An Agent-based Architecture for Tuning Parallel and Distributed Applications Performance

Sherif A. Elfayoumy and James H. Graham

Computer Science and Computer Engineering
University of Louisville
Louisville, Kentucky 40292, USA

ABSTRACT

This paper introduces a new software system model for improving the performance of parallel and distributed applications adaptively and on a real-time base. This model uses a performance agent as an adaptive controller. The performance agent has a multi-layer architecture and employs several intelligent agents to suggest and actuate modifications to the performance parameters. These intelligent agents are responsible for collecting and analyzing the performance parameters and metrics, and deciding on the required modifications. The system model has a distributed decision nature to increase the reliability and balance the overheads. An experiment was designed and implemented to test this system model, and it showed promising results with initial testing on a parallel system.

Keywords: performance tuning, distributed systems, parallel programming, and intelligent agents.

1 INTRODUCTION

Most scientific and engineering applications are resource intensive with steadily increasing computational demands. As scientific applications become more complex, new architectures are needed to serve these applications needs. Many hardware architectures have been developed through the last two decades to support high performance computing. Parallel and distributed computing systems are recognized today as an efficient computer architecture that can provide the potential for application performance superior to that achievable from any single processor system.

In parallel environments, the performance parameters space is larger than that in the sequential case. Therefore, dealing with performance in parallel environments tends to be more complex, since simple analysis of individual performance parameters is usually not a viable option because local optimum often does not lead to global optimum. Also, the alternative of simulating various performance parameters is time consuming, and, for many large applications, it may be impossible to accomplish in a reasonable amount of time. This situation led to concentrating numerous scattered efforts on developing specific tools to help users instrument, analyze, predict, and tune the performance of their parallel and distributed applications using broad range of approaches. Most of the developed tools tend to be used at run-time, using performance data gathered by instrumenting the application program [2,3,4,5,8]. A survey conducted by Pancake and Cook [6] has revealed the fact that tool use is still appallingly low among the high performance computing community. Cherri et al. [7] have cited the following three critical causes for this situation; (1) current tools are difficult to understand by scientific users, (2) tools vary widely across parallel platforms, and (3) current tools lack specialized support for heterogeneous and/or scalable applications.

The next section introduces a new approach to tune the performance of parallel applications by employing intelligent agents. This approach is discussed in details in sections 3, and 4. Results obtained from applying this approach to a parallel 2D Jacobi Solver are reported in section 5. Finally, the conclusion and future directions are discussed in section 6.

2 AGENT-BASED APPROACH

The objective of this research is to develop a new model for adaptive performance tuning of parallel and distributed applications that is based on intelligent agents. This approach uses a performance agent, as an adaptive controller, to monitor the application behavior during run time, and adapt and tune the application performance parameters values (PPVs). A general model for this approach is depicted in figure 2.

The performance agent regularly receives the performance data from logical process (LPs) and the system, and then it starts analyzing these values. Based on the analysis, performance agent properly modifies the PPVs and arrange for the new values to be sent to the corresponding LPs. If the performance agent decided that no changes could improve the performance at this time, the application would continue using the same PPVs. The performance agent provides the application with better PPVs more accurately and more timely. Unfortunately, this may be difficult to achieve because many performance parameters and metrics are correlated, and this operation is expected to be time consuming. Thus, the agent uses heuristic techniques to find PPVs

that lead to better performance. This research recommends that the performance agent should exhibit intelligence, autonomy, adaptability, collaboration, and reusability. The next two sections discuss the performance agent architecture and show how it supports all the recommended features.

