Evaluation of Agent Teamwork
A High Performance Distributed Computing Middleware System

Final Report

Solomon Lane
Agent Teamwork Research Assistant
October 2006 – March 2007

Introduction

I spent the last two quarters working as a research assistant for Professor Munehiro Fukuda as part of his Agent Teamwork project. This final report documents the work I did, my experiences, and some of lessons I learned in the process.

Agent Teamwork is a High Performance Distributed Computing (HPDC) middleware system under ongoing development. Performance optimization is driving much of this work. To support this effort my goal as a research assistant was to evaluate Agent Teamwork's performance against a contemporary alternative.

The Agent Teamwork middleware system provides two general functions, each of which introduce overhead into the system. Firstly it provides a mechanism to dispatch jobs to the distributed computing resources and terminate them when they are finished. Secondly it provides a framework to programmatically coordinate theses resources.

There is no single contemporary HPDC middleware that provides an exact match with the features of Agent Teamwork. Instead there are several popular middleware components that provide different sets of features. Therefore I had to evaluate Agent Teamwork's two main functions against different contemporary products.

In order to evaluate the Job Dispatch and Termination function, I built a reference platform that provided the same functions. This platform however did not support a similar enough framework; therefore I had to evaluate Agent Teamwork's framework performance against a different product which did not provide a comparable Job Dispatch and Termination function.

I will begin by providing a brief overview of the terminology used within this paper and their definitions. I will then review the development of the reference platform development and the Job Dispatching and Termination performance evaluation, including some of the challenges I came across. This will be followed by a description of the framework level performance evaluation and final conclusion.

Terminology

The following outlines the key concepts and terminology discussed within this paper.

Grid vs. Cluster

A computing grid is commonly distinguished from a computing cluster by the geographic distance between members. A cluster would be a group of computers in the same room or building and connected to the same physical network, while the members of grid could be located anywhere and many members are likely connected over a wide area network.

The distinction can also be compared with their relationships between the individual nodes. For the purpose of this paper, a grid refers to a group of computers that are

loosely connected in terms of awareness of peers. Computers in the same grid may all run software that accepts individual job submissions, but there is nothing inherent in the grid to coordinate related jobs or even recognize jobs as being related.

A cluster on the other hand will have a head node that is aware of the clusters members and their state. Whereas a grid will provide infrastructure services such as authentication and authorization for job submission. A cluster can dynamic schedule and allocate resources based on their current state.

Platform

For the purpose of this paper, a HPDC platform has software that provides infrastructure and scheduling services. Infrastructure services include authentication and authorization, job submission, and file transfer for job deployment. Scheduling services include dynamic resource identification and allocation, scheduling policies, and coordinating job execution.

Framework

Framework has a related set of software libraries that are used to implement a programming model. The Single Program Multiple Data (SPMD) programming model is commonly used to achieve data level parallelism in HPDC. MPIJava is a Java implementation of the Message Passing Interface standard. MPIJava provides a framework for programming in the SPMD model.

Middleware

A system that provides both a platform and a framework for developing and running HPDC applications.

Job dispatching and termination function

This section will review the performance evaluation of Agent Teamwork's job dispatching and termination function, including a discussion of the development of the reference platform that it was evaluated against.

Because the Globus Toolkit (GTK) is, according to the New York Times, considered the de facto standard in grid computing, I was tasked with evaluating Agent Teamwork's job dispatching and termination performance against a GTK based reference platform. However the GTK is only a toolkit and it does not provide a complete middleware system. Therefore in order to build a reference platform with services comparable to Agent Teamwork I had to integrate the GTK with the OpenPBS scheduler and the MPICH-G2 MPI framework. This turned out to be a significant challenge.

Reference Platform Overview

This section describes the reference platform that I built on the lab computers that make up our test environment.

Hardware Details

The test environment consists of 66 computers, 32 of these machines come from a shared student Linux lab and 34 dedicated machines come from the distributed systems lab. These machines are divided into two clusters, each with one head node and 32 members. The individual cluster nodes are interconnected at 1 or 2GBps within a cluster and the head nodes are connected to the campus backbone at 100MBps. The specifics of each cluster are summarized below in Table 1. I built the reference platform and evaluated the performance of Agent Teamwork using the same 66 computers.

Medusa Cluster	**Phoebe Cluster**
a 32-node cluster for research use	a 32-node cluster for instructional use
Head Node: **specification outbound** 1.8GHz Xeon x2, 512MB memory, and 70GB HD 100Mbps	**Head node:** **specification outbound** 1.5GHz Xeon, 256MB memory, and 40GB HD 100Mbps
Computing nodes: **#nodes specification inbound** 24 3.2GHz Xeon, 512MB memory, and 36GB HD 1Gbps 8 2.8GHz Xeon, 512MB memory, and 60GB HD 2Gbps	**Computing nodes:** **#nodes specification inbound** 16 1.5GHz Xeon, 512MB memory, and 30GB HD 100Mbps 16 1.5GHz Xeon, 512MB memory, and 30GB HD 1Gbps

Table 1: Cluster Comparison

Globus Grid

The GTK primarily provides grid level infrastructure services, such as authentication and authorization for job submission and dispatch. My first step to build the reference platform was to install the GTK on all 66 machines. This involved, building GTK from source, installing and configuring it. I also had to configure a local certificate authority in order to support the grid security services. The GTK provides no hierarchy or implicit cooperation between the machines. From any machine with the proper credentials, I could submit random or related jobs to any of the machines in the grid. There is nothing inherent in the grid to coordinate related jobs or even recognize jobs as being related.

When a job submission is accepted a Job Manager is started to handle the job. Fork is the default Job Manager provided by the GTK and it simply executes the job on the local machine. After successfully installing, configuring and debugging GTK, the next step in building the reference platform was to integrate the Open PBS schedule.

