1993

An Interpretive Framework for Application Performance Prediction

Manish Parashar
Syracuse University

Salim Hariri
Syracuse University

Tomasz Haupt
Syracuse University

Geoffrey C. Fox
Syracuse University

Recommended Citation

Parashar, Manish; Hariri, Salim; Haupt, Tomasz; and Fox, Geoffrey C., "An Interpretive Framework for Application Performance Prediction" (1993). *Northeast Parallel Architecture Center*. 57.
https://surface.syr.edu/npac/57

An Interpretive Framework for Application Performance Prediction [*]

Manish Parashar, Salim Hariri, Tomasz Haupt and Geoffrey C. Fox
Northeast Parallel Architectures Center
Syracuse University

Presented at
International Conference on Parallel & Distributed Systems (ICPADS), 1993

Abstract

Software development in parallel/distributed environment is a non-trivial task and depends greatly on the availability of appropriate support in terms of development tools and environments. Performance prediction/evaluation tools form a critical part of any software development environment as they enable the developer to visualize the effects of various design choices on the performance of the application. This paper presents an interpretive model for a source driven performance prediction framework. A prototype framework based on the proposed model has been developed for the iPSC/860 system. Numerical results obtained on this system are presented. These results confirm the potential of interpretive performance prediction techniques and their applicability.

Keywords: *Performance prediction, Performance interpretation, Parallel/Distributed software development, System & Application characterization.*

1 Introduction

Software development in any Parallel/Distributed computing environment is a non-trivial task and requires a thorough understanding of the application and the system architecture. This is apparent from the fact that currently, applications are able to achieve only a fraction of peak available performance [1]. During the course of parallel/distributed software development, the developer is required to select the optimal hardware configuration for the particular application, the best decomposition of the problem on the selected hardware configuration, the best communication and synchronization strategy to be used, etc. The set of reasonable alternatives that have to be evaluated is very large. Selecting the best alternative among these is a formidable task and depends greatly on the availability of appropriate development support. It is imperative therefore, that evaluations tools be provided as part of any parallel/distributed software development environment, to enable the developer to visualize the effects of various design choices on the performance of the application.

Conventional evaluation/prediction tools and techniques are either tuned to specific systems, or are too general and lack feasibility and accuracy needed in parallel/distributed software development. Analytic models for parallel/distributed systems [2, 3] lead to large state spaces which result in impractical evaluation times. These techniques can be made tractable by introducing simplistic assumptions, but this makes them unrealistic and inaccurate. Monitoring techniques [4, 5, 6], on the other hand, require extensive experimentation and data collection on the actual hardware. The process is not feasible or cost-effective since parallel/distributed systems are expensive resources and usually not freely available for such experimentation. Furthermore, these techniques are intrusive and can alter the execution of the application. A detailed survey of existing evaluation tools and techniques can be found in [7].

In this paper, we present the design of a practical performance prediction framework targeted to parallel/distributed computing environments. The framework uses a novel interpretive approach to provide accurate and cost-effective performance prediction. A comprehensive characterization methodology is proposed to abstract the system and application components of the computing environment into a set of well defined parameters. An interpreter engine then interprets the performance of the abstracted application in terms of the parameters exported by the abstracted system. The parameters required to abstract a system component can

---

[*] The presented research has been jointly sponsored by DARPA under contract #DABT63-91-k-0005 and by Rome Labs under contract #F30602-92-C-0150. The content of the information does not necessary reflect the position or the policy of the sponsors and no official endorsement should be inferred.

of the tuple can be Compound, Simple, or Void. A compound component can be further decomposed into one or more levels in the hierarchy. A simple component represents the lowest level in the classification hierarchy and exports actual timing information required to abstract that component. A void component implies that the particular component is not applicable at that level. An SAU is considered compound if at least one of its components is compound. Further, every SAU has at least one component that is not void. Every leaf SAU in a SAG is simple.

