Field-Based Job Dispatch and Migration

Somu Jayabalan

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computing and Software Systems

University of Washington

2012

Committee:

Munehiro Fukuda, Chair

Kelvin Sung

Charles Jackels

Hazeline Asuncion

Program Authorized to Offer Degree:

Computer and Software Systems - Bothell

University of Washington

Abstract

Field-Based Job Dispatch and Migration

Somu Jayabalan

Chair of the Supervisory Committee:
Munehiro Fukuda, Ph.D.
Computing & Software Systems

AgentTeamwork-Lite is a mobile-agent-based job scheduling and monitoring framework that has been developed in the concept of field-based job dispatch and migration where agents migrate over a computing-resource field to highest available computing nodes for executing their user jobs as if they were electrons sliding down on an electric field. The agents keep monitoring their computing-resource field and move their user jobs to better computing nodes. This paper presents the system design, execution model of the framework, and our performance evaluation using two applications: the Wave2D MPI-parallelized wave-propagation simulator and the Mandelbrot Fractal generator benchmark programs.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction

1.1 Overview

Field-Based job dispatch and migration is a distributed job scheduling and monitoring framework developed on AgentTeamwork: a mobile-agent-based grid-computing middleware [1]. A field refers to a computing potential resource field, computed from node's CPU and memory capacity. In an electrical potential field, electrons will slide from a low dense area to a high dense area. The same way, AgentTeamwork spawns agents upon a job submission and has them migrate from a lower-ranked computing node to a higher-ranked computing node. It consists of several mobile agents such as PFAgent(Potential Field Agent), Commander Agents and Sentinel Agents to monitor system resources, to execute user jobs and to move the jobs to best computing nodes.

My contribution to this research is to study the effect of CPU and memory factors in job execution performance, to determine the best computing node itinerary for achieving the fastest execution time, and to find out right time for moving the currently executing job from a poor performing node to a better performing node.

1.2 Background

The recent emergence of cloud services [2] created a quantum leap in high performance computing (HPC). HPC enables scientists and researchers to solve complex scientific data analysis problems, typically applications developed on simulation based algorithms. For instance on-the-fly sensor data analysis [3] uses temperature interpolation [4] algorithm to predict the temperature of sensor uncovered areas. In biological field, motif discovery algorithms [5] are used to study various network patterns such as protein-to-protein interactions and gene network of bacteria etc. Climate change analysis has two phases of big-data processing: 1) simulation and 2) post simulation analysis and provenance [6] that requires huge memory space and high-speed disk storage. BrianGrid is a neural network simulator that imitates the growth of electrical activation and synapses among neurons to examine the neural spike and radius history of many different layouts. FluTE is a publicaly available stochastic influenza epidemic simulation model [7] to calculate daytime susceptibility, infection for each person, transmitting infection, and responding to epidemic.

All these applications require many computing nodes, and their execution could last from couple of hours to several hours. During the course of application execution, computing node performance could vary from time to time. Hence, we need a set of job deployment, monitoring and migration tools to coordinate job execution in a distributed computing space. OpenPBS, Condor [6], and Globus [7] are some of the popular job deployment tools available to work with cloud services such as Amazon EC2. But these tools have some limitations in working with parallel applications. For example, Condor uses a centralized scheduling for parallel jobs. They are generally based on a centralized or a hierarchical job scheduling strategy that statically allocates computing resources to parallel jobs upon their invocation. During parallel job execution, even if the current executing nodes do not meet user-specified resource criteria, the job will be still executed on the same set of nodes. This motivates us to develop a migration-based job-scheduling tool to work with both parallel and sequential applications.

1.3 Research Objective

The UWB distributed systems laboratory (led by Professor Fukuda) has developed the AgentTeamwork framework based on a mobile agent platform to target both serial as well as parallel job scheduling and coordination. The current implementation of the framework supports job migration based on a static list of computing nodes. Since it was mainly focusing on fault tolerance, my research has enhanced the framework as AgentTeamwork-Lite to focus on job migration using dynamically evaluated criteria.

My research goal through the AgentTeamwork-Lite framework development focuses on

- Developing an algorithm to form the best computing node itinerary based on resource information (CPU and Memory),
- Enhancing AgentTeamwork to schedule and to migrate a job based on dynamically evaluated criteria,
- Finding out the job migration cost and appropriate time to migrate, and
- Demonstrating the performance gain of field-based job migration scheduling over default scheduling.

This research contributes towards high performance computing where job migration is commonly used. All complex scientific applications including the ones discussed above can be benefited from this research. It also relieves users from handling job coordination which includes process communication, synchronization and dynamic load balancing. Also as part of this research I have evaluated the performance difference between the default-scheduling, (i.e., static) and migration-based scheduling for serial and parallel program.

The rest of the document is structured as follows: Chapter 2 discusses the system design, factors affecting job-execution performance followed by migration algorithm; Chapter 3 discusses the performance evaluation; Chapter 4 discusses the related work in the same field, and Chapter 5 summarizes my research and concludes with the future work to be done.

Chapter 2: Methods

This chapter describes challenges and solutions in dynamic job scheduling, the AgentFramework-Lite design, and its migration algorithm.

2.1 Challenges and solution in job scheduling and coordination

In grid computing, job scheduling has the following challenges.

1) Master-slave node relationship – Grid computing uses a set of computing nodes. In general, a node identified as a master node keeps track of slave nodes. This creates a high dependency on the master node and if that node goes down, the entire grid is not available.

2) Resource criteria - The conventional job scheduling algorithms assign a computing-resource rank to each node in the grid based on its resource availability. A job scheduler then uses the rank together with user-specified resource criteria to choose specific nodes for job execution. Since resource information including CPU and memory will keep varying from time to time, node-ranking information needs to be constantly reevaluated.

3) Job migration – A computing-resource file needs to be created to run parallel applications. For instance, MPI applications require the mpd.hosts file that contains a list of computing nodes. Since job migration stops the execution at the current nodes and resumes it at another set of new nodes, such a resource file needs to be recreated. However, job deployment in most conventional tools does not support dynamic creation of a resource file, hence they do not support job migration for parallel applications.

To address the above challenges, we have designed AgentTeamwork-Lite: a field-based job dispatch and migration framework. It provides a common job-scheduling framework for both sequential and parallel applications. It consists of mobile agents to broadcast resource information with UDP messages, to build an itinerary of the best computing nodes, and to monitor and move job execution to light-loaded nodes. The framework is designed with the following design principles.

1) Decentralized grid – Each participating node periodically advertises its resource information with a UDP message. AgentTeamwork-Lite forms a virtual grid using these messages which eliminates the need for a master node. A node can also be easily added or removed from the grid by starting and stopping a daemon process.

2) Resource criteria – While a local node broadcasts its own resource information, it also receives resource information from its neighboring nodes and forms the best computing node itinerary periodically. A user does not need to specify the resource criteria. In other words, AgentTeamwork-Lite will automatically select the right computing nodes to execute the user's jobs.

3) Job migration – The best node itinerary is locally available at each node. During job migration, this itinerary is used to determine a destination node on which job execution will be resumed. Just before resuming a job, an agent will automatically recreate a resource file required for the parallel execution. Once the job is resumed, it will continue the parallel execution as if it started normally.

2.2 System Design

AgentTeamwork-Lite is an enhancement to AgentTeamwork. It reuses the AgentTeamwork execution platform and agent framework. The main focus of AgentTeamwork-Lite is self-organizing resource management and performance-centric job migration. It consists of six execution layers as described in Figure 1.

Layer 0: Hardware Layer: This layer represents a list of computer nodes connected to Local Area Network. Each segment represents a subnet. They can be interconnected by WAN.