OpenPBS Clusters

PBS stands for Portable Batch System. I used the free version, OpenPBS. It can be used to manage computing resources by defining resource types, such as time sharing or cluster nodes, and policy based work queues for allocating those resources. A number of parameters can be used to determine a node's suitability for allocation to a particular job.

OpenPBS consists of three daemons which I built from source. They are pbs_server, pbs_sched and pbs_mom. The pbs_mom daemon was installed on 64 computers that would act as cluster nodes. I divided these machines into two clusters by configuring their respective pbs_mom's to be managed by one of the cluster head nodes. I installed and configured pbs_server and pbs_sched on each of remaining two machines, Medusa and Phoebe, so they could act as cluster heads, each managing 32 nodes. As Figure 1 below indicates, the head nodes are running pbs_server and pbs_sched daemons while the cluster nodes run the pbs_mom daemon.

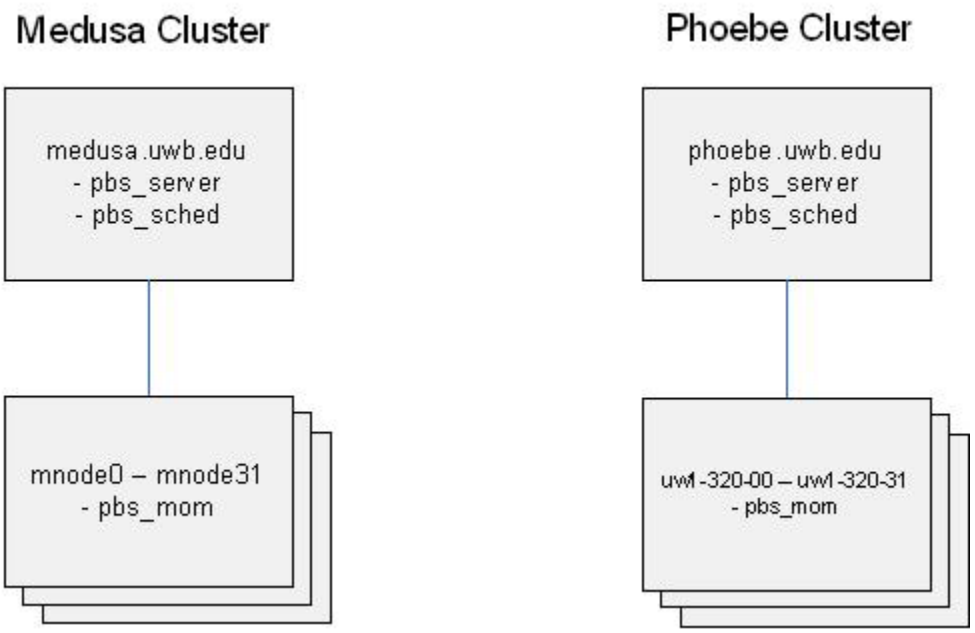


Figure 1 – PBS Clusters

Note that while these machines are all members of the same GTK grid, the PBS cluster configuration is unaware of the grid and has no dependency on it.

Grid/Cluster Integration

Initial OpenPBS integration is simply a matter of configuring the GTK to use the PBS job scheduler and then specifying the cluster head node when submitting a job. Running and coordinating the same job across multiple clusters is bit more complex.

First of all the coordination must be programmatically orchestrated using the GTK's DUROC library. Fortunately this can be handled by integrating the MPICH-G2 framework into platform. Secondly multi-cluster jobs must be described in the GTK Resource Specification Language, which I had to learn how to use.

MPICH-G2

MPICH-G2 is a globus/DUROC aware framework that implements the MPI standard. I built it from source which required configuring it to integrate with the local GTK installation. After configuring the installation I learned how to write c++ code using the framework and to build the necessary Makefiles to compile it. At this point the reference platform was complete.

Reference Platform Development Challenges

I had to overcome several challenges in order to complete the reference platform. This section highlights some of the major issues.

GTK Authentication

The cluster machines are under the administrative control of the university's systems administrators which introduced logistical challenges when administrative access was needed (e.g. access to the root account). To minimize this dependency I decide to run the globus software with a regular user account instead of running it under the root account. Unfortunately this approach resulted in a problem that was difficult to debug.

Specifically, the globus-gatekeeper must run with a host certificate that cannot be read by the user submitting a job. Because I ran both the gatekeeper and submitted jobs as the globus user, the host certificate remained readable by both the submitter and the gatekeeper. The actual problem was extremely subtle. The following steps take place in a Globus job submission:

1. The job submission connects to the gatekeeper and they mutually authenticate each other using the following certificates:
    a. Gatekeeper presents it's host certificate
    b. Submitter presents it's user certificate
2. After mutual authentication, the gatekeeper checks authorization. If authorized, the gatekeeper receives a delegated user proxy certificate from the submitter to be used for callbacks to the submitter.
3. The gatekeeper starts a job manager with the submitter's uid and provides the job manager with the location of the delegated user proxy.
4. The job manager establishes a call back to the submitter and must present the delegated user proxy certificate for authentication.

However, in step 4 the job manager actually performs a credential search using the standard globus credential search order which searches for host certificates before delegated proxy certificates. Because the host certificate was readable by globus, the job manager tried to use the host certificates for the call back, which was rejected.

The solution was to make the host certificate read only by root and run the gatekeeper as root which has authority to start the job manager with the globus user uid. Due to the subtle nature of the problem and logistical overhead of coordinating with the systems administrators it took several weeks and 10's of hours of straces, tcpdumps, gdb sessions, and message board postings to figure out the problem.

