Machine Learning Approach to Tuning Distributed Operating System Load Balancing Algorithms

Dr. J. Michael Meehan
Computer Science Department
Western Washington University
Bellingham, Washington, 98225 USA
meehan@wwu.edu

Alan Ritter
Computer Science Department
Western Washington University
Bellingham, Washington, 98225 USA
ritter.alan@gmail.com

Abstract

This work concerns the use of machine learning techniques (genetic algorithms) to optimize load balancing policies in the openMosix distributed operating system. Parameters/alternative algorithms in the openMosix kernel were dynamically altered/selected based on the results of a genetic algorithm fitness function. In this fashion optimal parameter settings and algorithms choices were sought for the loading scenarios used as the test cases.

1 Introduction

The objective of this work is to discover ways to improve the openMosix load balancing algorithm. The approach we have taken is to create entries in the /proc files which can be used to set parameter values and select alternative algorithms used in the kernel to apply a load balancing strategy. The /proc files are writable from user processes with appropriate permissions which creates the capability to write a user land script to execute a battery of tests, evaluate a fitness function, and set the parameters to a different set of values determined by a genetic algorithm. In this fashion we can search for configurations which improve the performance of the system as defined by the fitness function. There are many criteria which could have been used for the fitness function; throughput, response time etc. For these tests we chose a fitness function which measures throughput.

2 Background MOSIX and openMOSIX

This work utilizes the readily available open source distributed operating system openMOSIX. The openMosix project [2, 11] is derived from MOSIX [3,5,6,10] and is released under an open source software license. MOSIX has a long history having been implemented on top of various versions of UNIX. The current version of MOSIX exists as a patch to a Linux kernel. At the time the work described here was carried out the stable version of openMOSIX was a patch to the 2.4 Linux kernel. The 2.6 Linux kernel openMOSIX patch was in the process of being developed.

2.1 Load Balancing in openMosix

The standard load balancing policy for openMosix uses a probabilistic, decentralized approach to disseminate load balancing information to other nodes in the cluster. [4] This allows load information to be distributed efficiently while providing excellent scalability. The major components of the openMosix load balancing scheme are the information dissemination and migration kernel daemons. The information dissemination daemon runs on each node and is responsible for sending and receiving load messages to/from other nodes. The migration daemon receives migration requests from other nodes and, if willing, carries out the migration of the process. Thus, the system is sender initiated to offload excess load. In addition, a third daemon, the memory daemon, is started at system initialization to obtain an ongoing profile of memory utilization.

The information daemon has two basic functions, the sending and receiving of load information. The sending part operates as follows. A periodic alarm, set to 1 second in the current implementation, is used to trigger the sending of load information. Two nodes are selected at random from two groups. The first group consists of all nodes that have contacted us "recently" with their load information. The second group is chosen from all nodes in the cluster.[1]

These two nodes are then informed of the sending node's current load. The load is calculated as follows.

The load calculation is attached to the clock interrupt routine. At each clock interrupt the number of running processes is added to an accumulator.

1 The list of nodes comes from a configuration file. It is possible to select a node which is not up. openMOSIX can also utilize an auto discover option in which case only nodes that have been in operation since this node booted will be added to the list. It is still possible to select a node which is not currently up.

When the number of clock interrupts completes a second the load is calculated. The accumulated load is first normalized to the CPU speed of this processor using the maximum CPU speed in the system versus the local node's calculated CPU speed.[2] The load is computed as a combination of the old load modified by a decay value added to the newly accumulated load number. The system then compares the accumulated load for the last second with a value representing the highest load it has recorded over any one second period. The maximum of the two is retained. If the current load is smaller than the current maximum load the new maximum load is computed to be the old maximum load times 7 plus the newly accumulated load all divided by 8. In this fashion, the maximum observed load decays away gradually if the current load continues to be smaller than the maximum.

