Universitat Autònoma
de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de

Computadors i Sistemes Operatius

Including the Workload Effect in the Parallel Program Signature

Thesis submitted by **Javier Martinez Canillas** for the degree of Master in High Performance Computing by the Universitat Autònoma de Barcelona, under the supervision of Dr. Emilio Luque

Barcelona, June 2011

Including the Workload Effect in the Parallel Program Signature

Thesis submitted by Javier Martinez Canillas for the degree of Master in High Performance Computing by the Universitat Autònoma de Barcelona, under the supervision of Dr. Emilio Luque, at the Computer Architecture and Operating Systems Department.

Barcelona, June 2011

Supervisor

Dr. Emilio Luque

This work is dedicated to my parents
brothers and to the love of my life. I hope I
can give you back in some way all the love
received from all of you.

Abstract

Performance prediction and application behavior modeling have been the subject of extensive research that aim to estimate applications performance with an acceptable precision. A novel approach to predict the performance of parallel applications is based in the concept of Parallel Application Signatures [36] that consists in extract an application most relevant parts (phases) and the number of times they repeat (weights). Executing these phases in a target machine and multiplying its exeuction time by its weight an estimation of the application total execution time can be made.

One of the problems is that the performance of an application depends on the program workload. Every type of workload affects differently how an application performs in a given system and so affects the signature execution time.

Since the workloads used in most scientific parallel applications have dimensions and data ranges well known and the behavior of these applications are mostly deterministic, a model of how the programs workload affect its performance can be obtained. We create a new methodology to model how a program's workload affect the parallel application signature.

Using regression analysis we are able to generalize each phase time execution and weight function to predict an application performance in a target system for any type of workload within predefined range. We validate our methodology using a synthetic program, benchmarks applications and well known real scientific applications.

Resumen

La predicción del rendimiento y el modelado del comportamiento de las aplicaciones son tópicos ampliamente estudiados y se cuentan con numerosos trabajos de investigación que pretenden estimar el rendimiento de la aplicaciones con una precisión aceptable. Un nuevo enfoque para predecir enfoque para predecir el rendimiento de aplicaciones paralelas es el basado en el concepto de las firmas de aplicaciones paralelas [36] que consiste en extraer las partes mas relevantes de una aplicación (fases) y el numero de veces que se repiten (pesos). Ejecutando estas fases en una maquina destino y multiplicando su tiempo de ejecución por su peso, se puede obtener una estimación del tiempo total de ejecución de la aplicación.

Uno de los problemas es que el rendimiento de una aplicación depende de la carga de trabajo de esta. Cada tipo de carga de trabajo afecta de manera distinta el rendimiento que tiene una aplicación en un sistema determinado y por lo tanto el tiempo de ejecución de la firma.

Dado que las cargas de trabajo de la mayoría de las aplicaciones científicas paralelas, tienen dimensiones y rango de datos bien conocidos y que el comportamiento de estas aplicaciones es generalmente determinista, se puede obtener un modelo de como la carga de trabajo de un programa afecta su rendimiento. Hemos creado una nueva metodología para modelar como la carga de trabajo de un programa afecta a la firma de la aplicación paralela.

Usando análisis de regresión, hemos podido generalizar las funciones de tiempo de ejecución y peso para cada fase para predecir el rendimiento de una aplicación en un sistema destino para cualquier tipo de carga de trabajo dentro de un rango predefinido. Hemos validado nuestra metodología utilizando un programa sintético, aplicaciones de *benchmarks* y aplicaciones reales científicas bien conocidas.

Acknowledgements

First of all I want to thank to mi advisor Doctor Emilio Luque Fadón for his invaluable guide, patience and dedication. Without his constant support and orientation, this research would not be possible.

To Doctor Dolores Rexachs del Rosario for her supervision, recommendations and suggestion. Se tried very hard to help me take the best of my.

Also I want to thank to the supporters of this research work the MEC-MICINN Spain under contract TIN2007-64974, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120.

I want to thank my fellows at the CAOS department for their patience and support, especially to my research group.

Many thanks to my co-workers that behave as good friends in many moments during this work, thank you Carlos Nuñez, Hugo Meyer, Ronal Muresano, Marcela Castro, Carlos Brum and Abel Castellanos.

I want to give special thanks to Doctor Alvaro Wong. He not only probe to be a good friend and co-worker but also helped me greatly during all the master. His experience, advice and recommendations were crucial to finish this project.

To Doctor Benjamín Barán who was the one that introduced me to the fascinating world of research. He has been my mentor and an example to follow all over these years, showing me that research can also be made in Paraguay with sweat and hard work.

A special mention to my profesors, classmates and friends of the "Facultad de Ciencias y Tecnología de la Univesidad Católica", with whom I had shared my years of formation as an Engineer, especially to Roberto Rodríguez Alcalá, Raúl Gutiérrez, Miguel Prieto, Juan Mignaco, Rodrigo Villalba y Víctor González.

Finally I want to thank my wife Tami, for all her patience and for give me strengths when I needed. She was the one that make this work possible.

Contents

Chapter 1

Introduction

1.1 Motivation

Parallel application performance prediction is one of the challenges that we face today due the diversity of cluster systems. Being able to know how an application will perform in an execution environment could lead to make smarter decisions. These decisions can be to choose the best system to run an application, budget the cost of executions, job scheduling optimization or improve the administration of computational resources.

The problem is that predicting performance of a parallel application with acceptable accuracy is not a trivial task. There are many dimensions that have an impact in application performance. One of this dimensions to consider is the program's workload. Every type of workload affects differently how the application performs in a given system. Be able to model the workload's effect in a parallel application can led to more general and powerful estimation models.

Many scientific parallel application present two characteristics that allows to obtain a model for performance evaluation and prediction. First most applications used in High Performance Computing have workloads whose dimensions and ranges are well known by its users. Second, their behavior is deterministic and their computational time complexity changes with the workload in predictable way. We can use these characteristics to developed a methodology that allows to create a performance evaluation and estimation model for parallel applications.

A novel approach to predict the performance of parallel applications is based in the concept of Parallel Application Signatures [36]. Since parallel application have an repetitive behavior, the parallel application signature methodology is able to a create machine-independent model of the application that represents the program's characteristics. From this model, the most relevant parts (phases) of the application are extracted an this forms

a program's signature.

Extracting an application signatures with different workloads and executing in a target machine allows to build an analytical model of each application phases time complexity and weight function.

1.2 Context

This work introduces a new methodology to model the effect of the workload in the signature of message passing parallel applications to predict its performance in distinct target systems under different workloads.

There are many problems in computer science that have a high level of complexity. The best way to tackle a complex task is to breaking down the problem in many smaller sub-problems, a strategy known as divide and conquer that has been used since the times of the roman empire.

In computing, the complexity of a problem is measure as the computing time and memory needed to solve the problem. So dividing in smaller sub-problems means parallelizing the program in different process. This have two benefits, first since the processes run in parallel the time needed to solve the problem is reduced, we say that the program scale when running in different machines or processors. Second, since we have many processes to solve the problem, each process could use only a fraction of the data needed for the computation, this can be an advantage when all the data does not fit in a machine memory. To be able to run in parallel many processes a computing system that have a lot of computation units. These computation intensive machines are called supercomputers since they are far more powerful than conventional computers. Another approach is to use a cluster of computers with many processors connected by high speed interconnects.

High Performance Computing (HPC) is the field in computer science that studies the use of supercomputers and computer clusters to solve advanced computation problems. Figure 1.1 shows the relation between HPC systems and HPC applications. Most scientific applications are parallel to obtain the maximum speedup of the application and to compute it in the minimum time. Even when the application can be parallelized, most applications need that the processes are able to communicate to share information. Basically there are two paradigms to communicate parallel processes, shared memory and distributed memory. In shared memory, all the process share the same memory address space so they can use the memory to communicate. In distributed memory, every process has its own memory address space and the processes send messages each other every time they want to communicate. This is the reason why distributed memory is also calling

message passing paradigm. There is a third approach even when technically is only a combination of two. In hybrid parallel applications, processes can use both message passing and shared memory to communicate among themselves. This work focuses in message passing parallel applications. This type of applications are the most widely used since they scale better than shared memory ones.

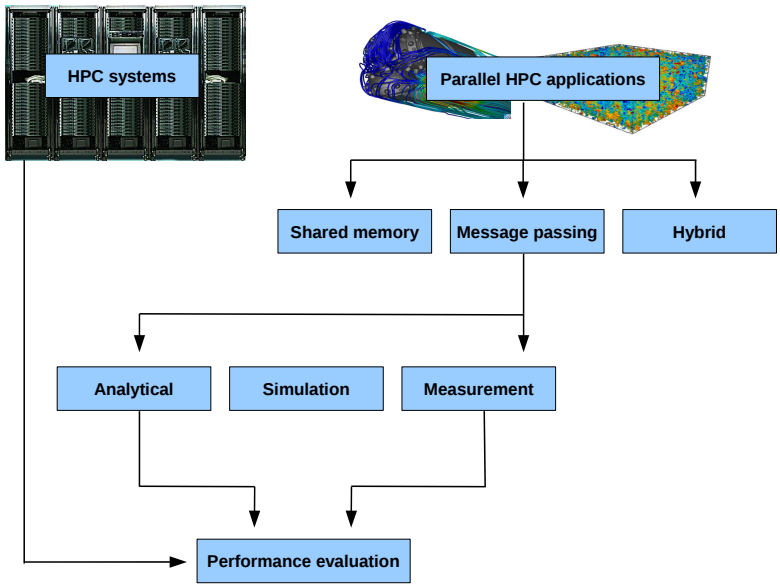


Figure 1.1: Context of this work

A problem that always arises in HPC is how to analyze and application performance in a given system. Basically there are three methods to analyze to analyze and application behavior. The first is to create and analytical model of the application, the second one is to simulate and application in a given environment, and the third one is to measure the application in execution. A novel approach is the one used in the pas2p (parallel application performance for performance prediction) methodology [36] where parallel application signatures are used to evaluate an application performance. The problem is that the performance of an application is tightly coupled with the type of workload. The same application can perform very differently with two different types of workloads. For this reason, is needed to model not only the application structure, but also how the workload affects and application performance and therefore its signature. The proposal is to use an approach that is a mixture between a measurement method and an analytical method to model how the workload affects and application in order to predict its performance.

