CSS497 Undergraduate Research

Real Application Evaluation Using Agent Teamwork with JNI

6/11/2008
CSS497 Final Report
Kevin Archer

Introduction

I spent the last quarter of a total of 11 weeks working as an undergraduate research assistant for Professor Munehiro Fukuda and Professor Michael Stiber on running a real application on Agent Teamwork. This final report documents how to run each version of the DCT simulation, the performance metrics gathered during the project, and the list of bugs and problems with the JNI code.

Agent Teamwork is a high-through-out grid computing middle system that employs mobile agents for job dispatching. Dissociated Cortical Tissue simulation is a simulation based off a Liquid State Machine that simulates neuronetworks. Its task is to simulate how they communicate and how they connect and form. To support the work being done with Agent Teamwork and the DCT simulation I was charged with parallelizing the simulation and test it on Agent Teamwork. Through the course of this project, however, the project's original goals had to be revised.

My project's original goals were two fold. First, I was to test a real c++ application on Agent Team using the JNI that was developed earlier by another student. In addition to testing the JNI I was also to use the same C++ application on Globus to do performance comparisons between Globus native execution and Agent Teamwork's native execution.

The second major goal was to address the concerns of Professor Michael Stiber. The first concern was to make his current DCT simulation run in a shorter period of time. The idea behind it was to be able to run the simulation using larger parameters and simulation space. The second concern of Professor Stiber was being able to run the simulation multiple times using different "scenarios" of parameters in order to explore different behaviors of the simulation. There are many different things which are not fully understood about neurons and synapse behavior which the simulation scenarios could examine.

To accomplish both goals I would take Professor Stiber's DCT simulation and parallelize the code in order to allow it to run in parallel on multiple computers. Once done I would then run it on top of Agent Teamwork to test for performance and fault tolerance.

These goals were changed during the project due to several factors. First, the code that constructed the JNI and native code execution of Agent Teamwork had several bugs within it. These bugs put the project behind schedule with parallelizing the simulation. The second factor was the unexpected performance result while just testing the parallel version of the simulation. The test revealed that the simulation was taking an enormously longer time to complete. After looking at the behavior some more it was discovered that the communication overhead of Message Passing was sucking away any performance improvement of parallelizing the simulation. With this knowledge further tests were done to determine what the overhead was for MPJ and the MPICH of JNI part for Agent Teamwork. The tests showed that the overhead caused by MPJ was the real culprit behind the performance issues. The overhead with the MPICH portion was negligible. With this knowledge it became clear that the parallel version of the DCT simulation was not going to meet the first concern of Professor Stiber. Thus it was decided to instead deal with the second concern of Professor Stiber which was to be able to

run multiple simulation scenarios in a relatively short time frame. Thus the original simulation code was modified to allow it to run multiple parameter files and scenarios using Agent Teamwork. Then the simulation was executed with Agent Teamwork using increasing number of computers from 1 to 32. This was done using 32 different parameter files for each test in order to have valid results.

Overview of DCT

The following diagrams document the overall structure of the simulation. Note that these show diagram from a high level abstraction.

MPI Version

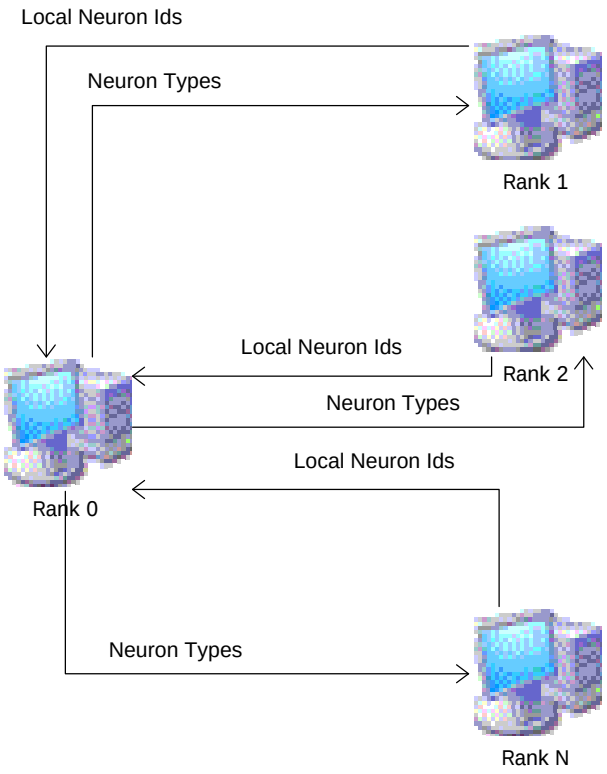
In the MPI version of the DCT simulation there are four major steps to the flow of the simulation. First, each rank after Rank 0 calculates the # of neurons they will simulate and then sets up the network simulation.

The second step is that Rank 0 determines the types of neurons based on the neuron information sent by all other nodes. Rank 0 then sends back the neuron type information about each ranks' neurons to the corresponding rank.

