Asynchronous and Automatic Agent Migration in MASS

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

Spring 2015

Project Committee:

Professor Munehiro Fukuda, Project Advisor

Professor David Socha, CSS595/596 Instructor

University of Washington Bothell

Abstract

Asynchronous and Automatic Agent Migration in MASS

Hung H. Ho

MASS is a parallelizing library that provides umlti-agent and spatial simulation over a cluster of computing nodes. The existing approach in MASS uses `callAll()` broadcast messages to execute a function at each `Agent` object. It also uses `manageAll()` to initiate spawning, migration, and termination of `Agent` objects. The goal of this project is to minimize this communication overhead by only broadcast mesages if need to, hence improve execution time and efficiency of MASS. We designed and implemented asynchronous and automatic migration (or auto migration in shorthand) functionality for agents of MASS library to achieve this goal. The same execution result is achieved by issuing a single `callAllAsync()` instead of multiple `callAll()` and `manageAll()` messages. This project also includes designing and implementing a distributed termination detection algorithm using the termination notification approach to detect when asynchronous agent execution has completed across all computing nodes. Experiments on the Mandelbrot Set problem show that the new approach achieves better performance than the current MASS synchronous approach when there are a few thousand agent migrations during execution. Additionally, its performance is similar to MPI Java under all computing node and thread configurations chosen in these experiments. Future work includes performing more experiments on asynchronous and auto migration functionality using real-life applications, such as Climate Change Simulation and Network Motif, and extends these functionalities to the C++ version of MASS. More user studies are needed to understand the usability of this new asynchronous and auto migration functionality of agents.

TABLE OF CONTENTS

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 in some cases is difficult using only mathematical methods. 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. Both are distributed among computing nodes. `Places` stay the same throughout the simulation, while `Agents` perform computation, migrate, and spawn among `Places` in each simulation round. 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 shows improved performance in certain cases and utilization of computing nodes and cores[2], some performance issues still remain. First of all, because agent execution and migration is done synchronously, in parallel applications, in which agents' workload cannot be equally divided among them, agents with lighter workload, which finish their assignments earlier in a period, will be idle while waiting for others to finish before proceeding to the next instruction. Additionally, agents have to be disseminated explicitly through broadcast messages rather than automatically by themselves. Issuing broadcast messages has communication overhead, so it should be done only when it is necessary. In the current approach, it is done more that needed as agents can migrate, spawn, and kill by themselves. Hence the current approach is inefficient, and slows down the simulation.

In this project, we address these issues by minimizing communication overhead through asynchronous migration and automatic migration (or auto migration, in shorthand) of agents. Hence, unless a coordination/synchronization among agents is needed, agents that finish their assignments can immediately migrate to other places and spawn new agents without having to wait for migration broadcast messages and synchronization with other agents.

1.2 Goals

The goal of the project is to implement both asynchronous migration and auto migration functionality of agents for the Java version of MASS library. Asynchronous migration and auto migration are implemented together because they are closely related. Both target improving MASS library performance by reducing agent migration communication overhead. `manageAll()` function call is eliminated and agents can now migrate asynchronously on their own. Auto migration reduces the number of remote migrations, which slow down execution. Additionally, asynchronous and auto migration help improve usability of MASS library, in particular, programmability, for programmers. First of all, 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 tasks at each place. For the Master Capstone Project and a 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 Set, and real-life applications, such as Biological Network Motif search and Climate Simulation; and evaluating their performance against existing functionalities of MASS library.

1.3 Report Structure

The project report is organized into the following chapters. In chapter 2, we look into existing related work. Chapter 3 discusses the design, implementation of the new functionalities, and how to verify them. Performance evaluation of asynchronous and auto migration as well as discussion of the results is presented in Chapter 4. Chapter 5, Conclusion and Future Work, summarizes the achievements and shortcomings of the project and layout possible improvements.

Chapter 2

LITERATURE REVIEW

2.1 Agent-based Parallel Computing

In any multi-agent distributed simulation system, communication is a significant overhead factor that drags down the performance of the system. 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. This is because 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.

Considerable research has been conducted to improve performance by reducing overhead in certain types of communication. Jang and Agha [3] implemented a framework which includes 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. González-Pardo et al. [4] also studies communication overhead among agents . 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 iterations.

Cherie Wasous [5] proposed two auto migration schemes 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 show little performance improvement but extensive performance studies have not been done.

In this project, rather than communication overhead between agents as in previous research, 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 a new auto migration algorithm, and perform a more extensive performance evaluation on the new auto migration algorithm.

2.2 Distributed Termination Detection

Because execution of each agent 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 in detail in the 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 a phase, or wave, in some algorithms, needs 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 the controller initiation category. The initiator or controller propagates the termination detection message to other nodes or processes in the form of rings [6], trees [7], or graphs [8][9]. A termination detection message is called a signal in [7] and [9], and a 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, 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 the 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 to, or all of its neighbors reply with a "ready-to-terminate" (RT) message, it will reply with a RT message. Global termination is achieved when all the nodes return RT messages. In [10], each process 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 certain nodes based on certain conditions. Global termination is achieved when the master node concludes that the nodes that it keeps track of have finished their execution.

Chapter 3

METHOD

3.1 MASS Programming Model

The existing architecture is represented by Figure 3.1. Each computing node has a process, called `MProcess`, running on it. Each `MProcess` contains multiple threads, called `MThreads`. `MThreads` execute concurrently and perform computations on groups of `Place` and `Agent` objects, that are distributed at that computing node. `MThreads` perform computation on agents when receiving `callAll()` messages and migrate, spawn, or kill agents when receiving `manageAll()` messages. 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. These threads are destroyed and not re-used after each migration phase.

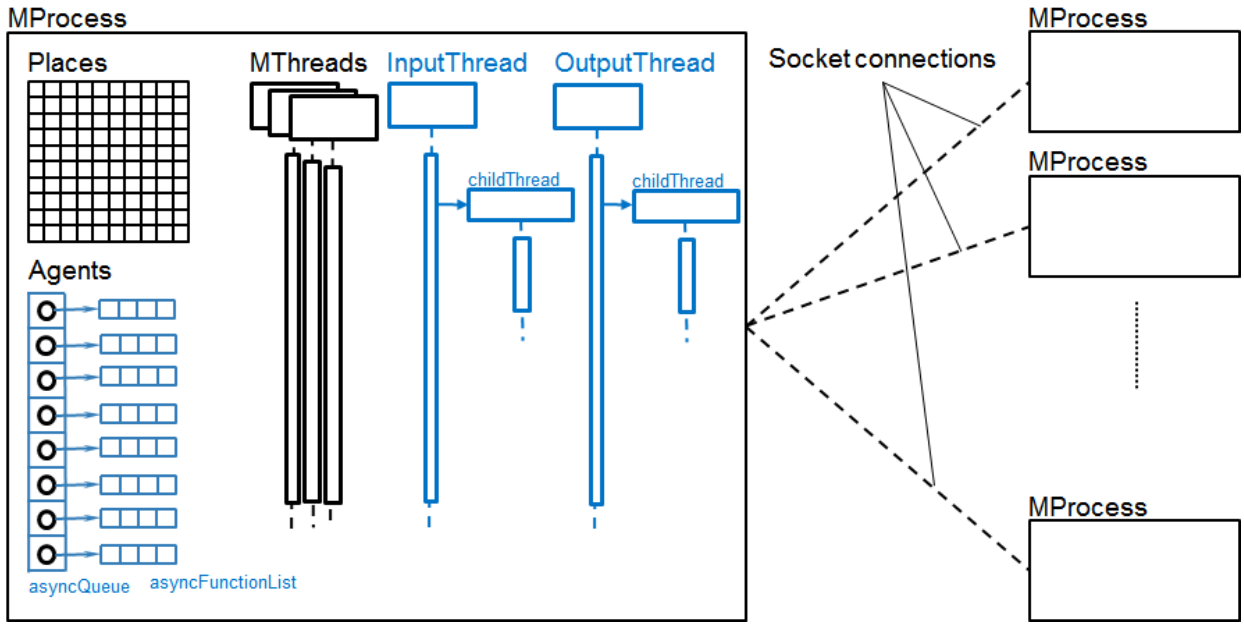


Figure 3.1: `MASS`'s architecture

New entities implemented in this project are shown in blue

MASS library employs a network of computing nodes, which communicate with one another using Socket API. 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` with 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(0`.
- `Agents` is a subclass of `Agents_base` with 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` with other properties and methods that are exclusive to master node

3.2 Asynchronous Migration

In order to achieve asynchronous and auto migration of agents, we implemented changes and additional components to existing architecture as highlighted in blue in Figure 3.1. They are called `AsyncCommunicationThreads`, which includes `InputThread` and `OutputThread` class. Previously, in synchronous execution, migration communication is scheduled to happen at the same time across all node. Hence, it is easy to use the main thread directly to handle communication even though `ServerSocket.accept()` is a blocking function call. However, in asynchronous migration, migration requests can happen at any time during execution, sometimes with multiple requests to the same node at once, hence the need for dedicated threads 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: `AsyncOutputThread` 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 are 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()`. In

addition to the usual `args` parameter, programmers need to supply a list of functions that will 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. Execution results are stored in a dedicated variable in each `Agent` object. This approach is chosen to improve usability because, unlike synchronous `callAll()`, each agent may need to collect multiple results during executing a `callAllAsync()`. Using this approach, users only need to call `appendAsyncResult()` whenever they need to collect an executing result. They do not have to re-implement the collection logic every time they use asynchronous migration functionality.

In `Agent` class, each agent also maintains its own `asyncArgument`, `asyncFuncList`, and `asyncResult` property. The names are self-explanatory with regard to their functionalities in the class. `Agent` class also has `myAsyncOriginalPid`, `myOriginalAsyncIndex`, and `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 enable programmers to make sense of the results that the agents return at the end of execution. Additionally, new methods `killAsync()`, `migrateAsync()`, and `spawnAsync()` are implemented and need to be called in `Agent`'s subclass instead of the original `kill()`, `migrate()`, and `spawn()`.

