Asynchronous and Automatic Migration of Agents in MASS library

Hung H. Ho

A report
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington Bothell

2015

Project Committee:

Professor Munehiro Fukuda, Project Advisor

Professor David Socha, CSS595/596 Instructor

University of Washington Bothell

Abstract

Asynchronous and Automatic Migration of Agents in MASS library

Hung H. Ho

This project looks into designing and implementing asynchronous and automatic migration (or auto migration in short) functionality for agents of MASS library. Rather than issuing multiple `callAll()` and `manageAll()` function calls, only one `callAllAsync()` function call need to be issued and all the agents will execute multiple functions, including spawn, kill, and migrate, asynchronously and independently of one another. This makes full use of computer resources and reduces communication overhead of issuing multiple funcion calls. The project is still ongoing so some sections are missing or may be changed in the final version.

TABLE OF CONTENTS

Page

Chapter 1: Introduction 1
1.1 Problem Definition 1
1.2 Goals 2

Chapter 2: Literature Review 3
2.1 Agent-based Parallel Computing 3
2.2 Distributed Termination Detection 4

Chapter 3: Method 6
3.1 MASS Programming Model 6
3.2 Asynchronous Migration 7
3.3 Auto Migration 12
3.4 Communication Channel 12
3.5 Distributed Termination Detection 14
3.6 Verification 17

Chapter 4: Experiments 20
4.1 Setup 20
4.2 Effect of number of agents 21
4.3 Others 22

Chapter 5: Conclusion and Future Works 23

Bibliography 24

List of Figures 26

List of Tables 27

Glossary 28

Appendix A: Experiment Result 29

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to University of Washington Bothell, Professor Munehiro Fukuda, Professor David Socha, and others who have helped with this project.

Chapter 1

INTRODUCTION

1.1 Problem Definition

Multi-agent simulation is a popular method to model and simulate large-scale social or biological agents and their emergent collective behavior, which may be difficult using only mathematical and macroscopic approaches. Many software frameworks have been proposed to tackle this requirement, such as MACE3J[1] and MASS[2].

MASS, Multi-Agent Spatial Simulation parallelizing library, has been developed at Distributed System Laboratory, UW Bothell, since 2010. The library employs the concept of `Places` and `Agents` to represent an individual simulation space or active animated entities. They are distributed among computing nodes. Places stay the same throughout the simulation while `Agents` migrate and spawn among `Places` in each simulation round, and perform computation. All computation is enclosed in each array element or agent. All communication is scheduled as periodic data exchanges among places or agents. `Agents` can spawn more agents or migrate to places and rendezvous with one another.

Although MASS is faster than sequential execution, some programming issues still remains. Because agent migration is done periodically and synchronously for all agents, in parallel application, where workload cannot be equally divided among them, agents with lighter workload who finish their assignments earlier in a period will be idly while waiting for others to finish and receive the migration call to migrate and/or spawning agents. Also agents have to be disseminated manually through function call rather than automatically. Issuing migration call has communication overhead so this is resource wasteful, not efficient and slow down the simulation.

In this project, we intend to address this issue by implementing asynchronous migration and automatic migration (or auto migration in short) of agents. Hence, unless a coordination/synchronization among agents is needed, agents which finish their assignments can immediately migrate and spawning new agents without waiting for synchronization with other agents and then being issued a migration call or being specified next places to migrate.

1.2 Goals

The goal of the project is to implement both asynchronous migration and auto migration functionality of agents for the Java version, and, if time permits, the C++ version, of MASS library. The reason for choosing this goal is because asynchronous migration and auto migration are closely related. They both target improving MASS library performance by reducing agent migration communication overhead. They also help improve usability of MASS library. Asynchronous migration allows programmers to specify all the functions that need to be executed in one `callAllAsync()` call. Auto migration frees programmers from the burden of figuring how to migrate their agents to perform data analysis task at each place. Thus, for a Master Capstone Project and six-month time frame, this goal is sufficiently challenging. Achieving this goal includes creating an architecture design of the new functionalities, which takes into account compatibility with existing functions of the library; implementation; testing for their correctness using classic parallel problems such as Mandelbrot and real-life application such as Biological Network Motif search and Climate Simulation; and evaluating their performance against existing functionalities of MASS library.

The project report is organized into following chapters. In chapter 2, we look into existing related works. Chapter 3 discusses about the design, implementation of the new functionalities, and how to verify them. Performance evaluation of asynchronous and auto migration are discussed in Chapter 4. Chapter 5 is Conclusion and future works.

Chapter 2

LITERATURE REVIEW

2.1 Agent-based Parallel Computing

In any multi-agent distributed simulation system, communication is always a big overhead factor that drags down the performance of the system. This is easy to understand because in a distributed system, communication usually involve sending data among computing nodes over the network using TCP/UDP or socket connection which is many times slower than reading and executing data from local memory. CPU has processing speed in GHz and RAM has read and write speed in Gibibits magnitude while LAN only has transfer rate of 100Mbits, which is 100 times slower. This inter-node communication overhead can happen between various entities, for example, between processes, between agents, between an agent and a process, etc.

Various researches have been done to improve performance by reducing overhead in certain type of communication. In [3], a framework is implemented which include middle-agent services to load-balance agents among computing nodes in the system and minimize communication between agents as well as agents and their environment. Communication overhead among agents is also studied in [4]. By analyzing historical message exchange information, an optimized topology is deduced before execution so that messages can be redirected efficiently and propagate to all agents in the system in a smaller number of iteration.

Cherie Wasous [5] proposed two auto migration scheme for MASS agents. In one scheme, the number of agents remain unchanged throughout execution. In another scheme, starting with four agents at the center, they will spawn and migrate to the edge of the Places matrix after each iteration. These two schemes dont show much performance improvement but extensive performance study has not been made.

In this project, rather than communication overhead between agents as previous researches, we want to address communication overhead between processes as well as process and agents. We will also revisit previous auto migration schemes for MASS library, try out new auto migration algorithm, and perform a more extensive performance evaluation on them.