Host Configuration & Cryptic Error Messages

There was wide variance in the system and network level configurations both on the individual hosts and networking equipment and dns servers. For example:

1. Some hosts had A-records in dns while others only existed in hosts files.
2. Some the hosts files were inconsistent
3. Some of the hosts in dns some had ptr records while others did not
4. Many hosts had different port acls between different groups of machines
5. Many hosts had different local port acls, for example many hosts could connect to localhost:port but not "the local hostname":port
6. General variance in file system permissions and sudoers file propragation

I only discovered the many inconsistencies because the GTK and OpenPBS had dependencies on these configurations. Futhermore these dependencies manifested themselves in cryptic error messages that often required extensive strace analysis to uncover. For example the following error occurs due to the local port acl describe in example 5 above:
: globus_init: failed
globus_module_activate(GLOBUS_DUROC_RUNTIME_MODULE)

As I debugged these individual issues, I would then coordinate with the systems administrators to make the necessary configuration changes.

Performance Evaluation

This section describes the performance evaluation technique and discusses the results.

Methodology

My objective was to evaluate the performance of the job dispatch and termination overhead of the platform, not the job execution performance. Therefore I could not measure performance from within a test program. Instead I needed to measure how long it took a job submission to be deployed, executed and cleaned up. By using a test program that would have a negligible impact on overall runtimes, I was able to evaluate dispatch and termination performance by starting a timer when the job was submitted and stopping when the results were returned.

The professor had already created and used such a program to measure Agent Teamwork's performance in this area. However this program was not compatible with MPICH-G2 framework which is c++ based so I ported the test program to the MPICH-G2 framework. The program designates one host as a master and the rest as slaves and sends a single byte of data between the master and each slave. In each test program the algorithms are identical.

The test program was run with 2-64 nodes across the two clusters in a depth-first node distribution series and a breadth-first node distribution series. The runs in the depth-first series used all 32 nodes in the first cluster before using any additional nodes from the second cluster. The runs in the breadth-first series always used the same number of nodes from each cluster.

After running several evaluations I observed that roughly 1 of every 5 runs would vary wildly from the rest of the times. Therefore I decided to run each series 5 times, throw out the outlier and average the remaining times. These averages were compared to the professors published performance data for Agent Teamwork.

Challenges

When I compiled the first evaluation we noticed that globus (GTK) performance seemed to be the same at several node counts and would then jump by 10 seconds, and repeat the

pattern. After tracing the jobs in more detail I discovered that this was due to the way globus monitored the OpenPBS job managers.

In order to run a job in the GTK/OpenPBS environment, the globusrun command is used to submit the job to the globus gatekeeper, which coordinates dispatch of the job to the OpenPBS clusters. Once the job is dispatched to the OpenPBS clusters, the GTK job manager polls the OpenPBS clusters every ten seconds to check on the job status. Once the job is complete, globusrun exits.

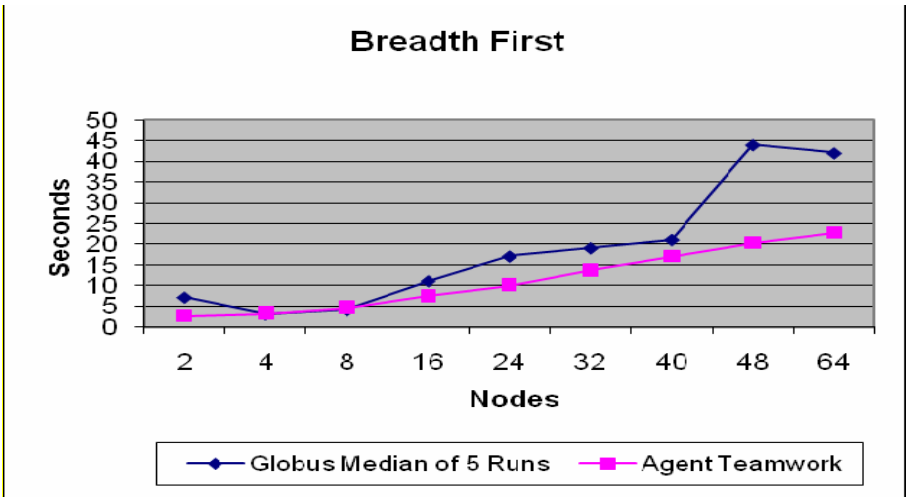
By simply timing how long it took for a globusrun submission to exit my results were off by almost ten seconds. Fortunately the OpenPBS logs record the exact time that the job completes. Therefore I changed the measurement to start the timer when globusrun is launched, and use the end time from the OpenPBS logs. This allowed me to accurately time the job completion within one second, which is the resolution of the OpenPBS logs.

Evaluation Results

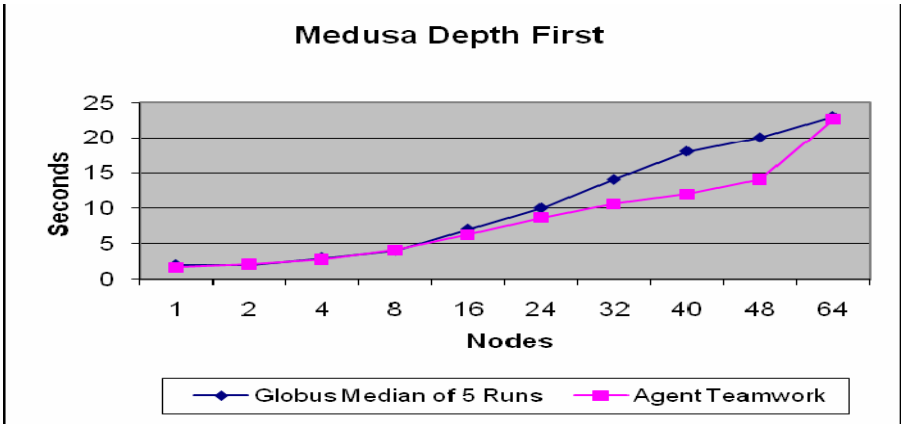
I made two interesting observations with regards to the reference platform. First of all, when running the same test, depth first, on the phoebe cluster multiple times, rouglhy one out of five of the results would vary wildly while the other three or four tended to cluster towards a same result. For example in five runs with two nodes on the phoebe cluster, I measured two seconds twice, one second twice, and seven seconds once. There was also some variation in multiple depth first runs on the Medusa cluster, but it was far lest pronounced.