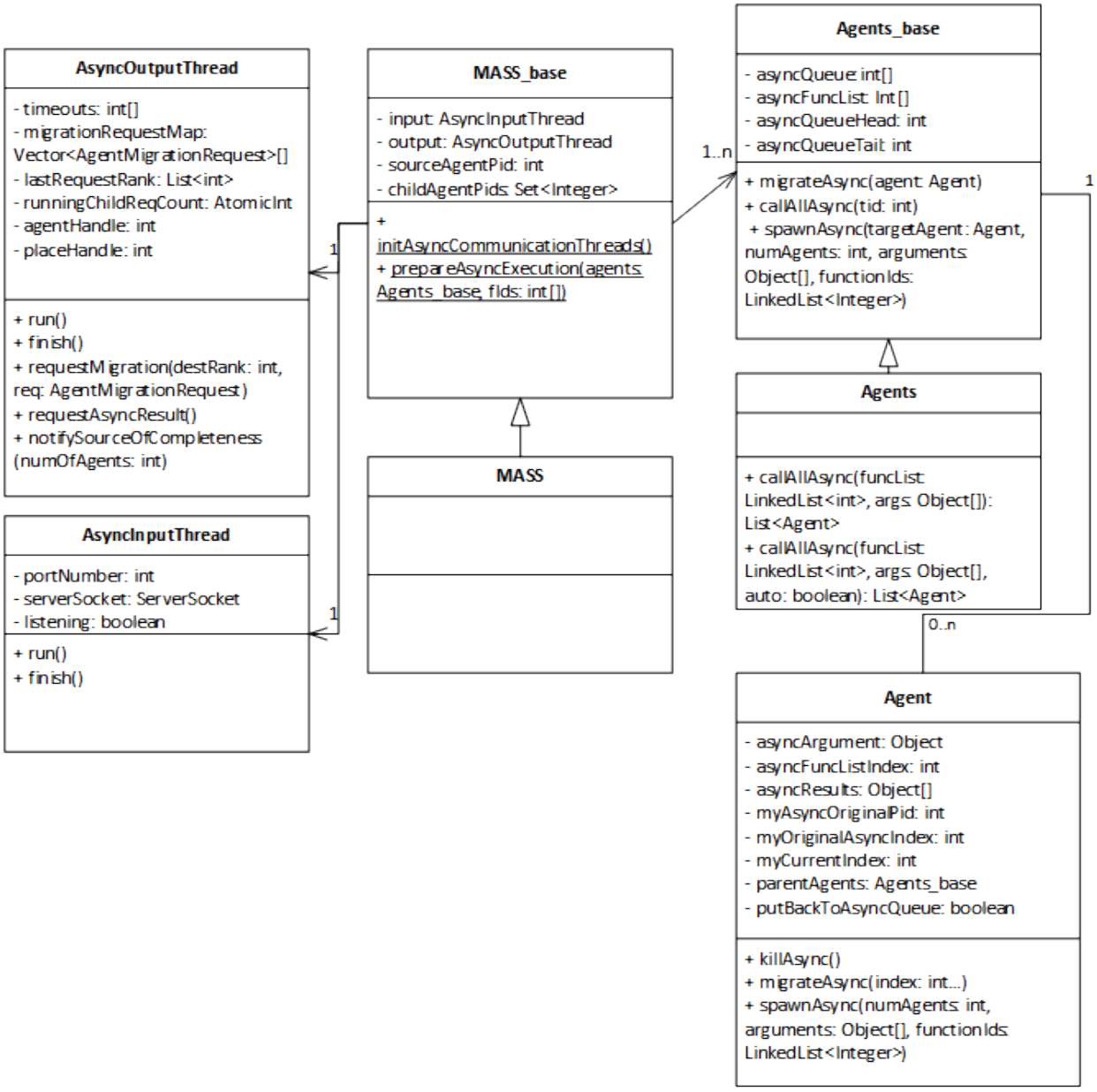


Figure 3.2: Class diagram for asynchronous functionality

Due to MASS library complexity, only new properties and functions implemented in this project are listed in this diagram.

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 the 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 constructs messages of type `AGENTS_CALL_ALL_ASYNC_RETURN_OBJECT` to send to each slave node and adds 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 more details in Section 3.6, to determine whether to continue executing `Agents_base.callAllAsync()` or start collecting results. Once the algorithm decides that all execution has finished, the master issues result requests to its slaves and returns the result collection to the caller.

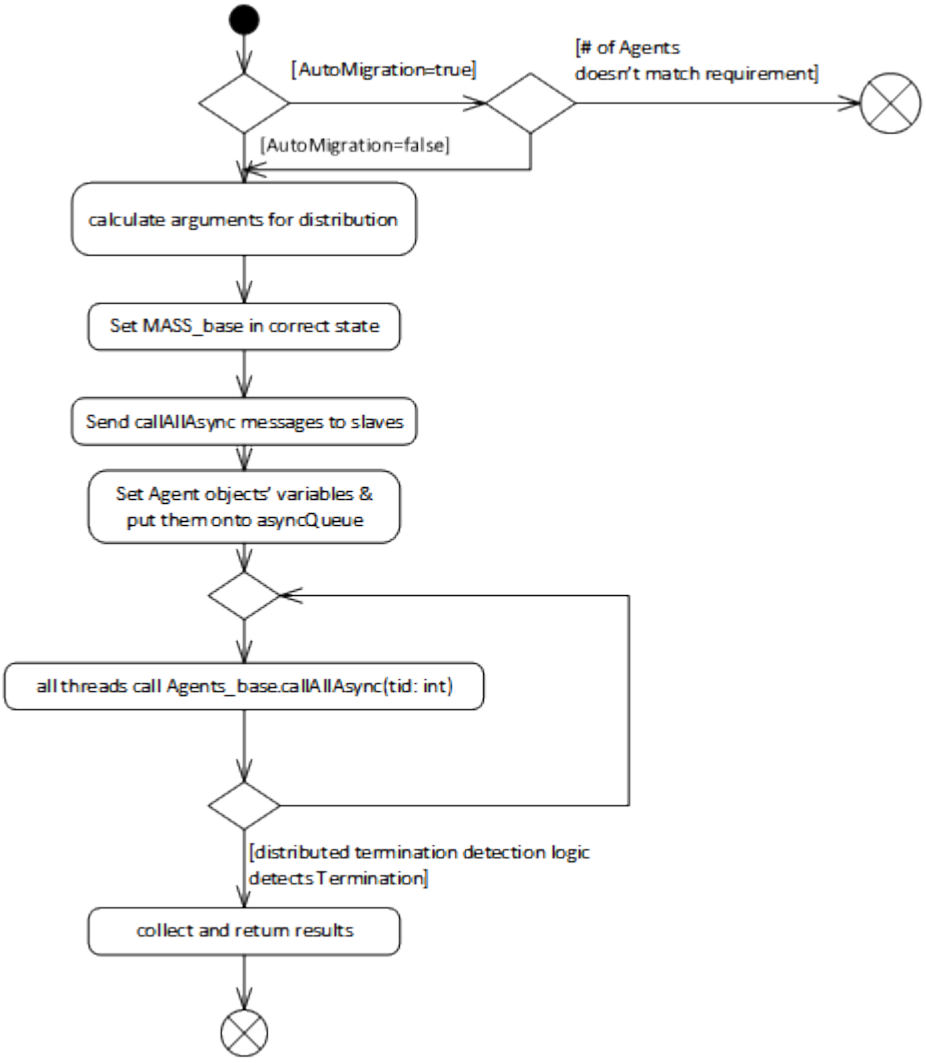


Figure 3.3: `callAllAsync()` activity diagram

Activity `Agents_base.callAllAsync()` in Figure 3.3 is broken down into detailed activity diagram in Figure 3.4. Each thread in each computing node dequeues agents from `asyncQueue` and executes their `asyncFunctionLists`, 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 or more left 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.

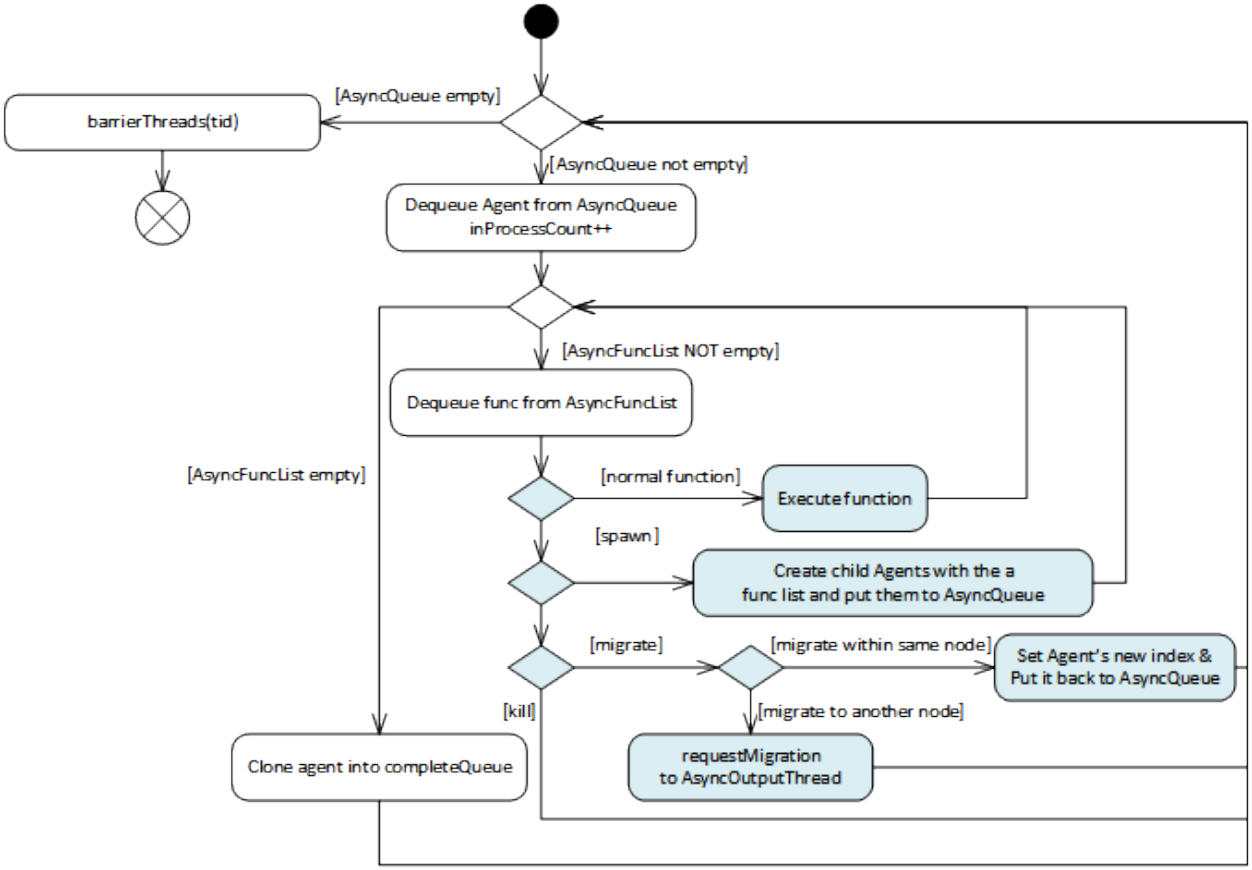
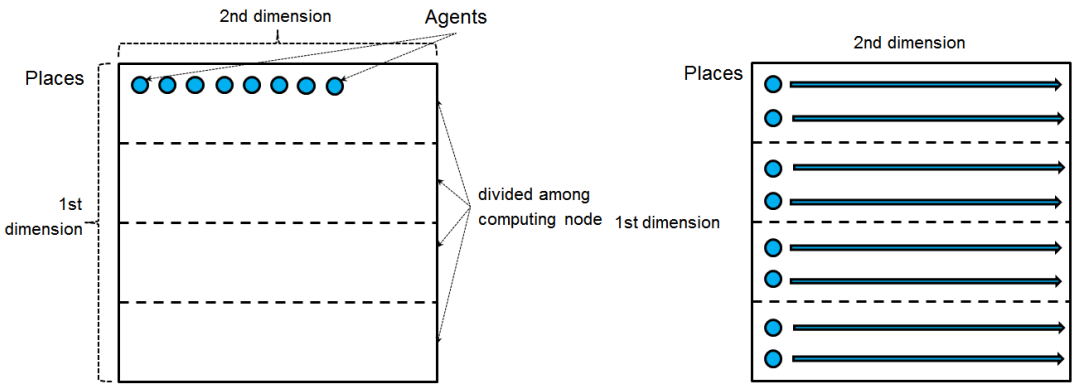


