Dynamic Load Balancing in Multi-Agent Spatial Simulation

Bhargav Mistry and Munehiro Fukuda
Computing & Software Systems
University of Washington, Bothell
18115 NE Campus Way, Bothell, WA 98011
{bhargavm, mfukuda}@uw.edu

***Abstract*—This paper presents dynamic load balancing in a parallelizing library for multi-agent spatial simulation (named MASS). Our load-balancing algorithms calculate per-thread CPU load right after every function call and adjust a data size to be passed to each thread for the next function call. We implemented three different thread-based load-balancing algorithms, each using (1) an entire history, (2) a recent time window and (3) a slope of the CPU loads. The paper presents our implementation of these three algorithms in MASS as well as performance evaluation with two multithreaded applications: Wave2D and SugarScape. Furthermore, to demonstrate the slope-based algorithm's superiority to the other two, we compared them over a cluster of computing nodes.**

I. Introduction

It is more than 20 years that agent-based modeling received a popularity for simulating an emergent collective behavior of social and biological agents since Swarm was released from Santa Fe Institute [1]. Because most researchers are not computing specialists, their paramount focus has been placed on how to model their applications. However, the more precision of simulation pursued, the larger problem size required. This in turn means that simulations must handle a mega number of agents, which quickly exceeds computing resources provided by a single computer and thus needs parallelization. MASS, a parallelizing library for multi-agent spatial simulation is one of the tools to parallelize agent-based models (ABMs). The library composes a user application of multi-agents running on a distributed array that is mapped over a cluster system. Agents can migrate from one to another array element, which results in actual agent migration to a remote cluster node.

Of importance is how to map a distributed array over cluster nodes and CPU cores at each node. Unless agents migrate uniformly over such a distributed array, some particular elements may unreasonably receive more agents. Therefore, parallelization of ABMs needs some dynamic load-balancing (DLB) mechanism. Remapping a given array over a cluster system requires computing nodes to exchange array elements, whose cost may be more expensive than imbalanced computation. Based on this assumption, we first focused on DLB among CPU cores or threads. This is achieved by calculating the CPU load of each thread and comparing it with the other threads in their computation pool. Threads with the higher load then transfer their excessive amount of computation to those with the lower load. A load-balancing algorithm kicks in either periodically or when it senses load imbalance. There is a very critical tradeoff between the complexity of the DLB algorithms and the weight of the CPU loads that should be balanced. If a given DLB computation is heavier than an actual application load, it neutralizes merits brought by load balancing. In periodic load balancing, if the algorithm kicks in too frequently, it poses an overhead on the application itself. On the other hand, if the algorithm kicks in less frequently, the application runs into the risk of a prolonged imbalanced state and the resources utilization is deteriorated. In order to avoid these problems, DLB algorithms should be light in nature so that their execution cost is negligible.

This paper compares three thread-level load-balancing algorithms, each based on (1) an entire history, (2) a recent time window and (3) the latest-slope of CPU loads. In the following discussions, we simply call each algorithm the history-based, the window-based and the slope-based algorithm respectively. We have implemented these three algorithms in the MASS library where threads work on their respective slice of a given distributed array and dynamically move their slice boundaries for better load balancing. This paper demonstrates that, among these three algorithms, the slope-based algorithm works best due to its light-weight computation and lower complexity.

The rest of the paper is organized as follows: Section 2 reviews the conventional DLB algorithms and proposes a very simple slope-based algorithm; Section 3 explains our implementation of DLB algorithms both at the application and the MASS library levels; Section 4 compares the slope-based algorithm with the others; and Section 5 concludes our discussions.

II. Dynamic Load Balancing Algorithms

This section first reviews two conventional dynamic load-balancing (DLB) algorithms such as the entire history and the recent time window algorithms, and thereafter proposes the slope-based algorithm as a light-weight form of load balancing.