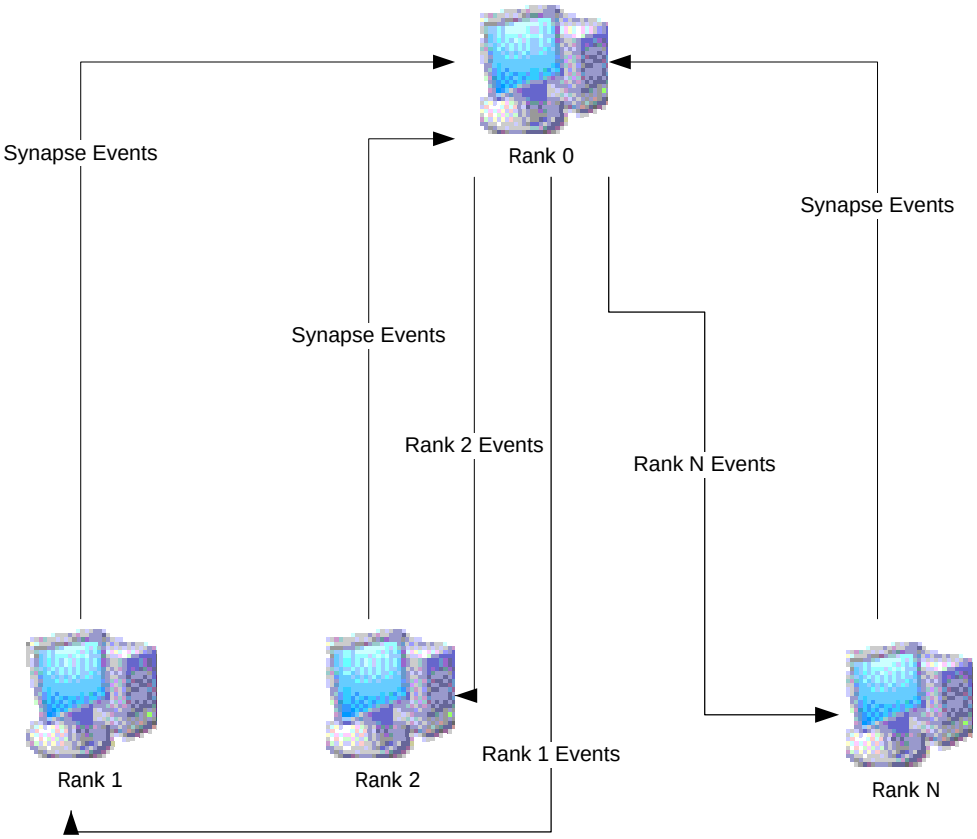
Third, each rank enters the growth and connectivity loops. It begins by first entering into the network class to simulate the connectivity of the neurons. Inside this class it loops through first each neuron and sends to Rank 0 any events that were generated. Rank 0 then takes all of those events and schedules them in a queue and sends back to each Rank the events they need to go through. Then each rank finishes by processing each synapse object and event and then loops back to the beginning to start again.

Fourth, after simulating the connectivity the simulation moves to updating Growth. Here if new connections are made on different ranks then that information is sent between the two ranks which have the connection. Through this each rank will update each neuron and synapse object in the simulation to reflect these conditions.