2.3 Application Module

The application module is responsible for abstracting the application description into a set of parameters which define its structure and performance. These parameter are then exported to the interpreter engine so that their performance can be interpreted in terms of the parameters exported by the systems module. The application module is composed of two components: (1) Machine Independent Abstraction Module and (2) Machine Specific Filter. The machine independent abstraction module is responsible for characterizing the application into an abstraction graph according to application abstraction model defined below. This graph is then passed through the machine specific filter where it is augmented to incorporate machine specific information based on the mapping inputs provided. The application module is designed to be general enough to handle any structured application description.

2.3.1 Application Abstraction Model

The application abstraction model recursively characterizes an application description into *Application Abstraction Units (AAU's)*. AAU's represent the fundamental unit of abstraction of the application description. An AAU can be of two types: A *Simple* AAU that cannot be further decomposed. It exports a set of parameters which abstract the portion of the application description associated with it. A *Compound* AAU can be recursively decomposed into a set of simple or compound AAU's. Various classes of simple and compound AAU's are used to represent standard programming constructs. The AAU's are combined so as to abstract the control structure of the application, to form the *Application Abstraction Graph (AAG)*. The communication/synchronization structure of the application is superimposed onto the AAG by augmenting the graph with a set of edges corresponding to the communications or synchronizations between AAU's. The structure generated after augmentation is called the *Synchronized Application Abstraction Graph (SAAG)*.

The SAAG is then passed through the machine dependent filter which uses the input mapping information to define a *Mapping Abstraction Function (μ)* from the SAAG to the SAG so as to assign: (1) Every $AAU_i \in SAAG$ to an $SAU_j \in SAG$ on which it is to be interpreted; and (2) Every communication or synchronization edge in SAAG to an ordered set $\{\mathbf{sau}\}$ which represents the actual route followed by the particular communication/synchronization for the specified mapping (e.g a communication from an external unit to a hypercube node has to be routed through the SRM).

2.4 Interpretive Engine

The interpreter engine is responsible for the actual performance interpretation. It uses the system, application and mapping abstractions, to predict the performance of the application. The iterpretation model and algorithm are described below.

2.4.1 Interpretation Model

The interpretation model consists of two components: (1) An *Interpretation Function (Ω)* that interprets the performance of an individual AAU and (2) An interpretation algorithm which recursively applies the interpretation function to the SAAG to predict the performance of the corresponding application.

Definition 1 *The* **Interpretation Function** (Ω) *operates from the set* { **AAG** } *to the set* $\Re$ *of real numbers and assigns to each* $AAU \in SAAG$ *a subset of* $\Re$ *which represents the performance statistics of that AAU. i.e.*

$$\Omega(AAU_i, SAG, \mu) : AAU_i \mapsto \{\mathbf{R}\};$$

where $AAU_i \in \{\mathbf{AAU}\}$ *&* $\{\mathbf{R}\} \subset \Re$
& where μ *is the mapping abstraction function.*

Before we state the interpretation algorithm, the following terminology needs to be defined: A *Chain* of a SAAG is defined as a set of contiguous AAU's in that SAAG. The first AAU is called the *Head* of the chain. *Evaluating* an AAU consists of applying the interpretation function Ω to the AAU to obtain its performance statistics. A *Red* AAU denotes an AAU that has been evaluated. An *Active* AAU is an AAU whose immediate predecessors are red AAU's. An *Active Chain* is a chain whose head is active.

Let τ_{AAU_i} denote the time (measured from the start of the application) at which AAU_i became active. τ_{Start} represents the beginning of the application execution and the reference point for all measurements made by the interpretation model. Let δ_{AAU_i} denote the execution time of AAU_i returned by the interpretation function Ω. The interpretation algorithm can now be defined as follows:

Algorithm 1 *Interpret*

```
(1) color Start AAU red;
$\tau_{Start} \leftarrow 0$;
$\delta_{Start} \leftarrow \Omega(AAU_{Start}, SAG, \mu)$ [evaluate $AAU_{Start}$]
(2) for each active AAU
repeat until there is no active AAU
```

```
        • for each AAU ($AAU_i$) in the associated active chain
        repeat until $AAU_i$ cannot be evaluated due to synchronization requirements
            $\tau_{AAU_i} \leftarrow \tau_{AAU_{i-1}} + \delta_{i-1}$
            $\delta_i \leftarrow \Omega(AAU_i, SAG, \mu)$ [evaluate $AAU_i$]
            color $AAU_i$ red
        end  repeat
    end  repeat (3) if all leaf AAU's of the SAAG are not red
        ERROR
    end  if

End
```