Figure 3.4: `Agents_base`'s `callAllAsync()` activity diagram

3.3 Auto Migration

In addition to implementing the asynchronous migration feature, the project also implements agent auto migration feature. While asynchronous migration improves MASS performance, auto migration aims at improving programmability for users. Instead of having to determine how to migrate agents in a pattern that has the best performance by themselves in a data analysis job, programmers only need to specify the list of functions that need to be executed at a place and MASS will do the remaining work. Since remote migration is very costly, any chosen agent migration pattern needs to minimize the number of remote migrations. As places are divided by their first dimension, in a data analysis problem, agents should migrate along the last dimension of `Places` from the lowest index to the highest index in order to

minimize the number of remote migrations. This pattern is illustrated in Figure 3.5. When the number of agents is the same as the number of rows, the rows can be easily divided among the agents. The number of remote migrations is also minimized as they only happen the very first time when agents line up at the lowest index. When the number of agents is not equal to the number of rows, deciding how to assign the rows to agents so that remote migration is minimized is complex. As this is the first implementation of auto migration, in this project we only support the case when the number of agents equal the number of rows. As each agent traverses from the lowest index to the highest one of the last dimension of `Places`, assuming `Places` have a dimension of p_1 x p_2 x .. x p_n, the number of agents at the beginning of an auto migration call must equal p_1 x p_2 x .. x $p_{(n-1)}$ in order to cover all the places. Behind the scene, MASS automatically generates the complete list of



(a) default agents' location after initialization

(b) agents traverse from the lowest index to the highest index of the last dimension

Figure 3.5: Migration pattern of auto migration

functions, including computation functions and migration functions, that would be called if it were an ordinary `callAllAsync()`. It then proceeds to `callAllAsync()` with the complete function list as usual. Because of this approach, the performance of auto migration and asynchronous migration should be similar with no significant difference. Before proceeding to `callAllAsync()`, MASS also verifies if the current number of agents equals p_1 x p_2 x .. x $p_{(n-1)}$. If this condition is not satisfied, MASS will simply return and no execution will be done. Because an auto migration call can be part of many function calls from the users, if MASS creates more agents or deletes agents so the condition is satisfied for one auto migration function call, it may affect what the users intend to do in subsequent calls. For example, if users do not have enough agents before invoking auto migration, and MASS automatically create more agents to satisfy the constraint, then users create more agents

afterward without knowing of this, then memory overflow may happen. As such, it is better to show users an error and let them decide what is best for their program.

3.4 Communication Channel

Currently, the 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 one 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 a lower programming level compared 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 the 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, programmers can also choose when to close a connection. Additionally, socket connection is duplex. Programmers can send input and output data as many times as you want in one Socket 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`

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

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

two entities: `AsyncInputThread` and `AsyncOutputThread`. `AsyncInputThread` handles requests received by the `MProcess`, while `AsyncOutputThread` is used to send requests to other `MProcess`. When `AsyncInputThread` receives a request, it spawns a *childThread* and assigns it to handle that request. `AsyncInputThread` itself returns 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 in which they are created.

3.5 Remote Migration Request Bundling

In traditional synchronous migration, because all migrations happen at the same time during `manageAll()` execution, remote migrations of the same source and destination are grouped together and executed in one single remote migration request. In the new asynchronous approach, when an agent needs to migrate remotely to another computing node, the source node has two options: it can either address the request immediately and send the migration request out immediately, or it can wait for more requests to the same destination and send them all at once, saving network bandwidth and computing resources to construct and keep track of remote migration acknowledgements. Choosing one option over the other is essentially based on the trade-off between performance and network utilization.

On the one hand, sending migration requests as soon as possible ensures the agents are fully utilized. On the other hand, bundling requests with the same destination together reduces the overhead of constructing and keeping track of too many remote migrations. This problem is similar to the "small package problem" of TCP/IP networks, in which very small data are transmitted repeatedly, resulting in large overhead [12]. The problem is addressed in [12] by combining messages until their number reaches a threshold or a timeout happens before sending them out. By adjusting the timeout and the threshold of number of messages, network optimization can be achieved with acceptable performance degradation.

However, unlike the "small package problem", asynchronous remote migration request problem has one additional constraint. If the request threshold is too low and there are multiple agents (a computing node is supposed to be able to handle as many as 1 million agents at the same time) requesting to migrate to the same destination in a short period of time, exception of type `java.net.SocketException: Too many open files` will occur. Conversely, when the request threshold is too high, source computing node will hang on to remote migration requests for too long, result in performance inefficiency. The most efficient way to set the request threshold and timeout value is by adjusting these throughout execution according to the number of agents residing at a node and the remote destination they are going to migrate to at a certain time. Unfortunately, current MASS architecture does not support this knowledge, and accomplishing this will require code refactoring and implementing completely new features. These are worthy goals of a future project themselves and are outside the scope of this project. For the current project, message threshold is hard-coded to 1000 and timeout value is 30 miliseconds.

3.6 Distributed Termination Detection

In synchronous MASS implementation, each `callAll()` results in each computing node executing exactly one method. Thus, the 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. Consequently, 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. A 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 self-evident 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 times during `callAllAsync()` execution because it can become *idle* and *active* many times. For example, an *idle* node can become *active* again if new agents migrate to it from another node.