2.2 Distributed Termination Detection

Because agent execution is now asynchronous and independent of one another, we need a logic to detect when all agents have completed their execution. In this section, we review existing Distributed Termination Detection algorithms, which are devised to address this type of problem. As discussed more in subsequent Method chapter, MASS's new communication channel developed in this project will be completely asynchronous and ordering is not guaranteed (non-FIFO). As such, we only consider distributed termination detection solutions that accommodate this type of communication channel.

Existing solutions can be loosely classified into two approaches: controller initiation and termination notification initiation. Controller initiation means that a designated process or computing node is in charge of kicking off the termination detection algorithm, once it suspects that distributed termination has happened. If the algorithm returns no, then another round, also called phase, or wave, in some algorithms, need to be done again some time in the future. Termination notification initiation, on the other hand, means that when a process or computing node is terminated (becomes idle/passive), it notifies one or many other processes or computing node. When the number of notifications satisfies certain conditions defined by the algorithm, then the system is considered terminated.

[6], [7], [8], and [9] fall into controller initiation category. The initiator or controller propagates the termination detection message to other nodes or processes in form of ring [6], tree [7], or graph [8][9]. Termination detection message is called signal in [7] and [9], or probe in [6]. Upon receiving the message, each process will record its state or states into the message and propagate the message along the ring, or its child nodes, or neighbors. Eventually the message or messages return to the initiator, and based on the content it can decide whether or not global distributed termination state has been achieved.

[10] and [11] use termination notification initiation approach. In [11], when a process becomes passive (or idle), it will broadcast a termination detection message to its neighbors. Any passive neighbor who receives the message will forward it to its neighbors who have not received the message. If a passive neighbor has no more neighbors to forward the message or all of its neighbors reply with a "ready-to-terminate" (RT) message, it will reply with a RT message itself. Global termination is achieved when all the nodes return RT messages. In [10], eachl processes keeps track of a "ledger" table, which keeps track of the tasks that it receives or finishes. There exists a designated controller, which is in charge of determining global termination. The controller itself also has a "ledger" table, which keeps track of the states of all the processes' "ledger" tables. When a process becomes idle, it will update its ledger table and send the updated report to the controller. Upon receiving the report, the controller updates its ledger table. If the ledger table conditions are satisfied, global termination is concluded. Our distributed termination detection algorithm is similar to [10].

In our approach, each computing node keeps track of some other nodes based on certain conditions. Global termination is achieved when the master node concludes that the nodes that it keeps track of finish their execution.

Chapter 3

METHOD

3.1 MASS Programming Model

The existing architecture can be explained by Figure 3.1. Each computing node has a process, called `MProcess`, running on it. This is represented by the rectangles in the figure. Each `MProcess` contains multiple threads, called `MThreads`. `MThreads` execute concurrently and perform computation on groups of `Places` and `Agents` that are distributed at that computing node. In the migration/spawn/kill phase of agents, if there is a need for an agent to migrate to a different node, communication threads are created at the source nodes and destination nodes, to handle the migration and these threads stop and are destroyed after each migration phase.

The MASS library employs a network of computing nodes, which communicate with one another using Socket API. Originally each node establishes a connection to each of the other nodes at the beginning of execution and these connection remain open throughout the execution. Requests and responses are handled by the main thread in a synchronous first-in-first-out manner. Other classes in the library are as following

- `MASS_base` contains all the shared properties that are used by other classes. It also keeps track of the socket communication among the nodes.

- `MASS` is a subclass of `MASS_base` and has other properties that are used by the master node only.

- `Agents_base` encapsulates variables and logic related to different agent collections such as `callAll`, `manageAll`.

- `Agents` is a subclass of `Agents_base` and has other properties and methods that are only executed by the master node.

- `Places_base` captures logic and variables related to place collections such as `exchangeAll`.

- `Places` is a subclass of `Places_base` and has other properties and methods that are exclusive to master node

