Field-Based Job Dispatch and Migration

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

***Abstract*—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 performance 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 simulation and the Mandelbrot fractal generator benchmark programs.**

I. Introduction

The recent emergence of cloud services [1] has allowed more users to access parallel-computing resources by simply purchasing as many compute instances as needed, thus without any initial hardware investment. Since such cloud services also facilitate common programming and job scheduling environments such as OpenMP [2], MPI [3], OpenPBS [4], and Condor [5], users can easily develop and execute their parallel applications. However, the more users compete computing resources, the more load unbalance tends to occur at run time, which may unnecessarily delay any jobs. Needless to say, we need a run-time environment to reallocate a user job to faster or lighter loaded computing nodes. However, unless an application is parallelized with paradigm-oriented libraries such as MapReduce [6], most cloud services follow a centralized scheduling strategy that statically allocates computing resources to a job upon its invocation. Therefore the job tends to keep running on the same set of computing nodes even in the case if they may no longer meet the user-specified resource criteria. This motivated us to develop a migration-based job-scheduling tool to work with conventional parallel applications, (e.g. MPI programs).

We previously developed the AgentTeamwork parallel-computing middleware system [7] that used mobile agents to dispatch, launch, monitor, and move a user job at remote computing nodes, particularly focusing on fault tolerance. The problem of its implementation was the use of a static list of computing nodes for resuming a parallel-computing job at more robust nodes upon abnormality. We slimmed the system down by removing its expensive fault-tolerance feature but facilitated performance-oriented job migration to light-loaded nodes as AgentTeamwork-Lite. The project 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 move a job, based on dynamically evaluated criteria,
- Finding out the job migration cost and the best timing to migrate a job
- Demonstrating the performance gain of field-based job migration scheduling over default scheduling.

This research project 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. To demonstrate the efficiency of our migration-based scheduling algorithm, we have compared execution performance between our migration algorithm and the default job scheduling, (i.e., static) in terms of serial and parallel program execution.

The rest of this paper is structured as follows: Section 2 discusses the system design, performance-affecting factors, and our migration algorithm; Section 3 shows our performance evaluation; Section 4 differentiates AgentTeamwork-Lite from the related work; and Section 5 concludes our discussions with the future work to be done.

II. Methods

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

A. Challenges and Solution in Job Scheduling and Coordination

Run-time job scheduling and migration has the three following challenges.

1) *Centralized scheduling:* Most job schedulers use the master node to keep track of each job execution. This centralization behaves as a performance bottleneck and a focal point of errors.
2) *Static scheduling:* Conventional parallel applications such as MPI programs keep running on the same machines until the end of the computation. This static scheduling causes a run-time mismatch between resource requirements and availability.
3) *Job migration:* In general, a parallel job requires a user or a job scheduler to declare machines to be used in a configuration file such as *mpd.hosts* in MPI. Job migration must be able to change this configuration file dynamically.

To address the above challenges, we have designed Agent Teamwork-Lite: a field-based job dispatch and migration framework. 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 so as to virtually create a computing-resource potential field, which eliminates the master node, adds new nodes, and remove unavailable nodes in a distributed fashion.
2) *Resource criteria:* - Based on the computing-resource potential field, each node finds where a job should migrate. Therefore, users do not need to specify destination nodes to run their job.
3) *Job migration:* - A mobile agent automatically carries an assigned user job with it, regenerates a configuration file for new destinations, and resumes the job at the destination nodes. Therefore, all users have to do is what data to capture and to retrieve upon a migration.

B. 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: The hardware layer represents a list of computer nodes connected to Local Area Network. Each segment represents a subnet. They are interconnected by WAN.

Layer 1: The UDP-broadcast space 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 within a single subnet. However, our broadcast layer facilitates UDP broadcast across subnets by establishing a TCP link between two representative nodes in any pair of subnets.

