Dynamic load balancing in MASS

Bhargav Mistry

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2013

Committee:

Munehiro Fukuda, Chair

Kelvin Sung

Michael Stiber

Hazeline Asuncion

Program Authorized to offer Degree:

Computing & Software Systems

University of Washington

Abstract

Dynamic Load Balancing in MASS

Bhargav Mistry

Chair of the Supervisory Committee:

Munehiro Fukuda, Ph.D

Computing & Software Systems

Dynamic Load Balancing (DLB) in Multi-agent Spatial Simulation (MASS) library is a thread-based load-balancing algorithm that calculates the CPU load, per thread, based on the computational time spent by CPU, using JAVA ThreadMXBean API. This thesis presents the system design, execution model of the three algorithms each based on (1) an entire history, (2) a recent time window and (3) a slope of the CPU load and our performance evaluation over two multi-threaded applications: Wave2D and SugarScape. MASS library single-node and multi-node versions are also evaluated with the objective that the Slope-based algorithm will be superior to the other two algorithms.

Table of Contents

Table of Figures

Chapter 1: Background

1.1 Overview of MASS library

Recent popularity of cloud computing have brought new opportunities of sensor based cloud computation that will facilitate on-the-fly analysis, simulation and prediction by feeding real time sensor data to cloud jobs. In agriculture, fruit crops can be protected from frost by predicting overnight temperatures using data detected by sensors placed in the farms. In the world of navigation, best route can be found by feeding real-time traffic data to traffic simulation models.

Swarm [7] is one of the many real-time applications that was developed to model a large number of social or biological agents and to simulate their emergent collective behavior in various simulated conditions. The application has been parallelized using threads. The research groups spend significant time parallelizing these types of applications using primitive libraries such as OpenMP, MPI and CUDA. Since they are not computing specialists, their paramount focus has been placed on how to model and speed up their own applications with GPUs, multi-cores, or a cluster depending on what resources are available to them. In order to solve these types of problems, a general purpose simulation environment on top of a cluster system was developed called MASS – a library for multi-agent and spatial simulation.

MASS composes of a user application with distributed array elements and multi-agents, each representing an individual simulation place or an active entity. All computation is enclosed in each array element or agent and data exchanges among entities are periodic. The agents migrate to a different array element that can be on a remote cluster node. Each cluster spawns a process that can contain multiple threads for computation. These MASS features can therefore allow a user application to mimic actual phenomenon such as epidemic among people in FluTE[9] and traffic jams in MatSim[8].

In today's modern world where cost effectiveness is of utmost importance, it becomes very necessary that the computing resources, whether they are virtual, physical or cloud instances, are utilized to its fullest caliber. In order to achieve this, MASS library needs an inbuilt thread based load balancing algorithm that monitors the CPU load for each thread and balances it as needed so that all the cores of a particular cluster node are used to its fullest.

1.2 Conventional Network Node Load Balancing

Conventional load balancing between network nodes can be defined as the ability of the network cluster to balance the amount of network traffic and/or tasks received by a particular node in the cluster. The cluster network switch uses algorithms such as round-robin to balance the amount of tasks that go to a node. The network load-balancing techniques ensure that no particular node is overloaded with tasks. For the systems to be most effective, any imbalance in the load should be sensed and mitigated as soon as they occur. Since the mitigation process is dynamic, it is termed as "Dynamic Load Balancing (DLB)". A DLB algorithm offers many benefits to an application including higher throughput, reduced mean job response time, higher reliability, more resource utilization and adaptability to fluctuation in load. Typically a DLB algorithm uses up-to-date system state information to make more effective load distribution decisions and generally it consists of but is not limited to 4 policies [4] such as (1) information policy, (2) transfer policy, (3) selection policy and (4) location policy that govern the actions of a load-balancing algorithm. The information policy is responsible for deciding when and what information needs to be collected from other nodes. The transfer policy determines if a node qualifies to either receive a new task or transfer a task from it to another node. The selection policy is responsible for determining which task needs to be selected, and the location policy determines which task needs to be transferred where.

A network-level load-balancing algorithm makes sure that a node is not over loaded but it is not capable of regulating load on each thread that executes in the node. In a multi-threaded parallel application, a computationally heavy task is divided among threads. It might happen that not all processor cores are utilized equally just because the load is not equally distributed among threads. This introduces a problem, namely, thread load imbalance.

1.3 Dynamic Load Balancing among Processor Cores

A microprocessor has a set of registers that are used by independent execution entities such as "threads". A thread has its own set of variables and state. Load balancing among threads is achieved by calculating the load of each thread and comparing it with the other threads in the computation pool. The thread with the highest load is balanced by reducing the amount of computation and vice-versa. 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 algorithm and the complexity of the load that it is trying to balance. If the algorithm's computation is heavier than the actual load then it defeats the purpose of the balancing process. In case of periodic execution, if the algorithm kicks in too frequently then it poses an overhead on the application itself. On the other hand, if the algorithm kicks in less frequently then the application runs into the risk of a state of prolonged imbalance and the resources utilization is impacted. In order to avoid this, the algorithms should be light in nature so that their execution cost is negligible. In this thesis, I have compared three thread-level load-balancing algorithms each based on an entire history, a recent time window and the latest-slope of CPU load. In the following discussions, we simply call each algorithm History-based, Window-based and Slope-based algorithms. Out of the three algorithms, the Slope-based algorithm performance is superior because of its light weight computation component and lower complexity as compared to the other two.

1.4 Dynamic Load Balancing in MASS

The MASS library is a multi-threaded multi-process library that instantiates threads on different nodes. There can be either single node or multi nodes involved in the application execution. The library instantiates processes on different nodes and master node connects to all the slave nodes via Java Secure Channel (JSCH). Each node spawns a process that can contain multiple threads. The number of threads spawned on each node is equal to the total number of cores on that node. The simulation array data chunk received from the master node is equally divided among the threads. The threads work on their respective slice, and the boundaries are the thread bounds of that particular thread. The dynamic load-balancing algorithm measures the load on each thread and compares it with the other threads. The slice bounds of the thread with the highest load are shrank in size to reduce the computation space that in-turn reduces the thread load and vice-versa.

1.5 Research Objective

In this research, I have compared three load-balancing algorithms and have implemented them for multi-core load balancing. They are (1) the History-based algorithm, (2) the Window-based algorithm and (3) the Slope-based algorithm. The research objective is to analyze the performance of all the three techniques on a set of multi-threaded applications and to compare and determine that the performance of Slope-based algorithm is superior to the other two algorithm techniques.

Chapter 2: Dynamic Load Balancing Algorithms

This section introduces the conventional dynamic load-balancing algorithms, thereafter points out their drawbacks, proposes our Slope-based algorithm to address their gaps, and shows our brief strategies, both, at application and system-level algorithm implementations.

2.1 Conventional Algorithms

As conventional algorithms, we look at the Polynomial regression (fitting) based on complete history of previous readings and a window of recent trend for prediction purposes, each abbreviated as History-based and Window-based algorithms respectively.

2.1.1 Polynomial Regression (fitting) Overview

Polynomial Regression is defined as a form of liner regression in which the relationship between the independent variable x and dependent variable y is modeled as an nth order polynomial. The goal of regression analysis is to determine the expected value of the dependent variable y in terms of an independent variable x. A generalized simple polynomial equation of the nth degree would be as stated below:

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n$$

From the above equation, polynomial regression equation is used for computing the value of dependent variable y based on a series of independent values x and constants a1, a2, a3 and so forth. This makes a polynomial fitting equation an interesting method to predict the future values of y based on a set of previous historical x values.

In Figure 1, the curve plot is plotted using a predetermined (x, y) coordinates. Using a polynomial regression equation of certain degree, it is possible to predict the future value of y for a new value of x based on the history of previously recorded x values. This is one of the conventional methods used in predicting load values in the dynamic load-balancing domain.