The load information sent to the two randomly selected nodes is different from this load calculated for *internal use*. This is part of the strategy used to prevent *migration thrashing*. The actual load value sent to other nodes, known as the *export load*, is slightly higher than the internally calculated load. The export load value is a combination of the decaying maximum load value described above and a stabilizing factor. In addition, the export load is increased by the CPU utilized in the last second by any process *"recently"* migrated to this node. This is done to prevent receiving too many new processes too quickly. Finally a value is added which increases with the number of processes currently in the process table to make nodes with nearly full process tables less attractive than those with fewer processes.

The receiving portion of the information dissemination daemon performs the following actions. Upon receiving load information from another node the information is added to the local load vector. The standard implementation simply utilizes a circular queue of eight entries. The first entry is reserved for this node's internal load. Thus, the oldest information is overwritten by newly received information. After the new load information has been placed in the queue, a check is made to determine if there exists a node in the eight entries that has a lower load than this node. As we have seen, the export load sent by the other processors is padded somewhat when compared to our internally calculated local load. A target node for the migration is NOT selected at this point. Instead the *choose ()* function is executed to determine which process will be selected for migration. Determination of the process to be migrated is based on the following criteria.

Processes can be locked, meaning they are not subject to migration. This feature is accessible as an SVC and from the command line. If a process is still in its creation phase it is not considered by the *choose()* function. If a process is in the act of being transferred to another node it is obviously not considered. If a process is using a shared memory region it is not considered. If any process is found which has already been selected for migration the search is aborted. If a process has not accumulated enough CPU usage to reach *residency* on this node then it is not considered for migration. The residency period is set to an estimate of the amount of time it would take to migrate the process over the network. There is also a threshold value of CPU utilization which must be met for a process to be considered for migration. The attractiveness of a process as a candidate is based on its CPU use since we last visited its PCB looking for candidates. This is combined with a value which attempts to measure the process' contribution to the load currently on the machine. Processes which are "frequent forkers" are given an offset to make them more attractive for migration under the reasoning the once migrated they will continue to fork children thus spreading the load as they are bounced from one node to another.

Once the process to be migrated has been selected a flag is set in its PCB so that the next time the system attempts to dispatch this process it will instead be migrated. Just before placing a process into execution the dispatcher looks to see if the process is to be migrated. If it is, the *consider ()* function is executed. The *consider()* function runs in its own kernel thread and the dispatcher goes about its normal routine. The *consider()* function will select the target node for the migration. The target is determined by computing the *opportunity cost* for each of the eight nodes in the local load vector. For an explanation of the concept of an opportunity cost for migration target selection refer to the paper by Yair Amir et al. [1] For a simplified explanation of the way the opportunity cost is computed in openMOSIX consider the sum *(current cpu usage / maximum cpu usage) + (current memory usage / maximum memory usage)* for each node in the load vector. The *"marginal cost"* is the amount this sum would increase if the process was migrated to that node. The idea is that minimizing this marginal cost provides the optimal target node for migration. *The node with the least opportunity cost is selected.* An attempt is made to migrate the selected process to

2 The maximum CPU value is settable by the operator by placing a value in a file in /proc. It would be better it at initialization a node exchange values with each other node.

this node. The prospective receiving node may decline the migration in which case the next node in an array sorted by opportunity cost is tried. The local node is included in the target selection process. The local node is made less attractive by adding in offsets for situations such as nearly out of memory or the process table is nearly full etc. If the local node is selected then the process is simply "unmarked" for migration.

2.2 Observations

You will observe that whenever new load information arrives from another node, finding a node in the load vector (7 external nodes in the standard kernel) with a load less than this machine triggers the search of ***all*** process control blocks in the system. It is possible to do all the work to search all the PCBs to select a process and then not have anywhere to migrate it. It is also possible that the local node may end up with the least opportunity cost and so the selected process is not migrated. In this case all the effort of the *choose ()* and *consider ()* functions was for naught. In addition, all nodes selected as migration targets could refuse to accept the migration, although this is not likely.