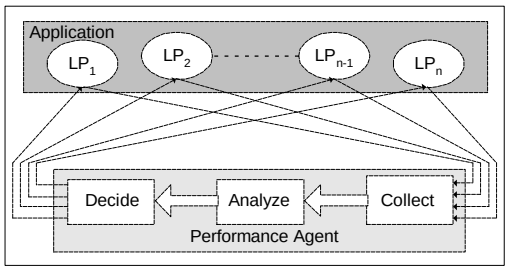

Figure 2. General Model

3 AGENT ARCHITECTURE

The performance agent architecture consists of a group of distributed agents, clustered in different levels. However, all agents have the same design, and one common goal of improving the overall application performance, agents in different levels have different domains, local goals, and responsibilities. This system architecture is depicted in figure 3 for a three levels hierarchy. It is worth mentioning that the number of levels depends on the underlying hardware architecture. Thus, if some of the processing nodes are clustered, this should add one more level to the model. The architecture shown in figure 3 has three different levels of agents, namely, logical process agents (LPAs), processor agents (PAs), and a master agent (MA). Agents on the same level are not aware of each other, and can only communicate with agents from different levels. The following sections provide descriptions of these different levels.

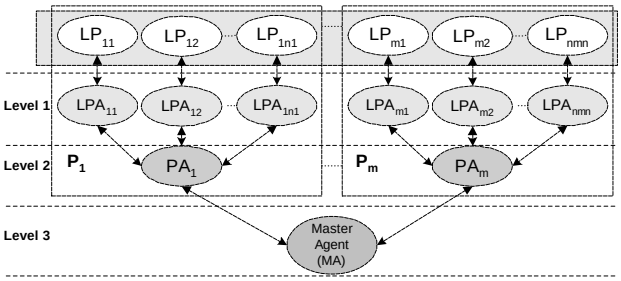

Figure 3. System Architecture

4 DESIGN OF AGENTS

In this model, each agent has three modules; namely, instrumentation module, analysis module, and decision module. The rest of this section is devoted to the detailed description of these three modules in the different types of agents. We will assume no special nodes clustering and thus three levels architecture will be considered.

4.1 Instrumentation Module

As shown in figure 5, this module is responsible for collecting data needed by the analysis module and it uses software sensors to capture application and/or system performance data. Capturing performance data can be as easy as receiving data from another agent, or as complex as instrumenting LPs or measuring system parameters. To instrument LPs, the application programmer should insert the instrumentation code within the LP to gather performance parameters and performance metrics while the application is being carried out. Direct measurements of system performance parameters always involve measuring resources availability and loading, such as processor load, cache size, and memory utilization.

However, sensors of LPAs use the instrumentation technique to gather performance data from the corresponding logical processes at frequency f_1, sensors of PAs receive summaries of data gathered at LPAs, in addition to the direct measurement of system performance using system calls at frequency f_2, where $f_2 < f_1$. Finally, the MA's sensor receives summaries of performance data gathered at the PAs at frequency f_3, where $f_3 < f_2$. The ideal design should assure lightweight sensors that minimally perturb the original system. Achieving this ideal is possible if sensor implementation can be customized to the execution environment and trade communication with computation, whenever possible, according to resources availability and allowed level of perturbation. Complexities involved in designing sensors include the trade-off between the frequency of measuring and/or instrumenting performance data, and the perturbation caused to the parallel application itself.

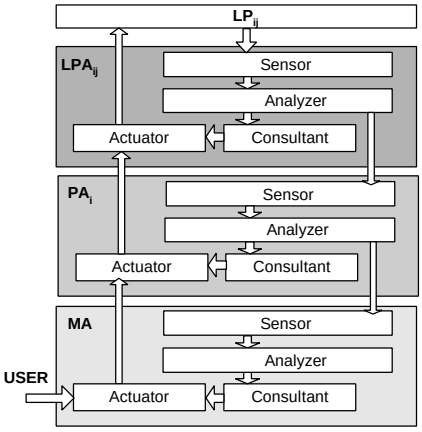

Figure 5. Agents Design

4.2 Analysis Module

This module analyzes the performance data received from the instrumentation module sensors and the analysis module of other agents using software analyzers. These analyzers perform three tasks: 1) filtering, 2) data synchronization, and 3) analysis. After receiving the performance data, analyzers start filtering these volumes. This task involves checking the validity and integrity of the data received. Then analyzers synchronize these volumes of data, such that they represent specific periods of the application space. Finally, analyzers start analyzing the synchronized performance data using statistical analysis. This step involves calculating statistical figures, such as means, variances, and/or standard deviations. After the analysis is completed, analyzers send summaries to the decision module, and/or the analysis module of other agents. It is worth mentioning that this module is almost identical in all the system model agents, even though they are in different levels. Also, the ideal analyzer should be light weighted, and minimally perturbing the original system. Achieving this ideal is possible if the sensor implementation can be tailored to the execution environment, such that the accuracy of its analysis is dependent on the available resources. In addition, the computation overheads involved in the analysis process are less expensive than the communication overhead involved in sending the entire volume of raw data to the decision layer.