Layer 1: UDP-Broadcast Space: This is an AgentTeamwork-specific message broadcast layer that allows each computing node to exchange resource information with UDP messages. In general, UDP messages are limited with-in a single subnet. However, our broadcast layer facilitates UDP broadcast messages across a subnet by establishing a TCP link between two representative nodes in the inter-connected subnets.

Layer 2: UWAgents: UWAgents is a Java-based mobile-agent execution platform developed by Distributed Systems Laboratory at UW Bothell as part of AgentTeamwork. It consists of UWPlace and agents. UWPlace is a daemon process running on each node and the agents coordinate job execution.

Layer 3: Computing-Resource Potential Field: This layer consists of Potential-Field Agents (PFAgents) launched with a UWPlace daemon at each nodes. A PFAgent periodically broadcasts its local computing resource information including CPU power, available memory and disk space. All these pieces of information are broadcast in a UDP packet through the same subnet, and relayed to remote subnets through a UDP relay node. It also receives resource information from the neighboring nodes and calculates the best resource itinerary.

Layer 4: Commander and Sentinel Agents: This fourth layer consists of commander and sentinel agents. A commander agent is launched when a user submits a new job. It then launches a sentinel agent which is responsible for executing the user job. It also constantly queries the local PFAgent to check whether the current computing nodes are better performing nodes or not. When it finds better performing nodes than the currently executing nodes, it halts the job execution at the currently executing nodes and resumes at a set of destination nodes determined by PFAgent. Once it completes the execution, the sentinel will notify the commander agent of its job termination.

Layer 5: Middleware Libraries: The fifth layer consists of middleware libraries such as MPI [8], OpenMP [9], and MASS (which we developed for parallel simulation) libraries [12]. The sentinel agent should support the resource file creation required by these middleware libraries.

Layer 6: Applications: This layer executes user applications. User applications are responsible for periodically taking the latest data snapshot that will be used for job resumption.

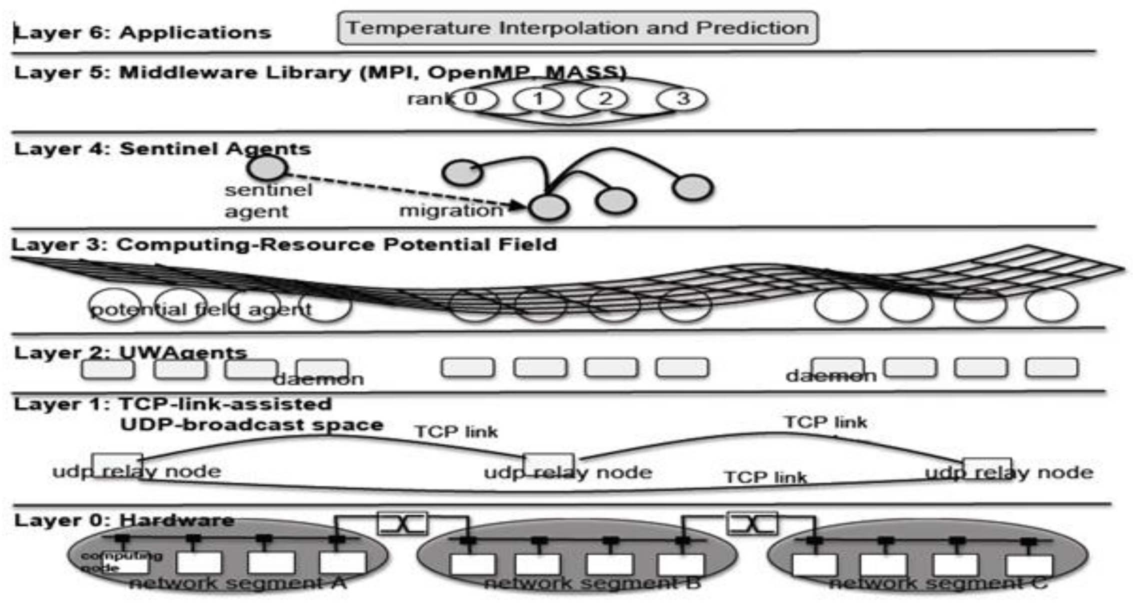

Figure 1. Execution layers

2.3 Execution Model

This section describes the workflow of job execution in AgentTeamwork-Lite. As illustrated in Figure 2, before a user job is scheduled, a daemon process (UWPlace) needs to be started at all participating computing nodes followed by starting a PFAgent at each node.

In Figure 2's scenario, a user submits a job to Node 0. UWPlace allocates a new commander agent to this job and passes the user program's name, arguments, and additional files to the commander agent. It then launches a sentinel agent locally at Node 0. Before starting a job execution, the sentinel agent checks with the local PFAgent for the best computing node. If the Node 0 is the better computing node, the sentinel will start job execution at Node 0, otherwise it will migrate to a better computing node, whereas a commander agent will still stay at Node 0. In Figure 2, the sentinel agent recognizes Node 1 as the best node, migrates to it, creates a resource file, and starts job execution. During the job execution, the sentinel agent periodically evaluates the better computing nodes with the local PFAgent. When it finds a better computing node, the sentinel suspends the current job execution and migrates to the better node, (i.e., Node 2). Resource validation and job migration are a recurring task in AgentTeamwork-Lite which lasts until the job completion. Once the job is completed, the sentinel agent will notify the commander agent (at Node 0) of this completion event. Note that a user program is responsible for periodically taking an execution snapshot in secondary storage that is accessible from all computing nodes.

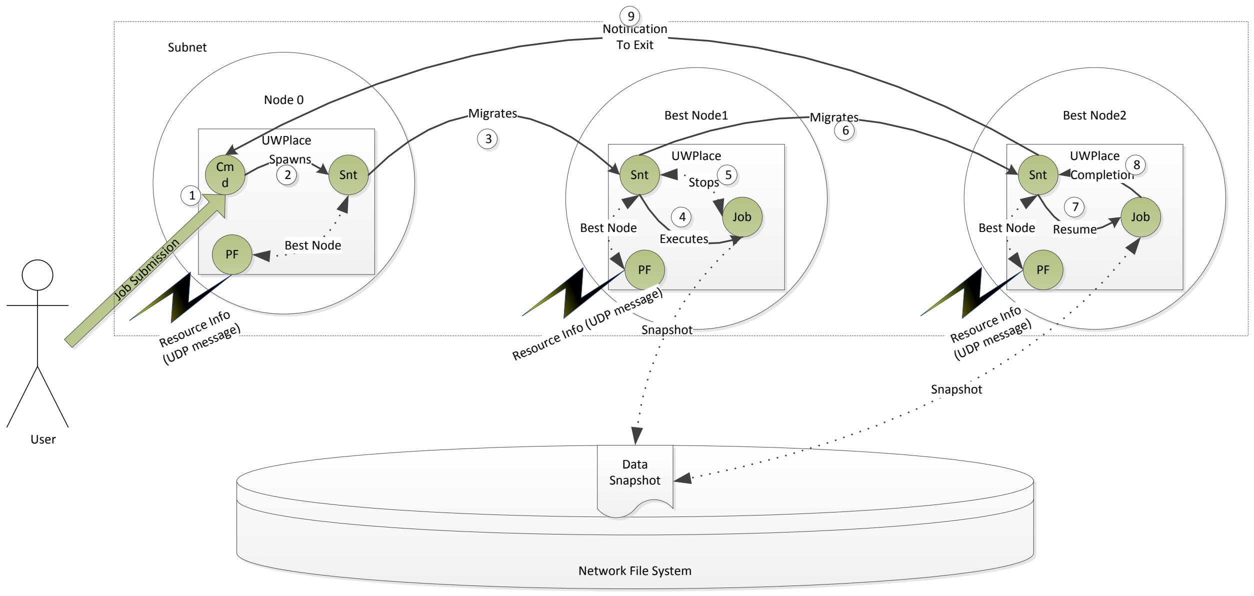

Figure 2. AgentTeamWork Execution Model

2.4 Factors affecting job execution performance

We define the measure of job execution performance as time elapsed for a job to run till its completion. Job execution performance depends on system capability and program behavior. System capability can be pre-determined with a few system attributes including CPU and memory. However, program behavior is hard to predict due to its dependency on an application and run-time environments. Although CPU and memory factors have a direct impact on job execution performance, they do not have an equal impact. To study their correlation, we want to measure job execution performance by running a real application in a controlled environment. We have used a sequential and a parallel application to study the effect of CPU and memory in terms of the application execution time. The application execution time is measured under three conditions. The first condition measures execution time when there is no CPU and memory activity, the second condition measures the time when CPU is being utilized (approximately 80%), and the final condition is to measure the time when memory is being utilized (approximately 80%).

