Hybrid Method to Predict Execution Time of Parallel Applications

Bernard Miegemolle [1]
Thierry Monteil [1,2]
[1] LAAS - CNRS; 7 avenue du Colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse; INSA; 135 avenue de Rangueil, F-31077 Toulouse, France
E-mails: bmiegemo@laas.fr / monteil@laas.fr
Phone number: +335.61.33.69.18 / Fax number: +335.61.33.69.69
Contact author: Bernard Miegemolle

Conference: CSC'08

Abstract - *Many researches are carried out in the domain of the execution time prediction for sequential or parallel applications. This data can be used by job scheduling algorithms present on grid or cluster infrastructures to improve their behavior. In real-time context, the prediction of execution time is a crucial data, which the respect of deadline constraints may depend on.*

Both domains introduce their own prediction models. In parallel job scheduling, historic-based models can be used to estimate the execution time of a job using an experience base of past executions of similar jobs. In real-time domain, the Worst Case Execution Time (WCET) of applications is notably computed from the profile of the applications.

In this paper, an hybrid method for predicting execution time of parallel applications is presented. This method relies on both profile-based and historic-based predictions. Programs profiles are analyzed in order to decompose them into a set of basic blocks. The execution time of each block is determined using past executions of the programs. Then, a prediction of the overall execution time can be performed by applying historic-based predictions model to estimate the execution count of basic blocks.

Keywords: Performance prediction, Execution time, Program analysis, Historic model, Parallel computing

1 Introduction

Many researches are carried out in the domain of execution time prediction for sequential or parallel applications. The objective is to associate a program with its execution context, since its execution time may depend on the application parameters, the computer hardware and configuration, the operating system, and other programs that are on the same computer.

In the domain of parallel computing, the prediction of execution time is an interesting input for batch schedulers [1][2][3]. In the AROMA scheduler [4], it will be used to satisfy QoS constraints. The good performance of the mapping algorithm is directly related to the accuracy of predicted values. An overestimated value will delay the execution of the application and an underestimated value will create an interruption of the application and perhaps no results.

In real-time context, the prediction of execution time is very important to validate the good behavior of the whole system. One solution, to validate the real-time constraints and so the deadline of the scheduled tasks, is to work on the Worst Case Execution Time (WCET)[5].

It is necessary to define an easy way to obtain a good estimate of the future execution time of an application. Some methods will be automated and some others will need assistance from the users. Two main techniques exist:

- predictions based on an historic of past executions [6][3],
- predictions based on the study of the program[7]. This is kind of techniques is used in the real-time domain to calculate the WCET.

The objective of this paper is to propose a model for execution time prediction that uses both techniques. This hybrid method relies on the study of the program structure in order to decompose it into a set of atomic parts named basic blocks. An historic of past executions is then used to estimate the execution time of the whole application.

1.1 Related Work and Contributions

Two main methods are used to evaluate the WCET of an application. The first one is dynamic and the other one is static. In the first case, the application is executed on the real architecture of processors, or simulated on a software model of the architecture, and its execution time is determined by measurements [8][9]. Real inputs are used. The main problem is to find a good set of inputs that leads to the maximal execution time. The main methods are used to find a good set of inputs:

- testing all possible inputs,
- taking a set of inputs given by the user, since he is supposed to have a good knowledge of his program and so of data that may maximize the execution time,
- creating some heuristics to find a good input with genetic algorithms for example,
- considering inputs as unknown values and solve formal equations.

In the static case, the structure and the source of the program are studied, in three steps methods:

- the flow analysis enables to decompose the program into atomic parts, and then all possible paths through the program are explored,
- the study of the low level expresses the time needed to execute atomic parts of the program on specific architecture,
- the value of the WCET is calculated with the two previous information.

Some restrictions are necessary to use these kinds of methods: no dynamic allocation structure, no recursion and no unbounded loop. Those conditions enable to bound the number of paths to explore. However, an increasingly number of paths is still a difficulty for applying these methods.

A commonly-used technique of performance prediction for serial or parallel applications in non-real-time domains is based on the study of past executions. It suppose that executions of similar applications in similar context lead to similar execution times [6][10][2] [3].