Synchronization of data volumes is relatively complex in time dependent systems than it is in time independent applications.

4.3 Decision Module

This module is the central part of the agent architecture, and it has two main components: 1) consultant, and 2) actuator. The consultant is responsible for making the proper decision of modifying PPVs, if any will lead to better performance. This decision is based on the analysis received from the analysis module. The actuator is responsible for sending these tuned changes to the actuator of the corresponding higher-level agent, or to the corresponding *LP* if it is the *LPA* actuator. Typically, simple systems have few performance parameters and performance metrics with clear relations that can be represented in the form of simple decision rules. Therefore, constructing a decision tree is considered possible in simple systems. However, constructing such a decision tree in complex systems that have many performance parameters and performance metrics is unfeasible. This is not only because complex systems involve many performance parameters, but also, because these parameters are conflicting and have poorly understood relations. We recommended using heuristic techniques such as neural networks and/or fuzzy logic to help consultants making their decisions. The model's distributed decision and locality of optimization features increase its reliability and help in distributing the workload associated with it.

Agent's consultants at different levels are concerned with different sets of performance parameters. Consultants of LPAs are only concerned with performance data that affect the corresponding logical process, also consultants of PAs are only concerned with performance date that affect all LPs running on the corresponding processor. Finally, the consultant of the master agent is only concerned with performance data that affect the entire application LPs. Accordingly, actuators of LPAs are responsible for implementing decisions made by their consultants, and/or received from the corresponding PA to the corresponding LPs. And, actuators of PAs are responsible for sending decisions made by their consultants and/or received by the MA to the corresponding LPAs. Finally, The actuator of the MA allows the user to steer the application performance by manually setting some PPVs. Also, this actuator sends decisions made by its consultant and/or the human user to all PAs.

This system model features many strengths such as, support of system heterogeneity, minimal dependence on the application, and ease of use and development. The application programmer needs only to decide on the application performance parameters and metrics, insert instrumentation code, and define the relations between the different performance parameters. The last task may not be an easy job in complex systems, so we recommend developing an intelligent module to help the application programmer finding these relations and rules, if any. The performance of the new system model is vital and should be carefully studied to assure minimum perturbation to the original system.

This agent-based model is implemented to improve the performance of a 2D Jacobi solver to assure its independence of the application. The 2D Jacobi solver is selected as one of the typical computationally intensive parallel application. The following section describes the 2D Jacobi solver briefly. Then, the working environment is introduced, and detailed design and implementation steps are discussed. Finally, results obtained are presented and discussed.

5 EXPERIMENT

The 2D Jacobi solver is an iterative method commonly used to solve the finite-difference approximation to Poisson's equation, which arises in many heat flow, electrostatic, and gravitational problems. Variable coefficients are represented as elements of a two-dimensional grid. It is an iterative method, meaning that it repeatedly updates an approximate solution, so that the solution converges as we execute more iterations. The number of iterations required depends on the grid entries, and the desired accuracy. At each iteration, the new value of each grid element is defined to be the average of its four nearest neighbors during the previous iteration. The original matrix (grid) is partitioned into sixteen sub-matrices, and each sub-matrix is passed to a logical process. Therefore, the 2D Jacobi is using sixteen LPs (LP_0, LP_1, ..., LP_{15}). The original matrix is partitioned by the root process (LP_0) using a row-wise block partitioning, where the matrix is divided into groups of contiguous rows. Each group contains a number of rows and is sent to a computing process.