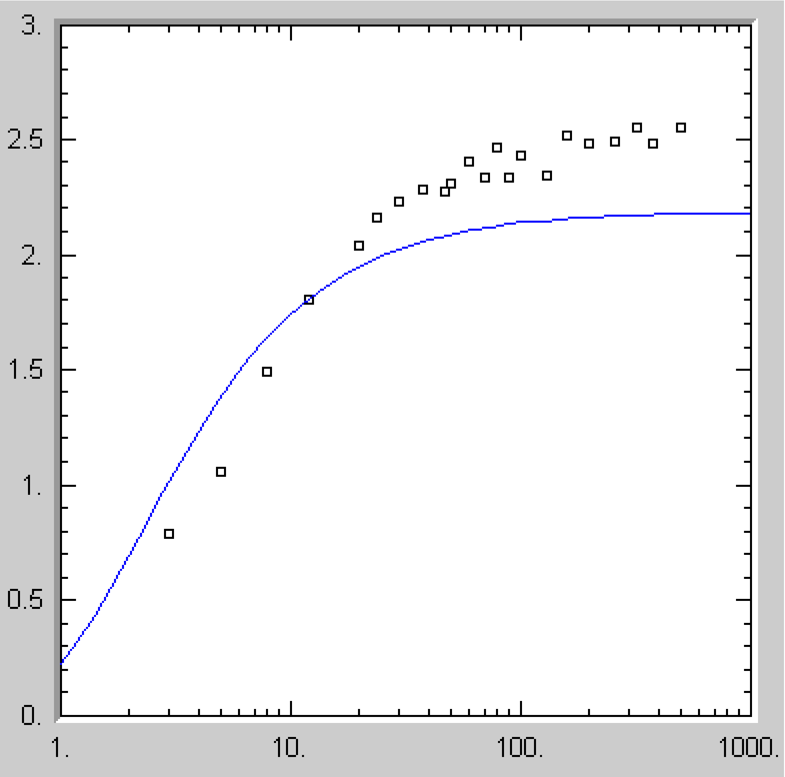

Figure 1: Graph to illustrate polynomial regression

2.1.2 Thread Load Measurement

A thread that executes a more computationally intensive task utilizes more CPU time. If we can measure the exact time spent by a thread running on a given CPU then we can deduce that this time component defines the load of that thread. Comparing this CPU time with other threads, we can determine which thread is computationally heavier and performing more work.

The JAVA management package offers ThreadMXBean class that provides various APIs to measure JAVA thread performances. The getCurrentThreadCpuTime() API provides the time spend by a particular thread running on the CPU in nanoseconds. The CPU time provided by this interface has nanosecond precision but not necessarily nanosecond accuracy. 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 support respectively. The thread load-balancing algorithms described below use the above described thread-load calculation technique.

2.1.3 History-based algorithm and prediction

The History-based algorithm takes into consideration the complete history of all previous events from the start of all threads. The algorithm computes the load value y for each thread and maintains a per thread list of load values on a time series x. Every time the algorithm takes the x snapshot of the current thread load, it computes the current load and adds it to the corresponding load pool as well as it also predicts the thread load that would be at the time of next snapshot for (x+1) time series candidate.

As described in section 1.4, multiple threads in the MASS application work on their respective simulation slice. Every load-balancing snapshot cycle compares the current thread load among each other and adjusts the slice boundaries of the most heavily loaded thread. The thread load or amount of thread computation can be controlled by increasing or decreasing the slice simulation space. Larger the space more the computation and vice-versa. Once an adjustment of boundaries is complete, the algorithm computes the prediction values of all the threads for a future time-series candidate. In the History-based algorithm, the prediction is calculated based on the complete previous history of readings from the simulation. After the predicted load values are computed for all the threads, the values are compared and the slice space is re-adjusted proactively for the thread with the highest predicted thread load. The threads are put to wait with the help of thread barrier logic before the slice bounds are adjusted. Once the adjustments are complete, all the threads are resumed to work on the newly formed slices. The polynomial regression is computed by expressing the data into a matrix form and thereafter performing Gauss-Jordan elimination. The data matrix values changes with the addition of every load value and hence previous values cannot be discarded to eliminate the overhead of data computation. The History-based algorithm uses polynomial regression of 4th degree to predict the load [3].

2.1.4 Window-based algorithm and prediction

The Window-based algorithm is similar to the History-based algorithm. The difference between the History-based and the Window-based algorithm is that the Window-based algorithm works on a sliding window [1] of history data. The sliding window can have the last n number of thread load readings to predict the next load value y for the future time series candidate x. The algorithm uses the same 4th level polynomial regression method used in History-based algorithm.

Here is an example to describe how a Window-based algorithm works:

The load pool for thread t1 has following values that were calculated till current time T: (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 then the sliding window values would be (9.8, 10, and 7). Polynomial regression is performed on this window data to get the 4th degree polynomial equation. This equation is used to predict future y value for time series candidate 40. The 4th degree polynomial regression is used on a 3 element window in order to maintain consistency with the history-based algorithm polynomial regression technique.

Some of the drawbacks/problems with the History-based and Window-based algorithms are: the simulation application has to store the history data in order to predict the next value. This history data needs to be stored in such a way that the data retrieval is not too expensive or else it can become an overhead and will nullify the load-balancing advantage. Since the history data increases with the simulation time, every load-balancing cycle will have more and more data to compute polynomial regression on and this will lead to slowing down the algorithm. The Window-based algorithm uses the last n element sliding window such that n < total history elements. This makes its prediction less accurate than History-based because the fitting is performed on less number of elements. The tradeoff is that the Window-based algorithm is faster and its execution cost remains constant, but at the cost of prediction accuracy.

2.2 Proposed algorithm

To address the problems with above described conventional algorithms, we proposed an algorithm called the Slope-based algorithm. The following section describes its design, execution and its implementation at the application and system level.

2.2.1 Slope-based algorithm and prediction

The proposed algorithm for this thesis is called the Slope-based load-balancing algorithm. As the name suggests, the Slope-based algorithm uses line slope between the two load value points on a graph. Load values y are plotted on the graph against the time series values x. The y coordinate are connected via lines and their slope values are calculated. The algorithm can be further illustrated using Figure 2;

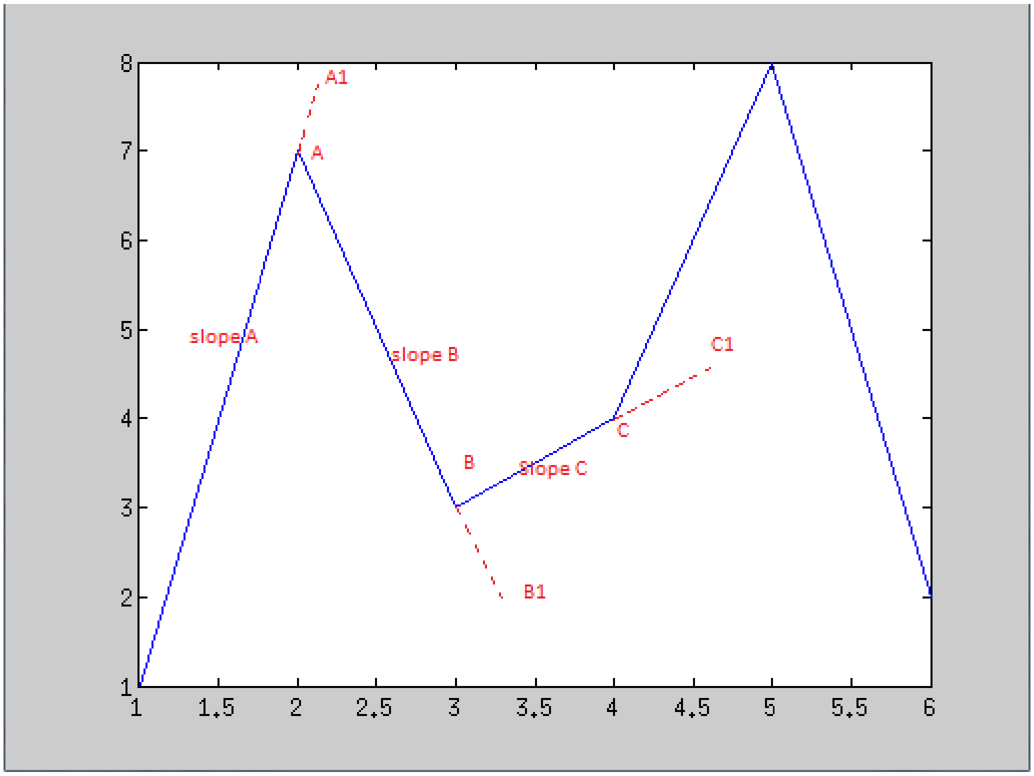

Figure 2: Slope based algorithm graph

The x-axis is the time series and the y-axis is the calculated load for each instances of x. the slope is calculated between two y points using the formula:

$$slope = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

As shown in the above figure, "slopeB" is calculated between point A and point B using the slope formula. Now if we assume that the slope remains constant then we can predict the value y by using the formula:

$$y_2 = \left(slope * (x_2 - x_1)\right) + y_1$$

This predicted value is B1.

