POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
CORSO DI LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

APPLICATION AUTOTUNING ON PARALLEL DISTRIBUTED ARCHITECTURES

M.Sc. Thesis of:
Monica Cecere

Student ID:
836391

Advisor:
Prof. Cristina Silvano

Co-Advisors:
Eng. Davide Gadioli
Prof. Gianluca Palermo

Academic year 2015/2016

A tutti quelli che mi hanno sostenuto.

Abstract

NOWADAYS, it is spreading more and more the use of applications that have a huge amount of data in input with properties not known at design stage. Computing systems most commonly used to perform this type of applications take advantage of the High Performance Computing technology, a very powerful and high speed technology. There are already several strategies to be applied in order to reduce consumption of these supercomputing systems, such as the approximate computing or the use of a autotuner to monitor the consumption of application and dynamically makes decisions about management of its resources.

The main objective of this thesis concerns the research and development of an autotuning strategy for the execution of a program, to be applied in HPC field. We want to find trade-off solutions between quality of service and performances in this area, which is a problem that affects both scientific and industrial environments. In particular, we want to try to contain within a specified value, the execution time or the power consumption used by the application. The purpose consists in being able to realize this trade-off between quality of service and time-to-solution or energy-to-solution in an automatic and general way, completely independent from the functional characteristics of the application, and apply it to applications with execution time and energy consumption not predictable in advance, but manageable and adaptable dynamically.

The framework developed, Ipazia, realizes the proposed ideas for solving this particular problem, focusing on the joint use of approximate computing and autotuning techniques in a parallel and distributed environment.

Sommario

Al giorno d'oggi si sta diffondendo sempre più l'utilizzo di applicazioni caratterizzate da un enorme ammontare di dati in input con proprietà non note in fase di progetto. I sistemi di calcolo maggiormente usati per eseguire questo tipo di applicazioni sfruttano la tecnologia High Performance Computing, una tecnologia molto potente e con elevate prestazioni. Esistono già diverse strategie da applicare per ridurre i consumi di questi sistemi di supercalcolo, come il calcolo approssimato o l'uso di un autotuner per monitorare i consumi di un'applicazione e prendere decisioni dinamicamente sulla sua gestione di risorse.

L'obiettivo principale di questa tesi riguarda la ricerca e lo sviluppo di una nuova strategia di autotuning per l'esecuzione di un programma, da applicare in ambito HPC. Si vogliono trovare solutioni trade off tra la qualità di servizio e le prestazioni in questo ambito, che è un problema che affligge sia gli ambienti scientifici che industriali. In particolare, si vuole cercare di contenere entro un determinato valore, il tempo di esecuzione o la potenza utilizzata dall'applicazione. Lo scopo consiste nel riuscire a realizzare questo trade off tra la qualità di servizio e il tempo di soluzione o l'energia di soluzione in modo automatico e generale, completamente indipendente dalle caratteristiche funzionali dell' applicazione, ed utilizzarlo per applicazioni con prestazioni di tempo e di consumo di energia non prevedibili a priori, ma gestibili ed adattabili in modo dinamico.

Il framework sviluppato, Ipazia, concretizza le idee proposte per la risoluzione di questo particolare problema, concentrandosi sull'utilizzo congiunto di tecniche di calcolo approssimato e autotuning in ambito parallelo e distribuito.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Problem and Motivation

High Performance Computing is the practice of aggregating computing systems in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large and complex problems in science, engineering, or business. Nowadays, as demand for processing power and speed grows, HPC will likely interest businesses of all sizes, particularly for transaction processing, data warehouses and high performance data analytics. All modern supercomputer architectures depend heavily on parallelism, and the number of CPUs in large-scale supercomputers increases steadily. HPC systems operate above a TeraFLOP or 1012 floating-point operations per seconds and probably they will reach the Exascale level in the following years. Thus the use of parallel processing and heterogeneous distributed architectures is strictly needed in order to obtain such high performance in an efficient and quick way.

Optimization performance problem affects many computer science and industrial fields. Since HPC systems are very powerful structures, they are also very consuming and many current researches are investigating this problem, proposing different types of solutions. Nevertheless, this problem is still far from being solved. The main optimization areas regard the reduction in the execution time and the power consumption, while maintaining the highest quality of service

possible.

The difficulty in solving this problem arises from the fact that many HPC applications have unpredictable behavior as it strongly depends not only on the load of the input data but also on the characteristics of the inputs themselves. However, in an HPC application that has to process a very huge amount of data, it is unfeasible to characterize it by reading all the incoming data before computation has started. Just thinking about the capacity of HPC to handle and analyze massive amounts of data at high speed, that can take months using normal computers, in days or even minutes, and of the ability of HPC systems to run data-intensive modeling and simulation problems at large scale, at higher resolution and with more elements.

The purpose of this thesis is to propose a methodology to solve the problem of unpredictability, at design time, of the characteristics of the data in input, in HPC applications.

We want to find a way to characterize the workload of an application before it has started computing it. We want to make target HPC applications flexible with respect to the characteristics of their workload and the possible introduction of little noise, and monitor their execution mostly in terms of time and power consumption.

Our objective consists of giving our contribution to solve the optimization performance problem of HPC applications. We address the problem of both performance modeling and performance automatic management. Target applications must be executed maintaining a certain quality of service. Having the quantity of one of these performance as an input, we would like to make the application tunable so that this given value of performance will be satisfied, whatever the data features of inputs are, and tolerating changes due to the environment. In particular, we address this problem for applications working in a parallel and distributed environment, which is the most common field for HPC programs.

Approximate programming is a technique used in contexts in which it is no more possible to produce precise results in short times or in critical situations. It can also be applied in the HPC field as a mechanism to reduce the execution time or power consumption by introducing some level of error. We want to further investigate this mechanism and apply it in the context described above. Our purpose is to make the application flexible with respect to performance, such as latency and power consumption, acting directly on this quantity of error to introduced.

Experimental results show the behavior of our framework under different levels of noise introduced. In perfect conditions and with low noise level our framework succeeded in maintaining the predefined performance constraint.

1.2 Contribution

We develop a methodology for solving the problem of the unpredictable behavior of HPC applications. In particular, the methodology proposed tries to solve the problem by suggesting a way to obtain an estimate of the distribution of the data in input without having to read them all. Through the Montecarlo sampling technique we provide a method for constructing the estimate distribution of the data in input in terms of probability density functions of each feature that characterizes each input.

Autotuning is a technique well used in order to satisfy quality of service level reducing computational costs in terms of latency and power consumption. A dynamic autotuner leverages application behavior by changing the values of dynamic knobs according to application requirements. In order to solve the optimization problem, in this thesis we investigate the use of the autotuning in a proactive way, that is, we apply autotuning techniques in programs that are not stream-based, and before the actually computation has started. We also use this mechanism in conjunction with approximate computing techniques, and we study the behavior of a tunable and approximate application controlled by an autotuner, in a parallel and distributed environment.

Regarding the performance modeling problem, we provide a mechanism to automatically construct models to predict the performance trend of the application, based on the accuracy level and the characteristics of data in input. Given a performance constraint in terms of time-to-solution, our approach is able to execute a parallel application within a predefined time, managing to get the highest possible quality of service with respect to the time-to-solution given.

We also evaluate different search strategies for better calculating the values the application needs for satisfying the constraints to which it is subjected. We take into account cases where it is not possible to use an exhaustive search solution because the search space is too large, or it is really unfeasible. This work provides a heuristic search strategy that can solve the search problem in less time that the exhaustive search method.

We develop a strategy to make a distributed application flexible and tolerant with respect to its input features and little fluctuations of noise. Therefore, we introduce a control mechanism that periodically updates the information for managing the application behavior, at run-time.

The parallel and distributed environment we address, has no shared memory. Therefore, we select the Message Parsing Interface library as a possible communication solution for achieving our goals. Through this library we can communicate with the different processes of the target application and send and receive

information. To make more effective the use of MPI library for the solution we want to developed, we evaluate several communication strategies, concerning speed, feasibility and overheads introduced by this mechanism.

Ipazia is the name of the framework developed in order to give our practical contribution in the directions described above. We evaluate its behavior using the mARGOt autotuning framework, being developed at the System Architecture group of Politecnico di Milano.

1.3 Organization of the Thesis

This thesis is structured as follow: in Chapter 2 there is the description of the background on which this work is based on. It introduces High Performance Computing concepts, parallel distributed programming and performance modeling methods.

Chapter 3 gives an overview on the state-of-the-art regarding methodologies and techniques developed in this thesis, focusing on the description of the mARGOt autotuner and the last works on proactive control of approximate applications.

In Chapter 4 we describe the methodology proposed step by step and the logic structure developed for building Ipazia framework.

In Chapter 5 there is the detailed description of the framework implemented, in terms of components and behavior.

Chapter 6 contains all the experiments we made in order to validate our proposed methodology and to gather information on the framework behavior.

Chapter 7 contains the conclusions of this work, with the description of both benefits and limitations, and the list of possible future works that can be done in order to improve and expand this thesis.

Finally, there is the bibliography.

CHAPTER 2

Background

2.1 Introduction

This chapter introduces the most important concepts this thesis is based on. First, there is a description about High Performance Computing and what are its main issues, then there is a section about parallel programming and in particular the use of Message Passing Interface as parallel library. The last section is about modeling techniques with the R software [13].

2.2 Definition of High Performance Computing Systems

High Performance Computing (HPC) most generally refers to the practice of aggregating computing systems in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large and complex problems in science, engineering, or business [14]. Nowadays, as demand for processing power and speed grows, HPC will likely interest businesses of all sizes, particularly for transaction processing, data warehouses and high performance data analytics. HPC systems operate above a TeraFLOP or 1012 floating-point operations per seconds and probably they will reach the Exascale level in the following years. Thus the use of parallel processing and heterogeneous distributed architectures is strictly needed in order to

obtain such high performance in an efficient, reliable and quick way [3].

Cloud Computing with its recent and rapid expansion and development has grabbed the attention of HPC users and developers in recent years. Cloud Computing attempts to provide HPC-as-a-Service exactly like other forms of services currently available in the Cloud such as Software-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-Service. HPC users may benefit from the Cloud in different angles such as scalability, resources being on-demand, fast, and inexpensive. On the other hand, moving HPC applications have a set of challenges too. One of them is the need of knowing how much energy-time an HPC application will consume computing a certain amount of data. Currently, performance modeling of applications on all levels of a system's architecture is of utmost importance, and it is an indispensable guiding principle in HPC [31]. However many HPC applications process a so huge amount of data that it is very difficult to truly characterizing the application behavior in terms of quality of service and power consumption trade-off without having a previous full knowledge of all features of input data, which could not be obtained in the majority of cases. So researches are moving towards the study of HPC application online autotuning [46] and one of the main challenges is focused on how to autotune a distributed heterogeneous application with no shared memory, since, this is the most common architectural configuration used in HPC systems.

2.2.1 Heterogeneous Multicore Architecture

A heterogeneous architecture can provide significantly higher performance than a homogeneous chip multiprocessor and can also be beneficial in systems with multithreaded cores [34]. Figure 2.1 shows a heterogeneous multicore architecture.

The Green500 list [12] of the most energy-efficient supercomputers demonstrates that heterogeneous systems are better energy efficient than homogeneous systems.

The top two systems of the Green500 list of November 2016, DGX SATURNV and Piz Daint, both use NVIDIA's P100 Tesla GPU, which exhibits excellent energy efficiency, and they both have heterogeneous architectures. They have a rating of 9.46 and 7.45 GigaFLOPs per Watt, respectively.

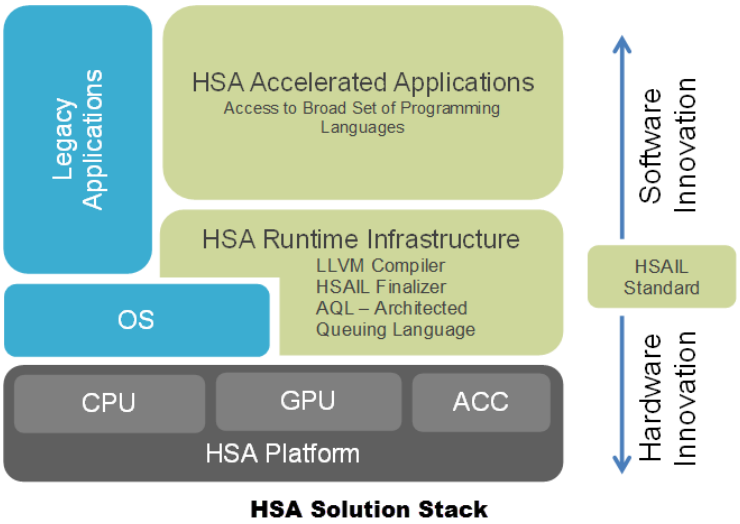

Figure 2.1: *Heterogeneous architecture.*

2.2.2 Distributed Architecture

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors. Another basic aspect of distributed computing architecture is the method of communicating and coordinating among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship.

Reasons for using distributed systems and distributed computing may include:

- The nature of an application may require the use of a communication network that connects several computers. For example data produced in one physical location and required in another location.

- There are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is beneficial for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can provide more reliability than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.

Figure 2.2 shows an example of distributed architecture.

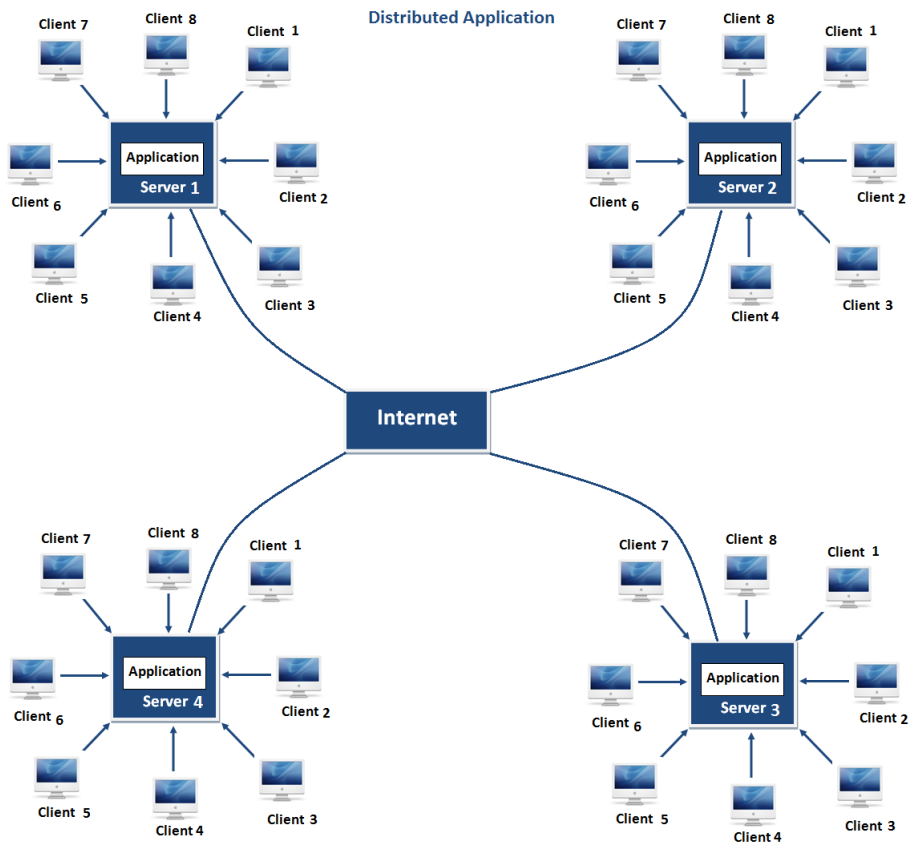

Figure 2.2: *Distributed architecture.*

2.3 Parallel Distributed Programming

We speak of parallel computing whenever a number of compute elements (cores) solve a problem in a cooperative way. All modern supercomputer architectures depend heavily on parallelism, and the number of CPUs in large-scale supercomputers increases steadily.

There are various approaches for parallel programming. The main issue is that the problem of locating data in a parallel context is very hard. A compiler would need to consider the whole code, rather than a subroutine at a time. Even then, results have been disappointing. More productive is the approach where the user writes mostly a sequential program, but gives some indications about what computations can be parallelized, and how data should be distributed. Annotating parallelism of operations explicitly is possible in OpenMP [7]. Such approaches work best with shared memory. By far the hardest way to program in parallel, but with the best results in practice, consists of exposing the parallelism to the programmer and let the programmer manage everything explicitly. This approach is necessary in the case of distributed memory programming.

2.3.1 Message Passing Interface

Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them. With logically distributed memory, the only way one processor can exchange information with another is through passing information explicitly through the network.

The use of explicit message passing (MP), i.e., communication between processes, is the most flexible parallelization method. In a message passing program, messages carry data between processes. Those processes could be running on separate compute nodes, or different cores inside a node, or even on the same processor core, time-sharing its resources. A message can be as simple as a single item or even a complicated structure, perhaps scattered all over the address space. If OpenMP is the most common way to program shared memory, Message Passing Interface (MPI) is the standard solution for programming distributed memory [23].

Today, the MPI standard is supported by several free and commercial implementations and has been extended several times. It contains not only communication routines, but also facilities for efficient parallel I/O. An MPI library is regarded as a necessary ingredient in any HPC system installation, and numerous types of interconnect are supported.

The current MPI standard in version 3.0 defines over 500 functions [31]. An MPI program consists of autonomous processes, executing their own code, in an

MIMD style. The codes executed by each process need not to be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible [15]. The MPI routines can be divided roughly in the following categories:

- **Process management:** This includes querying the parallel environment and constructing subsets of processors. See Figure 2.3.

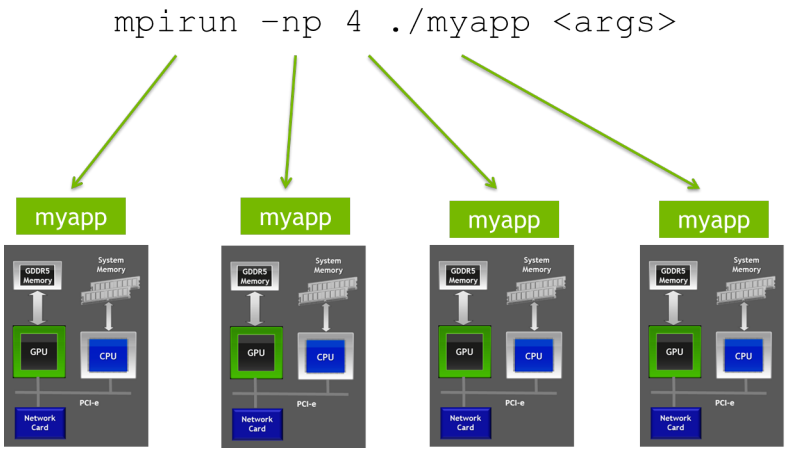

Figure 2.3: *Launch of 4 MPI processes.*

Users can decide how many processors launching and how many resources give to each of them through the execution instruction.

Run the MPI program using the `mpirun` or `mpiexec` command, depending on the implementation used. The command line syntax is as follows:

```
> mpirun [ options ] <program> [ <args> ]
```

A simple call is like that below:

```
> mpirun -np <num of processes> --host <hostlist>
    myprog.out
```

where

−np <number processes> Specify the number of processes to launch;

−−host <host1,host2,...,hostN> List of hosts on which to invoke processes;

myprog.out is the name of an MPI program.

For example:

```
> mpirun -np 3 --host node1,node2,node3 ./myprog.out
```

In this case, the default number of slots on each host is one, unless users explicitly specify otherwise.

`mpiexec` is a replacement program for the script `mpirun`, which is part of the MPICH package. It is used to initialize a parallel job from within a PBS batch or interactive environment. `mpiexec` uses the task manager library of PBS to spawn copies of the executable on the nodes in a Public Broadcasting Service (PBS) allocation [6].

Reasons to use `mpiexec` rather than a script (`mpirun`) or an external daemon (mpd):

- Starting tasks with the TM interface is much faster than invoking a separate rsh or ssh once for each process.
- Resources used by the spawned processes are accounted correctly with `mpiexec`, and reported in the PBS logs, because all the processes of a parallel job remain under the control of PBS, unlike when using startup scripts such as `mpirun`.
- Tasks that exceed their assigned limits of CPU time, wall-clock time, memory usage, or disk space are killed cleanly by PBS. It is quite hard for processes to escape control of the resource manager when using `mpiexec`.
- You can use `mpiexec` to enforce a security policy. If all jobs are required to startup using `mpiexec` and the PBS execution environment, it is not necessary to enable rsh or ssh access to the compute nodes in the cluster.

To initialize the parallel environment in the application, the first statement should be a call to `MPI_Init()`. If thread parallelism of any kind is used together with MPI, the init call must be different: `MPI_Init_thread(<arguments>)` initializes MPI context with a specified level of desired thread support, from a single-threaded application to one where multiple threads may call MPI, with no restrictions.

This allows parallel applications to support hybrid parallelization intra- and inter- nodes, usually using MPI and OpenMP [29] [30].

Upon initialization, MPI sets up the world communicator, which is called `MPI_COMM_WORLD`.

A communicator defines a group of MPI processes that can be referred to by a communicator handle. The `MPI_COMM_WORLD` handler describes all processes that have been started as part of the parallel program. If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`. You can also define multiple communicators for the same set of processes, either duplicating the first one or creating a new one.

Command

```
int MPI\_Comm\_dup(MPI\_Comm comm,
     MPI\_Comm *newcomm)
```

duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that copy callback may take is to delete the attribute from the new communicator. It returns in newcomm a new communicator with the same group, any copied cached information, but a new context.

When there is no need to copy the group information but you may only want to add a new reference the command

```
int MPI\_Comm\_create(MPI\_Comm comm,
     MPI\_Group group, MPI\_Comm *newcomm)
```

can be used. This function creates a new communicator newcomm with communication group defined by group and a new context. No cached information propagates from comm to newcomm. The function returns `MPI_COMM_NULL` to processes that are not in group. The call is erroneous if not all group arguments have the same value, or if group is not a subset of the group associated with comm. Note that the call is to be executed by all processes in comm, even if they do not belong to the new group. This call applies only to intra-communicators.

Example of creating a new communicator:

```
1 //Create new communicator between processes assigned to
2       MPI_COMM_WORLD communicator
3 MPI_Comm new_comm;
4 MPI_Group world_group;
5 MPI_Comm_group( MPI_COMM_WORLD, &world_group );
6 MPI_Comm_create( MPI_COMM_WORLD, world_group, &new_comm );
```

To retrieve environmental setup information about the number of processes in the parallel program users must call `MPI_Comm_size()` and the call `MPI_Comm_rank()` serves to determine the unique identifier (rank) of the calling process. The ranks in a communicator are consecutive, starting from zero. The call to `MPI_Finalize()` ends the parallel program and no MPI process except rank 0 is guaranteed to execute any code beyond it.

Example:

```
1  // Hello.cpp
2  #include <stdio.h>
3  #include <mpi.h>
4
5  int main(int argc, char** argv) {
6    int rank, size;
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &size);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12   printf("Hello World, I am %d of %d\n", rank, size);
13
14   MPI_Finalize();
15   return 0;
16 }
```

- **Point-to-point communication:** This represents a set of calls where two processes interact. These are mostly variants of the send and receive calls. See Figure 2.4.