It is import to understand that the system does not directly incorporate a lower bound for the difference between internal load and another node's load. This is handled by each node advertising a slightly modified version of its load rather than the value computed for internal use, the internal load versus the export load. Thus, there is in a sense a built-in offset between our load value and the numbers found in the load vector. Each machine believes each other machine is slightly busier than it really is. This built-in offset accomplishes two things. It prevents the long search of all process control block in the event that the difference between the local node and the remote nodes in marginal. In addition, it prevents, to some degree, migration thrashing which might occur if no retarding factor were applied.

3 Genetic Algorithm Fitness Function Parameter Space

You will observe from the description of the standard openMOSIX load balancing algorithm that there are a number of *"voodoo constants"* which appear in the equations used to regulate the load balancing algorithms. For each of these constants a /proc file entry has been created in order to test a range of values. Additionally, we provide a number of alternative algorithms to be tested in conjunction with the various parameter settings. Below we describe each parameter / algorithm choice dimension in the search space.

3.1 Reporting Frequency (MF)

The reporting frequency used in the standard openMOSIX load balancing algorithm is one second. This value is, of course, somewhat arbitrary. We turn the reporting frequency into a parameter that can be varied between 0.01 of a second and 2 seconds in increments of 0.01 of a second. The way the code is implemented in the standard kernel the load reporting frequency is intimately tied to the calculated load values. Every 1/100th second each node receives a timer interrupt and adds the current length of the run queue into an accumulator [11]. After ***t*** of these time intervals the local load is calculated and sent off to two randomly selected nodes. If ***t*** is too large, then the spread of information is too slow. On the other hand if it is too small then the reported loads are inaccurate and subject to transient spikes as the estimation of load is based on sampling the length of the run queue.

3.2 Amount of Hearsay Information (HEARSAY_INFO)

In one of the predecessors to the current MOSIX system [4], the load reporting messages included not only information about the sending node's load, but other information from its load vector as well. In openMOSIX, however, no hearsay information is reported due to the overhead of integrating the newly received values into the old load vector [5]. Recent work [8] has suggested that hearsay information may improve the response time in openMosix. We have thus made the amount of hearsay information a parameter of the search space. Note that it would be possible for each node to send a different amount of hearsay information. We have chosen not to allow this degree of variability in our current experiments due to the combinatorial explosion of the size of the search space this would cause and the difficulty in interpreting the results. In the current experiments, when the degree of hearsay information of varied the same value is applied to all nodes equally. This value varies between 1 and the number of entries kept in the internal load vector.

3.3 Load Vector Size (INFO_WIN)

The load vector size, λ, is considered a critical factor in the load balancing algorithm [8]. It must be large enough to facilitate rapid information dissemination, yet not so large that outdated information is propagated. It is widely believed that it should be tuned for each specific cluster of computers. In the original openMosix kernel, the load vector is a compile-time constant with a default value of 8 which allows for retaining 7 external load values.

We use the load vector size as a parameter of the

search space, and let it vary between 1 and the number of nodes in the cluster.

3.4 LOAD_TARGETS_ANY

This is the number of nodes to send load messages to every MF/100 second(s). This value ranges from 1 to half the load vector size. In the standard kernel it is always 2.

3.5 LOAD_TARGETS_UP

This is the number of nodes to send load messages to every MF/100 second(s) that we have talked to *recently*. This parameter ranges from 1 to half load vector size. The standard kernel attempts to pick one of the two nodes it reports to, to be one that has had a recent communication. The rational given for this is that it causes newly joined nodes to be more quickly assimilated into the system.

3.6 MIN_CHOOSE

This is the minimum number of tasks to look at in *choose ()*. The standard kernel traverses the entire task structure list. In our experiments the fraction of the total list to traverse can be varied. This parameter allows us to set a minimum number to be traversed regardless of the number of processes currently executing.

3.7 choose_frac

This is the fraction of the task list to traverse in *choose ()*, see above. The motivation for traversing only a fraction of the total processes comes from considerations similar to the reasoning behind *statistical best fit algorithms* used to search for storage area allocations in real memory variable partition storage allocation systems. Refer to any standard operating systems text for a discussion of this concept.