A. Conventional Algorithms

Most conventional algorithms are based on polynomial regression. It is defined as a form of liner regression in which the relationship between the independent variable x and y is modeled as an n^{th}-order polynomial as shown below:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n \quad (1)$$

The goal of regression analysis is to determine the best coefficients a_0 through a_n of the polynomial so that it projects

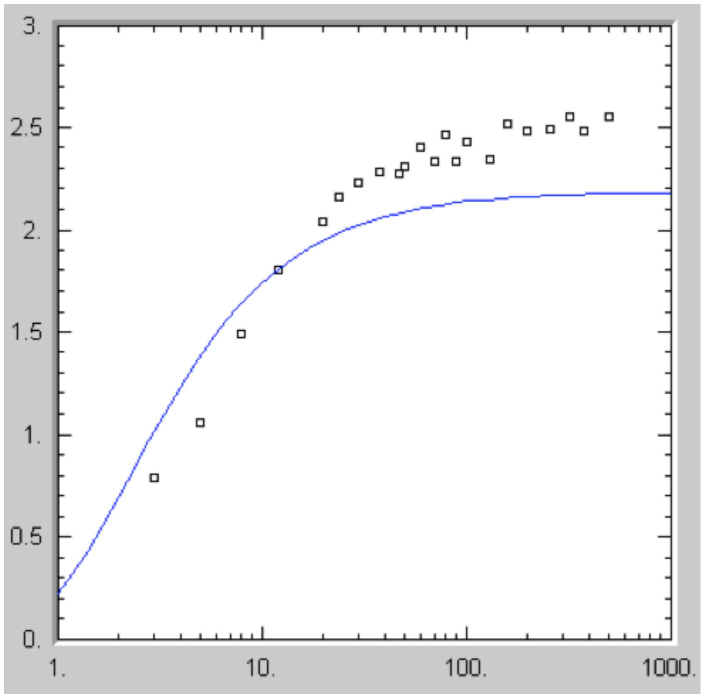

Fig. 1. An illustration of polynomial regression

the curve closest to a given history of (x, y) values (see Fig. 1). The polynomial is then used to estimate the value of y when a future value of x is given. In load balancing, variable x represents time t and variable y corresponds to the computation amount, (or more specifically thread computation) at time t. Polynomial regression is computed by expressing these (t, y) values into a matrix form and thereafter performing the Gauss-Jordan elimination.

1) History-based algorithm and prediction: The history-based algorithm takes into consideration a complete history of all previous (t, y) pairs from the launch of each thread. The algorithm measures each thread's CPU load y at each time t, adds it to the per-thread list of (t, y) pairs, recomputes coefficients a_0 through a_n, and finally estimates a new CPU load at time $t+1$. When agents move over a distributed array as we implemented in the MASS library, each thread takes care of agents on a different slice of the array. The larger slice the more computation. Therefore, a checkpoint will take place periodically to compare each thread with its neighbors in their CPU loads and to adjust their slice boundaries, so that the most loaded thread will get a narrower slice. Once a boundary adjustment is complete, the algorithm predicts the future CPU loads of all the threads. The prediction is calculated based on the complete previous history of CPU loads. After a prediction is made, thread slices are re-adjusted proactively for the thread with the highest predicted CPU load.

The history-based algorithm uses polynomial regression of 4^{th} degree to predict the future load [2]. Since the algorithm keeps adding a new load value to the data matrix used for the polynomial regression, its computation overhead gets increased monotonously.

2) Windows-based algorithm and prediction: The window-based algorithm is similar to the history-based algorithm except it works on a sliding window of history data [3]. The sliding window can have the last N number of thread-load captures to predict the next load value y for the future time-series candidate t. The algorithm uses the same 4^{th}-level polynomial regression adapted to the history-based algorithm.

Here is an example to describe how the window-based algorithm works: Let us assume that the load pool for thread t1 has the following values: (5, 2, 5.4, 3.3, 9.8, 10, and 7) for time series values (5, 10, 15, 20, 25, 30, and 35) respectively. In order to predict the future load value for time-series candidate 40, a sliding window of last N elements is chosen and a polynomial fitting is performed. If $N = 3$, the sliding window values would be (9.8, 10, and 7). Polynomial regression is performed on this window of data to get the 4^{th}-degree polynomial equation. This equation is then used to predict the future value y for time-series candidate 40.