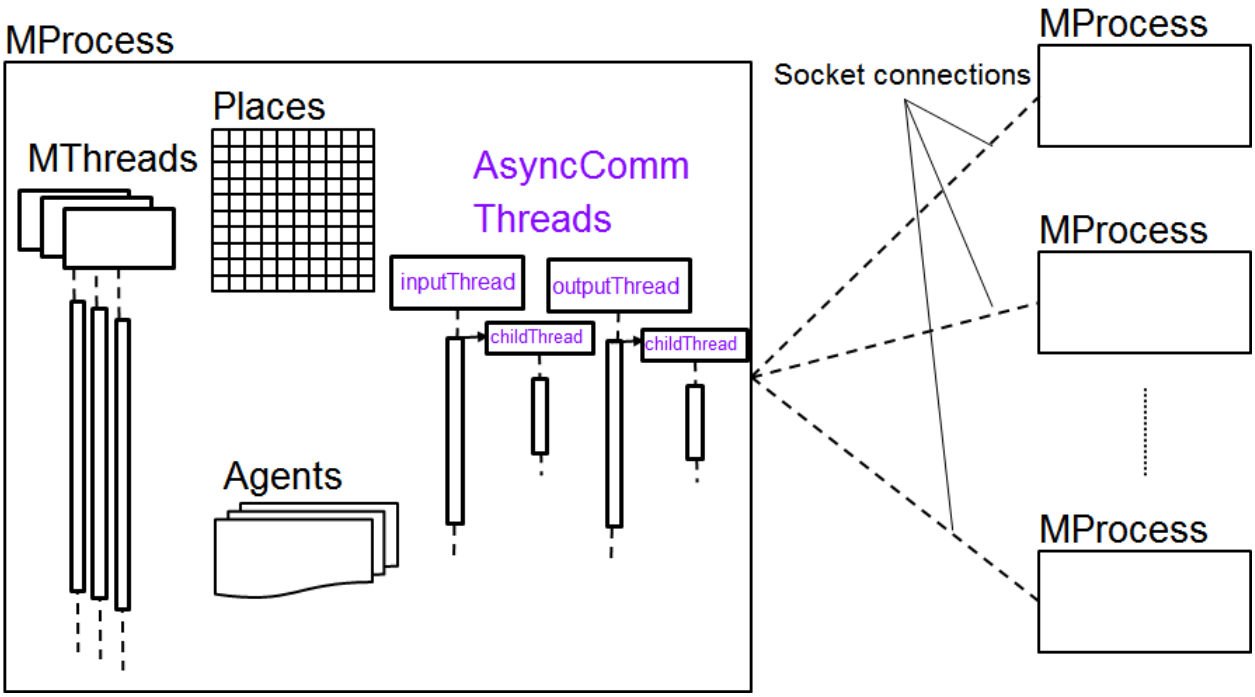


Figure 3.1: `MASS`'s architecture

New entities are shown in purple

3.2 Asynchronous Migration

In order to achieve asynchronous and auto migration of agents, we propose changes and additional components to existing architecture as highlighted in purple in Figure 3.1. They are called `AsyncCommunicationThreads`, which includes `AsyncInputThread` and `AsyncOutput -Thread` class. Previously in synchronous execution, communication is scheduled to happen at the same time across all node. Hence, it is easy to use main thread directly to handle communication. However, in asynchronous migration, communication can happen at any time during execution, sometimes with multiple requests at the same node at once. Therefore, there need to be a different thread to handle this.

More properties and methods for asynchronous migration functionality are also added into existing classes as illustrated in Figure 3.2. `MASS_base` maintains two variables: `AsyncOutput -Thread` and `AsyncInputThread` to handle requests and responses from other nodes. `Agents_base` now has an `asyncQueue` to hold agents that need to be executed by `MThreads`. Agents are dequeued and executed by `MThreads` until there is no agents left on the queue. `Agents` class has a new method `callAllAsync()`. Programmers who want to use asynchronous and auto

migration functionality need to call this method instead of the existing `callAll()`. Beside the usual `args` parameter, programmers need to supply a list of functions that need to be executed by each agent. The method returns once all agents have finished executing the method list. The return value is a list of copied instances of the agents which contains the results that they collect during execution. In `Agent` class, each agent maintains its own `asyncArgument`, `asyncFuncList`, and `asyncResult`. The names are self-explanatory. `Agent` class also has `myAsyncOriginalPid`, `myOriginalAsyncIndex`, `myCurrentIndex` property to keep track of its original position at the beginning of `callAllAsync()` execution. Because agents can migrate, spawn, and kill anytime during `callAllAsync()` execution, their final location at the end of execution can be different from their original position. These properties help avoid confusion and help programmers make sense of the results that the agents return at the end of execution. New methods `killAsync()`, `migrateAsync()`, and `spawnAsync()` need to be called in `Agent`'s subclass instead of the original `kill()`, `migrate()`, and `spawn()`.

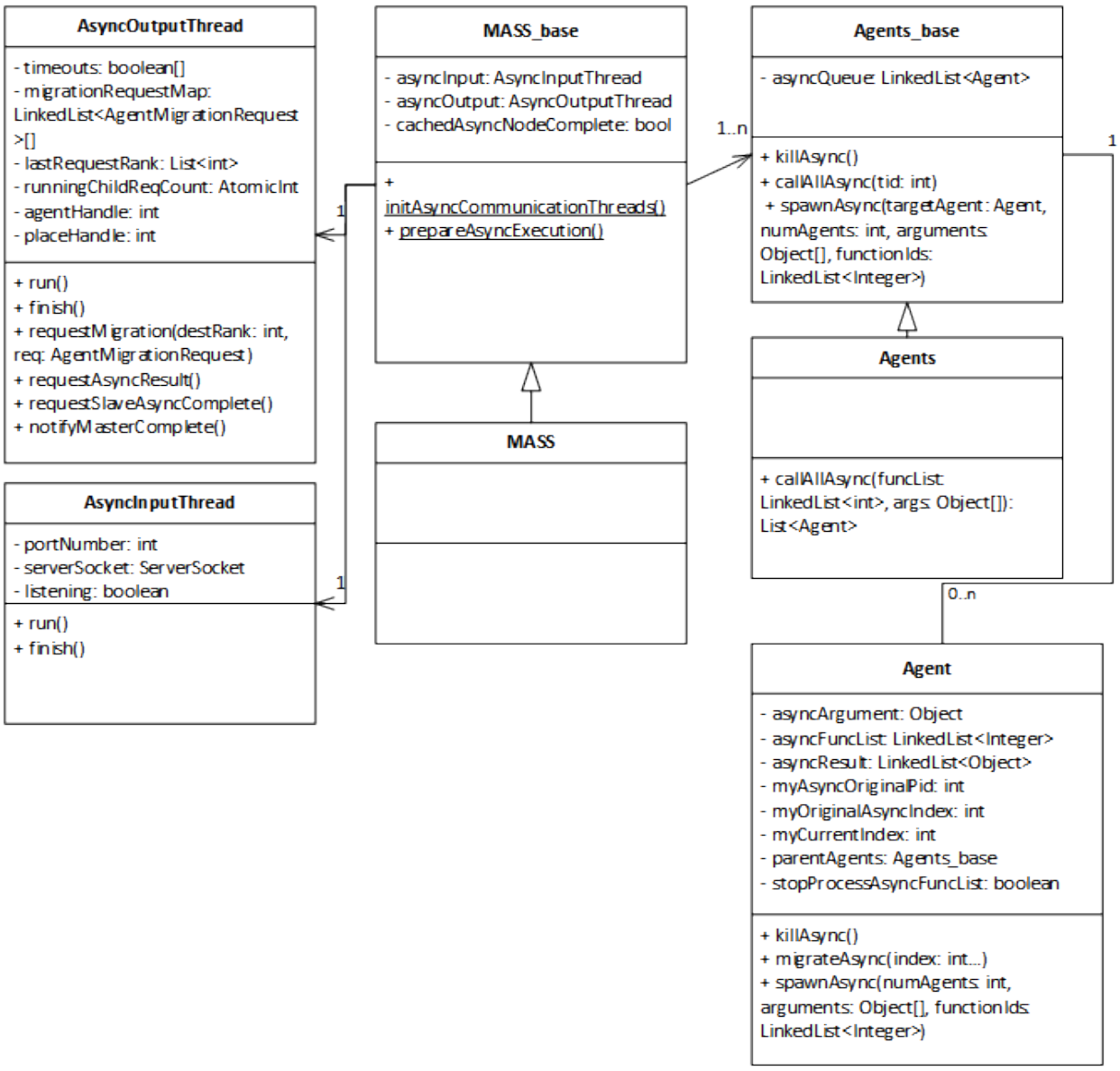


Figure 3.2: Class diagram for asynchronous functionality

Figure 3.3 shows the activity diagram of `callAllAsync()` execution. First, the master resets properties of `MASS_base`, `Agents_base` object, and `Agent` objects discussed previously in Figure 3.2. The master node then calculates the distribution of the `args` parameter among the existing agents. Because the master node has knowledge of agent population at each computing node, it knows which parameter needs to be sent to which node to be

distributed to the correct agent. It then construct `callAllAsync` messages to send to each slave node and add the arguments for each node to the corresponding message.