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

In implementation, a computing node uses `originatorPid` to store its originator's process ID, `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 Figures 3.6, 3.7, and 3.8. Figure 3.6 shows what needs 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 adds nodes' `Pids` to its `receiverList` if they have agents at the beginning of execution. Likewise, each slave node with initial local agents sets 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 ← 0
end if
```

(b) at slave node

Figure 3.6: Initialization of distributed termination detection algorithm

Figure 3.7 shows the logic when remote migrations occur. If the receiver of a remote migration was in the *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`.

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 3.7: Distributed termination detection algorithm handling of remote migrations

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` 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 3.8.

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 3.8: Distributed termination detection algorithm's main function

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

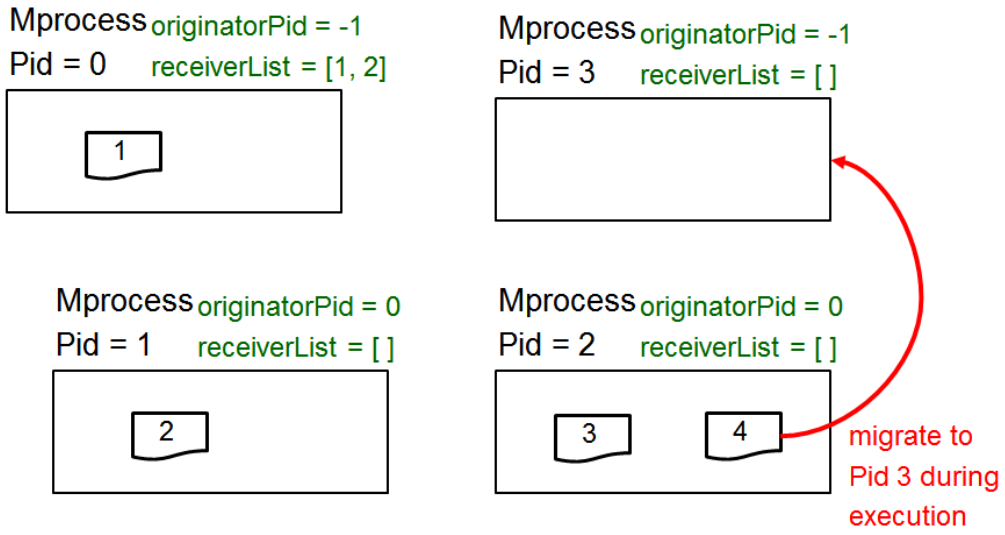
Suppose the algorithm is incorrect. That means when node 0, the master node, becomes *idle* and starts issuing `AsyncResultRequest`s to collect results, there exists at least one node which is still in *active* state.

If the node is still *active*, it has not sent *idle notification* to its *originator*, based on Algorithm 3.8a. If its *originator* is the master node, then the master node's `receiverList` is not empty. Therefore, the 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. The *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 3.6b, that node has 0 as its *originator*. □

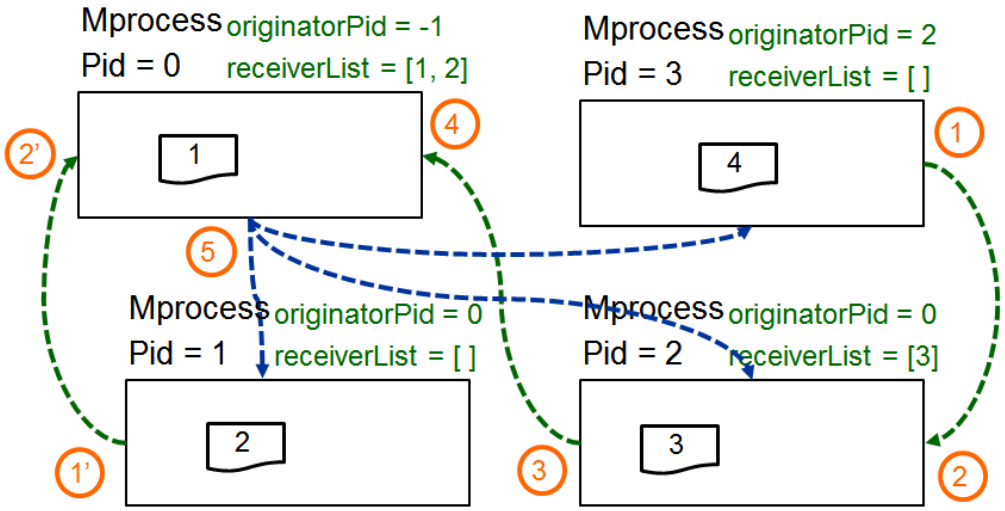
Let us demonstrate the algorithm with an example illustrated in Figure 3.9. A distributed system has four computing nodes. There are four `MASS` agents at the beginning of `callAllAsync()` execution. The agents are numbered from 1 to 4. Agent 1 resides at node 0. Agent 2 resides at node 1. Agent 3 and 4 reside at node 2, while there is no agent at node 3. From Algorithm 3.6b, node 1 and 2 set their `originatorPid`s to 0, which is the master node's `Pid`, while the master node adds 1 and 2 to its `receiverList`. Neither the `originatorPid` nor the `receiverList` of node 3 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 3.7a, node 3 considers node 2 as its *originator* and sets its `originatorId` to 2. Based on `ACK` response, node 2 adds 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, and is therefore considered *idle*. Hence, it in turn sends *idle notification* to its *originator* 0. After removing 2 from its `receiverList`, node 0 becomes *idle*, which allows it to issue `AyncResultRequest` to collect the execution results.



(a) Beginning of execution



(b) End of execution

Execution steps: (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 3.9: Example of Distributed Termination Algorithm

3.7 Verification

The goals of verification are as follows:

Code correctness In deterministic application, the new approach should produce the same result as the sequential execution. This is achieved by comparing results from both approaches on classic deterministic parallel problems, such as Mandelbrot. If the results are the same, 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 approach. Again, this is verified by applying the two approaches to parallel problems such as Climate Change Simulation. Data analysis performance is measured in terms of execution time under various configurations. Execution time is measured in milliseconds (ms). Execution configuration includes different numbers of computing nodes and numbers of threads per node when executing an application.

Scalability The new approach should be capable of handling hundreds of thousands of agents and places per node. Execution configurations with a high number of agents will be 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 Chapter 4 “Experiments”.

Chapter 4

EXPERIMENTS

The first goal of conducting the present experiments is to evaluate the performance of asynchronous and auto migration migration against synchronous migration under various configurations: different numbers of computing nodes, threads, and agents per node. The second goal is to verify that adding auto migration logic to asynchronous migration does not affect performance significantly. The last goal is to compare performance of MASS versus the Java-implemented version of MPI, a popular parallel framework. For each configuration, execution for both the asynchronous and synchronous approach is performed three times, after which the results for each approach are averaged.

4.1 Setup

Experiments were conducted using a machine grid available at the Linux lab, located at building UW1-320 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. In this problem, points in 2-D space are colored based on a certain algorithm. For this experiment, we use the *Escape time* algorithm. The algorithm performs the same calculation at each point. Based on the point's x- and y-coordinate, the calculation will have a different number of iterations. The number of iterations determines the point's color. Available colors are picked beforehand and stored in an array. The number of iterations is the index of the color in the array. The *Escape time* algorithm is summarized in Figure 4.1.

The Mandelbrot Set problem was chosen is because the number of iterations, and, hence, execution time vary among points. This simulates the condition in which agents, which are distributed to calculate the number of iterations at each point, will finish at different times. Agents that finish earlier than the rest will remain idle. Solving the problem using a combined asynchronous and auto migration approach, as well as a synchronous migration approach, and then comparing the results will prove whether asynchronous migration can utilize this idle time.

All experiments were 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 across computing nodes and among

$x_0 \leftarrow 0$
$y_0 \leftarrow 0$
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

the threads at each node, which can reach maximum of 4. Agents are distributed at the leftmost place at each row. They migrate and perform the calculation at each place until they reach the rightmost place of each 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 of places. Agents will start from the leftmost of the chunk, migrate, and perform the calculation at each place until the rightmost end of that chunk.

4.2 Performance with 4032 agents