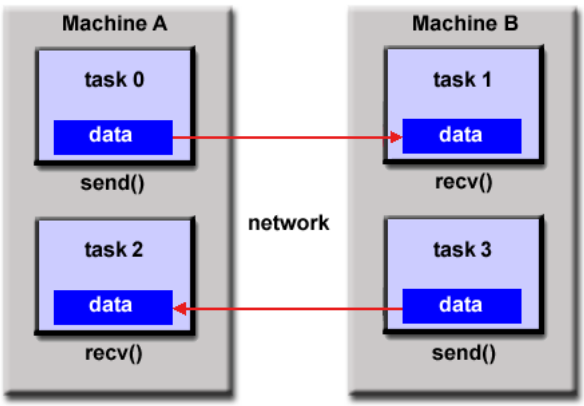

Figure 2.4: *Example of message exchange between two processes.*

An MPI message is defined as an array of elements of a particular MPI data type. Data types can either be basic types or derived types, which must

be defined by appropriate MPI calls. The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions. For any message transfer to proceed, the data types on sender and receiver sides must match.

The basic MPI function to send a message from one process to another is through the `MPI_Send()` call and a message may be received with the `MPI_Recv()`. In each of them, there have been appropriate arguments describing message buffer, number of items to send/receive, MPI data-type of the single item, destination/source rank, message tag and the communicator.

Example of a simple send/receive communication:

```
1  if (myTaskID==0) {
2    MPI_Send(myInfo,1,MPI_INT,/* to: */ 1,/* tag: */ 0,
3          MPI_COMM_WORLD);
4  } else {
5    MPI_Recv(myInfo,1,MPI_INT,/* from: */ 0, /* tag: */ 0,
6          /* status: */ &status,MPI_COMM_WORLD);
7  }
```

MPI has a number of different send modes. These represent different choices of buffering (where the data is kept until it is received) and synchronization. Note that "nonblocking" refers only to whether the data buffer is available for reuse after the call.

- `MPI_Send` will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- `MPI_Bsend` may buffer. It returns immediately and you can use the send buffer.
- `MPI_Ssend` will not return until matching receive posted
- `MPI_Rsend` may be used only if matching receive already posted.
- `MPI_Isend` is a nonblocking send. But not necessarily asynchronous. You can not reuse the send buffer until either a successful, wait/test or you know that the message has been received. Note also that while the `I` refers to immediate, there is no performance requirement on `MPI_Isend`. An immediate send must return to the user without requiring a matching receive at the destination.

- `MPI_Ibsend` is buffered and nonblocking.
- `MPI_Issend` is synchronous and nonblocking.
- `MPI_Irsend` is a `MPI_Rsend` but nonblocking.

- **Collective calls:** In these routines, all processors (or the whole of a specified subset) are involved. See Figure 2.5.

  Examples are the broadcast call, where one processor shares its data with every other processor, or the gather call, where one processor collects data from all participating processors [23].

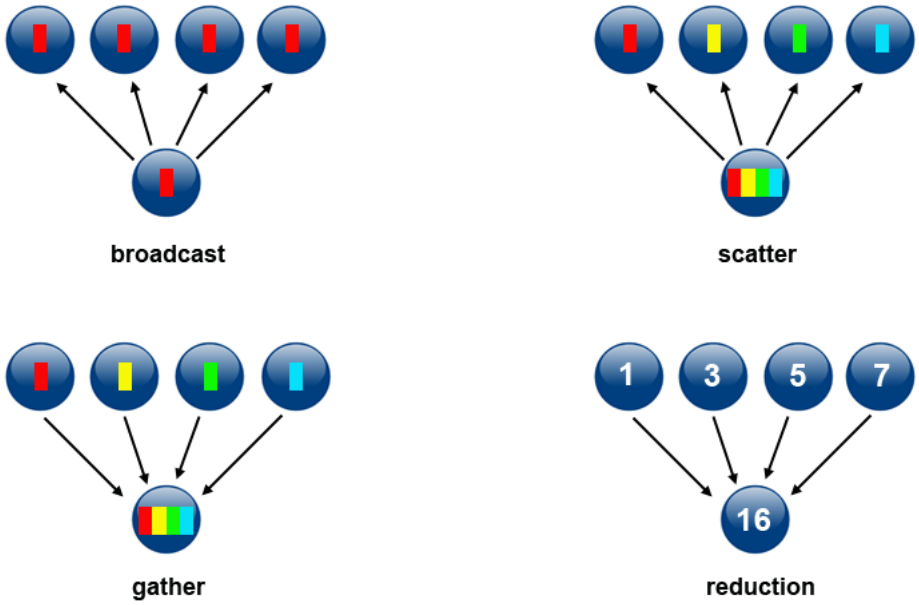

Figure 2.5: *Four types of collective calls.*

2.3.2 Hybrid Parallelization

Modern architectures are often a mix of shared and distributed memory. For instance, a cluster will be distributed on the level of the nodes, but sockets and cores on a node will have shared memory. Intuitively it seems clear that a mix of shared and distributed programming techniques would give code that is optimally matched to the architecture.

A common setup of clusters uses distributed memory nodes, where each node contains several sockets that share memory. This suggests using MPI to communicate between the nodes (inter-node communication) and OpenMP for parallelism on the node (intra-node communication) [29] [30] [35].

In practice this is realized as follows:

- On each node a single MPI process is started (rather than one per socket or core).
- This one MPI process then uses OpenMP (or another threading protocol) to spawn as many threads as the number of independent sockets or cores on the node.
- The OpenMP threads can then easily access the same shared memory [23].

2.4 Model Application Behavior

Performance modeling is a very wide area which researches are focused on. Works in [18] [19] describe automatic mechanisms to generate an empirical model for a given application pattern, used later to predict top 20 I/O parameters that give best performance on HPC systems.

Work [38] instead proposes regression modeling as an efficient way for accurately predicting performance and power for applications executing on any microprocessor configuration in a large micro-architectural design space. It also applies statistical modeling and inference. In HPC such approach can be used, for example, for predicting application performance on a single node, or performance of the part of the application running on a single node.

Other works apply machine learning techniques in order to model error and cost functions, as well described in [50].

In general, all these methods follow more or less the same steps. First, you must collect training data by executing the programs on a set of representative parameters and inputs. Then you have to build the model that best fits those data and validates it. In this work, we choose to use R software to construct our models [13]. R provides a wide variety of statistical techniques, such as linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering and others [51]. One of the most frequent used techniques in statistics is linear regression [43], used to investigate the potential relationship between a variable of interest (the response or output variable) and a set of one of more variables (the independent or input variables). There are flexible facilities in R for fitting a range of linear models from the simple case of a single variable to more complex relationships [11]. If we call Y the response variable and X_1, X_2 ... the input variables we have:

$$Y = f(X_1, X_2, ...X_\mathrm{k}) + \epsilon \tag{2.1}$$

There is a functional link on average between output and input variables, represented by a first component $f(X_1, X_2, ...X_k)$, which is called systematic component. The other component, ϵ, represents the portion of the response variable that cannot be traced back to systematic or easily detectable factors but can due to the case and, more in general, to various causes not taken into account in the regressive model. This functional link can be represented by a linear function and so the model associated is called multiple linear regression. It has the following formulation:

$$Y = \beta_0 + \beta_1 X_1 + ... + \beta_k X_k + \epsilon \tag{2.2}$$

where β_0 is the known term, while $\beta_1, ..., \beta_k$ are called regression coefficients and together with the error variance, are the model parameters to be estimated on the basis of sample observations.

Several models, nonlinear in appearance, can be linearized by appropriate transformations of variables. Transformations commonly used applies logarithm or other similar tricks. When in the model there are input parameters with a grade more than one, it is called polynomial regression, but the model continues to be linear in the parameters. The following example shows a parabolic regression model with only two input variables:

$$Y = \beta_0 + \beta_{12} {X_1}^2 + \beta_{13} X_1 X_2 \tag{2.3}$$

Notice that it is taken into account the interaction factor between the input variables ($X_1 X_2$).

When the parameters are displayed in a different form, we have nonlinear regression.

In R software, estimation of model parameters can be done by the command `lm()`. It has as arguments the symbolic description of the model to be estimate and the dataframe where there are the variables (values) of the model.

For example in a dataframe with these variables

```
Fertility Agriculture Examination Education Catholic
Infant.Mortality
```

we can generate the simplest linear regression in this way:

```
> mod<-lm(Fertility ~ Agriculture + Examination +
Education + Catholic + Infant.Mortality, data=swiss)
```

If data are in a csv file we can load it in a dataframe with the commands

```
d = read.csv("datamodel.csv")

// create dataframe
df = data.frame(d)
```

The command `summary(mod)` shows the result:

```
> summary(mod)

Call:
lm(formula = Fertility ~ Agriculture + Examination +
    Education + Catholic + Infant.Mortality,
    data = swiss)

Residuals:
 Min       1Q   Median       3Q      Max
-15.2743  -5.2617   0.5032   4.1198  15.3213

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)      66.91518   10.70604   6.250 1.91e-07 ***
Agriculture      -0.17211    0.07030  -2.448  0.01873 *
Examination      -0.25801    0.25388  -1.016  0.31546
Education        -0.87094    0.18303  -4.758 2.43e-05 ***
Catholic          0.10412    0.03526   2.953  0.00519 **
Infant.Mortality  1.07705    0.38172   2.822  0.00734 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
    0.1 ' ' 1

Residual standard error: 7.165 on 41 degrees of freedom
Multiple R-squared:  0.7067,Adjusted R-squared:  0.671
F-statistic: 19.76 on 5 and 41 DF,  p-value: 5.594e-10
```

The simplest way to see if it is a good model is to observe the Adjusted R-squared. The greater it is, the better is the model. In general a regression model with an adjusted R-squared above 0.60 is considered a good one.

There are other formulations to try, in order to increase the value of the adjusted R-squared, for example the "complete" linear regression model $y\ x_1 * x_2 * \ldots * x_{\mathrm{n}}$ takes into account all the possible interactions between input variables.

Once having constructed a model for the data, we can use it to predict the response variable. In the command, we have simply specified the model to be used, the dataframe and the type of interval to be used.

Example:

```
> predict(mod, swiss, interval="confidence")
              fit      lwr      upr
Courtelary   74.61530 68.88485 80.34574
Delemont     82.50994 77.44342 87.57647
Franches-Mnt 85.91826 79.88753 91.94899
...
```

A prediction interval is an interval associated with a random variable yet to be observed, with a specified probability of the random variable lying within the interval. Prediction intervals can arise in Bayesian or frequentist statistics. A confidence interval is an interval associated with a parameter and is a frequentist concept. The parameter is assumed to be non-random but unknown, and the confidence interval is computed from data. Because the data are random, the interval is random. A 95% confidence interval will contain the true parameter with probability 0.95.

In order to have a good regression model, there are some tricks that can be used: for example, you can construct the model of the log of the response variable if the relationship is not so linear. Otherwise R provides many other functions aimed to construct more precise but complicated models, for example you can use the `glm` command for modeling data whose response variable is not continuous, or with the `nls` command you can specify a non linear relationship between variables.

CHAPTER 3

State-of-the-art

3.1 Introduction

This chapter gives an overview on the state-of-the-art regarding methodologies and techniques developed in this thesis, in particular we describe autotuning techniques used to satisfy generic application requirements and workloads about the problem for approximate programs. We present the different autotuning strategies existing in literature and we focus on the description of the mARGOt autotuner.

3.2 Tunable Approximate Programs

In this work, we will focus on a particular but increasingly class of programs that are tunable and use approximate computing techniques in order to trade off accuracy of results and some performance metrics such as latency or energy consumption [39].

A tunable application is a program whose behavior in terms of performance can be influenced by manipulating a set of application parameters, called knobs. These knobs are domain-specific and can be defined at design time and exposed to the application, or even extracted at run-time [33].

A set of knob values with which the program can be executed is called *Con-*

figuration, because it can be used to address a specific point in a multi-objective space of application optimization function, in order to configure approximation and performance aspects of the computation of that application (accuracy, time, energy...). In general a configuration is chosen as it is the solution of a particular mono or multi-objective optimization function.

In both embedded and HPC fields, a common type of optimization function tries to maximize or minimize a particular metric in contrast with another one with which there is a trade-off, such as maximize accuracy with respect to computational time or energy, or vice versa, minimize time or energy with respect to general quality of service needed. From now on, we will refer to a general performance parameter, such as time, energy, as a *Metric*. Indeed the optimal configuration could not remain the same during the execution, since there could be changes in many aspects that will influence the execution at run time, such as changes in workloads, resources or hardware. An *Operating Point* is a tuple of a configuration plus metrics at which the application operates in a particular time window execution.

In HPC field, it is very important to spend efforts in minimizing time to solution and energy to solution [24] [44] [48]. This is a concept that must apply to all phases of the applications, from the start to the end. Generalizing we can refer to a general metric to solution to minimize(maximize). A metric to solution is a quantity of that metric referring to the whole computation of the application, not a single phase. In this work, we try to find a way for maintaining execution of a general application in a given metric-to-solution constraint and we focus mainly on knobs used for tuning the level of accuracy of an application that uses approximate computing techniques.

Approximate computing is a way of reducing energy and time required to execute applications at the cost of reducing precision [50] [16] [32]. It is more and more popular since the domains of modern applications are evolving and they are no longer relegated only to implementations of mathematical functions. So in many new problem domains, for example in the computer graphics or in the data mining fields, there is no need to compute a precise output for a given input but it is sufficient producing an approximated result [21].

In approximate programs, application knobs are mostly used for controlling and tuning the level of approximation of different components. There exists a lot of techniques that cover different layers of computer science, ranging from purely hardware to purely application level [40].

For a matter of portability, the techniques at application level are the most common. Examples are:

- **Loop Perforation** to transform loops to execute a subset of their iterations, thus having a gain in performance metrics [52].

  This technique consists of skipping some iterations based on different strategies regarding the determination of the perforation rate. With Modulo strategy, for example, perforation rate is determined executing or skipping every n-th iteration the included instruction. This approach is particularly suitable when there is a uniform distribution of the execution of load and data among all iterations.

- **Data-type precision** Another example consists simply of varying the memory width where a variable is stored, between low and max data-type precision. For example, in a C++ environment, programmers can switch between `int` and `float`, or between `float` and `double` data-types and study the loss in accuracy and gain in performance.

Another feature of applications we will cover in this work is the unknown workload in input. With the word "unknown" we are referring to applications having to process an amount of data with different features per input. This amount is so huge that applications cannot afford to classify all the features reading first the whole data input, since this wastes too much time.

Data input features are particular features that influence performance metrics. For example a common data input feature is the length of the data input. If the application has to compute a vector of double, the time necessary to the application grows with the length of the vector in input. In a graph problem to find the shortest path between two graph nodes, data features can be the number of nodes and the number of arches of the graph in input. The bigger are these features, the longer time the application needs to find a solution.

Understanding the way data input features influence those metrics is very important in order to better understand how the application will behave with data input with different characteristics. This will be better explained in the modeling construction Section 4.3.

3.3 Autotuning

Autotuning is defined as the task to automatically find the best configuration satisfying the application requirements. In HPC field, autotuning is a technique well used in order to satisfy quality of service level reducing computational costs in terms of latency and power consumption.

In the past, there have been approaches that have assessed application autotuning as one of the possible instruments for managing in an efficient way

computing resources in homogeneous [27] and heterogeneous [41] [20] [36] architectures, even when these applications can change as a result of a porting to another platform. Despite that, the problem is still far from being solved.

Autotuning techniques have been investigated at the System Architecture group of Politecnico di Milano: there is an autotuning of applications for homogeneous multi-core architectures [25] which is mostly related to the runtime management phase; an autotuning for heterogeneous architectures has recently been proposed in [42]. There are also other studies still developed at Politecnico more focused on the off-line phase which performs the collection of data of the application in order to find better configurations and change between them with software knobs [47].

There are 3 main categories of autotuning techniques:

1. **Design-time Autotuning** This type of autotuning consists of performing a design space exploration of performance values of applications at design-time and finding optimal configurations which the application can be executed with and which probably impose a certain trade-off value. But the configuration chosen off-line could not be longer valid in applications that might change their requirements at runtime depending on external conditions. Thus a more reactive and self-aware way to compute the optimal configuration is usually needed.

2. **Dynamic Autotuning** This category is the most widespread and self-aware autotuning technique.

   Definition 3.3.1. ***Dynamic autotuning***

   A technique through which applications can be dynamically adapted to changing conditions at runtime.

   An autotuner leverages application behavior by changing the values of dynamic knobs according to application requirements. It also can let the application to self-adapt based on a monitoring infrastructure that senses the execution context performing like an observer that keeps track of performance metrics of applications thanks to which autotuner can make a decision on the next part of computation.

   In order to use a dynamic autotuner an application must expose dynamic knobs which can be tuned by the autotuner. The work in [33] describes a framework to extract the dynamic knobs automatically, transforming static configuration parameters into dynamic ones, calibrate and tune them. In particular PowerDial enables applications to execute responsively in the

face of power caps and it can significantly reduce the number of machines required to service intermittent load spikes, enabling reductions in power and capital costs. It uses the calibrated dynamic knobs to move the application to a Pareto-optimal point. PowerDial has a feedback mechanism that allows the system to monitor application performance, a control component which converts the feedback into a desired speedup and an actuator which converts the desired speedup into settings for one or more dynamic knobs.

In general autotuning consists of a first Design Space Exploration part in which the autotuner must understand application features and then uses it at run-time for computing the optimal operating point every time there are changes in conditions, sensed through the monitoring infrastructure.

Researches at Politecnico di Milano are very active in the dynamic autotuning field: hereafter there is a brief description of the 2PARMA project and the current ongoing ANTAREX project, led by Politecnico di Milano, then we present the mARGOt dynamic autotuning framework, which is the one we choose to use in our framework.

2PARMA [45] This project focuses on the development of parallel programming models and run-time resource management techniques to exploit the features of many-core processor architectures. In particular one of the objectives of this project meets the challenge we are interested in: to explore power/performance trade-offs and to provide runtime resource management and optimization in a parallel environment. For the optimization part, this project provides a runtime manager with metadata information covering both design time and run time knowledge of both hardware and software, combining a DSE tool used at design-time to find the set of operating points plus the info collected at run-time on the system workload and resource utilization. The run time manager is also used to handle the dynamism in the control flow and data usage by determining suitable allocation strategies that meet the application needs. Finally the runtime manager is responsible for the adaptive power management of the many-core architecture. The DSE phase is made by extending the MULTICUBE Explorer framework to support runtime dse.

ANTAREX [46] The ANTAREX ongoing project explores autotuning techniques in HPC systems. Its main goal is to express by a Domain Specific Language the application self-adaptivity and to runtime manage and autotune applications for green heterogeneous HPC systems up to Exascale. The related tool flow [Figure 3.1] operates both at design-time and runtime.

The specification of runtime adaptability strategies will rely on the DSL im-

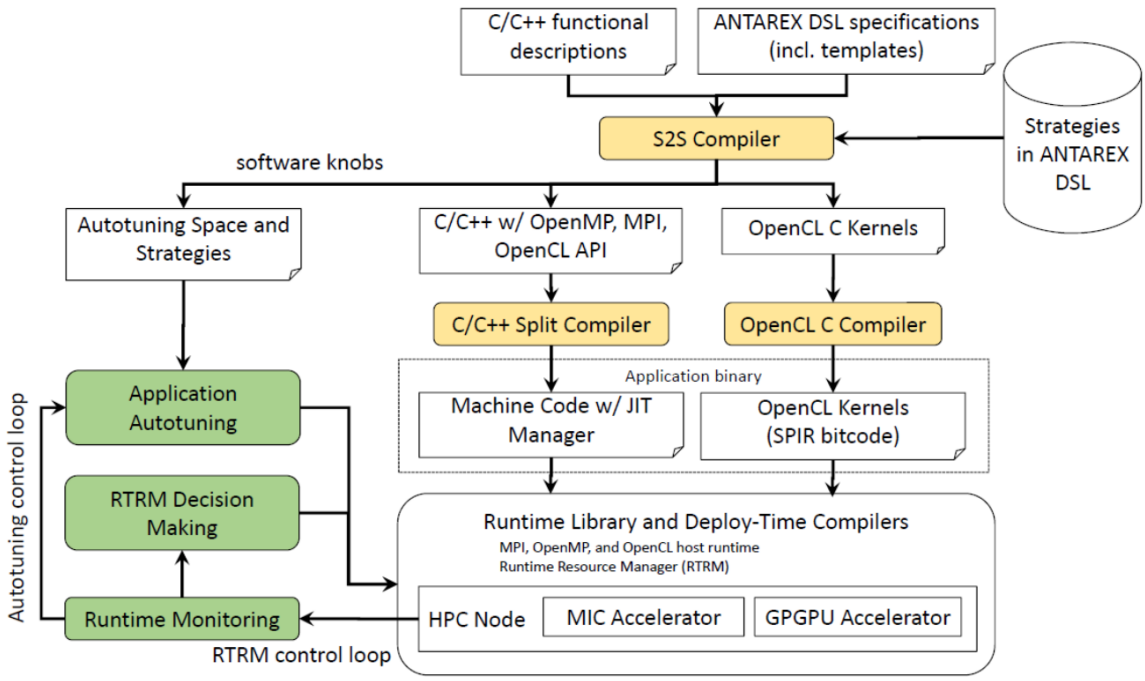

Figure 3.1: *The ANTAREX tool flow [46].*

plementing key concepts from Aspect-Oriented Programming (AOP) [37]. It also offers runtime resource and power management combining application progress and dynamic requirements; autotuning capabilities enabled by the DSL in the applications; information coming from the processing elements of the IT infrastructure and its performance knobs.

3. **Proactive Autotuning** Autotuning can also be done at runtime in a proactive way, for selecting the best configuration for a non-streaming application before it is executed. So, differently from the dynamic autotuning, this one is not used in response of something, but to choose how the application must be executed based essentially only on particular data input features to compute. There is very little amount of work on exploiting autotuning functions in a proactive way. Our work goes in the direction of investigating this innovative way of using autotuning frameworks. The main work handling this problem is [50], described in Section 3.4.

3.3.1 mARGOt Framework

mARGOt framework is a dynamic autotuner and it is the autotuner we use in this work. It is being developed at the System Architecture group of Politecnico di Milano [25]. At [4] you can find the gitlab repository of the mARGOt autotuner framework. It is also one of the tools used in the ANTAREX project [1]. mAR-

GOt is a runtime framework used to dynamically adapt applications in multicore architectures. It was designed for the tunable applications class that exposes dynamic knobs. This framework requires to have some knowledge about the application and this knowledge is represented as a list of Operating Points, explained previously. These data are collected during the performance profiling phase done at design time and they are all Pareto-optimal with respect to the target metrics.

This framework is composed of two big infrastructures: the Monitoring infrastructure and the Application-Specific Run-Time Manager [Figure 3.2].