The second thing I noticed is that the variations were far wider when running a breadth first series on the reference platform. This leads me to suspect that the reference platform is far more sensitive to something network related, though I was unable to determine the root cause. Since the majority of values tended to cluster, I decided to run each test five times and take the median value. The results were plotted against the professors published values for Agent Teamwork performance.

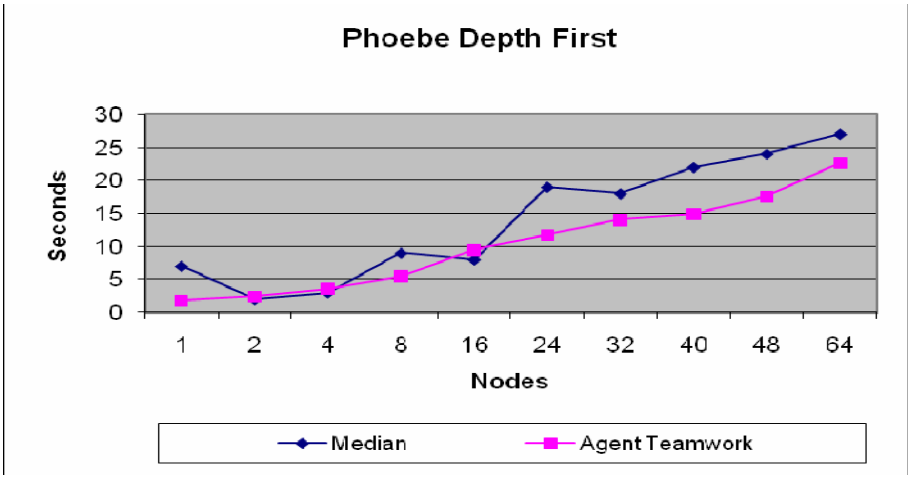
The following three graphs compare the results of Agent Teamwork's performance against the median of five runs using Breadth First, Medusa Depth First and Phoebe Depth First. The raw data can be found in Appendix B.



Graph 1: Breadth First evaluation.



Graph 2: Medusa Depth First Evaluation



Graph 3: Phoebe Cluster Depth First Evaluation

As the results show, Agent Teamwork's job dispatch and termination performance was comparable with the reference platform in a depth first configurations. And agent teamwork tended to outperform the reference platform with a large number of nodes in a breadth first configuration.

Framework/Communication Overhead

Agent Teamwork is completely implemented in java and it provides an MPI implementation called MPJ. MPI is supported on the reference platform by the c++ MPICH-G2 implementation. Because we did not want the framework performance results skewed by performance differences between c++ and java, I was asked to compare Agent Teamwork's framework level execution performance against MPIJava.

MPIJava is a popular Java implementation of the MPI. MPIJava does not provide platform services for dynamic node allocation, instead MPIJava requires the user to provide a static list of computers to use at runtime.

These MPI frameworks primarily provide standard communication mechanisms such as initialization, barrier, broadcast, scatter, gather, and point-to-point messages. To evaluate the framework execution performance I was asked to write three benchmark programs that have framework communication intensive algorithms.

The Benchmark Programs

The three benchmark programs are MD, a molecular dynamics simulation; Wave2D, a wave dissemination simulation; and Mandelbrot, a Mandelbrot generator. With the exception of Mandelbrot I had to code each program twice, once for MPIJava and once for Agent Teamwork MPJ. Mandelbrot had already been written in MPIJava by Josh Phillips so I only had to port it to Agent Teamwork MPJ.

Benchmark Development Challenges

MPIJava is mature framework that was relatively straightforward to code in. However it is important to remember that MPIJava does not provide any of the platform level services provided by Agent Teamwork. Because of Agent Teamwork's advanced features and its earlier stage of development it presents more challenges to the programmer.

Agent Teamwork Programming Model

The main difficulty with the Agent Teamwork programming model is due to a side affect of its job snapshot feature. Agent Teamwork takes regular runtime snapshots of a program and is capable of migrating a running job from one node to another for load balancing and dynamic failure recovery. Professor Fukuda summarizes the issue as follows:

The problem is that Java does not serialize an application's program counter and a stack. To handle this problem, we decided to partition a user program into a collection of func_n methods. In this scheme, the user program wrapper schedules the invocation of these functions and takse a snapshot at the end of each function call.

This requires each function to return an int representing the name of the next function to call. Because functions are called by returning an int, you cannot pass parameters to

functions and you cannot directly return from a function call. This requires setting globals for any parameters that need to be passed and results in the following general programming model:

```
func_0( ) {
  statement_1;
  statement_2;
  statement_3;
  return 1;
}
func_1( ) {
  statement_4;
  statement_5;
  statement_6;
  return 2;
}
```

Furthermore snapshots need to be taken frequently, especially in the case of large ‘for loops’ or the snapshot images can become too large. This requires converting a ‘for loop’ into a series of functions such as:

```
public int func_2( ) {
    if ( i < max ) {
        // for loop;
        return 3;
    }
    else {
        // move on
        return 4;
    }
}

public int func_3( ) {
    …..
    // leave function for snapshot every 50 iterations
        if (i % 50 == 0) {
            i++;
            return 2;
        }
        i++;
        return 2;
}
```

The other main challenge had to do with the maturity of code base.

Code Maturity

Because Agent Teamwork is still under ongoing development, programming for performance evaluation has also provided useful real world testing of the code. Through the process of coding and debugging these test applications I uncovered several bugs and race conditions in the framework.