The number 4032 is significant, because in the 2-D space of size 4032 x 4032, this number of agents satisfies the contraint for the first version of auto migration, as discussed in detail in Section 3.3. Each place is visited once by agents while the number of remote migrations is minimized.

Figure 4.2 provides an overview of performance of MPI and MASS with synchronous, asynchronous, and auto migration under various numbers of computing nodes and numbers of threads per node. Overall, execution time reduces and performance improves when the number of nodes and threads increases. Performance of synchronous agent migration of MASS does not improve as much as with the other approaches. Furthermore, the trend of performance when the number of computing nodes increases is the same for different number of threads per node configurations. Hence, there is no interaction effect between number of nodes and number of threads per node in this case.

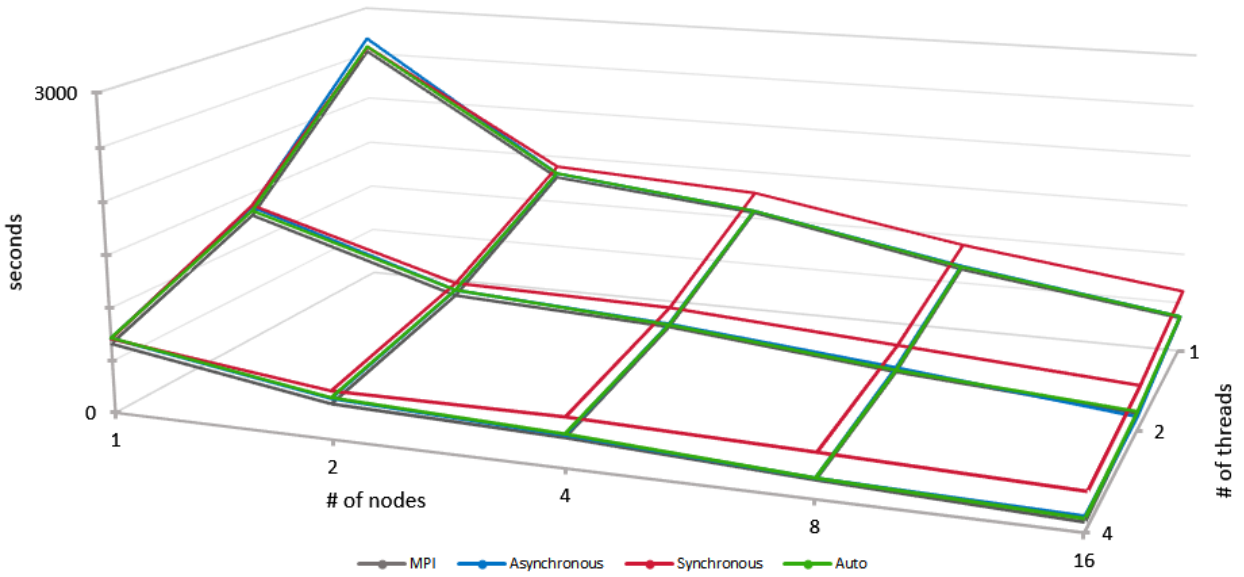


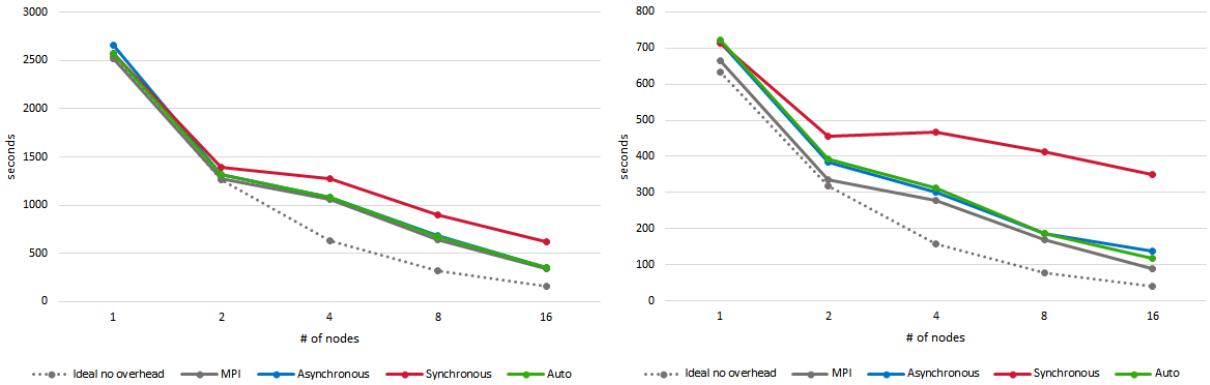
Figure 4.2: Overall performance of various MASS agent migration types, when agent size is 4032, and MPI

The effect of the number of nodes on performance is shown in Figures 4.3. Figure 4.3a illustrates the performance when the number of computing nodes increases from 1 to 16 while the number of threads per node is fixed at 1. As we have already observed, the performance improves when the number of nodes increases. Execution time of asynchronous and auto migration in MASS closely resembles each other and that of MPI, while that of synchronous migration is higher, especially when the number of computing nodes increases. This show that adding auto migration logic, that automatically identifies the migration pattern, to asynchronous migration does not alter performance significantly. In addition, the performance does not improve significantly when increasing the number of nodes from 2 to 4. This is because the bulk of calculation is concentrated at the center of the space, and the total execution time can only be as fast as the slowest node. When there are two nodes, the heavy calculation portion is equally divided among the node. That is why execution time of 2 nodes is half of that of the 1-node configuration. When the number of nodes is 4, the heavy calculation portion is still only concentrated in the second and third quarter. Hence, performance improves, but not as dramatically.

Figure 4.3b shows the performance when the number of computing nodes increases from 1 to 16 while the number of threads per node is fixed at 4. We also observe the same pattern as when there is 1 thread per node. The overhead of asynchronous and auto migration in MASS compared to MPI is more distinct in this configuration. This is because, unlike MASS,

connections among computing nodes in MPI are established before execution happens and are only closed explicitly long after. Connections among nodes in MASS must be established at the time of execution and are closed automatically when execution finishes. The reason we did not exclude MASS connection establishment time when comparing MASS and MPI is because this is user's perceived performance of the libraries. Users perceive execution time of MASS, or MPI, as the time from which they submit a task until they get the result. Whether that period includes setup time or is just pure execution time is irrelevant to the end-users. As such, we chose to include MASS connection establishment time in the performance evaluation.

Furthermore, MASS also spends time to construct `Place` and `Agent` objects for each execution. In this case, 16 millions `Place` objects and 4 thousands `Agent` objects are needed.



(a) 1 thread per node

(b) 4 threads per node

Figure 4.3: Effect of number of computing nodes on MASS and MPI performance, agent size = 4032

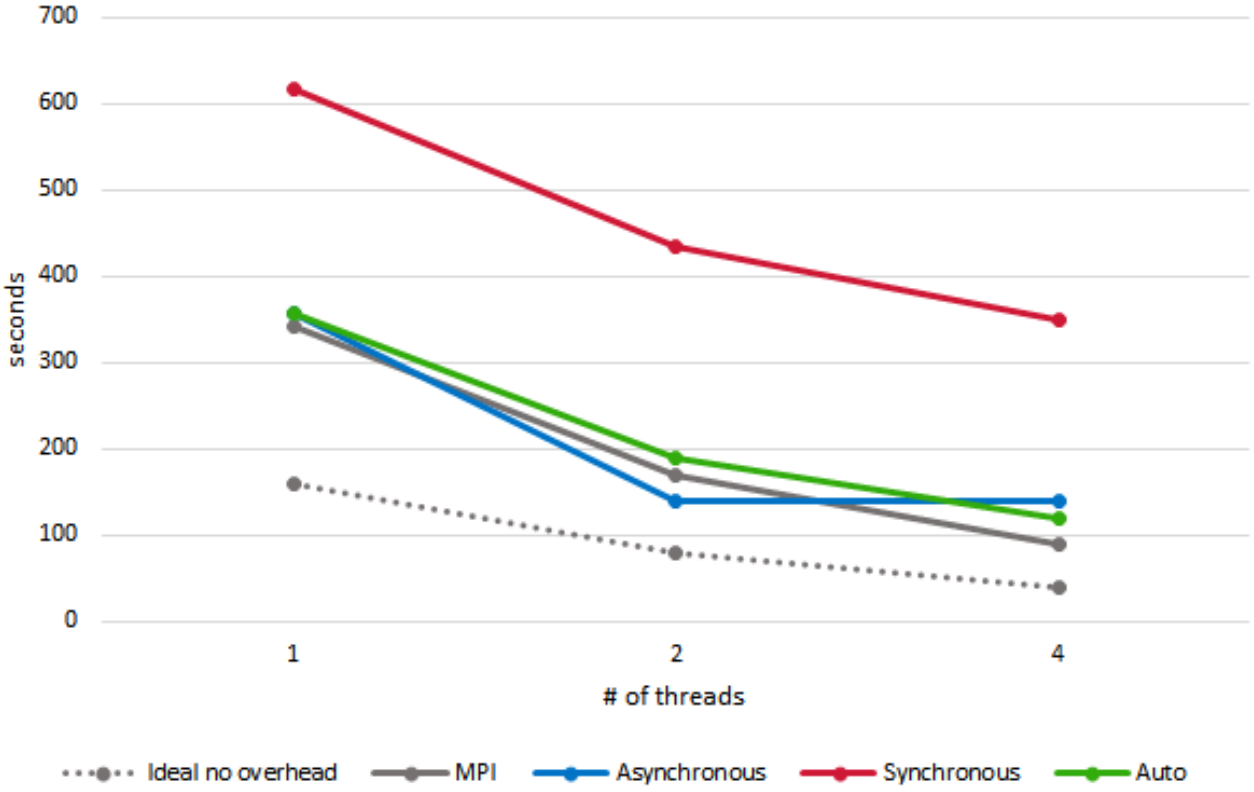


Figure 4.4: Effect of number of threads on MASS and MPI performance

16 computing nodes configuration. Agent size = 4032

Figure 4.4 shows the effect of the number of threads per node on performance. The number of nodes is fixed at 16. Again, performance improves when the number of threads per node increases. This is because when there are more threads per node, more CPU cores are utilized to perform computation. Hence execution time is sped up. Performance of synchronous migration in MASS is worse compared to that of asynchronous and auto migration in MASS, and MPI.

The exact performance numbers for Figures 4.2, 4.3a, 4.3b, and 4.4 are shown in Tables A.4, A.5, A.8, and A.9 in Appendix A.

4.3 Performance with 903168 agents

In addition to experiments using 4032 agents, we also conducted experiments with 903168 agents. Firstly, this number was chosen because we want to ensure that asynchronous and auto migration can handle large numbers of agents. Currently, MASS has a limit of 1 million

agents per node. As the experiment was conducted with a minimum of 1 computing node, we picked the number of agents to be at most 1 million in order to satisfy the limit of number of agents per node while ensuring consistency across different numbers of node configurations. Secondly, 903168 was chosen instead of exactly 1 million because the agents need to be equally divided among places, which are of size 4032 by 4032; computing nodes, which have values of 1, 2, 4, 8, and 16; and threads, which have values of 1, 2, and 4.

Additionally, as discussed in Section 3.3, for a `Places` group of dimension 4032 x 4032, the number of agents needs to be 4032 to perform auto migration in MASS. Hence, no experiment for auto migration was conducted for this configuration of 903168 agents.

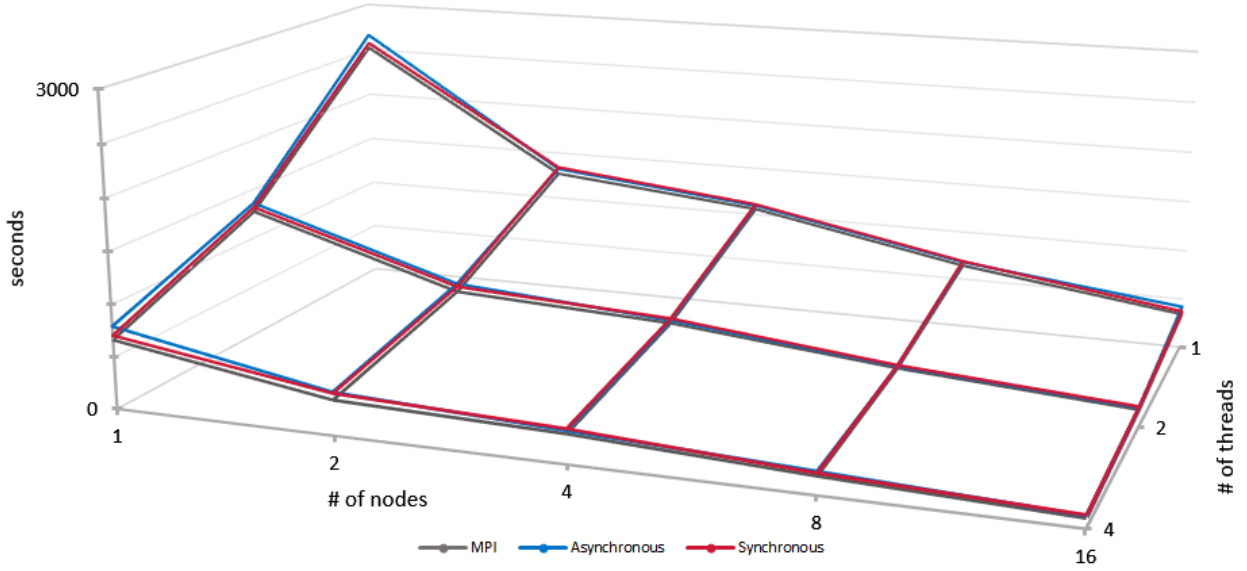
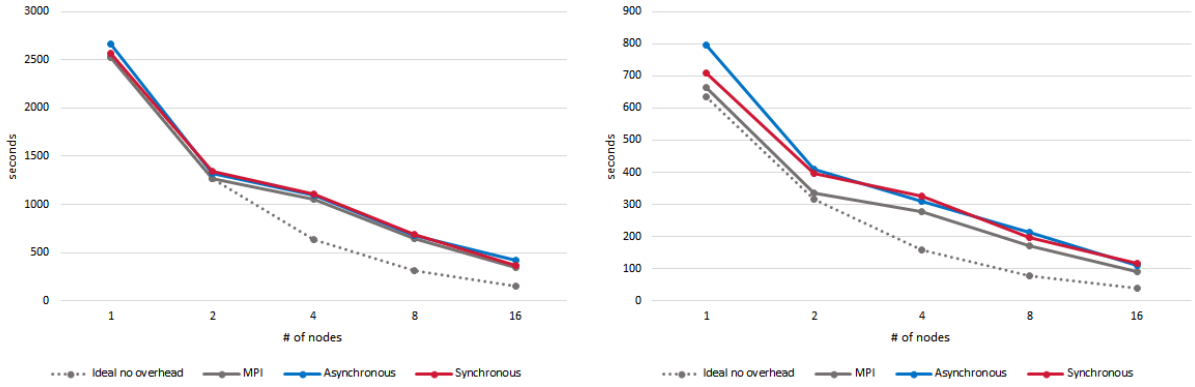


Figure 4.5: Overall performance of various MASS agent migration types, when agent size is 903168, and MPI

Figure 4.5 shows overall performance of various MASS agent migration types, when agent size is 903168, and of MPI. Overall, performance improves when the number of threads and the number of computing nodes increase. There is no significant difference in performance between any configurations. Again, the trend of performance when the number of computing nodes increases is the same for different number of threads per node configurations. Hence, there is no interaction effect between number of nodes and number of threads per node when the number of agents is 903168.

The effect of the number of nodes on performance is shown in Figures 4.6. Figure 4.6a illustrates the performance when the number of computing nodes increases from 1 to 16

