid monitoring systems," *FUTURE GENERATION COMPUTER SYSTEMS,* vol. 21, pp. 163--188, January 2005.

[35] H. L. Truong and T. Fahringer, "SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid," in *European Across Grids Conference*, 2004, pp. 202-211.

[36] A. W. Cooke, A. J. G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. J. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. A. Coghlan, S. Kenny, and D. O'Callaghan, "R-GMA: An Information Integration System for Grid Monitoring," in *CoopIS/DOA/ODBASE*, 2003, pp. 462-481.

[37] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing,* vol. 30, pp. 817-840, 2004.

[38] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, "MonALISA : A Distributed Monitoring Service Architecture," *CoRR,* vol. cs.DC/0306096, 2003.

[39] A. W. Cooke, A. J. G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S. M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. Oâ€™Callaghan, and J. Ryan, "The Relational Grid Monitoring

Architecture: Mediating Information about the Grid," *Journal of Grid Computing,* vol. 2, pp. 323--339, December 2004.

[40] Z. Balaton and G. Gombás, "Resource and Job Monitoring in the Grid," in *Euro-Par 2003 Parallel Processing*. vol. 2790: Springer Berlin / Heidelberg, 2003, pp. 404-411.

[41] A. Cooke, A. J. G. Gray, and W. Nutt, "Stream Integration Techniques for Grid Monitoring," in *Journal on Data Semantics II*, 2005, pp. 136--175.

[42] M. Gerndt, R. Wismuuller, Z. Balaton, G. Gombás, P. Kacsuk, Z. Námeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef, "Performance Tools for the Grid: State of the Art and Future," APART White Paper, 2004.

[43] "The Grid Laboratory Uniform Environment (GLUE).": http://www.cnaf.infn.it/sergio/datatag/glue/index.htm.

[44] X. Zhang, J. L. Freschl, and J. M. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems," in *HPDC*, 2003, pp. 270-282.

[45] I. LeGrand and H. Newman, "Monalisa: An Agent Based, Dynamic Service System To Monitor, Control And Optimize Grid Based Applications," *Computingin High Energy Physics,* 2004.

[46] R. Wolski, "Dynamically forecasting network performance using the Network Weather Service," *Cluster Computing,* vol. 1, pp. 119--132, March 1998.

[47] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications,* vol. 20, pp. 287-311, 2006.

[48] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of High Performance Computing Applications,* 2000.

[49] B. Mohr and F. Wolf, "KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs," *Lecture Notes in Computer Science,* vol. 2790, pp. 1301 - 1304, Jan 2003.

[50] R. H. Pesch, J. M. Osier, and C. Support, "The gnu Binary Utilities," Free Software Foundation, Inc, Manual.

[51] R. L. James and S. Eric, "EEL: machine-independent executable editing," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* La Jolla, California, United States: ACM Press, 1995.

[52] D. M. Pase, "Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide," IBM CorporationJuly 13 1998.

[53] H.-L. Truong, T. Fahringer, and S. Dustdar, "Dynamic Instrumentation, Performance Monitoring and Analysis of Grid Scientific Workflows," *Journal of Grid Computing,* vol. 3, pp. 1--18, June 2005.

[54] 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,* vol. 28, pp. 37--46, 1995.

[55] J. K. Hollingsworth, R. B. Irvin, and B. P. Miller, "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool," in *PPOPP*, 1991, pp. 189-200.

[56] J. K. Hollingsworth, B. P. Miller, M. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," in *International Conference on Parallel Architectures and Compilation Techniques*, 1997, pp. 201 - 212.

[57] A. Morajko, O. Morajko, T. Margalef, and E. Luque, "MATE : Dynamic Performance Tuning Environment," *LNCS,* vol. 3149, pp. 98-107, 2004.

[58] G. Costa, A. Morajko, T. Margalef, and E. Luque, "Automatic Tuning in Computational Grids," in *Applied Parallel Computing. State of the Art in Scientific Computing*, 2007, pp. 381-389.

[59] E. Argollo, "Performance prediction and tuning in a multi-cluster environment," in *Departament d'Arquitectura de Computadors i Sistemes Operatius*. vol. Phd Barcelona: Universitat Autonoma de Barcelona, 2006.

[60] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A.

Su, and D. Zagorodnov, "Adaptive computing on the Grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, pp. 369--382, 2003.

[61] E. César, J. G. Mesa, J. Sorribes, and E. Luque, "Modeling Master-Worker Applications in POETRIES," *HIPS 2004,* pp. 22-30, 2004.

[62] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems," *Cluster Computing,* vol. 6, pp. 7-17, 2003.

[63] B. Javadi, M. K. Akbari, and J. H. Abawajy, "Performance analysis of heterogeneous multi-cluster systems," *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on,* pp. 493-500, 2005.

[64] E. Heymann, M. A. Senar, E. Luque, and M. Livny, "Adaptive scheduling for master-worker applications on the computational grid," *Grid Computing-GRID,* pp. 214-227, 2000.

[65] A. Morajko, P. Caymes, T. Margalef, and E. Luque, "Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications," in *Computational Science – ICCS 2005*, 2005, pp. 132-139.

