# Language and Debugging Support for Multi-Agent and Spatial Simulation

Niko Simonson  Sean Wessels  Munehiro Fukuda[*]

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011

## Abstract

*The MASS (Multi-Agent Spacial Simulation) library facilitates parallelization of applications that are viewed as interaction among up to millions of agents behaving over a shared virtual space and that are thus fitted to simulation of ecological, social, and physical mechanisms. The library invokes user-defined functions of all agents and array elements as well as exchanges data among them in parallel. The key to success of this library implementation is to accelerate function invocation with preprocessor-generated code and to facilitate an application debugging environment. This paper presents the design strategy, implementation, and usability of the MASS library preprocessor and debugger.*

## 1 Introduction

Multi-agent individual-based models view computation as interaction among agents and individuals, each autonomously behaving in a shared simulation environment. They have been used for years to simulate ecological, social, and physical mechanisms that are generally difficult only with mathematical formulae. To parallelize these models, we are developing the MASS (Multi-Agent Spatial Simulation) library that updates the status of all objects at once with the *callAll* method and exchanges data among all objects at once with *exchangeAll* method. These methods are attributed as (1) one-sided parallel communication from the main function to all array elements and (2) one-sided parallel communication from each element. Therefore, MASS benefits not only multi-agent models but also data-intensive computation with its parallelization.

We implemented MASS in Java from the viewpoint of its widely used and convenient graphics features. However, due to Java's nature as well as the multi-agents' behavioral complexity, MASS encounters the following four challenges: (1) parallelization is killed by the slow Java reflection that is used to identify a user function called from *callAll/exchangeAll*; (2) *exchangeAll* incurs substantial communication overhead if applied to computationally fine-grained elements; (3) a programmer needs to check inter-element communication flow at an application level; and (4) agent migration is difficult to keep track of at a user level.

To address these problems, we have developed a language preprocessor and GUI-based debugger for the MASS library. The preprocessor inserts additional code in a given application for calling a user function from *callAll/exchangeAll* without using Java reflection and for transferring multi-element data in bulk. The debugger runs between a user application and the underlying MASS library to capture all the library calls so that it graphically shows each object's status, monitors inter-object communication, keeps track of agent migration, and stops/resumes the user program at a break point.

This paper describes the preprocessor-assisted MASS performance improvement and library extension, presents the features and internal design of the MASS debugger, and demonstrates the uniqueness and usability of these software tools in comparison with related work.

## 2 MASS Library

### 2.1 Execution Model

*Places* and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *place* objects. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place* with array indices, and interact with other *agent* objects as well as multiple *place*s.

As shown in Figure 1, parallelization with the MASS library uses a set of multithreaded communicating processes

[*]Corresponding author. Email: mfukuda@u.washington.edu, Phone: 1-425-352-3459, Fax: 1-425-352-5216

that are forked over a cluster and are connected to each other through ssh-tunneled TCP links. The library spawns the same number of threads as the number of CPU cores per node. Those threads take charge of method call and information exchange among *place*s and *agent*s in parallel.

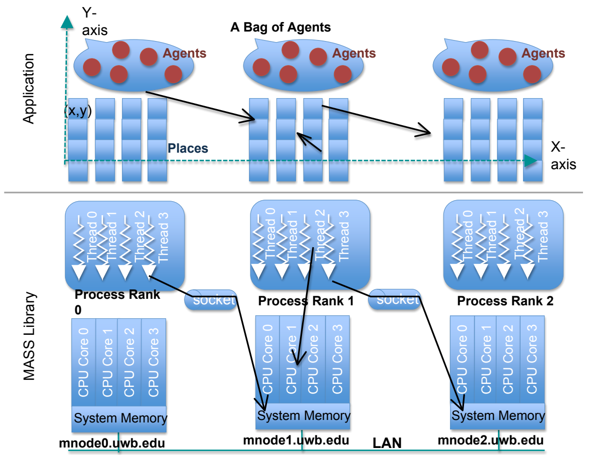


**Figure 1. Parallel execution with MASS library**

## 2.2 Library Specification

A user designs behaviors of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively, populates them through the *Places* and *Agents* classes, and performs their computation through the following methods.