while the number of threads per node is fixed at 1. Figure 4.6b shows performance for the same range of number of nodes, but the number of threads per node is fixed at 4. As we have already observed in Figure 4.3, the performance improves when the number of nodes increases. Similar to experiments when the number of agents is 4032 in Figure 4.3, performance of asynchronous and auto migration in MASS closely resembles that of MPI. However, in this case, performance of synchronous migration is also similar to that of the other two approaches. This is because, for 903168 agents and a space of 4032 by 4032 places, each agent only needs to visit 18 places to ensure each place is visited once. This means that the master node only needs to issue 18 `callAll()` and `manageAll()` instead of 4032, as in previous experiments. Therefore, communication overhead is significantly reduced for synchronous migration approach. This change in the total number of agents does not give any advantage to asynchronous the migration approach. Again, we also observe here that the performance does not improve significantly when increasing the number of nodes from 2 to 4. The reason is the same as discussed previously.



(a) 1 thread per node

(b) 4 threads per node

Figure 4.6: Effect of number of computing nodes on MASS and MPI performance, agent size = 903168

Figure 4.7 shows the effect of the number of threads per node on performance. The number of nodes is fixed at 16. Again, performance improves when the number of threads per node increases. There is no significant difference in performance between asynchronous migration and synchronous migration in MASS. Performance of MPI is better than MASS because of the overhead in connection establishment, as well as `Place` and `Agent` object initialization in MASS.

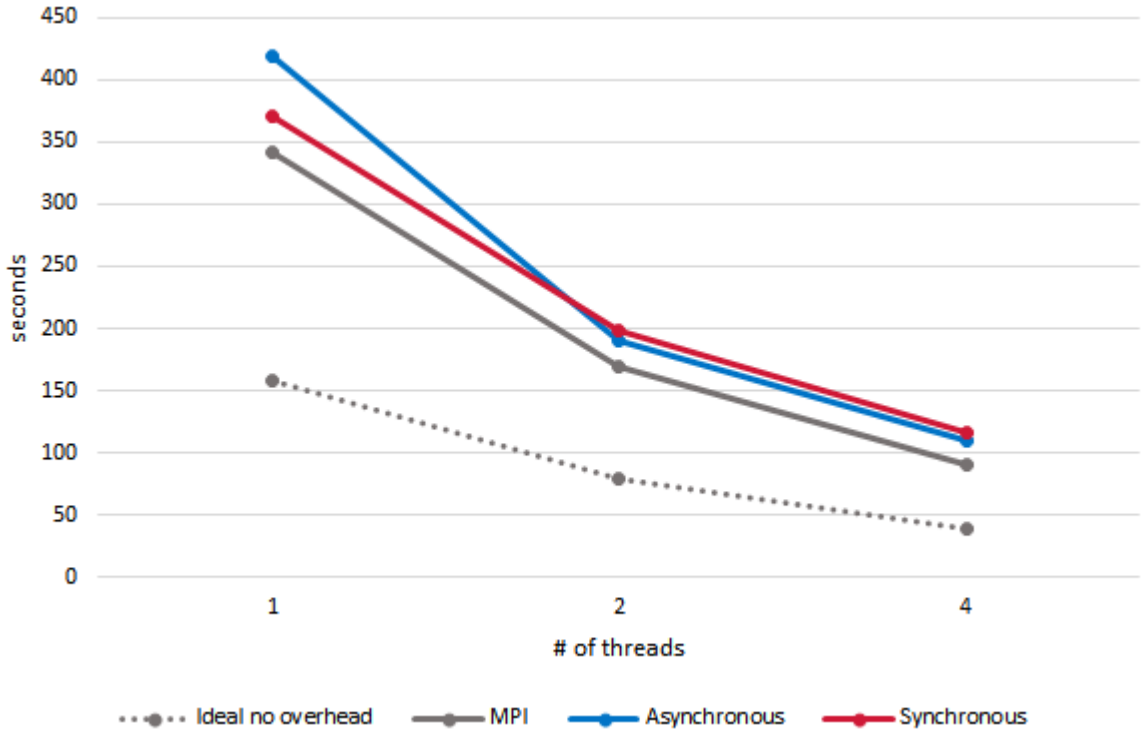


Figure 4.7: Effect of number of threads on MASS and MPI performance

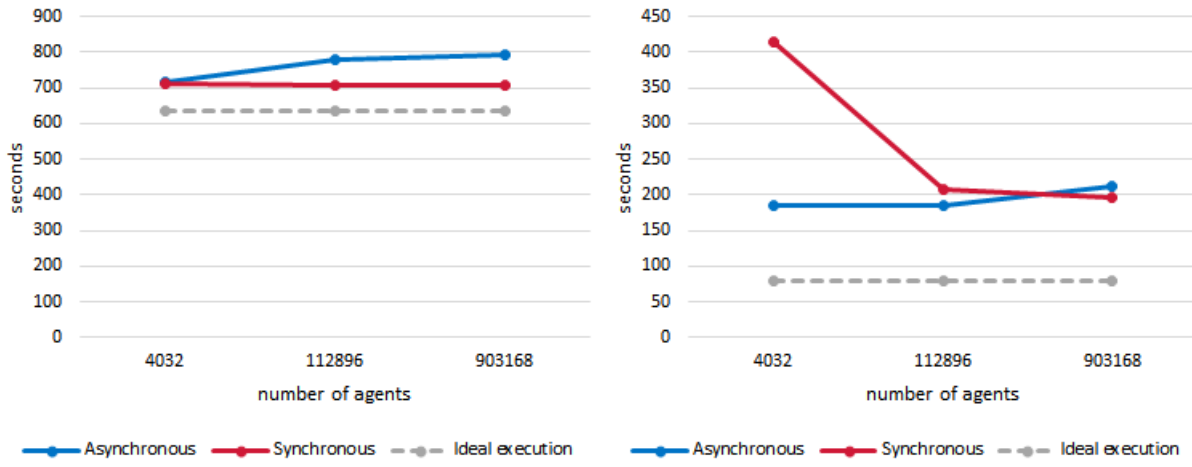
16 computing nodes configuration. Agent size = 903168

The exact performance numbers for Figures 4.5, 4.6, and 4.7 are shown in Tables A.6, A.7, and A.9 in Appendix A.

4.4 Effect of number of agents

This experiment is intended to evaluate how the asynchronous approach will perform when the number of agents increases. We compare performance of asynchronous and auto migration when there are 4032, 112896, and 903168 agents. These numbers are chosen so that agents are distributed evenly among rows of places. The number of threads per node is fixed at 4. The number of computing nodes is 1 and 8. The 2 values allow us to evalute performance when the agents are concentrated in one node, which has no remote migrations, as well as when they are distributed among a number of computing nodes, which require remote migrations. Average results are plotted in Figure 4.8. The exact performance numbers are listed in Tables A.2 and A.3 in Appendix A.

The optimal execution number is deduced by dividing sequential performance, which is



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

Figure 4.8: Effect of number of agents on performance

2521 seconds and is summarized in Table A.1, by total number of threads. We observe the followings from Figure 4.8.

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

- Asynchronous migration performs worse than synchronous migration in both configurations when there are 903168 agents, but is 58% faster than the synchronous approach when there are 4032 agents in 8-node configuration.

- None of the approaches performs as well as the ideal goal, which is when there is no overhead in parallelization.

These observations can be explained based on the fact that when there are more agents, synchronous execution needs to issue fewer `callAll()` and `manageAll()` calls. Hence, its communication overhead is reduced and performance improves. This gets even better in the case of 1-node configuration because there is no inter-node communication, only synchronization among local threads. The Asynchronous approach, on the other hand, has the size of the `asyncQueue` increased. Dequeuing agents from `asyncQueue` needs to be synchronized among threads. Therefore, its overhead actually increases when the number of agents increases.

Additionally, each approach suffers from different kind of overhead, so none performs as well as the optimal. Overall, the experiments also show that asynchronous approach can handle 16 million places and 900 thousands agents at one node.

4.5 Asynchronous migration performance on real-life application BioNet Network Motif

Toward the end of the project, we also incorporated the asynchronous migration feature into existing MASS implementation of real-life application BioNet Network Motif [13]. The problem definition is to construct all subgraphs of a certain size, called motif size, of an input graph. In MASS simulation implementation of the problem, each place represents a graph node. Agents will traverse the graph by migrating among places, constructing subgraphs along the way. If there are more than one path to traverse from a node, agents will spawn additional child agents to traverse these extra paths. When a subgraph has the required size, agents deposit that subgraph result to its current place and kill itself. When there is no more agent to execute, subgraph results are collected from all the places.

We did not incorporate the auto migration feature into this application because agents need to traverse graph nodes, not matrix, which is not supported by the current implementation of the auto migration feature. We reused experiment results for the sequential and the synchronous approach conducted in [13]. Their exact results are shown in Table B.1, B.2, and B.3 in Appendix B. Exact results for the asynchronous approach conducted in this project are shown in Table B.4 and B.5. For each configuration, execution was performed three times, after which the results are averaged.

Follow the same convention as [13], the results for the asynchronous approach are also measured in miliseconds.

4.5.1 Motif size 4

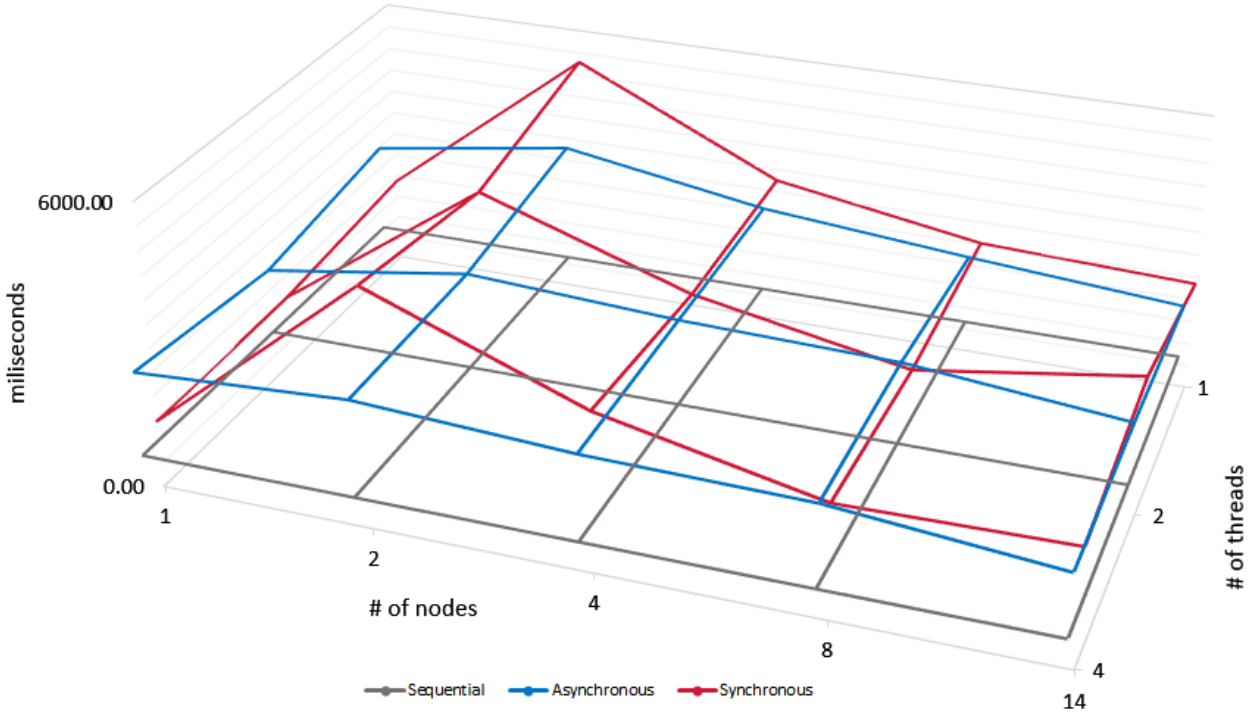
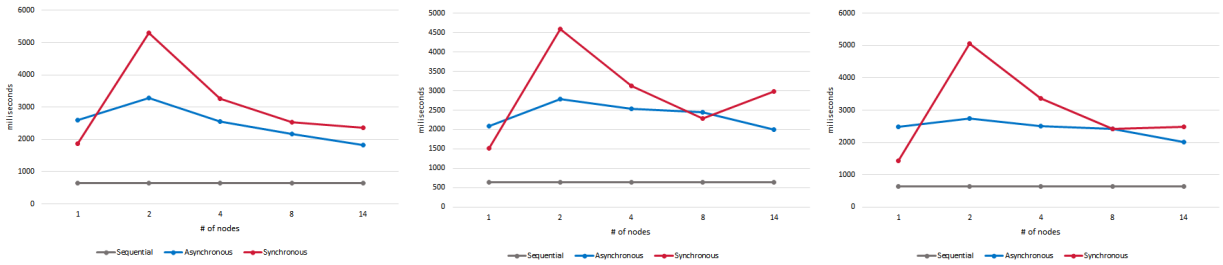


Figure 4.9: Overall performance of BioNet Network Motif implemented using asynchronous and synchronous agent migration of MASS, when motif size is 4

Figure 4.9 shows overall performance of asynchronous and synchronous migration approach for motif size 4. Except for the 1-node configuration, the asynchronous approach performance is better or similar to that of the synchronous approach, although it is still not as good as the sequential approach. There is also no interaction effect between number of nodes and number of threads per node in this case. The difference in performance is more obvious when displaying the data using line charts in Figure 4.10.

When there are more than one computing node, the asynchronous migration approach performs better than the synchronous approach because agent can migrate freely among places without the need of `manageAll()` function calls, which have communication overhead. When there is one computing node, as there is no need to broadcast messages, performance of synchronous approach improves. However, the asynchronous approach still needs to construct additional data structures and uses `asyncQueue` to enqueue and dequeue agents in this configurationft. Hence, its performance is worse than the synchronous approach in this