[66] P. Caymes-Scutari, A. Morajko, E. César, G. Costa, J. G. Mesa, T. Margalef, J. Sorribes, and E. Luque, "Development and Tuning Framework of Master/Worker Applications," in *Automatic Performance Analysis*, Dagstuhl, Germany, 2006.

[67] J. Dongarra, J.-F. Pineau, Y. Robert, and F. Vivien, "Matrix product on heterogeneous master-worker platforms," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* Salt Lake City, UT, USA: ACM, 2008.

[68] J. F. Pineau, Y. Robert, F. Vivien, Z. Shi, and J. Dongarra, "Revisiting Matrix Product on Master-Worker Platforms," *Research Report,* vol. 39, 2006.

[69] M. H. MacDougall, *Simulating computer systems : techniques and tools*. Cambridge, Mass.: MIT Press, 1987.

[70] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled implementation of the Message Passing Interface," *Journal Of Parallel And Distributed Computing,* vol. 63, pp. 551--563, May 2003.

[71] E. Argollo, D. Rexachs, F. Tinetti, and E. Luque, "Efficient Execution of Scientific Computation on Geographically Distributed Clusters," *PARA,* pp. 691-698, 2004.

[72] E. Argollo, J. R. d. Souza, D. Rexachs, and E. Luque, "Efficient Execution on Long-Distance Geographically Distributed Dedicated Clusters," *PVM/MPI 2004,* pp. 311-318 2004.

[73] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. E. Hofman, "Optimizing Parallel Applications for Wide- Area Clusters," *IPPS/SPDP,* pp. 784-790, 1998.

[74] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems : concepts and design*, 2nd ed. Harlow, England ; Reading, Mass.: Addison-Wesley, 1994.

[75] D. Mills, "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," University of Delaware, RFC 20301996.

[76] R. L. Ribler, H. Simitci, and D. A. Reed, "The Autopilot performance-directed adaptive control system," *Future Gener. Comput. Syst.,* vol. 18, pp. 175--187, 2001.

[77] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed, "Autopilot: Adaptive Control of Distributed Applications," in *HPDC*, 1998, pp. 172-179.

[78] P. Caymes-Scutari, A. Morajko, E. Cesar, J. Mesa, G. Costa, T. Margalef, J. Sorribes, and E. Luque, "Entorno de Desarrollo y Sintonizacion de Aplicaciones Master/Worker," in *CACIC - Workshop de Procesamiento Distribuido y Paralelo (WPDP)*, Argentina, 2005.

[79] G. Costa, A. Morajko, T. Margalef, and E. Luque, "Automatic Tuning in Computational Grids," in *Workshop On State-Of-The-Art In Scientific And Parallel Computing* Umeå, Sweden: June 18-21, 2006.

[80] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link, "Globus eXtensible Input/Output System (XIO): A Protocol Independent IO System for the Grid," *ipdps,* p. 179a, 2005.

[81] G. Team, "The Dynamically-Updated Request Online Coallocator ". vol. May 2007: http://www.globus.org/toolkit/docs/2.4/duroc/, 2007.

[82] J. Nagle, "Congestion control in IP/TCP internetworks," Ford Aerospace and Communications Corporation, RFC 8966 January 1984.

[83] W. R. Stevens, *UNIX Network Programming, Volume 2: Interprocess Communication, 2nd edition*. New Jersey: Prentice Hall PTR, 1998.

[84] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2003, p. 21.

[85] M. Gerndt and E. Kereku, "Selective Instrumentation and Monitoring," *International Workshop on Compilers for Parallel Computers,* 2004.

[86] Intel, "Intel MPI benchmarks (formally known as Pallas MPI Benchmarks)."

[87] A. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," *Proceedings of ClusterWorld,* vol. 2003, 2003.

[88] A. J. Dorta, J. M. Badía, E. S. Quintana, and F. de Sande, "Implementing OpenMP for Clusters on Top of MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2005, pp. 148-155.

[89] G. Jost, H. Jin, D. a. Mey, and F. F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster," NASA, NAS Technical ReportNovember 2003.

[90] B. M. Chapman, L. Huang, H. Jin, G. Jost, and B. R. d. Supinski, "Support for Flexibility and User Control of Worksharing in OpenMP," NASA, NAS Technical ReportOctober 2005.

[91] D. Loveman, "High performance fortran," *IEEE [see also IEEE Concurrency] Parallel & Distributed Technology: Systems & Applications,* vol. 1, pp. 25-42, 1993.

[92] C. Barton, C. Cascaval, and J. Amaral, "A Characterization of Shared Data Access Patterns in UPC Programs," *Lecture Notes in Computer Science,* vol. 4382, p. 111, 2007.

[93] L. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," *ACM Sigplan Notices,* vol. 28, pp. 91-108, 1993.