***Places Class***

- ***public Places( int handle, [String primitive,] String className, Object argument, int... size )*** instantiates a shared array with *size* from *className* or a *primitive* data type as passing an *argument* to the *className* constructor. This array receives a user-given *handle*.
- ***public Object[] callAll( String functionName, Object[] arguments )*** calls the method specified with *functionName* of all array elements as passing *arguments[i]* to element[i], and receives a return value from it into *Object[i]*. Calls are performed in parallel among multi-processes/threads. In case of a multi-dimensional array, *i* is considered as the index when the array is flattened to a single dimension.
- ***public Object[] callSome( String functionName, Object[] argument, int... index )*** calls a given method of one or more selected array elements. If *index[i]* is not negative, it indexes a particular element, a row, or a column. If *index[i]* is negative, say $-x$, it indexes every $x^th$ element. Calls are performed in parallel.
- ***public void exchangeAll( int handle, String functionName, Vector<int[]> destinations)*** calls from all elements a given method of all destination elements, each indexed with a different *Vector* element. Each vector element, say *destination[]* is an array of integers where *destination[i]* includes a relative index (or a distance) on the coordinate $i$ from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.
- ***public void exchangeSome( int handle, String functionName, Vector<int[]> destinations, int... index)*** calls each of the elements indexed with *index[]*. The rest of the specification is the same as *exchangeAll()*.

***Agents Class***

- ***public Agents( int handle, String className, Object argument, Places places, int population )*** instantiates a set of agents from *className*, passes the *argument* to their constructor, associates them with a given *Places* matrix, and distributes them over these places, based on *map()* that is defined within the *Agent* class.
- ***public void manageAll()*** updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, *kill()*, *sleep()*, *wakeup()*, and *wakeupAll()*. These methods are defined in the *Agent* base class and may be invoked from other functions through *callAll()* and *exchangeAll()*.

## 2.3 Design Issues

From the user viewpoint, *Places* and *Agents* are theoretically considered as an array or a collection of *Place* and *Agent* objects respectively. However, their underlying implementation is not so simple in order to not only serve as a general simulation framework but also to work over a distributed-memory cluster. We need to address the two design challenges below:

**Language Issues:** Unless a user implements base methods of the *Place* or *Agent* class, *Places* and *Agents* do not know any methods of a user-defined class. This in turn means that the *callAll/Some* and *exchangeAll/Some* methods cannot invoke a user function simply through object casting. Instead we need to use Java reflection to interrogate a user-defined class. The problem is that the reflection works one order slower than a direct function call in general. This slow performance kills parallelization where MASS calls the same function of all objects at once. To pursue both naming flexibility and high-speed invocation of user functions, we design a Java preprocessor that inserts additional code to match the names of user functions defined in MASS methods and the actual function bodies to invoke, so that the library calls user functions without the reflection.

**Debugging Issues:** Since *Places* and *Agents* may be allocated over a distributed-memory cluster, the status and execution of their elements is not always visible and traceable where the main program is running. For debugging purposes, users are responsible to collect remote element status by manually inserting combinations of *callAll/Some* and *exchangeAll/Some* methods as well as adding additional graphics code into their application programs. In particular, it is tedious work for application designers to keep track of migrating *Agent* objects over different computing nodes. We address these debugging issues by designing a wrapper that covers the original MASS library and facilitates GUI-based debugging features.

The next two sections explain these solutions.

