Using implicit fitness functions for genetic algorithm-based agent scheduling

Sankaran Prashanth, Daniel Andresen
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS USA

Abstract

In a distributed network, servers have various capabilities that make them more suitable for certain tasks. Mobile agents move from server to server based on various scheduling algorithms. Choosing a server for a particular agent can be difficult for many users. We present the design of a system which uses a genetic algorithm (GA) to "learn" the type of server best suited to an agent based on parameters such as bandwidth, latency, and CPU availability supplied by the Network Weather Service (NWS). Unlike typical GA implementations, we use an implicit fitness function defined as the amount of work done per unit time on the actual servers. We also use a 'hinting' system to further improve results through allowing agents to share information on server performance for particular agents. We provide experimental results indicating that the GA performance improves significantly over time, and provides a significant advantage over round-robin scheduling.

Keywords: *genetic algorithms, mobile agents, scheduling*

1 Introduction

Mobile agent systems provide the mechanisms for moving code from machine to machine across the Internet, based on various scheduling mechanisms ranging from direct user direction to dynamic load balancing. For systems such as the Aglets mobile agents framework from IBM, users typically guide the agents manually [5]. For other scheduling systems, users are expected to indicate in advance the resource needs of their computation, which may be difficult or impossible [7, 3].

To select an optimal server for a task, a programmer needs to have information on the performance parameters of a server. When a user has information on these performance attributes for all the servers, an appropriate selection can be made using that information. A network-monitoring tool like Network Weather Service can obtain this information. This tool runs processes on servers that continuously monitor various performance parameters of the servers and the network and store the data at some central location from which all the information can be accessed by any user [8].

A genetic algorithm is used to work on multiple attributes, where each attribute is a performance parameter of a server. Thus a genetic algorithm can run crossovers and mutations on various chromosomes representing different servers to create a new child that would best represent the set of server performance parameters for the task being considered. This new child can be used as a benchmark along with other rules for the selection of a server from the available ones as the closest match to the child.

For genetic algorithms a "fitness" function is required to evaluate the chromosome, which typically must be custom tailored for each problem. The fitness function for a CPU-bound process, for instance, would look quite different than for an agent requiring extremely low latencies. Developing explicit fitness functions for chromosome evaluation can be a difficult task for users[4].

In this paper, we combine the learning capabilities of genetic algorithms with the Grid distributed information infrastructure provided by the Network Weather Service to provide an adaptive scheduling

system for the Aglets system. We utilize a 'hinting' server to allow agents to exchange information on server performace. We also provide an alternative to the explicit construction of fitness functions by using actual agent performance as an *implicit* fitness function, evaluating chromosomes by their similarity to the chromosome of the servers offering the best actual performance.

The paper is organized as follows: Section 2 discusses the design and integration of genetic algorithms and Aglets; Section 3 gives the experimental results; and Section 4 discusses related work and conclusions.

2 Integrating Aglets and GA-based scheduling

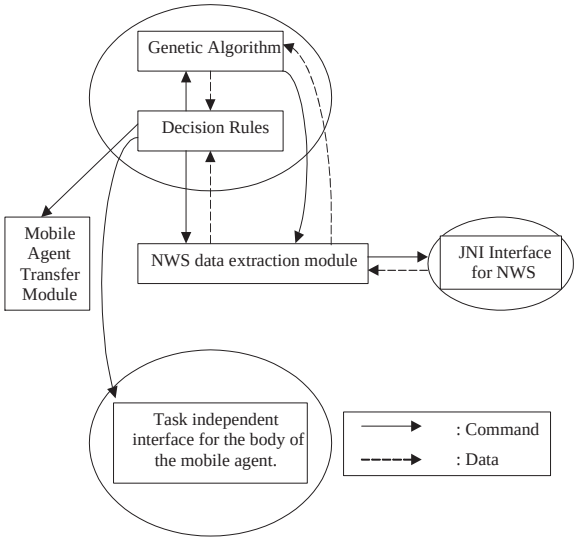

Figure 1. Agent architecture

Four parameters constitute a chromosome, corresponding to the four performance parameters of servers supplied by the NWS. The chromosome is a string of four substrings, each of which is the representation of a parameter values. The four parameters in the implementation are percentage CPU availability, number of CPUs, network bandwidth, and network latency. The fitness value is not a function of the parameters but the task performance on the server, namely, the number of work units executed by the task per second on the server. The intent of the GA is to generate one child chromosome that represents the GA-generated ideal characteristics for a server to give best performance.