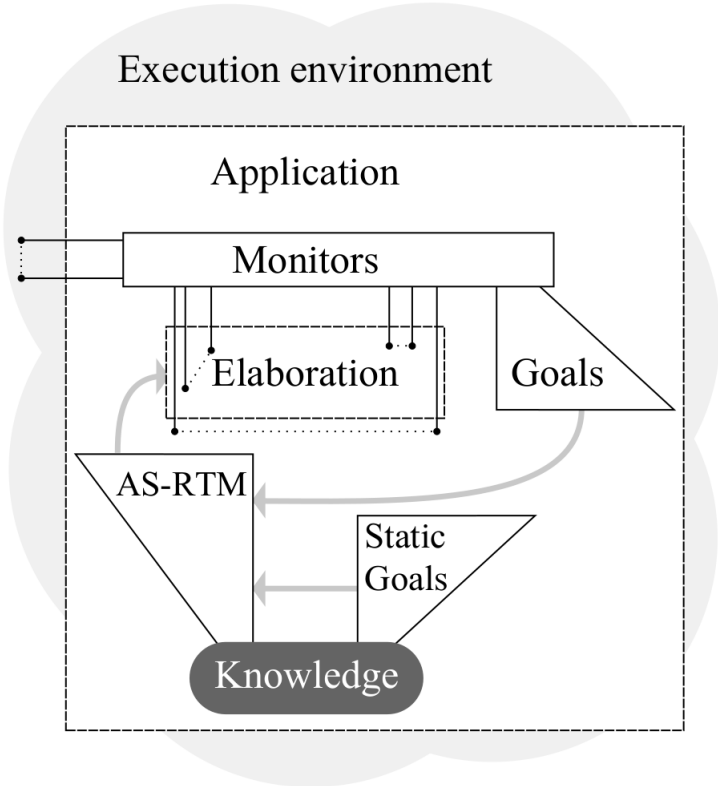

Figure 3.2: *mARGOt structure [25].*

The first one is used in order to keep updated the knowledge of the framework at runtime, regarding the application running it. Different types of monitors can be used to observe both high-level and low-level metrics. A requirement of the application on a monitored value is expressed by a Goal object. It requires to define a monitor to be used to keep track of the metric, the value of the goal, the statistical properties of the monitored metric and the comparison function. If a Goal is related to a metric not monitored, it is called Static Goal.

The Application-Specific Run-Time Manager instead exploits design-time knowledge to select the best configuration for an application, taking into account information coming from the monitors, the goals defined in the application and

the type of optimization function (rank value). If there is a "normal" Goal, this runtime manager compares the goal value against the observed one, while if there is a Static Goal, this manager compares the goal value only against the one in the current Operating Point. An optimization function is a combination of objective metrics to be maximize or minimize. An application can have more that one optimization function, with different constraints. To do so, mARGOt provides the State concept, which is the object wrapping a particular state of the application characterized by a particular set of goals, monitors and "rank value". This framework also realizes the separation of concepts between functional and extra functional properties, facilitating the integration in the application providing a tool that automatically generates the required code from XML configuration files.

3.4 Control Problem of Tunable Programs

This problem affects in particular the controlling of non-streaming tunable programs which cannot be observed in order to adjust the computation. So far, indeed, autotuning techniques were mainly used for observing and adapting streaming programs since every part of the execution is strictly linked to the previous one. The proactive control technique is one of the solution found for solving this problem.

The work in [50] formulates the control problem as a constrained optimization problem, focusing on application whose knobs can control the level of approximation of different components and dealing explicitly with the problem of input variability. It also addresses the inverse problem with respect to dynamic autotuning. Starting from the Definition 3.3.1, the inverse problem tries to find the best configuration for an application execution, given a set of application knobs and the data features of the input. The work [50] also provides the description of the Capri system that uses machine learning techniques to learn cost and error models for the program and uses these models to determine the configuration that optimize performance metrics, such as running time and energy spent, while meeting a desired level of approximation.

In order to solve the control problem for a given tunable program, these information are needed:

1. *Output function:* as for classical tunable programs, the output value can be a function of the data features of the input and the knob settings.

2. *Quality degradation:* also this function can depend only on data input features and knobs.

3. *Cost:* this function expresses the cost of computing the output for an input with a configuration. This cost can be whatever performance metrics to be optimize.

As we have seen, for building cost and error models they use machine learning on the data obtained by executing the programs on a set of representative inputs. They profiled the program exhaustively over the knob space.

Two search algorithms are evaluated for the design space exploration of knob settings:

- **Exhaustive search**

  It performs a full search over the entire space of configurations and for each of them uses the error model to determine if that configuration lays in the feasible region. The cost model is then used to find a minimal cost point in that region.

- **Precimonious search [28]**

  It is a heuristic-based search strategy which starts from a *change set*, made up of all configurations set to the highest values of performance they realize. At each iteration, this algorithm lowers all configurations in that set by one level, and if this new setting satisfies the accuracy constraint the algorithm goes on to the next iteration. Otherwise it tries to find subsets of knobs whose settings can be lowered while satisfying the accuracy constraint and chooses the lowest cost subset from these, which becomes the new change set. This strategy takes less time to be executed, but the solution it finds can be a local minimum.

In this work, we also evaluate two search strategies very similar to the ones described above, but with a few differences, especially regarding the second strategy. They are later described in Subsection 5.3.4.

3.4.1 Capri System

Capri is the system implemented to evaluate the work [50]. Figure 3.3 shows an overview of this system.

It must be provided, for a given program, with a set of training inputs and metrics for quality and cost of the output. As we can noticed, it is mainly divided in an offline and online part. The offline part deals with the learning phase and builds up the models required. The online portion, instead, is composed of a controller which, having these models and an input, solves the control problem to estimate optimal knob settings. This implementation in particular uses Bayesian

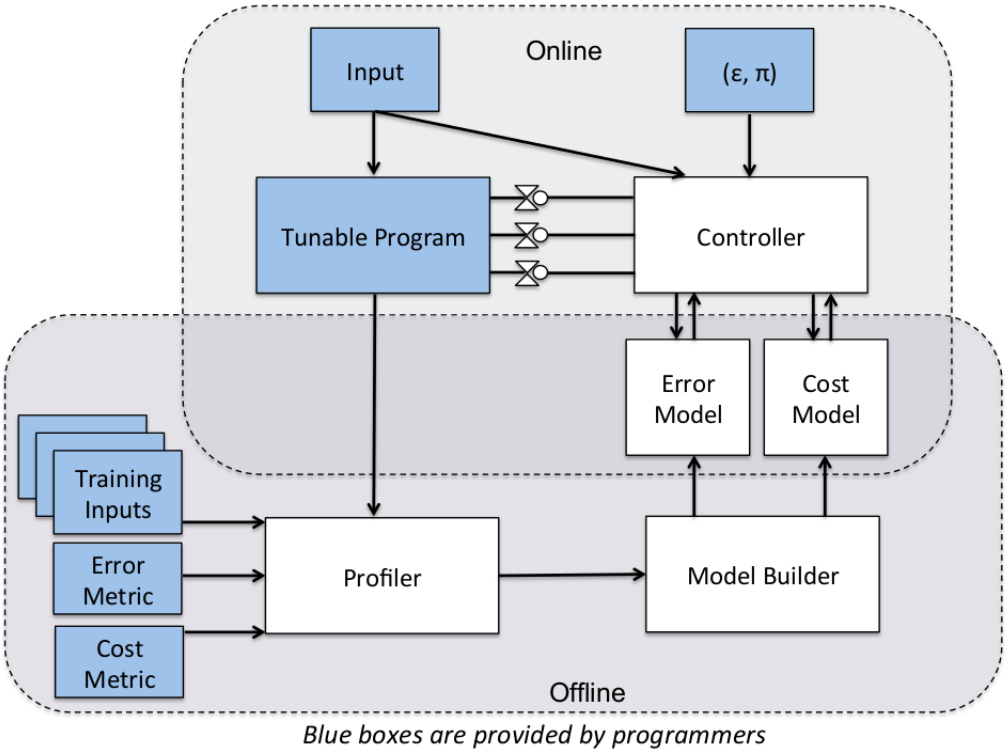

Figure 3.3: *Overview of Capri system [50].*

network for learning the error model, and the tree-based model in the M5 system for the cost model. It uses the open-loop control to control approximation. Thus there is no way to correct for model inaccuracy, but stability is not an issue since there is no feedback.

Results obtained evaluating this approach on different programs show that obviously training time increases with the number of training inputs, but since it is done offline, it is not so important. The accuracy of cost and error models is evaluated by comparing predictions against empirical measurements. It shows that for some programs the accuracy is pretty good, but for others it does not reach an acceptable level. However, they state that more accurate models do not necessary enable Capri system to produce better solutions, since some models are used only for doing things with a low impact on the overall solution. Then, they evaluate the speedup of the controller and the results show that speedup depends on the application and its constraints but overall this system can yield significant speedups in running time. For the effectiveness in finding optimal knob settings, they compare models built using linear regression against using the learning techniques described above and the results show that using the second strategy is fairly successful across the constraint space. In the end, they evaluate the use of exhaustive and Precimonius search algorithm and they

find that Precimonious is significantly faster that the other, but the solution found maybe be worse than the one using exhaustive search.

3.5 Contribution of this Work

The last approach described [50] is very similar to the one adopted in this thesis, but our work introduces many innovative elements: first, we work in a parallel and distributed environment, where memory is not shared, then we integrate the mARGOt autotuner and evaluate its behavior in distributed environments in a proactive way. Then, we try to satisfy a given metric-to-solution global constraint acting on knobs in every application process, still dealing explicitly with the problem of input variability. Another innovation point is that our framework can automatically generate at runtime simple predictive models according to the application using it. All these new elements are well described in Chapter 4 and Chapter 5.

CHAPTER 4

Proposed Methodology

4.1 Introduction

This chapter explains the methodology proposed in this thesis and the logic structure developed for implementing it in the Ipazia framework described in Chapter 5. First, there is a more detailed description of the addressed problem and its context, expanding the concepts already introduced in Chapter 3. Then, we go in depth in the design flow, giving a description of the methods and concepts used and developed in the proposed methodology.

4.2 Problem Description

Optimization performance problem affects many computer science and industrial fields. Since HPC systems are very powerful structures, they are also very consuming and many current researches are investigating this problem, proposing different types of solutions that we have analyzed in Chapter 3. Nevertheless, this problem is still far from being solved.

In this work, we also give our contribution in this direction. In particular, we focus on the HPC context where environments are parallel and distributed, with no shared memory. The specific problem we address is the satisfaction of a global constraint (mostly regarding performance) in HPC applications with

input variability which, as we have seen, has not been treated in the current state-of-the-art described previously. This global constraint is given in terms of a particular metric-to-solution that the application must meet. In HPC, popular metric-to-solutions are the Time-to-Solution or Energy-to-Solution, since these are the main optimization areas.

The difficulty in solving this problem arises from the fact that many HPC applications have unpredictable behavior as it strongly depends not only on the load of the input data but also on the characteristics of the data input themselves. Therefore, knowing the load of incoming data is not enough in order to understand how the application will behave in terms of time or energy performance. In an HPC application that has to process a very huge amount of data, it is unfeasible to characterize it by reading all the incoming data before computation has started. Just thinking about the capacity of HPC to handle and analyze massive amounts of data at high speed, that can take months using normal computers, in days or even minutes, and also thinking about the ability of HPC systems to run data-intensive modeling and simulation problems at large scale, at higher resolution and with more elements. As seen in Chapter 2, HPC meets data-intensive challenges which are already enlarging HPC's contributions to science, commerce and society. So we have to imagine another way to characterize these applications spending little time and maybe analyzing only a part of the data in input. The detailed description of the characterization method proposed is given in Section 4.3.

Furthermore, this work addresses this type of problem for parallel applications with more or less the following structure: they process the data in input using multiple instances of themselves, each one executing in a separate process with its own memory space and computing only its own part of the whole workload. There is a master process which acts as a dispatcher of inputs to the other worker processes that actually perform computation on the data. Results are then collected by the master. There are in literature many works on performance sustainability for parallel applications, but none of them gives an answer regarding such circumstances. We developed a way through which the global constraint is somehow spread on all worker processes based on their own workloads, thus allowing them to perform local optimization while meeting only its own part of the global constraint and remaining totally unaware of global dynamics. More precisely, we found that spreading the global constraint on a single data input is an effective way for doing this.

Now the problem is how to extract the quantity of the global constraint for a precise input. Obviously, it depends on how many data there are as inputs and what are the characteristics of the data input themselves. So, first of all, we

should get an idea of how the application will behave on average, still in terms of how much metric-to-solution given it would consume. As we have said, applications considered are tunable, so another degree of freedom in the application behavior regards the configuration (knobs setting) under which it is executed. In particular, a configuration sets a precise level of accuracy in the application. Essentially, we need some application models, function of both knobs setting and data input features, which must be able to give us predictions relating to the part of the metric-to-solution that the computation of a single input would require. As mentioned in Chapter 3, tuning the level of accuracy means changing the performance of the application, since in HPC applications there is always a trade-off between accuracy and performance metrics such as execution time and energy used. We will consider only time metric for the experimental results in this work, but the whole methodology is suitable for every type of metric used.

Another problem regards the use of autotuning techniques in worker processes. We indicate as worker process a process in the MPI environment which is not the master one. In the applications considered, worker processes are the ones that actually perform computation on the inputs. We have said that integrating an autotuner in an application would lead to a saving in time or energy used and so we would use a dynamic one in order to allow a local optimization. Issues in this phase concern more or less on how a worker process can obtain its own autotuning knowledge, and all the options treated in this methodology are well described in Subsection 4.3.4. We will use the autotuner in a proactive way, thus investigating furthermore this new autotuning capability and its effectiveness.

In the end, we also handle the problem of dynamically recomputing the part of the metric-to-solution to give to each input on average. This is important because the first time it was computed maybe would not been so precise, since we have not read all data in input. Therefore, as the workers completed computation on an input, the part of the metric-to-solution estimated for that input could have been too much or too low, and so we propose a method to adjust this quantity of metric to spread for the next inputs that have not been already sent. This is done in order to still satisfying the global constraint but allowing a more accurate computation if there is much more metric remaining or forcing to lower the accuracy level in case the new quantity of metric estimated is less. The computation of the new quantity of metric will be calculated on the remaining inputs to be sent.

4.3 Methodology

In this section we approach every topic mentioned in Section 4.2 in detail, describing step by step the mathematical constructs, algorithms developed and de-

cisions taken representing a possible solution to the problem addressed in the introduction.

In Figure 4.1 there is a global schematic design flow of the methodology proposed in this thesis. Each block is explained in the next subsections.

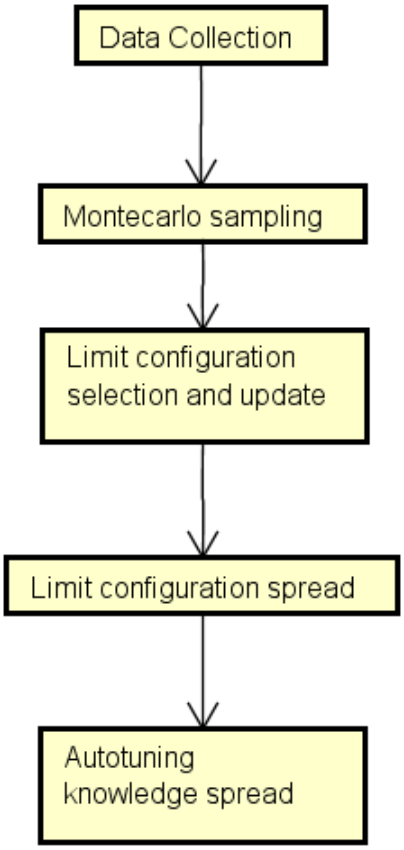

Figure 4.1: *Proposed methodology phases.*

4.3.1 Data Collection

The first step in our methodology consists of a *profiling phase* of the application. In this phase information regarding metrics and knobs are collected. With the word information we indicate the collection of metric values about executing the application under different input samples and different knob configurations. The metric values to be collected are those that the application requires both in the metric-to-solution global constraints and in local constraints. Local constraints are then managed in the mARGOt framework, while information about the global metric are necessary for the computation phase of metric-to-solution to be allocated for each input.

After collecting these data, a relationship between metrics and both knobs and

input features must be found. In this work, we decide to use the R software [13] for doing this. We started from a simple linear regression in order to construct these models: for each metric, for example, one model to predict the time spent on execution of a single input and one to predict the level of accuracy of a single input.

Then we used these models in order to make predictions on the metric values. A model must predict a precise metric as a function of the accuracy level (set by a particular configuration) and the input features of the single data considered. This process is valid also in a distributed environment where machines have different architectures and so they can show different performance for the same input and configuration, since in the input features can be encase all the parameters that our framework and the mARGOt framework must not tuned.

These prediction models are then exploited in many phases of the methodology, both from the master and workers processes, if they have access to this knowledge.

In this phase we also collect information about the trend of the metric-to-solution given as global constraint.

4.3.2 Montecarlo Sampling

Since the target application has a huge data input, we are not able to read the whole workload in order to understand how the application will behave, without spending too much time on this phase. Therefore, we choose to adopt the Montecarlo sampling approach. This means that we pick up a quite-random number of samplings from the whole data in input and we collect the features of these samplings in order to construct a Probability Density Function (PDF) regarding each type of input features the input must have. All the PDFs built give us an overview of how can be the data distribution in input.

For example, if data in inputs are vectors, one possible input feature can be the length of them. Therefore, the PDF of the length of the vectors gives us information about how many vectors have a length over 200, how many of them have a length less than 10, and so on. In this way we can get an idea of how much time the application will execute the whole data input, how much power it needs, and so on.

Starting from this distribution then we calculate the quantity of metric-to-solution to allocate for a given input. Obviously, the data distribution found in this way is only an estimation of the real data distribution in input and so, exactly for this reason, we need to update this distribution as inputs are executed.

Figure 4.2 and Figure 4.3 show examples of PDFs. Each bean shows the density of a molecule/ligand in input with a number of atoms between the values

written on the x axis.

For doing this we face the problem of how many samplings we need for computing the PDFs. First of all, as we have already said, we cannot examine the whole data in input, but just a part of it. We adopt the Cartesian product approach for building a unique PDF enclosing all input features information. For every bean of the PDF of an input features, we construct as many beans as the number of others PDF input features multiply each others. So, the number of samplings must be less than the number of total inputs, but also it has to satisfy a precise inequality formula, otherwise we have no income in time to do the sampling instead of reading all inputs.

If we call x the parameter we have to set in order to create a number of samplings to use, below there is the mathematical formulation to derive the range of values x must take, starting from the number of data in inputs:

$$numsamplings < \frac{|data|}{x} \tag{4.1}$$

Then:

$$x > \sqrt[n]{|data|} \tag{4.2}$$

Where:

n: Number of data features of each input.
$numsamplings$: Number samplings to collect.
$|data|$: Cardinality of data in input.

4.3.3 Limit Configuration Selection

In this phase we develop a mathematical method for extracting the configuration that on average would meet the global constraint(i.e. the metric-to-solution given). From then, we address this configuration as the *global limit configuration*. Indeed, we must point out that the sentence *configuration that on average would meet the global constraint* has a particular meaning for us. In this phase, we do not really want to obtain the configuration that meets the global constraint optimizing it in an extreme way, but we need that configuration which acts as a watershed between configurations that are good (satisfy the constraint) and the ones that are not good (do not satisfy it). Since we have used a local autotuner in order to improve performance and have gain for example in time, energy, now we do not perform a search of the best configuration absolute, but of a limit one. This leads to a more freedom in the autotuning phase.

The formulation is expressed as:

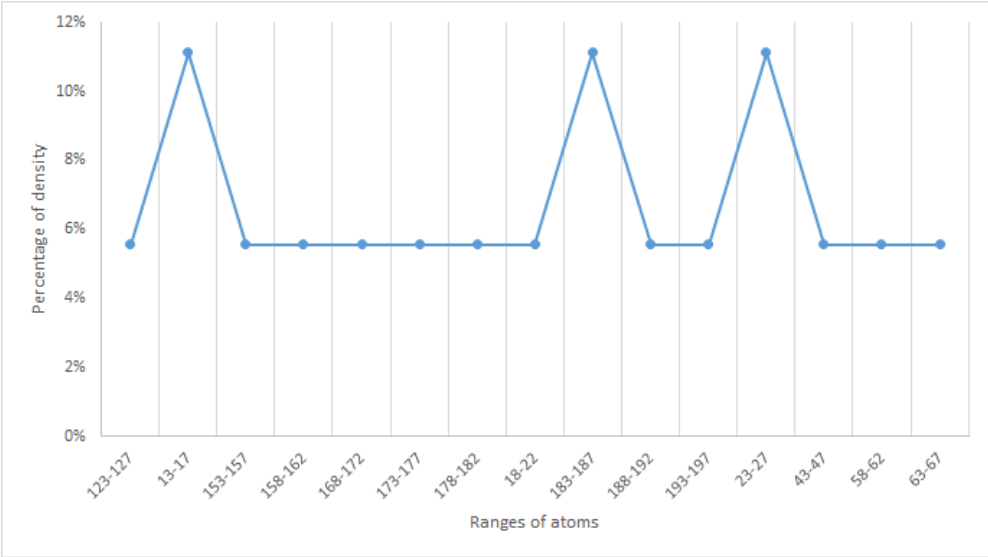

Figure 4.2: *Probability Density Function of molecules on their number of atoms.*

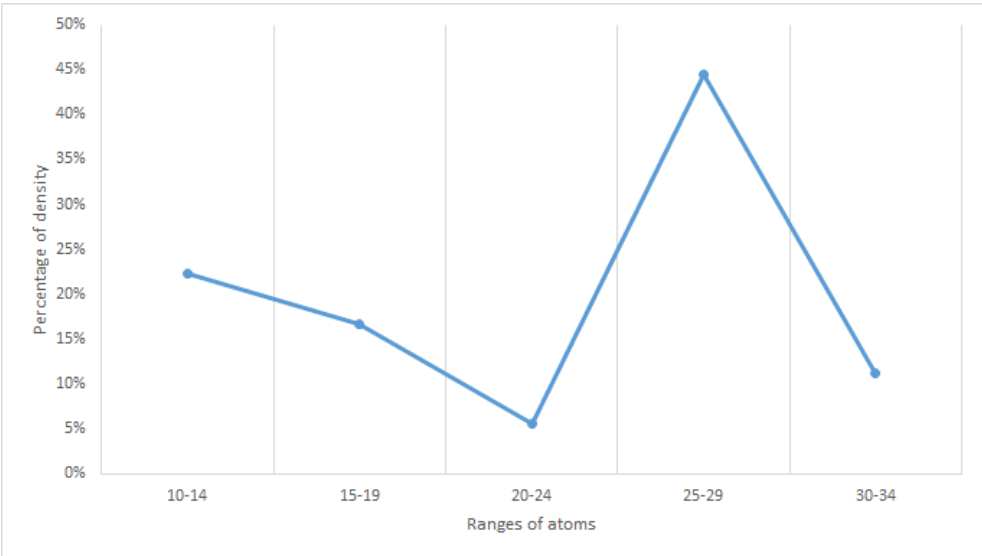

Figure 4.3: *Probability Density Function of ligands on their number of atoms.*

$$\sum_{i}^{k} p(d_{\mathrm{i}}) \cdot f(d_{\mathrm{i}}, OP) \leq \frac{MtoS(\cdot numresources)}{|data|} \tag{4.3}$$

Where:

k: Number of different beans constructed with different input features.

d_{i}: Input features vector of i-th bean.

$p(d_{\mathrm{i}})$: Probability density function of input features d_{i}.

$f(d_{\mathrm{i}}, OP)$: Predictive function of the MtoS metric.

OP: Operating Point on average that must be found.

$MtoS$: Metric to solution global value given.

$numresources$: Optional parameter in case the MtoS value must be spread on the number of resources available in a parallel environment (ex. in case the metric considered is the total time).

$\leq$: To be set according to the one given in the global constraint problem.

$|data|$: Cardinality of data in input.

4.3.4 Constraint Spread

Once the limit configuration has been selected, it must be sent to every worker. In our methodology, we decide to spread this configuration over a single input. Our solution operates in this way: for every input, in the input submission phase, it is estimated the part of metric-to-solution that a worker would spend in order to compute this input, according to input features and the global limit configuration selected before. Also in this phase the models previously built are used. This means that every worker has to receive a precise part of the global metric regarding each of the inputs it has to compute. We call this part of global metric a *soluion* for a precise input.

