Implementation of MASS C++ Library over cluster of multi-core computing nodes

Narayani Chandrasekaran

Master of Science

in

Computer Science & Engineering

University of Washington

2012

Faculty Advisor: Munehiro Fukuda, Ph.D.

Abstract
With growing complexity of computer simulations and availability of multi-core processors, parallel computing has become an important in all fields of science and technology, mainly to improve the speed of computation. Some of the existing libraries for parallelization include OpenMP, MPI, and MapReduce. But these libraries have few limitations such as difficulty in mapping the application algorithm to the underlying architecture of the parallelization tools. MASS is a parallelization library that is designed to overcome these limitations. It composes a user application of distributed array elements and multi-agents, each representing an individual simulation place or an active entity. All computation is enclosed in each of elements and agents that are automatically distributed over different computing nodes. Their communication is then scheduled as periodical data exchanges among those entities using their logical indices. This paper presents the implementation of “Places” functionalities of MASS library in C++.

Keywords: OpenMP, MPI, MapReduce, MASS, Places, Multi-core, Distributed array
1.0 Introduction

With growing complexity of computer simulations and availability of multi-core processors, parallel computing has become an important in all fields of science and technology, mainly to improve the speed of computation. Some of the existing libraries for parallelization include OpenMP, MPI, and MapReduce. But these libraries have few limitations. For example, Map reduce is simple to use but requires data to be in the form of key value pairs, which is not suitable for all application. MPI, OpenMP or Hybrid of OpenMP or MPI requires user to have an understanding of parallel programming paradigm. Coding the sequential version of the application is itself difficult; hence parallelization of such scientific applications by mapping to the algorithm to the underlying parallelization tools becomes very exhaustive. MASS (Multi-Agent Spatial Simulation) library is being developed with goal of providing the users with a new parallelization library that facilitates:

- Automatic parallelization
- Utilization of SMP (Symmetric Multi-Processor) cluster.
- Abstraction of parallelization constructs from the user: No awareness of processes, threads and their communication.
- Single programming paradigm i.e. provides both distributed memory and shared memory model.

It is a part of the Sensor Grid research project that is currently underway at the UWB distributed systems lab and designed to be suitable for scientific computations including molecular dynamics, home automation, artificial society to name a few. MASS (Multi-Agent Spatial Simulation) has been implemented in java by Timothy chaung[]. But Java version has not met the performance requirement of MASS library. The performance issues are due to Java reflection being used by the library to invoke user defined functions and communication overhead. C++ version of MASS library is hoped to have better performance than Java version. Scope of my project is to implement “Places” functionalities of MASS (Multi-Agent Spatial Simulation) library in C++.
The following section describes related work in section 2.0, Main components of MASS library in section 3.1, Execution Model in section 3.2, Program specification and coding examples in section 3.3 Section 4 contains evaluation of MASS C++ library in terms of performance and usability. Section 5 contains problems encountered conclusion and future work.

2.0 Related Work

Literature survey was done to study existing libraries similar to MASS library. The study was done for distributed array, multi-agents and patterns for improving performance in parallel computing.

2.1 Distributed Array

The Global Array programming model is portable as it is compatible with most languages used for technical computing and does not rely on compiler technology for achieving parallel efficiency. Global Arrays combines the advantages of a distributed memory model and shared memory shared memory model Advantages of Global Arrays include avoidance of false sharing, data coherence overheads and avoidance of redundant data transfers. This toolkit takes care identifying data location and mapping indices. This functionality reduces huge programming effort and the possibility of error. Other functionalities of Global Arrays toolkit include core operations of the applications, task parallelism and data parallelism. Some of the features of this toolkit include provision for ghost cells and periodic boundary conditions, sparse data management, Mirroring i.e Caching distributed memory data in shared memory and synchronization control. Parallelization of user application is done with the use of Message Passing Interface (MPI), and GA provides various degrees of control to the application developer to exploit data locality for increased performance and multiple levels of memory hierarchy for data access optimization. However, all of the performance optimization features GA offers require advanced understanding of parallel programming paradigm. Even though MASS has some similarity to other distributed shared array supporting systems like UPC (Unified Parallel C), related to allocating global shared arrays, it is unique in implementing both one sided and collective operations as the form of user-defined remote method invocations rather than providing users with a system-predefined operations.

2.2 Multi-agents

MACE3J which is a Java base Multi-Agent Systems (MAS) Simulation platform and related works done. The design criteria and advantages of using MACE3J for multi-agent simulation platform have also been discussed. MACE3J supports the following features:
Facilitates simulations to re-run exactly and also takes into account timing aspects of simulation such as message delay.

- Allows flexible data gathering and visualization
- Allows runtime control of simulation parameters.
- Reusable components for constructing agents.

