TO: **Professor Munehiro Fukuda**
FROM: **Abdulhadi Ali Alghamdi**
SUBJECT: **TERM PAPER- Performance Analysis of Hybrid OMP/MPI and MASS, MASS Library Development**
DATE: **June 13, 2015**

Overview

Over the course of the first half of Spring quarter, 2015, I worked on two main applications that test the performance of Multi-Agent Spatial Simulation (MASS) in comparison with hybrid OpenMP (OMP) and MPI: Sugarscape, and Wave2D. These two applications are parallelized in a cluster of **up to 4 machines**, utilizing **4 processes**, and **4 threads**, provided by the University of Washington--Bothell in their Linux lab.

Over the course of the second half of Spring quarter, 2015, I worked on refactoring and modifying the Java MASS library. Specifically, I was providing custom classes for recoverable exception handling, classloading, interfaces, and general refactoring. After that, I worked on integrating the debugger to the library.

Performance and Programmability: Hybrid MPI/OpenMP vs MASS

Sugarscape

Sugarscape is a social simulation about monitoring multiple agents' survivability in an environment (places: a 2-dimensional grid). Their survivability depends on their sugar consumption, their metabolism, and their location in the grid.
In the simulation, I allocate the sugar and the places, followed. Then, I create the agents in the number specified by the user. Each process then is allocated a chunk of the agents, given their relative position and metabolism to the rest of the agents handled by other processes. After that begins the traversal procedure: some agents find sugar to consume, all agents have their metabolism changed accordingly, and all agents relocate randomly to survive.

Wave2D

Wave2D is an environmental simulation about how waves disseminate from a single point of causation. A wave is disseminated north, south, east, and west, in an equilibrial fashion. The new cell of a wave is calculated by using Schrödinger's wave formula when t >= 2; however, when less than that, assisted formulas were used (all provided in CSS 543's assignment #2).

Performance Analysis (P=processes, T=threads, ~time in microseconds)

My metrics for the tests conducted were, in terms of PXT, were as follows:

- 1X1
- 1X4
- 4X4

The reason behind these choices was that only at provided PXT metrics were the results significant to report. Generally, both MPI and MASS show a significant drop in time after utilizing 4 threads at a time; however, as the number of processes increases, MASS performs similarly to MPI. Below are my collected results, provided specific variables for each test.

Noteworthy is results were collected at around 0200-0300--low traffic time. Outliers (off by 30% from average margin) were excluded. Average was of 20 runs.

Sugarscape

`agents = 60 | max sim. time = 20 | sim. size = 70`

MPI	MASS
P=1, T=1 ~8922	P=1, T=1 ~12132
P=1, T=4 ~5801	P=1, T=4 ~7819
P=4, T=4 ~4914	P=4, T=4 ~6661

Wave2D

`N (sim. size) = 20`

MPI	MASS
P=1, T=1 ~7511	P=1, T=1 ~10590
P=1, T=4 ~4202	P=1, T=4 ~5898
P=4, T=4 ~3660	P=4, T=4 ~5053

Final Notes

Overall, MPI initially performed better than MASS. As the cluster was better utilized, MASS began to improve drastically--still, however, not remarkably comparable to MPI's performance. My assumption is that as the cluster grows vastly, the two performances will eventually become well comparable to one another.

Another note to mention is the time it took to learn and implement MPI code versus MASS code. Needless to say, MPI took way longer to learn, design, and implement than MASS did: it took me about 1 effective hour to learn MASS library, whereas it took me about 7 effective hours to learn how to use hybrid MPI and OpenMP.

Java MASS Library Development

After the first half of the quarter was over, I began preparing for improvements on the Java MASS library. My two main duties were as follows:

- Enhance the learning, design, implementation, and debugging experience of the Java MASS
    - Exception handling-specific classes
    - Central factory for classloading
    - Interfaces for unit-testing
    - General refactoring for code consistency
- Integrate the debugger into the library.

Enhancing MASS Usage

During early stage development of the hybrid and MASS applications, it came to my attention that MASS was quite difficult to work with and understand. There exists code smells, inconsistencies in development patterns, and design flaws that hinder progress in MASS releases. As I was approaching the end of the first half, Matthew Sell and I began discussing what development MASS could use to make it more mature of a library.
Below, I introduce what I believe are the four most cost-effective changes that need to be fully implemented before releasing the library.

Exception Handling

By far, exception handling is a high priority in making the library ready for release. Currently, a single MASS application will terminate properly. As soon as we attempt to run multiple applications, however, the whole system will terminate because of one application reaching a fault. For instance, multiple applications running on Glassfish would result in the server shutting down because of one fault, rendering the other perfectly healthy applications dysfunctional.

Classloading

Currently, classloading occurs in Agents_base and Places_base. This is fine, as separation was required in order to understand where from the errors were coming. Unless very frequent changes occur in the design and structure of those classes, loading should occur in a factory–an isolated class where it is easy to incorporate and implement features. This should help not only understand the operations of Agents and Places, but pinpoint where classloading occurs should it happen in more classes in the future.

Interfaces

Major classes in the library have no interfaces. This is required for more efficient unit testing, as well as understanding what functions do in the library; the separation of functions into abstractions will allow newcomers on the research team to more fluently understand the development of the library.

General Refactoring

This is an ongoing procedure to guarantee code consistency and organization. From my understanding, researchers come and go on the MASS library, which leaves plenty of space for conflicts in approach, design, and implementation. Consistent general refactoring for the library at the end of every development span will make it much easier

for the newcomers and returning members to start where they/others left off without any lag in understanding the sequence and approach to the development of the library.

Integrating the Debugger

Currently, the debugger runs as a separate program to the MASS library. How it runs is by overriding DebugData functions in custom Agents and Places, copying Debugger files into MASS application directory, allocating a Place for them, and exchanging the data in each iteration. For more info, refer to Hongbin's MASS Debugger Manual.

My work is to reduce the complexity of the debugger by integrating it to the library, and automating the functionality of it so the developers do not need encounter any difficulties in such a basic feature of the library. The idea is as follows:

1. Automate Debugger classloading in MASS directory.
2. Allocate Place for debugger in Places_base:
    a. Add an additional Place to the size needed by application.
    b. Secure Place from application access and use.
3. Handle exchangeAll and callAll iterations in the application.

Currently, there is no reliable way to substitute the exchange of data without creating multiple Places for multiple debuggers, should the application require multiple matrices. This, of course, can be averted through data exchange manipulation, such as:

- Handlers (for whether to use Places/Agents being passed through Places constructor
- Interface Agents and Places to implement the debugger functions (setDebugData & getDebugData) vs overriding the aforementioned functions.
- Create a custom class that iterates exchangeAll and callAll as needed for the debugger simultaneously alongside the application.

Final Notes

MASS library, once learned and used, serves as a good hassle-free multi-agent library for distributed systems; however, personally speaking, a good portion of learning the library came from previous exposure to it. I think that researchers working on the library, as they come and go, could use the following recommendations:

- Comprehensive documentation: many team members may not have the luxury of time to respond to every communication occurring from the new team member, in which case some development time is lost to attempting to figure out how things work.
- Readily-available unit testing: time implementing the features was compromised with developing custom classes that handle certain unit testing aspects.
- Refactoring: code smells are relatively minor in this development stage of the library; however, as time moves on, it'll become increasingly more difficult to control the patterns and readability of the code. Maintenance schedule around the year would be excellent to solve that problem.