Figure 3 shows the job execution time of a sequential application run with a single node. The average job execution time under no load is 13.99 minutes, whereas the same job on the same node took about 14.26 minutes and 14.06 minutes respectively with 80% CPU load and with 80% memory utilization. This experiment indicates that CPU load affects job execution time approximately 3.9 times more than memory load, (i.e., (14.26-13.99)/(14.06-13.99) = 3.85).

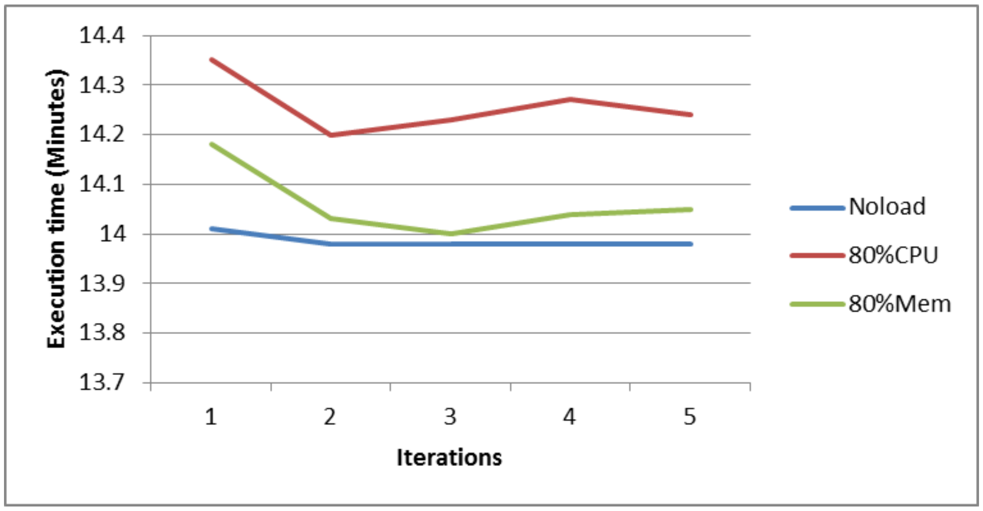

Figure 3. Mandelbrot Execution performance CPU vs. Memory load factors

Figure 4 shows the time elapsed to execute Wave2MPI (an MPI program) with 3 nodes. The average job execution time in an ideal scenario is 15.09 minutes, whereas the execution time increased to 17.04 and 15.58 minutes respectively with 80% CPU load and with 80% memory load. This experiment indicates that 80% of CPU load has affected job execution time four times more than 80% memory load, (i.e., (17.04-15.09)/(15.58-15.09)=3.98).

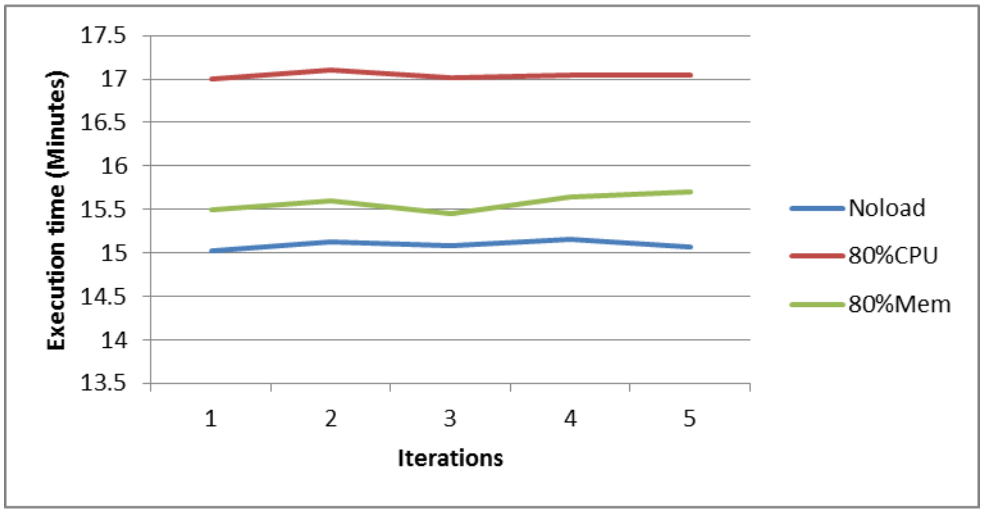

Figure 4. Wave2DMPI Execution performance CPU vs. Memory load factors

These experiments indicate that CPU and memory load do not have an equal impact to application execution time and the ratio between CPU and memory factors varies between applications and nodes. However, CPU load always surpass the memory load in affecting job execution time and the ratio is always more than three. Hence, we have given three times higher weight to CPU power than memory power when ranking each computing node. At the same time, we have confirmed that CPU load does not affect job execution performance until CPU utilization exceeds 80%. The same way memory load does not affect job execution performance until memory consumption exceeds 80%.

2.5 Migration algorithm

Job migration is the process of moving a job from one to another computing node. It is a form of dynamic load balancing in distributed computing. A migration algorithm is a policy to govern job migration, basically to determine timing and a destination to migrate, (i.e., when and where to migrate) a job. The following describes our migration algorithm:

i) When to migrate

Timing to migrate a job is determined by the rank of the current executing node. A PFAgent calculates its local node *i*'s rank from the system information as follows

$$\text{rank}\ (r_i) = \text{cpu\_idle}[\%] + \left({}^{\text{memory\_free}[\%]}/_{3}\right)$$

In Linux, the sar (System Activity Report) command is used to report system loads. “sar 1 1” gives the CPU load information in the last one minute. The CPU idle percentage is calculated by parsing the sar output.

The process status command (ps aux) gives the details of all processes of all users in terms of CPU and memory information. Memory free percentage is computed from its output column “RSS” (real memory size). Since our experiments indicated that, CPU power has a bigger impact to job execution performance than memory usage, we have given a higher weight to CPU, (i.e., three times more than memory). The above experiments also indicated that job execution performance is not affected until CPU load or memory load reaches 80%. This means that job execution performance will get affected when the current node rank becomes 26 or less (rank = 20% + (20%/3) = 26%). Each PFAgent broadcasts its local system’s information every 60 seconds and estimates the best node every two minutes. To mitigate this time lag, we want to add some buffer to this rank. Therefore we decided to move a job when the current node’s rank becomes less than or equal to 45 (**rank (r_i) <= 45**).

ii) Where to migrate

80% CPU load on a 1 GHz CPU is not the same as 80% CPU load on 2 GHz CPU. Hence, we cannot depend on each node’s individual rank to determine the best node. We need to find out each node’s relative rank with respect to its peers. The relative rank (R_i) of a node *i* is calculated as follows

CPU power (c_i) of a node *i* is calculated as

$$Cpupower\ (c_i)\ =\ cpu\_idle[\%]\ *\ \#\,CPUs\ *\ \#\,Cores\ *\ cpuSpeed$$

The CPU idle percentage is calculated from “sar”’s output together with other CPU-related information, the number of CPUs (# CPUs), the number of CPU cores (#Cores), and the CPU speed are calculated from linux command “proc/cpuinfo”.