In a first approach, a classification of the applications can be done [6]. In this case, a set of templates is used to identify the different types of applications. These templates contain information such as the application type (batch or interactive, parallel or serial), the submission queue, the user, the binary name, the parameters, the number of computing nodes, etc. The main difficulty is to define a good set of templates that leads to a correct number of categories to classify the jobs into. The number of categories must be sufficient to group only related jobs together, but not too high to make accurate predictions. In [10], the authors propose algorithms that automatically determine a set of appropriate templates.

Another approach uses instance-based learning techniques, also called locally weighted learning techniques [11]. The principle is to memorize all executions with all inputs and outputs. When a prediction must be calculated with specific inputs, one or several instances with similar inputs are searched in the set of past executions. Their outputs and resources utilization are used to estimate the execution time. To find the similar experiments, a distance is calculated. The Euclidean distance is used in this article as in [2] [3]. A vector E is defined to represent all inputs of the program (N^V values):

$$E = \{E_v\}_{v\in[1;N^V]} = [E_1, E_2, \dots, E_{N^V}]$$

The Euclidean distance function is:

$$D\left(E^{(1)}, E^{(2)}\right) = \sqrt{\sum_{v=1}^{N^V} d\left(E_v^{(1)}, E_v^{(2)}\right)^2} \qquad (1)$$

The distance d between two input values depends on the type of the values. Indeed, they can be linear or nominal (a nominal attribute is a discrete attribute which values are not necessarily in any linear order, such as colors). Linear inputs can also be continuous or discrete. An heterogeneous distance function is used [12]:

$$d\left(E_v^{(1)}, E_v^{(2)}\right) = \begin{cases} 1 \text{ if } E_v^{(1)} \text{ or } E_v^{(2)} \text{ is unknown,} \\ N(E_v^{(1)}, E_v^{(2)}) \text{ if } E_v^{(x)} \text{ is nominal,} \\ L(E_v^{(1)}, E_v^{(2)}) \text{ if } E_v^{(x)} \text{ is linear.} \end{cases} \qquad (2)$$

where:

$$N(E_v^{(1)}, E_v^{(2)}) = \begin{cases} 0 & \text{if } E_v^{(1)} = E_v^{(2)}, \\ 1 & \text{in the other cases.} \end{cases},$$

$$L(E_v^{(1)}, E_v^{(2)}) = \frac{|E_v^{(1)} - E_v^{(2)}|}{max_{E_v^{(x)}} - min_{E_v^{(x)}}}$$

The estimation of a future execution time ($T(E^*)$) for a specific input E^* is performed by merging several past experiments with different methods: model of the k-nearest neighbor [13], locally weighted polynomial regression [14], or weighted average [3][2]. This last model is used in this paper, and relies on the following equation:

$$T(E^*) = \frac{\sum_E \left[K\Big(D\left(E^*, E\right)\Big) \cdot T(E)\right]}{\sum_E K\Big(D\left(E^*, E\right)\Big)} \quad (3)$$

K expresses the proximity of the experiments. It can be defined by numerous weighting functions [11]. In this paper, the Gaussian function is chosen: $K(d) = e^{-\left(\frac{d}{k}\right)^2}$

Our contribution is to merge both profile-based techniques and instance-based learning ones to produce execution time predictions. Even if our work is notably related to WCET calculation methods, the objective is not WCET prediction but the estimation of a program running time for specified inputs, in order to provide scheduling algorithms with information about submitted applications to schedule. Using our prediction technique is quite easy to set up since it only needs the use of standard GNU profiling tools and of a simple database.

2 Definition of the Prediction Model

The hybrid model of prediction proposed in this article uses the historical approach based on locally weighted learning techniques, combined with a flow analysis of the program. Annotations inside the program source can eventually be used to improve the prediction accuracy.

The general functioning of our hybrid prediction model is given by figure 1. An experience base is progressively built using different executions of the program. Our prediction method is based on the use of standard GNU profiling tools: *gprof* and *gcov.* The program is decomposed into several basic blocks, using the *gcov* tool. The unit execution time of each block (T^{BB}) is computed using the different past executions, and is supposed to be constant from one execution to another. The execution count of each block (N^{BB}) depends on the program inputs. In the database, each input set is mapped to the number of blocks executions. When a prediction has to be performed, this number is estimated using the different entries in the database, weighted by the distance between the entries and the query. Then, the total execution time can be calculated from the estimated N^{BB} and the computed T^{BB}.