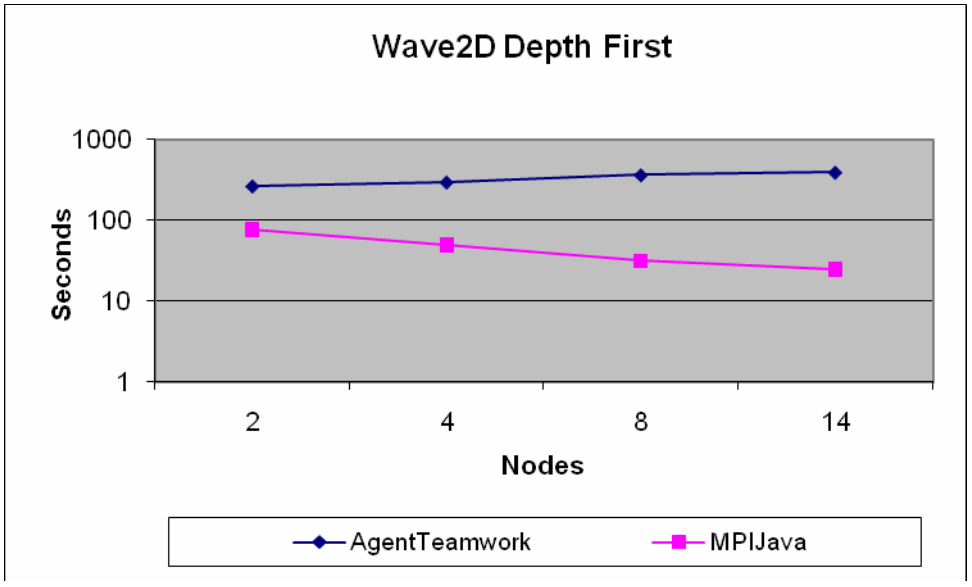
Evaluation Methodology

As of this writing Agent Teamwork versions of MD and Mandelbrot cannot be run across the clusters due to outstanding bugs in the framework. The Agent Teamwork version of Wave2D does not rely on the same framework elements with the outstanding issues and can therefore run across the clusters. However other framework issues still prevent Wave2D from completing successfully when run on more than a handful of nodes with anything more than a trivial workload. Therefore I was unable to perform a complete evaluation with any of the benchmark programs. However I have included some partial results with Wave2D.

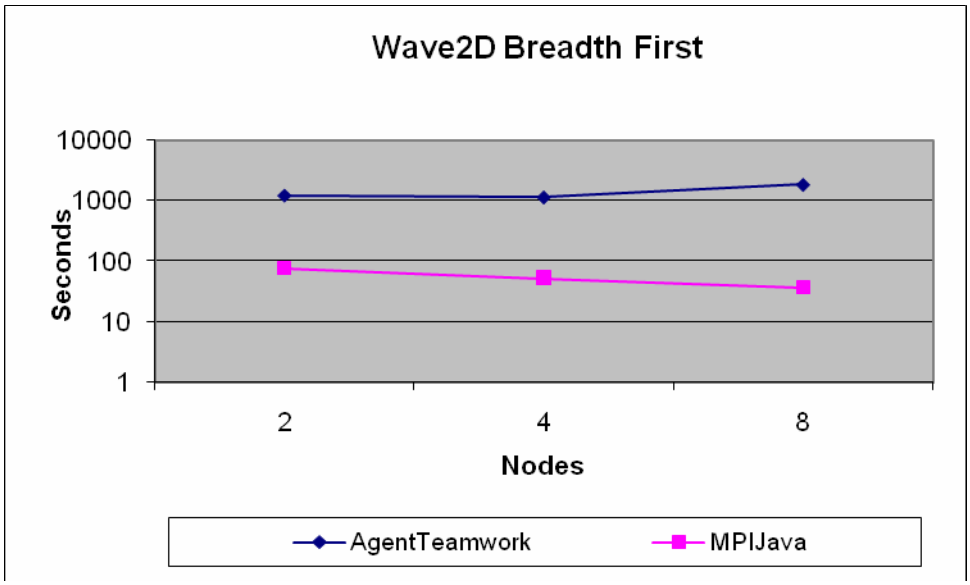
These partial results were obtained by running the Agent Teamwork and MPIJava versions in a depth first series on the Medusa cluster and a limited breadth first series with both clusters.

Evaluation Results

As the following partial results show in the two graphs below, the Agent Teamwork version is two orders of magnitude slower than MPIJava. In the course of testing I observed that the Agent Teamwork seemed to get slower and slower with each iteration of the main algortihm. I suspect that this is related to the snapshots which are taken at each iteration. At this point however framework debugging is ongoing.



Graph 4: Wave2D Depth First Evaluation



Graph 5: Wave2D Breadth First Evaluation

Future Work

Framework debugging must certainly continue. The professor also plans to develop a pre-processor to convert conventionally programmed code into the snapshot-able func_n model. In my opinion this pre-processor will be critical to allow Agent Teamwork to move from running in the lab to becoming a publicly available middleware.

Conclusion

My goal was to evaluate Agent Teamwork's performance against a contemporary alternative. To accomplish this goal I created an evaluation of the system's two general functions. For the job dispatching and termination function, I created a reference platform by installing and integrating globus, open PBS and MPICH-G2. For the framework function evaluation, I wrote three benchmark programs that have framework communication intensive algorithms in order to compare Agent Teamwork's framework level execution performance against MPIJava.

My work showed that Agent Teamwork provides solid job dispatch and termination performance, while there is still work to be done at the framework layer. While I encountered some challenges, especially in the development of the reference platform, I feel that I have gained significant experience with globus, openPBS and the mpi during my time as a research assistant. This project provided me with extensive debugging experience with advanced tools such as tcpdump, strace, and gdb. I also gained experience with performance analysis and writing parallel programs. Working with both the current popular technologies and the cutting edge Agent Teamwork has provided me with new insights and understanding of HPDC.