Layer 2: UWAgents is a Java-based mobile-agent execution platform we developed as part of AgentTeamwork [8]. Each computing node runs a daemon process (named *UWPlace*) to exchange agents.

Layer 3: The computing-resource potential field consists of Potential-Field Agents (PFAgents) launched on a UWPlace daemon at each node. A PFAgent periodically broadcasts its local computing resource information including CPU power, available memory size, and disk space. Such information is broadcast in a UDP packet through the same subnet, relayed to remote subnets through UDP relay nodes, and eventually delivered to all remote nodes to calculate the best resource itinerary.

Layer 4: Commander and sentinel agents accept and move a user job respectively. Receiving a user job from a commander,

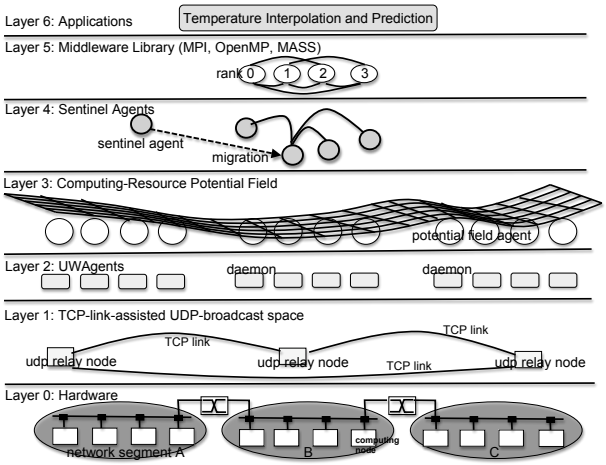

Fig. 1. Execution layer.

a sentinel migrates over the computing-resource field to the highest performance computing nodes for executing the job as if they were electrons sliding down on an electric field. The sentinel agent keeps interacting with PFAgents to move the ongoing job to even better nodes until it notifies the commander of the job completion.

Layer 5: Middleware libraries includes MPI [3], OpenMP [2], and MASS [9] (which we developed for parallel simulation) libraries. The sentinel agent should support the resource file creation that is required by these middleware libraries.

Layer 6: The application layer corresponds to user programs. They are responsible for periodical snapshot capturing for job migration purposes.

C. Performance-Affecting Factors in Job Execution

We define the measure of job execution performance as the time elapsed for a job to run until 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 measured job execution performance by running a real application in a controlled environment. The application execution time was measured under the following three conditions: (1) no CPU and memory load, (2) 80% CPU load only, and (3) 80% memory load only. For all the experiments shown in the rest of this paper, we have used computing nodes, each with the specification summarized in Table I.

Table II shows average job execution time of Mandelbrot and Wave2D MPI, each respectively running with a single and three computing nodes. In Mandelbrot, (i.e., a sequential program), the average time among five executions under no load, 80% memory load, and 80% CPU load was 13.99, 14.06,

Resources	Specification
CPU	Xeon @ 1.8GHz with 1MB cache
#Cores	4 in total
Memory	2GB
Network	1Gbps

TABLE I. COMPUTING NODE SPECIFICATION USED FOR ALL EXPERIMENTS

and 14.26 minutes respectively. This experiment indicates that CPU load affected job execution time approximately 3.8 times more than memory load. In Wave2D MPI, (i.e., a parallel program), the average execution time under no load, 80% memory load, and 80% CPU load was 15.09, 15.58, and 17.09 minutes respectively. The parallel execution showed the similar trend: 80% of CPU load affected job execution four times more than 80% memory load.

Execution	Sequential	Parallel
Programs	Mandelbrot min (slow-down)	Wave2D MPI min (slow-down)
No load	13.99	15.09
80% Memory	14.06 (0.5%)	15.58 (3.2%)
80% CPU	14.26 (1.7%)	17.04 (12.9%)
Slow-down ratio	$1.7/0.5 = 3.8$	$12.9/3.2 = 4.0$

TABLE II. EXECUTION PERFORMANCE UNDER DIFFERENT CPU AND MEMORY LOADS

These experiments indicate that CPU and memory load do not have an equal impact to application execution time. The ratio between CPU and memory factors varies between applications and nodes. However, CPU load affects job execution at least three times more than memory load. Hence, we will give a three-time higher weight to CPU power than memory power when ranking each computing node. At the same time, we have confirmed that both CPU and memory loads do not affect job execution performance until their utilization exceeds 80%.