Since the window-based algorithm uses the last N-element sliding window such that $N <$ the total number of history elements, the prediction will be less accurate than the history-based algorithm, however the algorithm works faster and its execution cost remains constant.

B. Proposed Algorithm

To address the problems with the conventional algorithms, we propose an algorithm called the slope-based algorithm. The following describes and differentiates our algorithm from the related work.

1) Slope-based algorithm and prediction: As the name suggests, our slope-based algorithm uses the line slope between two load values y_1 and y_2 on a graph, corresponding to time t_1 and t_2 where $t_1 < t_2$:

$$slope = \frac{y_2 - y_1}{t_2 - t_1} \tag{2}$$

If we assume that the slope remains constant in the near future, we can predict the next load value y_3 at time t_3, using the following formula:

$$y_3 = (slope \times (t_3 - t_2)) + y_2 \tag{3}$$

Fig 2 illustrates our algorithm. Focusing on slope C between time = 3 and 4, C1 is the predicted CPU load at time = 4.5. In similar to the history and widow-based algorithms, the slope-based algorithm periodically compares each thread with its neighbors in their CPU load to adjust their boundaries. Thereafter, all the threads' CPU loads are predicted using the above formula, and thread slices are re-adjusted for the most loaded thread.

2) Other slope-based algorithms: The rate-of-change load-balancing algorithm [4] considers several load-balancing phases such as when to initiate task migration, where to send tasks, how many tasks to migrate, and which tasks to migrate to different nodes over the network. The algorithm maintains a per-node load table whose information is collected through message exchanges among all nodes. It uses a slope-based design where loads are plotted on a graph and a line is drawn to connect two adjacent load values. The algorithm considers

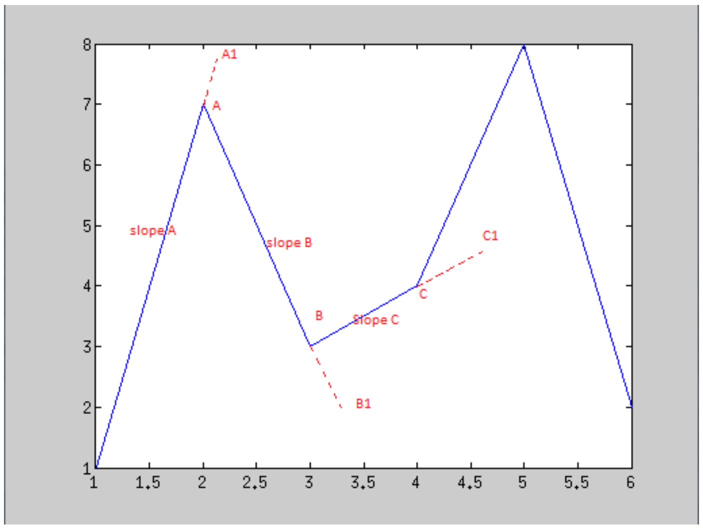

Fig. 2. Slope-based algorithm

transferring a task to another node if the next predicted value is going down below a pre-determined low threshold value. The algorithm looks at a slope that is going in a negative trend. A node below the low threshold (called a *sink*) sends a message to one of remote nodes that are listed in the load table as overloaded nodes (called *sources*). Then, the source node sends tasks to the sink node. After the task transfer is complete, the algorithm updates the corresponding load tables that maintain tasks to process as well as source nodes to send a message to in the future.

In contrast to the rate-of-change algorithm, our slope-based algorithm has the following advantages: load imbalance is adjusted between all neighboring threads by moving their slice boundaries; such adjustments are made proactively using predicted CPU loads; and our algorithm does not have to maintain any load tables.

III. Implementation

We have implemented the three DLB algorithms: (1) the history-based, (2) the window-based, and (3) our slope-based algorithms at both application and MASS library levels. We used the following two applications to measure their execution performance.

A. Applications

1) Wave2D: simulates wave dissemination using Schrödinger's wave equation. The simulation starts with dropping a bucket of water onto the center of water surface in a larger square container. As illustrated in Fig. 3, waves propagate toward the walls, and thereafter repeat bouncing and interfering with each other.