Memory power (m_i) of a node *i* is calculated as

$$Mempower\ (m_i)\ =\ mem\_free\%\ *\ TotalMemory$$

Total memory is calculated from linux command “free –m”.

$$RelativeRank(R_i) = \left(\frac{cpupower(c_i)}{\max(cpupower)}\right) + \left(\left(\frac{mempower(m_i)}{\max(mempower)}\right) \Big/ 3 \right)$$

Here max(cpupower) and max(mempower) refers to the maximum CPU power and maximum memory power of all the nodes in a user-defined cloud space. Once the relative rank is calculated, each PFAgent sorts them in an ascending order. The top node in the sorted list is the higher performance computing node. In summary, a node’s individual rank (r_i) and relative rank (R_i) are used respectively when and where to migrate.

Chapter 3: Performance Evaluation

We have used Wave2DMPI and Mandelbrot as benchmark applications to study difference between AgentTeamework-Lite's migration-based job scheduling and the default (i.e., static) job scheduling in terms of execution. The default scheduling uses the same set of computing nodes until the completion of a given job, whereas migration-based scheduling moves the job execution to better performing nodes.

Wave2DMPI is a two dimensional wave simulation application based on Schroedinger's equation that calculates wave height on each cell, using its surrounding cells' wave height. This application is parallelized with the Java MPI libraries [10]. The parallelization approach partitions a simulation space in smaller stripes, each assigned to a different node.

Figure 5 illustrates the Wave2DMPI's execution time with the default scheduling and migration-based scheduling. This job is executed with three computing nodes and the simulation size was set to 1000 units. The figure indicates that migration-based scheduling completes the job execution approximately 40% faster than the default scheduling.

3.1 Default vs. Migration based scheduling

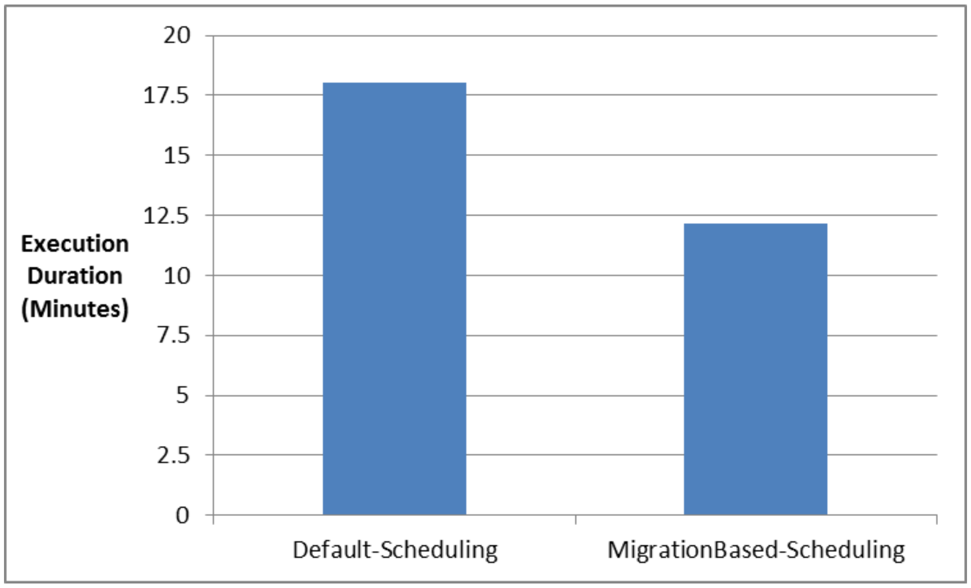

Figure 5. Wave2DMPI Default vs. Migration based scheduling Execution

Figure 6 records snapshots of the three computing nodes' individual rank (r_i) that participated in the default scheduling when executing Wave2DMPI. During the course of execution, the nodes' ranks kept fluctuating due to their varying system performance. In the most time, the individual rank (r_i) of these nodes was hovering around 40, which means that other applications running on these nodes consumed the system resources heavily (approximately 80%). This clearly explains why we see the longer execution time with the default scheduling.

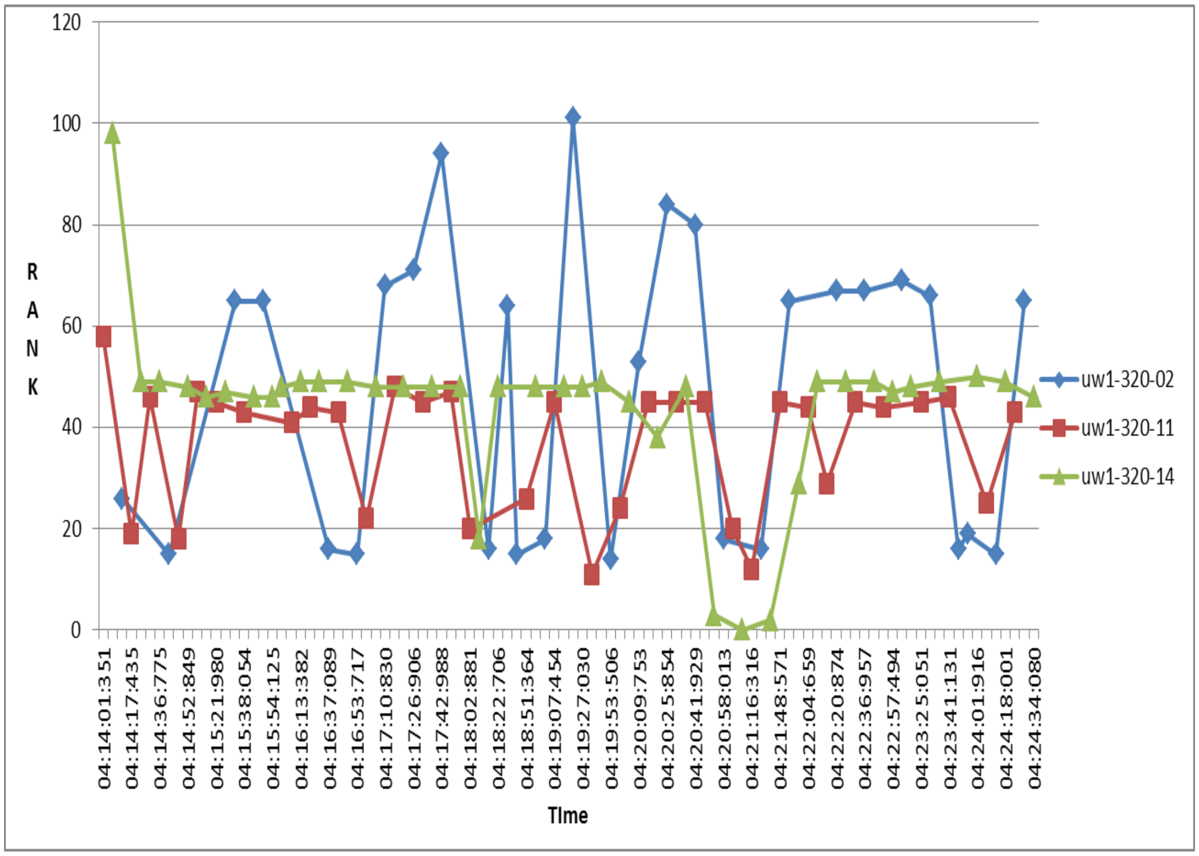

Figure 6. Wave2DMPI Execution nodes' rank (Default scheduling)

Figure 7 illustrated a job migration itinerary when executing Wave2DMPI with AgentTeamwork-Lite's migration-based scheduling. It uses a combination of preemptive and non-preemptive job migration. Preemptive job migration means that the job is preempted, forced to migrate, and resumed at a different node. Non-preemptive job migration takes place before job execution (i.e., initial job execution). In the figure, a job is initially submitted to the uw1-320-00 node, which spawns commander and sentinel agents. Since the sentinel agent detects a better computing node than the uw1-320-00 node, it migrates to a next node in the itinerary that is uw1-320-22 (in non-preemptive migration), creates a resource file, and starts the execution. After a while, the sentinel agent detects that the current executing node's (uw1-320-22's) individual rank (r_i) is less than 46, thus stops the job execution, and migrates to a next node in the itinerary which is uw1-320-19. Since this migration enables the program to execute on better computing nodes, its execution time is faster than the default scheduling.