After distributing the messages, the master's threads will start executing `Agents_base.callAllAsync()`. Because agents can now migrate, spawn, and kill asynchronously, even when a node's threads finish executing `callAllAsync()`, there still may be more agents migrating from other nodes, that need to be executed. Therefore we need a Distributed Termination Detection algorithm, which is discussed in Section 3.5, to determine whether to continue executing `Agentsbase.callAllAsync()` or start collecting result. Once the algorithm decides that all execution has finished, the master issues result requests to its slaves and return the result collection to the caller.

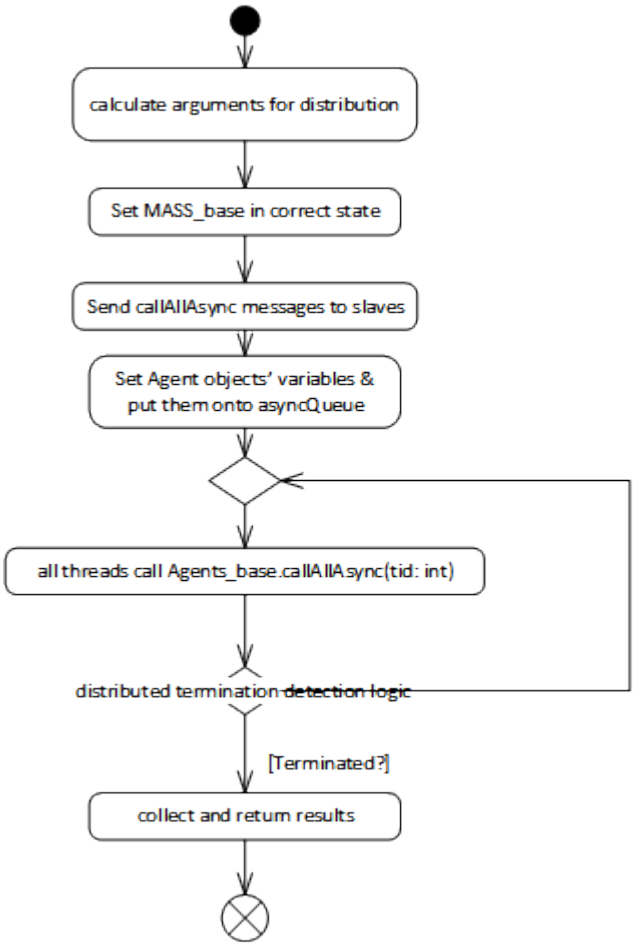


Figure 3.3: `callAllAsync()` activity diagram

Activity `Agents_base.callAllAsync()` in Figure 3.3 is broken down into detailed ac-

tivity diagram in Figure 3.4. Each thread in each computing node dequeues agents from `asyncQueue` and execute their `asyncFunctionList`s until there is no more agent to dequeue. While executing functions in an agent's `asyncFunctionList`, if the function is to kill that agent, the thread will stop executing its functions and dequeue the next agent, if there is still one in the `asyncQueue`. If the function is to migrate the agent, depend on whether the migration is locally at the same node or a remote migration, the thread will enqueue the agent back into the `asyncQueue` or create a remote migration request and pass it to `asyncOutputThread`. If the agent has no more functions to be executed, it will be cloned and enqueued into `completeQueue` for result gathering later.