D. Migration Algorithm

We define our migration algorithm as a policy to govern job migration, basically to determine a timing and a destination to migrate, (i.e., when and where to migrate) a job.

1) **When to migrate**
It 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

$$r_i = \textit{cpu\_idle}[\%] + \frac{\textit{memory\_free}[\%]}{3} \quad (1)$$

In Linux, the *"sar 1 1"* (system activity report) command gives the CPU load information in the last one minute. The CPU idle percentage is calculated by parsing the sar output. The *"ps aux"* (process status) command 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). Based on our experiments in Section II-C, we have given a three-time higher weight to CPU than memory. Since our experiments have also confirmed that 80% or more CPU and memory utilization, (in other words, 20% or less CPU and memory availability) slows down job execution, we should move a job when the current node rank drops down below $rank = 20\% + (20\%/3) = 26\%$ or less. However, there is a time lag: each PFAgent broadcasts its local system information every 60 seconds and estimates the best node every two minutes. Therefore, we should add some buffer to this rank. For this reason, we decided to move a job when the current node's rank drops down to or below 45, (i.e., $rank(r_i) \leq 45\%$).

2) **Where to migrate**
80% CPU load on a 1GHz CPU is not the same as 80% CPU load on a 2GHz 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 node i is calculated as follows.
CPU power (c_i) of a node i is calculated as:

$$c_i = \textit{cpu\_idle}[\%] \times \#CPUs \times \#cores \times \textit{cpu\_speed} \quad (2)$$

The CPU idle percentage is calculated from *"sar"* output together with other CPU-related information, the number of CPUs (#CPUs), the number of CPU cores (#cores), and the CPU speed, all calculated from the Linux system file *"proc/cpuinfo"*.
Memory power(m_i) of a node i is calculated as:

$$m_i = \textit{mem\_free}[\%] \times \textit{total\_memory} \quad (3)$$

Total memory is calculated from linux command *"free -m"*. Finally, relative rank(R_i) is defined as:

$$R_i = \frac{\textit{cpu\_power}(c_i)}{max(\textit{cpu\_power})} + \frac{\frac{\textit{mem\_power}(m_i)}{max\ (\textit{mem\_power})}}{3.0} \quad (4)$$

Here max(cpupower) and max(mempower) refer to the maximum CPU power and maximum memory power among all the nodes in a user-defined cloud space. Once all the relative ranks are calculated, each PFAgent sorts them in a descending order. The top node in the sorted list is the highest 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.

III. PERFORMANCE EVALUATION

This section compares default versus our migration-based scheduling, and discusses about AgentTeamwork-Lite's scheduling costs.

A. Default versus Migration-based Scheduling

We have used Wave2DMPI and Mandelbrot as benchmark applications to study difference between AgentTeamwork-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.

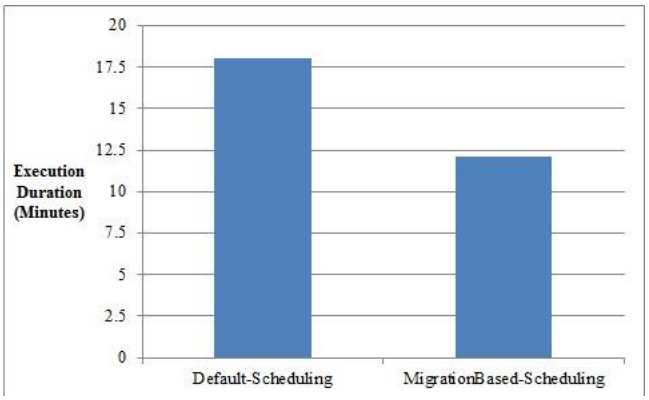

Fig. 2. Wave2DMPI default vs. migration-based scheduling execution.