3.8 VV1

In order to prevent *migration thrashing* the load calculation is not allowed to change dramatically from one calculation to the next. The method used in the standard kernel to accomplish this is to multiply the old load by a fraction to *"decay it away"* and allow the rest of the load to be represented by the current load calculation. The standard kernel uses 7/8 of the old load and $1/8^{th}$ of the new value. In our experiments we allow these proportions to vary.

*load = (VV1/8)*old_load + (8- VV1)/8*new_load*

where VV1 varies from 1 to 7.

3.9 INVERSE_LOAD_REPORTING

This value determines whether we are using load reporting controlled by the MF value or are running an alternative algorithm known as inverse response ratio load reporting. [8]. This approach was implemented in the DICE distributed operating system in a slightly different form from what is implemented in our current experiments. The basic idea behind the inverse reporting frequency algorithm is that the busier a system is the less often it will notify other systems of its load. If a system is very busy then other nodes are not concerned with being up to date regarding its load, as that system has no excess load to offer other systems anyway. This approach also has the nice property that the overhead associated with the algorithm decreases the busier the system becomes. It also has the interesting property that an "infinitely busy" system is essentially a crashed node, as it will never report its load. This information is inferred from the lack of a load message. For further information concerning the inverse response ratio approach see [8].

3.10 The Linux /proc File system

New files were created for each of these values in the Linux /proc file system. Thus a user-space process, with appropriate permissions, can set and get the current value of these parameters by simply reading or writing a file. The ability to search the parameter space from user space is essential to this work, as the overhead of running the GA in the kernel would skew the results.

4. Hypothesis Representation and Evaluation

Hypotheses are represented as a set of values for the parameters of the load balancing algorithm. To measure the fitness of a given hypothesis we set the hypotheses' parameter values on each of the nodes in the cluster. Next we start up a fixed number of processes doing a fixed amount of computation. As each of these processes exits it records the completion times in log files. The fitness of a given hypothesis is then the reciprocal of the sum of all these times. Using the reciprocal gives us the property that more favorable hypotheses have a higher fitness value, which is useful for the roulette selection used in the GA's Crossover and Mutate operations.

The Genetic Algorithm used is as follows:

Generate a random population, P
while |P| > f do

Evaluate the fitness of each m 2 P
Write each m 2 P to the log file
Crossover (P)
Mutate (P)
Remove the r|P| least fit members from P
end while

4.1 Preliminary Runs

The approach described was run on a cluster of 11 homogeneous Pentium IV class machines for 48 hours using a subset of the parameters described above in order to test the approach. These preliminary results indicated that for that the stressors used and using only throughput as the fitness function we can improve on the standard openMOSIX parameters for the cluster used in the experiment. The stressors used in the preliminary runs were completely synthetic tests which were totally CPU bound processes. Successive spikes of 90 processes each were injected on two fixed nodes for repeatability from one iteration of the genetic algorithm to the next.

4.2 Subsequent Experiment Sets

The system was executed on a collection of 13 heterogeneous nodes using three different initial population sizes. For each of the three subsequent tests the experiment was configured to remove up to half of the population after each generation depending on how promising the member was. Next the GA performed the crossover and mutation phases. The experiments were set to terminate when the population sized was lass than or equal to four. Only three parameters were allowed to be varied for these three experiments. This is because it took approximately one week of cluster time to execute the experiment varying only three parameters across the search space. We wished to obtain some initial results before embarking on longer tests. The three parameter values varied in these tests were INFO_WIN, MF, and choose_frac. We are currently running the experiments with all parameters being varied and will in all probability have those results available by the time this paper is presented

The first of the three tests utilized an initial population size of 120. This test ran for 7 generations. The log of this and all other tests referred to herein can be accessed at this link.

The fitness function values, which are the reciprocal of the total of all process run times is shown below with the parameter values which produced them for the seventh generation.