The multi-agent simulation environments described in this paper focuses on parallel execution of coarse-grained cognitive agents, each with rule-based behavioral autonomy. These systems provide agents with interest managers that work as inter-agent communication media to exchange spatial information as well as multi-cast an event to agents. From the viewpoints of agent-to-space or agent-to-event proximity, PDES-MAS recursively divides each interest manager into child managers, structures them in a hierarchy, and maps them over a collection of computing nodes for parallelization. MASS is different from MACE3J in handling fine-grain reactive agents that sustain a partial view of their entire space and interact with other agents in their neighbourhood. MASS also instantiates a large number of array elements and define their logical connection with exchangeAll.

### 2.3 Patterns for improving performance in parallel computing

Currently MASS java version does not support overlapping communication and computation and hence there is performance overhead when computing nodes have to exchange data with each other. There are three patterns [9] to for overlapping communication and computation including over decomposition, Non-blocking communication and speculation are discussed. And also, a very high-performance, memory-mapped interconnection system, called Merlin [10] is discussed which is shown to have performance improvement, in distributed applications involving data transfers between computing nodes. Merlin provides a very high performance 'anticipatory' approach to sharing memory in multicomputer environments. Literature synopsis also focusses on a concept called ‘reflective memory’ which is related to storing the shared information in the physical memory of each processor and any change to a shared location must be reflected to all logically equivalent locations in the memory of processors which share
this space. The technique discussed above is also related to the concept of overlapping communication and computation.

3.0 Method

This section describes the execution model, components, language specification and implementation of MASS library with respect to “Places” functionalities.

3.1 Main components of MASS library

The two main components of MASS library are ‘Places’ and ‘Agents’. Places are array of elements that are dynamically allocated over the cluster of computing nodes. Each element of ‘Places’ is called a place. Fig 1 represents “Places” and place for allocated to one SMP (Symmetric Multi -Processor) over the cluster. “Places” are capable of exchanging information with any other “Places”.

Agents are set of execution instances that are capable of migrating to any other “Places”, interacting with other “Places” and interaction with other agents as well.

3.2 MASS Execution Model

Figure 2 represents the execution model of MASS library. MASS library assumes cluster of multi-core computing nodes as the underlying architecture. Socket communication is used for synchronization between multi-threading processes that are spawned over the cluster. Each multi-core computing node spawns same number of threads as that of CPU cores per computing node. Mutexes are used for synchronization between threads. The threads spawned execute the functionality to be executed by a computing node in parallel.

During the initialization, the simulation space, based on its dimension as specified by the user, is mapped to Places and divided equally among the number of processors available for computation including the master node, as shown in the Figure 2. The “Places” allotted to each computing node are then equally divided among threads in each computing node (static scheduling) as
shown in Figure 2. The region highlighted in red shows the thread range. Each node will initialize itself as a server for exchanging data during computation, if required.

![Figure 2: MASS Library Execution Model](image)

### 3.3 Programming Specification for MASS C++

All processes involved in the computation of the user application of MASS library must call MASS::init() and MASS::finish() at the beginning and end of the code respectively, to start and finish the computation together. MASS::init() spawns and initializes threads based on the number CPU cores per computing node. MASS::finish() cleans up all the threads and closes all the socket connections. Snippet of MASS C++ programming interface is shown in Table 1.

<table>
<thead>
<tr>
<th>static void</th>
<th>init(string *args, int nProc, int nThreads)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Involves nProc processes in the same computation and has each process spawn nThreads threads.</td>
</tr>
<tr>
<td>static void</td>
<td>finish()</td>
</tr>
<tr>
<td></td>
<td>Finishes computation</td>
</tr>
</tbody>
</table>

**Table 1: MASS Interface**

#### 3.3.1 Call Method

It is a user provided framework for MASS library to choose the methods to be invoked. It is designed as a switch case. Each function should be given a function id. The function id will be used by the MASS library to identify the function to be invoked.

Code snippet of an application using MASS C++ library is shown in Figure 3.
3.4 Places Functionalities

3.4.1 CallAll()

Invokes user defined function, specified with a function id by passing argument[i] to element[i], on all individual “Place” objects. This is done internally using factory pattern. Return values for each computing node is an array of return value of each “place” object. Array of return values are sent over the SSH channel to the master node. Calls are performed in parallel among all multi-processors/threads. Each thread invokes callAll function in its thread range in parallel, which is highlighted in Figure 3.
3.4.2 Callsome()

Invokes user defined function, specified with a function id by passing argument[i] to element[i], on individual “Place” object. Return values are sent over the SSH channel to the master node. Highlighted region in Figure 4 represents the callsome function being invoked in one “Place” object.