During a cycle of load-balancing algorithm execution, the current load is recorded for all the threads in the thread pool. A slope value is calculated for the line between the current and the previous load coordinate for the same thread. The same operation is performed for all the threads and compared. The simulation slice space for the thread with the highest load is adjusted. As described above, assuming that the load will remain constant, the future y (load) value is predicted for all the threads, using the formula stated above. The predicted values are compared and the simulation slice region boundary is re-adjusted proactively for the highest loaded thread. The algorithm uses both the above stated formulas (1) slope formula to calculate load and (2) y2 formula to predict future load. These predicted values (i.e. y2) are also compared and the simulation slice region boundary is re-adjusted.

2.2.2 Other Slope-based algorithms

The Rate-of-change load-balancing algorithm [10] considers different phases such as when to initiate task migration, where to send the task, how many tasks to migrate and which task to migrate to different nodes over the network. The algorithm maintains a load table per each processing node. It uses Slope-based design where the load is plotted on a graph and the lines are drawn connecting two load values. The algorithm considers transferring a task to another node only if the next predicted value is going below a pre-determined low threshold value. The algorithm only looks at a slope that is going in a negative trend. The node with the “low” threshold sends a message to all the other nodes requesting tasks. The node with a higher load sends tasks to the lower node. After the transfer is complete, the corresponding load tables that maintain the current number of tasks are updated.

The Slope-based algorithm discussed in this thesis is different as compared to the above described algorithm. We consider both the slope values such as positive and negative, per load balancing snapshot. We compare the current slope values of all the threads and we adjust the slice bounds based on that value and then we consider the predicted value and re-adjust the bounds proactively. Our algorithm does not maintain a load table also the simulation slice is not adjusted based on any pre-determined upper and lower thresholds. Our algorithm balances the load among the cores and not across nodes on the network.

2.3 Application-level Implementation Strategy

We will implement the proposed algorithm on two applications: Wave2D and SugarScape. Our strategy is to divide the simulation space into smaller slices and each slice will be allocated to a different thread. The algorithm will have the capability to increase or decrease the slice breadth size depending on each thread’s load component which results in load balancing. Please refer to Chapter 3 for details.

2.4 System Level Implementation Strategy

As part of the system level implementation strategy, we will implement the proposed algorithm to Multi-agent spatial simulation (MASS) library. The library is a multi-threaded multi-process parallel simulation execution library. The following summarizes the MASS library specifications:

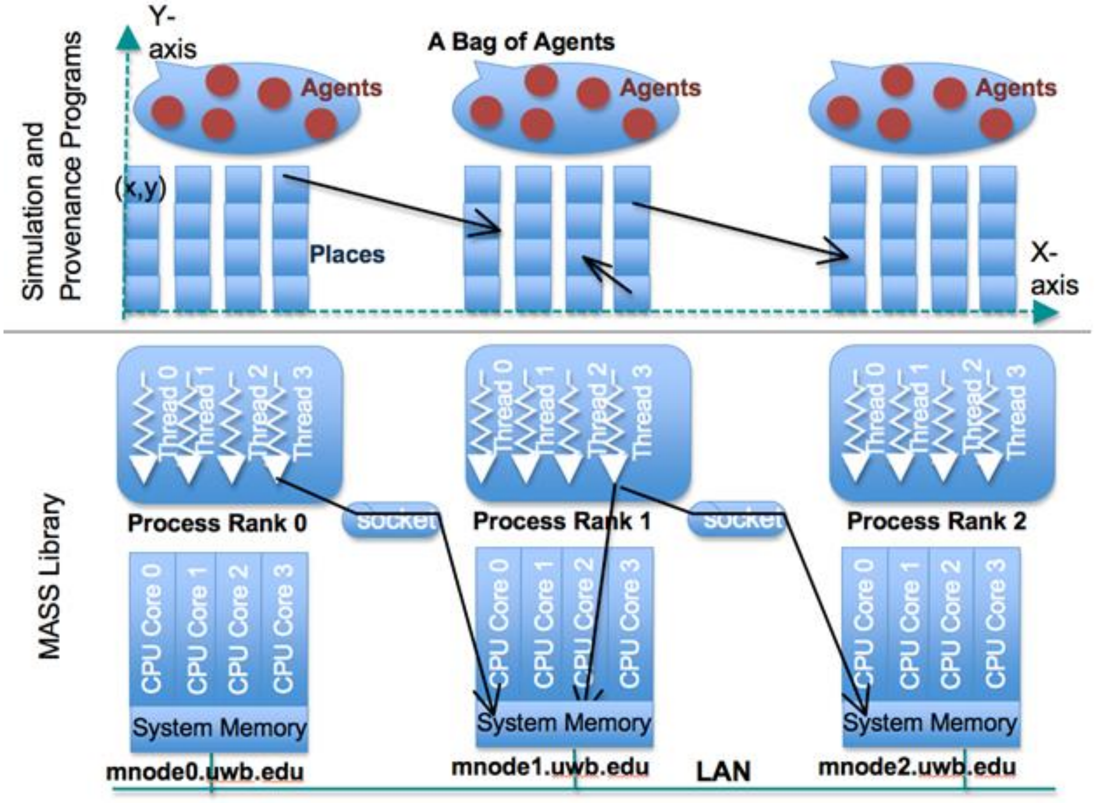

Figure 3: MASS library in action

Figure 3 illustrates the MASS library structure. Places and agents are the key elements of the MASS library. “Places” is a multi-dimensional array of elements that are dynamically allocated over the cluster of multi-core computing nodes. Each element called Place, is pointed to by a set of network-independent array indices and is capable of exchanging information with any other places. Agents are a set of execution instances that can reside on a place, migrate to any other places with array indices (thus duplicating themselves), and interact with other agents as well as multiple places [6]. As shown in the Figure above, 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 places and agents in parallel.

The threads that are spawned on the nodes use their respective chunk of data to work on. Dynamic load balancing algorithm would be implemented on MASS such that the algorithm executes on all the nodes in the cluster. Based on the thread load on each node, the chunk (slice) size will be increased or decreased. The algorithm will ensure that all the cores on all the processing nodes are utilized to its fullest.

Chapter 3: Implementation

In this chapter we describe the implementation(s) of all the three load-balancing algorithms at the application level as well as system level.

3.1 Application Level load balancing

Below we use two applications: Wave2D and SugarScape. Both these applications are spatial or multi-agent simulations and are multi-threaded and hence they are ideal candidates for our DLB algorithms.

