Automatic Tuning of Master/Worker Applications*

Anna Morajko, Eduardo César, Paola Caymes-Scutari,
Tomás Margalef, Joan Sorribes, and Emilio Luque

Computer Science Department. Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
{ania,paola}@aomail.uab.es
{eduardo.cesar,tomas.margalef,joan.sorribes,emilio.luque}@uab.es

Abstract. The Master/Worker paradigm is one of the most commonly used by parallel/distributed application developers. This paradigm is easy to understand and is fairly close to the abstract concept of a wide range of applications. However, to obtain adequate performance indexes, such a paradigm must be managed in a very precise way. There are certain features, such as data distribution or the number of workers, that must be tuned properly in order to obtain such performance indexes, and in most cases they cannot be tuned statically since they depend on the particular conditions of each execution. In this context, dynamic tuning seems to be a highly promising approach since it provides the capability to change the parameters during the execution of the application to improve performance. In this paper, we demonstrate the usage of a dynamic tuning environment that allows for adaptation of the number of workers based on a theoretical model of Master/Worker behavior. The results show that such an approach significantly improves the execution time when the application modifies its behavior during execution.

1 Introduction

The Master/Worker (M/W) paradigm is one of the most commonly used by parallel/distributed application developers. In this paradigm, a master process distributes a set of data to be processed among a set of worker processes that receives this data, processes it and returns the results to the master. This structure fairly faithfully represents the developer abstract concept. It can be applied to a wide range of applications and is therefore fairly easy to treat and manage. However, the actual behavior of this structure depends on several features (target system, number of available processors, computing capabilities, communication features, input data) that cannot be controlled by the application developer and can only be found out during runtime. In order to reach high performance indexes and eliminate performance bottlenecks, the behavior of the particular application must be analyzed and problems that appear during the execution must be determined.

One of the major performance bottlenecks in the Master/Worker paradigm is the inadequate number of workers. When there are not enough worker processes, the master process distributes the data and becomes idle as it waits for results. On the other hand, if there are too many workers, the amount of data is divided into small pieces and the communications saturate the system. Therefore, it is important to find an optimal number of workers. This number depends on: the computing volume per

* This work has been supported by the MCyT (Spain) under contract TIC2001-2592 and has been partially supported by the *Generalitat de Catalunya* – GRC 2001SGR-00218

datum, the volume of data sent to and received from the worker processes, the computing capabilities of each of the system's processors and the latency and bandwidth of the communication network.

In many cases, these features are not completely static and change dynamically during the execution of the application (e.g., the computing requirements evolve during execution of the application or the computing capabilities of the processors change due to an additional load in the system). In these situations, the optimal number of workers is not fixed, but changes during the execution of the application and it must be tuned dynamically.

In the following sections of this paper, we present a complete performance optimization scenario that considers the problem of the number of workers in a dynamic approach. Section 2 presents example automatic analysis and tuning environments. In Section 3, we describe the performance model used to calculate the optimal number of workers. In Section 4, we analyze the tuning of the number of workers using the MATE environment that supports the dynamic tuning of parallel applications. In Section 5, we present the results of the experiments conducted in the MATE environment to dynamically tune the number of workers using the presented performance model. Finally, Section 6 shows the conclusions of this study.

2 Related Work

The optimization process requires a developer to go through the application performance analysis and the modification of critical application parameters. First, the performance measurements must be taken in order to provide information about the application. Then, the analysis of this information is carried out. It finds performance bottlenecks, deduces their causes and determines the actions to be taken to eliminate these bottlenecks. Finally, appropriate changes must be applied into the application.

To reduce developers efforts, an automatic analysis has been proposed. Tools using this type of analysis are based on the knowledge of well-known performance problems. They are able to identify critical bottlenecks and help in optimizing applications by giving suggestions to developers [1, 2, 3, 4].

Such tools require a certain degree of knowledge and experience of parallel/distributed applications. To tackle these problems, it is necessary to provide tools that automatically perform program optimizations during run time. Active Harmony [5] is a framework that allows an application for dynamic adaptation to network and resource capacities. The application must be Harmony-aware, that is, to use the API provided by the system. The project focuses on the selection of the most appropriate algorithm. Active Harmony automatically determines good values for tunable parameters by searching the parameter value space using heuristic algorithm. MATE uses a distinct approach in which performance models provide conditions and formulas that describe the application behavior and allow the system to find the optimal values. The AppLeS [6] project has developed an application-level scheduling approach. It combines dynamic system performance information with application-specific models and user specified parameters to provide better schedules. A programmer is supplied information about the computing environment and is given a library to facilitate reactions to changes in available resources. Each application then selects the resources and determines an efficient schedule, trying to improve its own performance without considering other applications. MATE is similar to AppLeS in that it tries to maximize the performance of a single application. However, MATE focuses on the efficiency of resource utilization rather than on resource scheduling.