Sending this value is important since it has to be used in the local autotuning phase for making the worker satisfy the global constraint it cannot see.

4.3.5 Configuration Update

If our models are not so good, predictions can be wrong and our application may not actually satisfy the global constraint. To put a remedy to this, we have considered to introduce a periodic recomputation of the global limit configuration. It can be done every number of results returned in master or something else. In this way we adjust the previous estimation of the input features distribution and the

new configuration to select now is related to to the new metric-to-solution value. From now until the next configuration update, the inputs that have yet to be sent will gain a constraint value associated updated. For example, after 100 results received there is the recomputation of the configuration. If the metric of interest is the execution time, its new value to be considered is the old value minus the time spent until now. The new configuration selection will take into account the advance or the delay in the metric consumption and recalibrate its usage.

We decide to perform this adjustment only during the dispatching phase, so, once all the data in input have been assigned together with their regarding metric-to-solution information, no configuration update will happen.

4.3.6 Autotuning Knowledge Spread

Each worker will perform a local autotuning phase. Therefore, we have to think also about how each worker can gain all the knowledge related to the input to execute. As well described in Section 3.3, an autotuner needs a knowledge on which to perform optimization. In particular we must give it the Operating Point list related to each input. An operating point in our case will contain a configuration and the values of the metrics on which autotuning is performed. As we have seen, the configuration is a set of knob settings, while the metrics of interest are values that the application probably would use during the execution of a precise input. The metric values to put in the operating point are computed starting from the models developed previously. So, each predicted metric value owns inherently knowledge about input features.

We think of four possible spreading knowledge strategies for providing the knowledge to the autotuner. They go from the fully distributed version to the fully centralized one. Hereafter the detailed description of the strategies developed:

1. **Fully Distributed:** In this strategy, the master process sends for each input the solution the worker has to satisfy while executing that input. This method is addressed as fully distributed since we can figure out a situation where all workers know which is the list of configurations and which are the models to be used. The master process does not necessarily compute the operating points list to send to them since each worker is able to calculate it by itself. The minimal communication strategy thus rely on the sending of the solution per input alone, or the global limit constraint itself. This approach seems to be the one with the less overhead and so the fastest one, but we actually cannot say it a priori since processes could lie on machines with different characteristics. Figure 4.4 shows the scheme of this method.

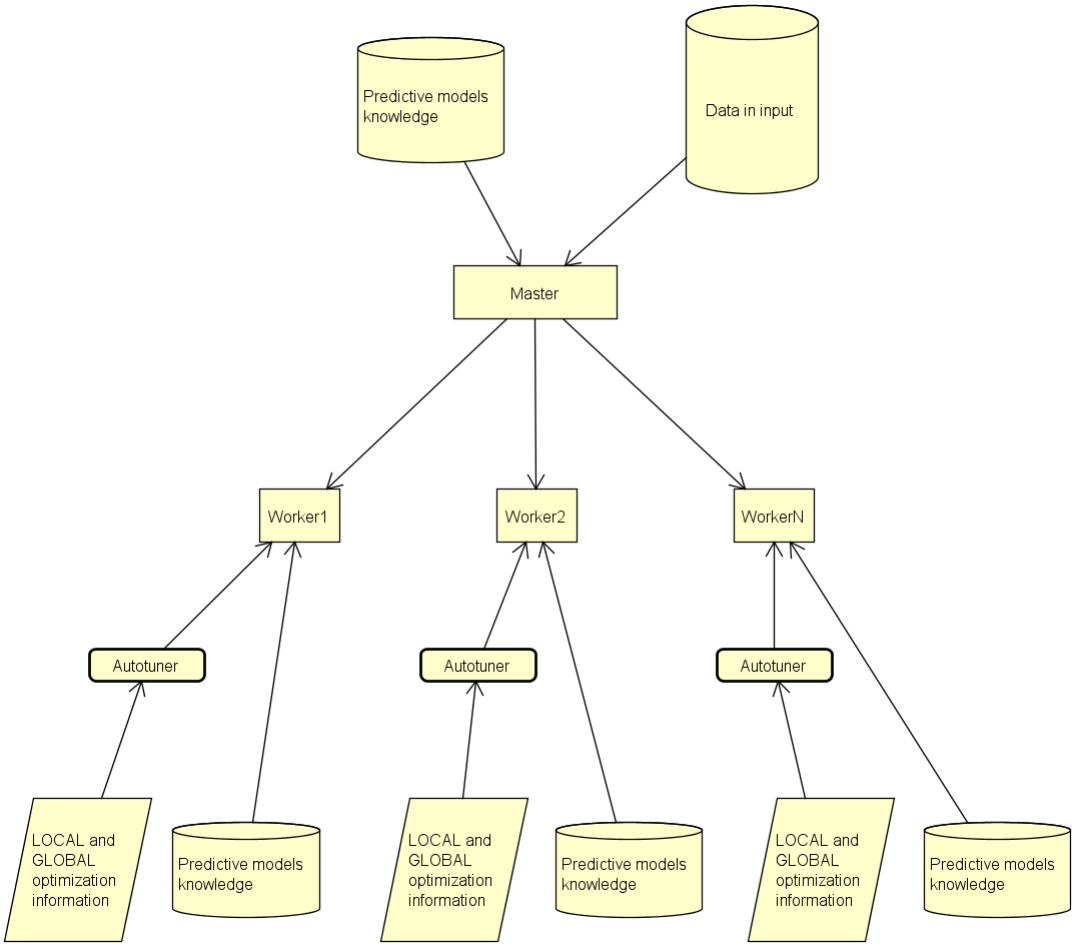

Figure 4.4: *Fully distributed knowledge location.*

2. **Partially Distributed:** This strategy assumes that each worker process does not have the predictive models knowledge, so it is not able to produce the list of operating points for the inputs it has to execute. The operating points list is thus provided by the master process, together with the part of the global metric-to-solution for that input.

   With this strategy, we can expect that the overhead for the communication is very high, but we still want to evaluate it since in an environment where the master process acts only as a data input dispatcher it may be faster if the workers spend their time to perform only the real computation. Maybe the master process can also lie on a more powerful machine with respect to the workers, and so with this strategy we could actually have some improvement. Figure 4.5 shows the scheme of this method.

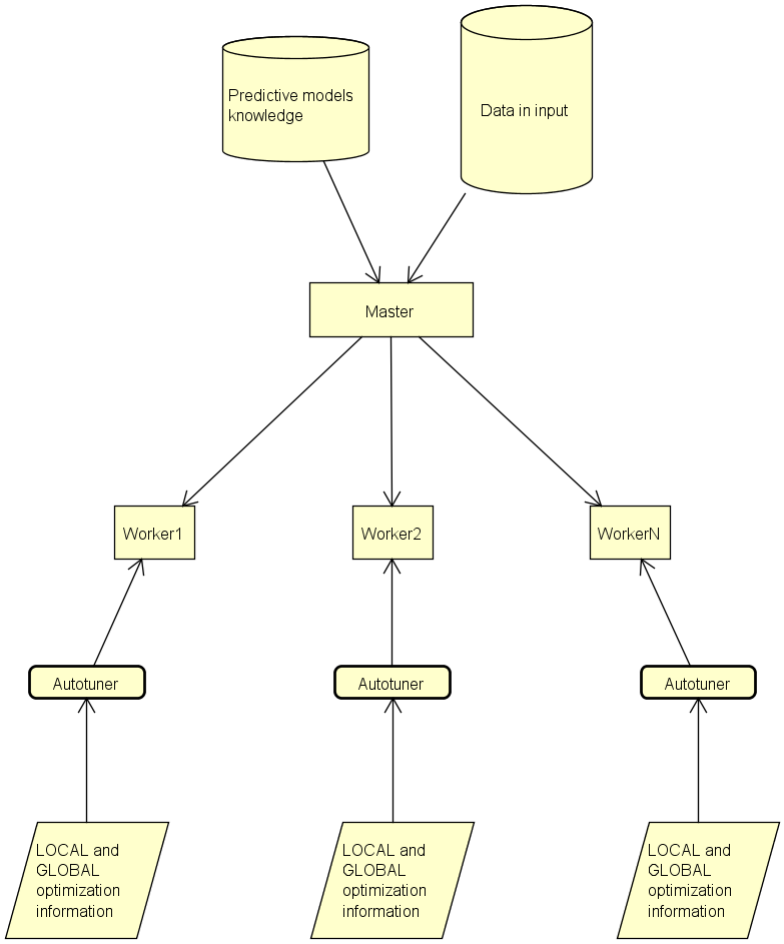

Figure 4.5: *Partially distributed knowledge location.*

3. **Partially Centralized:** This is a mixed strategy between the partially distributed and the fully centralized. As for the partially distributed strategy, here the master computes the solution per input and sends also the operating points list for every input, but this list is first filtered. The master sends only the operating points that lie in the feasible region traced by the solution per input. In this way the exchange of messages is lowered, and so also for the overhead, but the workers have now a limited space to explore in the autotuning phase. So, they could not have directly a further constraint on the global metric because each operating point in the list received is feasible. But with this strategy it can also happened that they do not find a feasible operating point in that list for their local optimization problem. Figure 4.6 shows the scheme of this method.

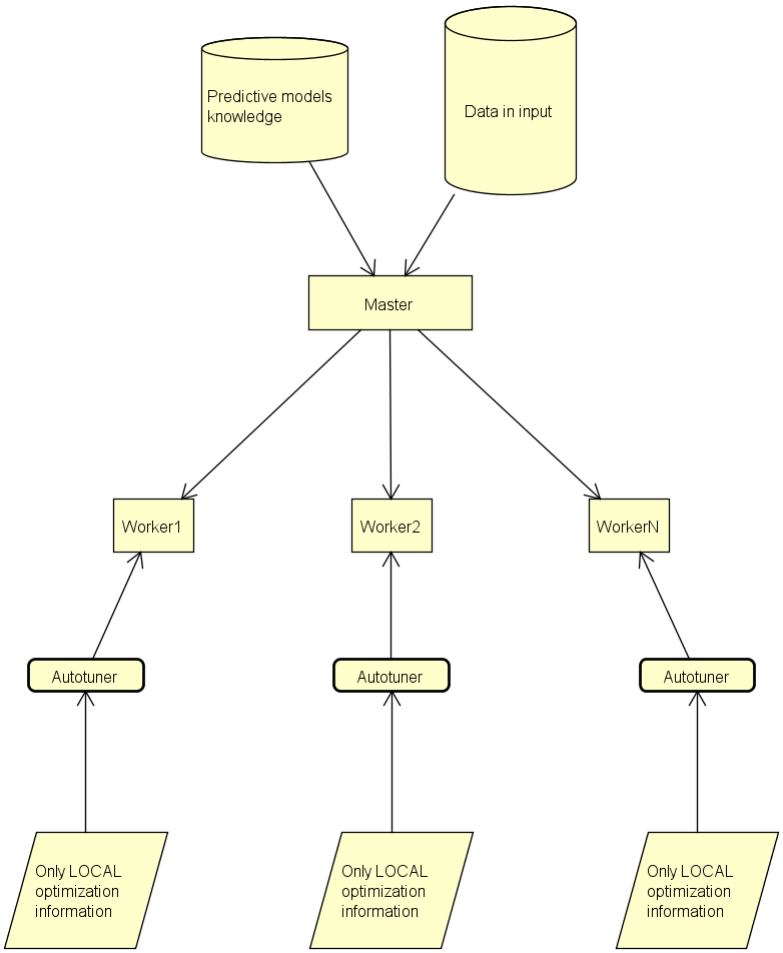

Figure 4.6: *Partially centralized knowledge location.*

4. **Fully Centralized:** All the phases for finding the best operating point for a precise input with precise input features are performed by the master process. Workers may not have autotuning knowledge nor the autotuner itself. They will wait only for the best configurations chosen for their inputs by the master.

   This strategy could be hardly applicable since master is almost always blocked in performing the autotuning for each input, but we have to take into account also this approach because we must not forget the environment we have addressed. It has no shared memory, so it is perfectly legal to consider also this method. Figure 4.7 shows the scheme of this method.

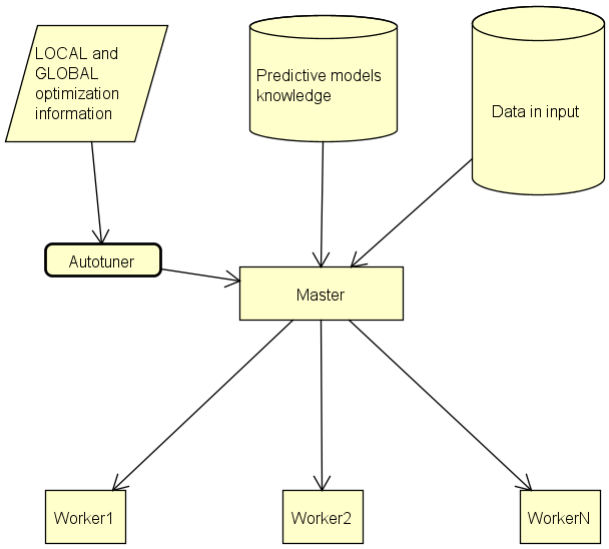

Figure 4.7: *Fully centralized knowledge location.*

Obviously, they would gain different execution times and different overheads, but we cannot say a priori which one could be the optimal strategy because we cannot say which performance characteristics will have the system on which the application will be executed. It can happen that the master process will lie on a very powerful system while the workers will lie in a worse environment, and, since the master essentially will act only as a dispatcher, maybe it is faster if it also computed the operating points list for each input for each worker, and the workers will use their time and power only for the real computing stage. As we have seen, the environment of our thesis can also be distributed and with non identical machines.

CHAPTER 5

The Ipazia Framework

5.1 Introduction

This chapter describes in details Ipazia, the framework implementation developed to evaluate the proposed methodology. The name of our framework derives from the italian name of Hypatia of Alexandria. Hypatia was a mathematician, astronomer, and philosopher of the ancient Greece. She devoted herself to learning and teaching in an environment considered new and hostile for a woman at that time. Similarly to what she dealt with the dissemination of knowledge and her discoveries to her disciples in such an environment, in our framework we want to do the same by first learning, discovering and then spreading the information gathered to the "disciples" workers in such a new distributed environment.

In this chapter, we first outline the global structure of our framework and the structure of a generic target application. Then, we discuss about the implementation and architectural choices, focusing on each module of the methodology described in Chapter 4. Finally, we describe the integration process in a target application.

5.2 General Description

Figure 5.1 shows the general overview of the framework architecture. The Ipazia framework is divided in two modules: the back-end and the front-end. The back-end performs the heavy computation phase, implementing the functionalities described in Chapter 4. The front-end is the interface exposed toward the target application. In particular, the application will interact only with the Ipazia object.

Our framework is agnostic with respect to the framework used by the slaves to select the most suitable configuration, according to the application requirements. In this thesis we used the mARGOt autotuner, which does not interact directly with our framework. It is the application that links the autotuner to Ipazia, as depicted in Figure 5.1.

The Ipazia framework is implemented as a C++ [49] library that is linked to the target application. This library requires external libraries and packages as dependencies. The uppermost dependency is obviously the MPI library described in Subsection 2.3.1. In particular we use the MPICH Hydra distribution [5]. Another important dependency is the RInside package [9]. The latter is a package that provides C++ classes which embed the R programming language in C++ applications. In particular, it provides an instance of an embedded R interpreter, which we use for several operations. This shared library supports MPI environment. To read configuration XML files, we choose to adopt the Pugixml parser [8].

5.3 Ipazia Back-End

Ipazia back-end implements the logic for all the methodology steps described in Chapter 4. It contains functionalities for estimating the data distribution in input, building the predictive models and getting the predictions from them, solving the global constraint given and computing the knowledge to send to the autotuner. A simple representation of the framework back-end structure is shown in Figure 5.2.

5.3.1 Model Building

This functionality is performed in the initialization of the framework, and it is required to build the predictive models for computing the values of the metrics of interest the application would consume, to be later used in the framework. It is implemented in the Predictive Model object by the *build* functionality. The *build* functionality takes in input the training set file, containing the list of the

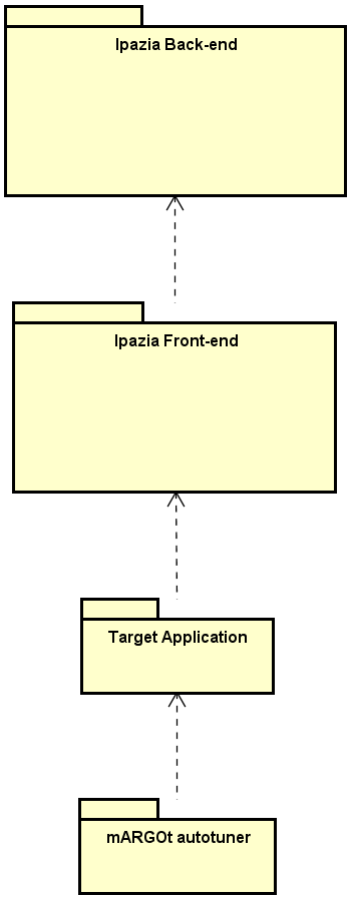

Figure 5.1: *General overview of Ipazia framework architecture.*

data that characterizes the application behavior, obtained in the profiling phase of the targetr application (Subsection 4.3.1). A better description of the structure of the training set file is shown in Section 5.5 and in Listing 5.5.

The model building functionality uses the RInside instance to access to the R object, which actually performs modeling operations. In this first release we implemented only the automatic build of completed linear regressions, but other types of automatic models generation can be further added. Therefore, the *build* functionality reads the data in the training set file and build a predictive regression model for each metric of interest present in the application. The regression model computed is a complete one and reduced by performing the *step* functionality provided by the R software.

The Predictive Model structure is shown in Figure 5.3.

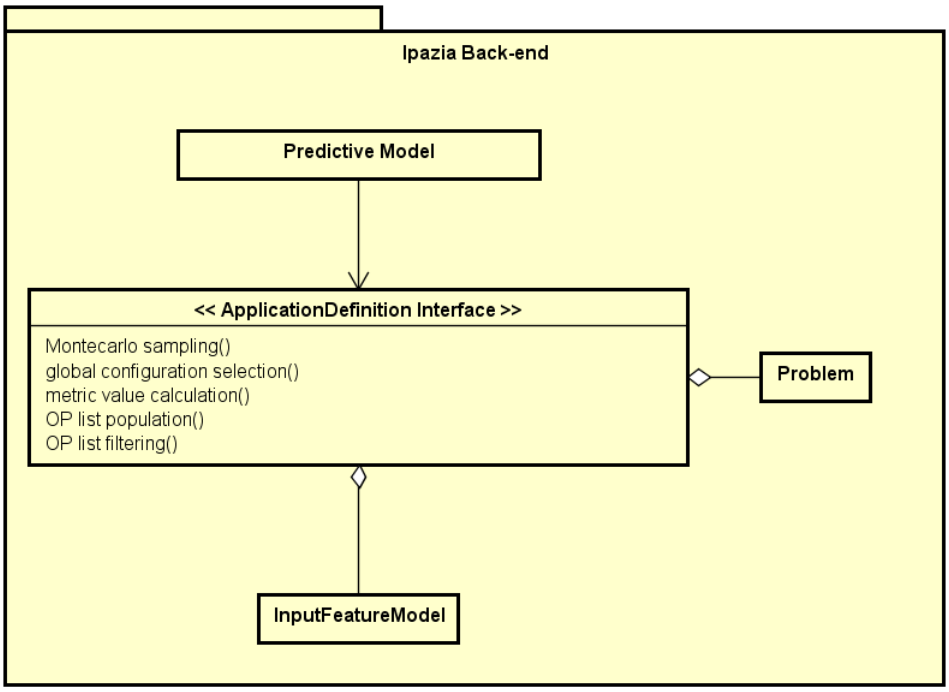

Figure 5.2: *Ipazia back-end structure.*

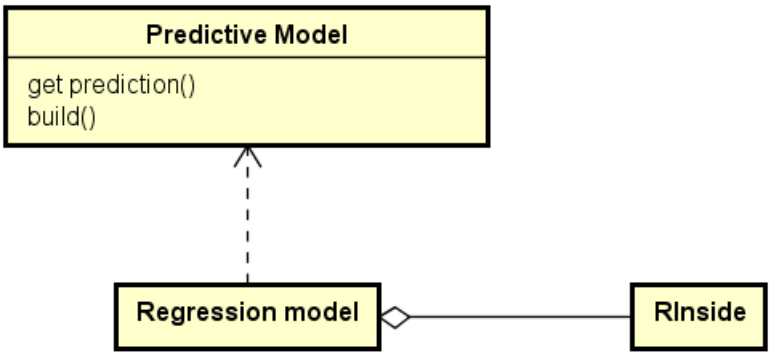

Figure 5.3: *Predictive Model object structure.*

5.3.2 Input Characterization

This functionality is required to build the Probability Density Functions for each input feature that is present in the application. This is implemented in the objects Application Definition Interface and InputFeature Model.

The Application Definition Interface contains a reference to the data in input of the application, and performs the Montecarlo sampling functionality. It extracts features from a set of samples of data in input. This requires that the data in input must be an iterable object. Each input of the iterable object must also

implement a templated function that returns the vector of all the features of that input. The framework samples from the iterable object and calls the templated method on each object extracted to obtain its features. The method throws an exception in case it has not been implemented.

5.3.3 Prediction Retrieving

This functionality is required to compute the prediction of a particular metric, according to a configuration and the input features. It is implemented in the Predictive Model object by the *get prediction* functionality. It takes as input the metric to predict, a configuration and the features of the input, and calculates the quantity of metric that an input with that features would consume.

5.3.4 Best Global Limit Configuration Selection

This functionality searches for the best global limit configuration that satisfies the global constraint according to the formula in Equation 4.3. It is implemented in the objects Problem, InputFeature Model and Application Definition Interface. The Problem object contains the information about characteristics of the global metric-to-solution to satisfy. These information are the metric of interest, its value and if it can be spread over the number of processes or not. The framework must know if the value of that metric can be parallelized or it is consumed by new resources. The Application Definition Interface uses the Problem information for performing the global configuration selection functionality. This functionality consists of using the PDFs previously calculated and stored in the InputFeature Model object and the *prediction retrieving* functionality, for selecting the best global limit configuration more appropriate for this input features distribution, that satisfies the global constraint stored in the Problem object.

The search of the global limit configuration can be done using two different strategies:

- *Exhaustive search strategy* performs a full search over the entire space of configurations and for each of them uses the predictive models to determine that configuration which acts as a watershed between configurations that are good (satisfy the constraint) and the ones that are not good (do not satisfy it) (Subsection 4.3.3), using the formula in Equation 4.3.

- *Precimonious-like strategy*, instead, creates an operating points list with the metric values filled with average values of the input features collected in the Montecarlo phase. For example, if the application has a global time-to-solution value to stay within, then this strategy sorts the list from the configuration with the higher predicted time value to the one with the lower

predicted time value. Then, it applies the formula in Equation 4.3 using this sorted list, and ends the search as soon as it finds a feasible configuration, that is a configuration whose time metric satisfy the global constraint. In this way, not all configurations and input features space must be examined. This is a heuristic.

5.3.5 Autotuning Knowledge Computation

This is requested to compute the knowledge that the autotuner needs in order to perform the optimization on a single input execution. It is implemented in the Application Definition Interface through the functionalities *Metric value calculation*, *Operating points list population* and *Operating points list filtering*:

1. **Metric value calculation** This functionality is used for retrieving the quantity of global metric for a precise input. It can be called after having found the limit configuration for the first time.
2. **Operating points list population** This functionality computes the operating points list for a precise input. This list is intended to be the list of all configurations given in the knobs list file and the predictions on the metrics of interest regarding the autotuning local solver phase. So, for example if the global constraint regards the energy metric but the autotuner is set to minimize error, also prediction on the error metric must be provided for each configuration.
3. **Operating points list filtering** This functionality filters the operating points list calculated with the previous function in order to delete those points that do not satisfy the constraint on the quantity of global limit metric their input would spend.