5.1 Working Environment

This experiment is implemented on a Beowulf type cluster [1] that is composed of eight processing nodes (node0, node1, ..., node7). Each of these nodes is an Intel Pentium II processor at 400 MHz, with 128 Mbytes of memory and 4.3 Gbytes hard disk. These nodes are connected with a fast Ethernet backbone at 100Mb/sec. Each node runs Linux RedHat 5.1, kernel 2.0.34, operating system with a shared file system. The message passing interface (MPI) standard library is used to implement the application and system model. The Argonne National Laboratory's implementation of the MPI (MPICH 1.1.0) is adopted. Also, the GNU C-compiler (gcc 2.7.2.3) is used.

5.2 Design and Implementation

The 2D Jacobi is designed to use 16 logical processes (LP_0, LP_1, ..., LP_{15}). Each of the 8 processors runs two LPs, such that, processor P_i runs processes LP_{2i} and LP_{2i+1}. LP_0 partitions the original matrix into 16 blocks; each of these blocks consists of a number of continuous rows. Then, LP_0 sends to each LP its corresponding block (sub-matrix), as shown in figure 7. Once a LP receives its partition it starts computation as described above. On finishing the computation, each LP sends the processed block to LP_0. LP_0 receives different blocks from different LPs and restore them in the original matrix. This is considered a complete iteration. The experiment has been run for 200 iterations.

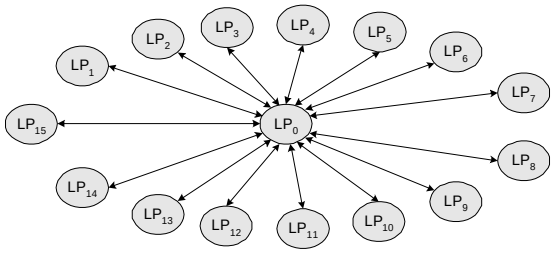

Figure 7. 2D Jacobi topology

Having the topology of the parallel application (2D Jacobi) already known, the performance agent model is created automatically. The performance agent starts 16 LPAs, 8 PAs, and a master agent (MA). Each processor in the system runs two LPs,

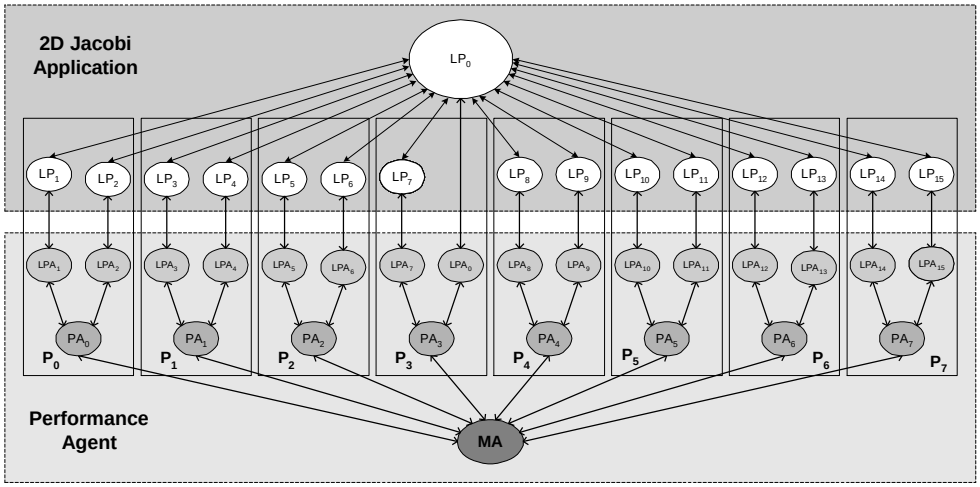

Figure 8. Application and Performance Agent Architecture

two LPAs, and one PA. The MA is assigned to P_0. The entire system architecture is shown in figure 8. The created model is independent of the application communication or the interaction between its LPs.

The 2D Jacobi application is considered simple since it has few performance parameters and performance metrics. The following performance parameters are indicated to affect the application performance. **Computing algorithm:** This is the computing algorithm used to calculate the new cell value. There could be two or more different algorithms with complexities of the same order, such that, one is preferable than the others in certain conditions such as the memory allocation technique. **Memory allocation:** Static and dynamic memory allocations are two different techniques of allocating memory. Usually, no one can decide which technique is better a priori. **Local matrix size:** The size of the sub-matrix assigned to each processor is an important performance parameter.