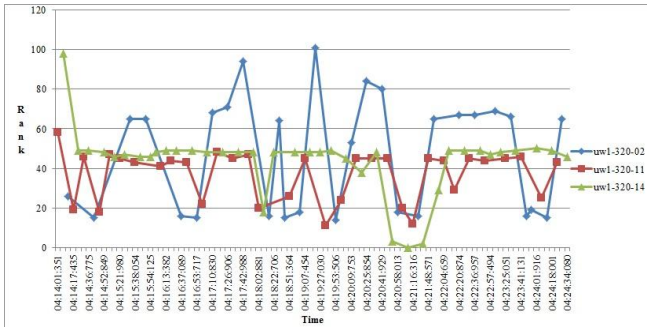

Fig. 3. Wave2DMPI execution nodes' rank (in default scheduling).

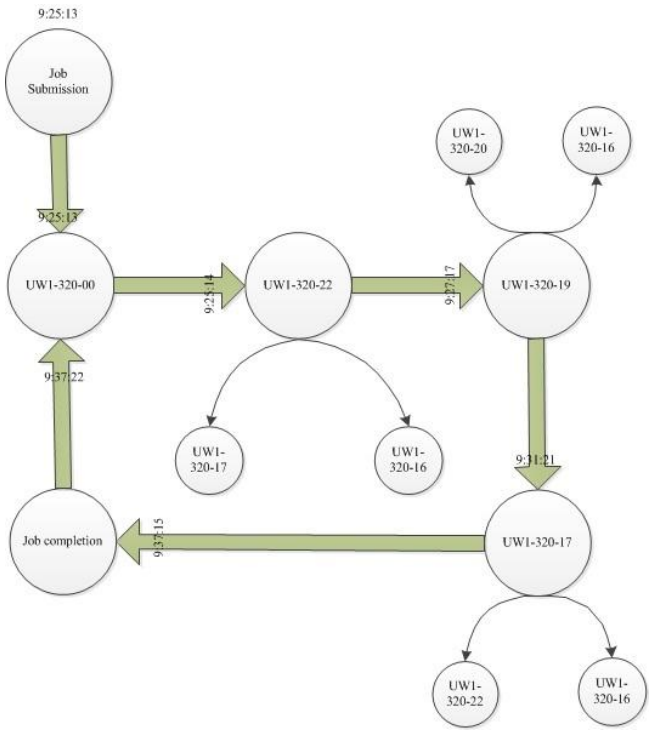

Fig. 4. Job migration history in Wave2D with migration-based scheduling.

Wave2DMPI is a two dimensional wave simulation program based on Schroedinger's equation that calculates the 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 2 compares the Wave2DMPI's execution time with the default scheduling and migration-based scheduling. This job was executed with three computing nodes, and the simulation size was set to 1000×1000 units. The figure indicates that migration-based scheduling completes the job execution approximately 40% faster than the default scheduling.

Figure 3 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 observed the longer execution time with the default scheduling.

Figure 4 recorded the history of job migration when we executed Wave2DMPI with AgentTeamwork-Lite's migration-based scheduling. It uses a combination of preemptive and non-preemptive job migration. Preemptive job migration means that a 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 was initially submitted to node uw1-320-00, which spawned commander and sentinel agents. Since the sentinel agent detected a better computing node than uw1-320-00, it migrated to a next node in the itinerary which was uw1-320-22 (in non-preemptive migration), created a resource file, (i.e., mpd.hosts), and started the execution. After a while, the sentinel agent detected that the current executing node's (i.e., uw1-320-22's) individual rank (r_i) dropped below 46%, thus suspended the job execution, and migrated to a next node in the itinerary which was uw1-320-19. Since this migration enabled the program to execute on better computing nodes, its execution time was faster than the default scheduling.