3 Performance Model for the Number of Workers

In this section, we present the problem of determining a suitable number of workers for a M/W application. We will only consider this problem for homogeneous M/W applications, defining these as applications where all tasks (i.e. a set of data to be processed by each worker) are approximately of the same size and require the same processing time. In actual fact, these kinds of applications exhibit a similar performance to a balanced M/W application with the same total processing time and the same global communication volume, as shown in [7]. This is an important observation, because in homogeneous application it is easier to determine the appropriate number of processors to be used.

For this analysis, we have assumed that the following conditions are met:

- There is just one process (master or worker) per processing element.
- The master process distributes all available data among workers, then waits for all results and, eventually sends a new set of tasks to workers, which means that the application could be iterative.

In addition, we will use the following terminology to identify the different parameters that will form part of the performance model:

- tl = fixed network time overhead per message, in ms.
- λ = communication cost per byte (inverse bandwidth), in ms/byte.
- v_i = size of tasks sent to worker i, in bytes.
- v_m = size of results sent back to master from each worker, in bytes.
- V = total data volume (Σ ($v_i + v_m$)), in bytes.
- n = current number of workers in the application.
- tc_i = time that worker i spends processing a task, in ms.
- Tc = total computing time (Σ tc_i)
- Tt = total time spent on an application iteration (execution time). Our objective is to estimate and minimize this magnitude.
- Nopt = number of workers needed to obtain the minimum Tt (best performance).

It can be seen that the parameters that must be monitored in order to apply the performance model associated to a M/W application are:

- tl and λ which could be calculated at the beginning of the execution and should be re-evaluated periodically to make allowances for the adaptation of the system to the network load conditions.
- Task sizes (v_i) have to be captured when the master sends tasks to workers.
- Result sizes (v_m) have to be captured when the master receives results from workers.
- The time the workers spend on each task (tc_i) has to be measured in order to calculate the total computing time (Tc).

Now, we can describe the analysis performed in order to construct the performance functions associated to this kind of application. We should point out that these functions are defined to enable the optimization of the execution time of the application (Tt).

First, the master sends a set of tasks to each worker. If the communication protocol is asynchronous then the network overhead (tl) for one message overlaps with the communication time of the previous one ($\lambda * v_i$), otherwise both times should be added.

The time spent on this operation is $n*tl+\lambda*\sum_{i=0}^{n-1} v_i$ if the communication protocol is synchronous but, if the protocol is asynchronous then it depends on the relation between the network overhead (tl) and the communication time ($\lambda * v_i$).

If tl is greater than $\lambda * v_i$ (communication time of the tasks sent to one worker) then it is $n*tl$ (network overhead) $+\lambda * v_i$. This is the overhead of sending messages to all workers plus the communication time of the last message, otherwise, it is $tl+\lambda*\sum_{i=0}^{n-1} v_i$. This is the overhead of the first message plus the communication time of all messages.

Then, as every worker spends the same time processing its tasks, we just have to add the processing time of one worker (the last one to receive a task); which is tc_i.

At this point, processing has finished and we must evaluate what happens to the results sent back to the master. We only need to add the communication time for the last message, which is tl + $\lambda * v_m$ (communication time of one answer). This last statement only holds if the master has completed the data distribution before there is an answer from a worker, otherwise it will not be ready to receive messages when the last worker sends its results back.

This never happens before the optimal number of workers if $tl \geq \lambda * v_i$, but may not be true if $tl < \lambda * v_i$ or when the communication protocol is synchronous. In the latter case, the following condition must also hold: the time spent by the master to distribute the tasks ($tl+\lambda*\sum_{i=0}^{n-1} v_i$ or $n*tl+\lambda*\sum_{i=0}^{n-1} v_i$) must be greater than the response time of the first worker ($2*tl+\lambda*v_i+tc_i+\lambda*v_m$).