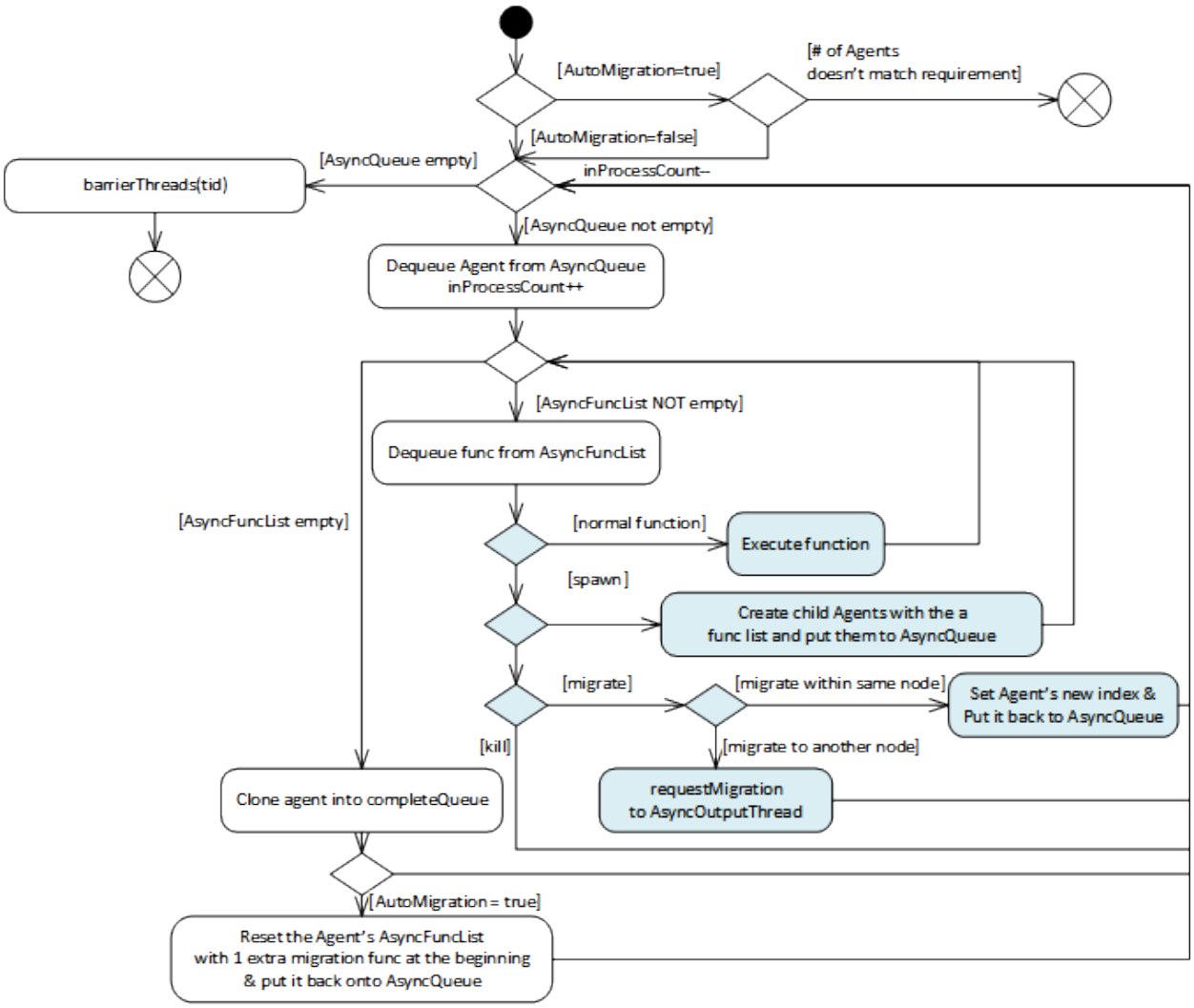


Figure 3.4: `Agents_base`'s `callAllAsync()` activity diagram

3.3 Auto Migration

Auto Migration works are scheduled in Spring quarter.

3.4 Communication Channel

Currently MASS programming model uses Socket API to handle communication. The implementation can only handle synchronous communication because it uses `ServerSocket.accept()`, which is a blocking function call, on the main thread. Hence, the function can only be used

when a computing node is expecting a communication request from another one. Asynchronous communication cannot use this existing infrastructure as requests can be sent between any two nodes at any time during execution. Instead, a completely new component, `AsyncCommunicationThreads`, which handles asynchronous communication between computing nodes, is implemented.

There are two alternatives in implementing `AsyncCommunicationThreads`, using either HTTP protocol or Socket API. The advantages and disadvantages of these two approaches are shown in Table 3.1. In order of importance,

Effort The alternative should be easy for end-users, in this case, programmers who consume MASS library, to use. HTTP protocol scores low in this criteria because it requires each computing node to be set up as a server in order to receive requests. This may also involve configuring the system firewall. On the other hand, Socket API does not require these extra setups.

Flexibility Socket API has lower programming level comparing to HTTP protocol. Not all communication in asynchronous migration is in the form of sending a request and waiting for a response. For example, when a slave node notifies master node that it is idle, it does not need to wait for a response from the master. HTTP protocol does not allow this while Socket API does. With Socket API, you can also choose when to close a connection and connection is duplex. Programmers can send input and output data as many times as you want in one Sokcet connection.

Programmability . Both HTTP protocol and Socket API satisfy this criteria as there are existing libraries in C++ and Java that support them.

Because of Socket API's advantage in Effort and Flexibility criteria, it is a clear choice for implementing `AsyncCommunicationThreads`

Table 3.1: Comparison of alternatives for `AsyncCommunicationThreads`

Criteria	**HTTP Protocol**	**Socket API**
Effort	Low†	High
Flexiblity	Low	High
Programmability	High	High

†Low/Medium/High where High is the best suitable for the criteria

In order to accommodate asynchronous requirement, in which many requests can be sent or received at the same time, as well as maximize resource usage, the socket will be implemented in a non-ordering manner. `AsyncCommunicationThreads` consists of two entities:

`AsyncInputThread` and `AsyncOutputThread`. `AsyncInputThread` handle requests received by the `MProcess`, while `AsyncOutputThread` is used to send requests to other `MProcess`. When `AsyncInputThread` receives a request, it spawns a *childThread* to and assign it to handle that request. `AsyncInputThread` itself return immediately to listen for more requests. The same logic is applied for `AsyncOutputThread`. This ensures that simultaneous requests can be handled and none will be dropped. Note that requests are handled in non-FIFO manner in this logic, even though Socket API communcation is FIFO, because *childThread*s are not guaranteed to run in the order that they are created.

3.5 Distributed Termination Detection

In synchronous MASS implementation, each `callAll()` results in each computing node executing exactly one method. Thus, master node knows when to collect the result of `callAll()`. In asynchronous implementation, however, `callAllAsync()` results in asynchronous execution, spawning, migrating and killing of agents many times in one function call. The number of agents at the beginning of the function call may not be the same at the end. The agent can end up at a different computing node when execution finishes. Computing node, that finishes executing its agents, can become active again because of other agents migrating to it. Hence, the master node needs an efficient logic to determine when all computation has been completed so it can collect results and return to caller.

Before describing the algorithm here are some obvious facts:

- Each computing node can only be in two states: *active* or *idle*. A node is active if it has agents with functions to execute or sent/received requests to process, otherwise it is *idle*.

- Agents come to existence at a node in two ways: either they are there at the beginning of `callAllAsync()` function call or they are migrated from another node during execution.

These facts lead to following definitions:

Definition 1. *If node A changes from idle state to active state because of agents migrating to it from node B then B is called the originator of A and A is a receiver of B.*

This definition leads to following observations:

- A node can have only one *originator* yet many *receivers*.

- A node, that is idle at the beginning of `callAllAsync()`, has no *originator* until it becomes *active* upon arrival of migrating agents from another node.

- A node can also have different *originators* at different time during `callAllAsync()` execution because it can become *idle* and *active* many times, each time because of migrating agents from a different node.

Definition 2. *A node with at least one agent at the beginning of* `callAllAsync()` *is a master node's receiver and has master node as its originator.*

In implementation, a computing node use `originatorPid` to store its originator's `Pid` and `receiverList` to store its receivers' `Pids`.