2.1 Block Decomposition

A block is a maximal sequence of instructions which has one and only one entry and exit point. It contains only simple instructions, without any branching or function call. The execution time of a block is supposed to be independant from the inputs of the program, and thus to be constant on a same architecture. This is a simplification since modern processors often propose optimization mechanisms, such as caches or pipelines, that could produce different execution times for a same block.

In computing programs, the main blocks are always executed many times. Thus the cache, for example, introduces different times only for the first execution which is neglected because the block will be executed many times after. A function f (with f being an element of the set F of all program functions) is composed by different blocks b with different execution times T_b^{BB}. The number of executions $N_b^{BB}(E)$ of the block b depends on the input E of the program . So the total execution time $T_{Exec}(E)$ of a program is:

$$T_{Exec}(E) = \sum_{f \in F} \left[N_f^F(E) \cdot \sum_{b \in BB_f^F} N_b^{BB}(E) \cdot T_b^{BB} \right] \quad (4)$$

where:

- $N_f^F(E)$ is the number of times that the function f is executed,
- BB_f^F is the set of blocks that composes the function f.

2.2 Past Execution Study

The first step is to compute the T_b^{BB} values. This is done by using the different experiences (previous runs) contained in the database. When a new execution of the program is performed, the new experience obtained enables to compute new T_b^{BB} values in order to improve their precision.

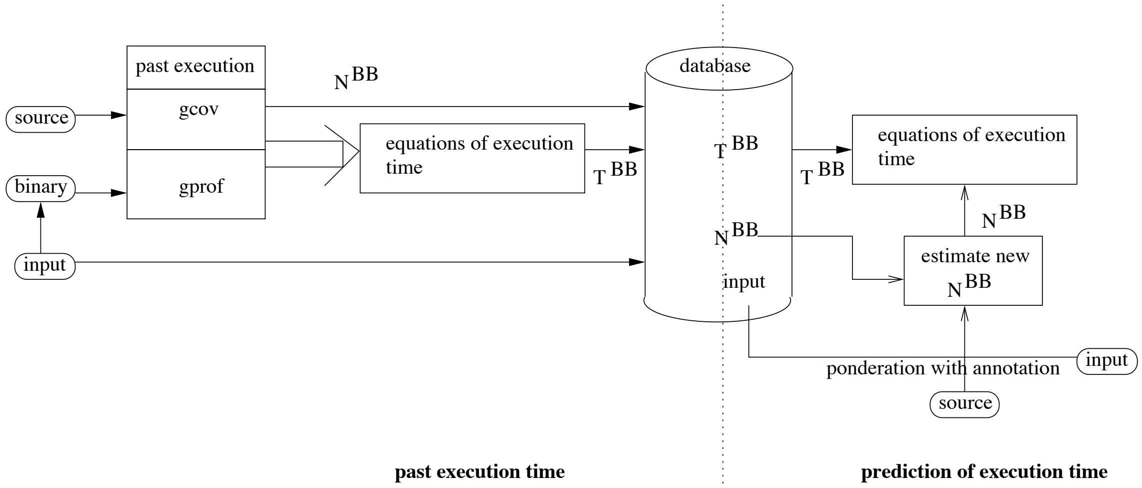


Figure 1: Principle of the hybrid prediction

The execution time of the basic blocks are supposed to be constant for each block, whatever the inputs of the program are. Thus, considering the previous runs of the program, we obtain a set of equations (one equation per run with its own inputs E):

$$T_f^F(E) = \sum_{b \in BB_f^F} N_b^{BB}(E) \cdot T_b^{BB} \qquad (5)$$

where $T_f^F(E)$ is the execution time of the function f depending on the inputs E of the program.

In this equation, for each experience, the $T_f^F(E)$ value is given by the *gprof* profiler, and the $N_b^{BB}(E)$ values are obtained thanks to *gcov*. These data vary from one run to another depending on the inputs of the program. The T_b^{BB}, which are unknown, do not depend on the inputs.