Wave2D

Overview

Wave2D simulates wave dissemination using Schrodinger’s wave equation. The wave mimics a water wave. The simulation space mimics a container and the wave propagation gives a feeling as if the water was dropped at the center of the square container and it propagates towards the walls. Once the wave reaches the wall, it bounces and starts traversing back towards the center. At this point it also meets the already on-going wave that is moving towards the walls. When the two waves meet, a stationary wave is formed when they intersect and the propagation continues. The stationary wave reduces the propagation velocity of the wave that is moving towards the walls and towards the end of the simulation; the wave comes to a standstill. The graphic pallet is refreshed with the latest computed data at the end of every simulation cycle. The computation is illustrated in the Figure 4 below:

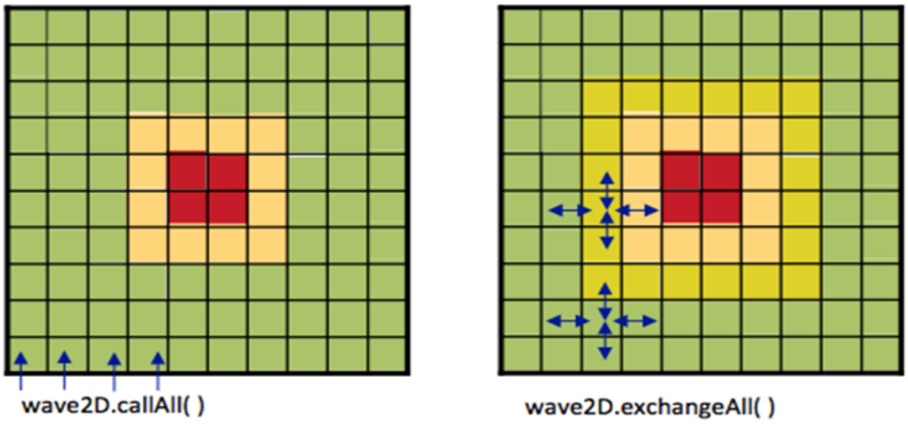

Figure 4: Wave2D application in action

This JAVA application uses a three-dimensional simulation space with dimensions N x N x 3. The simulation has a runtime of time T. At time T-0, the simulation array space NxNx0 is commutated and populated as per Schrodinger's equation. At time T-1, the simulation array space NxNx1 is computed and populated. At time T-2, the simulation array space NxNx2 is computed and populated. This is the initial start of the wave propagation. Now all the three layers are populated with data. Going forward, the top most simulation layer NxNx0 requires the other two layers, NxNx1 and NxNx2 data to compute its new values. At T-3 or later, the data from the simulation layer NxNx1 is copied to NxNx2, data from simulation layer NxNx0 is copied to NxNx1 and new values are computed for NxNx0th layer. During the course of simulation the data propagates from 0th layer to 1st layer to 2nd layer. Each cell in the array requires values from all its neighboring cells situated at north, south, east and west directions to compute its value, as show in Figure 4.

DLB algorithm implementation

The simulation array NxNx3 is divided into vertical slices so that each thread gets one slice to work on. Let us assume that the simulation space dimensions are 100x100x3 and we have total 4 threads that would be working on the application. Then the simulation space would be divided among 4 threads into four stripes of 25x100x3. In case of an odd number of simulation space size, the remainder space would be given to the last slice. For example 21x50x3 would be divided among 4 threads as three stripes of 5x50x3 and one more stripe of size 6x50x3. The threads perform computation on their respective slices and maintain slice bounds. They do not enter each other's slice spaces. Each thread starts from top left corner and traverses the slice from left to right.

As described in the previous sections, DLB algorithms calculate the CPU time spent by each thread processing the slice. Depending on the heavier thread, the slice boundaries are increased or decreased to control the amount of work that thread does, which in turn means controlling the load value. At the end of each simulation cycle, the threads are set in the wait mode by a thread barrier and the slice boundaries are adjusted. After the boundaries are adjusted, the threads are notified and they resume computation on newly adjusted slices. The benefit of this implementation is that the thread that was lightly loaded now has a bigger slice to work on and it has taken excess load from other threads thus increasing its core utilization.

SugarScape

Overview

SugarScape is based on a multi-agent based social simulation that follows some rules. The simulation has agent colonies and a simulation space where agents interact with each other as well as the environment conditions. Sugar in the simulation could be seen as a metaphor for resources in an artificial world through which the examiner can study the effects of social dynamics such as evolution, marital status and inheritance on populations.

Our application uses a two dimensional array simulation space. Each cell space of the array is termed as "Place". The agents are instantiated randomly on the simulation space where they take up a "place". The simulation space contains two "sugar hills". The agents have properties like metabolism and ability to consume sugar. They also have a visibility range. Each agent is able to see and check cells equal to its visibility capacity in all directions. During every simulation cycle, agents scan for sugar around their surrounding "place" and tend to move to a "place" that has a high Sugar value. An agent selects a space and consumes sugar. During the process of sugar consumption, an agent's metabolism value decreases and the pollution value in that "place" space increases. During the course of simulation cycles, an agent will die if the metabolism value reduces to zero and the agent object is removed from the active simulation. Figure 5 illustrates how agents move in the simulation space.

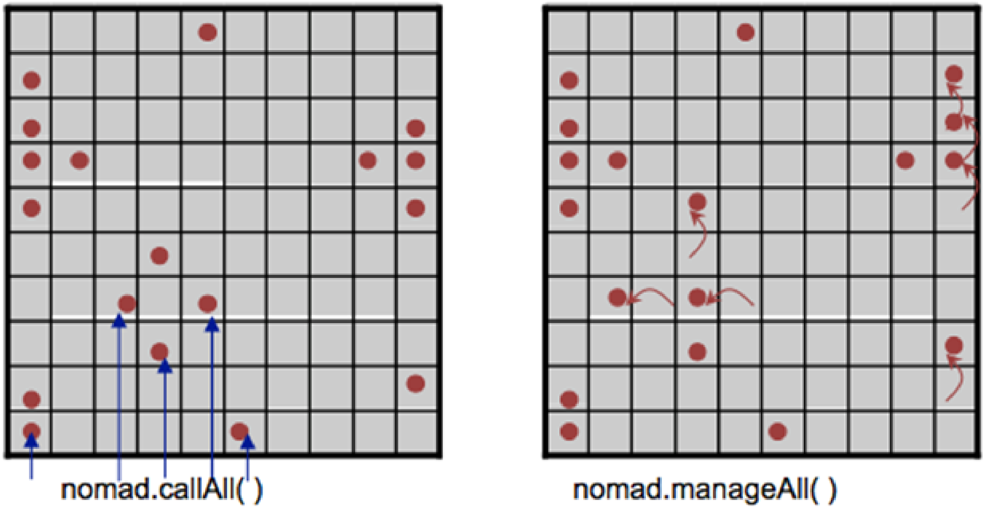

Figure 5: SugarScape simulation showing agents movement

The two-dimensional simulation space is divided into slices similar to Wave2D application. During each simulation cycle, agents sense the "sugar" values in neighboring cells and they designate their migration place. If two agents try to move to the same cell then the agent with lower “agentId” value gets preference and is migrated to that location. Agents can migrate to neighboring slice cells during the migration phase.

Each thread works on a slice and similar to the Wave2D application, the slice boundaries are adjusted for the most heavily loaded thread.

3.2 System level load balancing

After verifying the efficiency of the application-level load-balancing algorithms, we implemented them on the MASS library. In the following section we describe algorithm implementation on MASS and its performance.

Mass Single Node

Overview

As described in section 1.1, the MASS library provides parallel execution platform for the on-the-fly simulation algorithms. The library is designed to execute in single-node-multi-threads or multi-node-multi-threads mode. In the single-node-multi-threads mode, a single process is created on the master node and it can host multiple threads. The MASS library has two main functions, “callAll” and “exchangeAll”. The “callAll” method executes the computation component in each place object (or a cell of the simulation space) and “exchangeAll” initiates the transfer of the data either among the local places or among places from remote nodes. In case of single node, “exchangeAll” method initiates exchange among local places only. The simulation cycle executes “callAll” and “exchangeAll” consecutively.