# Appendix

## A. ClusterSim – a Multi-Cluster Simulator

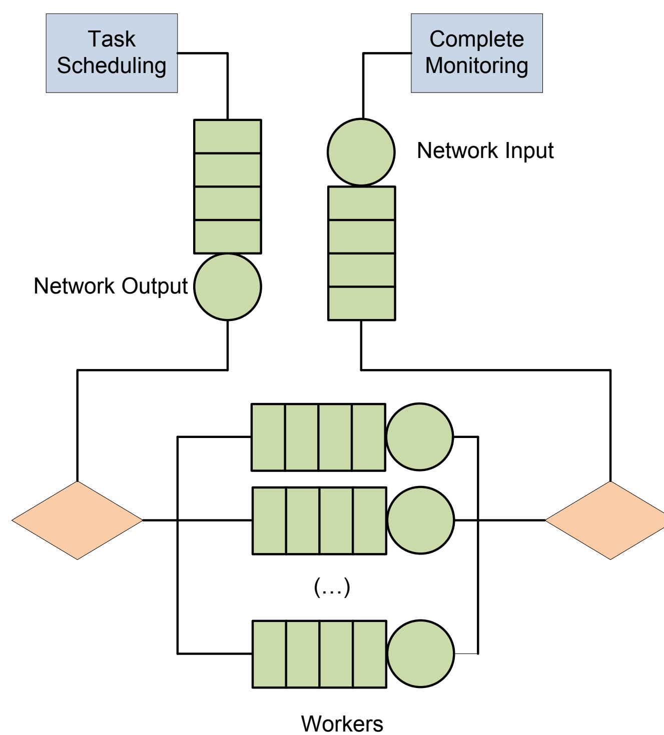



Figure 72 – Queue system used to simulate the ClusterSim Cluster Simulator

Once we found out how a Master-Worker application behaves on some sample scenario, we choose to build an approximation of such system in a discrete event simulator. The

implementation is done over the SMPL library and has events for three queue types basically. The internal load structure and the internal behavior were built to mimic the GMWAT application template.

The three queue types are, Network Input, Network Output and Workers. The SMPL provides an API to represent queues called facilities. Each facility may have one or more servers. All facilities used have one server. The item that is queued is called client on SMPL semantic. For each conceptual queues presented on Figure 72, we create three distinct events:

- Arrival – when a client arrives on queue.
- Request – when a client probes for server occupation.
- Release – when a client exist from queue.

The implemented event and its semantics are:

- EVT_WORK_GENERATION – controls the moment where the master schedules the work.
- EVT_NET_OUT_ARRIVAL – Work arrival on queue Network Output.
- EVT_NET_OUT_REQUEST – Server probe for queue Network Output.
- EVT_NET_OUT_RELEASE – Work transmitted using queue Network Output.
- EVT_NET_IN_ARRIVAL – Work arrival on queue Network Input.
- EVT_NET_IN_REQUEST – Server probe for queue Network Input.
- EVT_NET_IN_RELEASE – Work transmitted using queue Network Input.
- EVT_WORKER_PROC_ARRIVAL – Arrival of a work to be processor on some worker identified by client.
- EVT_WORKER_PROC_REQUEST – Server probe for work to be processed.
- EVT_WORKER_PROC_RELEASE – Work processed on worker identified by client.
- EVT_WORKER_QUEUE_RELEASE – latency displaced from EVT_NET_IN_ARRIVAL to mimic the framework code.
- EVT_STARTUP_END – used to graph MW phases.
- EVT_FINALIZATION_START – used to graph MW phases.
- EVT_FINALIZATION_PRE_START – used to graph MW phases.

- EVT_DT_EVENT_RECEIVED – event to identify that a dynamic tuning event trace is received by tuning tool.
- EVT_DT_ACTION_RECEIVED – event to identify that a dynamic tuning action change is received by the application such as grain size change.
- EVT_DT_INIT_LOOP – initiates the tuning process.
- EVT_ACTIVATE_WORKER – event to ask for worker activation.
- EVT_DEACTIVATE_WORKER – event to ask for worker deactivation.