The definition of *active* and *idle* are formally redefined as follows.

Definition 3. *A node is active if it has agents with functions to execute or sent/received requests to process, and its* `receiverList` *is empty. Otherwise, it is idle.*

Based on these definitions, the distributed termination detection algorithm is outlined in Figure 3.5, 3.6, and 3.7. Figure 3.5 shows what need to happen at master node and slave nodes, respectively, at the beginning of `callAllAsync()` in order to uphold Definition 2. Because the master node has knowledge of each slave node's local agent population, it add nodes' `Pids` to its `receiverList` if they have agents at the beginning of execution. Likewise, each slave node with initial local agents set its `originatorPid` to 0, the master node's `Pid`.

for all slave node i **do**
  if i has local agents **then**
    add i to $receiverList$
  end if
end for

(a) at master node

if has local agents at beginning **then**
  $originatorPid \leftarrow 0$
end if

(b) at slave node

Figure 3.5: Initialization of distributed termination detection algorithm

Figure 3.6 shows the logic when remote migrations happen. If the receiver of a remote migration was in *idle* state and becomes *active* after receiving it, the receiver will set its `originatorPid` to the `Pid` of the sending node and include this information in the `ACK`, that is sent back to the sender. On the other side, if the `ACK` indicates that the sender has become an *originator*, it will add the receiving node's `Pid` to its `receiverList`.

When a computing node transitions from *active* state to *idle* state (based on Definition 3), it will send an *idle notification* to its *originator*, if it has one. On the other side, when a computing node receives an *idle notification* from its receiver, it removes that receiver's `Pid`

process migration request
if transition from *idle* to *active* **then**
  $originatorPid \leftarrow senderPid$
  indicate sender is *originator* in ACK
end if

(a) at receiving node

process ACK
if chosen as *originator* **then**
  add receiver's Pid to $receiverList$
end if

(b) at sending node

Figure 3.6: Distributed termination detection algorithm handling of remote migrations

from its `receiverList`. If its `receiverList` becomes empty, the *originator* becomes *idle*, and it will send `idle notification` to its own *originator*, if it has one. When the master node becomes *idle*, it will collect results from all slaves and pass them back to the caller. This logic is shown in Figure 3.7.

if no agent and request to process && $receiverList$ is empty **then**
  send $idleNotification$ to *originator*
end if

(a) receiver role

Require: on received $idleNotification$
remove sender's Pid from $receiverList$
if $receiverList$ is empty **then**
  if master node **then**
    collect async results from slaves
  else
    send $idleNotification$ to *originator*
  end if
end if

(b) originator role

Figure 3.7: Distributed termination detection algorithm's main function

Proof. The Distributed Termination Detection Algorithm correctness is proven as follows.

Suppose the algorithm is incorrect. That means when node 0, the master node, becomes *idle* and starts issuing `AsyncResultRequest`s to collect results, there exists at least one node which is still in *active* state.

If the node is still *active*, it has not sent *idle notification* to its *originator*, based on Algorithm 3.7a. If its *originator* is the master node, then master node's `receiverList` is not empty. Therefore master node is not *idle* and does not issue `AsyncResultRequest`s

(CONTRADICTION). If its *originator* is another slave node, applying the same reasoning recursively on that node's own *originator* until the *originator* is the master node, we will come to the same contradiction as above. *Originator* eventually will be 0 because the agent or agents that make a node *active* either exist from the beginning of `callAllAsync()` execution, or are spawned during execution. In both cases, they or their parent agents, must exist at a certain node at the beginning, and, based on Algorithm 3.5b, that node has 0 as its *originator*. □

Let us demonstrate the algorithm with an example illustrated in Figure 3.8. A distributed system has four computing nodes. There are four `MASS` agents at the beginning of `callAllAsync()` execution. The agents are numbered from 1 to 4. Agent 1 resides at node 0. Agent 2 resides at node 1. Agent 3 and 4 reside at node 2, while there is no agent at node 3. From Algorithm 3.5b, node 1 and 2 set it `originatorPid` to 0, which is master node's `Pid`, while master node adds 1 and 2 to its `receiverList`. Neither node 3's `originatorPid` nor `receiverList` are set because it is *idle* at this moment.

During the course of execution, agent 4 migrates from node 2 to node 3. This makes node 3 transition from *idle* to *active*. According to Algorithm 3.6a, node 3 considers node 2 as its *originator* and sets its `originatorId` to 2. Based on `ACK` response, node 2 add 3 to its `receiverId` accordingly.

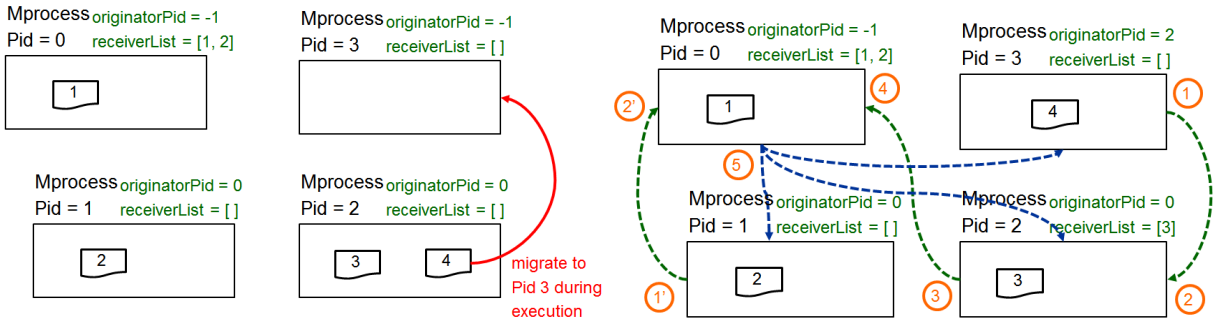
When all agents' functions are executed on all nodes, based on Definition 3, node 1 and 3 become *idle* because their `receiverList`s are empty. They will send *idle notification* to their respective *originator*s, which are node 0 and 2. Node 0 and 2 will remove 1 and 3 from their `receiverList`s. After this step, node 2's `receiverList` becomes empty therefore it is considered *idle*. Hence, it in turn sends *idle notification* to its *originator* 0. After removing 2 from its `receiverList`, node 0 becomes textitidle so it can now issue `AyncResultRequest` to collect the execution results.