DLB algorithm implementation

The MASS library initializes the simulation space for applications as Wave2D does. The DLB algorithms are implemented in such a way that the simulation driver maintains a count the callAll and exchangeAll methods are called. A consecutive call of “callAll” and “exchangeAll” method completes one computational cycle. A consecutive call of callAll and exchangeAll would increment the count two times. A user selected load-balancing algorithm is set to kick in on a predetermined count value and when the method-count value matches this count, the load-balancing algorithm kicks in. The threshold count is configurable. All the threads call the callAll method and go into wait state and once all the threads are in the wait state, the main thread resumes all the threads to call the exchangeAll method. The load-balancing algorithm is executed between the two method calls when the threads are in the wait state. The CPU time of the threads are calculated and compared and the slice bounds are adjusted. The method counter is set to 0 after successful completion of DLB algorithm execution.

MASS Multi Node

Overview

MASS multi-node implementation involves multiple nodes connected over the network. The topology consists of the master and slave nodes connected to each other. The nodes are connected using the JSCH library. The master node reads a machine file that contains the list of slave node hostnames and initiates connection via JSCH using SSH credentials. Once the connection is established, the master node instantiates a process and copies dependency jar files. The process can have multiple threads running in it. Once the topology cluster is up and running, the master node splits the simulation space into equal chunk sizes and they are transferred across to the slave nodes. The "callAll" method is called on the master as well as slaves via a serialized command object. After "callAll" method is completed, "exchangeAll" method is called on all the nodes to perform the data exchange, this method brings all the slave nodes to data sync. The call of the two methods depends on the type of application. Users must ensure when the application simulation cycle is completing and set the DLB threshold count appropriately. For example if an application calls "callAll" two times before calling "exchangeAll" then that would be considered as one cycle and the DLB threshold count must be configured to 3. The method execution commands are always initiated by the master node. In case of single node MASS library implementation, the DLB algorithm kicks in on single node but in case of multi-node implementation, the DLB algorithm kicks in on all the nodes independently. This not only improves the efficiency of the master node but all the nodes involved. When the simulation ends, the processes are destroyed and nodes are disconnected.

DLB algorithm implementation

During the simulation cycle, when the slave nodes receive the data chunk, the chunk is further divided into slices among total number of threads running on that slave node. The DLB algorithm executes independently on the master node and the slave nodes. At the end of each command execution the threads are moved to wait state and at the end of exchangeAll execution, data is exchanged among the slave nodes and the master node. As described in the above section, the counter logic executes between the methods calls on slave nodes and executes the DLB algorithm when the threads are in the wait state.

DLB Performance Testing Strategy

Application level DLB strategy

In case of application-level DLB implementations, the total execution times of applications Wave2D and SugarScape are recorded for varying number of threads such as 2, 3, 4, 5, 6, 7 and 8 for all the 3 DLB algorithms. The execution time of each application is also recorded without any load balancing algorithms for the baseline.

System level DLB strategy

In case of MASS single node, the same application level testing strategy is followed. In case of MASS multi-node implementation, the execution time was recorded for 3 and 4 threads since the node machines used had two dual-core processors.

Chapter 4: Performance Evaluation

In this section we compare the Slope-based algorithm with the two conventional DLB algorithms for their application and system-level execution. The evaluation system we used has the following specifications:

Mode	Cores	MHz	Memory	Network
Single-node	8 cores	1600	5GB	Linux LAN
Multi-node	4 cores	1800	2GB	Linux LAN

Table 4.0: Single node and multi-node test machine configurations

4.1 Application level Load Balancing Performance

In this section we describe the performance and analysis of the application Wave2D and SugarScape.

Wave2D

(a) <u>History Based DLB algorithm</u>

Table 4.1 and Figure 6 represents the results of the History-based algorithm on Wave2D multi-threaded version.

Reading no.	Threads	Time(ms)
1	2	968561
2	3	1246358
3	4	1557244
4	5	1861636
5	6	2181457
6	7	2520160
7	8	2842075

Table 4.1: Wave2D execution time for history based algorithm

Graph:

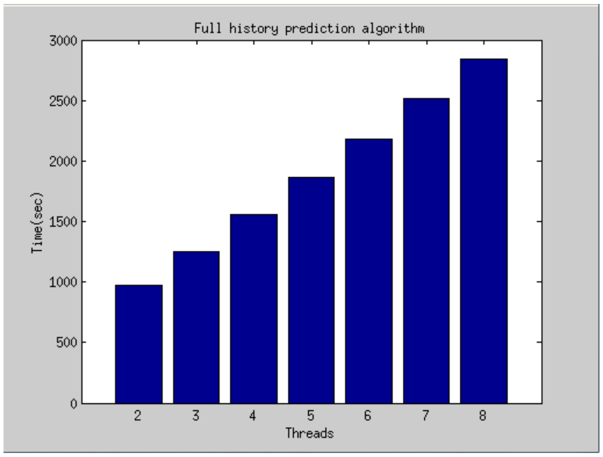

Figure 6: Full history based algorithm performance

We see that as the number of threads increases, the DLB algorithm has to perform polynomial fitting for each thread based on its increasing history data. This overhead increases the execution time and defeats the load-balancing purpose. The percentage increase in execution time from threads 2 to 3 to 4 to 5 and so on till 8 is 28.6%, 24.94%, 19.54%, 17.17%, 15.52% and 12.77% respectively.

(b) Window based DLB algorithm

Table 4.2 and Figure 7 show the performance readings for Window-based DLB algorithm on Wave2D using increasing number of threads.

Reading no.	Threads	Time(ms)
1	2	242125
2	3	177270
3	4	134649
4	5	116390
5	6	106373
6	7	99849
7	8	98231

Table 4.2: Wave2D performance for window based DLB algorithm.

Graph

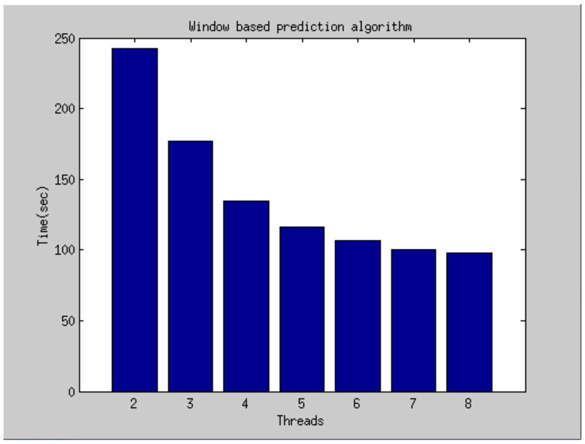

Figure 7: Window based algorithm performance on Wave2D

Unlike History-based DLB algorithm, the Window-based algorithm has fixed amount of polynomial fitting computation overhead since it uses a fixed element sliding window. This show reduction in application execution time as number of threads increases. The percentage reduction in execution time for threads 2 to 3 to 4 till 8 were recorded as 26.78%, 24.04%, 13.56%, 8.60%, 6.13% and 1.62% respectively.

(c) Slope Based Algorithm

Table 4.3 and Figure 8 show the performance readings for Slope-based DLB algorithm on Wave2D application.

Reading no.	Threads	Time(ms)
1	2	240645
2	3	173043
3	4	133165
4	5	120654
5	6	105060
6	7	98929
7	8	99569

Table 4.3 Execution times for Slope based DLB in Wave2D

Graph:

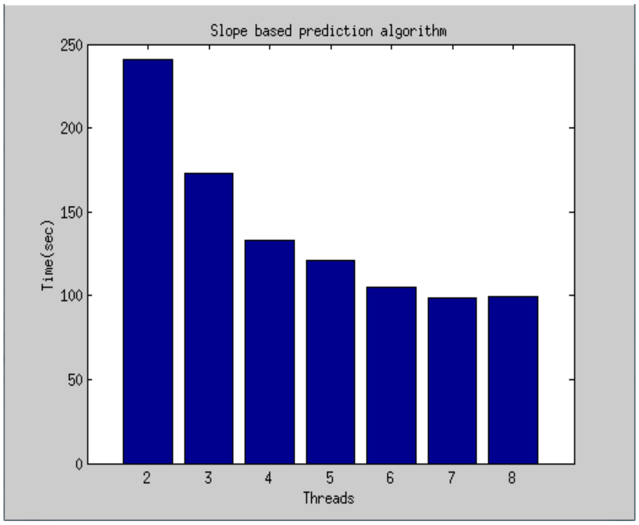

Figure 8: Slope based algorithm performance on Wave2D

Similar to previous two algorithms, performance of Slope-based DLB algorithm was recorded and following are the percentage decrease in the execution time as the number of threads increases from 2 to 6. The percentages recorded were 28.09%, 23.04%, 9.39%, 12.92% and 5.83% respectively. The percentage values are better as compared to History-based and Window-based algorithms because the Slope-based algorithm does not use polynomial fitting as well as it is much more computationally lighter than the other two algorithms. There was no improvement seen for 7 and 8 threads. This could be because of the Operating system thread management overhead that might have been introduced with the increasing number of threads.