1.3 Objective

The objective of this work [22] is to create a methodology that allows to model how the application workload affects the program's signature. Analyzing the interaction between processes in a parallel application we can extract executable signatures that characterize the computational and communication behavior for a given workload. Executing the signature in a target system allows to predict an application full execution time for one type of workload. Because scientific data-independent application behavior is deterministic, we can execute the application with different types of workloads and these execution are points of the functions that defines each application phase execution time and weight. Using regression analysis we can estimate for each phase the time execution and weight function and build an analytical model that mimics the program's behavior. Also, the exact numbers of points needed to reconstruct the functions are known. To estimate an application performance in a target machine for a given type of workload is a matter of executing a set of signatures to calibrate the model, and use this model to predict the application performance for that specific workload.

1.4 Thesis Organization

The rest of the document is organized as follows:

Chapter 2 "Concepts" introduces some concepts which later chapters are built upon.

Chapter 3 "Related Works" presents the previous research work that is related and from some ideas were borrowed.

Chapter 4 "Parallel Application Signatures" introduces the concept of signatures of an message passing parallel application. What is the conceptual basis of the signatures, how they are created and executed to evaluate an application performance.

Chapter 5 "Modelling the Workload Effect in the Parallel Signatures" explain the methodology to model the effect that the workloads has in the parallel application signatures. What is the motivation to analyze the workload effect and how to model for predict an application performance with many types of workloads.

Chapter 6 "Methodology Experimental Validation" shows the experimental results used to validate the methodology with a set of applications with different workloads.

Chapter 7 "Conclusions and Future Work" summarizes some conclusions, the contributions of this work and the future lines that remains open to further research activities.

Chapter 2

Concepts

2.1 Computational Complexity Theory

The study of Algorithms is one of the most important fields in computer science. Not only because algorithms are an effective method to express instructions but also because they can be studied in a language and machine-independent way. This means that we need techniques that enable us to compare the efficiency of algorithms without implement them [29].

One way to measure the efficiency of an algorithm in a machine-independent way is to count the number of steps that it needs in order to finish its computation for any given input. With this model of computation a number of steps can be associate with any operation. For example mathematical operations (+, -, *, /, etc) can assumed to be carried in one step and a memory operation in two steps. Using this method, two sorting algorithms can be compared in a machine-independent way for example.

However, to understand how well or bad an algorithm is in general, we must know how it works in all cases. An algorithm that performs well for a given input, can perform really bad for a different input. So, any algorithm has three different complexities that have to be evaluated to really understand its behavior [29]:

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken in any instance n.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken in any instance n.
- The *average-case complexity* of the algorithm, which is the function defined by the average number of steps over all instances of n.

From these three complexities, the worst-case is the most useful. Since the complexity depends on the input data and one usually does not known which input is going to be used with an algorithm. Being pessimistic and considering the worst case is the best way to go.

Each of these time complexities define a numerical *complexity* : $T \rightarrow N$, where T is the time needed to finish the computation and N is the problem size. Usually these numerical functions are not known and are too complicated, so we need a more simplistic notation to referrer to them. A simpler and widely used notation is the "Big Oh" or asymptotic complexity notation.

Since the exact functions that models best, worst and average time complexity of an algorithm are very difficult to work precisely, it is easier to think in terms of upper and lower bounds of the time complexity function. If we can express the time complexity function as a different function $O(g(n))$ that is an upper bound of the real function $f(n)$, we can simplifies the analysis not taking into account details that are not relevant when comparing different algorithms

The formal definition of an upper bound function in the Big Oh notation is as follows [29]:

- $f(n) = O(g(n))$ means $c * g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c * g(n)$, for large enough n (i.e: $n \geq n_0$).

The notation discards constants and lower order terms and denotes the complexity of the algorithm when its input size n tends to infinity. While doing this, the notation group functions in classes, so any given function belongs to a particular class of complexity and is equivalent with respect to the others functions in this class. Fortunately, only a small set of function classes tend to cover almost all the algorithms that we can analyze. So, to analyze the complexity of a given algorithm, this can be classified in one of a small group of time complexities functions. Table 2.1 list these most common complexities function classes in order of increasing dominance.

Table 2.1: Common time complexities used to estimate the phases execution time

Running time	Class name
$O(1)$	Constant functions
$O(logn)$	Logarithmic functions
$O(n)$	Linear functions
$O(nlogn)$	Superlinear functions
$O(n^2)$	Quadratic functions
$O(n^3)$	Cubic functions
$O(c^n)$	Exponential functions
$O(n!)$	Factorial functions

2.2 Workload Characterization

As stated before, the performance of an algorithm is usually measured by its running time, expressed as a function size of the problem its solves. Another term to refer to this size of the problem is the workload, the amount of processing that the computer has been given to do at a given time [4]. When evaluating a computer in terms of performance, usually two metrics are used:

1. Latency: the time between a user request and a response to the request from the system.

2. Throughput: How much work is accomplished over a period of time.

Both metrics are based on time and since the running time of an algorithm is a function of the problem size, the performance of any type of computer system cannot be determined without knowing the workload. Even the same algorithm can have different running times and complexity functions depending of the input data. For example the famous quicksort sorting algorithm [14] has an average-case complexity of $O(nlogn)$ and a worst-case complexity of $O(n^n)$. That means that different inputs may lead to very different performance results. Depending of the type of workload, the running time of the algorithm can vary from slightly more than linear to quadratic.

This is the reason why is very important to understand a program's workload, be able to characterize and model it. Workload characterization consists of a description of the workload by means of quantitative parameters and functions [4]. The level of detail in workload characterization depend on the context of the evaluation. For example if the analysis is to measure the performance of different sorting algorithms, it is enough to

considerer the workload as the input size of the array to sort. But when studying CPU architectures, a much more detailed characterization is required.

One of the most important uses of performance evaluation is to compare different systems. For these comparison to be accurate, the systems have to be compared under equivalent conditions and in particular under the same workload. This motivates the canonization of a select set of workloads than are then ported to different systems and used as the basis for comparison. Such standardized workloads are called benchmarks [11].

The reasons why benchmarks are so popular in the High Performance Computing field is that most scientific parallel application has a predictable behavior. These applications has small parts where most of the processing occurs. So a set of synthetic applications or application Kernels can be developed that mimic the behavior of real applications when they are computing their most intensive part. This can be done because the workloads of real scientific applications are affected mostly by a set of parameters and input data size. The workload can be decomposed in a set of dimensions whose data ranges are known by the users of the application.

This two properties of the application's workload: The fact that they have an impact on the application performance and that in the case of scientific applications, their dimensions and data range values are well known, makes the workload characterization a fundamental part of any study of the performance of parallel application.

2.3 Regression Analysis

The classical way of doing science is to use the scientific method. That means observing a phenomena and collecting the data for a latter analysis. When collecting data usually there are many variables involved. Sometimes it is necessary to determine the causal effect of one variable upon another. Fortunately statistics provide us with tools and techniques to do this.

Regression Analysis is a statistical tool for the investigation of relationship between variables [34]. Data is assembled on the underlying variables of interest and regression is employed to estimate the quantitative effect of the casual variables upon the variables that they influence. So, the goal of regression analysis is to determine a function of the independent variables called the regression function that best fit a set of data observations.

To model this relationship, a number of regression models are used. Observational data are modeled by functions that represents a relationship between a number of parameters and one or more independent variables. If the relationship between the variables is linear,

we use a linear regression model. Conversely, if the relationship is not linear, we use a nonlinear regression model.

2.3.1 Linear Regression

The goal of regression analysis is to determine the values of parameters for a function that cause the function to best fit a set of data observations. These data observations are usually composed of two variables. An independent variable (X) and a dependant variable (Y). In linear regression, the function is a linear (straight-line) equation.

That means that the best fit regression function $f : X \rightarrow Y$ can be denoted as an linear equation:

$$y = a_0 + a_1 * x + \epsilon \tag{2.1}$$

One of the simplest linear regression model is using two points (x_0, y_0), (x_1, y_1) and create a regression model that is:

$$y = y_0 + (y_1 - y_0) * \frac{(x - x_0)}{(x_1 - x_0)} \tag{2.2}$$

2.3.2 Polynomial Regression

Not always the relationship between the dependant and independent variables can be describe as a straight line. This may mean one of two things:

- The relationship between variables is linear but cannot be modeled using a straight line.
- The relationship between variables is not linear and a non-linear regression model will best fit the data set.

Another powerful form of linear regression is the polynomial regression in which the relationship between the independent variable and dependent variable is modeled as an polynomial of some order n. The polynomial regression fits not only relationships that are polynomial (i.e: quadratic or cubic) but also non-linear relationship between the dependent variable and the independent variable.

Even when they can fit a non-linear relationship between variables, the polynomial regression function is linear, in the sense that the relationship between variables is described as linear. This is function can be approximated using a polynomial expression or more formally any function can be denoted as a finite length expression of variables and

constant using only operations of addition, subtraction, multiplication and non-negative integer exponents.

So, in the case of the polynomial regression, the best fit regression function can be denoted as a polynomial of order k:

$$y = a_0 * x^0 + a_1 * x + a_2 * x^2 + \cdots + a_{k-1} * x^{k-1} + a_k * x^k + \epsilon \tag{2.3}$$

2.3.3 Interpolation and Extrapolation

Regression analysis is used to obtain a model for a function that best fit a data set. Once the model has been obtained, regression models can be used to predict a value of the dependant Y variable given known values of the X independent variable, since the regression models are expressed in terms of the independent variable X.

When using the regression function to predict a value that is between the range of the data set we say we are interpolating the function. In the other hand if we are predicting a value that is outside the range of the data set, we say we are extrapolating the function.