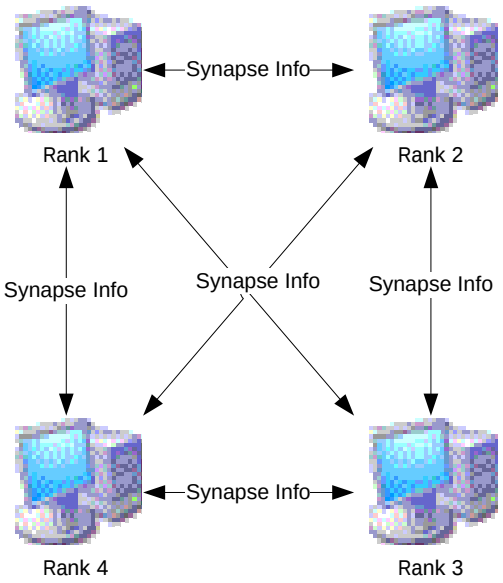
Initial Setup Communication



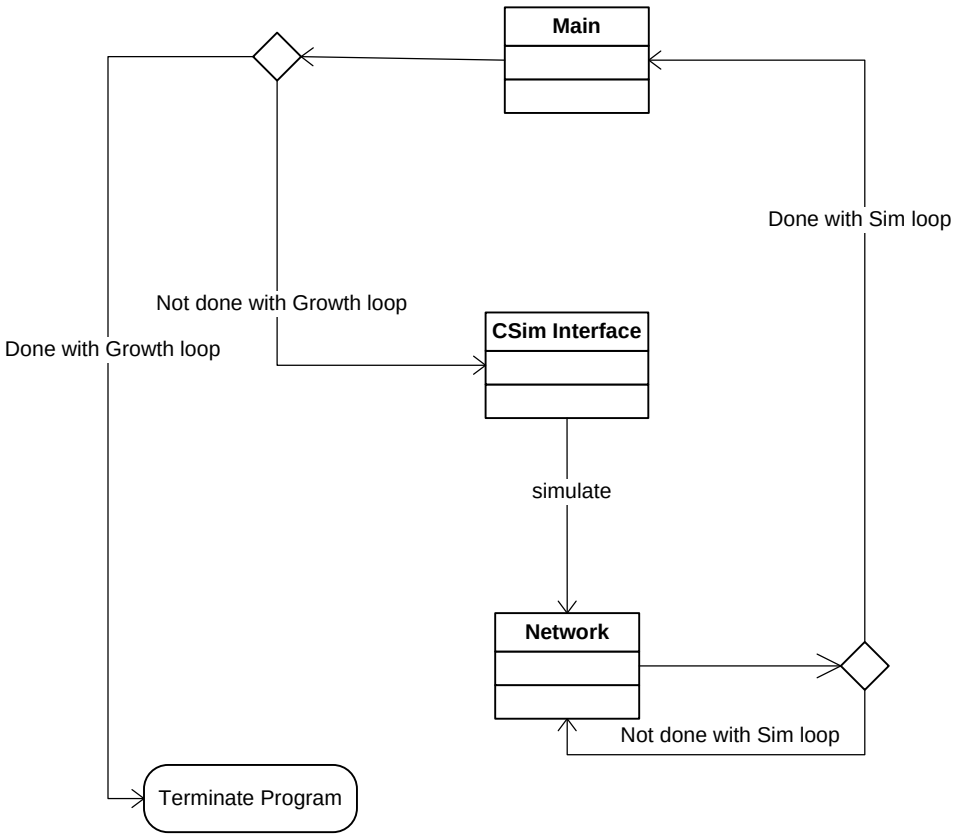
Event Communication



Growth Update Communication



Basic Simulation Flow



Multi Scenario Version of DCT

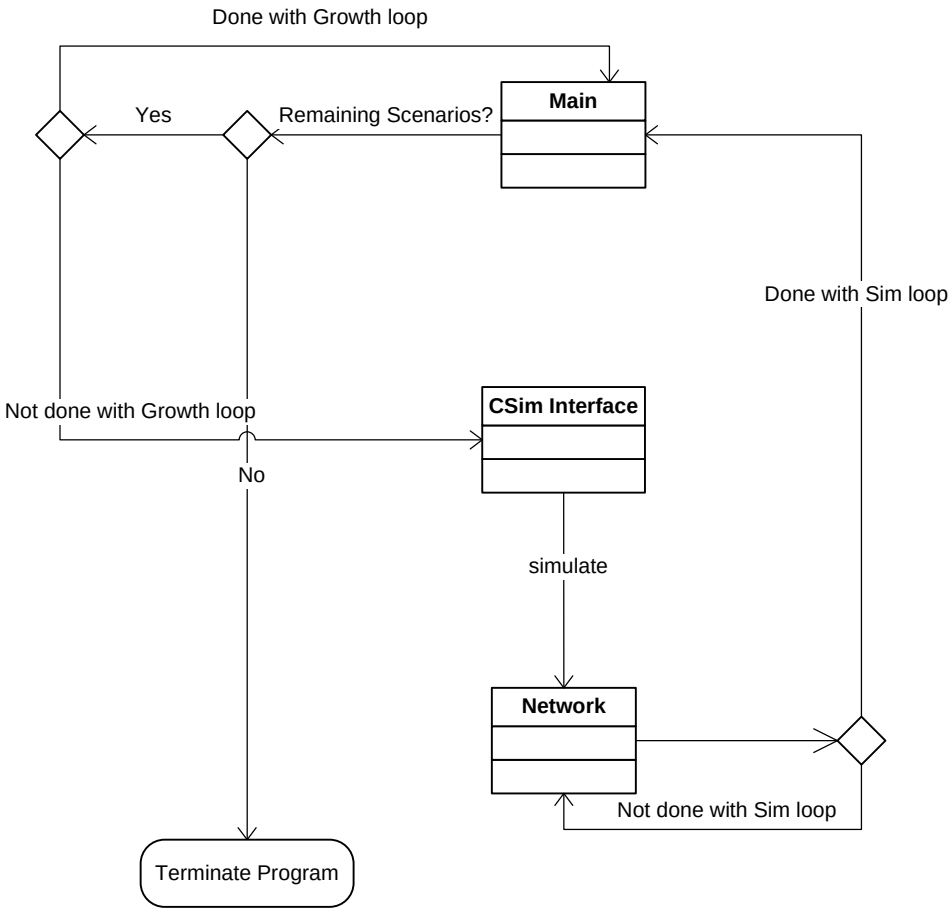
In this version there are five major steps to the flow of the simulation.

First, each rank will calculate how many scenarios they will be running and then the offset into those scenarios. The offset allows each rank to only read the correct parameter files and the output files to write to.

Second, the simulation reads from the scenario files and sets up the simulation.