The expressions to calculate the total iteration time are formed by adding these quantities together, if the communication protocol is synchronous and $n*tl+\lambda*\sum_{i=0}^{n-1} v_i \geq 2*tl+\lambda*v_i+tc_i+\lambda*v_m$ then we get:

$$Tt = n*(tl+1)+\lambda*\sum_{i=0}^{i=n-1} v_i + tc_i + \lambda*v_m$$

But, if the communication protocol is asynchronous we get:

$$Tt = 2*tl+\lambda*\sum_{i=0}^{n-1} v_i + tc_i + \lambda*v_m$$

$$if((tl \leq \lambda*v_i) and (tl+\lambda*\sum_{i=0}^{n-1} v_i > 2*tl+\lambda*v_i+tc_i+\lambda*v_m))$$

Or

$$Tt = n*tl+\lambda*v_i+tc_i+tl+\lambda*vm \quad (if\ tl > \lambda*v_i)$$

Considering that tc_i = Tc/n, v_i = p*V/n (a portion *p* of the overall data volume which is distributed among the workers), and v_m = (1-p)*V/n (the remaining portion of the overall data volume which are the results that workers return to the master) we could rewrite these expressions as:

$$Tt = n * tl + \frac{Tc}{n} + tl + (n-1)*(p+1)*\lambda*\frac{V}{n} \quad \text{if protocol is synchronous and } n*(tl + \lambda * V/n) > 2*tl + \lambda*V/n + Tc/n \tag{1}$$

Or

$$Tt = \frac{(2*tl + \lambda*V*p)*n + Tc + \lambda*(1-p)*V}{n} \quad \text{if protocol is asynchronous and } (tl \le \lambda * p * V/n) \text{ and } (n \le \lceil(\lambda*V+Tc)/(\lambda*p*V-tl)\rceil \tag{2}$$

Or

$$Tt = n*tl + \frac{Tc}{n} + tl + \lambda*\frac{V}{n} \quad \text{if protocol is asynchronous and } (tl > \lambda*\frac{p*V}{n}) \tag{3}$$

If we calculate δTt/δn = 0 for expression (1) then we will obtain an expression to calculate the number of workers needed to minimize Tt when the communication protocol is synchronous, which is:

$$N_{opt} = \frac{1}{2}\sqrt{(\lambda*V+4Tc)/tl} \tag{4}$$

And, if we calculate δTt/δn = 0 for expression (3) then we will obtain an expression to calculate the number of workers needed to minimize Tt when the communication protocol is asynchronous, which is:

$$N_{opt} = \sqrt{(\lambda*V+Tc)/tl} \tag{5}$$

We cannot do the same with expression (2) because it can easily be demonstrated that for this expression: $\lim_{n\to\infty} Tt = 0$. But, if the number of workers (n) grows, then the message size (v_i) decreases and, consequently: tl > λ*p*V/n when n > λ*V/(2*tl). This means that expression (5) can be also applied from the time this condition holds. With expressions (1), (2) and (3), we have a model of the behavior of an application, and we have expressions (4) and (5) to tune the number of workers of the application.

Figure 1 shows the expected execution time for an example M/W application considering expression (2) and compares the results of predicted values to the real execution times. This figure presents also the optimal number of workers provided by expression (4). It can be observed that the predicted behavior matches well the real behavior.

4 Tuning Number of Workers with MATE

The performance model described in the previous section provides the optimal number of workers for a particular situation. However, in many cases the developer of an M/W application cannot know all of the details needed to provide such an optimal number. Moreover, in many cases the conditions change during the execution of the application (for example, systems with shared load) and the optimal number of workers is not fixed, but evolves during the execution of the application. In these cases, number must be adjusted on the fly during the execution of the application.

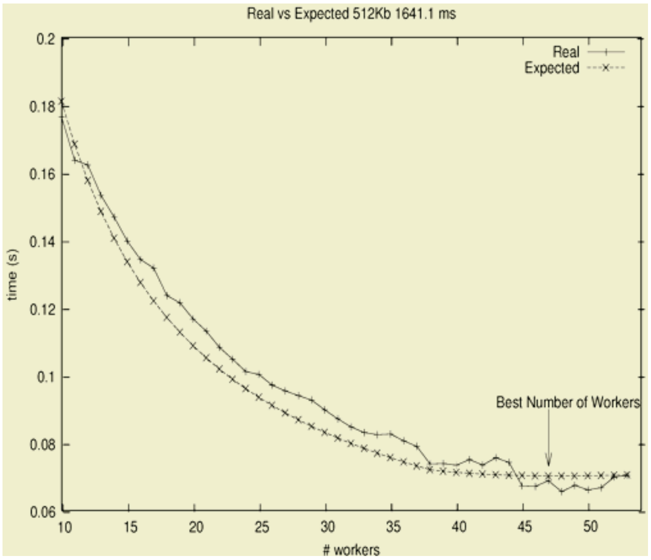

Fig. 1. Real vs. expected execution time, showing the use of expressions (2) and (4)