Since the regression model is only an approximation of the real function, both interpolating and extrapolating a value contains errors. Nevertheless, if the regression model was chosen carefully, this error can be below an acceptable threshold. Since the regression function, models the function between the range of the data set, it has much less error than extrapolation. Since there is not information about the function outside the range, extrapolation has to deal with the uncertain. For this reason interpolation is more frequently used than extrapolation [34].

Chapter 3

Related Work

3.1 Introduction

The study of applications performance has always been a topic of research in computer science, some authors even say that it is a basic element of experimental computer science [10]. This is even more true in the field of High Performance Computing (HPC). Since HPC deal with complex problems that needs to be decomposed in smaller parts and be computed in parallel to be solved in a reasonable time. So obtaining the maximum performance of an application in a given system is a necessary goal. The objectives of performance evaluation are to compare design alternatives when building new systems, identify the problems in a existent system to solve them and asses the capacity requirements when setting up systems for production use.

There are three main factors that affect the performance of a computer system. These are [10]:

1. The system's design.
2. The system's implementation.
3. The workload to which the system is subjected.

To design and implement a computer system are probably the two topics more studied and understood in the computer science field. Almost every computer scientist has studied in depth courses on data structures, algorithms, computer architecture, operating systems and compilers. Sadly, performance evaluation in general and workload modeling in particular are typically not given much consideration in academic curricula [11].

So, there exists a strong correlation between the performance of a computing system and the workload that is processing. An application can behave differently and its performance can vary greatly depending on which workload was chosen for the application. That is why it is very important to consider not only how to construct performance models but also how to model a system workload.

In this section we analyze previous research activity found in the literature to build performance evaluation and prediction model and how they deal with the workload. There are many previous works that build an application model for performance prediction. But most approaches can be classified in one of three groups: analytical models [20, 5, 12, 27], simulation based [35, 23] and performance evaluation by measuring [30, 25]. In the next subsection we will look at each approach.

3.2 Analytical Models

Computer applications are mathematical or numerical models to solve problems of different areas such as physics, chemistry, medicine, engineering and economics. So it is natural to represent programs using an analytical model. These models resemble key characteristics of an application. A performance prediction in a target system can be obtained using these analyzing these mathematical models.

In [30] is proposed an analytical model for a performance metric. The performance metric is called HINT, a performance benchmark that allows to measure the overall performance of a wide variety of computing machines with a range of memory sizes and time scales. The HINT analytical model allows to predict the performance curves using hardware specifications as input parameters.

Another performance analytical performance prediction model is proposed for Carrington et al [5] analyzes an application memory and communication usage patterns to build an analytical performance model. They map the information with a machine profile and use simulation to estimate the application performance in a variety of compute platforms. This method needs to analyze hardware characteristics while our approach only needs information about the application behavior and does not need to simulate the underlying hardware since our signature can be directly executed on it.

Lu and Reed [19] used curve fitting of historical trace data using polylines. These curves characterize the application behavior and can be used to analyze its performance. Our analytical model also is based in curve fitting using regression functions. But instead of build a curve fitting function using historical trace data, our approach use the measures of the signature running in a target system, with a fraction of the cost because the

signature only contains the application's most relevant phases.

Parallel applications can not be easily modeled in detail with analytical models [15]. Mathematical models present a trade off between simplicity and accuracy. The model is an abstraction of a real system. Its complexity grows as more details are added to that abstraction. Also, significant expertise is needed to derive even a simple analytical model of an parallel application [27].

3.3 Simulation Models

Simulation based approaches simulate the execution of an application in a specific environment to estimate the real performance.

Labarta et al [16, 13] also uses simulation to evaluate the performance of message passing parallel applications. In their work they use the Dimemas simulator, which is a trace driven and rebuilds the behavior of a parallel application using a trace file and some architecture specific parameters.

Another work that uses simulation is Mambo [3] which was develop for IBM to be a full system simulator for modeling PowerPC based systems. Since the system is highly modular, the implementation support multiple simulation modes from a simple purely functional simulation of the PowerPC instructions to cycle accurate simulation of an entire system. Mambo also includes trace collection and debugging interfaces to allow detailed analysis of the simulated hardware and software.

Simulation models are difficult to develop and the simulation model has to be validated and verified. Also, a deep understanding of the underlying system one is trying to simulate is needed.

3.4 Measurement Techniques

Measurement techniques are based in the fact that scientific deterministic application spends most of its computation time executing a few segments that has a high level of repetition. This property allows to evaluate an application performance without the need to measure the whole application. In this subsection we analyze the two measurement techniques that are found in the literature that are benchmarks and application measurement.

3.4.1 Benchmarks

As stated before, scientific applications spend most of its computation time executing a few mathematical operations. So one can thing of these most executed operations are the kernel of an application. Since the application spend most of its execution time computing this code, here should be a correlation between the performance of the application kernel and the complete application.

In order to estimate the performance of computing systems, in the literature have been proposed a number of application kernels known as benchmarks. This benchmarks reflects the behavior and workload of many common applications so they are useful to compare different systems.

The benchmarks more widely used are:

- LINPACK Benchmark [9]: developed at the Argone National Laboratory, is one of the most used benchmarks. It is used for example to compare the performance of the 500 more powerful cluster computers in the world (Top500). The principal characteristics of LINPACK is that it make an extensive use of floating point operations and spent most of its computation time executing routines known as BLAST (Basic Linear Algebra Subroutines).

- SPEC Benchmakrs [8]: The Standard Performance Evaluation Corporation to measure computers performance, are Benchmarks created to obtain a performance metric to compare compute intensive workloads in different systems. The benchmark is focused in comparing CPU and hence FPU speed.

- NAS Parallel Benchmarks [2]: is a benchmark suite developed for the NASA Advanced Supercomputing Division (NAS) to evaluate the performance of parallel systems. The benchmarks consist of five kernels that represents programs used to simulate fluid mechanics.

Since benchmarks represents real applications kernels. They can be used to estimate the performance of real applications. One can choose an application benchmark that executes these same most used mathematical operations and whose workload type is similar to the one used by the real application. Because deterministic application performance does not depend on the data input value, only with the input data size and the mathematical operations that use this input, the benchmark application can give us an idea of how the target system meet the application demand for computational resources.

The problem is that benchmarks are to specific too generalize a parallel application behavior [30].

3.4.2 Application Measurement

Most scientific applications have a high level of repetitive behavior. Some approaches as the one used for the SimPoint tool [28, 26] analyzes shared memory applications to find similar patterns in different execution times. Algorithms are proposed to automatically group similar portions of program's execution into phases. First, similar execution intervals are identified, second each similar execution interval is compared with a basics blocks vector. Finally, an analysis is made to group each similar execution interval in portions that are representative of the overall execution time of the application. The goal is to obtain these portions of the application that are representative to simulate them in different architectures.

Yang et al [39] also extract an application most executed segments and used them for performance prediction. They develop a methodology to predict an application performance using partial executions. They argue that relative performance can be observed without running a parallel application in full since most parallel codes are iterative after a minimal startup period.

Sodhi and Subhlok [31] develop a method to create what they called a Performance Skeleton of an application. A Performance Skeleton is a synthetic program that represents a real application. Analyzing the skeleton performance reflects the real application performance in any scenario. Our application signature also reflect the performance of the application it represents but it is not a synthetic application that behaves like a real application, it is composed of segments of the real application.

In [36, 38, 37], a different approach is proposed. Is based both in the fact that most parallel application have a repetitive behavior and that only partial executions are needed to estimate the total performance of an application. Instead of instrumenting the application on an instruction level basis, they instrument the application in the MPI library communication layer. Each communication primitive is traced and this information is analyzed to search for similar communication and computations patterns. With this information a high level model of the application is constructed and the most relevant phases of the application. To each phase a weight is assigned based in the number of times that it repeats. The number of relevant phases of the application and each phase weight is known as an application signature. Executing this signature in a target machine allows to predict the total application performance with a minimal cost.

This methodology is known as PAS2P (Parallel Application Signature for Performance Prediction) and this research grew out of this earlier work and its goal is to include the workload effect in the parallel application signature.

Chapter 4

Parallel Application Signatures

4.1 Introduction

4.2 Conceptual Basis

Applications usually have different phases that repeat. This seems to be the case in serial [28, 33, 32] and parallel applications, both shared memory [26] and message passing applications[36].

We instrument the MPI library using function interposition and execute parallel applications in a parallel machine. The instrumented library produces a trace log. The data collected is used to characterize computation and communication behavior of the application. To obtain the machine-independent application model [38], the trace is logged by means of a logical global clock according to causality relations between communication events. The developed algorithm is inspired by Lamport's algorithm. Once we have the logical trace, it is processed using a technique that searches for similarity [37], to identify and extract the most relevant event sequences (phases) and assign them a weight based on the number of times they occur.

Afterward, to build the Parallel Application Signature, the last step is to re-run the application to create the coordinated checkpoints before each relevant phase happens, therefore, the signature will be defined by a set of executable phases and weights.

The execution of the signature in different target systems allows to measure the execution time of each phase, and hence to estimate the entire application s run time in each of those systems by extrapolation of each phase s execution time using the weights we had obtained.

The execution time of the application signature is a small fraction of the whole application s run-time.

It is important to notice that the signature creation and execution is a two step process. The first step is to analyze the application, build the application model, extract its phases and weights and use that information to build an executable signature. The second step is executing that application signature in a (normally) different system, to measure the phases execution time and predict the application total execution time.

In the next sections we explain in detail how these two steps are performed while the Figure 4.1 shows the process.