Third, the simulation enters into the main simulation loop. In this loop the simulation will go into the network level and simulate the connectivity by looping multiple times there. After this connectivity phase the simulation will return to the main loop and update growth information. After this main loop the simulation will write all information help in the simulation to the output file from the list of arguments first passed in.

The fourth step is that the simulation will loop back to the next scenario. Before it does this the simulation tries to reset all objects to make the simulation state pristine for the next scenario.



How to Run DCT Simulation

MPICH Version of DCT

In order to run this version of DCT several things are required. First, there must be a single parameter file in xml format. This file needs to have within it all the parameters needed to run a particular simulation scenario as defined by the KllSimulation. Second, in its current state this version of the Dissociated Cortical Tissue simulation needs as many output files as the number of machines that will run the simulation. All you need to do is provide some suffix name plus a number. The numbers should go from 1 to n where n is the number of computers you wish to use in the simulation. The DCT simulation stores at the end of its execution the results of the simulation in each xml file.

The simulation is found on the uwagent account on UWB network (Medusa) within the following directory: *~home/uwagent/agentteamwork-dev/applications/DCT/MPI* Inside this directory are all of the header and .cpp files needed to run the simulation. Inside this directory there must also be the files for the parameters and output as this version will read and write from this directory. To compile this version simply run the compile2.sh which is a shell script designed to compile all this simulation into one .so file. If you want to change which files that need to be compiled, simply open up this shell script and at the last instruction change the file names after this line: g++ -shared –o _libdriver.so_

GridJNI_library.cpp If you want to change the name of the file which is used to inject within Agent Teamwork simply change _libdriver.so_ to _lib<name desired>.so_ where <name desired> is the new name you wish to have.

After compiling the user program, it along with the file libJavaToCpp.so should be moved to the folder /files within the same directory. This is needed because from this folder does the file thread, which transports user's jobs to Agent Teamwork, sends the user's file and libJavaToCpp.so to Agent Teamwork.

After moving the two files to the /files directory then the simulation is almost ready to be executed on top of Agent Teamwork. The next step to do is check the runAteam.sh, which will run the user program, to see which computers should will be running. This is found after S_ which signifies the beginning declaration of what computers will run sentinel agents. The first computer name after S_ will be the main sentinel agent which just handles top end communication between each of its children sentinel agent processes. This agent doesn't do any work. All others afterward represent the sentinel agents that will do the actual work of the simulation. You may use any computer node that Agent Teamwork would have access to. One other thing to check is what computer will be running the bookkeeper agent. This is found after the B_ line. Another thing to check is the number of arguments that need to be passed in. This version of the simulation only needs four arguments in addition to the name of the user program and the port number which runUWPlace.sh is running under. By knowing which computers run the bookkeeper and sentinel agents you can move to the next step.

The next step is to log onto those computers that run the sentinel and bookkeeper agents and run the runUWPlace.sh. To run this shell script you simply need to type it in and pass a port # that you wish to use. This port # must be the same on all computers that run an agent from Agent Teamwork.

The next step after having all computer nodes running runUWPlace.sh is to run the file thread. The computer that runs this should be in *~home/uwagent/atteamwork-dev/GUI* directory. Once there execute this line: java -cp ..:. FileThread -p 12345 –d /home/uwagent/agentteamwork-dev/applications/DCT/MPI/files/ The port # can be changed by changing the # following the –p. If the simulation directory changes then simply replace */home/uwagent/agentteamwork-dev/applications/DCT/MPI/files/* with the full path name of the new location.

The final step to executing the simulation is to run the runAteam.sh. The usage of this command is: runAteam.sh, port # of runUWPlace.sh, name of user compiled program, and then any arguments the user program needs. The current simulation needs to be passed –t dice.xml –o output (Note output can be replaced by any suffix of the file name you want to have.

To sum up this is what is needed to run the simulation:
1. run compile2.sh
2. move _libJavaToCpp.so_ and _libdriver.so_ to */home/uwagent/agentteamwork-dev/applications/DCT/MPI/files*
3. run runUWPlace.sh port # on all machines that run agents
4. run java -cp ..:. FileThread -p 12345 –d /home/uwagent/agentteamwork-dev/applications/DCT/MPI/files/ within the *~home/uwagent/atteamwork-dev/GUI* directory.
5. run runAteam.sh port # of runUWPlace.sh libdriver.so –t parameter file –o output file
(i.e. runAteam.sh 30005 libdriver.so –t dice.xml –o output)

Multi Scenario Version of DCT

In order to run this version of DCT several things are required. First, there must be a single parameter file in xml format for each scenario to be executed. These file needs to have within them all the parameters needed to run a particular simulation scenario as defined by the KllSimulation. Second, in its current state this version of the Dissociated Cortical Tissue simulation needs as many output files as the number of scenarios. Thus there needs to be a 1:1 ratio of parameter and output file. The DCT simulation stores at the end of its execution the results of the simulation in each xml file.