The above algorithm proceeds down each active chain (i.e. depth first) in the SAAG, and evaluates each AAU of the active chain. It also updates a global time base (τ) as it proceeds. If the current AAU cannot be evaluated because it has to wait for synchronization, that AAU remains active and the algorithm moves to the next active chain. If at the end of the algorithm, the leaf AAU's of the (topmost) SAAG are not red, an error has occurred in the implementation of the application and has caused it to hang-up.

2.5 Output Module

The output module provides an interactive interface through which the user can access the interpreted performance statistics. The user has the option of selecting the type of information, the level at which the information is to be displayed. Performance statistics can be obtained at the following levels:

AAG Level: Performance information at the AAG level deals with the entire application. Statistics available at this level include cumulative execution times, the communication time/computation time breakup and the existing idle times.

Sub-AAG Level: Performance information at this level deals with the specified part of the AAG. Cumulative statistics for the specified subgraph are displayed.

AAU Level: Performance information at this level is specific to a particular AAU. All statistics relevant to that AAU are displayed.

Visualization software can be interfaced to this module to provide graphical displays of the available information. Animation capabilities can also be incorporated.

3 Numerical Results

This section presents some preliminary numerical results obtained through experimentation on a prototype performance prediction framework. The objective of this experiment was threefold: (1) To validate the system and application abstraction model and to demonstrate their feasibility and applicability. (2) To validate the performance interpretation model proposed. (3) To demonstrate the cost-effectiveness of the approach in terms of both, resources required and time taken.

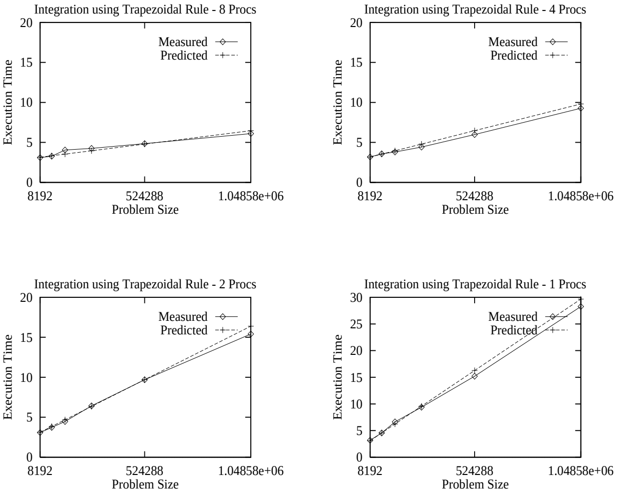

Figure 2: Comparison of Predicted and Measured Times (sec)) - Application I: Integration using Trapezoidal Rule

To meet the first objective we chose an architecture which is widely used to solve scientific and engineering applications. The computing system used consisted of an iPSC/860 hypercube connected to a 80386 based host processor. The applications chosen are a part of a standard benchmark set (The Purdue Benchmark Set [8]) and were written using FORTRAN 77 and the NX/2 communication library. The implementations were tweaked to incorporate a wide range of programming constructs.

Application I The first application evaluates the integral, T_N, of $f(x)$ using the trapezoidal rule.

$$T_N = h * (f(a)/2 + \sum_{i=1}^{N-1} f(a+ih) + f(b)/2)$$

The implementation uses the host-node programming model wherein the host program allocates the node processors and loads the node programs. It then uses cyclic distribution to distribute the integration domain among the nodes and broadcasts integration parameters. The host program receives the integral from the nodes after completion. The node processors calculate the integral over their domains and then perform a global sum. Node zero then sends the results to the host. The above procedure is repeated multiple times in a loop. The number of intervals into which the integration domain was divided was an external input. The number iteration were provided as external inputs.