case.



(a) 1 thread (b) 2 threads (c) 4 threads

Figure 4.10: Effect of number of nodes on performance for motif size 4

The synchronous migration execution time peaks when the number of nodes is 2. This is because when there are 2 nodes, agents need to migrate remotely between them. This remote migration overhead increase the execution time compared to the 1-node configuration, which has no remote migration. When the number of computing nodes increases to 4 nodes or more, performance of synchronous approach improves because the workload is now distributed among more nodes, which have more computing resources. This also explains the similar performance pattern seen in the asynchronous approach.

Performance does not improve much for both approaches when the number of threads per node increases. This is because Network Motif does not contain a lot calculation to perform. At each place, which represents a graph node, an agent only needs to add that node label into the subgraph it is constructing. Hence, the majority of the time is spent dividing the agents in the agent bag among the threads in the case of synchronous migration. For the asynchronous migration approach, the majority of the execution time is spent on enqueuing and dequeuing agents from the `asyncQueue`. These activities in both approaches require critical code sections, which cannot be parallelized and do not benefit from increasing the number of threads. Moreover, BioNet Network Motif application performs a lot of agent objects initialization and destruction, which involve OS IO calls that are also critical and cannot be parallelized.

4.5.2 Motif size 5

Figure 4.11 shows overall performance of asynchronous and synchronous migration approach for motif size 5. The asynchronous approach performance is better than that of the synchronous approach. It is not as good as the sequential approach. There is also no interaction

effect between number of nodes and number of threads per node in this case. The difference in performance is easier to observe when displaying the data using line charts in Figure 4.12.

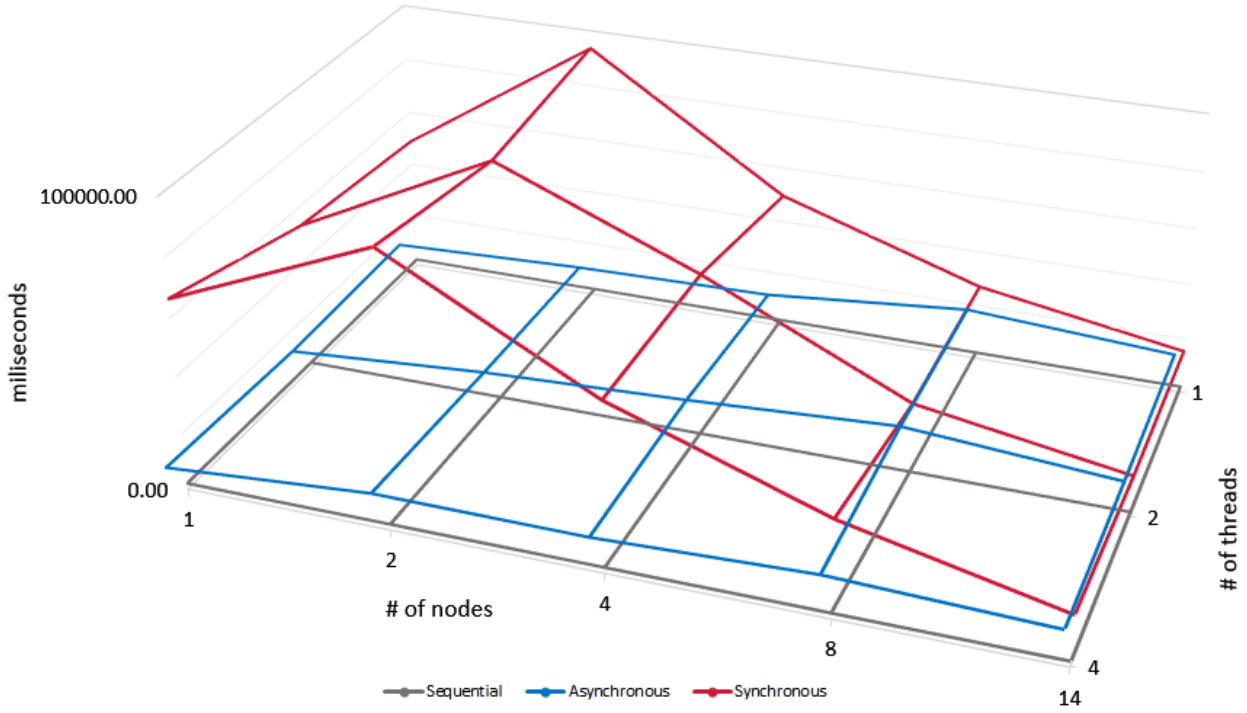
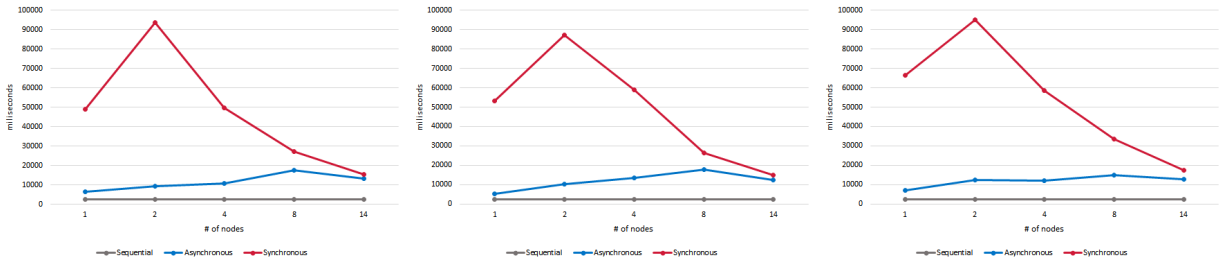


Figure 4.11: Overall performance of BioNet Network Motif implemented using asynchronous and synchronous agent migration of MASS, when motif size is 5

Similar to experiments with motif size 4 in Section 4.5.1, the synchronous migration execution time peaks when the number of nodes is 2, for the same reason discussed previously. Performance does not improve much for both approaches when the number of threads per node increases as well.

However, unlike in motif-size-4 experiments, execution time of the asynchronous approach increases when the number of nodes increases. This is because when there is only one computing node, there is no remote agent migration need to be done. However, when the number of nodes increases, the number of remote migration also increases while the benefit of increasing the computing resource does not offset this overhead much. Hence, execution time increases, but not by a lot comparing to the increase in the synchronous approach.



(a) 1 thread (b) 2 threads (c) 4 threads

Figure 4.12: Effect of number of nodes on performance for motif size 5

4.6 Summary

Through various experiments on both synthetic and real-life application using various configurations, we showed that the new asynchronous and auto migration have achieved their performance goals and is ready for production. In most of the chosen configurations and problem sizes, both asynchronous and auto migration have similar or better performance compared to the synchronous approach. In some cases, the asynchronous approach has similar performance as MPI. By experimenting using 903168 agents, we showed that asynchronous migration can still handle the required maximum number of agents per node previously defined. We also showed that adding auto migration logic on top of the asynchronous approach does not affect execution time. In short, we achieved the original goal of improving MASS library by reducing communcation overheads.

Chapter 5

CONCLUSION AND FUTURE WORK

In this project, we implemented both asynchronous migration and auto migration functionality of agents for the Java version of MASS library. This includes creating architecture and detailed design of the new functionalities, which takes into account compatibility with existing functions; code implementation; testing for correctness; and evaluating performance on the Mandelbrot Set problem. Experiments using the Mandelbrot Set showed that asynchronous and auto migration provide a better utilization of resources and performance when computation is not distributed evenly across agents. This improvement will help increase the appeal of MASS library to programmers compared to other parallelizing frameworks.

Throughout this project, I gained valuable knowledge regarding parallel programming and project management. I understood better how Java handles locks for thread synchronization and cached memory for shared variables. I also learned how to implement socket in a way that supports multiple connections at the same time. During this project, by using reports to keep track of progress, I ensured the project went according to schedule. Having weekly meetings with my supervisor also provided benefits in making sure everything went according to plan. I also gained a new perspective on project planning and budgeting, setting aside more time for documentation.

In term of result, I successfully implemented and verify correctness of the new asynchronous and auto migration feature that I planned at the beginning of the project. Experiments on the Mandelbrot Set problem and BioNet Network Motif simulation under various configurations showed that the new functionalities have improved performance for most configurations. I also completed my project within the time constraint and completed the final report substantially to document my work.

The project does not address all the issues of MASS. Firstly, we did not separate communication establishment among computing nodes, which needs to be done for each execution and contributes to performance overhead, from execution. Future work can improve MASS to behave like MPI, in which connections can be established once and reused for many executions. Secondly, in order to achieve asynchronous execution, more synchronized critical code sections have been introduced into the library. This contributes to the execution overhead. Hence, execution of asynchronous migration is not better than synchronous approach in all cases. Lastly, the auto migration feature implemented in this project only supports data-analysis problems in which places are distributed in an n-dimension matrix. Data-analysis