The simulation is found on the uwagent account on UWB network (Medusa) within the following directory: *~home/uwagent/agentteamwork-dev/applications/DCT/* Inside this directory are all of the header and .cpp files needed to run the simulation. Inside this directory there must also be the files for the parameters and output as this version will read and write from this directory. To compile this version simply run the compile2.sh which is a shell script designed to compile all this simulation into one .so file. If you want to change which files that need to be compiled, simply open up this shell script and at the last instruction change the file names after this line: g++ -shared –o _libdriver.so_ GridJNI_library.cpp If you want to change the name of the file which is used to inject within Agent Teamwork simply change _libdriver.so_ to _lib<name desired>.so_ where <name desired> is the new name you wish to have.

After compiling the user program, it along with the file libJavaToCpp.so should be moved to the folder /files within the same directory. This is needed because from this folder does the file thread, which transports user's jobs to Agent Teamwork, sends the user's file and libJavaToCpp.so to Agent Teamwork.

After moving the two files to the /files directory then the simulation is almost ready to be executed on top of Agent Teamwork. The next step to do is check the runAteam.sh, which will run the user program, to see which computers should will be running. This is found after S_ which signifies the beginning declaration of what computers will run sentinel agents. The first computer name after S_ will be the main sentinel agent which just handles top end communication between each of its children sentinel agent processes. This agent doesn't do any work. All others afterward represent the sentinel agents that will do the actual work of the simulation. You may use any computer node that Agent Teamwork would have access to. One other thing to check is what computer will be running the bookkeeper agent. This is found after the B_ line. After checking which computers are running agents the next thing to look at is the arguments. Currently all names of parameter files and output files are hardcoded into this shell script. As such if these file names change then that change should be reflected within this script. By knowing which computers run the bookkeeper and sentinel agents you can move to the next step.

The next step is to log onto those computers that run the sentinel and bookkeeper agents and run the runUWPlace.sh. To run this shell script you simply need to type it in and pass a port # that you wish to use. This port # must be the same on all computers that run an agent from Agent Teamwork.

The next step after having all computer nodes running runUWPlace.sh is to run the file thread. The computer that runs this should be in *~home/uwagent/atteamwork-dev/GUI* directory. Once there execute this line: java -cp ..:. FileThread -p 12345 –d /home/uwagent/agentteamwork-dev/applications/DCT/files/ The port # can be changed by changing the # following the –p. If the simulation directory changes then simply replace */home/uwagent/agentteamwork-dev/applications/DCT/files/* with the full path name of the new location.

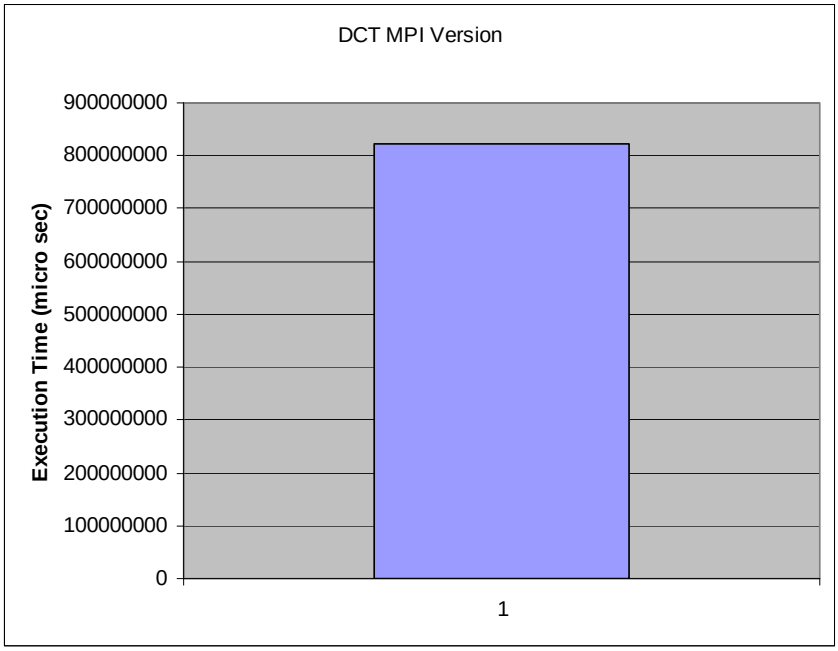
The final step to executing the simulation is to run the runAteam.sh. The usage of this command is: runAteam.sh, port # of runUWPlace.sh, name of user compiled program

To sum up this is what is needed to run the simulation:
1. run compile2.sh
2. move _libJavaToCpp.so_ and _libdriver.so_ to */home/uwagent/agentteamwork-dev/applications/DCT/files*
3. run runUWPlace.sh port # on all machines that run agents
4. run java -cp ..:. FileThread -p 12345 –d /home/uwagent/agentteamwork-dev/applications/DCT/files/ within the *~home/uwagent/atteamwork-dev/GUI* directory.
5. run runAteam.sh port # of runUWPlace.sh libdriver.so
(i.e. runAteam.sh 30005 libdriver.so)