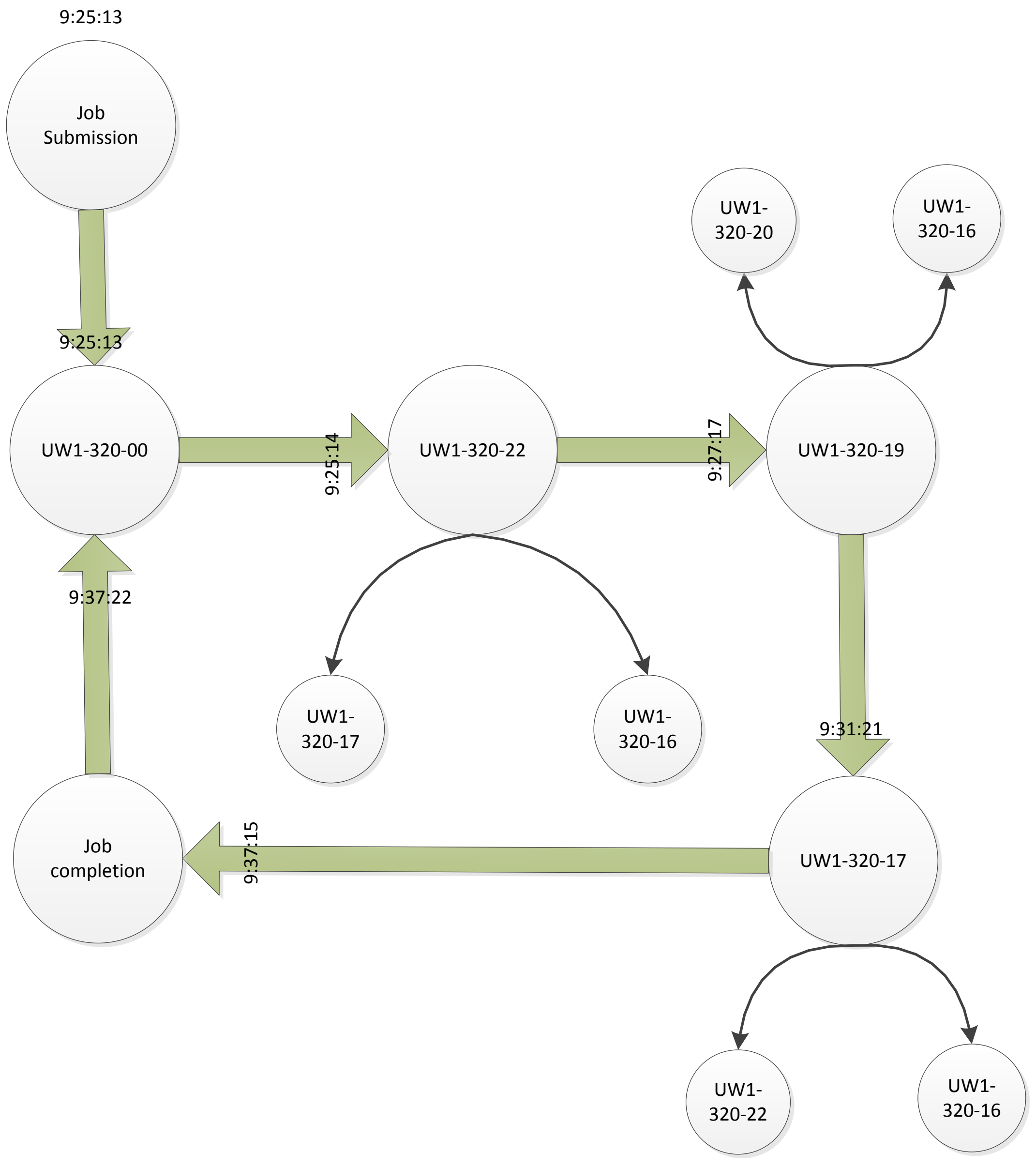

Figure 7. Wave2DMPI Migration based scheduling execution

We have also conducted a performance evaluation with a sequential application (i.e., a Mandelbrot fractal generator). Fractals are a geometric shape where each pixel refers to a point in a Mandelbrot set [11]. A point is considered to be in the Mandelbrot set if it converges to a number after a predefined number of iterations which requires computational intensive calculation. For example, a 600X800 pixel image can use up to maximum iterations of 500,000 for each of its pixel. This program is developed in Java programming language.

Figure 8 illustrates Mandelbrot's execution time with the default scheduling and migration-based scheduling. This job was executed with a single computing node to generate a 1000X1000 pixel fractal image with the max iterations set to 200,000. The figure indicates that the migration-based scheduling completes the job execution approximately 15% faster than the default scheduling.

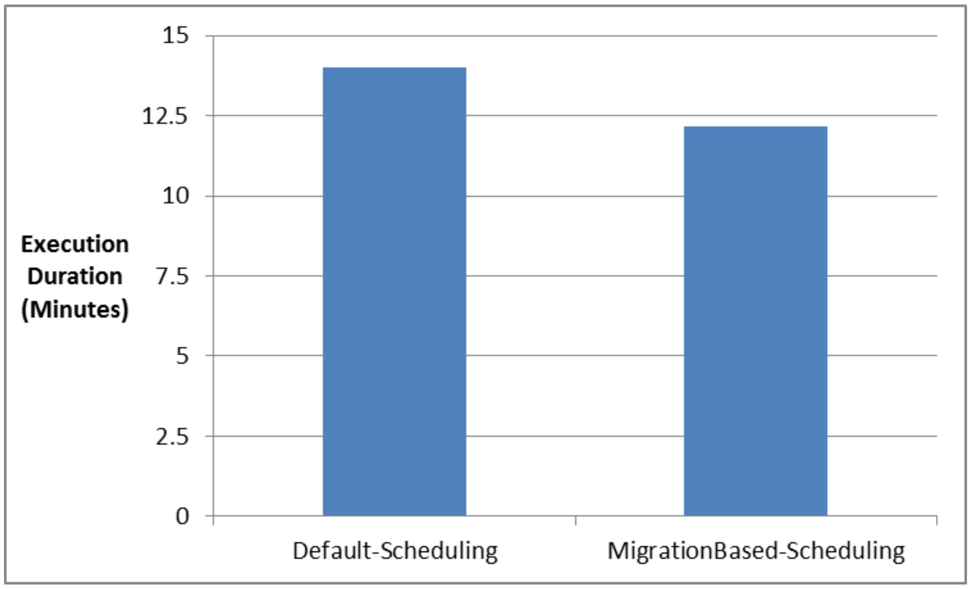

Figure 8. Mandelbrot Fractal Default vs. Migration based scheduling Execution

Figure 9 records the snapshots of the computing node's individual rank (r_i) that participate in the default scheduling when executing the Mandelbrot fractal. In this case, before executing the job, the node's individual rank (r_i) was at 50, but during program execution, it dropped to below 10. At the end of the execution, the rank comes back to 50 levels again. Most likely Mandelbrot application is consuming the system resources heavily that caused the individual rank to drop from 50 to 10.