![Figure 4: Representation of Callsome functionality](image)

3.4.3 ExchangeAll()

Allows a “Place” object to interact with other “Place” objects. ExchangeAll function is used to gather data from other “place” objects, if the information will be required for further computation i.e this function allows user to exchange data of each cell with the user defined neighboring cells in the simulation space. User has to define neighbors for each cell, in the form of relative indices to gather data. Figure 5 represents four neighbors for a “Place” object.

![Figure 5: Representation of ExchangeAll functionality](image)

exchangeHelper object instantiated during the initialization phase will be used for exchanging data among cells. During exchangeAll function, all local exchanges are performed first and if exchange needs to be performed with remote host, then each thread an entry to the exchange request map with remote hostname as the key and exchange request as the value. After local exchanges are done, each thread will begin processing a remote exchange request with the selected host on first come first serve basis. This is done to ensure that the connection object which is the socket descriptor is not shared by multiple threads.

When a thread picks up a host to process the exchange request, it checks the connection map to see if connection has already been established. If the value returned is null, then connection is established and entry is added to the connection map. The thread then starts processing the
exchange request. When there are more requests than the number of threads, the thread that finishes the processing its first request first will process the remaining requests.

3.5 Implementation

![Diagram of Master Node and Remote/slave node](image)

Figure 6 represents the design of master node and slave node for MASS library over the cluster. The implementation of each component with respect to “Places” is discussed below:

3.5.1 MASS

MASS is the infrastructure of the library on which the user application is executed, as shown in Figure 6. It is responsible for launching, synchronizing terminating remote process on cluster of computing nodes and maintains references to “Places” instances.

3.5.1.1 `init()`

Based on the number of processes as user input, remote hosts are identified from the machine.txt file. A SSH communication is established from master node to all the remote hosts. For each remote host, a MNode instance is locally created on the master node as a wrapper for maintaining the master and slave SSH connection. MProcess is then launched on the remote hosts using the channel object obtained by establishing the SSH channel. After all the remote processes have been launched, each process creates a pool of worker threads using the MThread instance that handles the coordination of multiple threads.
Libssh2 library is used to establish SSH channel and launch a process in a remote host. Throughout the application lifecycle, MASS provides useful methods for manipulating individual cells in the user defined simulation space, including direct control over “Place” instances.

3.5.1.2 finish()

It is called at the end of an application to deconstruct the cluster. This is done by sending termination commands to all slave processes and closing all connections for a graceful exit.

3.5.2 MNode

MNode is a wrapper for the SSH connection object that allows direct communication between the master node and the slave nodes. A collection of MNodes will be created during initialization to facilitate master-slave communication. Representation of MNode is master is shown in Figure 6. Each MNode instance contains a set of wrappers for sending and receiving communication messages. This channel is also utilized for when the slave nodes need to send return values to the master node. The data sent is serialized at the sender end and de-serialized at the receiver end respectively. This is done because objects are not automatically serialized in C++.

3.5.3 MProcess

MProcess is run on remote hosts as a launching platform for MASS functions, as shown in Figure 6. The MProcess facilitates all commands invoked by the master node and manages program flow on its behalf. The MProcess has three states in its lifecycle, initiation, running and deconstruction. During initialization, the MProcess establishes instantiates a pool of worker threads. After the initialization has finished, it sits in an infinite loop and is blocked on a read() call awaiting commands from the master node. Once a command is received, it calls the corresponding MASS function and return to the blocking read state for the next command. The master node can terminate all MProcess instances by sending the finish command. When a finish command is received, MProcess closes all existing connections and exits the loop.

3.5.4 MThread

MThread is class that is used to facilitate multi-threaded processing of the MASS library. Pthread library is used for instantiation, synchronization and termination of threads. All threads are synchronized on the MASS::STATUS variable that can be changed by the calling process to different states to either wake up the worker threads or put them to sleep.
3.5.5 ExchangeHelper

ExchangeHelper is a utility class used to establish and cache socket connection objects to facilitate inter-node communication. MASS initialization phase a server thread is instantiated and is blocked on accept( ) awaiting client connection requests.

If a connection between two nodes needs to be established, the node with lower process ID acts as a server and the node with higher process ID sends a connection request. The request is picked up by the server thread and a socket connection is established and cached on a connection map with the remote host name as its key and the connection object, which is the socket descriptor, as its value. This connection will be reused for future requests.

Connection establishment is done on demand. Whenever two nodes need to communicate with each other during an exchangeAll( ) call, one of the worker threads first attempts to retrieve a connection object from the connection map. If the return value is null, it then calls establishConnection( ) to establish connection to the remote host. All worker threads are synchronized on the connection map object to ensure that a cached socket connection is not shared by multiple threads.

4.0 Software and Hardware Requirements for MASS library

MASS library assumes cluster of multi-core computing nodes as the underlying architecture. Library requires Linux/Unix Operating system. Testing was conducted using 4- core machines connected over a 1 Gbps network.