2) SugarScape: observes social dynamics of artificial lives (modelled as agents) roaming over a two-dimensional space with two sugar mountains that attract these agents. Agents have some attributes such as their own metabolic rate, visibility range, and ability to consume sugar. Fig. 4 shows agents in search for sugar to survive through a simulation.

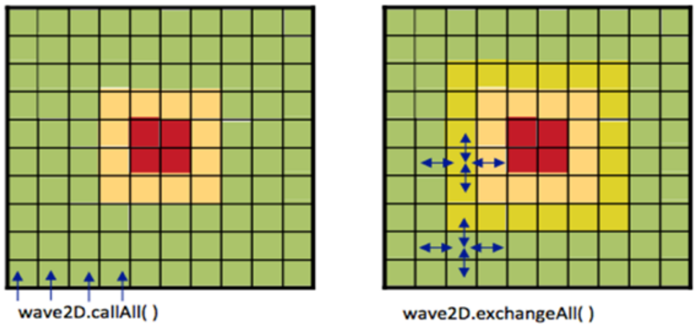

Fig. 3. Wave2D simulation in action

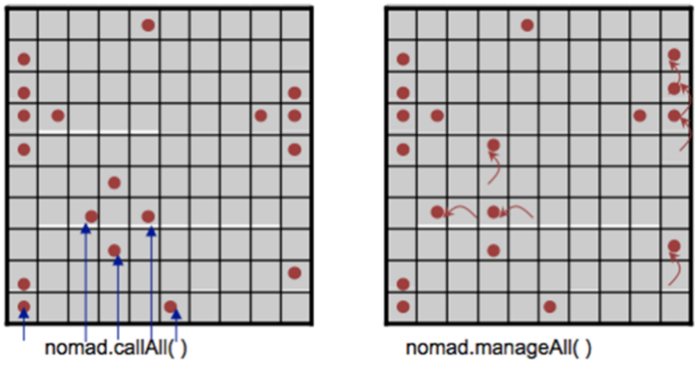

Fig. 4. SugarScape simulation using multi-agents

B. Application-Level Load Balancing

Both Wave2D and SugarScape use two-dimensional or $N \times N$ arrays. In wave2D, a wave height at element (x, y) at time t is computed from these heights at (x, y) and $(x \pm 1, y \pm 1)$ at time $t-1$ and $t-2$. Therefore, Wave2D actually distinguishes three $N \times N$ arrays, each at time t, $t-1$, and $t-2$. On the other hand in SugarScape, each array element (x, y) maintains the current amount of sugar and accepts at most one agent. If two agents try to move to the same cell, the agent with a lower identifier gets a preference to migrate to this location, which incurs more communication between (x, y) and $(x \pm visibility, y \pm visibility)$ elements.

To distribute such $N \times N$ arrays over a collection of threads, we divide them into vertical slices so that each of P threads receives a slice of $N/P \times N$ elements. For instance, if we distribute an $N \times N$ simulation space over four threads, each thread covers a slice of $25 \times 100 \times 3$ in Wave2D and 25×100 in SugarScape respectively. If the space is not divisible by the number of threads, which thus results in R remainders, each remainder is allocated to the first R threads. A simulation takes a form of repetitive updates of each array element. In each simulation cycle at time t, threads update the status of their own respective array slices by traversing their slice from the top to bottom rows, each scanning elements from left to right.

At the end of each simulation cycle, DLB algorithms are invoked to calculate the CPU time spent for each thread to process its own slice. For this purpose, we use the ThreadMXBean class that provides various APIs to measure JAVA thread performance. The getCurrentThreadCpuTime() API provides the time spend by a particular thread running on the CPU