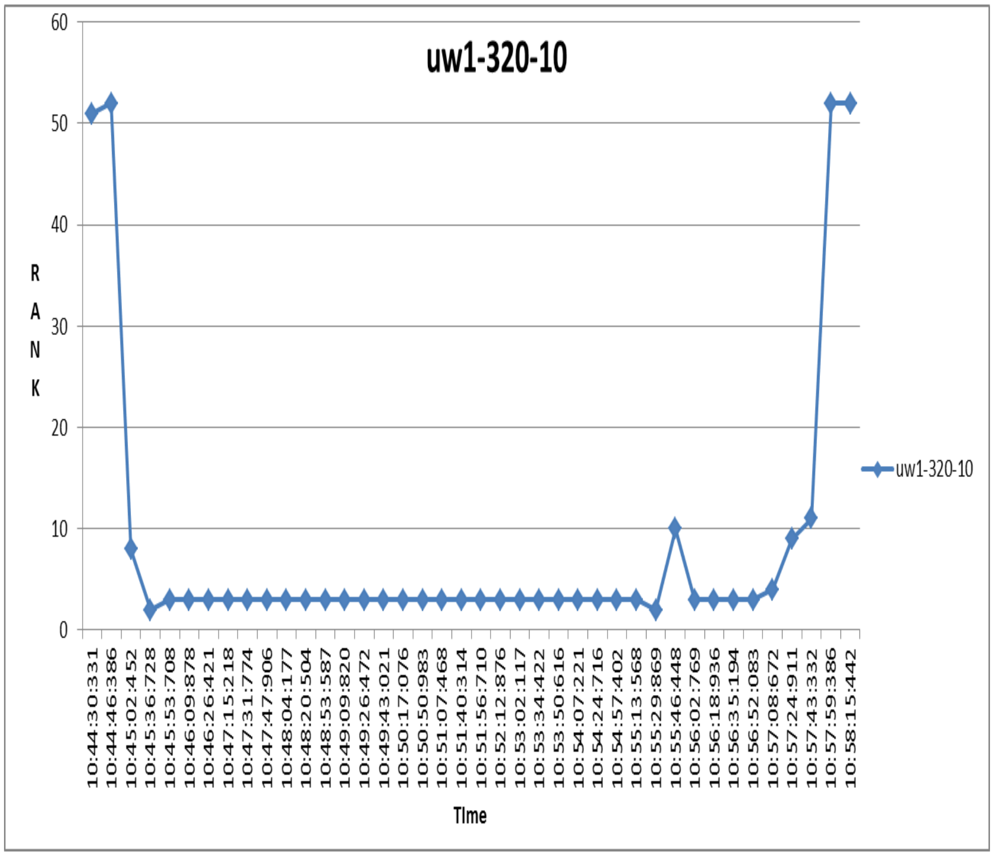

Figure 9. Mandelbrot Fractal Execution node's rank (Default scheduling)

Figure 10 illustrates a job migration itinerary when executing Mandelbrot program with AgentTeamwork-Lite's migration-based scheduling. In the figure, a job was initially submitted to the UW1-320-00 node that spawned a commander and sentinel agents. Since the sentinel agent detected that there are higher-rank nodes than the UW1-320-00 node, it migrated to a higher-ranked node in the itinerary that was UW1-320-20 and started the job execution. After a while, the individual rank (r_i) of the node UW1-320-20 dropped less than 46, hence the sentinel agent stopped the job execution, and migrated to a higher-ranked node in the itinerary, which was UW1-320-16. Even with the sequential job execution, we confirmed that AgentTeamwork-Lite's migration-based job scheduling was executing job faster than the default scheduling.

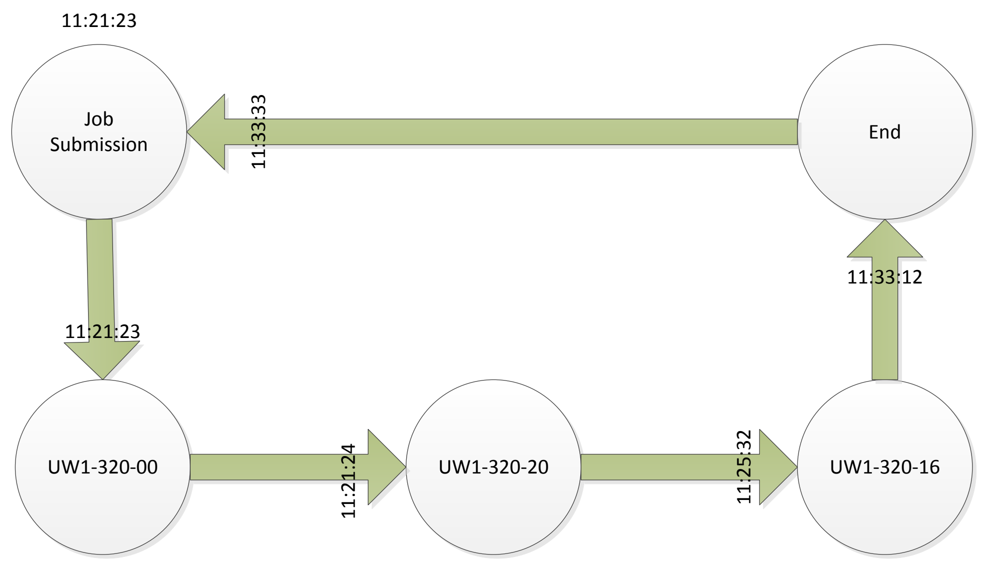

Figure 10. Mandelbrot Fractal Migration based scheduling execution

3.2 AgentTeamwork-Lite overhead

We analyzed AgentTeamwork-Lite's overhead in terms of job execution cost and migration cost. The job execution cost is the time elapsed to run a job until its completion. The time elapsed to complete a job execution with AgentTeamwork-Lite is a few seconds more compare to direct execution. Secondly, the sentinel agent checks the job completion at a regular interval (i.e., every two seconds). Therefore, the commander agent will always receive the delayed job completion notification that can be up to the maximum of two seconds. Figure 11 illustrates the Wave2DMPI's execution time with the direct and AgentTeamwork-Lite's execution. The average execution time difference between direct and AgentTeamwork-Lite's execution is 24 seconds.

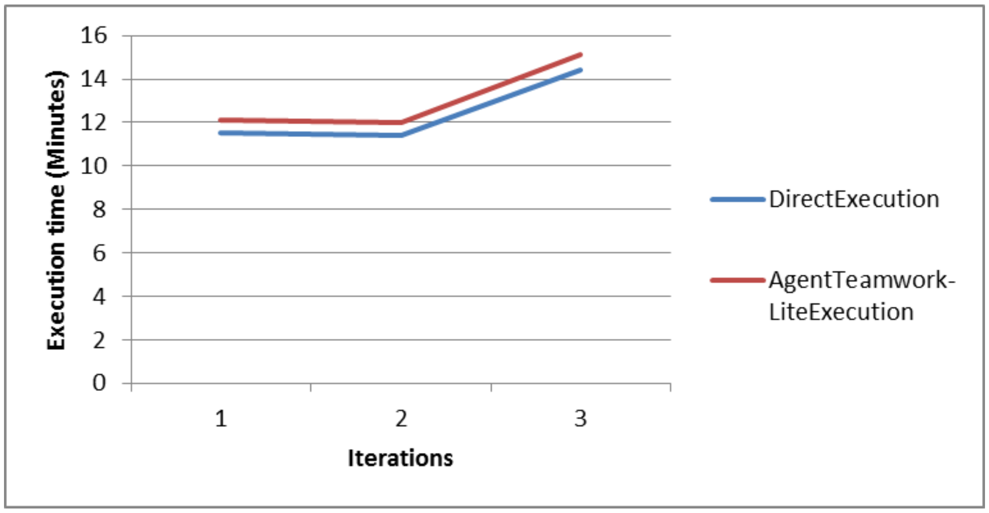

Figure 11. Direct vs. AgentTeamwork-Lite execution

The job migration cost is the total time involved in stopping the job execution and resumes at a different node. During the job migration neither the data nor the code is transferred over the network. It is just that the sentinel agent exits form the current executing node and starts at a remote node. When the sentinel agent resumes at a remote node, it passes additional command line switch "resume" to the user program that resumes the program execution with latest data snapshot. Hence, the job migration cost is in negligible amount (i.e., few milliseconds). Figure 12 represents job migration cost of Wave2DMPI with AgentTeamwork-Lite.