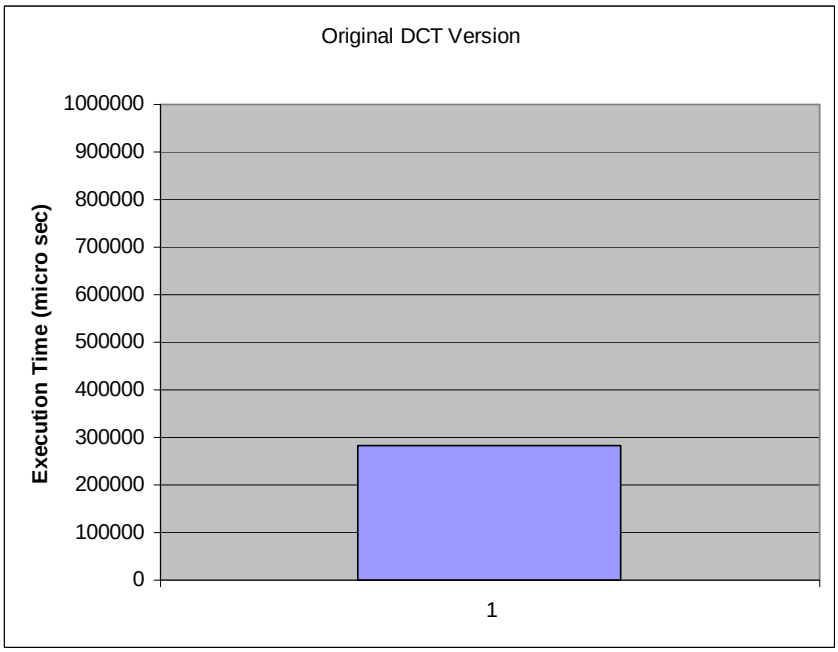
Results from Research Process

MPICH Version of DCT

The following graphs and tables show the overheads and results from this version of DCT.



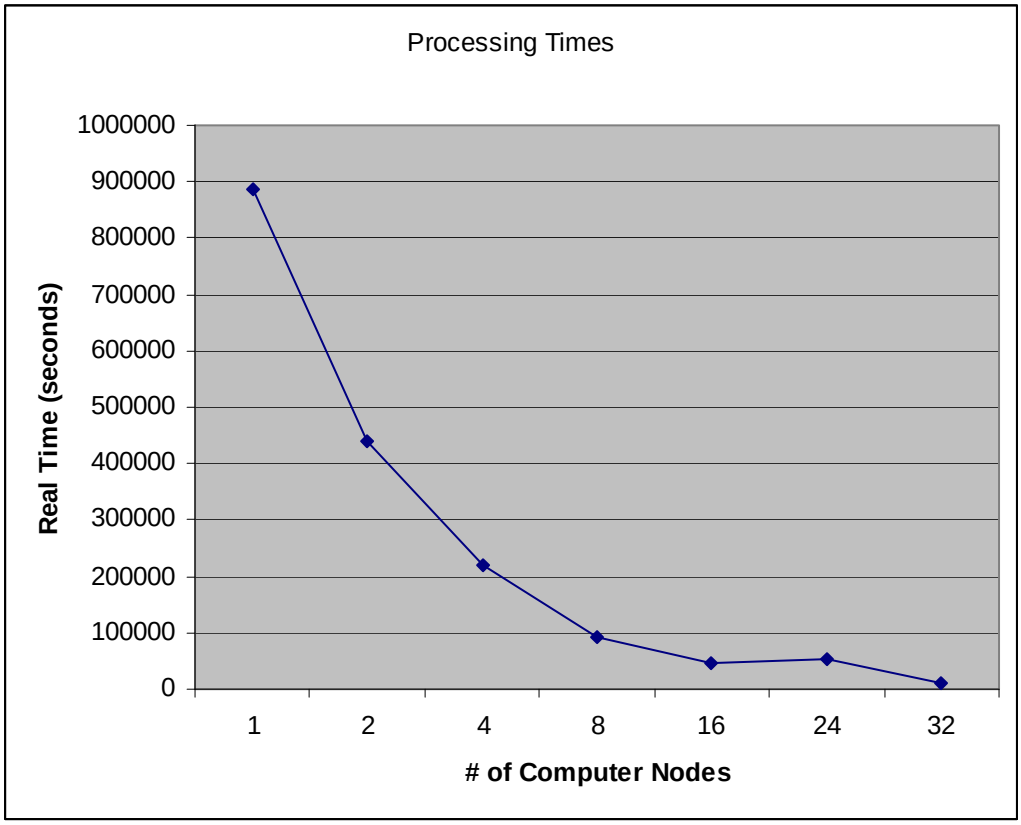
Simulation Work	Communication Overhead	Total Time
0.405 seconds	821.674 seconds	828.294 seconds
0.443 seconds	820.396 seconds	827.265 seconds
0.422 seconds	821.712 seconds	828.73 seconds