Comparative Analysis Wave2D

Table 4.4 and Figure 9 compare the three algorithms on Wave2D application and their execution time.

Reading no.	Threads	Full history	Window Based	Slope Based	Without DLB
1	2	968561	242125	240645	421252
2	3	1246358	177270	173043	326323
3	4	1557244	134649	133165	274472
4	5	1861636	116390	120654	345853
5	6	2181457	106373	105060	323918
6	7	2520160	99849	98929	308785
7	8	2842075	98231	99569	296340

Table 4.4 Comparative DLB algorithm analyses for Wave2D

Graph

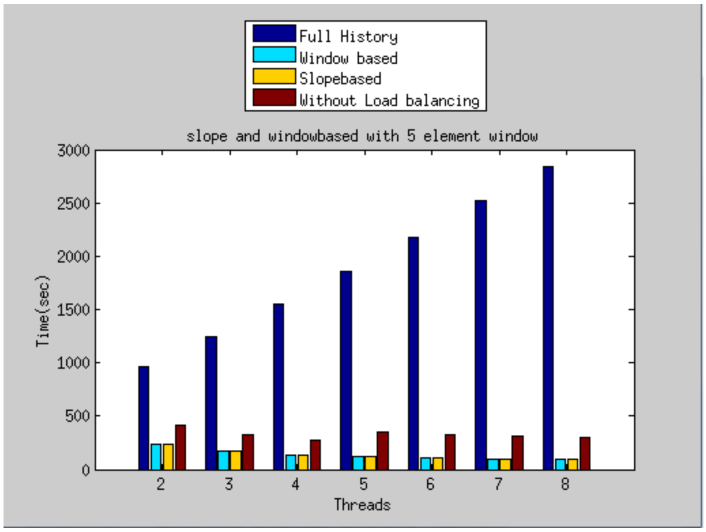

Figure 9: Comparative analysis of 3 DLB algorithms on Wave2D

From the above comparative analysis, we can conclude that the Slope-based algorithm performs better as compared to Window-based and History-based algorithms for threads 3, 4, 6 and 7. The percentage improvement values of Slope-based over Window-based are 2.38%, 1.10%, 1.23% and 0.92% respectively and over History-based are 86.11%, 91.44%, 95.18% and 96.07% respectively. The Window-based algorithm uses a 3 element sliding window and it’s a very strong contender with respect to the Slope-based algorithm.

Window Based with 5 elements

Table 4.5 and Figure 10 show performance analysis conducted on Window-based DLB algorithm with 5 elements sliding window. The readings were compared with Slope-based algorithm readings.

Threads	Slope based	Window based	Without DLB
2	240645	432846	421252
3	173043	337866	326323
4	133165	429283	274472
5	120654	376208	345853
6	105060	341139	323918
7	98929	318897	308785
8	99569	301778	296340

Table 4.5 Execution time of 5 element sliding window with slope DLB algorithm

Graph

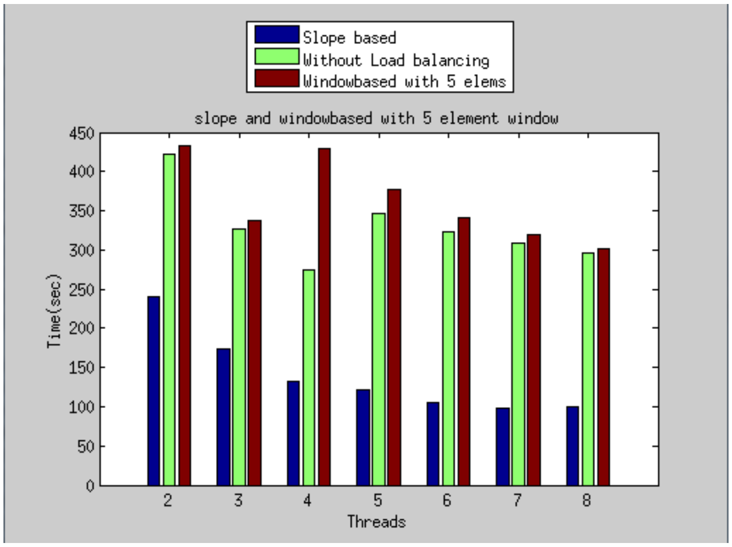

Figure 10: Comparative analysis of 5 element sliding window and slope based algorithm

The percentage improvement in Slope-based algorithm over Window-based with 5 elements for threads 2, 3, 4, 5, 6, 7 and 8 are 44.40%, 48.78%, 68.97%, 67.92%, 69.20%, 68.97% and 67% respectively. From the above comparative analysis, the Slope-based algorithm shows significant improvement over Window-based algorithm with 5 element sliding window. This is because the 5 element window introduces more computational overhead as compared to 3 elements. In case of the Slope-based algorithm the computational cost remains constant whereas in case of Window-based algorithm the computing overhead increases as the window size increases. This makes the Slope-based algorithm an ideal choice.

SugarScape

All the three algorithms were implemented on SugarScape application and performance readings were recorded. Table 4.6 and Figure 11 summarize the comparative analysis:

Threads	History Based	Window Based	Slope Based	Without Load
2	10302	2192	1887	1109
3	14849	3114	2348	1309
4	18655	2412	2546	1463
5	22424	2597	2826	1148
6	26442	3567	2837	1646
7	30473	3155	2997	1712
8	34916	3420	3100	2116

Table 4.6: Comparative analysis of DLB algorithms for SugarScape

Graph

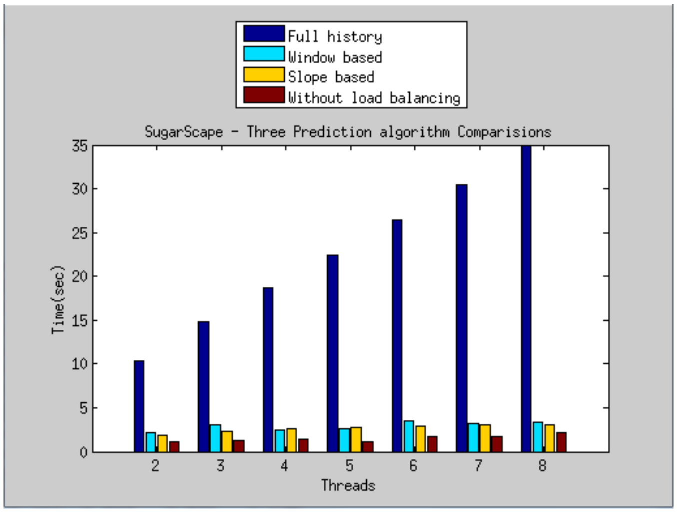

Figure 11: Comparative analysis of 3 DLB algorithms on SugarScape

From the above comparative analysis we conclude that Slope-based algorithm performs better than Window-based and History-based algorithms in SugarScape application. The percentage improvement in execution time of the application for Slope-based over Window-based for threads 2, 3, 6, 7, 8 are: 13.91%, 24.59%, 20.46%, 5% and 9.35% respectively. No improvement was observed for threads 4 and 5 and this could be because of the operating system thread management overhead. Performance improvement for Slope-based over History-based for threads 2, 3, 6, 7, 8 are: 81.68%, 84.18%, 89.27%, 90.16% and 91.12% respectively. From the above results we can conclude that performance of Slope-based algorithm is superior over the other two DLB algorithms and that is our ideal choice. One more point to note here is that the overall execution time of the SugarScape application in the absence of any DLB algorithm implementation was faster than with load-balancing. SugarScape application is computationally lighter as compared to Wave2D and because of this the cost of a DLB algorithm execution is more than the actual application computation cost. It is of utmost importance to understand that a DLB algorithm will introduce certain amount of overhead and should be used only on applications that are processor intensive.