5.4 Ipazia Front-End

Ipazia front-end is composed of the public methods exposed by the Ipazia object.

Front-end methods must interact with the back-end part for allowing the target application to retrieve information regarding the knowledge to send to the autotuner. It interfaces with the *Application Definition Interface*, acting as a dispatcher between the different spreading knowledge methods.

5.4.1 Front-End Implementation

Ipazia front-end exposes several functionalities towards the application: these functionalities are all the public ones in the Ipazia object. The front-end structure is shown in Figure 5.4.

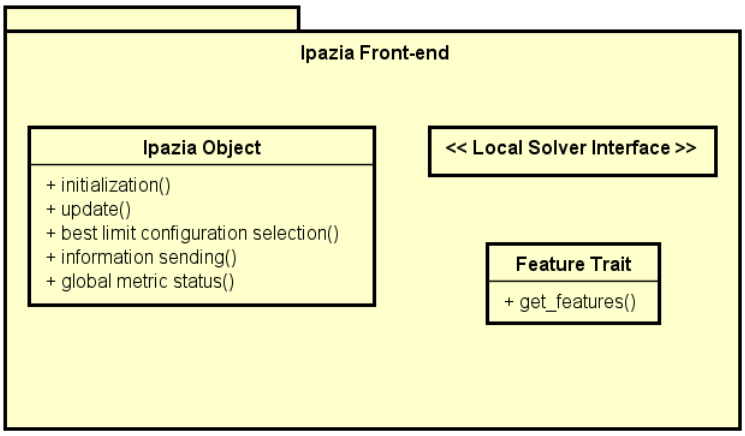

Figure 5.4: *Ipazia front-end structure.*

The *initialization* functionality initializes the Ipazia framework environment, setting all the information related to the target application, read from both the configuration files and the parameters passed in this method. It also needs a function as a parameter, in order to let the framework be able to retrieve the value of the consumed global metric-to-solution given, in any instant of time.

The *best limit configuration selection* functionality starts the framework procedure to calculate the optimal limit configuration.

The *global metric status* functionality alerts Ipazia framework of the number of inputs that have been completed until that moment. This is required in order to know when the framework has to make the update of the limit configuration.

The *information sending* functionality deals with the sending of autotuning knowledge according to the spreading knowledge method selected before. It sends different information based on the requirements on the worker side, that are well explained in the following Table 5.1.

The *update* functionality returns the configuration selected for an input, respecting both global and local constraints.

Local Solver Interface

This Local Solver Interface must be implemented in the target application. It provides the methods for making interfacing the Ipazia framework with the autotuner of the application. The interface stores the information that the autotuner needs in order to find the best operating point for a precise input. The parameters of the interface are the list of the operating points, and the constraint value that the computation of the input at issue must satisfy. This information is computed by the Ipazia framework. Developer of the target application must implement two functionalities: the *autotuner initialization* functionality, that initializes the

autotuner once for every process involved in the application, and the *solve* functionality, that starts the autotuner passing it the information required, calculates the best operating point for the input that satisfies both the global and the local constraints and returns the configuration chosen. In our implementation we use mARGOt autotuner. Local constraints must be defined directly in the configuration of the autotuner used.

Figure 5.5 shows the structure of the Local Solver interface.

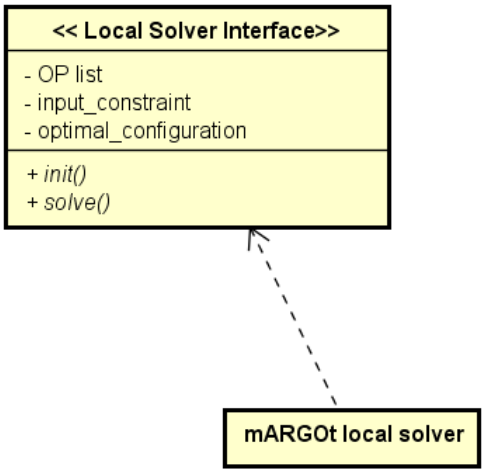

Figure 5.5: *Local Solver interface structure.*

5.4.2 Workflow

Figure 5.6 shows the workflow of a target application that uses Ipazia library. Remember that a target application is made up of a master process that acts as a dispatcher of inputs to worker processes, which actually do computation on the inputs.

Master process of target application acts as a planner on the quantity of metric-to-solution to be used by every input and acts as a dispatcher between the different methods of spreading the autotuning knowledge from master to every worker. The master can make calls relative to the setup of the framework environment and it also provides to Ipazia the information related to the global metric-to-solution status and the inputs computed when required. All these primitives are executed in a synchronous way in the application. Master process starts the first initialization of the optimal limit configuration. For doing this computation, Ipazia framework starts a separate thread, thus not blocking the application. The following operations of updating of the global limit configuration are also done in this separate thread by refreshing it.

Worker process performs the operations for initializing its Ipazia environ-

ment, receives inputs from the master process and then it calls the *update* functionality for getting the local optimal configurations with which computes the inputs received. Ipazia methods used in the worker process are all executed synchronously with the target application.

5.4.3 Execution Environments

In order to obtain the local optimal configuration starting from the one found as the best limit on average by the master, four steps must be done (actually three steps, except for one method) and according to the method selected for spreading autotuning knowledge to workers these steps can be performed by master or worker processes (Subsection 4.3.6).

Table 5.1 shows the correspondences between method selected and where the computation of that step is done. In the fully distributed environment, workers have knowledge related to the predictive models for filling the list of operating points to pass to the autotuner, so they can perform the *op list population* operation, which calculates the operating points list for a precise input, using the predictive models. After that, the *solve* functionality starts the autotuner and returns the configuration chosen that satisfies both the global and the local constraints. Workers in this environment must receive only the information on the configuration selected by the master (FULLY DISTRIBUTED A version), or on the quantity of metric reserved for each input (FULLY DISTRIBUTED B version), which are computed in the master by the *metric value calculation* functionality.

In the partially distributed environment, however, workers do not have predictive models knowledge, so master must send them both the quantity of metric reserved for each input and the list of operating points for each input. Once received all these information, workers can call the *solve* functionality, which acts in the same way as in the fully distributed environment.

The partially centralized environment is equal to the partially distributed one, except for the fact that the list of operating points that the master must send to the worker is a filtered one. More precisely, in this environment the master sends only the operating points that satisfy the global constraint. This leads workers to apply the autotuning phase only regarding their local constraints, since the global one is automatically satisfied.

In the fully centralized environment, the master performs all operations from the generation of the solution and the operating points list for each input to the execution of the autotuning phase on that input. The master must only send the configuration chosen for each input and the worker receiving it can soon do the computation of that input with the knob settings received. The worker may not provide the autotuner to the framework.

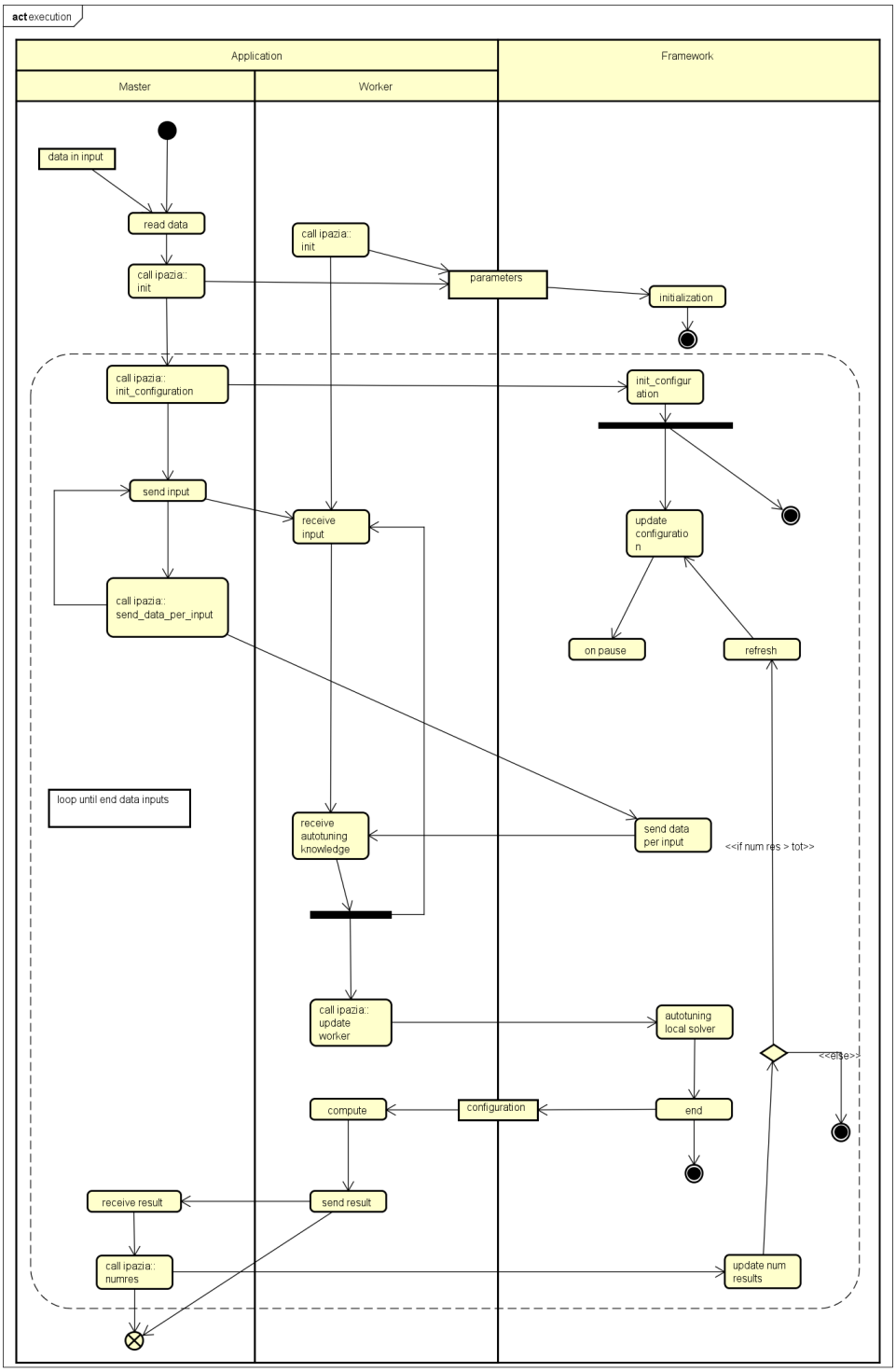

Figure 5.6: *Target application workflow.*

Table 5.1: *Summary of spreading autotuning knowledge functionalities*

Methods	Steps	Master	Worker
FULLY DISTRIBUTED A	metric value calculation	X	✓
	op list population	X	✓
	op list filtering	-	-
	solve	X	✓
FULLY DISTRIBUTED B	metric value calculation	✓	X
	op list population	X	✓
	op list filtering	-	-
	solve	X	✓
PARTIALLY DISTRIBUTED	metric value calculation	✓	X
	op list population	✓	X
	op list filtering	-	-
	solve	X	✓
PARTIALLY CENTRALIZED	metric value calculation	✓	X
	op list population	✓	X
	op list filtering	✓	X
	solve	X	✓
FULLY CENTRALIZED	metric value calculation	✓	X
	op list population	✓	X
	op list filtering	-	-
	solve	✓	X

5.4.4 Communication Pattern

The framework has a communication object that wraps all the MPI communication used by this framework. Ipazia uses its own MPI communicator to send and receive MPI messages. This communicator must be built on a group of processes including master and workers, obviously. Primitives provided by this file are listed in Figure 5.7.

They are related to the different information the master must send to the workers, accordingly to the environment in which the application executes. Note that this file also contains threaded queues for storing autotuning knowledge received in the workers.

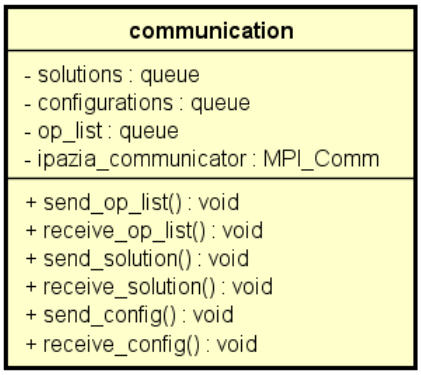

Figure 5.7: *Ipazia communication architecture.*

5.5 Integration in the Application

The framework requires several information that are provided through configuration files. It must have the knowledge about all the metrics of interest that are used in the application, all the knobs and their possible settings, and information on what are and how are the global constraints the whole application must meet. The framework also requires in input a training set containing the behavior of the application in terms of metric trend with respect to knob settings and input features. It must be used for fitting the model in the model building functionality.

An example of a configuration file is shown in Listing 5.1. This file is divided in three sections: the first section (lines 3-8) describes the metrics of interest for the target application. Each metric is represented by the `<Metric>` element, in which there must be specified the name of the metric in the "name" attribute. The metric object of the global constraint must specified further information, added in the element `<Problem>` (line 6). The `<Problem>` element must specify the value of the global constraint in the "constraint" attribute, the type of inequality

the application must apply for satisfy the global constraint is specified in the "type" attribute. There are four possible type value to specify:

1. LESS: the overall execution of the target application must produce a global metric quantity less than the value provided in the configuration file.
2. LESSOREQUAL: the overall execution of the target application must produce a global metric quantity less or equal than the value provided in the configuration file.
3. GREATER: the overall execution of the target application must produce a global metric quantity greater than the value provided in the configuration file.
4. GREATEROREQUAL: the overall execution of the target application must produce a global metric quantity greater or equal than the value provided in the configuration file.

The attribute "spreadable" must specify if the metric of the global constraint can be spread over the number of processes (yes) or not (no). The framework must know if the value of that metric can be parallelized or it is consumed by new resources. For example, the time metric can be spread, since, increasing the number of processes decreases the execution time, while the power consumption metric cannot be spread, since adding a process would increase the quantity of power consumed.

The second section (lines 9-12) describes the features that can be extracted by the input. Each feature is represented by the `<InputFeature>` element, in which there must be specified the name of the feature in the "name" attribute.

The third section (lines 13-17) describes the knobs the application has. Each knob is represented by the `<Knob>` element, in which there must be specified the name of the knob in the "name" attribute.

An example of a file containing the list of the possible knob settings that the application can use, is shown in Listing 5.2. It is made of a list of `configuration` elements (lines 3-9 for the first element of the list). Each `configuration` element is represented by the knob settings and their values for that configuration, through a list of `parameter` elements (lines 5-7 for the first element of the list). Each parameter must contain the name of the knob in the "name" attribute and the value of the knob of the relative configuration in the "value" attribute.

An example of a training set file is shown in Listing 5.5. This training set must have columns of all knobs, input features and metrics used in the applications.

Listing 5.1: *config_file.xml*

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <Ipazia FormatVersion="1">
3    <Metrics>
4      <Metric name="Accuracy"/>
5      <Metric name="Time">
6        <Problem constraint="25000000" type="LESS" spreadable
           ="yes"/>
7      </Metric>
8    </Metrics>
9    <InputFeatures>
10     <InputFeature name="NUM_ATOM_MOLECULE"/>
11     <InputFeature name="NUM_ATOM_LIGAND"/>
12   </InputFeatures>
13   <Knobs>
14     <Knob name="precision"/>
15     <Knob name="alpha_skip_factor"/>
16     <Knob name="beta_skip_factor"/>
17   </Knobs>
18 </Ipazia>
```

The order is not important provided that column names follow fixed patterns explained below.

- *pattern knob column label:* k<occupied position> in the knobs list in the configuration file (Listing 5.1 lines 14-16);
- *pattern input feature label:* i<occupied position> in the input features list in the configuration file (Listing 5.1 lines 10-11);
- *pattern metric label:* m<occupied position> in the metrics list in the configuration file (Listing 5.1 lines 4-7);

In this example there are three knobs, k0, k1 and k2, two input features i0 and i1, and two metrics of interest m0 and m1.

```
1 "k0","k1","k2","i0","i1","m0","m1"
2 0,0,0,123,21,4.67678,9775007
3 0,0,0,57,30,3.14085,6013007
4 0,0,1,74,6,7.01668,1414247
5 0,0,1,87,12,4.80814,3051925
6 0,0,2,74,6,7.01668,1732096
7 0,0,2,87,12,4.80814,2443932
```

Listing 5.2: *knob_file.xml*

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<configurations>
 <configuration>
  <parameters>
   <parameter name="precision" value="0"/>
   <parameter name="alpha_skip_factor" value="0"/>
   <parameter name="beta_skip_factor" value="0"/>
  </parameters>
 </configuration>
 <configuration>
  <parameters>
   <parameter name="precision" value="0"/>
   <parameter name="alpha_skip_factor" value="2"/>
   <parameter name="beta_skip_factor" value="1"/>
  </parameters>
 </configuration>
 <configuration>
  <parameters>
   <parameter name="precision" value="1"/>
   <parameter name="alpha_skip_factor" value="0"/>
   <parameter name="beta_skip_factor" value="1"/>
  </parameters>
 </configuration>
</configurations>
```

```
8  0,0,3,123,21,4.67678,2265001
9  0,0,3,57,30,3.14085,1932382
```

5.5.1 Master Side

The following code shows the code that the developer must include in the application code of the master process:

```
1  // Master process Ipazia calls
2  #include <ipazia/ipazia.hpp>
3
4  //GLOBAL declaration of IpaziaObject
5  ipazia::IpaziaObject my_ipazia_object;
6
7  my_ipazia_object = new ipazia::IpaziaObject();
8
9  // Initialization
10 my_ipazia_object->init(ipazia::Method::
      PARTIALLY_DISTRIBUTED, my_rank,
      computational_resources, ipazia::structures::
      Search_Type::EXHAUSTIVE, input_list,
      get_metric_value_consumed, path/to/config_file
      , path/to/knobslist_file, path/to/
      trainingset_file, mARGOt_local_solver);
11
12 // Find the global limit configuration the first
      time
13 my_ipazia_object->init_configuration();
14
15 // Send framework data for each input
16 my_ipazia_object->send_data_per_input(input,
      worker_rank);
17
18 // Receive information on the number of
      computations
19 // already completed
20 my_ipazia_object->receive_computation();
```

First, developers need to include the header of the framework that states the data structures used (line 2).

They must now declare the global object IpaziaObject as an interface to our framework (line 5) and create it (line 7).

The first operation to do after is the initialization of the framework (line 10). This function requires several parameters:

the first parameter (ipazia::Method::PARTIALLY_DISTRIBUTED) is the enum value of the execution environment where the application is launched. It can be one of the following:

- FULLY_DISTRIBUTED_A;
- FULLY_DISTRIBUTED_B;
- PARTIALLY_DISTRIBUTED;
- PARTIALLY_CENTRALIZED;
- FULLY_CENTRALIZED;

Their differences are described in Table 5.1 and in Subsection 4.3.6.

The second parameter (my_rank) is the rank of the actual process. In this case the rank is 0, since this process is the master.

The third parameter (computational_resources) is the number of computational resources of the application. A computational resource is a process that actually performs operations on the input. Therefore, for our target application it is the size of the MPI structure minus 1, the master process, that acts only as a data dispatcher.

The fourth parameter (ipazia::structures::Search_Type::EXHAUSTIVE) is the enum value of the search strategy selected for doing the computation of the best global limit configuration. It can be one of the following:

- EXHAUSTIVE: the search strategy selected is an exhaustive one.
- PRECIMONIOUS; the search strategy selected is an heuristic, based on the Precimonious strategy explained in [28].

The details in the behavioral differences are explained in Subsection 5.3.4.

The fifth (input_list) parameter is the whole data in input of the application.

The sixth parameter (get_metric_value_consumed) is the function that returns the quantity of the global metric spent until that moment.

The seventh parameter (path/to/config_file) is the path to the configuration file.

The eighth parameter (path/to/knobslist_file) is the path to the file with the list of the knob settings.

The nineth parameter (path/to/ trainingset_file) is the path to the training set file.

The last parameter (mARGOt_local_solver) is the pointer to the local solver the application would use to perform the local autotuning phase. We use the mARGOt autotuner as our local solver.

After the initialization, developers must start the search of the global limit configuration (line 13).

Then, in the input dispatching phase it is required to compute and send the Ipazia framework information to each workers, for every input sent to them (line 16).

The last operation the master must perform sends to Ipazia framework information on the status of the input computation (line 20). It informs the framework of the number of inputs completed until that moment.

5.5.2 Worker Side

The following code shows the code that the developer must include in the application code of a worker process:

```
1  //Worker processes Ipazia calls
2  #include <ipazia/ipazia.hpp>
3
4  //GLOBAL declaration of IpaziaObject
5  ipazia::IpaziaObject my_ipazia_object;
6
7  my_ipazia_object = new ipazia::IpaziaObject();
8
9  // Initialization
10 // Local Solver in Worker is needed except
11 // for Method FULLY CENTRALIZED
12 my_ipazia_object->init(ipazia::Method::
      PARTIALLY_DISTRIBUTED, my_rank,
      computational_resources, path/to/config_file,
      path/to/knobslist_file, path/to/
      trainingset_file, mARGOt_local_solver);
13
14 // Compute and receive optimal configuration for
15 // my_input and
16 // both global and local problems
```

```
17 my_configuration = my_ipazia_object->
      update_worker(input);
```

First, developers need to include the header of the framework that states the data structures used (line 2).

They must now declare the global object IpaziaObject as an interface to our framework (line 5) and create it (line 7).

The first operation to do after is the initialization of the framework (line 12). This function requires several parameters:

the first parameter (ipazia::Method::PARTIALLY_DISTRIBUTED) is the enum value of the execution environment where the application is launched. It can assume the same values as for the master process.

The second parameter (my_rank) is the rank of the actual process. In this case the rank cannot be 0, since this process is not the master.

The third parameter (computational_resources) is the number of computational resources of the application. A computational resource is a process that actually performs operations on the input. Therefore, for our target application it is the size of the MPI structure minus 1, the master process, that acts only as a data dispatcher.

The fourth parameter (path/to/config_file) is the path to the configuration file.

The fifth parameter (path/to/knobslist_file) is the path to the file with the list of the knob settings.

The sixth parameter (path/to/trainingset_file) is the path to the training set file.

The last parameter (mARGOt_local_solver) is the pointer to the local solver the application would use to perform the local autotuning phase. We use the mARGOt autotuner as our local solver. In a worker process, this is an optional parameter, since in the fully centralized environment the worker cannot have its local autotuner. It is required in all the other environments.

The last operation (line 17) the worker must perform calculates the best operating point using autotuning information received by the master or computing in this method, and the local solver.

CHAPTER 6

Experimental Results

6.1 Introduction

In this chapter we show experimental results collected to assess the proposed methodology and framework, and also to validate the effectiveness of our contribution in the HPC problem described in Chapter 4. First, we introduce the experimental setup used and show results related only to the global solver part that has to compute and spread the global constraint. Then we describe the benchmark application used for performing our experiments and propose their results and analysis. We also show information regarding overheads of the different autotuning spreading knowledge methods.

6.2 Experimental Setup

6.2.1 Intel Core i7 5500u

For the initial experiments we used a heterogeneous machine containing a dual-core processor Intel Core i7 5500u, with two logical cores per physical, clock-speed of 2.4 GHz with turbo speed up to 3.0 GHz.