The solution has a central agent as the parent of all the mobile agents which acts as the central point of information as well (Figure 1). The central agent receives the information on the task that needs to be executed and some static information regarding the available servers on the network. This information presently includes the location of the NWS NameServer process, the NWS MemoryServer process, NWS Sensor processes, the Aglet servers (Tahiti Server) and the number of CPUs on the machines being considered for the experiments.

The parent agent obtains all this information, creates the mobile agents, one for each task to be executed, assigns a task to each agent, passes the server information to them, and then leaves them to work autonomously in the network. In the meantime, all the mobile agents send important performance information to this parent agent. This information includes the present values of the performance parameters of the server on which the mobile agent is currently working (its chromosome) and the performance of the task on the server in terms of number of work units done per second and the location of the server. The information on performance parameters of each of the servers is obtained by calling a native method from the JNI interface that interacts with the appropriate MemoryServer of the NWS system to fetch the data (Figure 1). This information is stored and maintained by each of the mobile agents in their history. The parent agent sorts all the information from the mobile agents, and distributes only the relevant ones to the mobile agents, namely, the best two performances of any task on any server. Using this information, the mobile agents decide whether to continue working on the present server or to jump to a new server. On completion of the execution of their tasks, the mobile agents return to the central agent, submit their history of performance, and die.

The GA of the mobile agent is initialized by the history of the performance of the mobile agent. This history, as described earlier, contains data about the performance parameters of the server on

which the mobile agent was working at that time, the task performance obtained during that period when it was working on the server in number of work units done per second, and the location of the server. The mobile agent maintains two histories. One is its own private history, and the other is the combined history of the mobile agent and the information it receives from the parent or central agent. The GA is also initialized by the data it receives from the parent agent. The GA then generates a population from the above information obtained, each GA parameter being a performance parameter from the history of the mobile agents. The fitness value for each of the entries in the population is the value of the task performance on the server. Based on these fitness values the GA runs crossovers and mutations on one pair of entries selected as the best fit for the generation of the child. The resulting child is the GA-generated best server performance characteristics for the task being considered. The mobile agent then finds the current performance characteristics of the servers in the network, and selects the server with the chromosome "closest" to the generated optimum. The closeness value c is computed as

$$c = \sqrt{\sum_{i=1}^{4} \left(\frac{G_i - S_i}{G_i} \right)^2}$$

where G_i is the GA-generated parameter corresponding to the observed server parameter S_i.

Initially, a chromosome of the GA was represented as a string of 56 1's and 0's. fourteen characters represented each parameter: seven characters represented the integer part of the parameter value, and seven characters the decimal. The drawback of this implementation was that the number of CPUs, being a small integer value, did not have a good representation as a 14-character sub string in the chromosome. Hence the GA did not have a significant effect on the parameter. Additionally, our implementation was not normalized with respect to the parameters

The final GA chromosome representation is uses only four characters for the number of CPUs parameter assuming that the number of CPUs in the server's being considered does not exceed 15. This increases the odds that the GA will have a significant effect on the parameter.

The currently used decision rules for migration employ an average performance value for the decision making process. This average performance value is a monitor for the mobile agent, and it tries to continue increasing this value. This value is the average of the task performance values in the history of the mobile agent and is updated only if the current average is more than the previous average. The mobile agent jumps to the selected server only if the previous task performance of this mobile agent on the selected server is better than the task performance on the present server or if the previous task performance of this mobile agent in the selected server is better than the average value. The information for these conditions is obtained from the history that stores both the mobile agent history and the information obtained from the parent agent. The two conditions are satisfied only if the mobile agent has worked on the server before. If not, the mobile agent unconditionally jumps to the selected server. If none of the conditions are satisfied, then the mobile agent picks the second-best matching server to the child generated by the GA. The same decision rules are employed for this server as well. If again none of the conditions are satisfied for the second-best server, then the mobile agent stays in the current server. It can continuously stay in the same server three times only. The fourth time, the mobile agent searches its own history data for an unvisited server and jumps to any one of them randomly. If all the servers have been visited it then stays in the same server.

If the mobile agent jumps to a new server, it starts executing the task and, after a specific time interval, runs the GA that generates a new population using updated data from its own history and from the central agent to update the best available system for the execution of the task. And, based upon these new results the mobile agents decide whether to stay and continue execution or to move to the new available best system for the execution of the task. This continues until the execution of the task is completed by each of the mobile agents when they return back home to the central agent.