The following performance metrics are indicated to represent the progress in the application execution. **CPU load:** The percentage loading of the CPU is an indication of the processor ability of processing more or less data. **Memory utilization:** The percentage utilization of the system memory is an indication of the processor ability of processing more or less data. **Iteration completion time:** The time needed by a LP to finish a single local iteration is an important measure of this LP's behavior.

LPAs are responsible for collecting the performance parameters (computing algorithm, memory allocation technique, and local matrix sizes) and the iteration completion time metric. Each LPA receives these values from the corresponding LP once every four iterations. LPAs are also responsible for deciding on the computing algorithm. This decision is based on the measured iteration completion time. PAs receive the averages of performance data measured by the corresponding LPAs once every 8 iterations. Also, each PA is responsible for measuring its processor's percentage load and memory utilization. Usually, system calls are used to accomplish this task. Based on the performance data received and measured by PAs, each PA decides on the memory allocation technique, and sends the change in this performance parameter back to the corresponding LPAs, and then every LPA sends this change to its corresponding LP.

The MA receives the averages of the performance data measured and received by PAs. Based on these averages, the MA decides on the size of the matrices to be assigned to each processor. The local matrix to a processor is evenly divided among this processor's LPs. The MA sends these changes to the corresponding PAs, then each PA sends these changes to the corresponding set of LPAs, and finally, each LPA sends these changes to its corresponding LP. The MA calculates a load metric for each processor to evaluate the relative loading of all processors, and accordingly assigns matrix sizes. The load metric chosen for this application was: **load=w * cpu_load + mem_util** , where:

- load is the calculated measure for each processor loading.
- cpu_load is the percentage load of the CPU.
- mem_util is the percentage memory utilization.
- w is a constant weight that represents the relative importance of the cpu_load over or below mem_util. If *w* is greater than 1, then this means the cpu_util is more important than the mem_util, and vice versa.

It is important to mention that the model uses non-blocking communication between agents. This helps in reducing agent weight, and more importantly enables an agent to listen to messages coming from more than one sender at a time. LPs listen to the changes coming from LPAs, LPAs listen to parameters coming from LPs and changes coming from PAs, PAs listen to parameters coming from LPAs and changes coming from MA, and MA listen to parameters coming from PAs. This model design enables the system to run copies of four different programs, i.e. the application, LPAs, PAs, and the MA.

5.3 Results

Four experiments were designed to test the model. The first experiment was designed to compare the 2D Jacobi without the performance agent, against the 2D Jacobi with the performance agent, such that, all agents are functioning. The second experiment was designed to compare the 2D Jacobi without the performance agent, against 2D Jacobi with the performance agent, such that, only the PAs and the MA are in use. The third experiment was designed to compare the 2D Jacobi without the performance agent, against 2D Jacobi with the performance agent, such that, only the LPAs and the MA are in use. The fourth experiment was designed to compare the 2D Jacobi without the