We spawn only one application process on each core, and two fixed threads in each process.

We chose MPICH as our MPI implementation, the RInside library and Rcpp library which contains instance of R software for computing predictive models and mARGOt autotuner as the local solver of the application.

6.3 Assessing the Limit Configuration Selection

To assess and show the effectiveness of the global solver (planner) developed, we first perform some experiments using synthetic functions instead of real workloads. Applications considered simply have a master that sends inputs to workers. These inputs have different features. Metrics of interest are the accuracy level and the execution time. The predictive models for these metrics are simply mathematical functions of knobs and features.

More in detail, we performed these experiments using different function types: linear, quadratic and sinusoidal, in order to collect results of the behavior of our framework under different types of synthetic functions. In this chapter we show results from the experiments with the quadratic and sinusoidal functions. The metric values can be retrieved as follows:

$$Time = f_1(x, f) \tag{6.1}$$

$$Accuracy = f_2(x, f) \tag{6.2}$$

Where:
x: is the vector of knobs.
f: is the vector of input features.

Once received inputs, workers must simply call the `update_worker()` Ipazia function using as local solver the mARGOt autotuner, and with the configuration received they compute the Time for that input and do a sleep of Time milliseconds.

We perform several runs under different environment conditions:

1. First, we executed experiments under completely predictable conditions. The metrics are those extracted as Equation 6.1 and Equation 6.2.

2. Then, we introduced a Gamma noise of 10%. The expected time for an input is taken from the Equation 6.1 plus a random value extracted from a Gamma distribution.

3. Then, we perform the same experiment with a medium level of Gamma noise, of 25%.

4. Finally, we introduced a very high level of of Gamma noise, of 50%, in order to really stress the framework and see how it behaves in this extreme condition.

Noise introduced is modeled as a Gamma distribution. We chose this distribution since several works show that this is the best distribution to model time to solution behavior under unpredictable events. Indeed, it is not symmetric like the Gaussian, and so it captures in a better way the fact that the majority of unpredictable events led to increase execution time rather than decrease it. The Gamma distribution function is described in Equation 6.3 and Equation 6.4.

$$f(x; k, \theta) = x^{k-1} e^{\frac{-x}{\theta}} \frac{1}{\theta^k \Gamma(k)} \tag{6.3}$$

Where:

$$\Gamma(k) = \int_0^\infty s^{k-1} e^{-s} ds \tag{6.4}$$

is the Gamma function of Euler.

Below there is the cumulative distribution function formula:

$$F(x; k, \theta) = \int_0^x f(u; k, \theta) du = \frac{\gamma(k, \frac{x}{\theta})}{\Gamma(k)} \tag{6.5}$$

Gamma distribution has mean $\mu = k\theta$ and variance $\sigma^2 = k\theta^2$.

In our experiments, both parameters are positive real numbers with a shape parameter k and a scale parameter θ.

Figure 6.1 and Figure 6.2 show examples of PDFs and CDFs of Gamma distribution with several k and θ values.

6.3.1 Quadratic Function

$$Time = f_1(x, f) = x^2 + f \tag{6.6}$$

$$Accuracy = f_2(x, f) = x \tag{6.7}$$

There is one knob x and one input feature f. Knob x can assume values from 4 to 13. Feature f can assume values from 5 to 205.

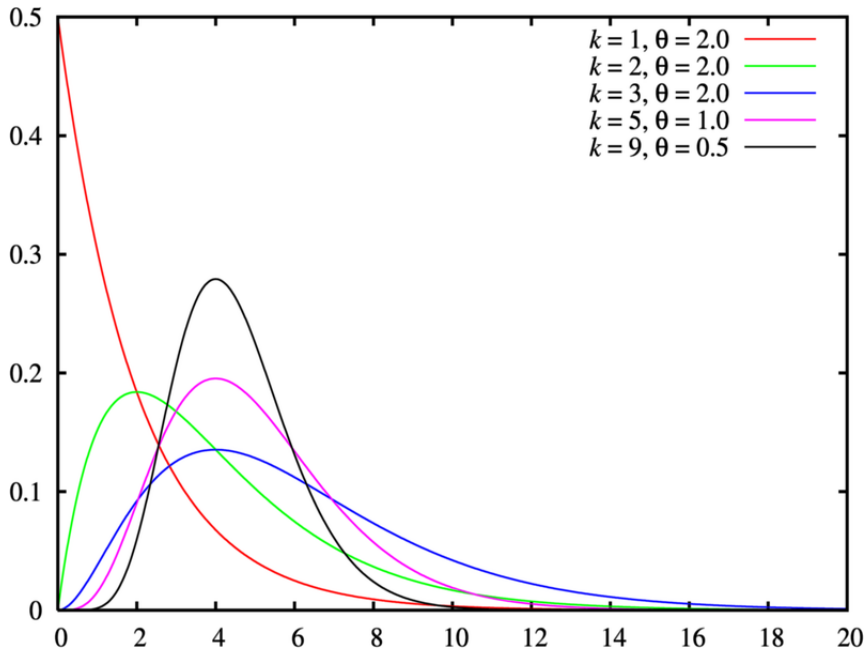

Figure 6.1: *Gamma probability density function.*

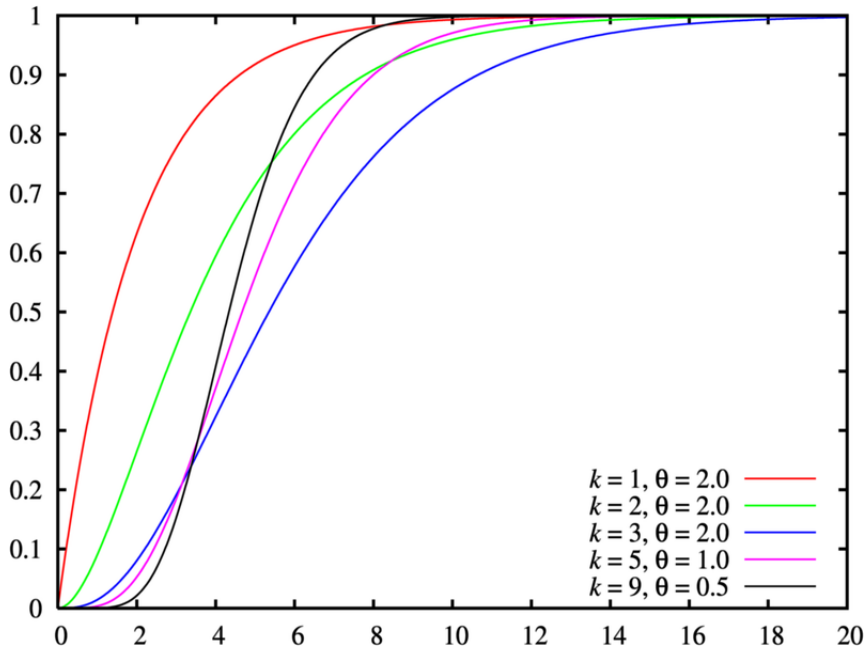

Figure 6.2: *Gamma cumulative density function.*

Synchronization points are set every 20% of inputs. We performed experiments using 1000 inputs, so we recomputed the limit configuration every 200 results received.

Values for the different executions are given for:

No noise introduced:

$$k = Time; \tag{6.8}$$

$$\theta = 1; \tag{6.9}$$

Low noise introduced:

$$k = Time; \tag{6.10}$$

$$\theta = 1,1; \tag{6.11}$$

Medium noise introduced:

$$k = Time; \tag{6.12}$$

$$\theta = 1,25; \tag{6.13}$$

High noise introduced:

$$k = Time; \tag{6.14}$$

$$\theta = 1,5; \tag{6.15}$$

Charts in Figure 6.3, Figure 6.4 and Figure 6.5 show the noise amplitude introduced in three different experiments. This noise amplitude is represented as a quantity of time added to the execution time metric of a precise input, for all the data in input.

Figure 6.3 contains the noise amplitude introduced in the experiment with low noise. There are some points where the noise introduced decreases the execution time of the data, but in the majority of them it gives a positive contribution to the time increment.

Figure 6.4 represents the noise amplitude introduced in the experiment with a medium level of noise. We can see that the average of noise values introduced is higher that the one of the previous chart, and so is the mean amplitude, too. The same considerations apply to Figure 6.5, containing values for the high noise level case. This is because we have not only increased the variance value of the Gamma distribution, but we have also moved its mean value upper. Our purpose is to stress at maximum the framework, in feasible noise conditions and also in excessively bad ones. We must say that, in the environment the framework is addressed to, it probably would never encounter conditions with such bad levels of noise as the ones in our medium and high noise cases, but we perform experiments also in those conditions because we want to explore the behavior of Ipazia framework at very extreme conditions.

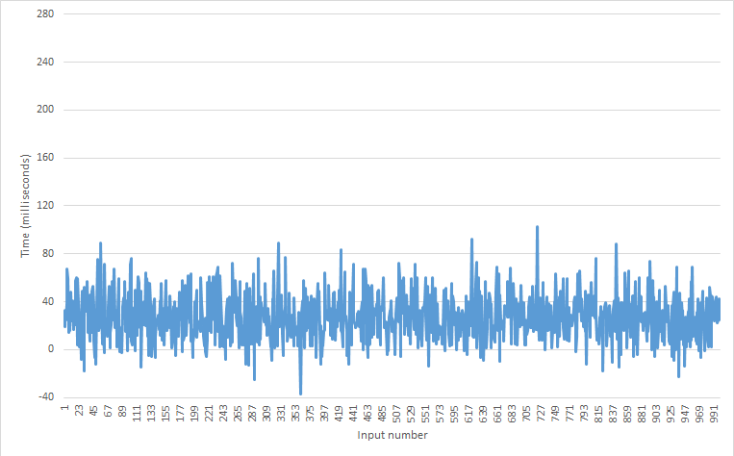

Figure 6.3: *Low noise introduced: time by varying the input.*

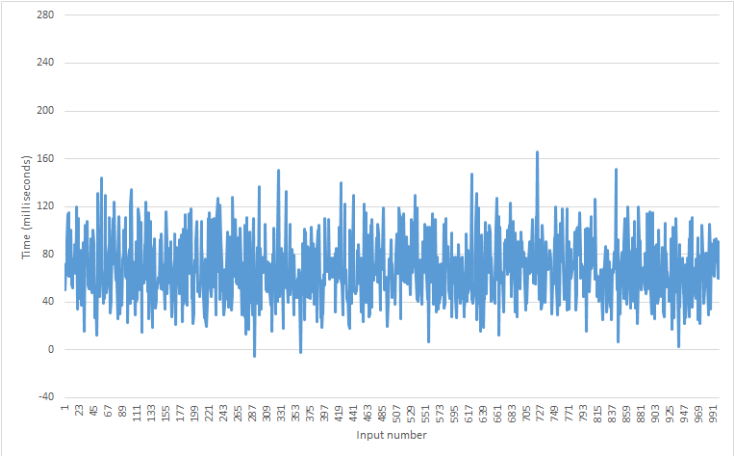

Figure 6.4: *Medium noise introduced: time by varying the input.*

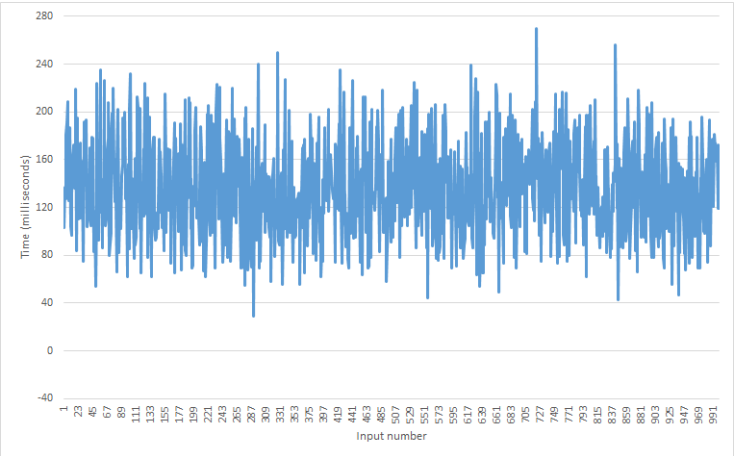

Figure 6.5: *High noise introduced: time by varying the input.*

For demonstrating the benefits of our framework, we have taken several values of time to solution and we have left to the framework the task of selecting the best limit configuration, according to the time-to-solution and the data features distribution. We have repeated this experiment for several noise values: no noise introduced, low, medium and very high noise introduced. Values collected are shown in Figure 6.6, Figure 6.7 and Figure 6.8. The accuracy level has been normalized between 0, which corresponds to the minimum level of accuracy, and 100, which corresponds to the maximum level of accuracy.

The time-to-solution values are expressed in milliseconds.

We did those experiments using seven time-to-solution values that cover the application behavior from the point where it is no longer possible to increase the accuracy level to the point where it is no longer possible decreasing it, taking as a reference the behavior of the application in perfect conditions. Therefore, in charts we show results collected in these two extreme points (where time-to-solution is equal to 300000 and where time-to-solution is equal to 100000, respectively), in points just after (time-to-solution equal to 350000) and just before (time-to-solution equal to 90000), respectively, and in three intermediate points, one taken in the middle of the two extreme points (time-to-solution equal to 200000), one taken 25% after (time-to-solution equal to 250000) and the last taken 25% before it (time-to-solution equal to 150000).

Figure 6.6 shows the normalized accuracy values by varying the time-to-solution given to the application, collected from experiments with different noise levels introduced.

Figure 6.7 shows the percentage of remaining time by varying the time-to-solution given to the application, always from data collected in all the experiments with different noise levels: no noise, low, medium and high noise.

Finally, Figure 6.8 shows the percentage of inputs successfully completed by the application within the given time-to-solution, by varying the time-to-solution constraint itself. As for the previous charts, also this experiment has been repeated for the four different levels of noise. This percentage is less than 100% in correspondence with a negative percentage of remaining time.

As we expected, the trend of the accuracy curve obtained in perfect conditions has an average accuracy level higher than the ones of the other experiments in quite all the points. The trend of the accuracy curve for the low noise case is very close to the trend in perfect conditions, in particular in the behavior in extreme values of the time-to-solution. As a consequence of the huge quantity of noise introduction, the accuracy trends of medium and high level cases have an average accuracy very low with respect to the other conditions. In particular, regarding results with the high level noise introduction, even with the greater

time-to-solution constraint the average accuracy is very far from the maximum value. However, these are exactly the results we expected from the experiments with this quadratic function.

If the remaining percentage of time is much greater than 0%, it means that Ipazia framework has taken too many margins in the calculation of the optimal limit configuration, and it has selected an accuracy level smaller than the best appropriate level for that time-to-solution and that workload. This should never happen, unless the accuracy level is already at the maximum. In this case, increasing the time-to-solution does only cause the increase in the percentage of remaining time. In the charts this situation is represented for time-to-solution values equal to 300000 and 350000 milliseconds. Regarding the experiments with no noise and with a low level of noise introduced, in the time-to-solution value of 300000 milliseconds, the relative accuracy is at maximum and the percentage of remaining time is nearly 0%. Increasing the time-to-solution until 350000 milliseconds has obtained the only effect of bring the percentage of remaining time above the 20% in the no noise experiment, and above the 10% in the low level noise experiment. Regarding the experiments with a medium and a high level of noise this does not happen in the same way, since the noise introduced in the computation of the final part of the workload is so high that Ipazia framework could not have the time to absorb this noise and do the recompuptation of the best limit configuration. A possible solution consists in the increase of the number of synchronization points, that can be set for example, every 10% or even 5% of the cardinality of data in input. As we have said previously, the situation with medium and high level of noise were treated only in order of stress the framework at the maximum, but they probably are not realistic situations in which Ipazia framework might operate.

However, if Ipazia fails in ending the computation within the time-to-solution constraint, it may be due to two reasons:

1. Time-to-solution given is actually not sufficient to compute the whole data in input. This situation can be seen in the experiments with no and low noise level, when time-to-solution given is small, precisely in points with time-to-solution equal to 100000 and 90000 milliseconds.

   With 100000 milliseconds as time-to-solution constraint, the accuracy level for both the experiments with no and low noise values is at the minimum value, but the percentage of the remaining time is less than 0%, breached 10% and 30%, respectively, and, as a consequence, the percentage of inputs computed within the given time-to-solution does not reach the 100%. Being already at the minimum level of accuracy, Ipazia framework can do nothing

in order to satisfy the time-to-solution constraint. Then, further decreasing the time-to-solution to 90000 milliseconds has obtained only effects on the percentage of remaining time and inputs completed, decreasing them.

2. The framework made a mistake in the calculation of the estimate distribution of the workload or it was not able to handle the quantity of noise introduced. The second situation is very well visible in experiments with medium and high noise values. In the point with time-to-solution equal to 200000 milliseconds, we can see that, meanwhile the percentage of the remaining time of the none and low noise levels is equal to 0%, with accuracy levels that not reached the maximum value, increasing the noise level led to a further decreasing of the accuracy level and the percentage of the remaining time, which goes below the 0%. It can be seen in a very evident way in the high noise curve: the percentage of the inputs successfully completed is just above the 80% for the 200000 time-to-solution constraint, and close to the 90% in the point with time-to-solution equal to 250000, where, instead, in the experiments with medium, low and no noise introduced, Ipazia successfully leads the application to complete the 100% of the input computations.

We must note that, nevertheless, in the remaining situations, Ipazia reaches in making the application satisfy the given time-to-solution by decreasing the accuracy level for the input computation, mostly in the experiments with no and low noise values introduced. The curves assume the trend that we expected: the accuracy curves trend decrease with the decreasing of the time-to-solution given until reaching the minimum possible level. Increasing the noise introduced, the problem becomes intractable, while still preserving the trend of the curves that we expected.

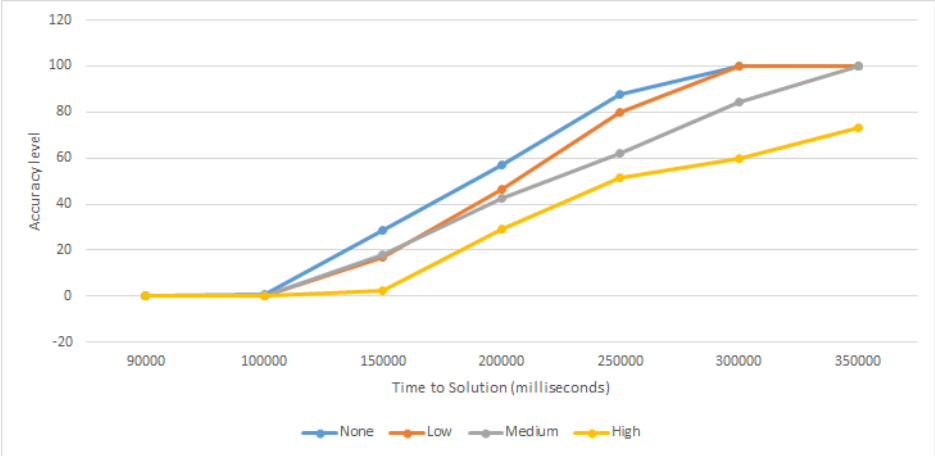

Figure 6.6: *Average accuracy trend by varying the time to solution.*

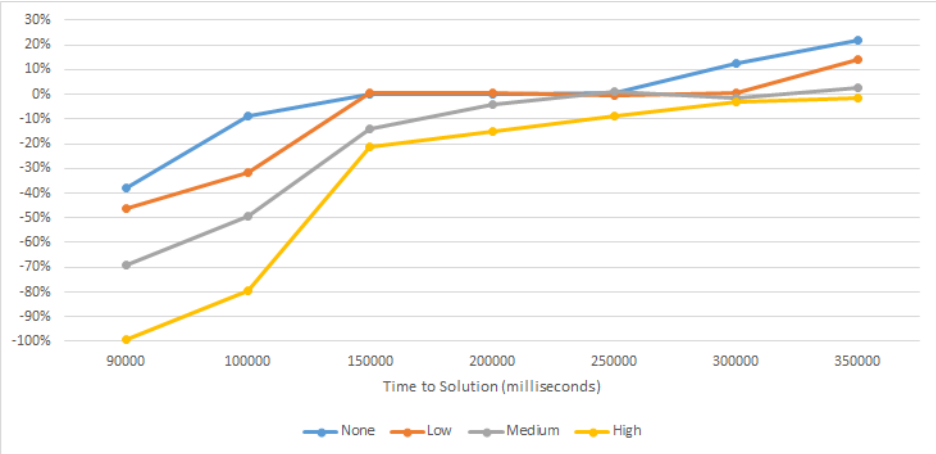

Figure 6.7: *Percentage of remaining time trend by varying the time to solution.*

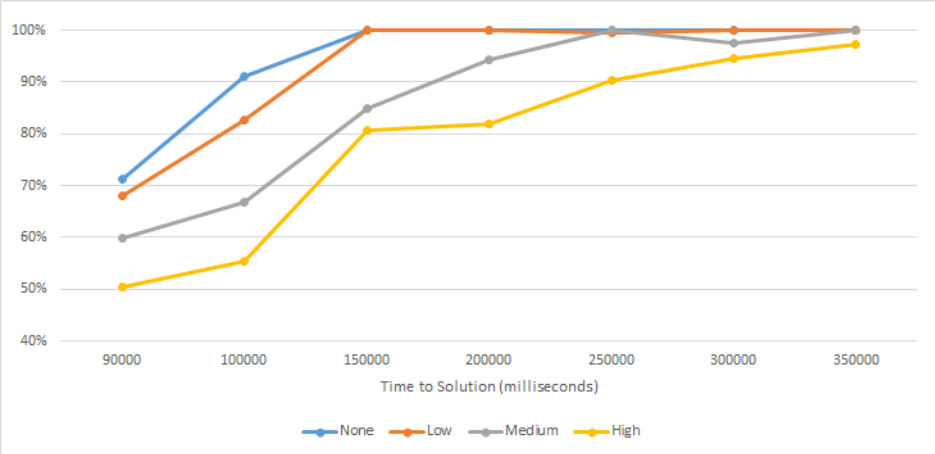

Figure 6.8: *Trend of the percentage of inputs successfully completed by varying the time to solution.*

Comparing Accuracy Level Trends of an Execution with the same Time-to-Solution Constraint

Charts in Figure 6.9, Figure 6.10, Figure 6.11 and Figure 6.12 show accuracy values with which inputs of a complete execution were computed, in the four experiments with different noise values. On the x-axis there is the number of the input computed with the accuracy value on the y-axis. The results represented have been collected from a complete execution with a time-to-solution equal to 250000 milliseconds. The accuracy level has been normalized between 0, which corresponds to the minimum level of accuracy, and 100, which corresponds to the maximum level of accuracy.

Figure 6.9 contains the accuracy values related to each input computed with no introduction of noise. The accuracy levels chosen by Ipazia are concentrated among the top, 100, and values just below 80. These values are just above and just below the mean accuracy value of 90, respectively. Ipazia has made minor adjustments to maintain the execution time within the given constraint, since there have been no big changes in the environment.

The same considerations apply to the experiment with low noise level introduction. In chart in Figure 6.10 the mean accuracy is less than the one in perfect conditions, but the accuracy selected by Ipazia framework is still just above and just below the respective mean accuracy value.

In experiments with medium (Figure 6.11) and high (Figure 6.12) values of noise the behavior is not the same of previous charts. The amplitude between the maximum and the minimum accuracy values selected is very wide. It goes from an accuracy value around 80 to a value below 40, in the experiment with medium noise, and from a value around 90 to a value just above 20, in the experiment with high noise. This happens because the framework handles a great change of the initial conditions, due to the introduction of high values of noise, and then the adjustments that Ipazia must make are wide and cover very different accuracy values from each other.