## 3 Preprocessor Design

### 3.1 Library Extension

We extend MASS to avoid Java reflection and to accelerate message exchange among *Place* objects as follows:

**Eliminating Java Reflection:** Given a function name in the MASS methods, we need to quickly identify its function body to invoke. A trivial but naive idea is to store user function names in a symbol table at compile time and thereafter to compare each symbol table entry with a function name specified in a MASS method each time it is invoked at run time. The problem is repetitive string comparisons that may weigh more than actual computation at each *place* or *agent*. Instead we use integer comparisons where user function names found in MASS methods receive a different integer, (i.e., a function id) at compile time and a MASS method invokes the user function corresponding to a given function id. Figure 2 shows preprocessor-generated example code that jumps from two MASS methods to a different user function: two user function names such as "exchangeArray" and "putArray" in *exchangeAll()* and *callAll()* (lines 1-2) receive a function ID respectively (lines 6-7); the original *exchangeAll()* and *callAll()* calls the preprocessor-generated *callMethod()* (line 12); and the control branches off to the corresponding user function, based on the function id (lines 14-15).

**Exchanging Each Place's Boundary Information:** The MASS library originally assumes that a *Place* object is used as an individual element of a distributed array. However, the cost for *exchangeAll/Some* is substantial to fine-grained computation at each *Place* object, because data exchange generally takes place with multiple neighbors. Therefore, a user wants to include a collection of array elements in each *Place* object that then exchanges its boundary elements (or shadow elements) in fewer packets with neighbors as shown in Figure 3. This reduces the frequency of communication over an entire array while increasing the

```
1    // the original MASS methods
2    myPlaces.exchangeAll(h, exchangeArray, neighbors);
3    myPlaces.callAll(putArray, args);
4
5    // preprocessor-generated code to jump user functions
6    public static final int exchangeArrayP_ = 0;
7    public static final int putArrayP_      = 1;
8
9    myPlaces.exchangeAll(h, exchangeArrayP_, neighbors);
10   myPlaces.callAll(putArrayP_, arggs);
11
12   public Object callMethod(int funcId, Object args) {
13      switch(funcId) {
14         case exhangeArrayP_: return exchageArray(args);
15         case putArrayP_:     return putArray(args);
16      }
17      return null;
18   }
```

**Figure 2. Two MASS methods and their preprocessor-generated code**

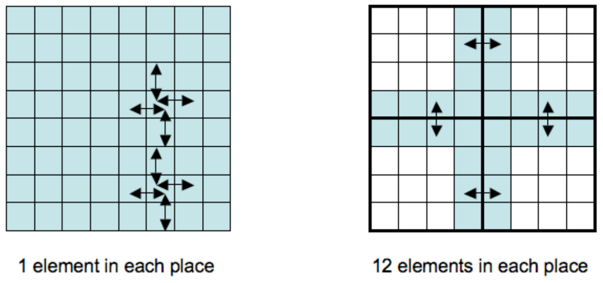


**Figure 3. Communication among neighboring Place objects**

computation amount per *Place*. A new MASS library function named *exchangeBulk* exchanges such boundary information among neighboring *Place*s. As shown in Figure 4, we achieve it by translating *exchangeBulk* into a combination of *exchangeAll* and *callAll* (lines 6-7): the former calls a given function of all neighboring *Place* objects to retrieve their boundary information, and the latter put the information into the local *Place*'s boundary space. The MASS preprocessor assumes that a user defines *exchangeArray* and *putArray*, where *Array* is a user-defined *Place* object, each actually achieving data retrieval and saving operations. If not, the preprocessor generates simple stub functions (lines 9-15). Thereafter, it converts this pair of *exchangeAll* and *callAll* into those calling the user functions with their function IDs as described in Figure 2.

