Invited Paper:
Development and Tuning Framework of Master/Worker Applications

Anna Morajko, Eduardo César, Paola Caymes-Scutari, José G. Mesa, Genaro Costa, Tomàs Margalef, Joan Sorribes, Emilio Luque
Computer Architecture and Operating Systems Dept. Universitat Autònoma de Barcelona.
{Anna.Morajko, Eduardo.Cesar, Tomas.Margalef, Joan.Sorribes, Emilio.Luque}@uab.es
{paola, genaro}@aomail.uab.es
josegabrielmesa xeenarts@gmail.com

ABSTRACT

Parallel/distributed programming is a complex task that requires a high degree of expertise to fulfill the expectations of high performance computation. The Master/Worker paradigm is one of the most commonly used because it is easy to understand and there is a wide range of applications that match this paradigm. However, there are certain features, such as data distribution and the number of workers that must be tuned properly to obtain adequate performance. In most cases such features cannot be tuned statically since they depend on the particular conditions of each execution. In this paper, we show a dynamic tuning environment that is based on a theoretical model of Master/Worker behavior and allows for the adaptation of such applications to the dynamic conditions of execution. The environment includes a pattern based application development framework that allows the user to concentrate on the design phase and makes it easier to overcome performance bottlenecks.

Keywords: dynamic tuning, performance analysis, performance model

1. INTRODUCTION

Parallel/distributed processing is a highly promising approach to attaining the high computing capabilities required by many application fields. However, the use of these computing systems involves several features that are not considered in classical sequential processing:

- Design and development of these applications imply the study and development of new parallel algorithms to solve the target problem.
- Programming of these parallel algorithms requires the use of some message passing library (PVM, MPI) to communicate the processes cooperating in the application. Therefore, the programmer must use a set of new primitives and match the parallel algorithm to the possibilities of the library.
- To accomplish the high performance expectations, programmers must analyze the performance of the application in order to determine the bottlenecks that appear during the execution of the application and modify the application to overcome them.
- In many cases, the behavior of these applications varies depending on the input data set or on the platform under use. In other cases the behavior can change dynamically due to the evolving of the application or the dynamic features of the system. In such cases, the tuning actions must be performed during the execution of the application, and this is a jeopardizing action.

In this context, new software tools that help the user in the application specification, hiding the low level details related to process communication, and automatically tuning the application performance are needed. It is necessary to offer a certain program specification framework that represents a higher level of abstraction. Such framework provides a set of well-defined structures in such a way that the programmer selects the structure under use and specifies the features directly related to the particular application (e.g. data sets, functionality). Finally, the framework generates all the code using the appropriate message passing library. In this approach, the programmer is constrained to use a given set of structures, but the specification is much simpler. On the other hand, since the applications follow well-known structures it is possible to develop performance models of such structures and tune the application performance on the fly.

In this paper, a complete environment that allows the programmer to develop a parallel/distributed application by using a programming framework and to execute the application under the control of an automatic and dynamic tuning system is presented. Section 2 describes the framework design for developing Master/Worker applications. Section 3 presents the performance model designed for this kind of application. In Section 4 we introduce the tuning system (MATE) and the integration of the performance model for Master/Worker applications. Section 5 shows some experimental results with an N-Body problem application developed using the framework and tuned by MATE. Finally, Section 6 presents some conclusions.

2. FRAMEWORK DESIGN

We consider that the frameworks have to fulfill the following requirements:

- *They should be generic and flexible.* Any application that matches the design pattern implemented by the framework could be written using it.
- *They should be easy to use.* The framework's implementation details must be hidden to the programmer providing a clear and easy to understand API.
- *They should be efficient.* Although these frameworks will be integrated into an automatic tuning environment, efficiency cannot be neglected.
- *They should be independent of the underlying communication library.* Library functions should not be directly included in the framework code.

From another point of view, the objective of using these frameworks in a dynamic tuning environment sets an extra requirement:

- They must be designed to simplify the monitoring and dynamic modification of generated applications.