To provide dynamic automatic tuning of parallel/distributed applications we have developed an environment called MATE (Monitoring, Analysis and Tuning Environment) [8, 9]. MATE performs dynamic tuning in three basic and continuous phases: monitoring, performance analysis and modifications. This environment dynamically and automatically instruments a running application to gather information about the application's behavior. The technique that fulfills these requirements is called dynamic instrumentation [10]. The analysis phase receives events, searches for bottlenecks applying a performance model and determines solutions to overcome such performance bottlenecks. Finally, the application is dynamically tuned by applying the given solution. Moreover, while it is being tuned, the application does not need to be re-compiled, re-linked or restarted. The knowledge to represent the performance model of each particular performance problem is specified in a component called a "tunlet". Each tunlet includes the information about the measure points to insert instrumentation into the target application, the performance model to determine the behavior of the application and the required modifications, and finally, the tuning actions to improve the application's performance.

We have defined two main approaches to tuning: automatic and cooperative. In the automatic approach, an application is treated as a black-box, because no application-specific knowledge is provided by the programmer. This approach attempts to tune any application and does not require the developer to prepare it for tuning (the source code does not need to be adapted). The cooperative approach assumes that the application is tunable and adaptable. This means that developers must prepare the application for the possible changes.

We have conducted a variety of practical experiments on parallel/distributed applications to check whether our approach really works. We have proven that it is effective, profitable, and can be used for a real improvement in program performance. Running applications under MATE control has allowed for adaptation of their behavior to the existing conditions and improvements in their performance.

To dynamically tune the number of workers, we determined conditions that a M/W application must fulfill (as this optimization belongs to the cooperative approach) and implemented a specific tunlet. The application must be based on iterations where all processes repeatedly perform all operations. During each iteration, the master distributes tasks to a specified number of workers and then waits for the results. It must synchronize the results before the next iteration. Tasks being distributed must be independent of each other. In addition, the task processing time cannot depend on the task content, but only on the task size. Finally, worker processes cannot exchange tasks with each other in order to calculate and provide results. The condition of the iteration-based application structure implies the existence of a significant number of iterations. If there is a small number of repetitions, the tuning overhead might be high and the improvement might not be seen.

The tunlet that optimizes the number of workers requires run-time monitoring of the functions responsible for exchanging messages (send and receive), in particular: send entry/exit, receive entry/exit events in the master process, and receive entry/exit and send entry/exit in all worker processes. Instrumenting these functions we are able to perform all measurements required by the performance model presented in Section 3 (expressions (4) and (5)).

The model is evaluated after each iteration when all measurements gathered from that iteration are available. If the computed optimal number of workers differs from the current value, the associated tuning procedure is invoked. In this case, we require the application to be prepared by the developer for the potential changes. The application must contain the specific variable that represents the number of workers. MATE will change this variable automatically. During execution, the application should be aware of the current number of workers and if it is different from the previous one, the new number must be used. This can only be done between two iterations because it is difficult to change the current work distribution that is already being processed. Once the number of workers has been adjusted, the work can be distributed adequately to all running workers.

If there are any new workers to be added, the new machines (processors) are required for them. There is no sense in running a new worker on the same machine where another worker is already running. In such a situation we would not gain anything since the CPU time is divided between both workers.

5 Experimental Results

In this section, the experimental results obtained by applying the tuning environment to a real Master/Worker application are presented. To conduct the experiments, we selected an intensive computing Forest Fire Propagation application called Xfire [11]. The Xfire application is a Master/Worker PVM based implementation of the simulation of the fireline propagation. It calculates the next position of the fireline considering the current fireline position and different aspects such as weather, wind, vegetation, etc. Experiments were conducted on a cluster of homogenous Pentium 4, 1.8 Ghz, (SuSE Linux 8.0) connected by a 100Mb/sec network.

Since we need to control the load in the system to reproduce the experiments several times, we created certain load patterns, so that we can introduce and modify certain external loads to simulate the system's time-sharing. We defined load patterns and executed the application with several fixed number of workers (2, 4, 6, and successively until 26) and also under the control of the MATE tuning environment where the number of workers is adapted dynamically. In every scenario one worker was executed in the same machine as master.

We have conducted our experiments in two scenarios:

- In the first scenario, Xfire was executed on different number of workers, without any tuning.
- In the second scenario Xfire was executed under MATE applying the tuning of the number of workers. The application started with one worker and then during the execution the number is changed according to the model described in Section 3. In this scenario one machine of the cluster was dedicated to run the analyzer, so that the analysis does not introduce additional overhead in the application.

Table 1 summarizes the experimental results. These results are also presented in Figure 2.