Computing the execution time of the program blocks is thus equivalent to solving a linear equation system, where T_b^{BB} are the unknowns, under the constraint of getting positive results. The number of equation systems to solve corresponds to the number of functions of the program.

However, the equation systems obtained are often ill-conditioned, and their resolution with standard methods are not possible. We have designed for this purpose an iterative resolution, which is adapted to the problem.

2.3 Prediction of Execution Time

The goal of the second step is to give an estimation of the $N_b^{BB}(E)$ values, depending on the inputs of the program. If we consider the equation (5), the T_b^{BB} values are known from previous step. An estimation of the execution count $N_b^{BB}(E)$ of the basic blocks has to be performed in order to predict the execution time of the program functions, and thus its global execution time.

This estimation will be done using the experiences stored in the database. Each one maps the inputs of the program with the corresponding execution count for each basic block. The idea is to use the historic-based model given by equation (3) to estimate the execution count of each basic block for the inputs of the query.

A distance-weighted average is thus used to form an estimate of the execution count of the program basic blocks. We use the Gaussian function to give more importance to the experiences which inputs are closed to the ones of the query. The k value enables to adjust the Gaussian function width, in order to fit both cases when there are many experiences in the database or not.

The distance function is defined as in equation (1). Nevertheless, this definition can be improved by adding a coefficient that enables to favor some inputs over others in the calculation of the basic blocks counts. Indeed, not all blocks depend on all inputs, and it can be useful to indicate which inputs are relevant to compute the count of each block. So, the distance becomes:

$$D\left(E^{(1)},E^{(2)}\right)=\sqrt{\sum_{v=1}^{NV} w_v\cdot d\left(E_v^{(1)},E_v^{(2)}\right)^2} \tag{6}$$

The w_v parameters indicate the importance of each input feature on the distance, and so on the computation of the block execution count estimate. Their values depend on the structure of the program, and can be given by the user through source code annotations. These annotations will indicate which inputs are relevant for the prediction of execution count of each basic block of the program. An example of annotation is given below:

```
#pragma etp dependency(size)
for ( int i = 0 ; i < size ; i++ ) {
     vector[i] = 0;
}
#pragma etp end
```

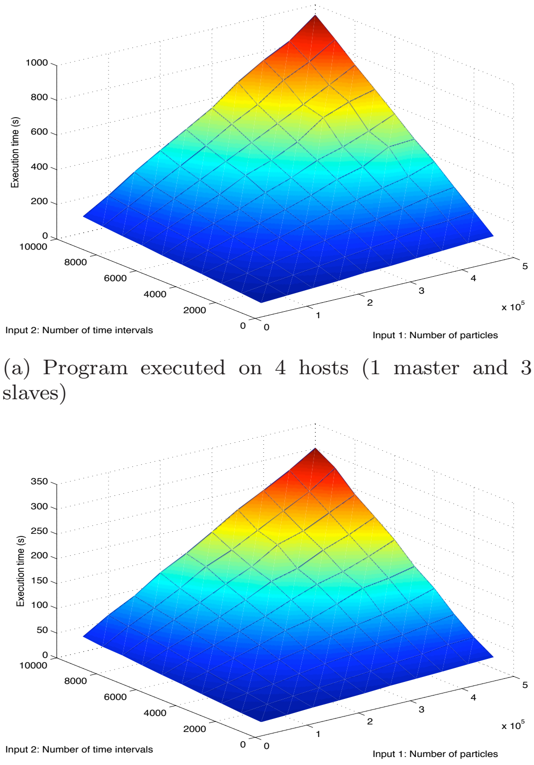
We have implemented a tool that deduces such information from annotations given by the user in C programs through `#pragma` directives. These directives are ignored by compilers such as GCC, and are analysed by our tool to get the w_v weighting parameters. Each basic block between the `etp dependency` directive and the `etp end` one (`etp` stands for "execution time prediction") depends on the specified inputs (`size` attribute in the example above).

3 Implementation and Performances

In order to experiment our hybrid prediction method, we use an MPI application which performs particle filtering [15]. This parallel application is designed using a master/slave model. The master task role is to synchronize the slaves, and to collect the information they have computed. The particles are equally distributed over the slaves, so that each slave is responsible of the evolution of a set of particles over the time.