To fulfill these requirements a great deal of care was taken over a design that uses object oriented techniques that allow for the encapsulation of the framework behavior and implementation, while leaving it down to the programmer to specify only those methods where the solution for the computational problem must be computed. This design is

[1] This work was supported by the MCyT under contract number TIN 2004-03388 and partially funded by the Generalitat de Catalunya – Grup de recerca consolidat 2001-SGR-00218.

based on the fact that for any programming paradigm (master-worker, pipeline, divide and conquer, and so on) two different elements are clearly distinguished: first, the processes which are in charge of performing the computation, and which have specific behavior in accordance with their role in the paradigm, and second the management of the data that has to be interchanged between those processes.

The resulting class structure is shown in Figure 1, where we can see that for each kind of process in a programming paradigm the related framework includes a class (*Process class*) to encapsulate its behavior, defining as virtual those methods that will be used by the programmer in a derived class (*MyProcess class*) to specify the particular computation performed by the process. In this derived class the programmer must codify the specific computation of the application through *computation methods*, as well as other methods for *initialization* and *finalization* code.

It can be seen that the framework also includes classes that encapsulate the management of the data that will be exchanged between processes (*Communication management class*). The idea is that an object of a particular process class owns an object of a communication management class associated with this kind of process. The communication management object is responsible for managing the logical channels used to communicate the process with others, and the data that is communicated through these channels. This object offers programmers a communication interface.

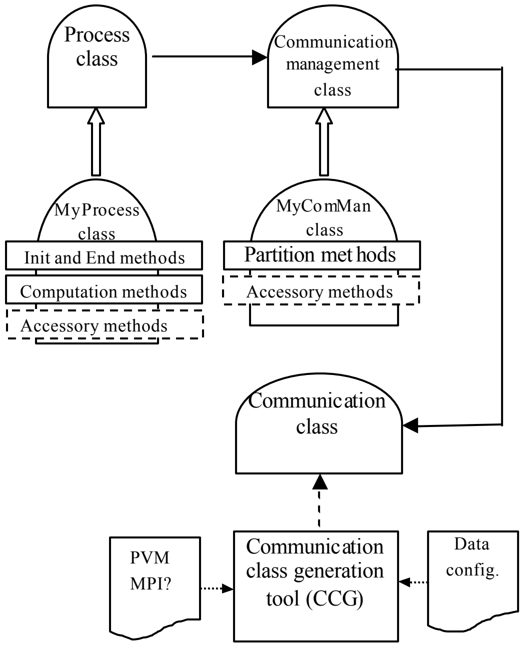

Fig. 1. Class structure for frameworks.

There are only two methods used by the communication management class that should be overwritten by the programmer, the method used to make partitions in the process data, which has to be sent to other processes, and the one used to fill process data structures with data received from other processes (*Partition methods*). This design is aimed at making data structures used in the process computation independent of data structures used in the information interchange with other processes, but maintaining the flexibility to decide how to map one kind of structure over the other.

Finally, to fulfill the requirement for an independent communication library, we have defined a *communication class* to encapsulate the communication library functions. This communication class offers a standard interface to PVM and MPI that is used by the *communication management class* and therefore hidden to programmers. This class is automatically generated by a tool (CCG – Communication Class Generator) that uses *data configuration* information given by the programmer.

Master/Worker implementation of Nbody problem

The framework has been used to develop a Master/Worker brute force Nbody application [5]. This application involves processing the force iteration between a set of *n* bodies in a 2D space for each instant of time. In every instant, the new position of each body is calculated and the current velocity is updated by the influence of the other bodies. This application was chosen because it can scale for a high number of workers and the computation effort is spitted in framework iterations.

The framework provides an infrastructure for developing an application under the Master/Worker paradigm. To develop a master/worker application the user has to extend the two Master and Worker base classes. Each of these two classes has virtual C/C++ methods that should be overwritten by the user code. Communication between the user code and the framework is made by polymorphism and the attributes inherited in the sub classing process.

The first step towards building a new application is to execute call the "create.sh" script file that generates a directory tree containing all the files needed by passing the application name. This creates a "master" and a "worker" directory containing the specific master and worker code provided by the framework, and skeleton classes that should be filled with the user code. The user should only modify the "_mymaster.cpp" and "_mymaster.h" in the "master" directory, and "_myworker.cpp" and "_myworker.h" in the worker directory.

Another directory created is the "struct", and this contains the user structures including the "myinstruct.h" and "myoutstruct.h" files. The user should edit these files and place C/C++ code definition of the input and output data structures. After coding the input and output structures, the user needs to code the master and the worker class. The framework used supports iterations and provides synchronous and asynchronous execution. This affects the way the master distributes the tasks between the workers.