Appendix A – Dispatch & Termination Evaluation Details

To measure the Job Dispatch & Termination performance on the reference platform I wrote a shell script that perfomed the following functions:

- Generated an RSL file for the particular job requirements (breadth or depth first and number of nodes)
- Recorded the current time and submitted the job.
- Obtained the pbs jobid from the globus logs and located the job termination time from the pbs logs

The full text of the shell script, test.sh, is located in Appendix F. The raw data in Appendix B was generated by running the script with corresponding options for distribution type and node count.

Appendix B – Deployment & Termination Evaluation Raw Data

Breadth First

Nodes	Globus/PBS Execution Time	Globus/PBS Median	Agent Teamwork Execution Time
2	2		
2	12		
2	3		
2	7		
2	7	7	2.663
4	3		
4	3		
4	8		
4	3		
4	3	3	3.245
8	4		
8	4		
8	4		
8	4		
8	4	4	4.678
16	98		
16	11		
16	11		
16	16		
16	11	11	7.385
24	99		
24	13		
24	17		
24	13		
24	17	17	10.034
32	107		
32	10		
32	25		
32	11		
32	19	19	13.621
40	104		
40	18		
40	13		
40	21		
40	21	21	16.93
48	111		
48	26		
48	31		
48	45		
48	44	44	20.206
64	34		
64	44		
64	73		
64	31		
64	42	42	22.603

Medusa Depth First

Nodes	Globus/PBS Execution Time	Globus/PBS Median	Agent Teamwork Execution Time
1	2		
1	2		
1	2		
1	2		
1	2	2	1.678
2	2		
2	2		
2	2		
2	2		
2	2	2	2.104
4	3		
4	3		
4	2		
4	3		
4	2	3	2.778
8	4		
8	4		
8	5		
8	4		
8	5	4	4.09
16	8		
16	7		
16	8		
16	7		
16	7	7	6.276
24	11		
24	10		
24	11		
24	10		
24	10	10	8.652
32	14		
32	14		
32	14		
32	14		
32	14	14	10.591
40	16		
40	18		
40	16		
40	23		
40	23	18	11.952
48	20		
48	16		
48	24		
48	22		
48	18	20	14.039
64	23		
64	22		
64	26		
64	27		
64	17	23	22.608

Phoebe Depth First

Nodes	Globus/PBS Execution Time	Median	Agent Teamwork
1	8		
1	2		
1	12		
1	2		
1	7	7	1.867
2	2		
2	7		
2	2		
2	1		
2	1	2	2.492
4	4		
4	8		
4	3		
4	3		
4	3	3	3.666
8	10		
8	9		
8	5		
8	10		
8	5	9	5.554
16	13		
16	8		
16	8		
16	12		
16	8	8	9.495
24	19		
24	11		
24	19		
24	11		
24	26	19	11.763
32	14		
32	18		
32	24		
32	15		
32	27	18	14.066
40	17		
40	22		
40	31		
40	21		
40	26	22	14.938
48	42		
48	17		
48	24		
48	22		
48	27	24	17.591
64	27		
64	32		
64	23		
64	22		
64	27	27	22.603

Appendix C – Framework Evaluation Details

In order to measure agent teamworks framework performance I wrote a shell script to run the Wave2D program across the cluster nodes in a depth or breadth first distribution. The text of the shell script, runWave2D.sh is located in Appendix F. The raw data for Agent Teamwork in Appendix D was generated by running the script with the following program options for the corresponding distribution type and node count:

```
runWave2D.sh -t <type> -n <nodes> -r 1000 -N 500 -xmin 200 -ymin 200 -xmax 300 –ymax
300
```

In order to measure Wave2D using MPIJava, I ran it with the prunjava script which presumably came with the mpijava distribution. The raw data for MPIJava in Appendix D was generated by running the script with the following program options for the corresponding node count:

```
prunjava <nodes> Wave2D -n 1000 -xmin 200 -xmax 300 -ymin 200 -ymax 300 -r 1000
```

The distribution was controlled by placement of hostnames in the machines file.

Appendix D – Framework Evaluation Raw Data

Wave2D Breadth First		
Node	AgentTeamwork	MPIJava
2	1210.748	75.602
4	1126.522	52.691
8	1839.943	37.087

Wave2D Depth First		
Node	AgentTeamwork	MPIJava
2	263.484	75.969
4	294.93	49.252
8	362.238	31.385
14	390.763	24.521