Table 1. Execution time of Xfire (in seconds) considering different number of workers, and Xfire under MATE

#workers	1	2	4	6	8	10	12	14	16	18	20	22	24	26
Execution Time	1209	624	345	249	206	181	166	156	144	137	130	129	122	125
Xfire + MATE Execution Time	Starting with 1 worker								141					

Figure 2 shows the execution time of Xfire application considering different number of workers and in the last column the execution time of Xfire under MATE. As it is indicated before, Xfire while executed under control of MATE starts with only one worker. When MATE receives all data from the first iteration, it evaluates the performance model and immediately detects the need of adding workers to reach the optimal number related to the initial total work. Then during the execution of the application the load is changed and the number of workers is adapted to the optimal number provided by the performance model.

It can be observed that execution time of Xfire under MATE is close to the best execution times obtained by different fixed number of workers. However, the re-

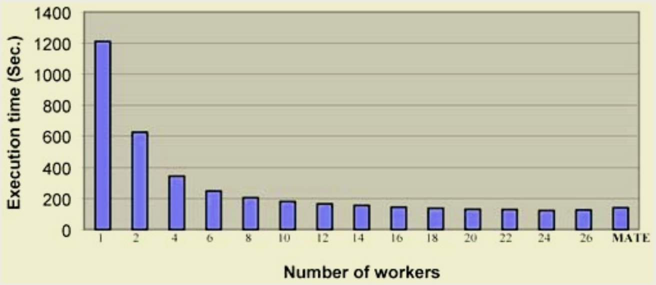

Fig. 2. Execution time of Xfire considering different number of workers and Xfire under MATE

sources devoted to the application using the MATE tuning environment are taken considering the actual requirements of the application and are used when they are really needed.

6 Conclusions

Parallel and distributed programming offer high computing capabilities to users in many scientific research fields. The performance of applications written for such environments is one of the crucial issues. Master/Worker is one of the most significant paradigms in these environments. The number of workers is a key issue in considering the performance of the application.

A performance model to evaluate the optimal number of workers has been presented. This performance model has been incorporated into the MATE automatic tuning environment by the corresponding "tunlet". The presented optimization scenario adapts the number of workers assigned to perform a specified amount of work to changing environment conditions. It requires the application to be prepared for the possible changes, i.e. adding or removing worker processes. MATE is able to estimate the application's performance by means of the analytical model, and to calculate and apply the optimal number of workers. The tuning action changes the number of workers by updating the variable value in the master process.

The experimental results show that the dynamic tuning approach significantly improves the execution times without consuming unnecessary resources when the application is executed under dynamic conditions (changes in the system load).

References

1. Espinosa, A., Margalef, T., Luque, E. "Automatic Performance Analysis of PVM applications". EuroPVM/MPI 2000, LNCS 1908, pp. 47-55. 2000.
2. Wolf, F., Mohr, B., "Automatic Performance Analysis of MPI Applications Based on Event Traces". EuroPar 2000, LNCS 1900, pp. 123-132. 2000.
3. Truong, H.L., Fahringer, T. "Scalea: A Performance Analysis Tool for Distributed and Parallel Programs". EuroPar 2002, LNCS 2400, pp. 75-85. 2002.
4. Miller, B.P., Callaghan, M.D., Cargille, J.M. Hollingswoth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam K., Newhall, T. "The Paradyn Parallel Performance Measurement Tool". IEEE Computer vol. 28. pp. 37-46. November 1995.
5. Tapus, C., Chung, I-H., Hollingsworth, J.K. "Active Harmony: Towards Automated Performance Tuning". SC'02. November 2002.
6. Berman, F., Wolski, R. "Scheduling From the Perspective of the Application". High Performance Distributed Computing 1996. Syracuse, NY, USA, August 1996.
7. César, E., Mesa, J.G., Sorribes, J., Luque, E. "Modeling Master-Worker Applications in POETRIES". IEEE 9th International Workshop HIPS 2004, IPDPS, pp. 22-30. April, 2004.
8. Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E. "Dynamic Performance Tuning of Distributed Programming Libraries". LNCS, 2660, pp. 191-200. 2003.
9. Morajko, A., Morajko, O., Margalef, T., Luque, E.. "MATE: Dynamic Performance Tuning Environment". LNCS, 3149, pp. 98-107. 2004.
10. Buck, B., Hollingsworth, J.K. "An API for Runtime Code Patching". University of Maryland, Computer Science Department, Journal of High Performance Computing Applications. 2000.
11. Jorba, J., Margalef, T., Luque, E., Andre, J, Viegas, D.X. "Application of Parallel Computing to the Simulation of Forest Fire Propagation", Proc. 3rd International Conference in Forest Fire Propagation, Vol. 1, pp. 891-900. Portugal, November 1998.