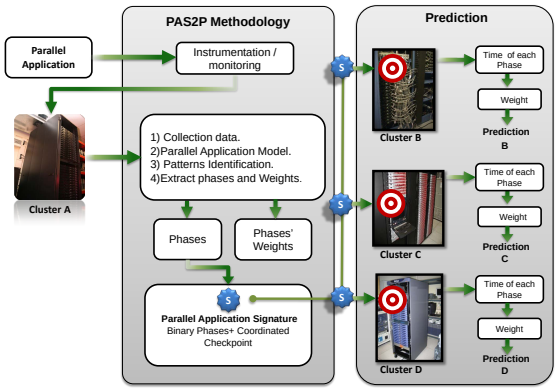


Figure 4.1: Parallel Application Signature Creation and Execution

4.3 Signature Creation

To create the signature first we build a machine-independent model of the application and then use that model to create a machine-dependent signature. Below we describe this process:

4.3.1 Machine-Independent Model Creation (Obtaining Phases and Weights)

As described before, we instrument the MPI library to obtain an application communication and computation trace. The trace contains all the communications events between processes and the computation time elapsed between communication primitives. In this context an event is a message sent or a message receive. With this information we build an application model and use this model to make a study about where in the application

the more computing time is spent (relevant phases), and how many times those phases repeats (weights).

4.3.2 Signature Creation for an Architecture (Phases Checkpoint)

When all the phases and its corresponding weights have been determined. The application is executed again to make a coordinated checkpoint [1] before each phase occurs. The checkpoint has to be coordinated to ensure that all process has a consistent view of the execution environment and can normally resume execution later on restart. The checkpoint for each phase and the application phase weight vector are the executable signature elements.

4.4 Signature Execution

One we have the application signature, we can run it on target machines to predict the full application execution time. For each phase, we use the checkpoints to restart the application before the phase begins and measure its execution time until the phase ends. Predicting the application total execution time is a matter of adding the multiplication of each phase execution time by its weight as shown in Equation 4.1.

$$PAET = \sum_{i=1}^{k} PET_i * W_i \tag{4.1}$$

where $PAET$ is the Predicted Application Execution Time, k is the number of phases, PET_i is the Phase i Execution Time and W_i is the phase i Weight.

This procedure works because the phase number and weight do not change from machine to machine. The phases represents the program structure and the weights represent iterators. These iterators can be constant or depend on the program's input data size. The only component that changes from machine to machine is the phases execution time. So it is the only thing that we have to measure in the target system.

Chapter 5

Modelling the Workload Effect in the Parallel Signatures

5.1 Introduction

The parallel application signatures can be used to predict an application's total execution time for a given workload. But parallel applications are executed with different types of workloads. Every type of workload affects differently how the application performs in a given system. Be able to model the workload's effect in a parallel application signature can led to more general and powerful estimation models. Many scientific parallel application present two characteristics that allows to obtain a model for performance evaluation and prediction. First most applications used in High Performance Computing have workloads whose dimensions and ranges are well known by its users. Second, their behavior is deterministic and their computational time complexity changes with the workload in predictable way. We used these characteristics to developed a methodology that allows us to create a performance evaluation and estimation model for parallel applications. We extract the most relevant phases of an application and execute these phases with different workloads to estimate the phases time complexity and build an analytical model.

This chapter describes the methodology used to model the workload effect in a parallel application signature. The methodology consist of many steps that builds an abstract model of the application's behavior while its workload is being varied. The methodology is based in the fact that most scientific parallel applications are deterministic and their workloads can be characterized by an domain expert.

Section 5.2 explains a characteristic of the parallel application signatures and why a single signature cannot be used to predict any type of wrokload. Section 5.3 states why its possible to model a scientific parallel application's workload while Section 5.4 analyzes

the effect that the workload variation has in the parallel application signatures. Finally Section 5.5 details the methodology used to model the workload effect in the program's signature.

5.2 The Problem with the Single Checkpoint

As explained in chapter 4, a parallel application signature has two components:

- A set phases P where each $p_i \in P$ is a relevant phase of the application.
- A weight vector W where each $w_i \in W$ represents the repetition count for each phase $p_i \in P$.

To estimate the total execution time of an application in a target system the signature is executed. That means that every phase p_i is executed an their execution times measured. A new vector T is created where each $t_i \in T$ represents the execution time for each phase p_i. Having both vectors we can predict the application's total execution time adding the multiplication of each phase execution time t_i by the number of times the phase repeats in the application w_i.

As stated before, a checkpoint library is used to create the set of relevant phases P. For each phase p_i a coordinated checkpoint is made. This coordinated checkpoint saves all the parallel process state such as CPU registers and memory contents. That means that is not only saved the .

To measure a phase in a target system the checkpoint is restarted. Every process continues execution before the phase occurs. Since the checkpoint saves not only the process code but also the process data, the signature cannot be parametrized with different input data.

Since the signature only contains a single checkpoint that executes the parallel application with one type of workload, cannot be used to predict with a different workload. Lets use as an example Los Alamos National Laboratory's Parallel Ocean Program (POP) [17] which is a real scientific parallel application. Figure 5.1 shows a number of POP executions with different input sizes. For each input size a parallel application signature was created and its execution time is different as well as the predicted total application execution time.

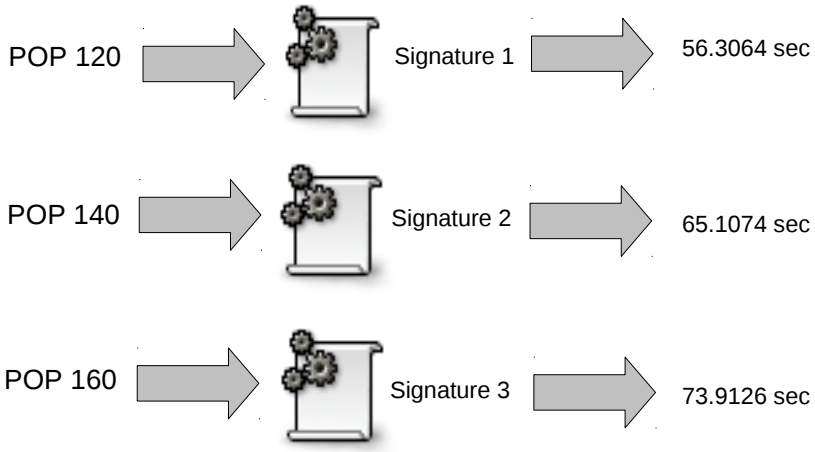


Figure 5.1: POP signatures execution times with different workloads

Every input size is a different type of workload. That means that the signature execution time depends on the workload. Formally we say that the signature execution is a function $signature : WL^n \rightarrow T \subseteq \Re$. Where WL^n is the workload of dimension n and T is the signature execution time.

So, if we have k different types of workloads, we need k different signatures to predict an application performance with every possible workload. This is even more problematic if the workload space is not finite, this mean that we need an infinite number of signatures to cover all possible the cases.

5.3 Workload Characteristics in Scientific Applications

Scientific applications usually have workloads whose characteristics are well known and understood. A workload can have many dimensions that affects the application behavior differently. An every dimension has a data range in which is valid to use the application.

This is one of the reasons why benchmarks applications can be developed. A benchmark is a synthetic program that executes mathematical routines used by real applications with input data that is similar both in size and characteristics as the one used by scientific applications. This could not have been possible if the input data that defines an application workload would not have been analyzed, understood and characterized.

It does not mean that characterizing and scientific application input data is easy, but its mean that it can be done. Either formally or at least empirical. Scientists known what are the input data dimensions and the values they can use for each dimension.

To better illustrate how a program's data input has many dimensions that affect differently the application behavior we will use a simple example. The Jacobi method is an iterative algorithm to resolve a system of linear equations where the largest absolute value for each row and column is dominated by the diagonal element [6]. This is a good example because many scientific problems can be modeled has a system of linear equations.

The workload for an application that implements the Jacobi method is composed of two dimensions:

- The matrix size that represents the system of linear equations.
- The converge condition that determines the numbers of iterations that the algorithm will do.

The matrix size is problem domain dependant and depends of the numbers of variables that every linear equation has. Each of these dimensions will affect the program behavior differently. This is not only the case for the program implementing the Jacobi method but for any scientific application whose workload's is composed of many dimensions.

Since the signature is composed of program segments, each dimension will have a different effect on the programs signature. In the next section we will explain how the workload affects the application signatures, what elements are altered by a workload variation and what remains the same independently of the program's workload.

5.4 The Workload Effect in the Signatures

As explained in section 5.2 a parallel application signature has two elements, a set of the phases P and a weight vector W. Also, when a signature is executed in a target machine we obtain a execution time vector T that contains each phase execution time.

Another characteristic of the signature is that its execution time changes with different types of workloads. Since the application and the signature execution time are functions of the workload.

So the natural question that arises is: *How does the signature elements change with the workload?*

A prior step needed to construct the parallel application signature is to build an application platform independent model. This model is an abstraction of the real application

using its communication and computation patterns. The model resembles how the application was developed so its not only platform independent but also workload independent. That means that the application phases are always the same independent of the input data used.

An element that changes with the workload variation is the execution time vector T. For different types of workload, each signature will have different execution times for each phase.

Each phase usually represents a code segment defined by the programmer in the parallel application. Each one of this code segments can use the input data to make computation in a different way. Because most scientific applications are deterministic, its phases are also deterministic and their behavior is determined by a some computational time complexity. So the execution time of a phase will change with different workloads and the variation in the execution time will depend of the time complexity of the phase. For example, if a phase has a time complexity of $O(N)$, then its execution time will vary linearly with the input data.

The other element that could be affected by the workload is the weight vector W. Just as a phase represents an application code segment, the weight of a phase generally represents an iterator. This iterator can be programmed explicitly or may be due to a stopping condition (i.e: a convergence threshold).

A program can have fixed iterators o they can vary with the workload. If an iterator is used to execute a code segment a fixed number of times, then the phase that represents this code segment will have a fixed weight that is independent of the type of workload. In the other hand if the iterator varies with the workload, then the phase weight will also change with different workloads.

5.5 Methodology to Model the Workload Effect

As stated before, the parallel application signature execution time is a function of the workload. That means that if we wan to use signatures to estimate an application total execution time with different workloads, we will need to build one signature for every type of workload we want to use.

This approach may work if an application have a small and finite set of workloads. But if the application have a many different workloads, this approach is very expensive both in computation time to build the signatures and in space to store the checkpoints. Also, if one of the dimensions in the workload can have infinite values, this approach is even less feasible since we cannot generate a infinite set of signatures.