4.2 System Level load balancing Performance

MASS library single-node on Wave2D

The three algorithms were implemented on the MASS library single-node running Wave2D application. Table 4.7 and Figure 12 show the performance results.

Threads	History based	Window Based	Slope Based
2	10554	10576	10256
3	8857	8232	7917
4	6971	6985	6383
5	6039	6146	5933
6	5407	5454	5595
7	5872	5461	5196
8	4889	4521	5128

Table 4.7 Execution times of all algorithms for MASS single-node

Graph:

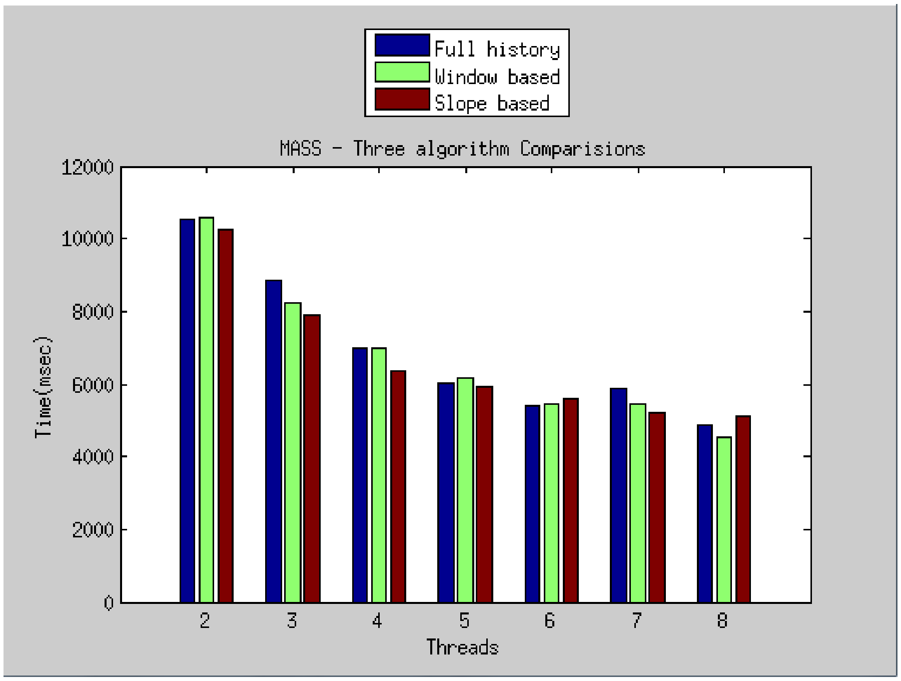

Figure 12: Comparative analysis of 3 algorithms on MASS single-node

From the above analysis, Slope-based algorithm performs the best for MASS single node running Wave2D application as compared to the other two. The percentage decrease shown in execution time by Slope-based algorithm over Window-based for threads 2, 3, 4, 5, and 7 are: 3.02%, 3.82%, 8.61%, 3.46%, and 4.85% respectively whereas improvement over History-based was: 2.82%, 10.61%, 8.43%, 1.75%, and 11.51% respectively. For 6 and 8 threads the performance comparison between Slope-based and History-based did not follow the trend and we can say that operating system level interference would have caused this.

MASS library multi-node on Wave2D

The machines used in the MASS library multi-node performance testing had 2 dual core processors that can run 4 concurrent threads. Table 4.8 and Figure 13 show the performance readings.

Threads	History Based	Window Based	Slope Based	Without DLB
3	11248	10611	8629	11622
4	10193	10623	9168	10478

Table 4.8 Execution times for MASS multi-node library.

Graph:

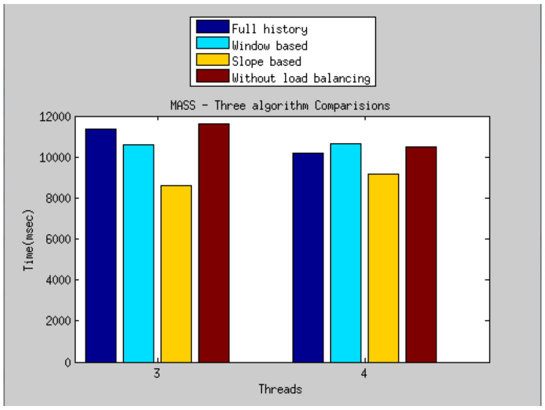

Figure 13: Comparative analysis of 3 algorithms on MASS multi-node

Wave2D execution times were recorded for all three DLB algorithms on MASS library multi-node. From the above comparative analysis we can conclude that Slope-based algorithm performs best as compared to the other two DLB algorithms on MASS multi-node Wave2D application. The percentage improvement in the performance of Slope-based algorithm over Window-based for threads 3 and 4 is: 18.67% and 13.69% respectively and for Slope-based over History-based are: 23.28% and 10.05% respectively.

4.3 Performance Summary

From all the above performance and comparative analysis we found that performance of Slope-based algorithm is superior to History-based and Window-based algorithms in application-level load-balancing as well as system-level load-balancing implementations.

Performance of History-based algorithm is the least favorable as compared to the other two and this is because its complexity is O (n). The data required for the algorithm for prediction increases with time and this also increases the execution time of the algorithm. At certain point, the computation cost of History-based algorithm exceeds the actual application load cost that we are trying to reduce. During our performance testing we are not storing the history data on the disk but for the applications that run longer simulation cycles, it would become necessary to store the history data on disk. The storage will introduce another complexity of efficiently retrieving it for prediction. This will further slowdown the application.

Performance of Window-based algorithm with a 3 element window is a very close contender to Slope-based algorithm. In this comparison we concluded that Slope-based algorithm performs better because our prediction logic is lighter than the polynomial fitting method with 3 elements window. The Window-based algorithm is faster since it uses a fixed element window but at the cost of prediction accuracy. When the window was increased to 5 elements in order to improve the prediction accuracy, the computation cost increased and the performance decreased. Unlike History-based algorithm, the Window-based algorithm will never have the need to store the data on the disk.

The Slope-based algorithm results for threads 4 and 5 for SugarScape application were out of trend (refer table 4.6). This could be the result of the thread management operations done by the operating system when the number of threads in a processor increases.

With respect to the overall performance, Slope-based algorithm is our ideal candidate for thread-level load-balancing in application-level as well as system-level applications.

Chapter 5: Future Work

As future work, we are planning to port our proposed algorithm to the following two platforms (1) C++ and (2) MASS GPU.

5.1 Porting of Dynamic load balancing algorithm to C++

Currently the thread-level load-balancing is implemented in JAVA using the JAVA management package APIs. This makes the algorithm restricted only to applications that use Java and JVM. It will increase the usage of the algorithm widely if the algorithm is also ported to C++. The following shows the key points that need to be considered while porting the algorithm to C++:

- A method needs to be designed to compute the exact time in milliseconds that a thread has spent on the CPU doing computation.
- The two-dimensional simulation space should be implemented in C++ that handles Places and agent objects.
- A garbage collection method needs to be implemented that cleans up the agent object memory when they die periodically.

We will examine all DLB algorithms, both at C++ and MASS C++ level that is currently being developed.

5.2 Load balancing algorithm on MASS GPU

MASS library has already been ported on GPU. After the load-balancing algorithm is ported on C++, it should be implemented on the MASS GPU version as well. Performance evaluation should be done on GPU and see if it improves the execution time of the applications running on GPU. This will further improve the performance of applications that use heavy computations and rely on GPUs. Some of the examples include: weather prediction algorithms, Thermodynamics and fluid dynamics computations and predictions, real time traffic data analysis and predictions etc.