	Avg. Task Time	Avg. User Time	Min. Task Time	Min. User Time
GA Based CPU Task	2.65 min	4.72 min	1.87 min	2.78 min
RR Based CPU Task	5.5 min	9.24 min	3.71 min	6.43 min
GA Based Thread Task	2.45 min	5.33 min	2.02 min	4 min
RR Based Thread Task	2.89 min	5.21 min	2.5 min	4.36 min
GA Based Network Task	3.63 min	6.7 min	3.06 min	6.84 min
RR Based Network Task	3.7 min	6.4 min	3.21 min	5.22 min

Table 1. Task test results

3 Experimental results

To demonstrate the value of our system, we conducted a series of experiments to compare the performance of an agent using our scheduling algorithm with one using the round-robin scheduling algorithm. Since all machines were local to our location, bandwidth and latency were relatively constant over our switched, 100Base-T network. Significantly varying bandwidth and/or latencies would tend to increase the value of dynamic scheduling [2]. The servers used consisted of five Sun Ultra 5s (265Mhz.), one 9-processor Sun E4000(183Mhz.), a dual processor Sun SPARC 10/512(85Mhz.), a three processor Sun SPARC 20/514(70Mhz.), and a dual-processor Pentium-II-based Linux server(300Mhz.), all on a switched Fast Ethernet network.

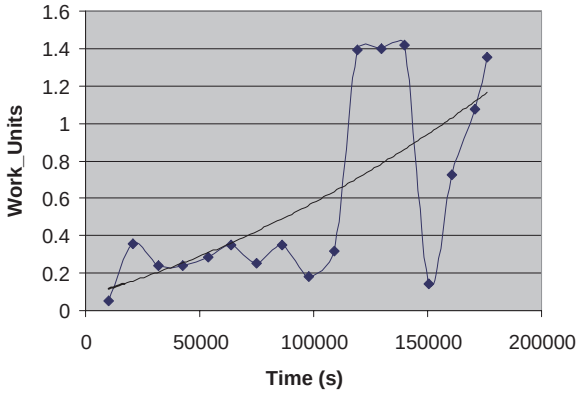

Figure 2. CPU-intensive GA performance over time

Our test applications consisted of three types of agents: CPU-bound, multithreaded, and network-based. These were intended to favor fast, multiprocessor, and low-latency/high-bandwidth systems, respectively. The CPU-intensive task simulates a simple genetic algorithm and performs extensive operations on strings and float values. The multithreaded task has three threads which continuously put values into a queue, and three threads which remove the value from the queue. Thread synchronization is the bottleneck for this task, affecting its execution time. The network task obtains one line at a time from a remote file, appends a number to the line, and copies it into another file on the same machine until the end of file is reached. This task is primarily latency-limited.

The tests are conducted on a pair of mobile agents at a time spawned by a central agent. The pair of mobile agents is initialized by an instance of a task each. The task is one of the three described above, namely CPU-intensive, thread-based, and network-based. The tests are conducted on each task, first using the genetic algorithm-based mobile agents and then using the round robin based mobile agents. All results are averaged over multiple runs.

The graph in Figure 2 shows the ideal case behavior of the genetic algorithm based mobile agents. It is clear that the mobile agent shows intelligent behavior by jumping to the server that outputs best performance. The trend line is the learning curve of the mobile agent, which, being linear, shows the good functioning of genetic algorithm in making selection of servers based on performance. The mobile agent shows a fairly randomized beginning where the agent just jumps to any server available studying its performance. The genetic algorithm