3.6 Verification

The goals of verification are as following:

Code correctness In deterministic application, the new approach should produce the same result as the previous approach or sequential execution. This can be done by comparing result from both approaches on classic deterministic parallel problems such as Mandelbrot. If the result are the same then the code is correct.

Performance improvement The new approach should show at least a 5-percent performance improvement in terms of data analysis execution time comparing to traditional



(a) Beginning of execution

(b) End of execution

1. Pid 3 sends *idle notification* to Pid 2 — **2.** Pid 2 remove 3 from its `receiverList` — **1'.** Pid 1 sends *idle notification* to Pid 0 — **2'.** Pid 0 remove 3 from its `receiverList` — **3.** Pid 2's `receiverList` is empty so it sends *idle notification* to Pid 0 — **4.** Pid 0 remove 2 from its `receiverList` — **5.** Pid 0's `receiverList` is empty so it collects results from slaves

Figure 3.8: Example of Distributed Termination Algorithm

approach. Again, this can be done by applying the two approaches to parallel problems such as Climate Change Simulation. Data analysis performance is measure in terms of execution time under various configuration. Execution time is measured in milliseconds (ms). Execution configuration includes number of computing nodes and number of threads per node when executing an application.

Usability The new approach should have enough documentation and method signatures should be easy to understand and convenient to use for MASS library users. Interview will be given out to existing users of MASS library at UW Bothell to measure the new codes usability. This should include questions that allow user to rate their experience working with the new functionality, how often do they choose the new approach over the synchronous one, how hard is it for them to switch their existing code to the new function and provide other verbose feedback as well.

Scalability The new approach should be able to handle hundreds of thousands agents and places per node. Execution configuration with high number of agents will used to verify this.

Based on these goals, we designed a set of test cases, performed the experiments, and evaluated the results. This process will be discussed in greater detail in Experiment and Result Evaluation chapter.

Chapter 4

EXPERIMENTS

At this moment, experiments are still being conducted to evaluate the performance of asynchronous migration. Their goal is to evaluate the performance of asynchronous and auto-migration migration against synchronous migration under various configurations: different number of computing nodes, different number of threads, and number of agents per node. For each configuration, execution for both asynchronous and synchronous approach are performed three times and the results are averaged.

4.1 Setup

Experiments are conducted using machine grid available at Linux lab at University of Washington Bothell. There are 16 machines total. Each has an Intel Core i7-3770 CPU at 3.40GHz speed and 16GB of RAM.

The problem chosen for the experiment is the Mandelbrot Set. Points in 2-D space are colored based on a certain algorithm. For this experiment, we use *Escape time* algorithm. The algorithm performs repeating calculation on each point. Based on the point's x- and y-coordinate, the calculation will have different number of iterations. The number of iterations determines the point's color. *Escape time* algorithm is summarized in Figure 4.1.

$x_0 \leftarrow 0$
$y_0 \leftarrow 0$
while $x * x + y * y < 4.0$ **and** $iteration < MAX\_ITERATION$ **do**
  $double\ xtemp = x * x + y * y + x_0$
  $y \leftarrow 2 * x * y + y_0$
  $x \leftarrow xtemp$
  $iteration++$
end while

Figure 4.1: *Escape time* algorithm

All the experiments are conducted on a 2-D space of size 4032 points by 4032 points. Each point corresponds to one place. A place's index corresponds to a point's coordinate.

This space size is chosen so places can be distributed evenly accross computing nodes and among the threads at each node, which can reach maximum of 4. Agents are distributed at leftmost place at each row. They migrate and perform calculation at each place until the end of the row. If there are more agents than rows, each row will be divided into equal smaller chunks so that each agent can have one chunk. They will start from the leftmost of the chunk, migrate and perform calculation until the end of their chunk.

The reason Mandelbrot Set problem is chosen is because the number of iterations, hence execution time, varries among points. This simulates the condition in which if agents are distributed to perform calculation of number of iterations for each point, some will finish first and remain idle. Solving the problem using asynchronous and auto migration approach as well as synchronous migration approach and comparing the result will prove whether asynchronous migration can utilize this idle time.

4.2 Effect of number of agents

The first experiment is to evaluate whether asynchronous approach can perform well when the number of agents increase. We compare performance of asynchronous and auto migration when there are 4032, 112896, and 903168 agents. These numbers are chosen because agents are distributed evenly among rows of places with this number. The number of threads per node is fixed at 4. The number of computing node is 1 and 8. Average results are plotted in Figure 4.2. The exact performance numbers are shown in Table A.2 and A.3 in Appendix A.

The optimal execution number is deduced by dividing sequential performance, which is 2521 seconds and is summarized in Table A.1, by total number of threads. We observe the followings from Figure **??**.

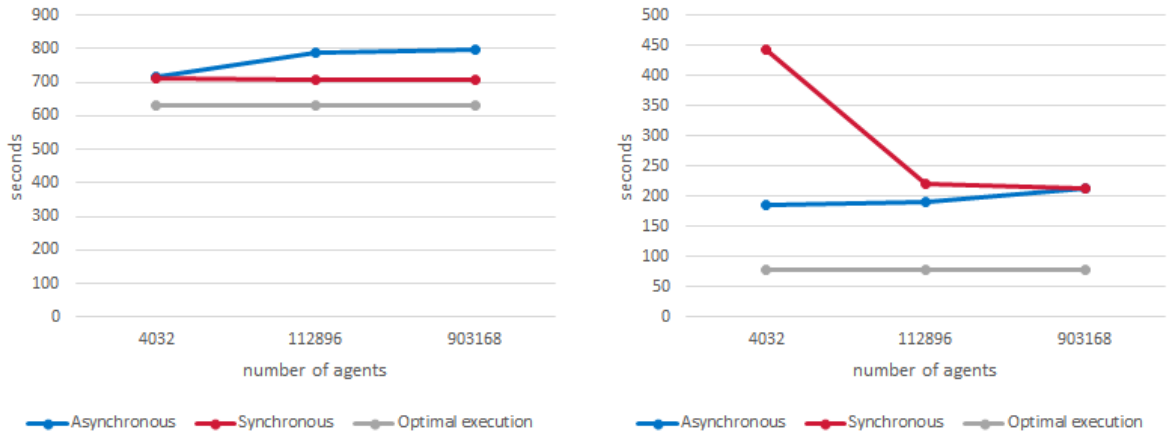
- Performance decreases for asynchronous execution and increases for synchronous when number of agents increase. This happens for both 1-node and 8-node configurations.