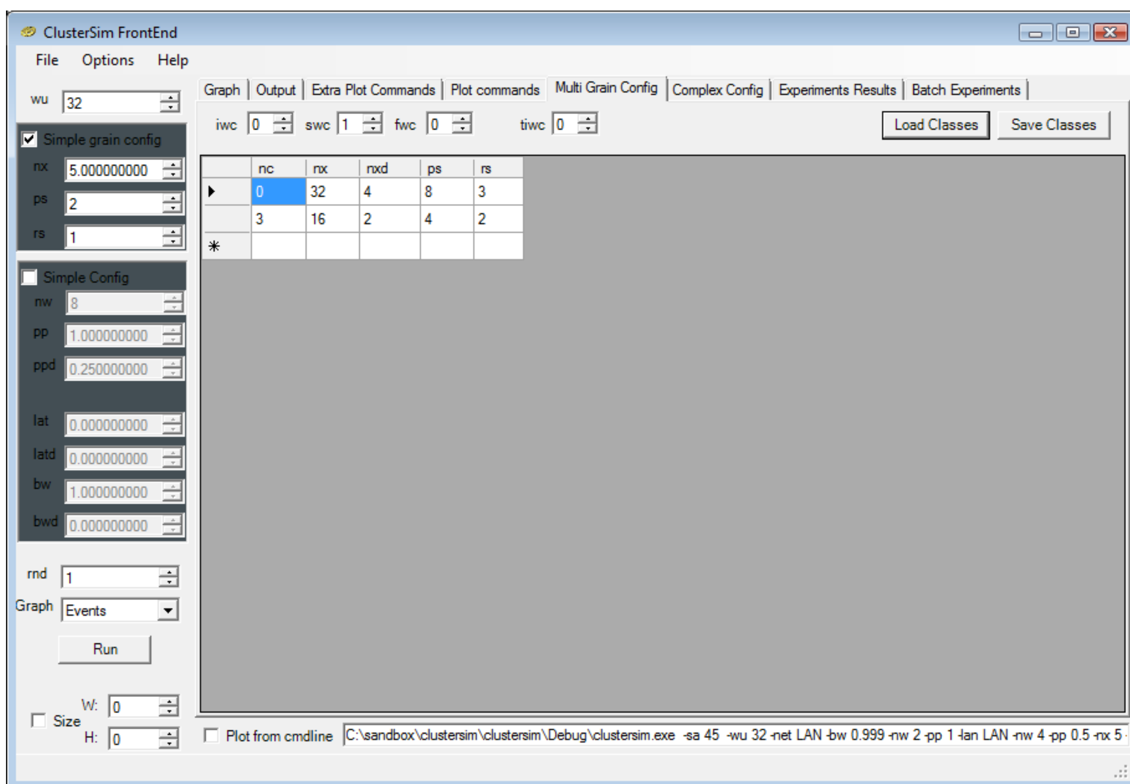



Figure 73 – Screenshot for ClusterSim front end that enables the configuration of multiple grain sizes and the multiplicity of composition/decomposition.

Figure 73 presents the GUI that facilitates the configuration of different data reuse scenarios and task composition and decomposition. The execution of SMPL engine code generates the events of task execution that are used to plot the screen presented by Figure 74. By such screen is possible to debug what task is assigned to what worker and how the task composition/decomposition works. The idea is not to debug large number

of workers but to identify on given different configurations how the task assignation and grain composition and decomposition affect the application.

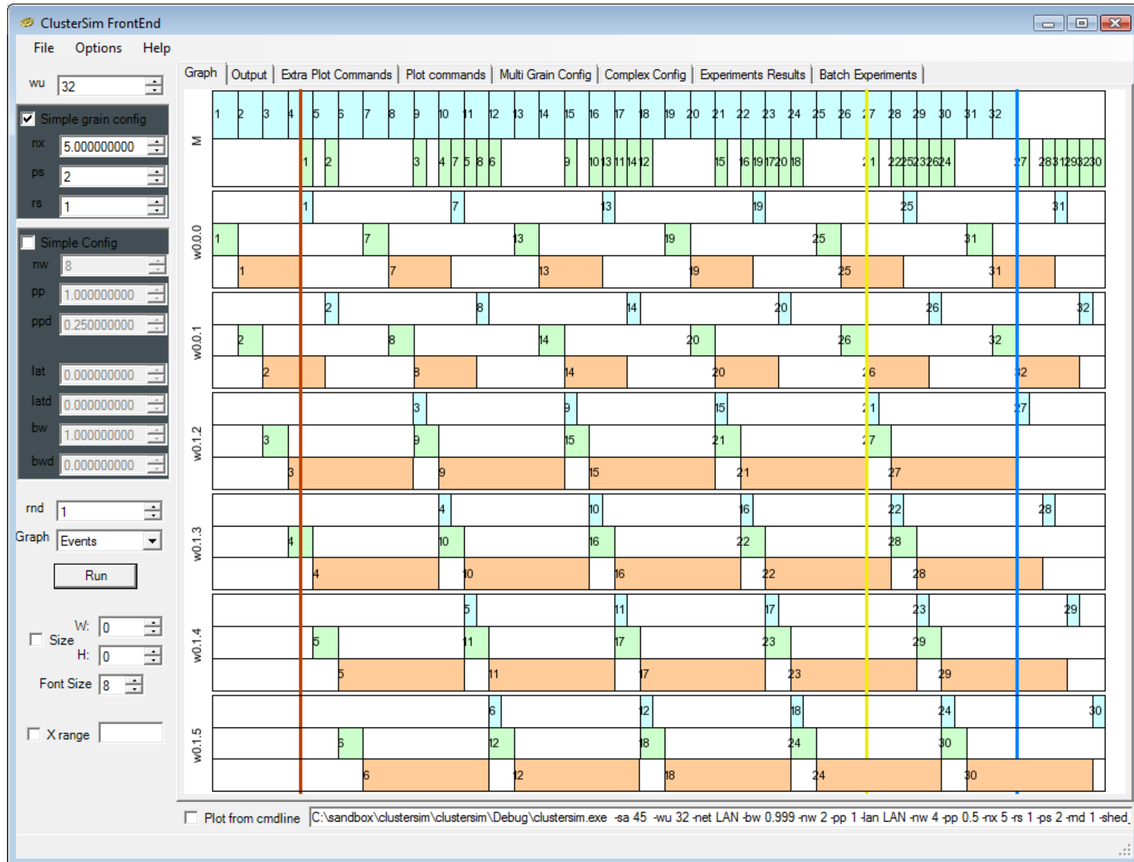



Figure 74 – Output of events from ClusterSim plotted using gnuplot. The numbers in the bars are the task numbers. The blue bars are sends and the green bars are receives. The orange bars are processing events from workers.