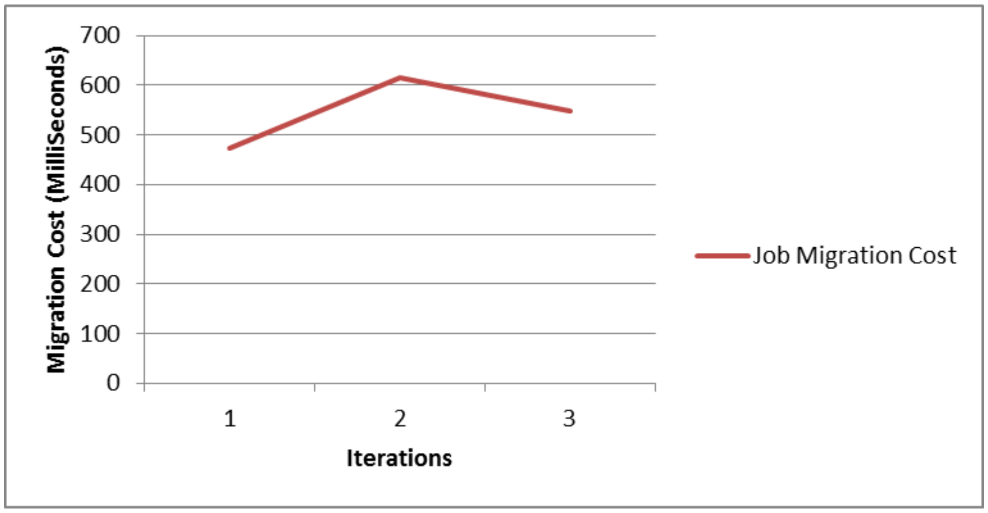

Figure 12. Job migration cost of Wave2DMPI

Chapter 4: Related work

In this section, we differentiate AgentTeamwork-Lite from its related systems in terms of migration-based scheduling, parallel application scheduling and resource discovery.

4.1 GridLab Resource Management System (GRMS)

GridLab Resource Management (GRMS) [12] is an open source dynamic grid scheduling with job migration and rescheduling system based on Globus Toolkit 2.4 [13].GRMS connects to the low-level Globus services deployed onto remote resources through a set of Java and C APIs that allow to implement various scheduling and policy plugins. Let us focus on GRMS' reschedule policy plugin that is closer to AgentTeamwork-Lite's job migration. The reschedule policy plugin checkpoints and moves a running job in order to release the amount of resources required by a pending job in queue.

Both GRMS and AgentTeamwork-Lite uses an application-based check-pointing approach. The difference is that AgentTeamwork-Lite's job migration aims at improving the currently running application's execution time, whereas GRMS focuses on the current job migration to schedule a newly incoming application when the system suffers from the lack of resources. Secondly, GRMS requires user to specify resource information along with a job submission request, which is not required in AgentTeamwork-Lite. In GRMS, resource discovery module uses both central (GIIS) and local information services (GRIS), whereas in AgentTeamwork-Lite, resource discovery is done through local PFAgent's broadcast message.

4.2 Condor

Condor [14] is a workload management system specialized for compute intensive jobs. It facilitates a job queuing mechanism, scheduling policies, priority schemes, resource monitoring, and resource management. A condor-pool consists of a master node that runs as a central manager, and a number of other machines that join the pool as participating resources. The central manager periodically receives status updates from the machines that are part of the pool, and does match making for pending job requests with appropriate, available resources. Condor moves job execution when the current executing node does not meet user-specified resource requirements. A checkpoint (system-level check-pointed) image is generated whenever it detects to move a job from one to another machine. A program must be linkage-edited with the Condor compiler in order for the Condor libraries to intercept system calls and to perform check-pointing while the program is running. However, the program should not invoke multi-process calls (namely fork(), system(), etc..), inter-process communication, network communication, alarms, and timers. Hence, Condor does not currently support job migration for parallel and multi-process applications. Note that Condor-MW used to support master-worker parallel programs where the master process had to take care of all snapshots of the worker-processes.

Contrary to centralized resource monitoring in Condor, AgentTeamwork-Lite performs decentralized resource monitoring by allowing each node to broadcast its computing-resource information. Since AgentTeamwork-Lite uses application-level check-pointing (in each of multiple processes) and creates a resource file such as mpd.hosts in MPI, it supports job migration for both sequential and parallel applications.

Chapter 5: Conclusion

Our main focus on this research was to develop a migration-based distributed job scheduling and monitoring framework using mobile agents. This thesis presented AgentTeamwork-Lite's system design and its execution model. Its unique design relieves users from specifying the computing resource requirements for their job execution. Our analysis has also demonstrated the advantages of AgentTeamwork-Lite's migration-based job scheduling over default scheduling as well as its capability to schedule both sequential and parallel applications. Our future work includes the following areas.

1. Check-pointing wrapper:

We must provide a common check-pointing wrapper library and interface to free application developers from writing their own check-pointing implementation that is currently not standardized and difficult. Secondly, the common interface will allow the AgentTeamwork-Lite to invoke the check-pointing function in a regular interval instead of having applications initiate checkpoints.

2. Experiment in Cloud:

Though AgentTeamwork-Lite has capability to work with cloud, we have not yet tried that in the actual cloud. We have evaluated AgentTeamwork-Lite performance only in the UWB distributed system laboratory. The next step is to test AgentTeamwork-Lite with cloud services including Amazon EC2.

3. MASS Scheduling:

AgentTeamwork-Lite currently supports scheduling Java and MPI applications. However, it can be easily enhanced to schedule any programming including MASS.

4. AgentTeamwork-Lite portal:

AgentTeamwork-Lite provides command line utilities to start daemon processes and to schedule jobs. We need a portal to allow users to submit, and schedule their jobs over the internet. This portal is currently under development in the UWB distributed system laboratory.

References

[1] M. Fukuda, C. Ngo, E. Mak and J. Morisaki, "Resource Management and Monitoring in AgentTeamwork Grid Computing Middleware," in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, August 2007.

[2] Z. Zhang and X. Zhang, "Realization of open cloud computing federation based on mobile agent," in *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference*, Shanghai, December 2009.

[3] B. S. a. E.-N. H. M. M. Hassan, "A framework of sensor-cloud integration opportunities and challenges.," in *In Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication, ACM*, January 2009.

[4] R. Jedermann, J. Palafox-Albarran, J. Robla, P. Barreiro, L. Ruiz-Garcia and W. Lang, "Interpolation of spatial temperature profiles by sensor networks," in *Sensors, 2011 IEEE*, Limerick, October 2011.