Non-MPI Form
0.279 seconds
0.282 seconds
0.219 seconds

Multi Processing Version of DCT

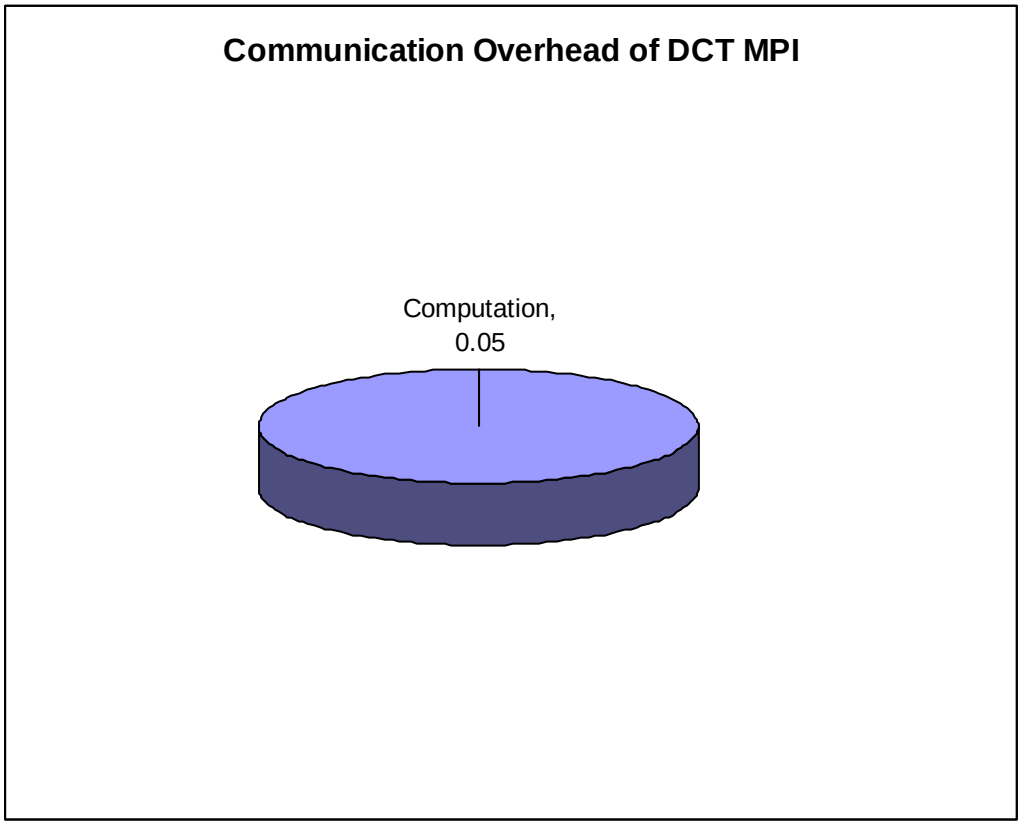
The following graphs and tables are the results of running this version of the simulation on Agent Teamwork.



# of computer nodes used	Simulation Time
1	886463.865 seconds
2	440853.734 seconds
4	221297.573 seconds
8	91992.146 seconds
16	47176.634 seconds
24	51524.564 seconds
32	11036.885 seconds

MPI/MPI-java Overhead Models

The following graphs and tables show the results of testing MPICH on Agent Teamwork using JNI code and MPJ on Agent Teamwork.



Simulation Work	Communication Overhead	Total Time
0.405 seconds	821.674 seconds	828.294 seconds
0.443 seconds	820.396 seconds	827.265 seconds
0.422 seconds	821.712 seconds	828.730 seconds

MPICH Sender/Receiver

This tables shows the results of a simple program in C++ where rank 0 just sent a single value to rank. This program used a loop of 100000.

Sender	Receiver
MPI Rank 0	MPI Rank 1
21.680 seconds	34.001 seconds

25.496 seconds	36.407 seconds
22.922 seconds	34.986 seconds

MPJ Sender/Receiver

This tables shows the results of a simple program in Java where rank 0 just sent a single value to rank. This program used a loop of 100000.

Sender	Receiver
MPI-java Rank 0	MPI-java Rank 1
17.064 seconds	17.389 seconds
16.975 seconds	17.268 seconds
16.894 seconds	17.202 seconds

MPICH Sender and Receiver

This table shows the results of a simple program in C++ where both rank 0 and 1 send and receive from each other a single value. This program used a loop of 10000 to match the same loop as the one used in DCT MPICH version and was executed three times.

Rank 0	Rank 1
807.253 seconds	807.255 seconds
808.005 seconds	808.006 seconds
807.378 seconds	807.503 seconds

MPJ Sender and Receiver

This table shows the results of a simple program in Java where both rank 0 and 1 send and receive from each other a single value. This program used a loop of 10000 to match the same loop as the one used in DCT MPICH version and was executed three times.

Rank 0
808.638 seconds
809.191 seconds
808.931 seconds

JNI Difficulties

Throughout this research project we encountered several bugs and issues related to JNI code that was originally created by another undergraduate student named Henry Sia. The following list and table show all the problems which were encounter and how each has been dealt with.