We have also conducted 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 given number after a predefined number of iterations, which requires computational intensive calculation. For example, a 600×800 pixel image can use up to maximum iterations of 500,000 for each of its pixel. This program is developed in Java.

Figure 5 compares Mandelbrot's execution time between the default scheduling and migration-based scheduling. This job was executed with a single computing node to generate a 1000×1000 pixel fractal image with the max iterations set to 200,000. The figure indicates that the migration-based scheduling completed the job execution approximately 15% faster than the default scheduling.

Figure 6 records the snapshots of the computing node's individual rank (r_i) that executed the Mandelbrot fractal with the default scheduling when executing the Mandelbrot fractal.

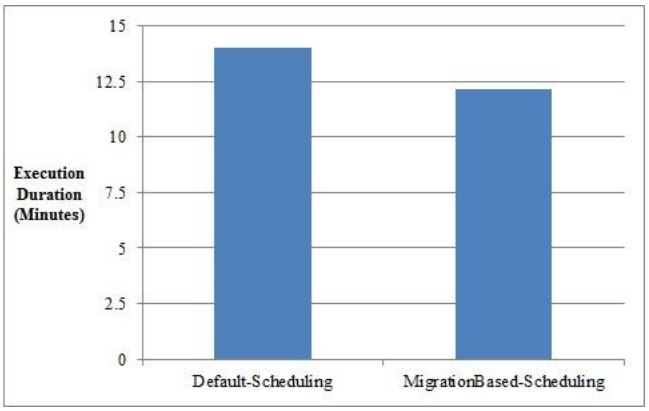

Fig. 5. Mandelbrot fractal execution with default vs. migration-based scheduling.

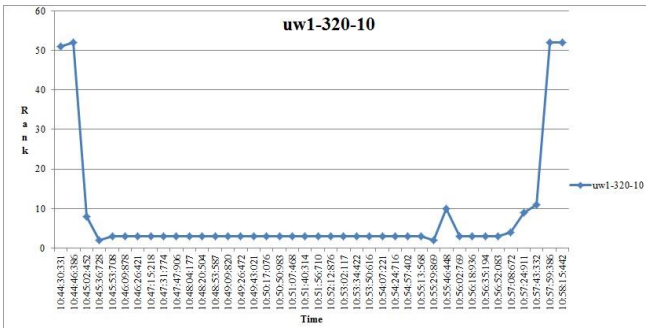

Fig. 6. Node rank transition in Mandelbrot fractal execution with default scheduling.

Before executing the job, the node's individual rank (r_i) was at 50, but during program execution, it dropped down below 10. At the end of the execution, the rank came back to 50 again. Most likely Mandelbrot consumed the system resources heavily that caused the individual rank to drop from 50 to 10.

Figure 7 recorded the history of job migration when we executed Mandelbrot with AgentTeamwork-Lite's migration-based scheduling. In the figure, a job was initially submitted to node UW1-320-00 that spawned a commander and sentinel agents. Since the sentinel agent detected that there were higher-ranked nodes than UW1-320-00, it migrated to one of these promising nodes in the itinerary which was UW1-320-20, and started the job execution there. After a while, the individual rank (r_i) of node UW1-320-20 dropped below 46%. Therefore, the sentinel agent suspended 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 executed job faster than the default scheduling.