Application II The second application evaluates e^* as follows:

$$e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} \left(1.0 + e^{(-\|i-j\|)}\right)$$

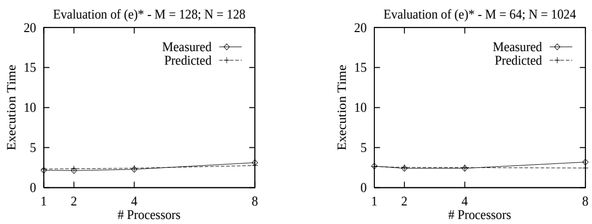

Figure 3: Comparison of Predicted and Measured Times (sec)) - Application II: Evaluation of e^*

The implementation of this application also uses a host-node programming model. The host processor broadcasts limits m & n to the nodes. A cyclic distribution is used to to distribute the summation domain across the nodes. Each node computes its partial sum. The global sum is then computed using a global reduction operator. Node zero then sends the results to the host. The above procedure is repeated multiple times in a loop. The limits are provided to the host as external inputs. Experimentation with the two applications applications consisted of varying two variables: (1) the external inputs and (2) the number of processing nodes used. The time on the host from the instant the node program was loaded till the final result was received, was measured.

The experimentation was performed in two phases: *Phase I* consisted of implementing the application and then running it for each combination of problem size and number of processors. The implementation was instrumented to measure execution times. Multiple runs were made for each case to account for noise in the measured timings. *Phase II* consisted of abstracting the application and feeding it to the performance prediction prototype. Then, using the interactive interpreter engine, prediction were obtained for all the desired combinations. A comparison between the measured and predicted times (in seconds) are plotted in Figure 2. The results obtain show that the predicted values lie within 15% of the measured results. This meets our second objective.

The cost-effectiveness of the interpretive approach is obvious from the fact the entire experiment was completed in a single run and on a Sun workstation.

4 Summary & Concluding Remarks

Evaluation tools form a critical part of any software development environment and enable the developer to visualize the effects of various design choices on the performance of the application.

In this paper we presented the design of a performance prediction framework which uses a novel interpretive approach to to provide accurate and cost-effective performance prediction. A comprehensive system abstraction model was defined which provides a methodology to characterize any parallel/distributed computing environment. A corresponding application abstraction model was also defined to characterize the structure of structured applications. Finally an interpretation model was defined which used the system and application abstraction to achieve performance interpretation. The interpreted performance of a standard parallel benchmark on a widely used distributed memory architecture was compared to the measured performance. The results obtained not only validated the accuracy and feasibility of the abstraction, and interpretation model, but also demonstrated its cost-effectiveness, both in terms of resources required and time taken.

References

[1] Glenn Zorpette, "Teraflops Galore", *IEEE Spectrum*, **29**(9):26–76, Sep. 1992.

[2] Horace P. Flatt and Ken Kennedy, "Performance of Parallel Processors", *Parallel Computing*, **12**(1):1–20, Oct. 1989.

[3] Reda A. Ammar and Bin Qin, "A Technique to Derive the Detailed Time Costs of Parallel Computations", *Proceedings of the* 12^{th} *Annual International Computer Software and Application Conference*, pp. 113–119, 1988.

[4] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, **1**(2):206–217, Apr. 1990.

[5] Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, and Roland Wismüller, *The Design and Implementation of TOPSYS*, Technische Universität München, Institut Für Informatik, July 1991, Ver 1.0.

[6] Markus Siegle and Richard Hofmann, "Monitoring Program Behavior on SUPRENUM", *ACM SIGARCH, Proceedings of the* 19^{th} *International Symposium on Computer Architecture*, pp. 332–341, May 1992.

[7] Manish Parashar, Salim Hariri, Tomasz Haupt, and Geoffrey C. Fox, "An Interpretive Framework for Application Performance Prediction", Technical Report SCCS-479, Northeast Parallel Architectures Center, Syracuse University, Syracuse NY 13244-4100, Apr. 1993.

[8] J. R. Rice and J. Jing, "Problems to Test Parallel and Vector Languages", Technical Report CSD-TR-1016, Computer Science Department, Purdue University, Purdue University, Dec. 1990.