List of Problems	Status
Output Buffer Problem (JNI)	Possibly Fixed (no known issues to date)
Overflowing of user program arguments	Fixed
User program register local variables (JNI)	Solution found (not tested)
User program snapshot (JNI)	Solution Found (not tested)
MPICH/MPJ Bcast not working	Fixed
MPICH/MPJ ISend and IRecv not working	Fixed
Mapping Java data types to C++	Fixed
Mapping MPJ data types to MPICH	Fixed
Dynamic Opening Problem (file too short)	Possibly Fixed (/tmp)
File I/O through JNI code	Unknown

Output Buffer – There was an issue where an unexpected error would occur in the execution of the user program. This error seems to have stemmed from when the user program is calling some MPI function. This would then call the JNI code which had several print messages using cout to display debug information. The cout statements had the use of endl to end the line on the display. This seemed to cause a problem when the JNI called JavaToCpp.java and displayed to the buffer using it's I/O method.

Status – This problem has been addressed by using "\n" instead of endl and using cerr instead of cout. Currently it is unknown whether this has solved the problem but so far no other issues have arisen from output in JNI.

Overflowing of User Program Arguments – There was problem discovered about how user program arguments were being handled within JNI. The problem was that if there were more than two arguments being pass then those arguments would be stored in the stack and overwrite critical code for Agent Teamwork. This code handled the spawning of a new Thread to handle and incoming message to a sentinel agent.

Status – This problem has been fixed. It was discovered that problem stemmed from the fact that the arguments were being dynamically stored in memory. It was fixed by making the space preset instead of dynamic.

User Program Register Local Variables (JNI) – Originally with JNI there was no way for the user program to register variables that are needed to maintain the state of the application. The registering of such variables is needed so that the user program could then call the snap shot function and save its execution state in case of a fault.

Status – A solution has been found and has been coded. However, at this time it has not been thoroughly tested to ensure that it works.

User program snapshot (JNI) – Originally with JNI there was no way for the user program to call the snapshot function built into Agent Teamwork.

Status – A solution has been found and has been coded. However, at this time it has not been thoroughly tested to ensure that it works.

MPICH/MPJ Bcast Not Working – Although MPICH functions that mapped to MPJ functions such as

MPI_Rank and MPI_Send, MPI_Bcast was not working appropriately. The problem was that the ranks that were supposed to receive the broadcast were not receiving the values being sent. Upon investigation it was discovered that the ranks that were to receive the broadcast were not calling the Bcast function on the Java side of things in MPJ.

Status – This problem has been fixed. The solution was removing an if statement in JNI that only allowed the root rank to Bcast and not the other ranks.

MPICH/MPJ ISend and IRecv Not Working – There was a problem with the functions MPI_ISend and MPI_IRecv not working at all. If one tried to use them, then a fatal error message would display telling the user of an exception with MPJ. Upon investigation it was discovered that the wrong number of arguments were being pass from the JNI portion to the Java portion of Message Passing.

Status – This problem has been fixed. The solution was to add in the offset of the array being sent or received. In the MPICH version there is no array offset being passed, but in MPJ there is. The solution was to just pass a zero offset in order to mimic the behavior of MPICH versions of these calls.

Mapping Java data types to C++ - A problem was discovered with the use of data types other than doubles. The problem caused errors when trying to MPI functions with other kinds of data types that were not doubles. Upon investigation it was discovered that the problem was from the fact the C++ data types were being directly mapped to Java data types. This was a problem because C++ data types don't map to Java data types. For instance a C++ long doesn't equal a Java long.

Status – This problem has been fixed. The solution was to research how each data type was stored in C++ and then map it to a data type in Java that had the same storage.

Mapping MPJ data types to MPICH – A problem existed with the mapping of data types being used by MPICH and MPJ. This problem's source was from the problem discussed above about the differences between C++ and Java data types. Here on the JavaToCpp.java side the data types were being encoded to the wrong data types equivalent to C++.

Status – This problem has been fixed. The solution was the same as the problem above.

Dynamic Opening Problem (file too short) – There is a problem with the dynamic linking of the user program and trying to open it. The problem is that when each agent wishes to read a file the file is downloaded to the same shared directory as other agents. Thus there is the potential that the file could be overwritten while another agent still has the file being downloaded. This resulted in the dynamic linking to error saying that the file in question has not been fully downloaded. This problem becomes far more likely to happen as the number of computers being used in Agent Teamwork increases and as the size of the user program increases.

Status – A solution has been found for this problem. The fix was to make the files be downloaded to the tmp directory which is local to each machine. This way the files would never conflict on other machines. Though this is a straight forward solution it is unknown where this is the true solution to this problem. So far, however, this problem has not resurfaced since implementing this solution.

File I/O through JNI code – There is a problem in Agent Teamwork's JNI dealing with File I/O. Originally Agent Teamwork's JNI didn't support any functionality to handle user program file I/O in C++. The user program would just call the functions like normal and get the original behavior. This

was a problem since sometimes this would mean that a user program could fail in opening a file when running on Agent Teamwork since Agent Teamwork moved the user program to different areas and not the files they might read or write to.

Status – This problem still persists. However, it should be noted that a potential solution has been created. This solution, however, has not been added to the code and has yet to be tested.