### 3.2 Design Strategies

We design and implement the MASS preprocessor, based on the following two strategies. First, we use an existing Java compiler-compiler tools: JavaCC and JJTree

```
// A new MASS method to exchange boundary data
myPlaces.exchangeBulk(h, Array, neighbors);

// preprocessor-generated exchange/callAll from
// exchangeBulk
myPlaces.exchangeAll(h, "exchangeArray", neighbors);
myPlaces.callAll(h, "putArray", neighbors);

public Object exchangeArray(Object src) {
   return (Object)Array.getBoundary((int[])src);
}
public Object putArray(Object arg) {
   Array.putBoundary(inMessages);
   return null;
}
```

**Figure 4. exchangeBulk and its preprocessor generated code**

for parsing and optimizing MASS user programs. Second, we carry out two passes of MASS program translation: pass 1 converts *exchangeBulk* into a combination of *exchangeAll/callAll*, and pass 2 generates additional code to call a user function from a MASS library method with its function ID. The following details an implementation of our MASS preprocessor.

## 3.3 Implementation

The preprocessor performs its optimizations by running the input code through a Java parser. The parser emits tokens in response to the input code. Actions are taken on specific tokens to check conditions, set flags, and modify output. A grammar defines a roughly correct version of Java. It has been modified to create Abstract Syntax trees. From the grammar, a parser is generated (as UnparseVisitor.java) which by default will output any input which matches the Java language as defined by the grammar. As shown in Figure 5, the parse methods can be overwritten with MASSOptimizer, ExchangeBulkOptimizer, or ReflectionOptimizer to perform MASS optimizations. For example, when parsing a *MethodDeclaration* token, a flag will be set to indicate that a new method is being parsed and that a new scope must be placed on the stack. Subsequently, if a *ResultType* token is parsed while the *MethodDeclaration* flag remains set, the return type of the parsed method can be recorded. These optimizers in Figure 5 set flags in their *visit()* methods and implement the logic to respond to those conditions in their *find(Token)* methods.

The preprocessor has been tested on some MASS programs including two-dimensional wave simulation (Wave2D) and three-dimensional computational fluid dynamics (CFD). The verification and performance evaluation has been conducted by comparing manually-translated versus preprocessor-generated code. Figure 6 demonstrated the competitive performance of preprocessor-generated code in

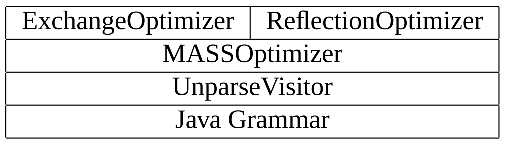


**Figure 5. MASS preprocessor implementation**

| Code | Total | exchangeAll | callAll |
|---|---|---|---|
| Manual | 9730.5 ms | 4605.25 ms | 1505.5 ms |
| Preprocessor | 9785 ms | 4366.25 ms | 1705.75 ms |

**Wave2D**

| Code | Total |
|---|---|
| Manual | 9730.5 ms |
| Preprocessor | 9785 ms |

**CFD**

**Figure 6. Preprocessor-generated code execution**

Wave2D and CFD when running the code four times on a 64-bit 2.27GHz Intel Core.

## 4 Debugger Design

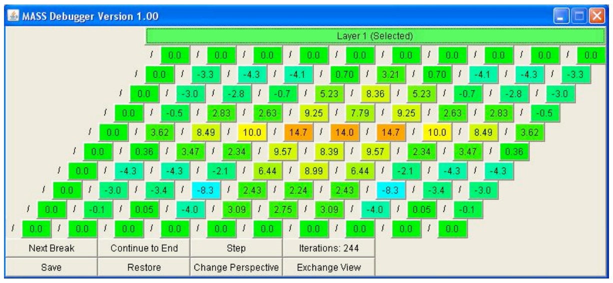
A simple debugging program was implemented to assist MASS developers. Currently, the debugger uses the multi-threaded Java version of MASS. It allows users to view a logical arrangement of their computational spaces' values through a 2D or 3D graphical view.

### 4.1 Debugging Features

The basic objective of the debugger is to display the contents of computational nodes in a human-understandable format, as shown in Figure 7. Its features are designed to support this goal:

- Displays results in a flat view or hawk's eye view.
- Opens windows to display additional dimensionality.
- Allows debugging in code or in GUI.
- Sets break points and defines iteration points.
- Advances to next break point or by iteration.
- Shows communication between logical nodes.
- Saves and restores computational values.