Framework execution involves a series of events that should be coded by the user. These basic events are master initialization ("_M_Initialize"), worker initialization ("_W_Initialize"), master data partition ("_M_Partition"), worker do work ("_W_DoWork"), worker final work ("_W_FinalWork"), master data recovery ("_M_Recover") and master final work ("_M_FinalWork").

In the NBody implementation, the function "_M_Initialize" is used to allocate memory for the array of bodies, and each body position and velocity are initialized. The _W_Initialize work member function is used to initialize the result buffer, an array for storing the computation result data. Each input structure is the set of bodies and the range of processing, which tells the worker the subset of bodies to process. Task generation is based on the number of available workers and the division is performed equally, each worker receives approximately the same amount of work ("_M_Partition"). The

"_W_DoWork" of worker consists of processing input structures. The input structure process consists of calculating, for each body within the defined range, the result acceleration based on the mass and the distance between the bodies. Result acceleration changes the current velocity, direction and position of the bodies. The process output result is an output structure containing the new position and velocity of the processed bodies. After processing each input structure, the corresponding result is placed in an output structure and stored in a custom worker attribute called "spaces". The collection of results is queued in the "_W_FinalWork" function and the "spaces" attribute is empted. The worker member function used to recover the data was "_W_DoWork", and the code simply copies the result content to the common body list structure. The "_M_FinalWork" function is used to store the result body list values to file.

3. PERFORMANCE MODEL FOR THE NUMBER OF WORKERS

In this section, we present the problem of determining a suitable number of workers for an 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 fact, these kinds of application perform similarly to a balanced M/W application with the same total processing time and the same global communication volume, as shown in [1].

For this analysis, we have assumed 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.
- ? = communication cost per byte (inverse bandwidth), in ms/byte.
- V = total data volume, in bytes.
- n = current number of workers in the application.
- Tc = total computing time (S tci)
- 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 to apply the performance model associated to an M/W application are:

- tl and ? 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.
- Sizes of messages that the master sends to workers and when it receives results from them, in order to calculate the portion of the data volume sent to workers (p*V) and the portion received from them ((1-p)*V).
- The time the workers spend on each task has to be measured in order to calculate the total computing time (Tc).

We have shown in [1] the analysis performed in order to construct the performance functions associated to this kind of application and the expressions for calculating the optimal number of workers for different situations.

In a Master/Worker application the Master initially distributes data among workers, and then those workers make some processing on this data. Finally, each worker sends its processing results back to the Master. We have called iteration to this process, and we have defined the expressions to calculate the total iteration time given different conditions:

$$Tt = 2*tl + l*\sum_{i=0}^{n-1} v_i + tc_i + l*v_m \quad \text{if } ((\mathrm{tl} \le l*\mathrm{v_i}) \text{ and}$$

$$(\mathrm{tl} + l*\sum_{\mathrm{i=0}}^{\mathrm{n\text{-}1}} \mathrm{v_i} > 2*tl + l*v_i + tc_i + l*v_m))$$

Or

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

Or

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

if (synchronous protocol) and

$$(\mathrm{n}*\mathrm{tl} + l*\sum_{\mathrm{i=0}}^{\mathrm{n\text{-}1}} \mathrm{v_i} > 2*tl + l*v_i + tc_i + l*v_m)$$

In [1] we assumed that the total data volume was constant, so increasing the number of workers implied smaller messages. Consequently, considering that tci = Tc/n, v_i = p*V/n (some 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 the above expressions as:

$$Tt = \frac{(2*tl + l*V*p)*n + Tc + l*(1-p)*V}{n}$$

$$if\ ((tl \le l*\frac{p*V}{n})and$$

$$(n \le \lceil (l*V + Tc)/(l*p*V - tl) \rceil)) \qquad \textbf{(1)}$$

Or

$$Tt = n*tl + \frac{Tc}{n} + tl + l*\frac{V}{n} \quad (\text{if } \mathrm{tl} > l*\frac{\mathrm{p*V}}{\mathrm{n}})$$

$$\text{if}(\mathrm{tl} > l*\frac{\mathrm{p*V}}{\mathrm{n}}) \qquad \textbf{(2)}$$