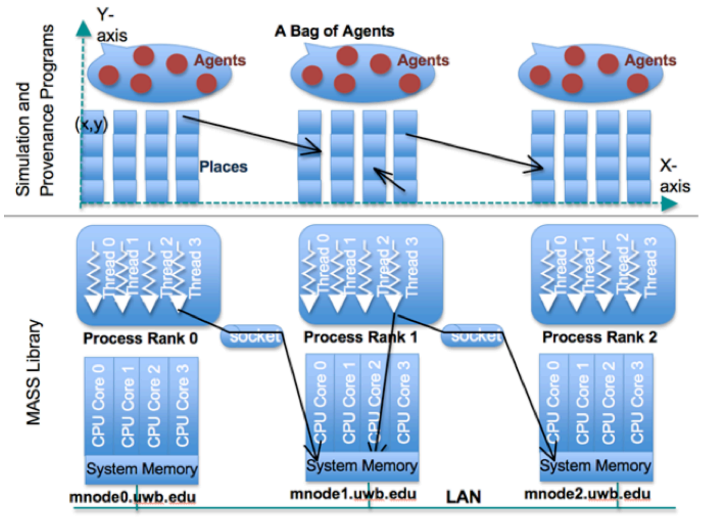

Fig. 5. The MASS library

in nanoseconds. A JAVA virtual machine (JVM) may disable CPU time measurement by default. The isThreadCpuTimeEnabled() and setThreadCpuTimeEnabled(Boolean) methods can be used to test if CPU time measurement is enabled and to enable/disable this measurement respectively.

To adjust the slice boundaries between neighboring threads, all the threads are set into the wait mode at the end of each simulation cycle. The main thread then adjusts slice boundaries based on a given DLB algorithm, and thereafter notifies all the other threads of newly adjusted slices.

C. System-Level Load Balancing

As a system-level implementation, we have integrated all the three DLB algorithms in MASS: a parallelizing library for multi-agent spatial simulation. The library supports multi-threaded multi-process parallel simulation. As illustrated in Fig. 5, *Places* and *Agents* are the key elements of the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element is called *Place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *Place* elements. *Agents* is a set of mobile objects that can reside on a *Place*, migrate to any other *Place*s with array indices (thus duplicating themselves), and interact with other *Agent*s through the local *Place* [5]. Parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster of multi-core computing nodes and are connected to each other through JSCH-tunneled TCP links. The library spawns the same number of threads as that of CPU cores per node. Those threads take charge of method call and information exchange among *Place*s and *Agent*s in parallel.

In MASS, threads at each node take care of their respective slice of *Place* elements. DLB algorithms can be implemented in MASS in the similar strategies as the application-level implementation. More specifically, each thread adjusts its own slice boundary in negotiation with its neighboring threads, based on their CPU load. In the following, we consider the single-node and the multi-node DLBs in the MASS library.

1) Single-Node DLB in MASS:

The MASS library is designed to run on a multithreaded single node or a cluster of multithreaded computing nodes. In an single-node execution, a MASS process is launched to execute multiple threads on a user-local node. The MASS library has two main functions: *callAll* and *exchangeAll*. The former executes the computation component in each of *Place* and *Agent* objects, whereas the latter transfers data between each *Place* and its neighboring objects. In general, Wave2D and SugarScape can be implemented in repetitive invocations of *callAll* and *exchangeAll*.

The DLB algorithms are implemented in such a way that the MASS library maintains a counter to keep track of how many times *callAll* and *exchangeAll* have been called. A consecutive call of *callAll* and *exchangeAll* increments this counter twice. A user-selected DLB algorithm is set to kick in when the counter reaches a predetermined value. (The value is configurable when the MASS library gets started.) The MASS library suspends all but the main thread that calculates the CPU time for all the threads and adjusts their slice boundaries. Thereafter, the main thread resets the counter to 0 and resumes the suspended threads.

2) Multi-Node DLB in MASS:

In a multi-node execution, a user-local node starts the master process that then launches slave processes remotely via JSCH. A simulation space, (i.e., *Places*) is then distributed over the master and slave processes. Each process further divides its own slice into sub-slices, each allocated to a different thread. The main function of a simulation program always runs within the main process and initiates *callAll* or *exchangeAll* that is propagated to the slaves and thus to all the threads over the system. This in turn means that each call of *callAll* or *exchangeAll* is invoked synchronously among all the threads.