B. 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 observed with AgentTeamwork-Lite is a few seconds more as compared to direct execution. Since the sentinel agent checks the job completion at a regular interval (i.e., every two seconds), the commander agent will always receive a delayed

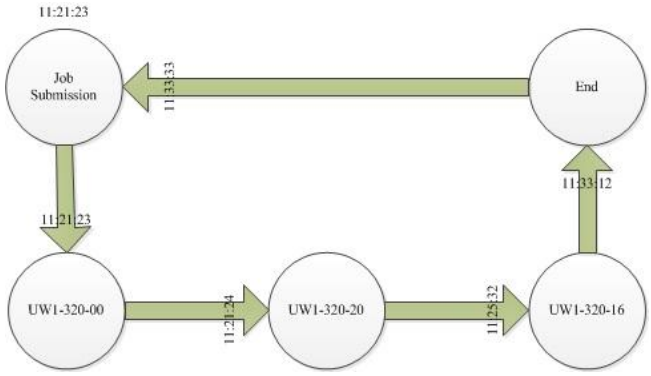

Fig. 7. Job migration history in Mandelbrot with migration-based scheduling.

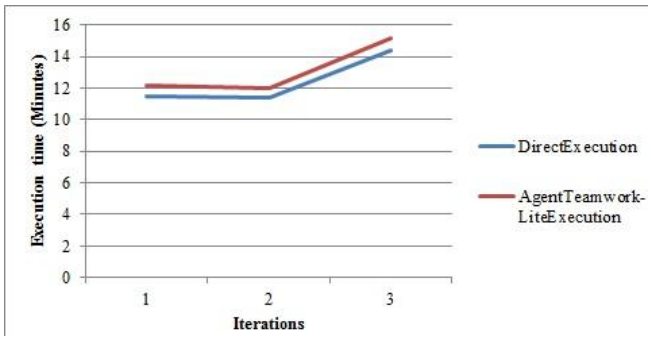

Fig. 8. Direct vs. AgentTeamwork-Lite execution.

job-completion message that can be up to the maximum of two seconds. Figure 8 compares the Wave2DMPI's execution time between the direct and AgentTeamwork-Lite's execution. The average time difference between direct and AgentTeamwork-Lite's execution is 24 seconds, which corresponds to only 3.17% of the direct execution time.

The job migration cost is the total time involved in suspending the job execution and resuming it at a different node. Since a job is periodically check-pointed at the user level, the job-capturing overhead depends on a user program. For migration, the sentinel agent simply exits from 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 at the user level. (Therefore, the data retrieval also depends on a user program.) Hence, the sentinel's job migration cost itself is in a negligible amount (i.e., several hundred milliseconds). Figure 9 shows a sentinel agent's job migration cost when running Wave2DMPI with AgentTeamwork-Lite.

IV. Related Work

AgentTeamwork-Lite is differentiated from the following systems in terms of migration-based scheduling, parallel application scheduling, and resource discovery.

GridLab Resource Management System (GRMS) is an open source dynamic grid (re-)scheduling system with job migration [12], based on Globus Toolkit 2.4 [13]. GRMS connects to the low-level Globus services that are deployed onto remote resources through Java and C APIs. With these

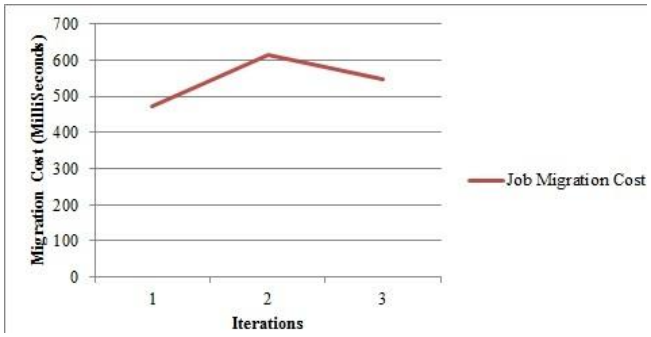

Fig. 9. Job migration cost of Wave2DMPI.

services, system designers can implement various rescheduling policy plug-ins. Both AgentTeamwork-Lite and GRMS use migration-based job scheduling. The difference is that GRMS initiates job migration when the system suffers from the lack of resources and thus reallocates a substantial amount of resources from the current to a pending job, whereas AgentTeamwork-Lite aims at improving the execution time of the current job. Another difference is that GRMS requires a user to specify resource requirements along with a job submission, whereas AgentTeamwork-Lite does not use user-specific requirements. In GRMS, the resource discovery module uses both central (GIIS) and local information services (GRIS), whereas in AgentTeamwork-Lite, resource discovery is achieved among PFAgents through their message broadcast.