The other view that allows a quick view of the application parallelism degree is the busy view presented on Figure 75. In such view we can analyzes the real use of the assigned workers. The execution from Figure 74 and Figure 75 are related to the same problem. The application has assigned six workers but uses only five.

Figure 76 presents the view of the workers, the network topology and the link properties. From that is possible to configure LAN and WAN parameters and also network contexts. Network contexts allows for configurations where a node access a WAN link over a NAT, for example. In such case consumes the WAN link using the LAN link.

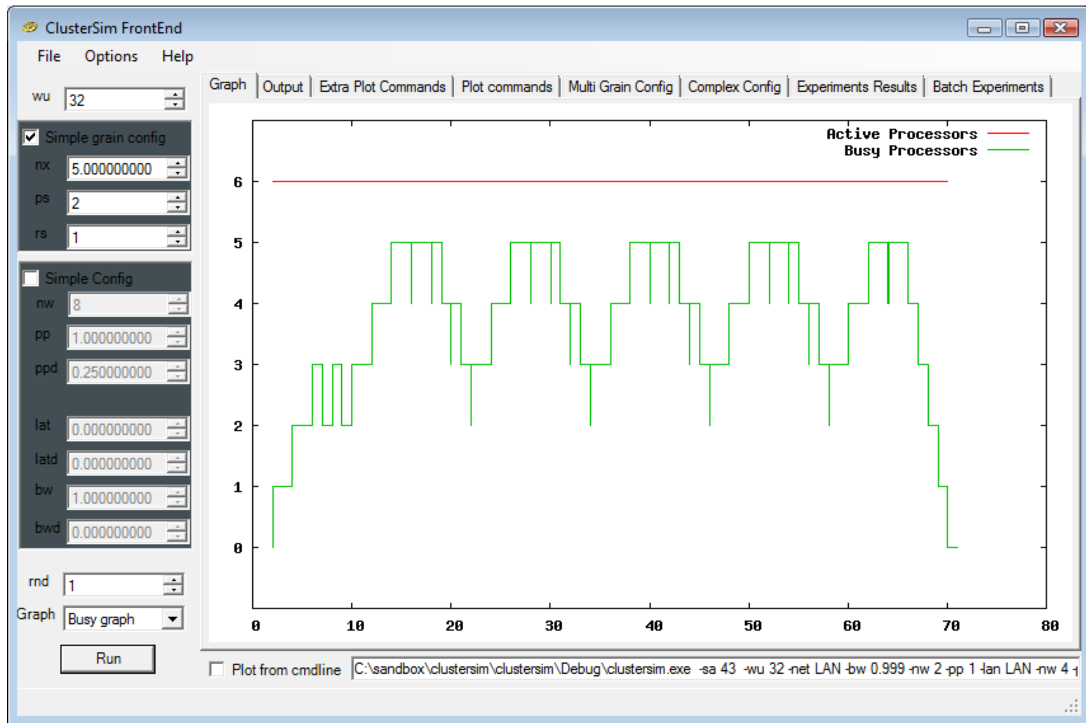



Figure 75 – Presents the amount of workers busy by time in application execution.