In a multi-node execution, each process counts the number of *callAll* and *exchangeAll* invocations. Since their invocations are all synchronized, all processes agree with when to activate a user-selected DLB algorithm. However, each process executes the DLB algorithm independently by adjusting only its own thread boundaries. Note that boundaries are not adjusted across the processes. In other words, our multi-node DLB does not exchange any data, (i.e., *Places* or *Agents*) among multiple nodes.

IV. Performance Evaluation

We have compared the slope-based algorithm with the history-based and window-based DLB algorithms for their application and system-level execution. Our evaluation used the following computing resources:

Mode	Cores	CPU Clock	Memory	Network
Single node	8 cores	1600MHz	5GB	N/A
Multi nodes	4 cores	1800MHz	2GB	Giga Ethernet

A. Application-Level Load Balancing Performance

The total time to execute Wave2D and SugarScape was recorded as changing the number of threads from 2 through 8 for each of all the three DLB algorithms. To compare the results with the baseline performance, we also recorded the execution time of these applications without activating any DLB algorithms.

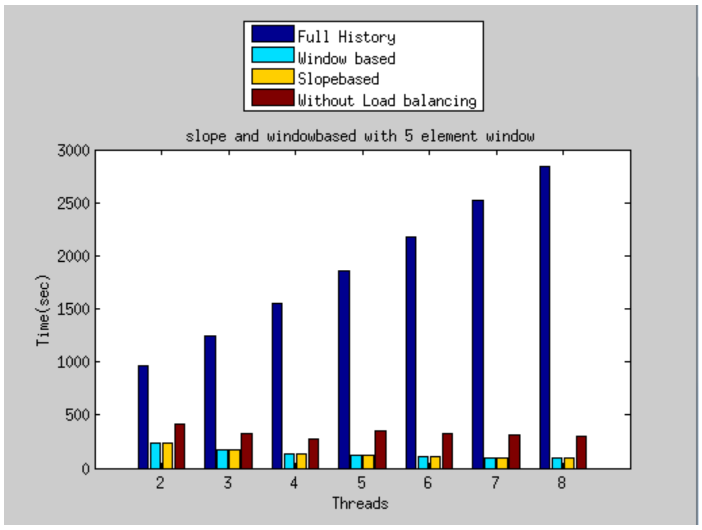

Fig. 6. Comparison of three DLB algorithms when running Wave2D

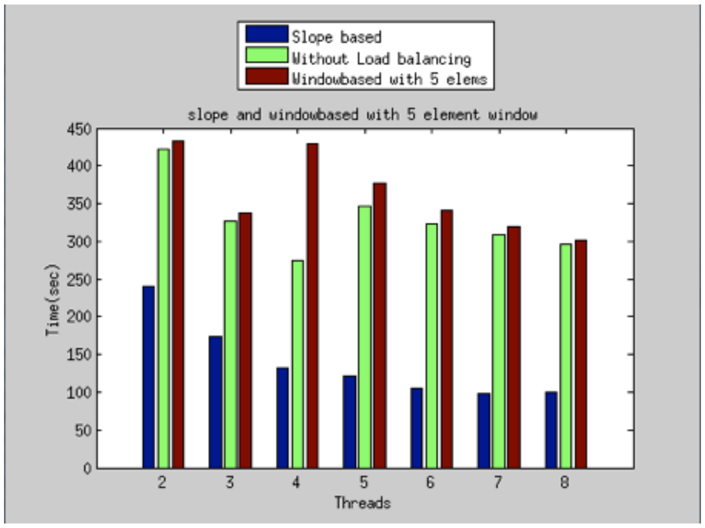

Fig. 7. Window-based with size 5 versus slope-based algorithms

Fig. 6 compares all the history-based, window-based, and slope-based DLB algorithms in Wave2D execution. Using 2 through 8 threads, the slope-based algorithm performed 0.9% through 2.4% better than the window-based DLB (where the window size = 3) and 4 through 28.5 times better than the history-based DLB.

Fig. 7 focuses on performance comparison between the sloped-based DLB and the window-based DLB where the window size = 5. The slope-based DLB ran 1.8 through 3.3 times faster than the window-based DLB with 2 through 8 threads.