problems are just a subset of tasks that MASS is able to solve. Besides, places are not always distributed in a n-dimension matrix. They can be used to simulate tree nodes or graph nodes, which represent various real-life entities, as well. Also, the number of agents at the beginning of auto migration execution needs to equal an exact number in order for it to proceed.Hence, this first implementation of the auto migration feature is not flexible.

Moving forward, many improvements can be made on the current project. Firstly, a C++ version can be implemented. Secondly, we are in the process of incorporating asynchronous and auto migration into existing real-life applications such as Network Motif and Climate Change Simulation. Thirdly, the auto migration feature can be improved to be more flexible in handling other types of problems, various number of agents, as well as different place distributions. Lastly, usability studies should be conducted to evaluate this aspect of the new approaches, as this is an important advantage of MASS. The new approaches should have enough documentation and method signatures should be easy to understand and convenient to use for MASS library users. To evaluate this, it will be necessary to survey existing users of MASS library; this will enable the measurement of the new code's usability. This should include questions that allow users to rate their experience working with the new functionality, and to report how often they choose the new approach over the synchronous one, and how hard is it for them to switch their existing code to the new function. Ideally, it should also elicit feedback in the form of comments.

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

[13] M. Kipps, W. Kim, and M. Fukuda, "Agent and spatial based simulation of biological network motif search," *Processing*.

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

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

Appendix A

MANDELBROT SET EXPERIMENT RESULTS

Table A.1: Sequential execution

Execution (seconds)			**Average**
#1	**#2**	**#3**	
2521	2525	2566	2537

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	782	793	781
1	4	903168	757	815	813	795
8	4	4032	188	185	185	186
8	4	112896	187	184	184	185
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	708
8	4	4032	425	409	406	413
8	4	112896	209	209	207	208
8	4	903168	195	201	195	197

Table A.4: Performance of asynchronous migration with agent size 4032

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2652	2660	2678	2663
1	2	1377	1318	1374	1356
1	4	722	715	715	717
2	1	1313	1313	1313	1313
2	2	701	681	697	693
2	4	386	383	385	385
4	1	1086	1078	1073	1079
4	2	576	572	554	567
4	4	301	302	303	302
8	1	679	684	679	681
8	2	368	366	366	367
8	4	188	185	185	186
16	1	353	366	352	357
16	2	185	186	192	188
16	4	135	141	141	139

Table A.5: Performance of synchronous migration with agent size 4032

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2569	2568	2568	2568
1	2	1314	1313	1495	1374
1	4	710	710	716	712
2	1	1391	1387	1388	1389
2	2	765	767	762	765
2	4	453	460	455	456
4	1	1272	1272	1272	1272
4	2	723	731	732	729
4	4	469	460	471	467
8	1	885	909	909	901
8	2	585	597	564	582
8	4	425	409	406	413
16	1	614	625	616	618
16	2	441	430	431	434
16	4	350	346	355	350

Table A.6: Performance of asynchronous migration with agent size 903168

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2651	2659	2675	2662
1	2	1392	1342	1342	1359
1	4	757	815	813	795
2	1	1316	1320	1326	1323
2	2	685	745	731	720
2	4	423	399	406	409
4	1	1096	1093	1097	1095
4	2	565	561	571	566
4	4	321	305	302	309
8	1	690	662	691	681
8	2	342	347	341	343
8	4	207	214	216	212
16	1	411	420	427	419
16	2	195	188	188	190
16	4	115	108	107	110

Table A.7: Performance of synchronous migration with agent size 903168

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2571	2571	2573	2572
1	2	1315	1315	1314	1315
1	4	707	708	708	708
2	1	1348	1346	1327	1340
2	2	691	692	690	691
2	4	406	390	396	397
4	1	1109	1110	1110	1110
4	2	579	585	585	583
4	4	336	317	322	325
8	1	678	692	699	690
8	2	351	351	351	351
8	4	195	201	195	197
16	1	375	375	363	371
16	2	196	202	196	198
16	4	117	116	116	116

Table A.8: Performance of auto migration

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2574	2575	2576	2575
1	2	1317	1319	1318	1318
1	4	739	712	713	721
2	1	1313	1316	1315	1315
2	2	694	695	696	695
2	4	386	401	393	393
4	1	1078	1087	1073	1079
4	2	568	554	553	558
4	4	309	313	317	313
8	1	681	677	655	671
8	2	341	341	340	341
8	4	187	184	190	187
16	1	362	354	354	357
16	2	191	185	190	189
16	4	109	132	116	119

Table A.9: Performance of Java MPI

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2521	2521	2521	2521
1	2	1270	1270	1270	1270
1	4	665	665	665	665
2	1	1270	1271	1270	1270
2	2	639	639	639	639
2	4	334	334	337	335
4	1	1062	1060	1054	1059
4	2	539	531	542	537
4	4	278	278	281	279
8	1	645	638	648	644
8	2	325	322	325	324
8	4	172	172	169	171
16	1	338	345	344	342
16	2	170	170	170	170
16	4	90	90	90	90

Appendix B

BIONET NETWORK MOTIF EXPERIMENT RESULTS

Table B.1: Average sequential execution, reused from [13]

	Motif Size	
Execution (miliseconds)	648	2459.83

Table B.2: Average synchronous migration execution, in miliseconds, motif size 4, reused from [13]

# of nodes	**# of threads**		
	1	**2**	**4**
1	1867.67	1515	1438.33
2	5294	4585	5063
4	3254	3121	3368
8	2529	2288	2410
14	2358	2985	2490

Table B.3: Average synchronous migration execution, in miliseconds, motif size 5, reused from [13]

# of nodes	# of threads		
	1	**2**	**4**
1	48838.5	53179	66333
2	93648	87070	95212
4	49570	59067	58436
8	27156	26466	33629
14	15400	14790	17431

Table B.4: Asynchronous migration execution, in miliseconds, motif size 4

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	2615	2605	2579	2600
1	2	2138	1906	2226	2090
1	4	2409	2363	2652	2475
2	1	3267	3312	3283	3287
2	2	2771	2784	2787	2781
2	4	2805	2742	2690	2746
4	1	2487	2614	2590	2564
4	2	2511	2580	2536	2542
4	4	2546	2373	2595	2505
8	1	2230	2129	2138	2166
8	2	2456	2285	2573	2438
8	4	2072	2561	2644	2426
14	1	1876	1782	1791	1816
14	2	2005	2100	2089	2065
14	4	2094	1947	1965	2002

Table B.5: Asynchronous migration execution, in miliseconds, motif size 5

# of nodes	Threads per node	Execution (seconds)			Average
		#1	#2	#3	
1	1	6696	6318	6750	6588
1	2	5257	5378	5166	5267
1	4	7800	6448	7057	7102
2	1	8108	8388	11587	9361
2	2	10455	11044	9769	10423
2	4	13927	12002	11700	12543
4	1	12136	8695	11518	10783
4	2	11247	14624	15031	13634
4	4	13505	11157	12040	12234
8	1	18407	19036	15092	17512
8	2	18598	18772	16201	17857
8	4	11665	16950	16452	15022
14	1	11555	14748	13317	13207
14	2	11901	13486	11883	12423
14	4	12727	12435	13005	12722