- Asynchronous migration performs worse than synchronous migration in 1-node configuration when there are large number of agents but 58% faster than synchronous approach in 8-node configuration.

- None of the approaches performs as good as the optimal.

This can be explained as follows:

- When there are more agents, synchronous execution needs to issue less number of `callAll()` and `manageAll()` so its communication overhead is reduce and performance



(a) 1-node configuration (b) 8-node configuration

Figure 4.2: Effect of agent size on performance

improve. In the case of 1-node configuration, it is even better because there is no inter-node communication, only synchronization among local threads. Asynchronous approach, on the other hand, has the size of `asyncQueue` increase. Dequeuing agents from `asyncQueue` needs to be synchronized among threads. Therefore, its overhead actually increase when the number of agent increases.

- Each approach suffers from different kind of overhead so none performs as good as the optimal.

The experiment also shows that asynchronous approach can handle 16 million places and 900 thousands agents at one node.

4.3 Others

Other experiments are still going on and scheduled to finish in the next couple of weeks.

Chapter 5

CONCLUSION AND FUTURE WORKS

This report shows the progress of the project so far on designing and implementing asynchronous migration functionality. Some experiments have been done that show performance improvement of the new approach. The project is on track to finish by the end of Spring quarter 2015.

BIBLIOGRAPHY

[1] L. Gasser and K. Kakugawa, "Mace3j: fast flexible distributed simulation of large, large-grain multi-agent systems," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*. ACM, 2002, pp. 745–752.

[2] T. Chuang and M. Fukuda, "A parallel multi-agent spatial simulation environment for cluster systems," in *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*. IEEE, 2013, pp. 143–150.

[3] M.-W. Jang and G. Agha, "Agent framework services to reduce agent communication overhead in large-scale agent-based simulations," *Simulation Modelling Practice and Theory*, vol. 14, no. 6, pp. 679–694, 2006.

[4] A. González-Pardo, P. Varona, D. Camacho, and F. d. B. R. Ortiz, "Optimal message interchange in a self-organizing multi-agent system," in *Intelligent Distributed Computing IV*. Springer, 2010, pp. 131–141.

[5] C. L. Wasous, "Distributed agent management in a parallel simulation and analysis environment," Ph.D. dissertation, University of Washington, 2014.

[6] E. W. Dijkstra, W. H. Feijen, and A. M. Van Gasteren, "Derivation of a termination detection algorithm for distributed computations," in *Control Flow and Data Flow: concepts of distributed programming*. Springer, 1986, pp. 507–512.

[7] R. W. Topor, "Termination detection for distributed computations," *Information Processing Letters*, vol. 18, no. 1, pp. 33–36, 1984.

[8] F. Mattern, "Asynchronous distributed terminationparallel and symmetric solutions with echo algorithms," *Algorithmica*, vol. 5, no. 1-4, pp. 325–340, 1990.

[9] T.-H. Lai, "Termination detection for dynamically distributed systems with non-first-in-first-out communication," *Journal of Parallel and Distributed computing*, vol. 3, no. 4, pp. 577–599, 1986.

[10] R. F. DeMara, Y. Tseng, and A. Ejnioui, "Tiered algorithm for distributed process quiescence and termination detection," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 11, pp. 1529–1538, 2007.

[11] D. M. Dhamdhere, S. R. Iyer, and E. K. K. Reddy, "Distributed termination detection for dynamic systems," *Parallel computing*, vol. 22, no. 14, pp. 2025–2045, 1997.

[12] J. Nagle, "Rfc 896 - congestion control in ip/tcp internetworks," 1984, accessed: 2015-01-25. [Online]. Available: http://tools.ietf.org/html/rfc896

LIST OF FIGURES

Figure Number Page

3.1 `MASS`'s architecture 7
3.2 Class diagram for asynchronous functionality 9
3.3 `callAllAsync()` activity diagram 10
3.4 `Agents_base`'s `callAllAsync()` activity diagram 12
3.5 Initialization of distributed termination detection algorithm 15
3.6 Distributed termination detection algorithm handling of remote migrations . 16
3.7 Distributed termination detection algorithm's main function 16
3.8 Example of Distributed Termination Algorithm 18
4.1 *Escape time* algorithm 20
4.2 Effect of agent size on performance 22

LIST OF TABLES

Table Number Page

3.1 Comparison of alternatives for `AsyncCommunicationThreads` 13

A.1 Sequential execution 29

A.2 Performance of Asynchronous migration with various agent size 29

A.3 Performance of Synchronous migration with various agent size 30

GLOSSARY

PID: : Process ID. A unique non-negative integer value assigned to each computing node. Master node's Pid is always 0.

Appendix A

EXPERIMENT RESULT

Table A.1: Sequential execution

Execution (seconds)			**Average**
#1	**#2**	**#3**	
2521			

Table A.2: Performance of Asynchronous migration with various agent size

# of nodes	**Threads per node**	**Agent size**	**Execution (seconds)**			**Average**
			#1	**#2**	**#3**	
1	4	4032	722	715	715	717
1	4	112896	769	809	793	790
1	4	903168	757	815	825	799
8	4	4032	188	185	185	186
8	4	112896	187	201	184	191
8	4	903168	207	214	216	212

Table A.3: Performance of Synchronous migration with various agent size

# of nodes	Threads per node	Agent size	Execution (seconds)			Average
			#1	#2	#3	
1	4	4032	710	710	716	712
1	4	112896	708	707	707	707
1	4	903168	707	708	708	707
8	4	4032	425	502	406	444
8	4	112896	209	226	228	221
8	4	903168	226	216	195	212