Or

$$Tt = (n+1)*tl + l*p*V + \frac{Tc}{n} + l*(1-p)*\frac{V}{n}$$

if (synchronous protocol) and **(3)**

$$(\mathrm{n}*\mathrm{tl} + l*\mathrm{V} > 2*tl + l*{}^{V}\!/_{n} + {}^{Tc}\!/_{n})$$

If we calculate dTt/dn = 0 for expression (2) and (3) we will obtain the expressions to calculate the number of workers needed to minimize Tt, which is:

if protocol is synchronous and

$$\mathrm{n}*(\mathrm{tl} + l*\mathrm{V/n}) > 2*tl + l*\mathrm{V/n} + Tc/\mathrm{n}$$

$$N_{opt} = \sqrt{{}^{((1-p)l*V+Tc)}\!/_{tl}} \qquad \textbf{(4)}$$

if protocol is asynchronous and

$$(\mathrm{tl} > l*\frac{\mathrm{p*V}}{\mathrm{n}})$$

or

if protocol is not synchronus and $(tl \leq l * p * V / n)$ *and* $(n \leq \lceil (l * V + Tc)/(l * p * V - tl) \rceil$

$$N_{opt} = \sqrt{(l * V + Tc)/tl} \qquad \textbf{(5)}$$

As we said before, these expressions are valid when the total amount of data remains constant, but in the NBody application the whole data is sent to each worker, which means that data volume depends on the number of workers. As a consequence, expressions (1) and (2) become:

$$Tt = \frac{(2 * tl + l * n * V_m) * n + Tc + l * V_w}{n} \qquad \textbf{(6)}$$

and

$$Tt = n * tl + l * V_m + \frac{Tc}{n} + tl + l * \frac{V_w}{n} \qquad \textbf{(7)}$$

And expression (5) becomes:

$$N_{opt} = \sqrt{(l * V_w + Tc)/tl} \quad \text{if} (\text{tl} > l * \frac{\text{V}_\text{m}}{\text{n}}) \qquad \textbf{(8)}$$

$$N_{opt} = \sqrt{(l * V_w + Tc)/l * V_m} \quad \text{if not} \qquad \textbf{(9)}$$

4. MATE

In this section we present MATE (Monitoring, Analysis and Tuning Environment) [2, 3] that provides dynamic automatic tuning of parallel/distributed applications. During run-time MATE automatically instruments a running application to gather information about the application's behavior. The analysis phase receives events, searches for bottlenecks by applying a performance model and determines solutions for overcoming 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. To modify the application execution on the fly MATE uses the technique called dynamic instrumentation [4].

MATE consists of the following main components that cooperate among themselves, controlling and trying to improve the execution of the application:

- Application Controller (AC) – a daemon-like process that controls the execution of the application on a given host (management of processes and machines). It also provides the management of process instrumentation and modification.
- Dynamic monitoring library (DMLib) – a shared library that is dynamically loaded by AC into an application process to facilitate instrumentation and data collection. The library contains functions that are responsible for registration of events with all required attributes and for delivering them for analysis.
- Analyzer – a process that carries out the application performance analysis, it automatically detects existing performance problems "on the fly" and requests appropriate changes in order to improve the performance of the application.

An important issue is the representation of knowledge of the performance problems that we can utilize when optimizing an application. In MATE, we use the following terms for knowledge: measure points, performance model, tuning points/actions. A measure point is a location in a process where the instrumentation must be inserted in order to provide measurements. A performance model consists of activating conditions (conditions in the application behavior considered to be a bottleneck) and/or formulas that model the application, allowing determination of the optimal conditions. Tuning points are the application components that must be changed to improve performance. Tuning action represents the action that must be performed on a tuning point. The knowledge required to represent the performance model of an application bottleneck is specified in a component called a "tunlet".

Tunlet implementation

To dynamically tune the number of workers the cooperative approach must be chosen since it is required to have certain information about the application. We implemented a specific tunlet using the knowledge provided by the framework presented above. The application is based on iterations where all processes repeatedly perform all operations. In every 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. Worker processes calculate the results and send them back to the master.