The developer can set break points in program iteration. This differs from traditional debugging code break points, and is more akin to setting break points based on variable values. However, the developer can specify when in the driving code that an iteration occurs, allowing a more fine-grained approach than might be initially apparent.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | -3.3 | -4.3 | -4.1 | 0.70 | 3.21 | 0.70 | -4.1 | -4.3 | -3.3 |
| 0.0 | -3.0 | -2.8 | -0.7 | 5.23 | 8.36 | 5.23 | -0.7 | -2.8 | -3.0 |
| 0.0 | -0.5 | 2.83 | 2.63 | 9.25 | 7.79 | 9.25 | 2.63 | 2.83 | -0.5 |
| 0.0 | 3.62 | 8.49 | 10.0 | 14.7 | 14.0 | 14.7 | 10.0 | 8.49 | 3.62 |
| 0.0 | 0.36 | 3.47 | 2.34 | 9.57 | 8.39 | 9.57 | 2.34 | 3.47 | 0.36 |
| 0.0 | -4.3 | -4.3 | -2.1 | 6.44 | 8.99 | 6.44 | -2.1 | -4.3 | -4.3 |
| 0.0 | -3.0 | -3.4 | -8.3 | 2.43 | 2.24 | 2.43 | -8.3 | -3.4 | -3.0 |
| 0.0 | -0.1 | 0.05 | -4.0 | 3.09 | 2.75 | 3.09 | -4.0 | 0.05 | -0.1 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

**Figure 7. Debugger GUI's hawk's eye view of computational node values.**

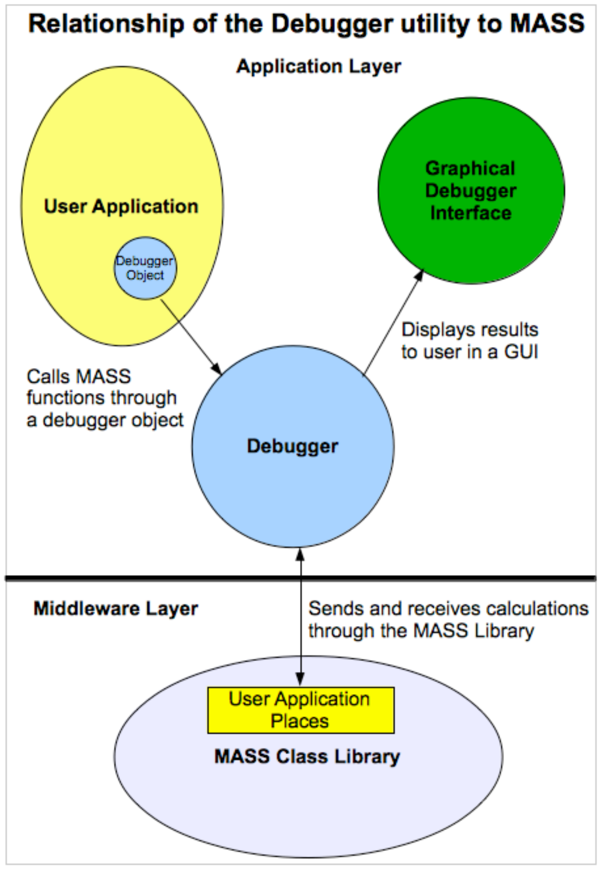

Relationship of the Debugger utility to MASS
Application Layer
User Application
Debugger Object
Graphical Debugger Interface
Displays results to user in a GUI
Calls MASS functions through a debugger object
Debugger
Middleware Layer
Sends and receives calculations through the MASS Library
User Application Places
MASS Class Library

**Figure 8. Interaction between the debugger, a user application, and MASS.**

## 4.2 Design Strategies

Design of the debugger followed a twofold strategy. The debugger wraps functions of the MASS class library as shown in Figure 8, and development is driven by the need for features to show information in terms of the MASS application developer's perspective. The development of the debugger relied on an iterative approach that focused on design, planning, and prioritization of features. A basic goal is to implement features as simply as possible. Early in the process it became apparent that there were two challenges: accessing MASS and creating the graphical interface. Comparing the two, using MASS was simpler than creating the graphics, but had yet to be thoroughly explored. The debugger is the first application to demonstrate that multiple classes can utilize MASS concurrently.