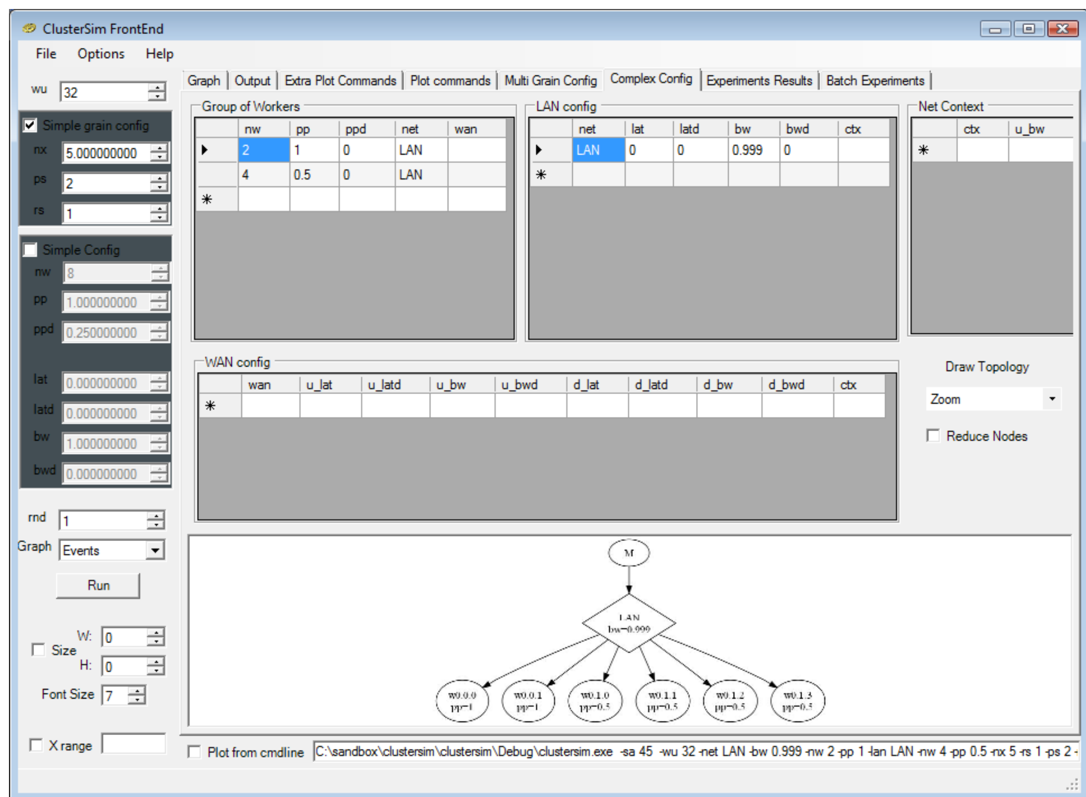



Figure 76 – GUI for complex topology configuration.

## B. GMWAT – a Hierarchical Master/Worker Application Template with Support for Dynamic Grain Size Selection

When we thought about working with change in application compute to communication ratio, we take a parallel of o Master-Worker matrix multiplication program to identify what characteristics could be generalized in order to facilitate the development of application with such characteristics. First we choose the Master-Worker paradigm. The implementation should have uniform tasks sizes in computation and communication. The common characteristic that makes an application to change the compute to communication ratio is the data reuse among the generated tasks.

We search for available implementation like Skeletons, AppLeS and Quiron but none of them provides the requirements to be used for dynamic tuning of number of workers and grain size. The AppLeS does not fix the size of communication. The Skeletons does not allow for working with more than one input and output data sizes for tasks. And the Quiron have static structures for the parallel machine topology and data problem mapping.

To facilitate the development of parallel applications suitable for dynamic tuning of grain size and number of workers in computational Grids we decide to create an application template that can be easily used to fix application behavior to such characteristics. The main requirements are:

- Tasks should have a strong type and uniform.
- Task should be composed by data segments or chunks.
- Task can be decomposed to change grain size.
- Should support the following process roles:
  - Master
  - Submaster
  - Worker
  - Communication Manager
- Can add and remove workers.
- Support of iterative applications.
- Can change the grain size within or between iteration.

To separate the computation from communication we use two classes *ComputeProcessor* and *MPIHandle*. The *ComputeProcessor* manage the task generation, processing and routing, and the *MPIHandle* handles the asynchronous transmission from a *ComputeProcessor* and other processors. The Figure 77 presents the iteration between the *ComputeProcessor* and *MPIHandler* instances. These classes follow the singleton pattern on application process. The interface between the two classes follows the queue semantics with the methods *'give'*, *'notifyInput'*, *'notifyOutput'* and *'report'* send.

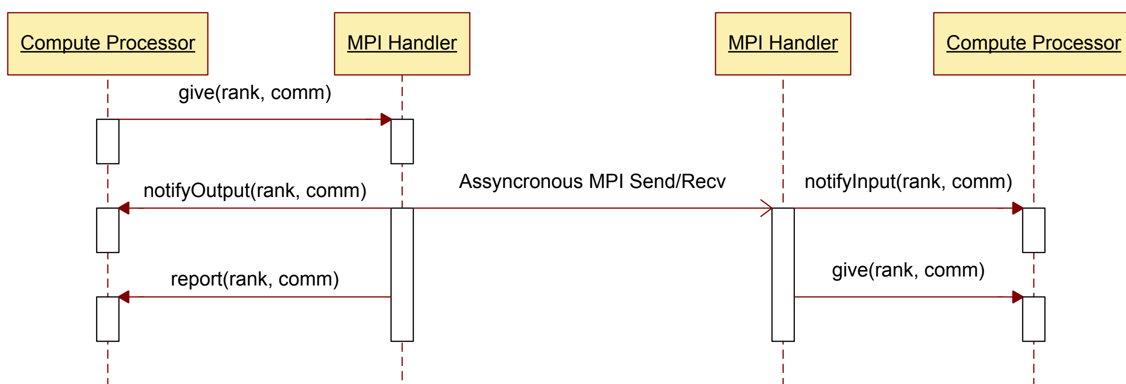



Figure 77 – Sequence diagram of the API exposed for communication between the classes *MPIHandler* and *ComputeProcessor*. It also presents relevant template function that can be used to measure time spent in the communication process.

The ComputeProcessor class is an abstract meta-class placeholder for more elaborate processors. The implemented specialization and its responsibilities are:

- Master – loads problem data from disk and schedule the tasks to other processors. It controls the number of iterations, the grain size and the amount of work for each iteration.
- SubMaster – waits tasks from some processor and schedule the load from those tasks to other processes. It is capable of break tasks to schedule finer grains to other processors.
- Communication Manager – just acts as a proxy among two processes A and B. If the task received is from A, it is forwarded to B, and if it is from B, it is forwarded to A.

- Worker – waits tasks from a processor, execute its load and returns the result task.