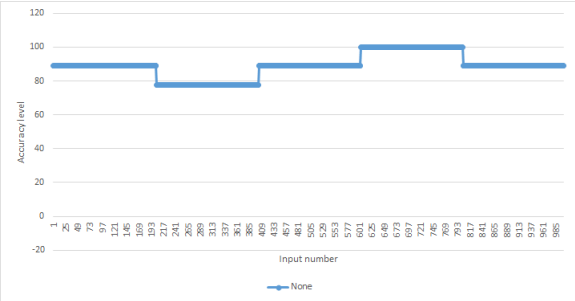

Figure 6.9: *No noise introduced: accuracy trend by varying the input.*

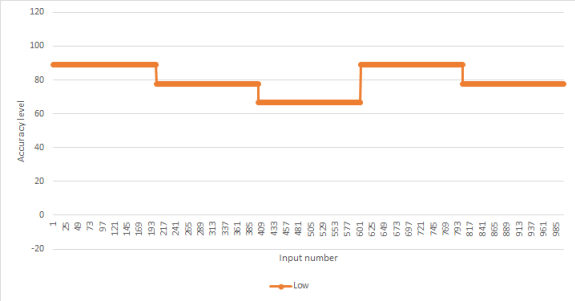

Figure 6.10: *Low noise introduced: accuracy trend by varying the input.*

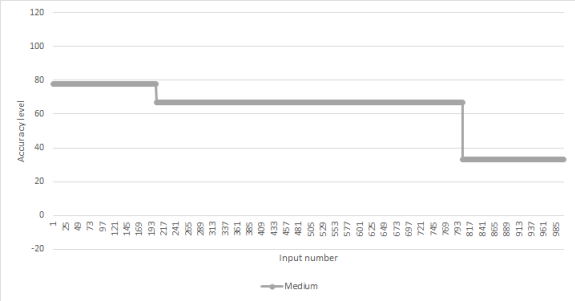

Figure 6.11: *Medium noise introduced: accuracy trend by varying the input.*

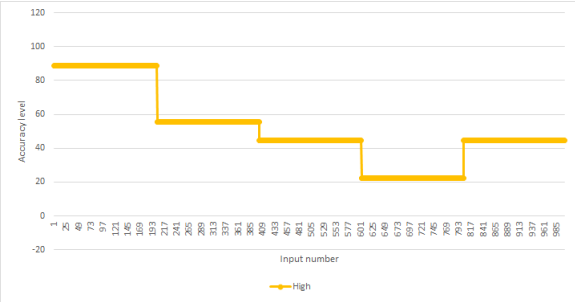

Figure 6.12: *High noise introduced: accuracy trend by varying the input.*

Overall Comment on the Experimental Results

Our experiments show that introducing noise has led to obtain an average accuracy less that the one of the experiments under perfect conditions, with the same Time to Solution. By comparing results under different noise values we can see that when no noise has been introduced, mean accuracy level chosen is concentrated in a small set of accuracy level (between 80 and 100), the same applies more or less in the case with low noise, while the accuracy level range in case of medium and high noise is very wide, even from 90 to 20. This is the response to the huge positive noise introduced. But pictures also show that our framework is flexible and even if we introduce some noise this is extinguished through synchronization points and Ipazia succeeded in maintaining a certain accuracy level as far as possible.

6.3.2 DTLZ 2 Function

We show also a set of experiments performed on a DTLZ function. DTLZ is a test suite of benchmark problems, created by Deb et al., well described in [22]. It is a collection of multi-objective test problems, which are scalable to any number of objectives, and so, it has facilitated several recent investigations into multi-objective problems. It is made of nine problems. In our experiments we used the DTLZ2 problem, described in Equation 6.16 and Equation 6.17. Its parameter domain is [0,1]. We implemented the version in [2].

$$f_1(\vec{x}) = (1 + g((\vec{x}) \cos(x_1 \frac{\pi}{2}) \tag{6.16}$$

$$f_2(\vec{x}) = (1 + g((\vec{x}) \sin(x_1 \frac{\pi}{2}) \tag{6.17}$$

Where:

$$g(\vec{x}) = \sum_{x_i \in \vec{x}} (x_i - 0.5)^2.$$

$$0 \leq x_i \leq 1, i = 1, ..., n.$$

A Pareto Front example with 3 functions is shown in Figure 6.13.

This is the formulation we used for our metrics:

$$Error = f_1(x, \vec{f}) = (1 + g((\vec{f}) \cos(x \frac{\pi}{2}) \tag{6.18}$$

$$Time = f_2(x, \vec{f}) = (1 + g((\vec{f}) \sin(x \frac{\pi}{2}) \tag{6.19}$$

Where:

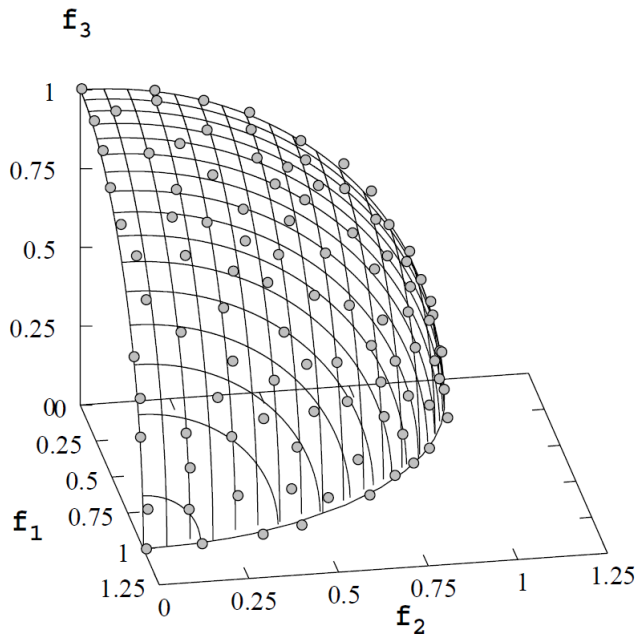

Figure 6.13: *Pareto front of DTLZ2 with 3 functions.*

$$g(\vec{f}) = \sum_{f_i \in \vec{f}} (f_i - 0.5)^2.$$

$$0 \leq f_i \leq 1, i = 1, 2.$$

We used one knob x and two features f_1, f_2 for each input. Knob x can assume one of the following values that we have taken from [2].

```
0.2973856025
0.4426370244
0.5602489444
0.6613433487
0.7500827675
0.8283237130
0.8967005327
0.9547573070
1.0000000000
```

We rejected the knob value that would set to zero the execution time for that input, because it would have altered our experiments.

Features instead can assume a random value in the range (0,1]. As we have done for the quadratic synthetic function, we made experiments by introducing

several quantities of noise. Synchronization points are set every 20% of the inputs. We performed experiments using 5000 inputs, so we recomputed the limit configuration every 1000 inputs done.

Gamma parameters used for different executions are the same used in Subsection 6.3.1.

To assess our framework, we have taken several values of time to solution and we have left to the framework the task of selecting the best limit configuration, according to the time-to-solution and the data features distribution. We have repeated this experiment for several noise values: no noise introduced, low, medium and very high noise introduced. Values collected are shown in Figure 6.14, Figure 6.15 and Figure 6.16. This time we collected the error values, which have been normalized between 0 (minimum error level, which corresponds to the maximum accuracy) and 100 (maximum error level, which corresponds to the minimum accuracy). The time-to-solution values are expressed in milliseconds.

The choice of the set of time-to-solution values to display has been done applying the same considerations made for the previous experiment. Therefore, we perform seven experiments for the different level of noise used.

The same considerations made for the previous experiment apply also to these charts.

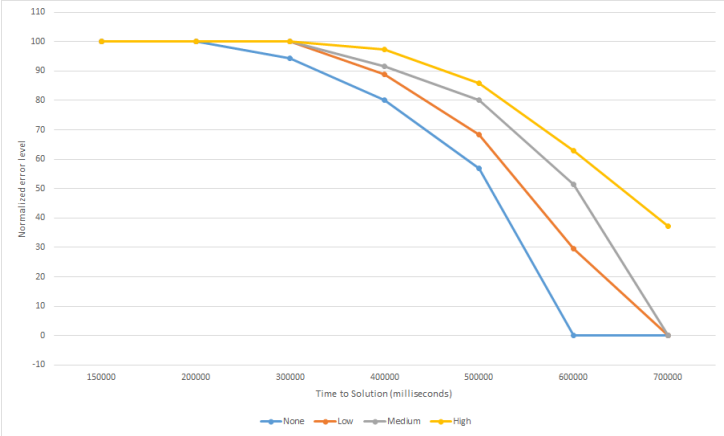

Figure 6.14: *Average error trend by varying the time to solution.*

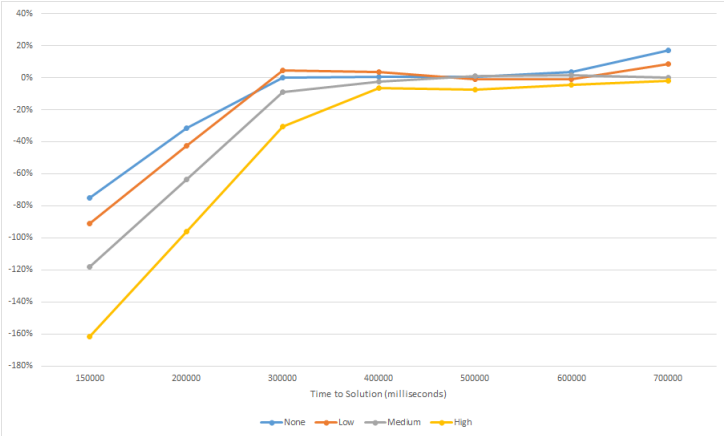

Figure 6.15: *Percentage of remaining time trend by varying the time to solution.*

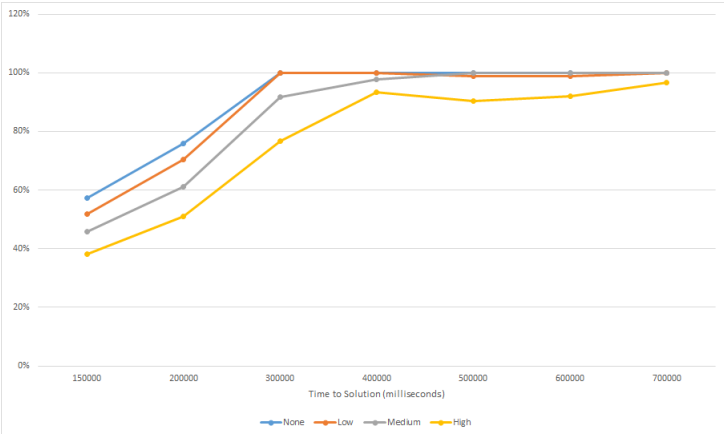

Figure 6.16: *Trend of the percentage of inputs successfully completed by varying the time to solution.*

6.4 Application Case Study

6.4.1 Drug Discovery - Miniapp

We carried out our experimental results also on a real application: a pharmaceutical-design application, namely LiGen, that has to compute the best matches between each couple of molecule and ligand in input [17]. The data in input are couples of molecule and ligand. Either of them are simply vectors of atoms.

The actual execution part is made of nested loops, where for each molecule orientation it is computed the match with each ligand orientation. The output is simply the maximum value of the match between that couple of molecule-ligand.

We have inserted some approximation knobs that can be tuned by our framework. In particular we have found three parameters that can be changed to monitor the approximation level, the execution time and the energy used:

1. *Precision*: this knob sets the data structure type to use in the application for managing all data/operations. It can take these values:
   - 2: knob set to value 2 means that computation is done using doubles;
   - 1: knob set to value 1 means that computation is done using floats;
   - 0: knob set to value 0 means that computation is done using integers.
2. *Alpha skip factor*: this knob controls the skip factor for the loop iterating on each atom in a molecule. It is the outer loop of the actually computation phase. Formally it can assume all values between 0 and 359 (since molecule rotation is perform in degrees), but the application is tiled with a tile equal to 10, and so this knob can only assume values between 0 and 9. For example, an alpha skip factor equal to 0 means that no cycle is skipped, while an alpha skip factor equal to 3 means that application will execute this loop once and then will skip 3 cycles.
3. *Beta skip factor*: the same definition of alpha skip factor applies also to beta skip factor, but this is applied to the inner loop, which iterates on the atoms of a ligand.

Therefore, each configuration in this application has the following structure: `<precision, alpha_skip_factor, beta_skip_factor>`.

As it regards to the metrics of interest, for this experiment we considered only the level of approximation and the execution time. In particular, the global constraint value is set on the time.

So each tuple of metrics of interest has the following structure: `<accuracy, time>`.

These two metrics have a trade off, since increasing the level of accuracy means to spend more execution time.

Applications are compiled with mpirun command, using the optimization flags -O2, -Wall (warn all), -Wextra, -g2, -pg and -std=c++11. An example of launch command is the following:

```
>mpirun -np <num-processes> ./<application_executable>
```

In our experiments we considered only molecules that have a number of atoms between 5 and 200, while ligands considered have a number of atoms between 5 and 30.

6.5 Application Case Study: Experimental Evaluation

At first, we measure execution time consumed for the maximum accuracy level. Reported results are related to the experiments that produced the mean time over three runs. These are direct program executions and measurements, not simulations or emulations. Time measured is the total time of the application, until all processes have completed.

6.5.1 Overheads Characterization

In this subsection we provide information regards both communication overheads characterization and overheads introduced by Ipazia framework. We collected this information only on the real application study, because only here we can draw significant results, and we examine the whole set of spreading autotuning knowledge, well described in Table 5.1: fully distributed, partially distributed, partially centralized and fully centralized.

Table 6.1 shows the trend of communication overheads alone, that is, time spent to send Ipazia information through master and workers. The execution times shown involve only one sending operation. In the fully distributed A version and in the fully centralized methods, the framework has to send only a configuration, that is a list of three float values regarding this experiment. In the fully distributed method B version, instead, the framework has to send the quantity of time metric for a given input, that is a double value. In partially distributed and partially centralized Ipazia has to send an operating points list.

We can notice that these communication overheads are in the order of microseconds. Even the sending of all operating points list, with length equal to 108 in this experiment, and with every operating point having 3 knobs and 2 metrics, lies in the order of microseconds. Obviously, overhead of the partially centralized method depends on the number of operating points to be sent, but the

upper limit is represented by the overhead of the partially distributed method, which does not exceeded the time value of one millisecond. Therefore, in case the execution time of a single input lies in the order of milliseconds, we may not taken into account these communication overheads, or we can also formalize those with a constant value. In the fully distributed A and B cases, the communication overheads are very very small, lying in the order of few microseconds.

Table 6.1: *Summary of average communication overheads of the different spreading autotuning knowledge methods. Execution time refers to one sending operation.*

Methods	Time (microseconds)
FULLY DISTRIBUTED A	3,36
FULLY DISTRIBUTED B	13,64
PARTIALLY DISTRIBUTED	576,94
PARTIALLY CENTRALIZED	502,87
FULLY CENTRALIZED	3,37

Table 6.2: *Summary of average overheads introduced by our framework.*

Call	Time (milliseconds)
init	561
init limit configuration	18166
autotuning knowledge computation	2204

Table 6.2 shows overheads introduced by our framework, measured in milliseconds and mediated on several executions and on the different search mechanisms and on the different environment used (Table 5.1). The first line refers to the overhead introduced by the initialization of the framework. In this method, Ipazia sets up its environment, by reading the configuration files, and builds the predictive models from the given training set file. The overhead is constant and does not exceeded the second.

The second line refers to the overhead measured in the phase of calculating the optimal limit configuration for a given workload. It has been averaged on several executions with the same workload but different search strategies used. This overhead is much higher than the previous one, since the operations that the framework must done in this phase are many more. Therefore, we would expect a much lower value than the one measured. This large latency is primarily due

to interfacing with the model, which consists of making numerous accesses to the R software, regarding this experiment. Using a different model to interface with can be a possible solution to decrease this overhead. Details regarding the overhead introduced by each search strategy evaluated in this experiment are shown in Table 6.3.

The last line shows the average overhead measured on the computation of the autotuning knowledge in the different methods that we proposed, master side. Table 6.4 goes in the detail of the overheads measured in each method.

We can see that overheads are in the order of milliseconds, therefore they are not so negligible, but there are some specifications to introduce: this framework is meant to be used in application computing a so huge quantity of inputs using several parallel processors. Overheads shown in Table 6.2 are not all present in the whole application life. For example, the time spent for the update of the global limit configuration is lost only during the first computation, since the following times it is performed in a separate thread. Regarding the ipazia init, it obviously is called only one time per processor in the application, and the delay is quite constant and independent from the length of configuration files it has to read, since it remains in terms of milliseconds.

In Table 6.3 there are information regarding the execution time spent, in milliseconds, in the two different search methods that we used in this experiment for evaluating the methodology: the exhaustive and the precimonious-like strategies.

Exhaustive search strategy performs a full search over the entire space of configurations and for each of them uses the predictive models to determine that configuration which acts as a watershed between configurations that are good (satisfy the constraint) and the ones that are not good (do not satisfy it) (Subsection 4.3.3), using the formula in Equation 4.3.

Precimonious-like strategy, instead, first creates an operating points list with the time metric values filled with average values of the input features collected in the Montecarlo phase, then sorts the list from the configuration with the higher predicted time value to the one with the lower predicted time value. Then, it applies the formula in Equation 4.3 using this sorted list, and ends the search as soon as it finds a feasible configuration, that is a configuration whose time metric satisfy the global constraint. In this way, not all configurations and input features space must be examined. This is an heuristic and we decided to evaluate this strategy since in HPC applications it may not to be possible to apply an exhaustive search method when the knobs and the features space to explore is too wide.

We can notice that the exhaustive search method has an execution time bigger of an order of magnitude with respect to the execution time of the precimonious-

like search method. Time spent by precimonious-like method is not totally linear in the number of samplings since its duration depends also on the number of configurations that it explored. We can say that the higher the constraint it is, the less configuration it has to explore, since probably the best configuration will lie in the first positions of the sorted operating points list. Both the overheads are higher than the expected values, and this is still primarily due to interfacing with the model, which consists of making numerous accesses to the R software, regarding this experiment. Using a different model to interface with can be a possible solution to decrease this overhead.

Table 6.3: *Summary of average overheads introduced by the different search method used, with different number of samplings.*

Search method	Number of samplings	Time (milliseconds)
EXHAUSTIVE	6	29595
	5	27489
	4	19509
	3	9844
PRECIMONIOUS	6	7093
	5	6027
	4	2381
	3	1835

We have also collected overheads regarding the time spent for generate the knowledge to then give to the autotuner. We have repeated the execution using the different spreading autotuning knowledge methods (Table 5.1) and we have collected time values on the master side, but they can also be applied on the worker side, since steps to perform on master and on worker are complementary. Results are shown in Table 6.4.

Table 6.4: *Summary of autotuning knowledge computation times w.r.t the spreading knowledge method selected.*

Spreading method	Steps	Time (milliseconds)
FULLY DISTRIBUTED B	metric value calculation	11
PARTIALLY DISTRIBUTED	metric value calculation op list population	2893
PARTIALLY CENTRALIZED	metric value calculation op list population op list filtering	3177
FULLY CENTRALIZED	metric value calculation op list population solve	3045

The fully distributed method version A is not present since the master has to compute nothing.

Using the fully distributed method version B, the master process has also to access to the model and retrieve the value of the metric for the input given, therefore its relative latency is very small, only 11 milliseconds.

In the partially distributed method, master has to do the same as in the previous method, but it must also compute the operating points list of the input given, and so the overhead grows, reaching few seconds. This is always due to the accesses to the model through the R software.

The same considerations apply to the partially centralized strategy, in addition master filters the operating points list, before sending it. The overhead value is similar to the previous one. In the last method considered, instead, the master has to perform also the autotuning phase, thus the relative is greater than the one of the partially distributed method, but only slightly.

However, these overheads weigh on the application lifetime only one time, since the master process does nothing except for the dispatching work. We can considered them negligible if the amount of data in input is large enough.

6.6 Application Case Study: Profiling Phase

In this phase we have collected information related to the metrics of interest (accuracy levels and execution times) set by different combinations of configurations and input features.

We first have executed the parallel application with different numbers of data in input and we have gathered execution times information related to both total and per input time spent, under the configuration with maximum accuracy, which in this application is the `<2, 0, 0>` configuration, so with no loops skipped and a double precision. These experiments are needed in order to understand what is the maximum execution time the application will spend on these inputs and then it is used to set a valid metric-to-solution constraint on the execution time to test if our methodology actually satisfies the global constraint selecting a configuration with less accuracy than the perfect one, and actually produces a significant saving in time.

For gathering accuracy levels information we first executed each input with a configuration and then re-executed the same input under the max-accuracy configuration, i.e. the tuple `<2, 0, 0>`. The accuracy level is simply calculated as the inverse of the error. If we call $error$ the error measured, the accuracy $accuracy$ is defined in this way:

$$accuracy = \frac{1}{error} \tag{6.20}$$

This is a normalized accuracy, since the error is calculated as follow:

$$error = \left| \frac{perfectResult - inputResult}{perfectResult} \right| \tag{6.21}$$

Where:

$perfectResult$: Result computed with the max accuracy configuration (which is `<2, 0, 0>` in this experiment).

$inputResult$: Result computed using the configuration found by mARGOt autotuner.

After having collected all these data we first verified if different values of knobs and input features actually influence the application execution time and

accuracy levels. Results from the regression modeling phase show that our hypotheses were correct. The adjusted R-squared values are greater than 0,6:

- Adjusted R-squared value for the time model: 0,89.
- Adjusted R-squared value for the accuracy model: 0,95.

In the time model, all variables have a strong influence on the resulted time. Complete regression model has a much greater accuracy that the simple one. We can so say that probably a better model could be a nonlinear one, since there are relationships the linear model does not fully grasp. But we can also point out that the adjusted R squared is satisfying in both the complete and the simple regression model for execution time prediction. For the accuracy level model we applied the same considerations. There have been surely nonlinear relationships our regression modeling does not grasp at all, but the adjusted R squared is satisfying also this time. We have also perform the step phase in order to eliminate variables that do not give a huge contribution to the predictions.

However, since our framework does not have to compute in an automatic way the perfect predictive model for all possible numbers, values and types of knobs and metrics of interest, we performed all experimental results using complete regression models built automatically in the data collection phase of our framework. We can also stated that our methodology proposes estimations of both workload in input and consumption analysis, and since we recomputed the limit configuration many times and the application has local problems to solve, we do not really need the absolute perfect predictive models and the perfect results from them. We can accept a quite good predictive model that more or less allows us to discriminate between configurations good and not good [Subsection 4.3.3].

Figure 6.17 gathers differences between predicted and real execution times collected on a complete execution of the Miniapp. The chart shows the execution time in milliseconds by varying the input (and so, with different input features, the configuration selected can be different). Predictive models used are good for our experiment since the prediction curve follows more or less the trend of the one built with the actual values.

We performed the computing modeling phase using a training set made of measurements collected performing exhaustively running on the whole set of configurations, but not on the whole set of possible input features values, since they are too much.

Figure 6.18 and Figure 6.19 show the distribution of the data input features used in our experiments. On the y axis there is the number of molecules/ligands that have a number of atoms in the range indicating by the respective bean.