Fig. 8 compares all the three DLB algorithms in SugarScape execution. Again, the slope-based DLB performed best, actually 5.3% through 32.6% better than the windows-based DLB as well as 5.5 through 11.3 times better than the history-based DLB in most cases. However, we must note that SugarScape execution without using any DLB algorithms actually ran fastest. This is because SugarScape is computationally lighter than Wave2D and cannot compensate any

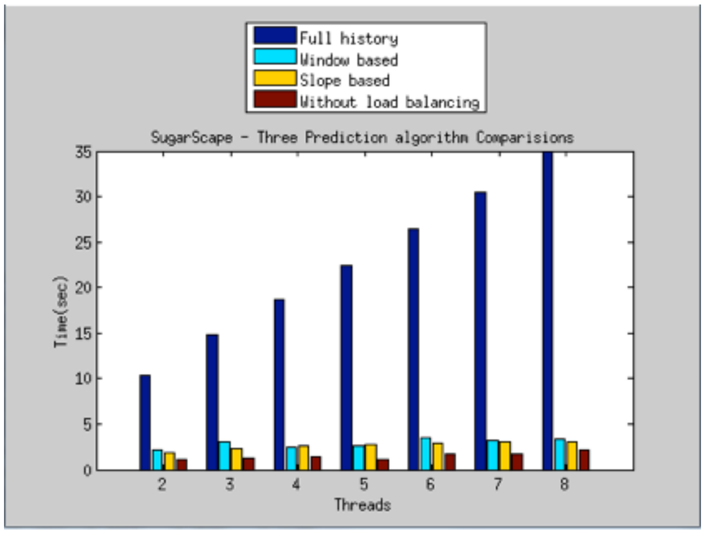

Fig. 8. Comparison of three DLB algorithms when running SugarScape

DLB overheads. It is of utmost importance to understand that a DLB algorithm will introduce certain amount of overhead and should be used for only applications that are computation intensive.

B. System-Level Load Balancing Performance

In a single-node execution, we used two to eight threads. In a multi-node execution, we recorded the execution of Wave2D and SugarScape over two cluster nodes, each using three and four threads. Figure 9 shows the performance of the three DLB algorithms that were integrated into the MASS library when we ran Wave2D with a single computing node. The slope-based algorithm performed best in most cases. It ran up to 13.0% faster than the window-based algorithm and up to 9.4% faster than the history-based algorithm. However with 6 and 8 threads, the slope-based algorithm was 3% to 4% slower than the other two. This is because a per-thread slice became too small to distinguish DLB algorithms and was effected by more thread management overheads.

In a multi-node execution, we used two computing nodes, each with two dual-core CPUs thus capable of running up to 4 threads. Fig. 10 shows the performance of the three MASS-integrated DLB algorithms when we ran Wave2D. The slope-based algorithm ran 23.0% and 15.8% faster than the window-based DLB with three and four threads per node respectively. It was 30.4% and 11.2% better than the history-based DLB under the same conditions.

C. Performance Summary

The history-based algorithm was unremarkable as compared to the other two. This is because of its increasing history data. Its overhead gets larger in proportion to the simulation time and neutralizes effects brought by DLB. When conducting our performance evaluation, we did not use any secondary storage to save the history data, which would be however required for a longer simulation run and further deteriorate the entire performance.