The tunlet that optimizes the number of workers requires run-time monitoring of the functions responsible for exchanging messages (sending and reception functions implemented in the framework). In particular, for these functions every entry or exit in the master process and in all worker processes is monitored. Instrumenting these functions and measuring the amount of data sent to the workers and received by the master, the total computational time of workers, the network overhead and bandwidth we are able to perform, are all measurements required by the performance model presented in Section 3 (expressions (6) and (8)).

During execution, the application should be aware of the current number of workers. The model is evaluated after every iteration when all measurements gathered from that iteration are available. If the computed optimal number of workers differs from the current number of active workers, the associated tuning procedure is invoked. In this case, we require the application to be prepared for the potential changes. For this, the framework provides a specific variable that represents the current number of workers, embedded in the developed application. MATE will change this variable by automatically updating its value related to the current environment conditions, and this new value will be used in the next iteration. 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, new machines (processors) are required for them. There is no sense in running a new worker on the same machine as one where another worker is already running. In such a situation we would not gain anything since the CPU time is divided between both workers, unless the target machine is multiprocess.

5. EXPERIMENTAL RESULTS

This section presents the experimental results obtained by applying the tuning environment to a base framework for Master/Work application. To conduct the experiments, we selected a brute force 2D N-Body implementation presented in Section 2. Experiments were conducted on a cluster of homogenous Pentium 4, 1.8 Ghz, (SuSE Linux 8.0) connected by 100Mb/sec network. Each experiment was performed many times and the average of the execution time for the application was calculated.

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, in particular an gradually ascending load pattern and a variable load pattern; then for each one of they we executed the application with several fixed number of workers (1, 2, 4, 8, 16, 19) and also under the control of the MATE tuning environment where the number of workers is adapted dynamically. In every scenario, each Worker was executed in an individual machine.

We have conducted our experiments in two scenarios:

- In the first scenario, N-Body was executed on different number of workers, without any tuning.
- In the second scenario the application 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 and Figure 2 summarize the experimental results to the variable load pattern. They show the execution time of N-Body application considering different number of workers and the execution time of N-Body under MATE. In each scenario N-Body was executed with the controlled variable load pattern.

#workers	1	2	4	8	16	19
Execution Time	64,49	34,61	18,09	10,37	11,83	15,49
N-Body + MATE		Starting with 1 worker and then tuning				
Execution Time		10,92				

Table 1. Execution time of N-Body (in seconds) in different scenarios under a variable load pattern.

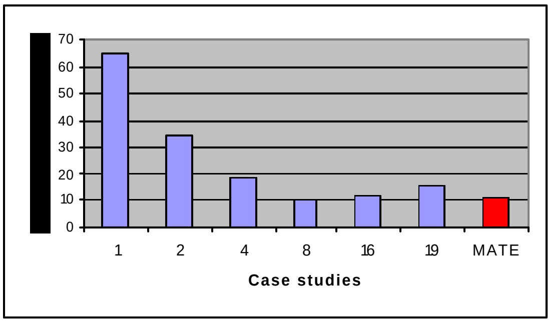

Fig. 2. Execution time of N-Body under a variable load pattern using different number of workers and MATE.

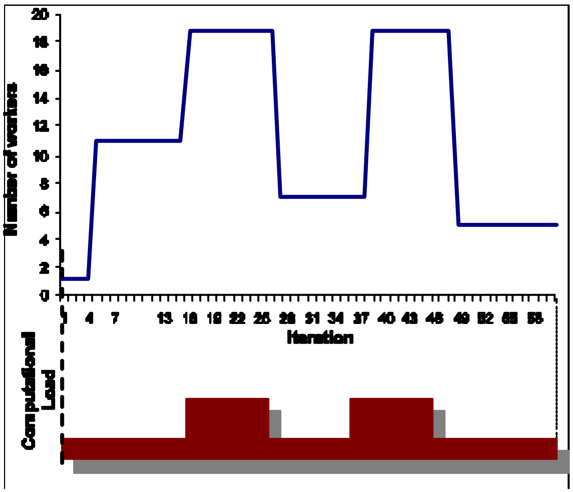

Fig. 3. Number of workers adaptation along the N-Body execution under MATE and a variable load pattern.

Table 2 and Figure 4 summarize the experimental results to the ascending load pattern. They show the execution time of N-Body application considering different number of workers and the execution time of N-Body under MATE and similarly as in previous experiment, in each scenario N-Body was executed with the controlled ascending load pattern.