It is our general belief that the thread-level load-balancing algorithm will inherently perform better on C++ as compared to JAVA just because of the extra JVM overhead. This is just an assumption and can be confirmed only after analyzing the performance results of the DLB algorithms on C++ and comparing it with the JAVA version.

Chapter 6: Conclusion

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 performance with them and our analysis has demonstrated that the Slope-based algorithm is not only superior but also light in terms of processing cost in both, application-level load balancing domain and System-level load balancing domain. Slope-based algorithm is around 81% faster than History-based algorithm and is 24% faster than Window-based algorithm on MASS multi-node library running Wave2D. The performance analysis were conducted in a controlled LAB environment with standard Linux machines with dual and quad core processors running at 1800MHz and 1500MHz respectively.

References

[1] Yang, L., Foster, I., & Schopf, J. M. (2003). Homeostatic and Tendency-based CPU Load Predictions Mathematics and Computer Science Division , Argonne National Laboratory , Argonne , IL 60439, 00(C).

[2] T. Chuang, Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library, Bothell: Master's thesis, Master of Science in Computing and Software Systems, University of Washington, 2012

[3] Inoguchi, Y. (2006). CPU Load Predictions on the Computational Grid *. Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), 321-326. Ieee. doi:10.1109/CCGRID.2006.27

[4] Mehta, M. a., & Jinwala, D. C. (2012). Analysis of Significant Components for Designing an Effective Dynamic Load Balancing Algorithm in Distributed Systems. 2012 Third International Conference on Intelligent Systems Modelling and Simulation, 531-536. Ieee. doi:10.1109/ISMS.2012.83

[5] Ajaltouni, E. E., Boukerche, A., & Zhang, M. (2008). An Efficient Dynamic Load Balancing Scheme for Distributed Simulations on a Grid Infrastructure. *2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, 61-68. Ieee. doi:10.1109/DS-RT.2008.38

[6] Chuang, T., & Fukuda, M. (n.d.). A Parallel Multi-Agent Spatial Simulation Enviornment for Cluster Systems,Bothell: Master's thesis paper, Master of Science in Computing and Software Systems, University of Washington, 2012

[7] Swarm main page, http://www.swarm.org/index.php/swarm main page.

[6] Multi-Agent Transport Simulation - MATSim, "http://www.matsim.org/."

[9] FluTE, an influenza epidemic simulation model, "http://www.cs.unm.edu/ dlchao/flute/."

[10] Campos, L. M., & Scherson, I. D. (n.d.). Rate of Change Load Balancing in Distributed and Parallel Systems 1 Introduction.

Appendix A: Slice Bounds adjustment Source code

Following is the source code that adjusts the boundaries of a particular slice when the threads are in the wait state.

```
            //Current thread is the thread with highest load
            //Get the lowerbound of the slice current thread is working on
            int lowerBoundOfCurrentThread = DLBParams.boundaryMap.get(threadId).getLowerBound();
            //Get the upperbound of the slice current thread is working on
            int upperBoundOfCurrentThread = DLBParams.boundaryMap.get(threadId).getUpperBound();

            if (lowerBoundOfCurrentThread == 0) {

                    Long thId = getRightPeerThread(upperBoundOfCurrentThread+1);

                    //Adjust the upper bound of the Slice
                    if ( lowerBoundOfCurrentThread != (upperBoundOfCurrentThread - 2) ) {
                            upperBoundOfCurrentThread -= 1;

    DLBParams.boundaryMap.get(thId).setLowerBound(DLBParams.boundaryMap.get(thId).getLowerBound(
) - 1);

    DLBParams.boundaryMap.get(threadId).setUpperBound(upperBoundOfCurrentThread);
                    }

            } else if (upperBoundOfCurrentThread == DLBParams.MAX_SIM_SIZE) {

                    Long thId = getLeftPeerThread(lowerBoundOfCurrentThread-1);
                    //adjust the lowerbound of the slice
                    if (upperBoundOfCurrentThread != (lowerBoundOfCurrentThread + 2)) {
                            lowerBoundOfCurrentThread += 1;

    DLBParams.boundaryMap.get(thId).setUpperBound(DLBParams.boundaryMap.get(thId).getUpperBound(
) + 1);

    DLBParams.boundaryMap.get(threadId).setLowerBound(lowerBoundOfCurrentThread);
                    }
            } else {
                    //Get the peer thread on the left side
                    Long leftThId = getLeftPeerThread(lowerBoundOfCurrentThread-1);
                    //Get the peer thread on the right side
                    Long rightThId = getRightPeerThread(upperBoundOfCurrentThread+1);

                    //Get the loads of the left and right peer threads
                    double leftThreadTime = DynamicLoadBalancer.threadTimeDiff.get(leftThId).getLast();
                    double                              rightThreadTime                              =
DynamicLoadBalancer.threadTimeDiff.get(rightThId).getLast();

                    //Adjust the current slice bounds in such a way that the second highest
                    //loaded thread is also balanced.
                    if (leftThreadTime < rightThreadTime) {
                            if ((lowerBoundOfCurrentThread+2) != upperBoundOfCurrentThread) {
                                    lowerBoundOfCurrentThread += 1;
                                    DLBParams.boundaryMap.get(leftThId).setUpperBound(
    DLBParams.boundaryMap.get(leftThId).getUpperBound() + 1);

    DLBParams.boundaryMap.get(threadId).setLowerBound(lowerBoundOfCurrentThread);
                            }

                        } else if ( (rightThreadTime < leftThreadTime) || (leftThreadTime == rightThreadTime) )
{
                            if (lowerBoundOfCurrentThread != (upperBoundOfCurrentThread - 2)) {
                                upperBoundOfCurrentThread -= 1;
                                DLBParams.boundaryMap.get(rightThId).setLowerBound(

    DLBParams.boundaryMap.get(rightThId).getLowerBound() - 1);

    DLBParams.boundaryMap.get(threadId).setUpperBound(upperBoundOfCurrentThread);
                            }
                        }
                }
```

Appendix B: User Manual

Following steps needs to be followed in order to implement the DLB algorithm on an existing multi-threaded simulation application that uses MASS library platform.

1) Copy the jar files DLB.jar to the location from where the application will be invoked.
2) Make sure the DLB.properties file is present at the directory location from where the application will be executed.
3) The structure of DLB.properties file is as follows:
```
dlbCount=4
HISTORY_BASED=false
WINDOW_BASED=false
SLOPE_BASED=false
```
4) By default all the algorithm flags are false; set the flag as true for the particular algorithm that you want to invoke.
5) Default counter value, dlbCount=4, and it is the optimum counter value. This can be configured to a desired number.
6) The User application main method should call the method MASS.initBoundaries(places) after the Places object is initialized. This step initializes the thread bounds for the DLB algorithms. The application will throw an error if this step is not completed.
7) Compile the application class with the MASS.jar, DLB.jar and jsch-0.1.44.jar files using the command:
```
javac –cp  ./DLB.jar:./MASS.jar:./jsch-0.1.44.jar <application_class_file>.java
```
8) After the compilation is successful, execute the MASS library with the application DLB using following command:
```
java -Xmx2g -cp ./DLB.jar:./MASS.jar:./jsch-0.1.44.jar:. Wave2DMass <ssh-user> <ssh-password> <port> <simulation_space_size> <maxTime> <interval> <processes> <threads>
```

   - ssh-user: the ssh user that can be used to ssh into different nodes in machine.txt file.
   - Ssh-password: the password for the ssh user
   - Port: Port on which the processes will be created and master node will connect to.
   - Simulation_space_size: the size of the 2 dimensional simulation array e.g. 50 then the space would be 50x50
   - maxTime: maximum time for which the simulation will run.
   - Interval: interval at which the graphics will kick in and print the results.
   - Processes: number of remote processes, this value will be (nodes from machinefile.txt+1)
   - Threads: threads that would be spawned in each node process.