Now since most scientific applications are deterministic and their behavior is guided by its computational time complexity. The question is if we really need to build a signature for every kind of workload. Can we obtain a prediction model that is general?

In Section 5.4 is explained what elements in a signature change with the workload. So in order to have a general prediction model, we need to model how this elements change with the workload. The two elements that change are the time and weight vectors that represents each phase execution time and frequency for a given workload.

If each phase execution time and weight can be modeled as function of the workload then we can build or general prediction model. Since we know that the workloads in a scientific parallel application have well known dimensions and that each dimension has values between a predefined range. Building the prediction model is possible if we can obtain for each phase $p_i \in P$:

- A function $execution_i : WL^n \rightarrow T \subseteq \Re$ that models the phase execution time for a given workload $wl \in WL^n$ of dimension n.
- A function $fweight_i : WL^n \rightarrow W \subseteq \Re$ that models the phase weight for a given workload $wl \in WL^n$ of dimension n.

To build the functions $execution$ and $fweight$ for each phase, a set of signatures are executed with different workloads. Each signature will have different execution times and weights for its phases. Thats mean that each signature will have its own phase execution time vector T and its phase weight vector W.

Since the phase number does not change with the workload. All the application phases are present in each signature as well as their execution time and weight. So for each phase we have a number of execution times and weights, one for each signature executed. So for each phase now we have a number of phase execution time and phase weights. These values of execution time and weight can be used to estimate the functions $execution$ and $fweight$. For each phase the $execution$ function is the phase time complexity and the $fweight$ function is the one that models how the phase weight is affected by the input data. Since the weights represents iterators and the application is deterministic, the function $fweight$ is also predictable.

These characteristics founded in most scientific parallel applications allows to develop a methodology to build a general prediction model for parallel applications. Using this model we can predict the performance of a parallel application for any type of workload between the predefined range without needing to create one signature for any type of workload.

We create only a small number of signatures for different workloads and measure the phases execution time and weights to obtain a number of points. Using this points and regression analysis we can estimate the time execution ($execution$) and weight variation ($fweight$) for each phase. This give us an analytical model with a number of variables. These variables are the execution times and weights for each phase in the signatures. So to predict the performance of an application in a target system for an unknown workload, first the needed signatures are executed and the analytical model parametrized, then for each phase the $execution$ and $fweight$ functions are evaluated and the execution times and weight for the phases of the unknown workload are obtained. One we have those values the predicted application execution time can be obtained. Figure 5.2 shows the steps used in the methodology.

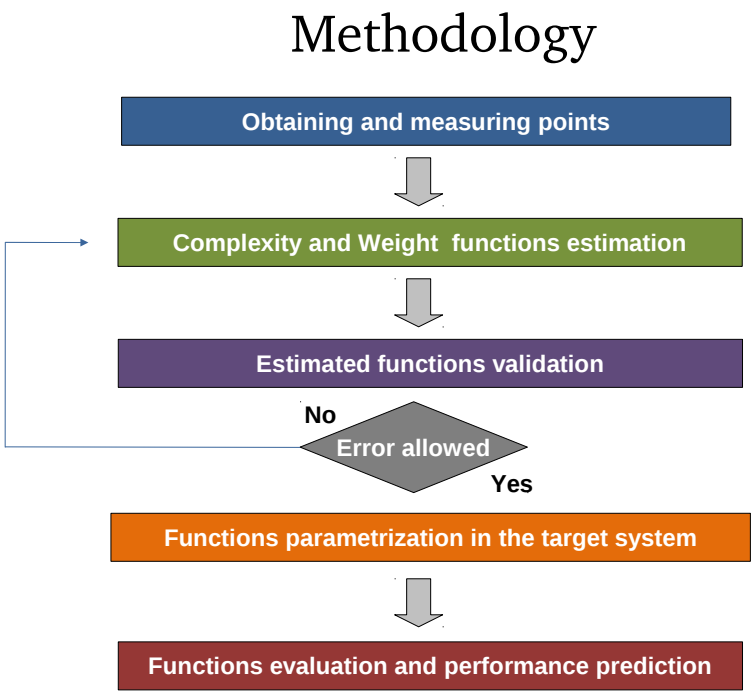


Figure 5.2: Methodology diagram

The first three steps happen in the source system and are used to build the analytical model. While the last two happen in the target system and are used to parametrize the model and predict the application performance. The next subsections explain in detail each step of the methodology.

5.5.1 Obtaining and measuring points

The first step in the methodology is to obtain and measure a number of points to estimate each phase time complexity and weight change. In this step a set of signatures of an application are executed with different workloads. As stated before, with each signature execution we can obtain phases execution times and weights.

Lets use an example to illustrate this step. Assume that there is an application whose workload is composed of one dimension. This dimension is a vector length that affects the execution time of its phases and one of the phases has a $O(N^2)$ (quadratic) time complexity. Figure 5.3 shows the execution of four signatures with different vector lengths and the execution time for this phase.

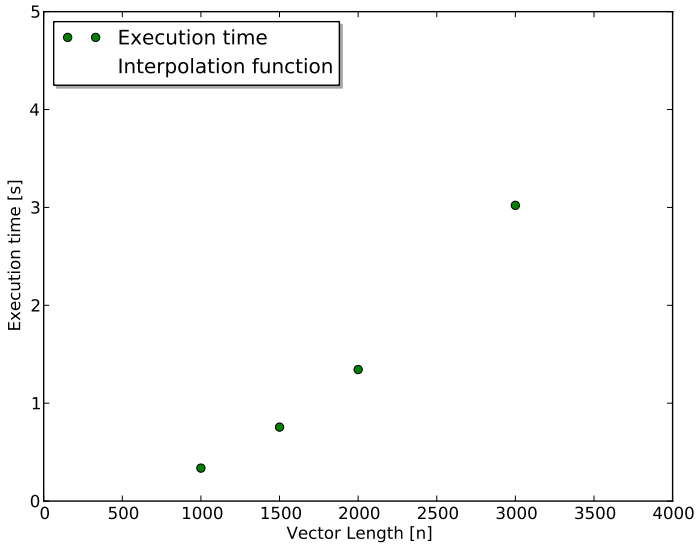


Figure 5.3: A phase execution times with different workloads

5.5.2 Complexity and Weight functions estimation

Once we have the execution times and weights for each phase with different workloads. We can estimate the phase execution time function $fexecution$ and the phase weight function $fweight$. Since a phase execution time depends on the phase time complexity, we use regression analysis to estimate the phase execution function using common time complexities for different algorithms. The number of points used to estimate the functions will depend of the kind of regression analysis we use. For example, we only need only two

points if we want to estimate the function using a linear regression but three points are need to use a quadratic regression function. As more complex is the regression function used, more points are needed. Figure 5.4 shows the execution time function estimation for the example phase used in the last subsection both using linear and quadratic regression functions.

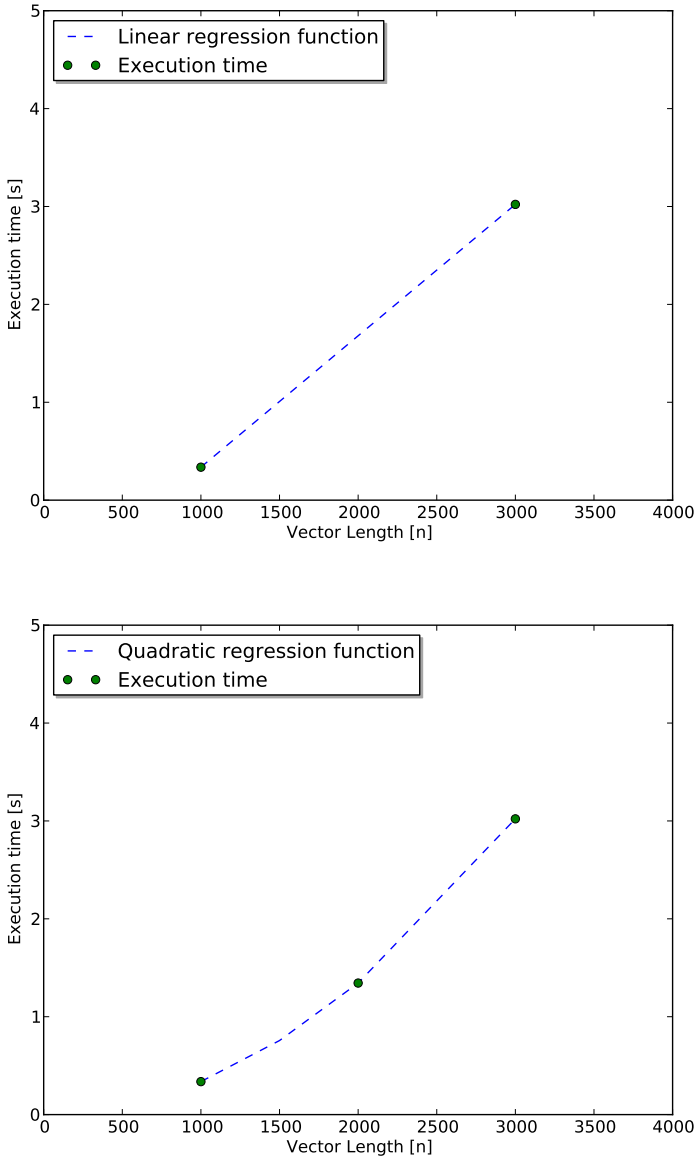


Figure 5.4: Phase execution time estimation using different regression functions

5.5.3 Estimated functions validation

Once we have an estimated function we need to validate it. We use interpolate our estimated functions to obtain the phase execution and weight values for a point that we have already executed. Since the signature was execute in the source system their values are known but were not used to estimate the function. With this procedure we can test the ability of the functions to generalize. Figure 5.5 shows the values obtained for our example phase for a different point using both the linear and quadratic function.