#workers	1	2	4	8	16	19
Execution Time	73,23	37,11	19,28	11,76	13,05	14,50
N-Body + MATE			Starting with 1 worker and then tuning			
Execution Time			12,46			

Table 2. Execution time of N-Body (in seconds) in different scenarios under an ascending load pattern.

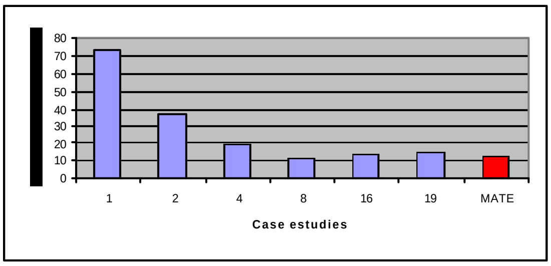

Fig. 4. Execution time of N-Body under an ascending load pattern using different number of workers and MATE.

In both experiments, N-Body while executing 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 seen in Figure 3 for variable load pattern and in Figure 5 for ascending load pattern. Then, as the time passed, load patterns vary and the number of workers in the application is adapted to use in each moment the number of workers needed to achieve an optimal performance. Notice that responses to the changes in the load pattern are introduced some iterations later in the application (normally one or two iterations). The analysis is done collecting data of one iteration and MATE introduces the required modification on the following iteration. Sometimes, in front of considerable changes in the load system, it could be needed an additional tuning because the tunlet can evaluate the model with estimated optimal number of workers and readjust it. That is the case of the first and second adaptation in Figure 4.

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

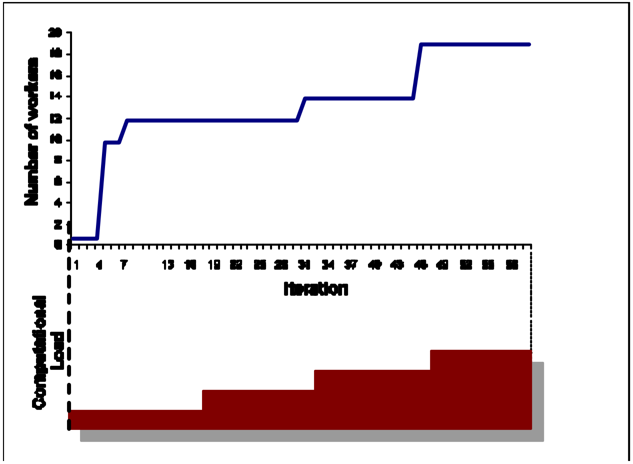

Fig. 4. Number of workers adaptation along the N-Body execution under MATE and an ascending load pattern.

6. CONCLUSIONS

Development of efficient parallel/distributed applications may be a difficult task for non-expert programmers. Tools must be provided that help a user in the development phase and provide automatic tuning of such applications.

In this paper we have described the framework for developing Master/Worker applications and the dynamic performance tuning tool. The framework facilitates the development of the application, hiding the low level details and performance tuning can successfully be carried out on the fly. Using this environment, programmers can design the application in quite a simple way, and do not need to worry about performance analysis or tuning, because dynamic performance tuning automatically takes care of these tasks.

The performance model for evaluating the optimal number of workers has been integrated in the MATE environment by the corresponding "tunlet".

We have conducted experiments with the parallel/distributed application developed using the presented framework and then tuned by the MATE environment. We have proved that it is effective and profitable. Running the application under MATE control has allowed for adaptation of the behavior to the existing conditions and improvements in performance.

7. REFERENCES

1. 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.
2. Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E. "Dynamic Performance Tuning of Distributed Programming Libraries". LNCS, 2660, pp. 191-200. 2003.
3. Morajko, A., Morajko, O., Margalef, T., Luque, E.. "MATE: Dynamic Performance Tuning Environment". LNCS, 3149, pp. 98-107. 2004.
4. Buck, B., Hollingsworth, J.K. "An API for Runtime Code Patching". University of Maryland, Computer Science Department, Journal of High Performance Computing Applications. 2000.
5. Wilkinson, B., Allen, M.: "Parallel programming - Techniques and Applications Using Networked Workstations and Parallel Computters". Pearson Prentice Hall. Second Edition. ISBN 0 13 140563 2. 2005.

Received: Jul. 2005.