Appendix E – test.sh shell script

```
#!/bin/bash

DIR="/home/globus/src/eval1"
EXE="/home/globus/src/eval1/InterCluster"
ARGS='"-m" "1"'

CLUSTERS=$1
EXPECTED_ARGS=2

usage () {
    cat <<USAGE

Description:
This script generates a temporary rsl file and runs an mpi job
on one or more pbs clusters accesible to a contact running the
Globus Gatekeeper with a properly configured pbs JobManager.

The script times the total job duration from submission to
completion and returns the time in seconds and the total number
nodes involved to standard out.  All other
output is sent to standard error.

Usage:
$0 <cluster count> <contact1>,<count1> [... <contactN>,<countN> ]

Example:
$0 2 phoebe:2119,32 medusa:2119,32

Other important settings are controlled by variables located
at the top of this script.  The current values are:

DIR="/home/globus/src/eval1"
EXE="/home/globus/src/eval1/InterCluster"
ARGS='"-m" "1"'

USAGE
exit 1
}

write_rsl_block () {
cat <<JOB_RSL_BLOCK >> $$.rsl
( &(resourceManagerContact="$1")
   (count=$2)
   (label="subjob $3")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX $3)
                (LD_LIBRARY_PATH /home/globus/globus/lib/))
   (directory="$DIR")
   (executable="$EXE")
   (arguments= $ARGS)
   (stdout=/home/globus/std.out$3)
   (stderr=/home/globus/std.err$3)
)
JOB_RSL_BLOCK
}

if [ ! -n "$1" ]; then
    usage
fi

if [ "$1" -gt 0 ];then
    EXPECTED_ARGS=$(($CLUSTERS+1))
    if [ $# -ne $EXPECTED_ARGS ]; then
        usage
    fi
else
```

```
    usage
fi

# start the job rsl file
echo '+' > $$.rsl

TOTAL_COUNT=0
SUBJOB=0
# shift off the cluster count
shift
for arg in "$@"; do
    CLUSTER=`echo $arg |awk -F ',' '{print $1}'`
    PCOUNT=`echo $arg |awk -F ',' '{print $2}'`
    TOTAL_COUNT=$(($TOTAL_COUNT+PCOUNT))
    write_rsl_block $CLUSTER $PCOUNT $SUBJOB
    SUBJOB=$(($SUBJOB+1))
done

START=`date +%D" "%T`
globusrun -f $$.rsl 1>&2

#----------------------
# Mad log parsing
#----------------------
# In a multicluster job, we want to parse the PBS logs for the cluster with
# rank0.  That way we get the full run time because rank0 exits after recieving
# data from all other ranks.  Rank0 is always in the first cluster in the rsl
#
# Gram should be configured on the first cluster to not delete the log so it
# can be parsed to obtain the PBS jobid.
# globus/etc/globus-job-manager.conf:         -save-logfile always
#                                             -save-logfile on_error
#
# Becuase we use nfs, the gram logs for both clusters will exist.  Wait until
# only one log remains before parsing
sleep 15
logs=`ls |grep gram_job_mgr_|wc -l`
while [ $logs -eq 2 ]; do
    logs=`ls |grep gram_job_mgr_|wc -l`
done
GRAM_LOG=`ls gram_job_mgr_*`
PBS_SUB_MARKER="JM_SCRIPT: job submission successful, setting state to PENDING"
PBS_SUB_TIME=`grep "$PBS_SUB_MARKER"  $GRAM_LOG | awk '{print $4}'`

JOBID_MARKER="GRAM_SCRIPT_JOB_ID ="
JOBID=`grep "$JOBID_MARKER" $GRAM_LOG  |awk '{print $9}'`

PBS_LOG_DATE=`date +%Y%m%d`
PBS_NQ_MARKER="PBS_Server;Job;$JOBID;enqueuing into batch"
PBS_START=`grep "$PBS_NQ_MARKER" /usr/local/openpbs/spool/server_logs/$PBS_LOG_DATE |awk -F ';'
'{print $1}'`
PBS_DQ_MARKER="PBS_Server;Job;$JOBID;dequeuing from batch"
PBS_STOP=`grep "$PBS_DQ_MARKER" /usr/local/openpbs/spool/server_logs/$PBS_LOG_DATE |awk -F ';'
'{print $1}'`

rm gram_job_mgr_*
#------------------

START_T=`date -d "$START" +%s`
PBS_SUB_TIME_T=`date -d "$PBS_SUB_TIME"  +%s`
PBS_START_T=`date -d "$PBS_START"  +%s`
PBS_STOP_T=`date -d "$PBS_STOP"  +%s`

TIME=$(($PBS_STOP_T-$START_T))
echo -e "$TOTAL_COUNT \t$TIME \tSTART:$START \tPBS_SUB:$PBS_SUB_TIME \tPBS_START:$PBS_START
\tPBS_STOP:$PBS_STOP"
rm $$.rsl
```