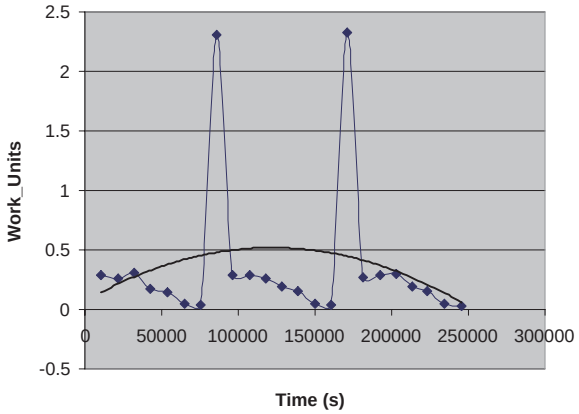

Figure 3. CPU-intensive RR performance over time

then takes over to make decisions. It jumps to an Ultra 5 because of its comparatively better performance. The GA realizes that the configuration of acrux gives the best performance among its known servers and hence stays there. After staying there for three time intervals, based on its decision rules the mobile agent jumps to the dual P-II as an unvisited server. Since the performance is very good, the GA tries to make the agent stay on that server. After three time intervals, however, it jumps to the E-4000, an unvisited server. It then returns to the dual P-II as its best server and remains until completion. In comparison to that, Figure 3 shows the normal behavior of the round robin based mobile agents, with substantially lower performance.

The overall performance of the genetic algorithm based mobile agents is better than the round robin based mobile agents in the case of CPU-intensive tasks. Contrary to our expectation, the thread-based task did not perform very well with the genetic algorithm-based mobile agents with respect to task time (Table 1). The thread-based mobile agents could not spawn multiple processes on the machines, and hence were not able to use the advantages of executing on a multiprocessor system.

The network-based task performed equally well with both round robin based mobile agents and the genetic algorithm-based mobile agents (Table 1). The network parameters for all the servers are similar and hence do not help the genetic algorithm in manipulating the parameter values in identifying the ideal network parameters for servers.

The mobile agents take a considerable amount of time in transferring themselves from one server to the other, giving the considerable difference between the task time and the user time.

4 Conclusions

We implemented a genetic algorithm that would work on four parameters, namely, percentage available CPU, number of CPUs, network bandwidth and network latency. The information for this is made available by the Network Weather Service tool that monitors the performance of servers and the network in the distributed system. Using this genetic algorithm, a distributed "hinting" system, and a few decision rules, the mobile agents are able to dynamically self-schedule without the need for an explicit *a priori* fitness function.

Other systems have explored runtime load balancing within a distributed system, although few within the context of Java-based mobile computation environments. The Plangent project adds scheduling capabilities to Aglets, but does not use genetic algorithms [6]. Recent work on mobile computing (e.g. [1]) uses bandwidth and data locality information to dynamically migrate computation, but requires a much greater degree of knowledge on the user's part than our work.

We have presented the test results that show, that the genetic algorithm based mobile agents give over 51% better performance in task time and over 48% better performance in user time for the CPU-intensive task as compared to the round robin based mobile agents. In contrast, the genetic algorithm based mobile agents give just over 15% better performance in task time over the round robin based mobile agents, and the round robin based mobile agents performed about 2% better in user time than the genetic algorithm based mobile agents for the thread-based task. From the observations this could be due to the inability of the mobile agents in taking advantage of executing the task on the mul-

tiprocessor systems, as the mobile agents could not spawn off threads as individual processes. Since the mobile agent has to work only on the performance output, the improvement in the overall performance of genetic algorithm based mobile agents is not satisfactory. Due to the overhead involved in the execution of the genetic algorithm in the genetic algorithm-based mobile agents, the performance of the round robin based mobile agents is marginally better with respect to user time.

The performance of the genetic algorithm based mobile agents as compared to the round robin based mobile agents has been proved to be definitely better in the execution of the CPU-intensive tasks. Though the genetic algorithm has shown only satisfactory performance improvement on the thread-based task and marginal performance improvement on the network based tasks, their performance degradation can be explained from the observations that have been made with respect to both the task time and user time. There is certainly a trade-off in between the performance gain and the overhead of execution of the genetic algorithm in the genetic algorithm based mobile agents. However, the test results show that the genetic algorithm-based mobile agents definitely perform well for some tasks, and and on par with round robin based mobile agents for others despite their inherent increased overhead.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–??, 1997.

[2] D. Andresen and T. McCune. Towards a hierarchical scheduling system for distributed www server clusters. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing(HPDC7)*, pages 301–309, Chicago, Illinois, July 1998.

[3] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile agents in distributed information revrieval. page Chapter 15, 1999.

[4] E. D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Pub. Co., 1989.

[5] D. B. Lange and D. T. Chang. Programming mobile agents in Java–a white paper. Technical report, IBM, Sept. 1997.

[6] A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. PLANGENT: An approach to making mobile agents intelligent. *IEEE Internet Computing*, 1(4):50–57, 1997.

[7] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[8] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, April 1998.