[5] G. Li, T.-M. Chan, K.-S. Leung and K.-H. Lee, "A Cluster Refinement Algorithm for Motif Discovery," *IEEE/ACM Transactions on Computational Biology and Bioinformatics, ,* vol. v7, pp. 654-668, Oct.-Dec. 2010.

[6] E. P. Salathé, L. R. Leung, Y. Qian and Y. Zhang, "Regional climate model projections for the State of Washington," *Climatic Change ,* vol. 102, no. 1/2, p. 51, Sep2010.

[7] D. L. Chao, M. E. Halloran, V. J. Obenchain and I. M. L. Jr, "FluTE, a Publicly Available Stochastic Influenza Epidemic Simulation Model," *PLoS Computational Biology,* vol. 6, no. 1, p. 1, Jan2010.

[8] C. Liu, Z. Zhao and F. Liu, "An Insight into the Architecture of Condor - A Distributed Scheduler," in *Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium*, Wuhan, January 2009.

[9] H. Morohoshi and R. Huang, "A user-friendly platform for developing grid services over Globus Toolkit 3," in *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference*, July 2005.

[10] R. L. Graham, B. W. Barrett, G. M. Shipman, T. S. Woodall and G. Bosilca, "Open MPI," *Parallel Processing Letters,* pp. 79-88, 2007.

[11] B. M. Chapman and F. Massaioli, "OpenMP," *Parallel Computing,* vol. 31, no. 10-12, p. p957, Oct 2005.

[12] 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.

[13] M. Baker, B. Carpenter, G. Fox and S. H. Koo, "mpiJava: An Object-Oriented Java Interface to MPI," *Lecture notes in computer science,* no. 1586, p. 748, 1999.

[14] "The Mandelbrot set," *Australian Mathematics Teacher,* vol. 67, no. 4, p. 36, 2011.

[15] K. Kurowski, B. Ludwiczak, J. Nabrzyski, A. Oleksiak and J. Pukacki, "Dynamic grid scheduling with job migration and rescheduling in the GridLab resource management system," *Scientific Programming,* vol. 12, no. 4, 2004.

[16] [Online]. Available: http://www.globus.org/.

[17] T. T. M. L. Douglas Thain, "Condor and the Grid," in *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons Inc., 2002 December.

Appendix A: Check-pointing Source Code

The following the check-pointing source code of Wave2DMPI program.

```
       // Saves current simulation space and system time into either data1.ser or
       // data2.ser, whichever file is older
    private void saveToFile() throws Exception{
       double space[][] = null;
        ObjectOutput out = null;
        boolean wroteTime = false;

        // rank 0 initializes output
        if (myRank == 0) {
            // for debug
            File output = swapFile( false );      // get file to save to
            System.out.println("saving to " + output);

            out = new ObjectOutputStream(new FileOutputStream
                        ( output ));
        }
        // collect sim spaces from 3 time periods: z[0], z[1], and z[2]
        for ( int period = 0; period < 3; period++ ) {
            space = collect( period);       // collect sim space from sub ranks

            // only rank 0 is responsible to save it.
            if ( myRank == 0) {
                try {
                    if (!wroteTime) {
                        out.writeInt(time);      // save current system time
                        wroteTime = true;
                    }

                    out.writeObject(space);          // save simulation space
                } catch (Exception e) {
                        System.err.println(e);          // print out error
                }
            }
        }
        if ( myRank == 0) out.close();                  // rank 0 closes output
    }

    private File swapFile( boolean resuming) throws IOException{
        File a = new File("data1.ser");
        File b = new File("data2.ser");
        // create files if they don't exist
        a.createNewFile();
        b.createNewFile();

        /*System.err.println("length : " + a.length() + "\t " + b.length());
       System.err.println("Datemodified : " + a.lastModified() + "\t " +
b.lastModified());*/
        // check if one of the file is empty to avoid EOF exception on resuming
        if ( resuming){
            if ( a.length() == 0 || b.length() == 0 )
                return ( a.length() == 0 ) ? b : a;
            else
            {
              if ( a.lastModified() > b.lastModified())
              {
                    if(a.length() >= (b.length()/2))
                           return a;
                    else
```

```
                        return b;
             }
             else
             {
                    if(b.length() >= (a.length()/2))
                        return b;
                    else
                        return a;

             }
           }

       }

       // return the older file to be overwritten or resumed from
       return ( a.lastModified() > b.lastModified() ) ? b : a;
   }


   // Reads in previously saved time and simulation space data from
   // either data1.ser or data2.ser
   private void readFromFile() {
      resume = true;
       double[][] space = null;

       try {
           // only rank 0 is responsible to read from file
           if ( myRank == 0 ) {
               // for debug
               String fileName = swapFile( true ).getName();
               System.out.println("resuming from " + fileName);

               // read simulation values from the older data file
               ObjectInput in = new ObjectInputStream ( new FileInputStream
                       ( fileName ));
              // System.out.println( "read time" );
               this.time = (int)in.readInt();         // set sim time
               //System.out.println( "set time" );

               // distribute the sub space and sys time for every period
               for (int period = 0; period < 3; period++ ) {
                   space = (double[][]) in.readObject();   // set sim space
                   distribute( space, period);             // send space
               }
               in.close();
           }

           else    // other processes receive sub spaces
               for (int period = 0; period < 3; period++) {
                   distribute( space, period); // get sub spaces
               }

       } catch (Exception e) {
          System.out.println("error at readFromFile");
           System.err.println(e);
       }
   }
```

Appendix B: User manual

1. To launch daemon process run the following command line

*/net/metis/home3/dslab/FieldBaseMigration/agents/drop/java -cp *:. AgentLauncher uwplace /classpath $(pwd) /command launch /portnumber <portNumber> /user <useraccount> /password <password> [/nodes <nodeslist>]*

Arguments:

portnumber – Pornumber on which UWPlace daemon process communicates (Example: 12345)
useraccount – User account to run daemon process. Before use this account, please make sure that ssh profile is saved in all participating nodes to allow remote login without providing the password every time.
password - password for the user account

The above command line launches UWPlace daemon process in all nodes returned by "/etc/hosts" command. If we need to restrict that to specific nodes, we can supply the optional switch "/nodes" followed by comma separated node names (Example: /nodes mnode1.uwb.edu,mnode2.uwb.edu).

PFAgent should be started immediately after the daemon process to broadcast resource information

*/net/metis/home3/dslab/FieldBaseMigration/agents/drop/java -cp *:. AgentLauncher pfagent /classpath /classpath $(pwd) /command launch /portnumber <portNumber> /user <useraccount> /password <password> [/nodes <nodeslist>]*

2. To schedule a user program run the following command line

/net/metis/home3/dslab/FieldBaseMigration/agents/drop/schedule.sh $1 $2 $3 $4 $5 $6 $7 $8 $9

Arguments:

Arg1 ($1) – Full path to AgentTeamwork-Lite's code

(use : /net/metis/home3/dslab/FieldBaseMigration/agents/drop)

Arg2 ($2) – Port number to communicate to UWPlace daemon process (should be same as above)

Arg3 ($3) – User account to execute the job

Arg4 ($4) – Password for the user account

Arg5 ($5) – Full path to job folder

(Example: /net/metis/home3/dslab/FieldBaseMigration/mandelbrot)

Arg6 ($6) - Number of computing nodes (1 for sequential program)

Arg7 ($7) – Command line (Example: prunjava 3 or java -cp *:.)

Arg8 ($8) – User program and Arguments

(Example: Mandellel.class_-1.5_0.5_0.0_1000_1000_100000_GRAD-RED_agenthosts.png)

Arg9 ($9): Additional files to upload to job execution directory. It should precede by “AF_”

(Example: AF_/net/metis/home3/dslab/NetBeansProjects/CSS497Project/upload/dslab@uw1-320-lab.uwb.edu/Mandelbrot/WorkContract.class)