Figure 78 presents the relationship among these processes roles. The *MasterProcess*, *WorkerProcess* and *CommunicationManager* extend the *ComputeProcessor*. The *SubMaster* extends the *MasterProcessor* and *Workerprocessor*.

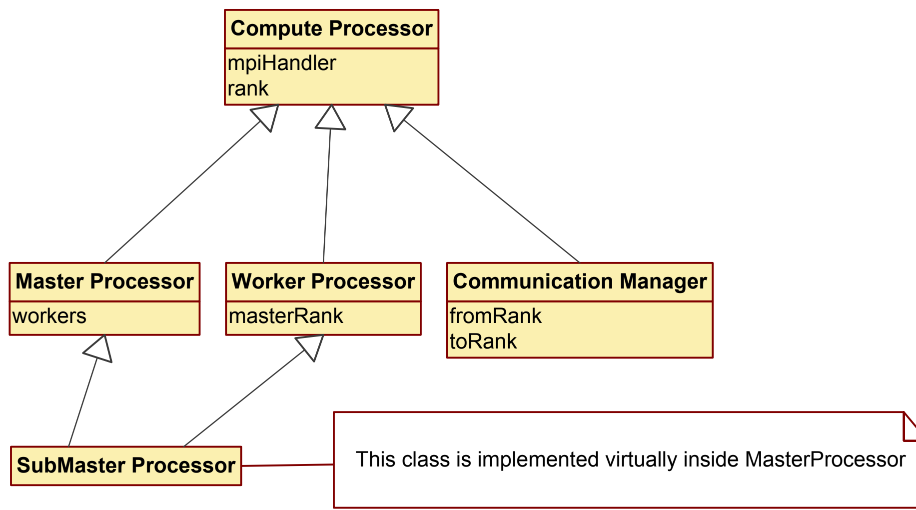



Figure 78 – Class diagram presenting the internal concepts of the process roles inside the framework.

The *MasterProcessors* groups the workers in clusters[7]. The master schedules the tasks independently for each cluster. Each cluster has its grain size, a number of workers and parallel input and output threads for sending and receiving tasks using the *MPIHandler* interface.

The MPIHandler internally is composed of Channels. Channels are threads classes from input and output data using the MPI library. We use the capability of message tags to differ messages from different channels. For example, if the master has two clusters, the MPI library can have two sends in parallel to two different processes, one from each cluster.

[7] Here the cluster word refers to a set of workers with same properties. For example, to group workers that shares a network link.

The tasks are the basic unit that a *ComputeProcessor* needs for processing. The task inside the template is implemented by class Work. Each work has a *WorkIndex* information which contains the grain size *gi* and the work sequence number *wn*. The load of a Work is represented by the class *WorkData*. *WorkData* instances from same *gi* are composed by fixed instances of *DataChunks*. Figure 79 presents these concepts and the cardinality of those compositions.

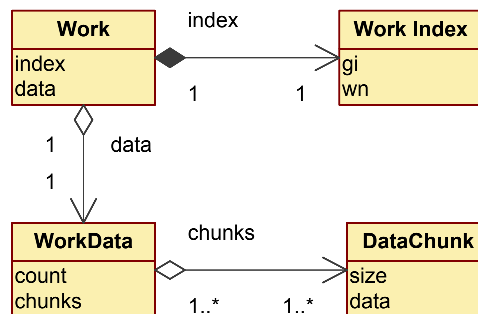



Figure 79 – Class diagram presenting the internal concepts of the load inside the framework.

The structure of *WorkData* in *DataChunks* allows for memory saving when we have reuse. If a *DataChunk* is used in many *WorkData* instances, there is no need of data redundancy inside some process. Note that we follow the concept of stateless worker. The *DataChunk* allows labeling. That may be use for *DataChunk* caching within processors.

The template interacts with non-template code or user code by a simple C API. The API is used for input data reading and output data write initialization and finalization, iteration management, task creation, composition, decomposition and processing and grain size management.

When the template code starts, it queries the user code about the problem load parameters in terms of number of work units by iteration and the number of iterations. It also query how the task composition/decomposition.

The Figure 80 presents a sequence diagram of a complete execution where the master sends a task and receives its results. The template code probes how the application

supports the different grain size by calling the function *'user_canBreakDown'* for the range of grain size values.