Appendix F – runWave2D.sh shell script

```
#!/bin/bash

# breadth first
# while i <= hostcount
# build hoststring, mnode2_
# build hoststring, uw1-320-00_uw1-320-01

. /etc/rc.d/init.d/functions

# globals
CL_medusa="CL_medusa"
CL_priam="CL_priam"
COMMANDER="localhost"
SENTINEL="perseus"
BOOKKEEPER="phoebe"

# MD default params
runCycles="1000"    # the of computation cycles to run for.  0 == forever
N="100"             # simulation_size^-2
xmin="40"
xmax="60"
ymin="40"
ymax="60"

#-----------------------
# FUNCTIONS
#-----------------------
function usage {
    cat<<HERE

usage: $0 -t [b|d|u] -n #nodes Optional_UPargs
 ex: $0 b 10

 -t   type of distribution:
      b(breath first) uses medusa and uw1-320 equally
      d(depth first) uses up medusa first thereafter uw1-320
      u(depth first) uses up uw1-320 first thereafter medusa

 -n   the number of nodes to be used for the execution

 Optional_UPargs (user program arguments)
 -r     the of computation cycles to run for.  0 == forever
 -N     the simulation size: N*N. default=100*100
 -xmin  cube size limit
 -xmax  cube size limit
 -ymin  cube size limit
 -ymax  cube size limit

Where cube size defaults to xmin=ymin=40, xmax=ymax=60

HERE
    exit
}

function runningUWPlace {
    status=`ssh -o BatchMode=yes -o StrictHostKeyChecking=no $1 'source
/etc/rc.d/init.d/functions;status runUWPlace.sh > /dev/null 2>&1;echo $?'`
    if [ $? -ne 0 ]; then
        return 1
    fi
    if [ $status -ne 0 ]; then
        ssh -o BatchMode=yes -o StrictHostKeyChecking=no $1 'cd ~solomonl/MA/agents ;
~solomonl/MA/agents/runUWPlace.sh > ~solomonl/`hostname`-UWPlace.log 2>&1 &'
    else
        ssh -o BatchMode=yes -o StrictHostKeyChecking=no $1 'killall -g runUWPlace.sh'
```

```
        sleep 8
        ssh -o BatchMode=yes -o StrictHostKeyChecking=no $1 'cd ~solomonl/MA/agents ;
~solomonl/MA/agents/runUWPlace.sh > ~solomonl/`hostname`-UWPlace.log 2>&1 &'
    fi
    status=`ssh -o BatchMode=yes -o StrictHostKeyChecking=no $1 'source
/etc/rc.d/init.d/functions;status runUWPlace.sh > /dev/null 2>&1;echo $?'`
    return $status
}

function medusaCluster {
    echo "Initializing Medusa Cluster"
    currentNode=$1
    nodeCount=$2
    i=0
    while [ $currentNode -lt $nodeCount ];do
        if [ $i -gt 31 ]; then
            echo "Failed to run UWPlace on enough nodes in the medusa cluster"
            exit 1
        fi

        runningUWPlace mnode$i
        if [ $? -eq 0 ]; then
            CL_medusa=$CL_medusa"_mnode$i"
            currentNode=$(($currentNode+1))
        fi
        i=$(($i+1))
    done
}

function priamCluster {
    echo "Initializing Priam Cluster"
    currentNode=$1
    nodeCount=$2
    i=0

    # init the cluster head
    runningUWPlace priam

    while [ $currentNode -lt $nodeCount ];do
        if [ $i -lt 10 ];then
            node="uw1-320-0$i"
        else
            node="uw1-320-$i"
        fi

        runningUWPlace $node
        if [ $? -eq 0 ]; then
            CL_priam=$CL_priam"_$node"
            currentNode=$(($currentNode+1))
        fi
        i=$(($i+1))

        # exit if we can't find enough nodes runningUWPlace
        if [ $i -gt 31 ]; then
            echo "Failed to run UWPlace on enough nodes in the priam cluster"
            exit 1
        fi
    done
}

function breadthFirst {
    # $1 == node count
    count=`expr $1 / 2`
    if [ `expr $1 % 2` -eq 0 ]; then
        medusaCluster 0 $count
        priamCluster 0 $count
    else
        medusaCluster 0 `expr $count + 1`
        priamCluster 0 $count
```

```
    fi
}

#-----------------------
# MAIN
#----------------------

if [ $# -lt 4 ];then
    usage
fi

# process args
for ARG in $*; do
case "$ARG" in
    -t)
        shift
        type=$1
        ;;
    -n)
        shift
        nodes=$1
        ;;
    -r)
        shift
        runCycles=$1
        ;;
    -N)
        shift
        N=$1
        ;;
    -xmin)
        shift
        xmin=$1
        ;;
    -xmax)
        shift
        xmax=$1
        ;;
    -ymin)
        shift
        ymin=$1
        ;;
    -ymax)
        shift
        ymax=$1
        ;;
     *)
        shift
        ;;
esac
done

echo "type = " $type " #nodes = " $nodes


# initialize the AgentTeamwork team
echo "Initializing the Commander: $COMMANDER"
runningUWPlace $COMMANDER
if [ $? -ne 0 ];then
  echo "Failed to start UWPlace on Commander: $COMMANDER"
  exit 1;
fi

echo "Initializing the Bookkeeper: $BOOKKEEPER"
runningUWPlace $BOOKKEEPER
if [ $? -ne 0 ];then
  echo "Failed to start UWPlace on Bookkeeper: $BOOKKEEPER"
  exit 1;
fi
```

```
echo "Initializing the Sentinel: $SENTINEL"
runningUWPlace $SENTINEL
if [ $? -ne 0 ];then
  echo "Failed to start UWPlace on Sentinel: $SENTINEL"
  exit 1;
fi

# prepare run commands
UPARGS="runCycles_"$runCycles"_simSize_"$N"_xmin_"$xmin"_xmax_"$xmax"_ymin_"$ymin"_ymax_"$ymax
inject="java -cp UWAgent.jar:GridTcp.jar:MPJ.jar:. UWInject -p 20000 $COMMANDER CommanderAgent
S_$SENTINEL"
userprog="B_$BOOKKEEPER -m 4 -j agents.jar,GridTcp.jar,MPJ.jar -s Wave2D U_Wave2D_dummy_2000_-
np_"$nodes"_"$UPARGS"
C_Wave2D_Communicator_DataLoc_Datatype_ExecStreamReaderThread_GridComm_IRecvThread_ISendThread_Jav
aComm_MPJBAndOp_MPJBool_MPJBOrOp_MPJBXorOp_MPJByte_MPJChar_MPJ_MPJDouble_MPJFloat_MPJInt_MPJLAndOp
_MPJLong_MPJLOrOp_MPJLXorOp_MPJMaxLocOp_MPJMaxOp_MPJMessage_MPJMinLocOp_MPJMinOp_MPJObject_MPJProd
Op_mpjrun_MPJShort_MPJSumOp_Op_Request_Status"

# execute the appropriate cluster configuration
case "$type" in
    b)
        echo "type: breadth first"
        breadthFirst $nodes
        echo "Injecting Program"
        $inject $CL_medusa $CL_priam $userprog
        ;;
    d)
        echo "type: depth first (medusa)"
        if [ $nodes -lt 33 ]; then
            medusaCluster 0 $nodes
            echo "Injecting Program"
            $inject $CL_medusa $userprog
        else
            medusaCluster 0 32
            priamCluster 32 $nodes
            echo "Injecting Program"
            $inject $CL_medusa $CL_priam $userprog
        fi
        ;;
    u)
        echo "type: depth first (uw1-320)"
        if [ $nodes -lt 33 ]; then
            priamCluster 0 $nodes
            echo "Injecting Program"
            $inject $CL_priam $userprog
        else
            priamCluster 0 32
            medusaCluster 32 $nodes
            echo "Injecting Program"
            $inject $CL_priam $CL_medusa $userprog
        fi
        ;;
     *)
        usage
        ;;
esac
```