5.0 Evaluation results and Analysis

“Places” functionalities of MASS C++ library is evaluated in terms of performance and usability. Spatial simulation analysis of MASS library is done using an application which is a two dimensional matrix with an initial value of sum of its coordinates. The value computed gets incremented by one for every iteration. Same application is used for programmability aspect as well.

5.1 Performance

Performance evaluation was done simulation size and application performance. All tests were conducted on cluster nodes with similar configuration. Each computing node was configured with Linux operating system with 4 cores and connected to 1Gbps network.

5.1.1 Simulation size granularity
Performance of multi-threaded version of MASS C++ library was evaluated using by using one computing node which is the master node. The application was run using 2, 4 and 6 threads each simulation space of size 100 X100, 300X300 and 500X500 for 900 iterations. Each of the iterations consisted of a single callAll function.

![Figure 7: Multi-threaded “Places” Granularity](image)

The result presented in Figure 5 shows linear scaling from simulation size of 300X300. Not much notable difference is seen for a simulation size with 100X100. Also, better performance was achieved using 4 threads. 6 threads has better performance than using 2 threads but the degradation in performance compared to 4 threads might be due to each computing having 4 cores.

### 5.1.2 Multi-Process “Places” Granularity

Performance of multi-process version of MASS C++ library was evaluated using by using one thread (main thread) for computing node which is the master node. The application was run using 1, 2 and 3 processors for each simulation space of size 100 X100, 300X300 and 500X500 for 900 iterations. Each of the iterations consisted of single callAll function.

Results presented in figure 8 shows linear performance from a simulation size of 300 X300. Not much noticeable difference is seen for a simulation space of size 100X100. It is due performance network latency.
5.1.3 Application Performance of “Places”

Figure 9 shows the execution time of the matrix application as discussed above. The test was conducting for a simulation space of 500X500 and each computing node with 4 threads and 900 iterations. Each iteration of the test consisted of callAll function and callSome function to display the result to output for the last iteration. The result shows scalable performance using MASS C++ library.
5.2 Usability

MASS library is much easier to use compared to MPI/OpenMP. Figure 10 shows the code snippet of the same matrix application coded using MPI and MASS library. The highlighted region shows the steps to done by the user while using MPI and MASS libraries respectively.

![Figure 10: a. Code Snippet using MPI](image1.png) ![Figure 10: b. Code snippet using MASS library](image2.png)

The steps to develop spatial simulation program and any program using MASS are as follows. First, create a class that inherits from Place. Second create call Methods within the classes and assign them functionIDs. Third, define main function in which the simulation will be run. In the main method, simulation starts with four parameters which includes dimension of the simulation space, size of the simulation space, max time to finish the simulation and interval on which the results are displayed to user console. Then MASS library is invoked, Places object is instantiated and desired operation is performed on each Place object using Places.callAll(). Thus MASS library provides automatic parallelization and synchronization thereby allowing the programmer to focus on the application rather than focusing on parallelization techniques.
The same application can be coded using combination of MPI and OpenMP to take advantage of multi-process and multi-core CPUs on the cluster. However, the array representing the simulation space must be created and distributed to multiple computing nodes, by the application developer by using several MPI calls. To utilize the multi-core CPUs the application developer has to carefully identify parallel regions and critical regions to avoid performance degradation. The differences in usability of MASS and MPI libraries are summarized in Table 2.

<table>
<thead>
<tr>
<th><strong>MPI/OpenMP</strong></th>
<th><strong>MASS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>User has to distribute the simulation space among process and keep track of the mapping.</td>
<td>User is process/thread unaware. Library takes care of mapping and synchronization</td>
</tr>
<tr>
<td>User has to call MPI_send() and MPI_Recv() in proper order to avoid deadlocks, in case data is to be accessed from the neighboring node.</td>
<td>Library takes care of communication and synchronization</td>
</tr>
<tr>
<td>MPI/OpenMP allows performance tuning by the application programmer. But requires user to have a certain degree of understanding of parallel programming concepts.</td>
<td>MASS library provides simple interface and provides automatic parallelization to utilize multiple multicore computing nodes. But does not allow user to optimize the performance beyond what the library does.</td>
</tr>
</tbody>
</table>

Table 2: Usability comparison between MPI/OpenMP and MASS library

### 6.0 Conclusion and Future Work:

MASS library is a part of sensor grid project. In this project, I have implemented the “Places” functionalities of the MASS library in C++. I have also analyzed the performance and usability
metrics for this implementation using a sample application using MASS C++ library. Performance analysis shows scalability of applications using MASS C++ library. However, there still scope for further fine tuning of the library. For example: One such optimization would be to incorporate overlapping of computation and communication to improve performance. Future work in this project will be to test MASS C++ with more benchmark applications, test exchangeHelper functionality, test with more number of computing nodes, implementation of “Agents” functionality.
Bibliography