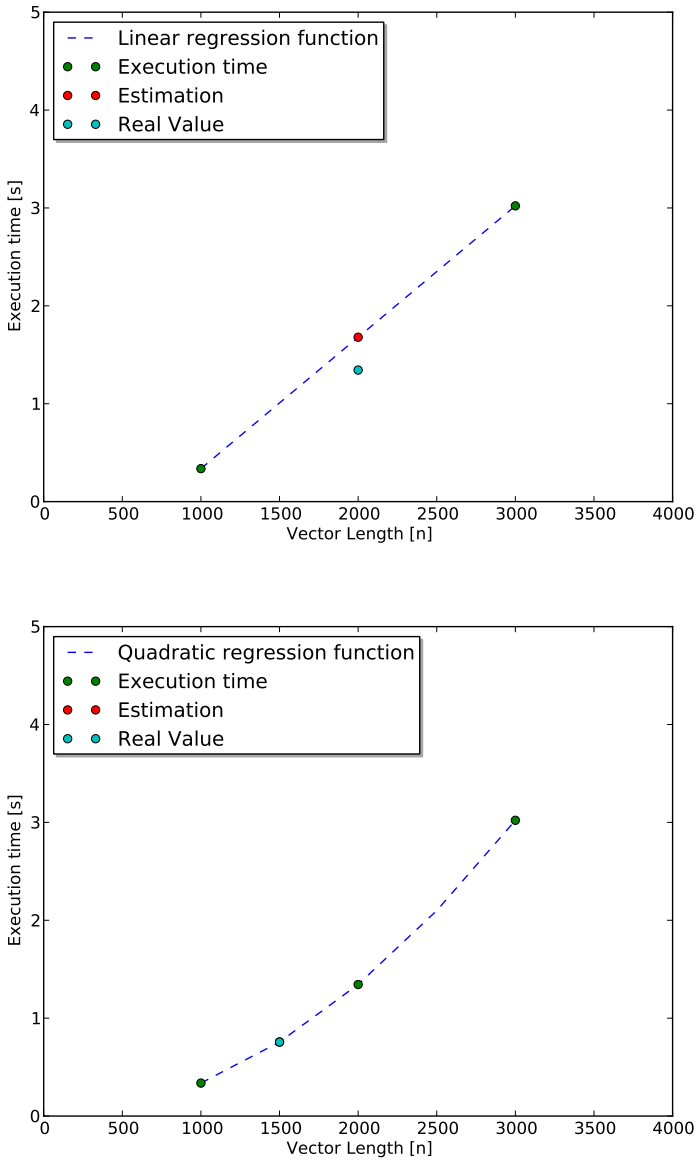


Figure 5.5: Phase execution time estimation functions validation

The first three steps of the methodology were presented here as if they are sequential to make the procedure more easy to understand. But as is shown in Figure 5.2, this first three steps are an iterative process. First two signatures are created and executed. The execution time and weight for each phase are used as the first two points to estimate. The first approach is to use linear regression function. Then a third signature is constructed and executed. The phases execution times and weight are measured and also are obtained using the estimated functions. Because both the real values and the interpolated ones are available, the functions ability to generalize can be test. If the error is above a threshold for some function, we use this third point and try to estimate the function using a different regression method. Then we create a fourth signature with a different workload to validate the new estimated function. We keep creating signatures until every function has been validated and the error is below our threshold.

To choose the workloads for creating the signatures we use a binary algorithm. The first two signatures are the endpoints of the value range for the workloads dimensions. The next point is the mean value for this two point. Then we randomly choose the segment that is bigger or less than the last mean value and use the mean value from the last mean value and the extreme value and so for.

With this procedure we know exactly for each function what regression method can be used for estimation and how many points (signatures) we need to execute in the target system to parametrize the estimated function.

5.5.4 Functions parametrization in the target system

This is the first step that happen in the target system. At this point we know how many phases and application has, what regression method can be used to estimate the complexity function $execution_i$ and $fweight_i$ for the $phase_i$ and how many signatures we need to execute in the target system to parametrize the functions.

So these signatures are executed and their phases execution times and weights measured. With this information the estimated functions $execution_i$ and $fweight_i$ are estimated.

5.5.5 Functions evaluation and performance prediction

The last step is to evaluate each function and predict the application performance. Since we have already the functions $execution_i$ and $fweight_i$ for each phase, the total execution

time for an application with a workload n can be predicted evaluating each of these functions to obtain the phase execution time and phase weight. So to obtain the Predicted Application Execution Time (PAET) for a workload n, the Equation 5.1 is used:

$$PAET_n = \sum_{i=1}^{k} fexecution_i(n) * fweight_i(n) \tag{5.1}$$

where k is the number of phases, $fexecution_i$ is the phase i execution time function and $fweight_i$ is the phase i weight function.

Chapter 6

Methodology Experimental Validation

6.1 Introduction

This chapter shows the results of the experimental validation of the methodology. For this a set of different applications that behave differently were used. Each application has its own communication patter, workload and behavior. Even when all the applications are message passing parallel applications, they are very different in resource requirements and application domain. To stress the methodology and be sure that it can model the workload impact in any scientific applications, a number of different test cases where made. This test cases is composed of synthetic applications, benchmarks and real scientific applications.

6.2 Scientific Applications

As stated before, to validate the methodology ability for prediction, a number of experiments were made with different parallel applications. The number and type of the application chosen were to reflect a broad spectrum both in application domain and characteristics. The applications range from simple synthetic applications to complex real scientific parallel programs. In this subsection each application used is presented and their behavior explained.

- Synthetic application: The application is composed of two phases. Each phases has different communication patterns and computational time complexity. The application has different inputs that changes how it behaves with the workload.

The most important is a vector length that is used in both phases to make a fake computation (the computation is done but the value is never used).

- NAS-CG (Conjugate Gradient): The application is one of the third benchmark applications found in the NASA Advanced Supercomputing Division's suite *NAS Parallel Benchmarks* [7] that are used in the experiments. It estimate the smallest value of a large sparse symmetric matrix using the inverse iteration with the conjugate gradient method for solving systems of linear equations.

- NAS-BT (Block Tridiagonal): Solve a synthetic system of nonlinear partial differential equations using an algorithm that involves a block tridiagonal solver kernel.

- NAS-SP (Scalar Pentadiagonal): Solve a synthetic system of nonlinear partial differential equations using an algorithm that involves a scalar pentadiagonal solver kernel.

- Sweep3D (3D Discrete Ordinates Neutron Transport): It solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by an IJK logically rectangular grid of cells [18].

- SMG2000 (Semicoarsening Multigrid Solver): Is a parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of a diffusion equation on logically rectangular grids [24].

- Parallel Ocean Program (POP): Is an ocean circulation model in which depth is used as the vertical coordinate. The model solves the three-dimensional primitive equations for fluid motions on the sphere under hydrostatic and Boussinesq approximations [17].

6.3 Execution Environment

The applications chosen are parallel applications that are computation intensive. Also, they are executed with relevant workloads to justify to signature creation. If the workload was very small, then the complete application could be executed in an execution environment to analyze its performance.

For this reason all the experiments were made in different computer clusters. All the parallel application signatures where extracted in a the cluster system A and executed in a different cluster system B. Table 6.1 shows cluster A and B characteristics.

Table 6.1: Cluster Systems Characteristics

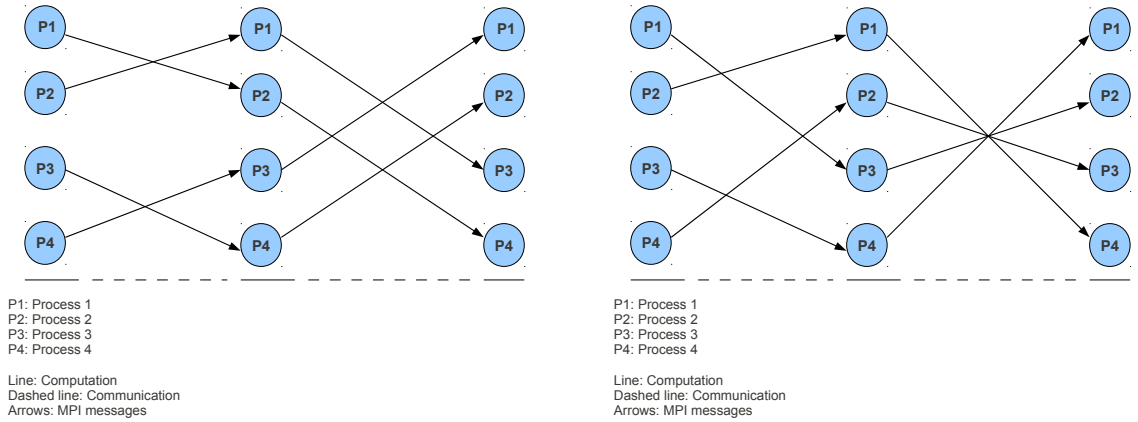
Cluster A	Dual-Core Intel(R) Xeon(R), 3.00GHz, 4MB L2(2x2), 12 GB RAM, Network Gigabit Ethernet (32 Nodes)
Cluster B	2 x Quad-Core Intel(R) Xeon(R), 2.66GHz, 2x6MB L2, 16 GB RAM, Network Gigabit Ethernet (8 Nodes)

6.4 Experimental Results

6.4.1 Evaluate Phase Identification Ability: Synthetic App

Before using our method with real parallel application. We wanted to validate our model using an application whose structure and behavior is well known to us. If we know the application behavior, we can validate our model in an experimental basis. To validate the model we use a synthetic parallel application that is composed of two parts. Each part is a sequence of the same phase. So the application has two relevant phases, $phase_1$ and $phase_2$. The application structure is shown in Figure 6.1. Each phase alternate communication and computation, using two different communications patterns and different computational time complexities. While $phase_1$ has a time complexity of $O(n)$, $phase_2$ has a time complexity of $O(n^2)$. The communication patterns used for each phase are shown in Figure 6.2.