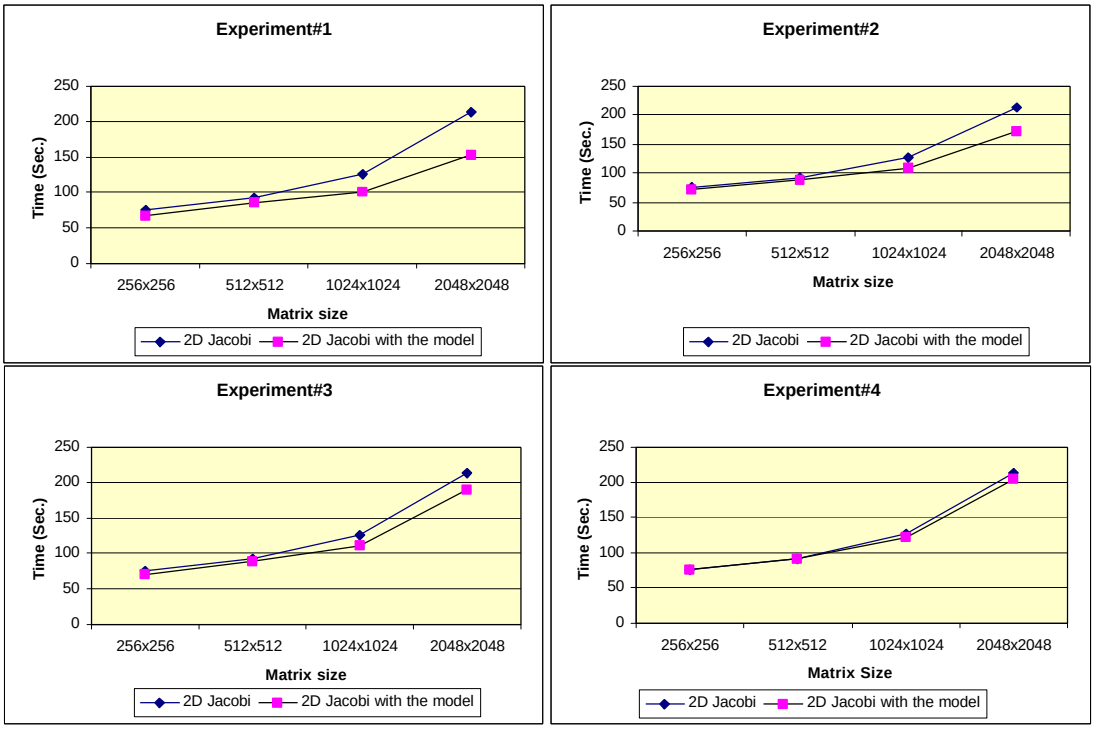

Figure 9. Experiments Results

performance agent, against 2D Jacobi with the performance agent, such that, only the LPAs and PAs are in use. Each of these experiments has been run on four different matrix sizes, 256x256, 512x512, 1024x1024, and 2048x2048. Results obtained are presented in figure 9.

Results obtained show that larger reduction in the computation-time is achieved by using the complete performance agent (maximum reduction of 28%), the first experiment. Less improvement in the system performance is achieved in the second experiment, where only PAs and the MA are in use (maximum reduction of 19%). Results obtained from the third experiment show that, if only LPAs and the MA are used, the performance agent causes moderate improvement to the system performance (maximum reduction of computation-time of 11.2%). The fourth experiment show that if only LPAs and PAs are used the performance agent will slightly improve the system performance. This indicates the effectiveness of the complete system model in tuning performance parameters and the importance of the MA in this specific application.

6 CONCLUSIONS

It is important to develop new tools, models, and architectures to help understand and improve the performance of parallel and distributed applications and to make better use of the allocated resources. Intelligent tools have great potential to assist the human user in tuning the application performance, but these tools must be easy to implement and understand for application programmers and designers.

The performance agent model presented in this paper has demonstrated promising results when applied to the 2D Jacobi problem. This new model showed much strength, such as minimal dependence on the application, support of heterogeneity, and ease of implementation.

REFERENCES

[1] J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, and C.V. Packer, "Beowulf: A Parallel Workstation for Scientific Computation," Proc. International Conference on Parallel Processing, Aug. 1995.

[2] K. Ekanadham, V.K. Naik and M.S. Squillante, "PET: Parallel Performance Estimation Tool," Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995.

[3] J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," Proc. Int'l Conf. on Supercomputing, Tokyo, Jul 1993.

[4] J.K. Hollingsworth and B.P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation," Proc. Supercomputing'92, Minneapolis, Nov 1992.

[5] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool, " IEEE Computer, vol. 28, no. 11, Nov 1995.

[6] C.M. Pancake and C. Cook, "What users Need in Parallel Tool Support: Survey Results and Analysis," Proceeding of Scalable High-Performance Computing Conference, Knoxville, Tennessee, May 1994.

[7] C.M. Pancake, M.L. Simmons, and J.C. Yan, "Performance Evaluation Tools for Parallel and Distributed Systems," IEEE Computer, vol. 28, no. 11, Nov 1995.

[8] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," Proc. Scalable Parallel Libraries Conference, IEEE Computer Society, Oct 1993.