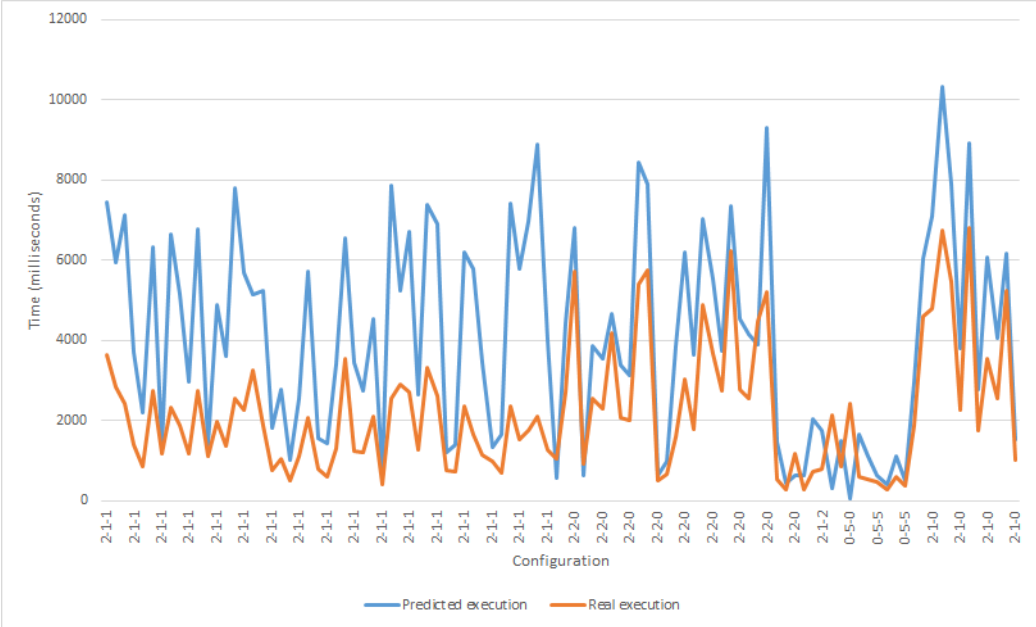

Figure 6.17: *Predicted versus real execution time trend by varying the configuration.*

For example, in the distribution considered there are 6 molecules with a number of atoms in the range [14,24] and there are 15 ligands with a number of atoms in the range [25,30]. We collected results on 100 couples of molecule-ligand in input. The distribution of the atoms in molecules is wide, since the number of atoms in molecule can range from 5 and 200, while ligands considered can have a number of atoms between 5 and 30.

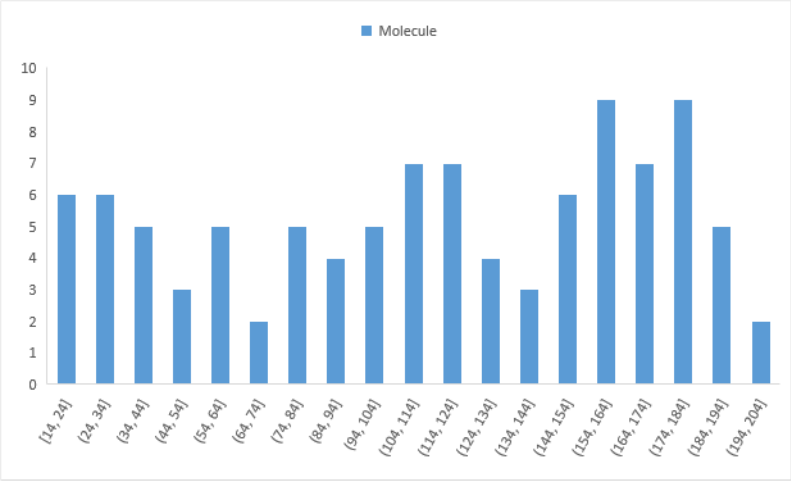

Figure 6.18: *Distribution of number of atoms of molecules in the data in input, for this experiment.*

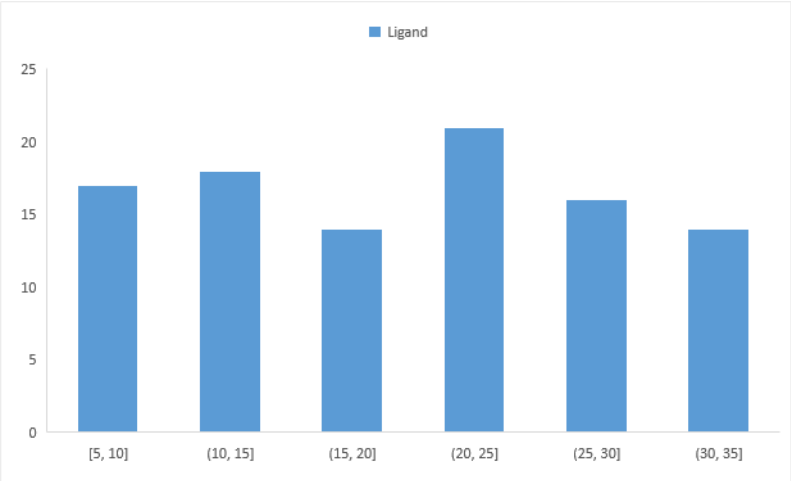

Figure 6.19: *Distribution of number of atoms of ligands in the data in input, for this experiment.*

6.7 Application Case Study: Configuration Selection

In this section we show results regarding the difference in configuration selection of the two search methods evaluated. Table 6.5 show several limit configuration computation results collected using exhaustive and precimonoius-like methods, with different constraint values related to the computation of a single input.

Table 6.5: *Summary of configuration selected by the two search methods used. Constraint value and time used are in milliseconds.*

Search method	Time constraint on a single input (milliseconds)	Configuration selected	Time used for configuration selected (milliseconds)
EXHAUSTIVE	6000	<2,1,0>	5790
	5000	<2,2,0>	4875
	4000	<1,2,1>	3978
	3000	<1,1,2>	2976
	2000	<2,4,0>	1985
	1000	<1,1,3>	974
PRECIMONIOUS	6000	<2,1,0>	5790
	5000	<2,2,0>	4875
	4000	<1,2,1>	3978
	3000	<1,1,2>	2976
	2000	<2,4,0>	1985
	1000	<1,1,3>	974

The configuration selected from the exhaustive search strategy for the different constraint values is always the same selected from the precimonious-like search, and it is always the optimal one. In the last column there is the time value predicted for the computation of a single average input of the workload, whose features are the average values computed from the feature distributions built in the Montecarlo sampling phase. We can see that, for example, with a time constraint equal to 6000 milliseconds and average input features, it has been selected a configuration which allows the execution of the average input to be less than the constraint, and in particular, to be equal to the predicted time of 5790 milliseconds. Precimonious-like method is not exhaustive, but our results show that

in this experiment it has always found the optimal solution, using less time than the exhaustive search method.

6.7.1 Global Constraint Satisfaction

As in tests performed using synthetic functions, we have collected results on several executions providing significant time-to-solution values: starting from a time-to-solution equal to 850000 milliseconds, where the application can be successfully executed at maximum accuracy level, to a time-to-solution not large enough to allow the complete execution within that constraint at the minimum accuracy level (230000 milliseconds). Values collected are shown in Figure 6.20, Figure 6.21 and Figure 6.22. The accuracy level has been normalized between 0, which corresponds to the minimum level of accuracy, and 100, which corresponds to the maximum level of accuracy. The time-to-solution values are expressed in milliseconds. Synchronization points are set every 20% of the inputs. We performed experiments using 100 inputs, so we recomputed the limit configuration every 20 inputs done.

Figure 6.20 shows the normalized accuracy values by varying the time-to-solution given to the application.

Figure 6.21 shows the percentage of remaining time by varying the time-to-solution given to the application.

Finally, Figure 6.22 shows the percentage of inputs successfully completed by the application within the given time-to-solution, by varying the time-to-solution constraint itself. This percentage is less than 100% in correspondence with a negative percentage of remaining time.

The overall shapes follow the trend found in experiments with quadratic and DTLZ 2 functions. Therefore, the same considerations regarding the experiment with the quadratic function in perfect and with low noise conditions can be applied also to these results. The average accuracy level decreases with the decreasing of the time-to-solution given. The percentage of remaining time is greater than zero in all the runs except for the last one, where it is not possible to end the application execution within the given constraint (230000 milliseconds) since the relative accuracy level is already at the minimum level. Thanks to our framework we are able to stay in the global constraint given and also in some cases we are able to obtain a substantial gain in time-to-solution with respect to the error introduced. This quantity of time saved can be thus used for example to perform some runs on some inputs that gave very inaccurate results or, on the contrary, in the case of this specific application, to find more precise matches between couples of molecule-ligand that at a first run have obtained a good matching value.

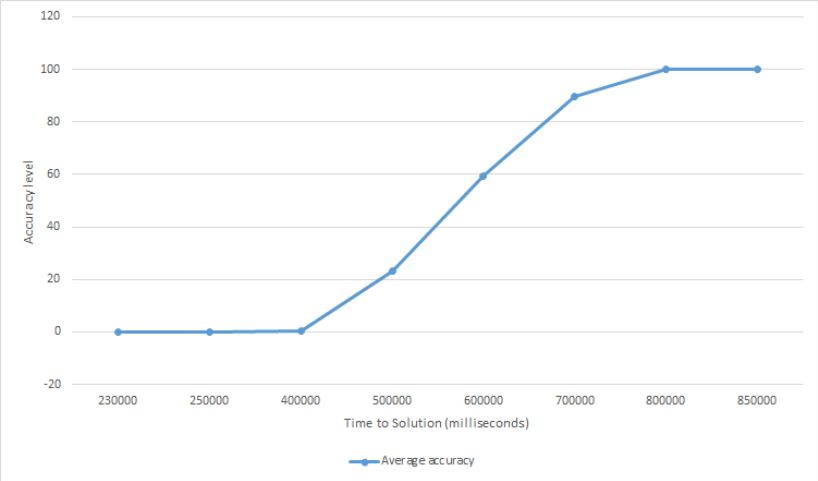

Figure 6.20: *Average accuracy trend by varying the time to solution.*

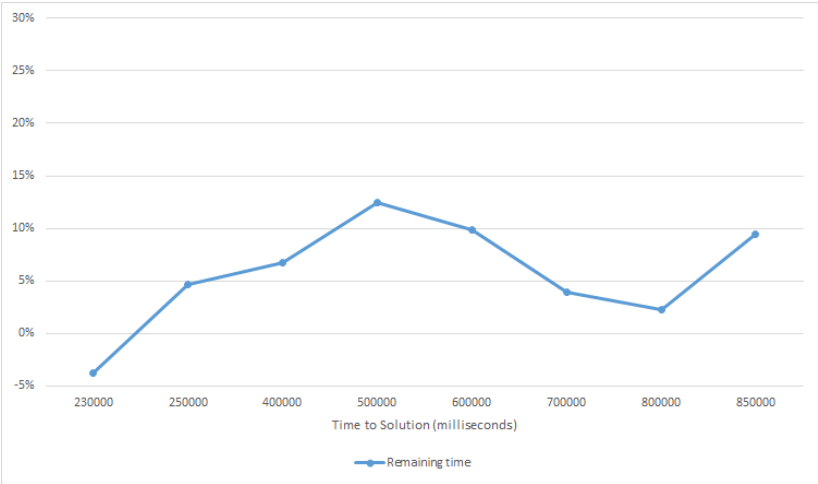

Figure 6.21: *Percentage of remaining time trend by varying the time to solution.*

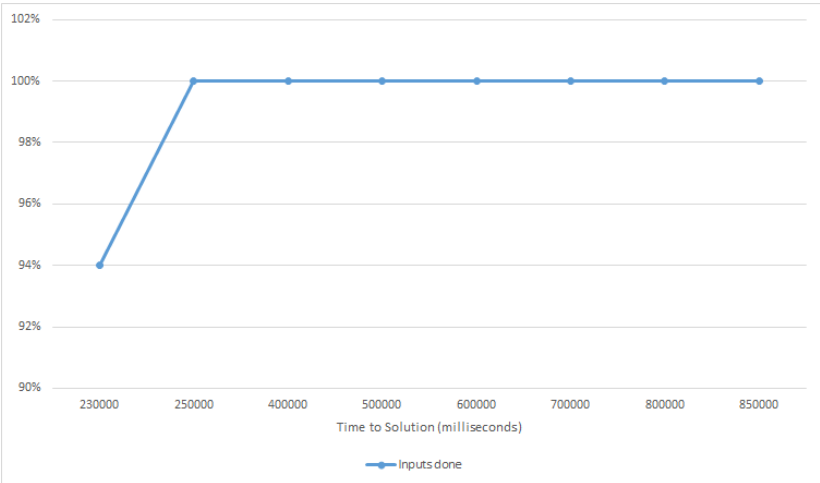

Figure 6.22: *Trend of the percentage of inputs successfully completed by varying the time to solution.*

CHAPTER 7

Conclusions and Future Works

7.1 Summary, Benefits and Limitations

In this thesis, we have faced some important issues related to the High Performance Computing field. We address the problem of both performance modeling and performance automatic management. We focus on the HPC context where environments are parallel and distributed, with no shared memory. The specific problem we dealt with concerns the satisfaction of a global constraint (mostly regarding performance) in HPC applications with input variability. This global constraint is given in terms of a particular metric-to-solution that the application must meet. In HPC, popular metric-to-solutions are the Time-to-Solution or Energy-to-Solution, since these are the main optimization areas. The difficulty in solving this problem arises from the fact that many HPC applications have unpredictable behavior as it strongly depends not only on the load of the input data but also on the characteristics of the inputs themselves. We also give a contribution in the investigation of the use of an autotuner in a proactive way, for choosing how the application must be executed based essentially only on particular data input features to compute.

Our solution consists of providing a framework to enable job scheduling in a distributed environment with no shared memory and to configure processes for respecting metrics-to-solution constraints, in particular time-to-solution con-

straints, which are the most required in HPC field. The framework developed has a modeling module, which deals with the automatic computation of predictive models for the metrics of interest. They are used both in the computation of the limit configuration and in the calculation of the list of operating points to pass to the autotuner locally. The methodology proposes a method for estimating the distribution of the data in input without having to read them all. It is based on the Montecarlo sampling technique.

The first advantage of our solution consists surely of the easiness of integration in the application. Ipazia framework must be configured using configuration XML files, thus it is completely independent of the application internal structure. Our framework has been developed to be interfaced with different autotuner frameworks, since the mARGOt that we used is not integrated in our framework. We also proposed different methods for spreading the autotuning knowledge in worker processes, that can be applied depending on the physical structures of the machines used.

Another advantage is the introduction of the periodic update limit configuration mechanism, that allows the application to be successfully tolerant of low noise levels, as shown in Chapter 6.

Our experiments show that the approach proposed in this thesis has proved to be effective in the applications considered and it can actually give a real contribution to the HPC problems described in Chapter 2 and in Section 4.2. The majority of experiments we have done show that Ipazia framework succeeded in guaranteeing the execution of all inputs within the time to solution constraint, in case the noise introduced is restrained.

Overheads values collected show that a distributed solution using MPI does not introduced excessive communication overheads and it is quite faster for performing these operations, but it can be improved as well. Overheads values introduced by the Ipazia framework, instead, are not negligible. The latency resulting from access to R has a big impact on the time to search the limit configuration. Sure, a possible limitation of this approach can be identified in the fact that our framework can choose a configuration that is not the best one, since through the Montecarlo sampling method it builds an estimated distribution of the data in input. However, this limitation can be absorbed since, in the context we have addressed, obtaining optimal values in the search of the global limit constraint to spread to workers is unfeasible due to the huge amount of data in inputs.

7.2 Future Works

The methodology and the framework developed can be further improved and expanded, by introducing several aspects:

- Introducing a better automatic building framework of predictive models. This can be an improved bound to the automatic model creation alone. We can think of integrating an external framework that computes in an automatic way, different types of predictive models and chooses in autonomy the best one to provide to our framework. Types used must take into account also machine learning mechanisms, in order to create very precise models when relationships are not so evident.

- Online learning: It is in some way an expansion of the automatic building models phase, but this also must wrap the profiling phase part of the application. With this improvement our framework can be able to generate predictive models in a complete automatic way, without that the user must provide any information regarding the behavior of metrics of interest with respect to knobs and input features.

- More heuristics: In case the design space of knobs, features and metrics is very large, other heuristics can be used in the limit configuration selection part. Precimonious-like heuristic that we have used is so fast, but in the worst case, that is when the satisfying configuration is situated in the final part of the configuration list, its performing time reaches the exhaustive performing time. Therefore, another improvement is to take into account heuristics that also in the worst case do not explore all the decision tree.

- Apply other topologies for MPI communication. We can extend our research by further studying alternative communication topologies for disseminating information through processes.

Bibliography

[1] ANTAREX tools. http://www.antarex-project.eu/dissemination#tools. Accessed: 2017-03-22.

[2] DTLZ 2. http://people.ee.ethz.ch/~sop/download/supplementary/testproblems/dtlz2/index.php. Accessed: 2017-03-22.

[3] High-Performance Computing (HPC). http://searchenterpriselinux.techtarget.com/definition/high-performance-computing. Accessed: 2017-02-18.

[4] mARGOt gitlab. https://gitlab.com/margot_project/core. Accessed: 2017-03-22.

[5] MPICH. https://www.mpich.org/. Accessed: 2017-02-25.

[6] mpiexec vs mpirun. https://www.osc.edu/~djohnson/mpiexec/. Accessed: 2017-03-19.

[7] Openmp. http://www.openmp.org/. Accessed: 2017-03-28.

[8] pugixml. http://pugixml.org/. Accessed: 2017-02-26.

[9] RInside package. https://CRAN.R-project.org/package=RInside. Accessed: 2017-02-25.

[10] RInside use. http://dirk.eddelbuettel.com/code/rinside.html. Accessed: 2017-02-25.

[11] Simple Linear Regression. https://www.r-bloggers.com/simple-linear-regression-2/. Accessed: 2017-02-18.

[12] The GREEN 500. https://www.top500.org/green500/. Accessed: 2017-04-1.

[13] The R Project for Statistical Computing. https://www.r-project.org/. Accessed: 2017-02-18.

[14] What is high performance computing? http://insidehpc.com/hpc-basic-training/what-is-hpc/. Accessed: 2017-02-18.

[15] *MPI: A Message-Passing Interface Standard*. 2012. Message Passing Interface Forum.

[16] Woongki Baek, Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *PLDI '10 Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[17] Andrea R. Beccari, Carlo Cavazzoni, Claudia Beato, Gabriele Costantino. Ligen: a high performance workflow for chemistry driven de novo design. *J. Chem. Inf. Model*, pages 1518–1527, 2013.

[18] Babak Behzad, Surendra Byna, Stefan M. Wild, Prabhat, Marc Snir. Dynamic model-driven parallel i/o performance tuning. *IEEE International Conference on Cluster Computing (CLUSTER)*, 2015.

[19] Babak Behzad, Surendra Byna, Stefan M. Wild, Mr. Prabhat, Marc Snir. Improving parallel i/o autotuning with performance modeling. *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 253–256, 2014.

[20] Janez Brest, Viljem Žumer. A simple method for dynamic scheduling in a heterogeneous computing system. *Journal of Computing and Information Technology*, 2002.

[21] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, 2013.

[22] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, Eckart Zitzle. Scalable test problems for evolutionary multi-objective optimization. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2001.

[23] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Publisher, 2011.

[24] V.W. Freeh, Feng Pan, N. Kappiah, D.K. Lowenthal, R. Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. *Parallel and Distributed Processing Symposium*, 2005.

[25] Davide Gadioli, Gianluca Palermo, Cristina Silvano. Application autotuning to support runtime adaptivity in multicore architectures. *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015.

[26] Davide Gadioli, Gianluca Palermo, Cristina Silvano. Auto-tuning support for manycore applications - perspectives for operating systems and compilers. *ACM SIGOPS Operating Systems Review archive*, 43:96–97, April 2009.

[27] Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, Cristina Silvano. Opencl application auto-tuning and run-time resource management for multi-core platforms. *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2014.

[28] Cindy Rubio Gonzalez, Cuong Nguyen, Hong-Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, David Hough. Precimonious: Tuning assistant for floating-point precision. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[29] William Gropp, Rajeev Thakur. Issues in developing a thread-safe mpi implementation. *EuroPVM/MPI'06 Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, pages 12–21, 2006.

[30] William Gropp, Rajeev Thakur. Thread safety in an mpi implementation: Requirements and analysis. *Parallel Computing archive*, 33:595–604, 2007.

[31] Georg Hager, Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011.

[32] Jie Han, Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. *18th IEEE European Test Symposium*, 2013.

[33] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, Martin Rinard. Dynamic knobs for responsive power-aware computing. *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 199–212, 2011.

[34] Abeer Hyari. A comparative study on heterogeneous and homogeneous multiprocessors. 2009. University of Jordan, Report Research.

[35] Gabriele Jost, Hao-Qiang Jin, Dieter an Mey, Ferhat F. Hatay. Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster. 2003. Computer Sciences Corp., Preprint.

[36] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 151–162, 2014.

[37] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, Anurag Mendhekar. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[38] Benjamin C. Lee, David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGARCH Computer Architecture News - Proceedings of the 2006 ASPLOS Conference*, 2006.

[39] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, Martin Rinard. Quality of service profiling. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE*, 2010.

[40] Hierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Luis Ceze, Natalie Enright Jerger, Adrian Sampson. A taxonomy of approximate computing techniques. Technical Report.

[41] Edoardo Paone, Francesco Robino, Gianluca Palermo, Vittorio Zaccaria, Ingo Sander, Cristina Silvano. Customization of opencl applications for efficient task mapping under heterogeneous platform constraints. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[42] Edoardo Paone, Davide Gadioli, Gianluca Palermo, Vittorio Zaccaria, Cristina Silvano. Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive opencl applications. *ASAP*, 2014.

[43] Vito Ricci. *Principali tecniche di regressione con R*, 2006.

[44] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, Martin Schulz. Bounding energy consumption in large-scale mpi programs. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[45] C. Silvano, W. Fornaciari, S. Crespi Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, A. Di Biagio, E. Speziale, M. Tartara, D. Melpignano, J.-M. Zins, D. Siorpaes, H. HÃ$\frac{1}{4}$bert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, S. Xydis, D. Soudris, T. Kempf, G. Ascheid, R. Leupers, H. Meyr, J. Ansari, P. Mähönen, B. Vanthournout. 2parma: Parallel paradigms and run-time management techniques for many-core architectures. *IEEE Annual Symposium on VLSI*, pages 65–79, 2010.

[46] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Joao Bispo, Radim Cmar, João M. P. Cardoso, Carlo Cavazzoni, Jan Martinovič, Gianluca Palermo, Martin Palkovič, Pedro Pinto, Erven Rohou, Nico Sanna, Kateřina Slaninová. Autotuning and adaptivity approach for energy efficient exascale hpc systems: the antarex approach. *DATE2016_ANTAREX*, 2016.

[47] Cristina Silvano, Giovanni Agosta, Gianluca Palermo. Efficient architecture/compiler co-exploration using analytical models. *Design Automation for Embedded Systems*, 11:1–23, 2007.

[48] Robert Springer, David K. Lowenthal, Barry Rountree, Vincent W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 230–238, 2006.

[49] Bjarne Stroustrup. *The C++ programming language*. PEARSON, 4th edition, 2014.

[50] Xin Sui, Andrew Lenharth, Donald S. Fussell, Keshav Pingali. Proactive control of approximate programs. *ASPLOS '16 Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[51] W. N. Venables, D. M. Smith, R Core Team. *An Introduction to R*, 2016. Available online at https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

[52] Mark Wyse. Modeling approximate computing techniques. *Academic paper*.