Figure 6.1: Synthetic application structure



(a) Phase 1 communication pattern

(b) Phase 2 communication pattern

Figure 6.2: Synthetic application phases communication patterns

The input could have many dimensions that affect the application differently. That means that an input element is a tuple where each element represents a different input data dimension that has a different impact on the program's workload. For example, the synthetic application can change the size of the MPI messages, the length of the vector its computes and the number of times each phase repeats (iterations). In these experiments, we change both the length of the vector used in the computation for each phase and the number of times each phase repeats.

We execute the signatures in the cluster A described in the Table 6.1 and measure for each input data size the phases execution time and weight. Then we used the methodology described in Section 5 to obtain the phases execution time and weight functions using regression analysis. Once we have this functions we predicted each phase execution time and weight for a unknown input data size using interpolation methods. Finally we used the Equation 5.1 to estimate the application total execution time for that input data size. Later we extracted signatures for that input data size and measure the real values and the real application execution time to validate the accuracy of our prediction.

Table 6.2 shows the values of each phase execution time and weight obtained for different vectors lengths and a fixed iterator. While Table 6.3 shows the values for a fixed vector length and different iterator values.

As expected, changing the vector length has a direct impact on the phases computation time since its time complexity is directed related with this value. Looking at Table 6.2 we can see that the phases weight does not change if we maintain the number of phases that repeats in the sequence. In the other hand, Table 6.3 shows how changing the value of the iterator affects the phases weight but has little impact in the phases time complexity.

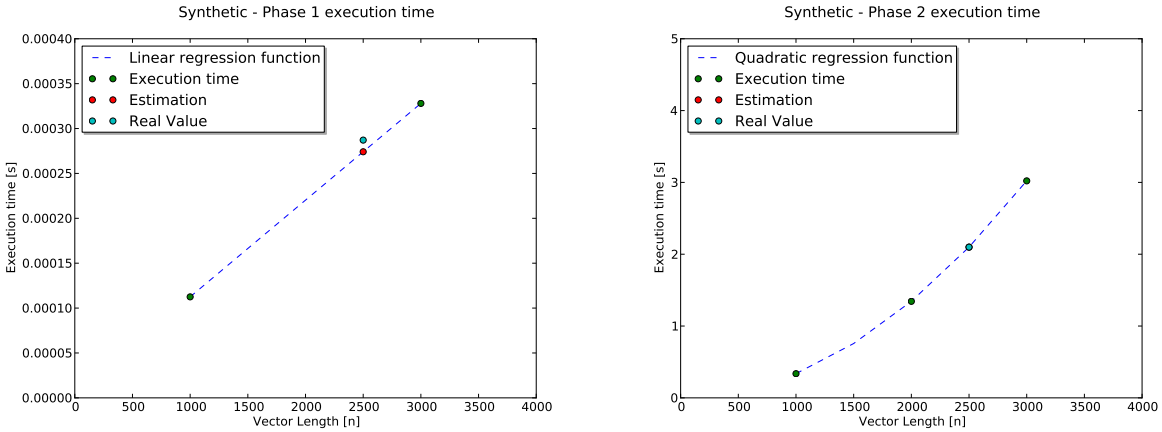
Table 6.2: Synthetic application results changing the input vector length

Vector Length	Phase 1		Phase 2	
	Exec. Time (Sec)	Weight	Exec. Time (Sec)	Weight
1000	0.0001125	100	0.33682	99
2000	0.0002157	100	1.34379	99
3000	0.0003280	100	3.02133	99

Table 6.3: Synthetic application results changing the phases iterations

Iterator Value	Phase 1		Phase 2	
	Exec. Time (Sec)	Weight	Exec. Time (Sec)	Weight
100	0.0001796	100	8.39032	99
200	0.0001081	200	8.39015	199
300	0.0001436	300	8.39035	299

Figure 6.3 shows graphically how $phase_1$ and $phase_2$ execution time changes as we increase the input vector size used in the computation for each phase. As expected, $phase_1$ changes linearly while $phase_2$ changes quadratically. The Figure also shows how we can use regression functions to generalize the partial function obtained executing the signatures. Since $phase_1$ complexity functions is linear, we need only two points and a linear regression function. With $phase_2$ we cannot use a linear function, since its complexity is quadratic, we need at least three points and a quadratic regression function to generalize its behavior. Using a linear function and interpolation will get a value with an error bigger than we can tolerate. Table 6.4 shows the values obtained interpolating the regression functions in the range 1000 to 3000 for an unknown input vector size 2500 for each phase. Table 6.5 shows the prediction time obtained summing the interpolation for each phase $fexecution(2500)$ function multiplied by the interpolation of the $fweight(2500)$ function.



(a) Phase 1

(b) Phase 2

Figure 6.3: Synthetic - phases execution times

Table 6.4: Synthetic phase execution time estimation and error for an unknown vector input size 2500 in the range 1000 to 3000

Phase 1			
Regression function	PPET (Sec)	PET (Sec)	PPETE (%)
Linear	0.0002719	0.0002871	5.31%

Phase 2			
Regression function	PPET	PET	PPETE
Linear	2.18256	2.09837	4.01%
Quadratic	2.09874	2.09837	0.01%

PPET: Phase Predicted Execution
PET: Phase Execution Time
PETE: Phase Predicted Execution Time Error

6.4.2 Analyzing Well Establish Workloads (Benchmarks Applications)

Once the methodology has been validated used the synthetic application, the next step was to experiment if the methodology works for modeling the effect of standard set of workloads using benchmarks applications. Since benchmarks are well suited to compare different systems under similar conditions and the goal of the methodology is to analyze the workload effect in the parallel application signature to predict an application's per-

Table 6.5: Execution Times on Cluster B

Application	SET (Sec)	PAET (Sec)	AET (Sec)	PETE (%)
Synthetic	2.10	207.80	209.87	2.07%

SET: Signature Execution Time
PAET: Predicted Application Execution Time
AET: Application Execution Time
PETE: Predicted Execution Time Error

formance, we needed a benchmark suite that is widely used to test our methodology. The NAS benchmark was finally the suite chosen because it suits our needs in great detail. As commented in Section 6.2, the applications used for validation from the NAS benchmark suite are the Conjugate Gradient (CG), Block Tridiagonal (BT) and Scalar Pentadiagonal (SP).

Since benchmarks are a canonization of a select set of workloads, all the application of the NAS suite have a group of type of workloads known as classes. These classes are a discrete set of workloads and stress the systems using different loads. Each type of workload is characterized by a set of parameters. The parameters are application specific and each workload has a different value for each parameter.

The NAS benchmarks applications comes with a number of stock classes that are the A, B, C and D. From theses classes the A is very small so we will not use it, since almost all the phases of the application will be relevant with such a small workload. Having only three types of workload is not sufficient to model some time complexity functions. For example, if a complexity functions is of type $O(n^2)$ we need at least three points to model the function and at least one different point to validate it. To have more points we created two more workload classes that do not come with the standard NAS installation. We called these two classes X and Y. X is a class whose workload is an intermediate value between class B and C, and class Y is an intermediate value between class C and D. Because each type of workload has its own set of parameters for each application, class X and Y were created obtaining the intermediate value for each class parameter. Table 6.6 shows the workload classes for each application used, the parameters that conforms the workload and the values used for each class.

Table 6.7 shows the workloads values used in the experiments and the value to be predicted while Table 6.8 shows the results predicting the unknown workload values.

Table 6.6: NAS applications classes, parameters and values used

Class	CG				BT		SP	
	na	non zero	niter	shift	problem size	niter	na	niter
B	75000	13	75	60	102	200	102	400
X	112500	14	75	85	132	200	132	400
C	150000	15	75	110	162	200	162	400
Y	825000	18	87.5	305	285	225	285	450
D	1500000	21	100	500	408	250	408	500

Table 6.7: Workloads values and unknown workload value to be predicted for the NAS applications

NAS Application	Workload Values Used	Workload Value to be Predicted
CG	B, C, D	Y
BT	B, C, D	Y
SP	B, C, D	Y

Table 6.8: Execution Times on Cluster B

Application	SET (Sec)	PAET (Sec)	AET (Sec)	PETE (%)
CG	15.82	1081.09	1085.32	0.38%
BT	86.96	806.17	814.42	1.01%
SP	70.85	1569.43	1572.84	0.21%

SET: Signature Execution Time
PAET: Predicted Application Execution Time
AET: Application Execution Time
PETE: Predicted Execution Time Error

6.4.3 Validation with Real Workloads (Scientific Applications)

To validate the proposal we extracted the parallel application signature not only from a synthetic application but also from three well known parallel applications. As stayed

before, an application input size affects the application performance in a scientific deterministic parallel application in different ways since the input is not usually a scalar. For example the Sweep3D application input can change both the phase execution time and phase weight as well. In the Sweep3D case, we treat both dimensions separated. The test case Sweep3D-n varies the Sweep3D input data size and the Sweep3D-i changes the applications iterations. Table 6.9 shows the applications, data ranges and unknown workload values to be predicted.

Table 6.9: Workloads values and unknown workload value to be predicted used in the experiments

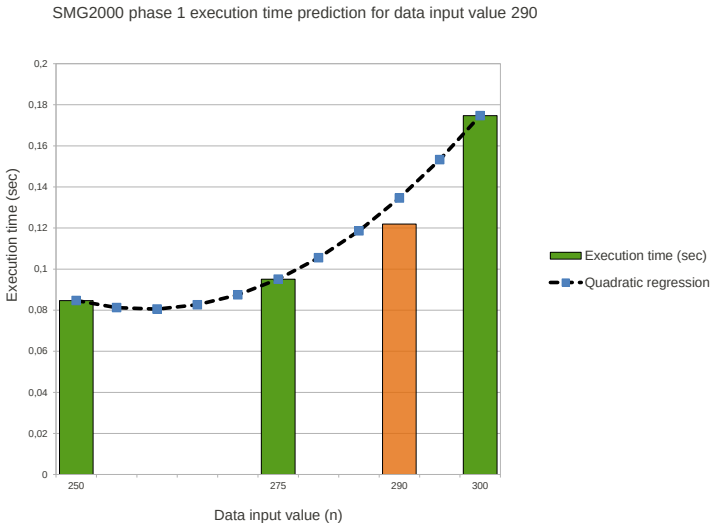
Application	Workload Values Used	Workload Value to be Predicted
SMG2000	250, 275, 300	290
Sweep3D-n	120, 140, 160	150
Sweep3D-i	8, 12, 16	10
POP	120, 140, 160	150

Figure 6.4 shows one phase execution time prediction for the Sweep3D-n and SMG2000 applications for an unknown workload value using regression analysis and interpolation techniques.