## 4.3 Implementation

The debugger program exists separately from the MASS library. Therefore, it is not affected by changes in the MASS implementation that don't affect MASS function signatures. In effect, it serves as an intermediary layer that wraps MASS functions. In addition to the debugger class, there are classes for graphical control objects: the display windows and buttons. When data is returned from the MASS classes, in addition to returning the data to the user, the debugger also instantiates graphical objects using Java's awt classes, and displays the results. The organization of the results are based on the user-defined sizes of the arrays that are passed to MASS when it is initialized. In this context, the implementation of the debugging features only has to use simple private counters and Booleans to keep track of iterations and breakpoints. When a breakpoint is hit, further function calls to MASS are suspended. Since all MASS results are returned in single arrays, there is no latency in updating individual node results from the debugger or user perspective. Depiction of exchange results are more difficult, because MASS does not provide return values for them. The user provides a vector of relative node coordinates where data is exchanged. The debugger applies the vector to each computational node to determine the coordinates where the data exchange occurred, and shows the last computation (from a MASS *CallAll* function) as the data that is sent. Checkpoints take the return values from a computation and store them in a file. The file is read back into an array of objects when the checkpoint is restored, and passed to MASS. It is important to note that this method of restoring a checkpoint is only valid if the user's places do not rely on local resources, such as a local system clock, for their calculations. The debugger offers three general types of public methods, as shown in the sample code of a simple driver application in Figure 9:

- MASS-equivalent functions with identical signatures
- MASS-equivalent functions with extended signatures for the debugger
- Debugger-only functions, such as setting break points.

The MASS functions with extended signatures combine MASS-equivalent functions with debugger functions, trading fewer function calls for ones with more parameters. Control buttons also implement debugger functions for features such as advancing one iteration or continuing to the next break point while the user's program is running.

```
// Initialization of MASS through debugger
debugger.degubInit(totalSize, totalDimensions,
      ''DebuggerDriver'', threads, 0, 999);

// Pure debugger functions
debugger.setTotalIterations(iterations);
debugger.setBreakPoint(20);
debugger.setStopOnBreakPoint(true);

// Use of MASS through debugger
while(!debugger.isFinished()) {
   // MASS-equivalent function called through
   // debugger (shows results in GUI by default)
   debugger.debugCallAll(0, (Object[])null);

   // ...with iteration set separately
   debugger.iterate();

   // MASS function with debugger parameters
   // (ticks iterator shows results in GUI)
   debugger.debugIterateCallAll(0, (Object[])null,
      true);
}

// end debugger operations and clean up graphics
debugger.finish();
```

**Figure 9. Code using debugger; breakpoint set for every 20 iterations**

## 5 Related Work

### 5.1 Preprocessors

Our MASS library and preprocessor design involves library-assisted parallel execution, preprocessor-assisted code generation, and code manipulation. Those techniques are found in the following four language systems.

Parallel Java Library promotes hybrid SMP cluster programming in Java by combining multithreaded programming constructs and MPI-based message-passing functions [4]. MASS and Parallel Java libraries both take a similar approach in hiding all underlying parallelization work with their Java library classes and methods. However, the major difference is that MASS does not distinguish shared and distributed memory but gives a consistent view of multi-agents running on a shared array regardless of underlying memory architectures.