Condor is a workload management system specialized for compute intensive jobs [14]. A condor-pool consists of (1) a master node that runs as a central manager and (2) a number of other machines that join the pool as participating resources. The central manager periodically receives status updates from the other machines, and performs match-making between a new job and the most suitable computing node. Condor moves a job execution when the current executing node does not meet user-specified resource requirements. A system-level checkpoint 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 Condor to intercept system calls and to capture on-going computation. 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 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 Condor, AgentTeamwork-Lite performs decentralized resource monitoring with distributed PFAgents. Since AgentTeamwork-Lite uses application-level multi-process check-pointing and dynamically changes a configuration file such as MPI's *mpd.host*, it supports job migration for parallel applications.

V. Conclusions

This paper presented AgentTeamwork-Lite's system design and its execution model. Our field-based job dispatch and migration relieves users from specifying computing resource requirements for their job execution. Our analysis has also demonstrated advantages of AgentTeamwork-Lite's migration-based job scheduling over default scheduling. At present we have completed the AgentTeamwork-Lite web portal where we can schedule computing jobs within the UW Bothell campus. Our future plan includes (1) automating computation check-pointing, (2) testing AgentTeamwork-Lite with cloud services such as Amazon EC2, and (3) supporting more parallel libraries such as GlobalArray [15] and our MASS library [9].

Acknowledgment

The authors would like to thank Mr. Scott Loomis for his development of AgentTeamwork's portal site to start and monitor job execution.

References

[1] Z. Zhang and X. Zhang, "Realization of open cloud computing federation based on mobile agent," *IEEE International Conference on Intelligent Computing and Intelligent Systems*, vol. 3, pp. 642–646, 2009.

[2] B. M. Chapman and F. Massaioli, "Openmp." *Parallel Computing*, vol. 31, no. 10-12, pp. 10–12, 2005.

[3] R. L. Graham, B. W. Barrett, G. M. Shipman, T. S. Woodall, G. Bosilca, and G. Editors, "Open mpi," *Parallel Processing Letters*, vol. 17, no. 01, pp. 79–88, 2007.

[4] OpenPBS, "http://www.mcs.anl.gov/research/projects/openpbs/."

[5] HTCondor, "http://research.cs.wisc.edu/htcondor."

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of the 6th Symposium on Operating System Design and Implementation - OSDI'4*. San Francisco, CA: Publisher, December 2004, pp. 137–150.

[7] M. Fukuda, K. Kashiwagi, and S. Kobayashi, "AgentTeamwork: Coordinating grid-computing jobs with mobile agents," *International Journal of Applied Intelligence*, vol. Vol.25, no. No.2, pp. 181–198, October 2006.

[8] M. Fukuda and D. Smith, "UWAgents: A mobile agent system optimized for grid computing," in *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*. Las Vegas, NV: CSREA, June 2006, pp. 107–113.

[9] T. Chuang, "Design and qualitative/quantitative analysis of multi-agent spatial simulation library," Master's thesis, Master of Science in Computing and Software Systems, University of Washington, 2012.

[10] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "mpijava: An object-oriented java interface to mpi," *Parallel and Distributed Computing - Lecture notes in computer science.*, vol. 1586, pp. 748–762, 1999.

[11] H. Prochazka, "The mandelbrot set." *Australian Mathematics Teacher*, vol. 67, no. 4, 2011.

[12] 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.

[13] H. Morohoshi and R. Huang, "A user-friendly platform for developing grid services over globus toolkit 3," in *Proc. of the 11th IEEE International Conference on Parallel and Distributed Systems*, vol. 1, July 2005, pp. 668–674.

[14] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[15] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. Vol.20, no. No.2, pp. 203–231, 2006.