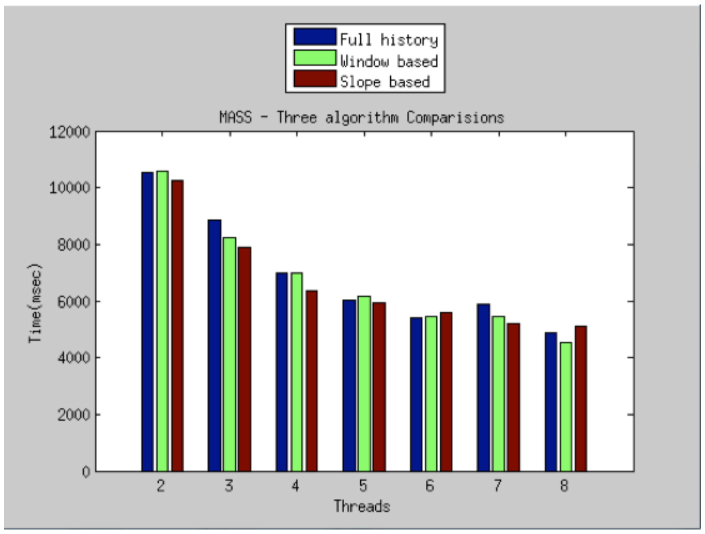

Fig. 9. Comparison of three DLB algorithms on a MASS single-node execution

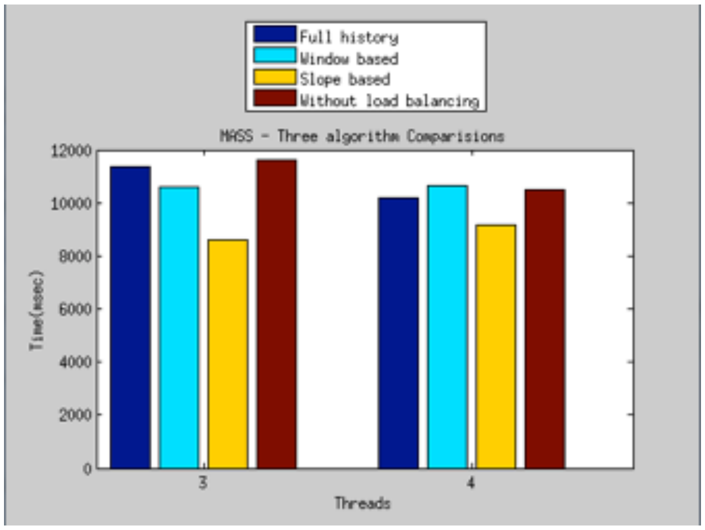

Fig. 10. Comparison of three DLB algorithms on a MASS multi-node execution

The window-based algorithm is quite competitive to our slope-based algorithm when it runs with its window size = 3. However, if its window size is increased to 5 for pursuing more accuracy of CPU load prediction, the logic itself gets too costly to maintain good execution performance.

The slope-based algorithm uses only the latest two load values and thus performs fastest. Although it is not so accurate for future load predictions as compared to the other two algorithms, its computational complexity is quite light and fits best to fine-grain parallel programs such as Wave2D.

V. Conclusions

Our main focus in this research was to develop a thread-level load balancing algorithm that is more efficient than the conventional DLB techniques. We compared the proposed slope-based DLB algorithm with the history-based and window-based algorithms for their execution performance. Our analysis has demonstrated that the slope-based algorithm performed best both at the application-level and the MASS library levels. Moreover, the slope-based algorithm worked 81% faster than the history-based algorithm and is 24% faster than the window-based algorithm with the MASS multi-node library when running Wave2D.

Currently the thread-level load-balancing is implemented in MASS Java, using its thread-management package APIs. This makes the algorithm restricted only to applications that use Java and JVM. To widely increase the usage of the DLB algorithms we discussed, we will port them both to MASS C++ and GPU, and examine their execution performance. We will also apply our DLB algorithm to more computation-intensive applications such as thermodynamics, fluid dynamics, and transport simulations.

References

[1] Swarm Summary, “http://savannah.nongnu.org/projects/swarm/.”

[2] Y. Inoguchi, “CPU Load Predictions on the Computational Grid,” in *6th IEEE International Symposium on Cluster Computing and the Grid*. Singapore: IEEE-CS, May 2006, pp. 321–326.

[3] L. Yang, I. Foster, and J. M. Schopf, “Homeostatic and Tendency-based CPU Load Predictions,” in *Proc. 17th IEEE International Parallel and Distributed Processing Symposium*. Nice, France: IEEE-CS, April 2003, pp. 42–50.

[4] L. M. Campos and I. Scherson, “Rate of change load balancing in distributed and parallel Systems,” in *13th International and 10th Symposium on Parallel and Distributed Processing*. San Juan: IEEE-CS, Apr 1999, pp. 701–707.

[5] T. Chuang, “Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library,” in *Master’s Thesis, University of Washington-Bothell, 2012*. Bothell, Washington: University of Washington, 2012.