INFO_WIN	choose_frac	MF	fitness
6	9/20	86	0.0005681818181818
6	12/20	46	0.0005646527385657 82
7	20/20	71	0.0005128205128205 13
8	12/20	86	0.0005197505197505 2

Table 1

Recall that the higher the value of the fitness function the better it is.

The second of the three tests used an initial population size of 200. This test required 8 generations with a final population of 4.

INFO_WIN	choose_frac	MF	fitness
11	13/20	40	0.000574 7126436 78161
11	3/20	62	0.000542 8881650 38002
13	9/20	51	0.000556 7928730 51225
11	13/20	51	0.000582 7505827 50583

Table 2

The third and final experiment used an

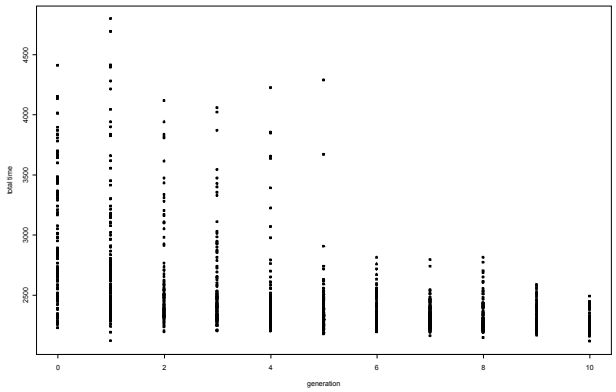
initial population size of 300. This test ran for nine generations with a final population size of three.

INFO_WIN	choose_frac	MF	fitness
12	13/20	221	0.000562 4296962 87964
11	6/20	36	0.000578 3689994 21631
11	13/20	221	0.000499 0019960 07984

Table 3

5. Conclusions

The following chart shows the elapsed execution times of the sum of all processes in the tests versus generations. It is clear from these results that we are able to improve the throughput of the system for the synthetic loads we utilized versus the standard kernel configuration. These initial results were obtained varying only three of the parameters available to us in the modified kernel. We anticipate that results of varying all parameters will produce even greater improvements.



References

1. [Ami00] AMIR Y., AWERBUCH B., BARAK A., et al. 2000. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. IEEE Tran. Parallel and Distributed Systems 7, 760-768.

2. [Bar] BAR M. Introduction to openMosix. From the openMosix web site, in the "Documentation" section.

3. [Bar85] A. Barak and A. Litman. “MOSIX: A Multicomputer Distributed Operating System.”, Software-Practice and Experience, 15(8):725-737, August 1985

4. [Bar85] BARAK, A. AND SHILOH, A. 1985. A Distributed Load-balancing Policy for a Multicomputer. Software: Practice & Experience 15, 901-913.

5. [Bar93] BARAK, A., GUDAY, S., WHEELER, R.G.The MOSIX Distributed Operating System - Load Balancing for UNIX. Lecture Notes in Computer Science Vol. 672 Springer 1993.

6. [Bar98] BARAK, A., LA'ADAN, O. 1998. The MOSIX multicomputer operating system for high performance cluster computing. Future Generation Computer Systems 13, 361-372.

7. [Mee95] *Distributed Interpretive Computing Environment (DICE): A Tool for Prototyping Distributed Operating Systems*, J. Michael Meehan, in *Teraflop Computing and New Grand Challenge Applications*, pg. 367-373, ed. Rajiv Kalia and Priya Vashishta (eds.) , Nova Science Publishers, Inc. 1995, ISBN 1-56072-247-9.

8. [Mee06] John Michael Meehan and A.S. Wynne “Load Balancing Experiments in openMosix”, International Conference on Computers and Their Applications , Seattle WA, March 2006

9. [May03] Maya, Anu, Asmita, Snehal, Krushna. MigShm: Shared memory over openMosix. http://mcaserta.com/maask/Migshm_Report.pdf. April 2003.

10. [Mos] MOSIX web site, http://www.mosix.org/

11. [opp] openMosix web site, http://openmosix.-sourceforge.net/