Different runs of the application point out that the execution time of the master task is always less than 0.5 second (in term of CPU time), and can be neglected in comparison with the execution time of slave tasks. The CPU time of the slaves mainly depends on two parameters: the number of particles and the number of time intervals considered (which is equivalent to the number of iterations of the particle filtering algorithm). For a given number of particles, the number of computational nodes (and thus of slaves) has an impact on the execution time, since the particles are equally shared out between the different slaves. The slaves are identical tasks, so each of them has the same CPU time.

The figure 2 shows the evolution of the execution time of slave tasks in function of the program inputs (the number of time intervals and the global number of particles for the whole program). We can notice that the number of particles and of time intervals have the same effect on CPU time, regardless of the number of tasks. However, if the number of slave tasks is multiplied by 3, the number of particles per slave is divided by 3, and so is the execution time of each task. Thus, the number of tasks of the application is also considered as an input.



(a) Program executed on 4 hosts (1 master and 3 slaves)

(b) Program executed on 10 hosts (1 master and 9 slaves)

Figure 2: Execution time of a slave task for different input values

The first step of the work is to create an experience base. In production environments, the base would be continuously filled with each executed job. By consequence, jobs have two successive states: first, they are considered as queries, in order to get a prediction of their execution time; then, they are executed, and profiled at the same time by *gprof* and *gcov* in order to become a new experience that will be stored into the database.

In this work, those two phases are completely separated. First, the experience base is created and filled with a set of experiences. Then, time predictions are performed for another set of random runs, and compared with measured times.

The goal of the learning phase is to determine the execution time of each basic block, and also to store into the database associations between inputs and execution counts of the different basic blocks. The application is run several times, with different input sets, i.e. with different numbers of computational nodes used (and so different numbers of slave tasks), and also with different number of particles and time intervals considered. The inputs are chosen in order to be the most representative as possible by covering the whole input space. The lowest number of runs is determined to be sufficient to solve the linear equation system (one run corresponds to an equation), and thus depends on the program structure, more precisely on the number of basic blocks per function of the program.

The next step is to compute a solution for each equation system obtained. The unknowns determined here correspond to the execution time of each basic block of the program. The systems are ill-conditioned, and their resolution requires non-standard methods. We have designed an algorithm of resolution which solve the system by successive iterations. At each iteration, a well-conditioned system is solved. An artificial error is introduced in the system in order to have a convergence towards a potential solution since the ill-conditioning of the system leads to a non-convergence.

At this point, the experience base contains all that is required to make predictions of execution times. In order to validation the hybrid approach introduced in this paper, several runs of the application for different inputs will be studied. For each of them, the execution time will be predicted using our approach, and then it will be measured. The inputs are chosen to cover the entire input space in terms of number of tasks, particles and time intervals. Most inputs used in these runs differ from the ones used during the learning phase. Nevertheless, some inputs are used in both kinds of runs because this case can actually be encountered.

The figure 3 shows the difference between predicted and measured execution times for different inputs. We can observe that these two values are very closed. The difference between them varies from 0.4% to 17,6%, with a mean value which is equal to 5,3%. The error between predicted and measured values is directly related to the distance between the query and the closest experiences stored in the database. Such results are quiete acceptable, since this approach is not designed to be used in real-time context.

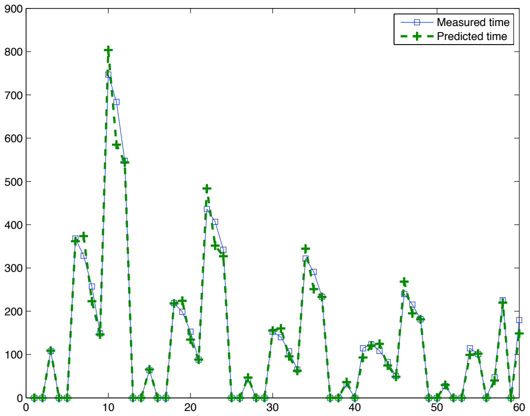


Figure 3: Predicted and measured execution times for different inputs

4 Conclusion and Future Directions

In this paper, an hybrid method for execution time prediction has been presented. It combines profile-based prediction techniques with historic-based ones, and mainly aims to provides scheduling algorithms with information about the submitted applications to schedule. In this approach, the application is divided into a set of basic blocks, for which execution time is considered constant and independant from the inputs. Nevertheless, their execution count depends on the inputs, and it will have a significant impact over the application execution time.