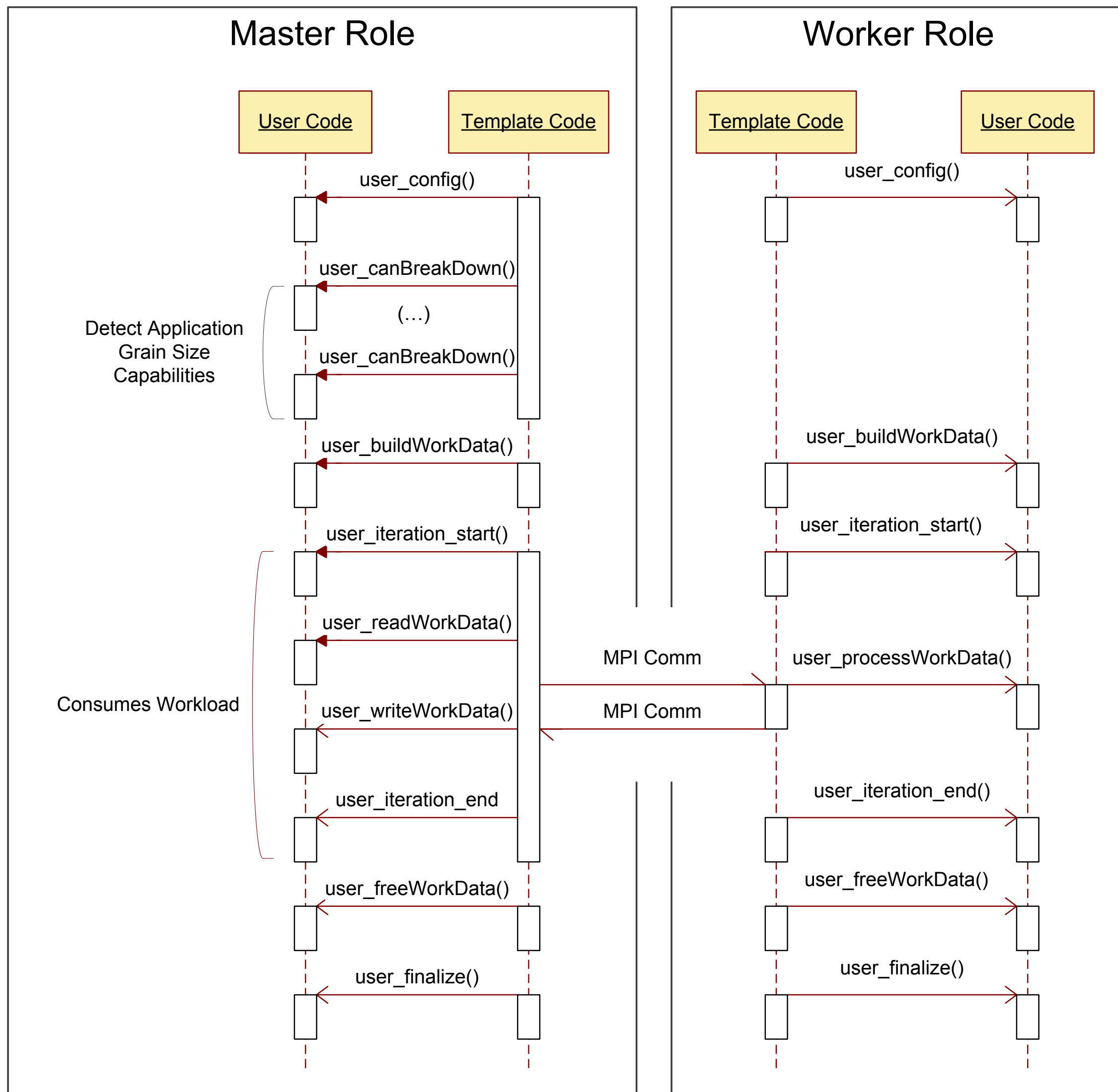



Figure 80 – Sequence diagram presenting the iteration among Master and Worker processes including the sequence of user function calling.

The template code accepts configuration parameters form which is possible to specify the following directives:

- Intial Grain Size – what grain size the clusters will work on.
- Base Grain Size – minimal grain size the master reads and writes.
- Clusters – the configuration topology as presented in Figure 81.

- Channels – the configuration of different channels for overlapped communications among different processors.

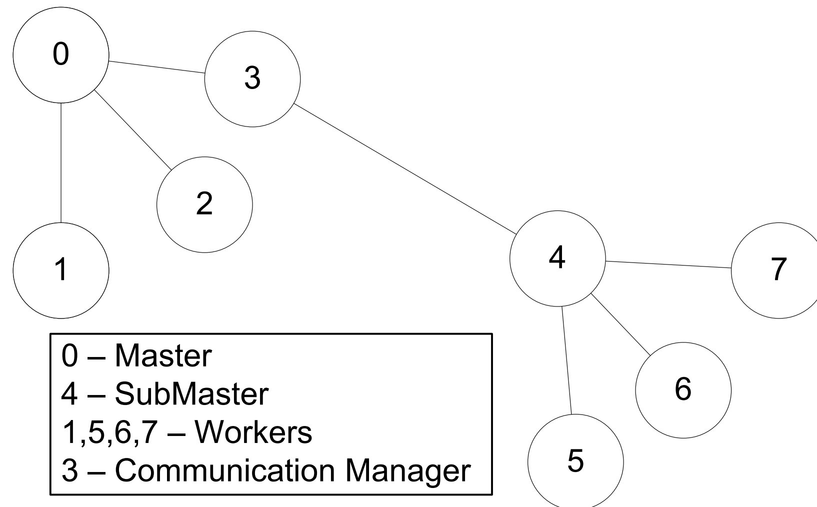



Figure 81 – It presents a multi-cluster configuration within a parallel machine with 8 processes.

An example of multi-cluster complex configuration is presented on Figure 81. Such configuration only requires an input parameter that indicates the roles for the processes. For this example, if the parameter '--gmat-clusters' receives the value '0_1-3,3.4_5-7" the template executes using the topology presented on Figure 81.