Table 6.10 shows for each application the results obtained using our methodology to predict an unknown workload value while Figure 6.5 shows the Predicted Application Execution Time (PAET) and Application Execution Time (AET) of each application.

We can see from the results that the Signature Execution Time (SET) is just a small fraction of the complete Application Execution Time (AET). Also the Predicted Application Execution Time (PAET) is very accurate with prediction errors that are below 4% and a worst error less 7%. For most application quadratic interpolation function were used. Only for the synthetic application a combination of linear and quadratic regression functions was used since it has two phases with different computational time complexity.

(a) Sweep3D-n



(b) SMG2000

Figure 6.4: Phase execution time prediction for Sweep3D-n and SMG2000

Table 6.10: Execution Times on Cluster B

Application	SET (Sec)	PAET (Sec)	AET (Sec)	PETE (%)	IF	NP
SMG2000	15.10	355.60	365.18	2.62%	Q	3
Sweep3D-n	2.85	979.86	1010.68	3.04%	Q	3
Sweep3D-i	2.15	1065.79	1058.91	6.87%	Q	3
POP	2.30	67.39	70.19	3.98%	Q	3

SET: Signature Execution Time
PAET: Predicted Application Execution Time
AET: Application Execution Time
PETE: Predicted Execution Time Error
IF: Interpolation function
NP: Number of points
L: Linear Q: Quadratic

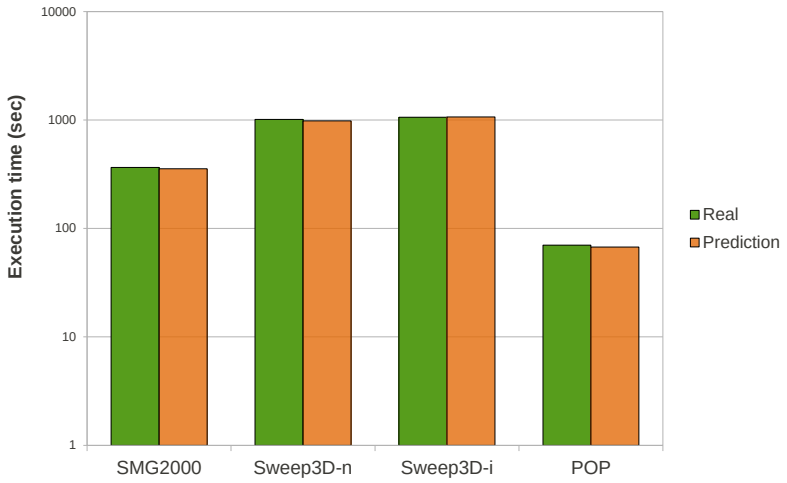


Figure 6.5: Applications execution time and predicted execution time

Chapter 7

Conclusions and Future Work

In this work we proposed a methodology to analyze and model the impact of an application workload in the parallel application signature. Extracting only a small set of signatures with different types of workloads and executing these signatures in a target machine allows to estimate each phase execution time and weight function. Because the signature contains the most relevant phases of an application, measuring each phase execution time and weight for different workloads allows to obtain the phases execution time function ($fexecution$) and weight function ($fweight$). Regression analysis was used to build the functions that model the phases behavior. Once the functions are obtained, an application performance can be predicted for any workload between a predefined range using interpolation methods.

Experiments were made using a synthetic application, applications from a benchmark suite used to compare different systems performance and well known real applications. The results obtained estimating the application's total execution time for these applications shows the effectiveness of the method.

7.1 Contributions

A methodology was presented [22] to model the impact that the variation of the workload has in the parallel application signature and use this information for performance prediction. A synthetic application was developed to analyze the variation in a program behavior caused by the workload variation.

A better algorithm to choose the workloads types was proposed [21]. It uses binary search to choose a workload type as a point to estimate the time complexity and weight function for each phase. Experimental validation of the methodology was made using real scientific applications.

7.2 Future Work

As future work we plan to continue extending our methodology to cover other dimensions that affect an application behavior such as its scalability. An open question is if we can model how an application will perform if we increase the number of processes and use more cores to make computation. Since the communication pattern is defined by the programmer and usually this pattern changes in a predictable way with the number of process used. It seems probable that we can mimic this communication pattern to add more process to the parallel application signature.

Another open line is to support parallel applications that communicate using shared memory and hybrid applications that use both message passing and shared memory. Finally the model can be extended for other parallel computation models that use specialized circuit such as the ones based in Graphics Processing Unit (GPU).

Bibliography

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[2] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[3] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, et al. Mambo: a full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[4] M. Calzarossa, L. Massari, and D. Tessera. Workload characterization issues and methodologies. *Performance Evaluation: Origins and Directions*, pages 459–482, 2000.

[5] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[6] J.W. Demmel and Inc Books24x7. *Applied numerical linear algebra*, volume 150. Society for Industrial and Applied Mathematics Philadelphia, 1997.

[7] Nasa Advanced Supercomputing (NAS) Division. Nas parallel benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html, 2011. [Online; accessed June-2011].

[8] K.M. Dixit. The spec benchmarks. *Parallel Computing*, 17(10-11):1195–1209, 1991.

[9] J.J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present

and future. *Concurrency and Computation Practice and Experience*, 15(9):803–820, 2003.

[10] Dror G. Feitelson. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 114–141. Springer Verlag, 2002.

[11] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. 2011.

[12] H. Gautama and A. van Gemund. Performance prediction of data-dependent task parallel programs. *Euro-Par 2001 Parallel Processing*, pages 106–116, 2001.

[13] S. Girona, J. Labarta, and R. Badia. Validation of dimemas communication model for mpi collective operations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 39–46, 2000.

[14] C.A.R. Hoare. Quicksort. *The Computer Journal*, 5(1):10, 1962.

[15] E. Ipek, B.R. De Supinski, M. Schulz, and S.A. McKee. An approach to performance prediction for parallel applications. *Euro-Par 2005 Parallel Processing*, pages 196–205, 2005.

[16] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In *Euro-Par'96 Parallel Processing*, pages 665–674. Springer, 1996.

[17] Los Alamos National Laboratory. The parallel ocean program (pop). http://climate.lanl.gov/Models/POP/, 2011. [Online; accessed June-2011].

[18] Los Alamos National Laboratory. Performance and architecture laboratory software. http://www.ccs3.lanl.gov/PAL/software.shtml, 2011. [Online; accessed June-2011].

[19] C. Lu and D.A. Reed. Compact application signatures for parallel and distributed scientific codes. In *Proc. of the 2002 ACM/IEEE conf. on Supercomputing*, pages 1–10. IEEE Computer Society Press, 2002.

[20] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Perform. Eval. Rev.*, 32:2–13, June 2004.

[21] J. Martinez Canillas, A. Wong, D. Rexachs, and E. Luque. Including the workload effect in the parallel program signature. In *High Performance Computing and Communications (HPCC), 2011 IEEE International Conference on*. IEEE, 2011.

[22] J. Martinez Canillas, A. Wong, D. Rexachs, and E. Luque. Predicting parallel applications performance using signatures: the workload effect. In *Computer Systems and Applications (AICCSA), 2011 IEEE/ACS International Conference on*. IEEE, 2011.

[23] Michael O. McCracken and Allan Snavely. A simulation toolkit to investigate the effects of grid characteristics on workflow completion time. pages 6:1–6:10, 2009.

[24] The National Nuclear Security Administration (NNSA). Purple archive. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/, 2011. [Online; accessed June-2011].

[25] C. Olschanowsky, M. Tikir, L. Carrington, and A. Snavely. PSnAP: Accurate Synthetic Address Streams Through Memory Profiles. *Languages and Compilers for Parallel Computing*, pages 353–367, 2010.

[26] E. Perelman, M. Polito, J.Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10. IEEE, 2006.

[27] G. Romanazzi and P. Jimack. Parallel performance prediction for multigrid codes on distributed memory architectures. *High Performance Computing and Communications*, pages 647–658, 2007.

[28] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 3–14. IEEE, 2002.

[29] S.S. Skiena. How to design algorithms. *The Algorithm Design Manual*, pages 32–40, 2008.

[30] Quinn O. Snell and John L. Gustafson. An analytical model of the hint performance metric. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[31] S. Sodhi, J. Subhlok, and Q. Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[32] A. Strube, D. Rexachs, and E. Luque. Software Probes: A Method for Quickly Characterizing Applications' Performance on Heterogeneous Environments. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 262–269. IEEE, 2009.

[33] A.O. Strube, D. Rexachs, and E. Luque. Software probes: Towards a quick method for machine characterization and application performance prediction. In *2008 International Symposium on Parallel and Distributed Computing*, pages 23–30. IEEE, 2008.

[34] A. Sykes. *An introduction to regression analysis*. Law School, University of Chicago, 1993.

[35] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PSINS: An open source event tracer and execution simulator for MPI applications. *Euro-Par 2009 Parallel Processing*, pages 135–148, 2009.

[36] A. Wong, D. Rexachs, and E. Luque. Parallel application signature. *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–4, 2009.

[37] A. Wong, D. Rexachs, and E. Luque. Extraction of Parallel Application Signatures for Performance Prediction. *High Performance Computing and Communications, 10th IEEE Int. Conf. on*, pages 223–230, 2010.

[38] A. Wong, D. Rexachs, and E. Luque. Parallel application signature for performance prediction. *In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010) CSREA Press.*, 2(408-414), 2010.

[39] L.T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. 2005.