The first step of the hybrid method is to compute the execution time of each basic block. This is done by solving linear equation systems obtained with previous executions of the application profiled by the *gprof* and *gcov* tools for different inputs. Then, the goal of the second step is to estimate the execution count of each block using locally weighted learning techniques. Annotations of the source code can be used to improve the accuracy of the prediction.

Our hybrid method for execution time prediction has been tested over an MPI parallel application, which performs particle filtering.

Good results are obtained since a comparison between predicted execution time and measured execution time shows that these two values are relatively closed: a mean difference of 5.3% has been observed between them for the experiences that have been carried out in this paper.

In a next future, the accuracy of the results can be improved by taking in account optimization mechanisms of modern processors, such as caches and pipelines. We also think about analysing object code instead of source code to determine the basic blocks, which will result in a more accurate determination of the basic blocks, corresponding exactly to what the processor executes. Finally, other distance functions, such as Minkowsky distance or Manhattan one, and other weighting functions will be tried in order to evaluate their impact on the estimations.

References

[1] W. Smith, V. Taylor, et I. Foster. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance.* Job Scheduling Strategies for Parallel Processing, Springer Verlag, 1999.

[2] L. J. Senger, M. J. Santana et R. H. Carlucci Santana. *An Instance-Based Learning Approach for Predicting Execution Times of Parallel Applications* Third International Information and Telecommunication Technologies Symposium, 2004.

[3] W. Smith et P. Wong. *Resource Selection Using Execution and Queue Wait Time Predictions.* Technical Report, NASA Advanced Supercomputing Division (NAS), Moffet Field (USA), 2002.

[4] P. Pascal, S. Richard, B. Miegemolle, and T. Monteil. *Tasks Mapping with Quality of Service for Coarse Grain Applications.* 11th International Euro-Par Conference (Euro-Par 2005), Lisbon, Portugal, Aug. 30 - Sept. 2 2005, Lecture Notes in Computer Science 3648 Parallel Processing, Springer, 2005.

[5] G. C. Buttazzo. *Hard-Real Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, Second Edition, Oct. 2004.

[6] R. Gibbons. *A Historical Application Profiler for Use by Parallel Schedulers.* Job Scheduling Strategies for Parallel Processing, Springer Verlag, 1997.

[7] Y.-T. S. Li and S. Malik. *Performance Analysis of Embedded Software Using Implicit Path Enumeration.* In Proceedings of the 32nd ACM/IEEE Design Automation Conference, 1995.

[8] J.-F. Deverge and I. Puaut. *Safe measurement-based WCET estimation.* 5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Palma de Mallorca (Spain), 2005,

[9] P. Atanassov and P. Puschner. *Dynamic Worst-Case Execution Time Analysis - A Comparison Between Two Target Platforms.* Technical Research Report, Technische Universität Wien, Institut für Technische Informatik, Austria, 2000

[10] W. Smith, I. Foster et V. Taylor. *Predicting Application Run Times Using Historical Information* Lecture Notes in Computer Science, 1998.

[11] C. Atkeson, A. Moore and S. Schaal. *Locally Weighted Learning.* Artificial Intelligence Review, Vol. 11, 1997.

[12] D. R. Wilson and T. R. Martinez. *Improved Heterogeneous Distance Functions.* Journal of Artificial Intelligence Research, Vol. 6, 1997.

[13] M. Iverson, F. Ozguner and L. Potter. *Statistical Prediction of Task Execution Times Through Analytical Benchmarking for Scheduling in a Heterogeneous Environment.* In Proceedings of the IPPS/SPDP'99 Heterogeneous Computing Workshop, 1999.

[14] N. Kapadia, J. Fortes and C. Brodley. *Predictive Application Performance Modeling in a Computational Grid Environment.* In Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

[15] S. Arulampalam, S. Maskell, N. J. Gordon and T. Clapp. *A Tutorial on Particle Filters for On-line Non-linear/Non-Gaussian Bayesian Tracking.* IEEE Transactions of Signal Processing, Vol. 50(2), February 2002.