Extensible PreProcessor (EPP) defines plug-ins for generating tiny data-parallel Java code [3], where the special modifier *"parallel"* given to a Java class generates additional multithreading code in its *run()* method that handles all data with virtual processors in parallel. Although MASS and EPP use a preprocessor approach for parallelization, EPP focuses on multithreading, whereas the MASS preprocessor extends its scope to hybrid SMP cluster computing.

MPIPP is another preprocessor tool that converts a user-defined data structure into MPI-derived data types [6]. It is similar that our *exchangeBulk()* function converts a certain range of boundary array elements to *Place.outMessage* as well as *Place.inMessages* back to the original elements. However, our MASS preprocessor is different in generating *get()* and *put()* methods to automate entire boundary-to-boundary element transfers.

Javassist facilitates a compiler-assisted Java bytecode manipulation that defines new classes, freezes existing classes, and customizes class members [1]. Therefore, Javassist can works as another option to optimize the *exchangeBulk()* function and to match user function names in the MASS library and their actual function bodies, by directly manipulating a user program. However, it would be the same amount of work required if our preprocessor were redesigned to introspect and manipulate all MASS keywords with Javassist.

### 5.2 Debuggers

The MASS debugger is a framework-oriented, library-based, visualization-focused, and data-parallel application debugger. In these categories, we found similarities to and differences from the following four products.

Hadoop and the MASS debugger are both framework-oriented debugging utilities. Hadoop uses a Java class based on the JUnit3 test case and performs testing using a virtual map cluster [8]. The MASS debugger debugs through the actual execution of the MASS program, as MASS controls where its computational spaces run. Hadoop Test Case output is console output, shown through the user's IDE. MASS debugger output uses its own graphical interface.

MPI Debugging Interface is used to provide the conceptual message passing state of the program [2]. The debugger implementation studies the communicator queues; the MASS equivalent of these are the parameters of its major functions. Both applications conclude that accessing the library class must be led by the debugger. Specific interfaces to display data are not implemented by the MPI debugger in contrast to the MASS debugger's GUI.

TotalView is a feature-heavy, commercial debugging interface designed for distributed software [7]. TotalView, like MASS, makes use of a GUI for data visualization but lacks concurrent display of values. The TotalView software is more sophisticated than MASS but also more complicated, and is not free. MASS, in contrast, is geared specifically to MASS users and is currently free. Using TotalView with MASS would strip away all the abstraction that the

MASS library is providing.

Global Arrays are a means to solve data-parallel jobs, and intersects with MASS's application areas [5]. It gives the user very fine-grained control over the data objects it uses, but at the cost of a great deal of programming complexity in implementation. In contrast, MASS aims explicitly to simplify and abstract away complexity. Global Arrays do not in themselves provide explicit debugging tools; while the MASS debugger is an extension of MASS that does not currently exist in Global Arrays.

In summary, we believe that our design strategy for the MASS preprocessor and debugger fits user requirements for multi-agent spatial simulation.

## 6 Future Work

### 6.1 Preprocessor

At present our preprocessor has the following limitations and issues: (1) MASS applications to be processed should not contain a method named "*callMethod*" or use a trailing underscore ("_") as part of a method name; (2) all methods to be called from *callAll/Some* or *exchangeAll/Some* must accept the same set of parameters in the same order; (3) user-owned non-MASS methods should not have names identical to MASS methods; and (4) MASS variables may not be recognized if they are cast or assigned to other classes at runtime. We believe that these limitations are acceptable.

As our future work items, we are developing C, C++, and CUDA-C versions of the MASS library. Since C and C++ allow programmers to use dynamic linking and function pointers, we do not see any necessity of developing a MASS preprocessor for them. However, CUDA-C has its own extensions to C. Although these extensions are quite unique to GPUs, we are planning to develop a preprocessor that assists C programmers in running MASS applications on GPUs. Our ultimate goal is to facilitate a Java or C++ version of the MASS library for GPU computation through a cascading code translation to CUDA-C.

### 6.2 Debugger

Currently, the MASS debugger focuses on the step-by-step display of computational node data and communication. The limitations include: (1) agents and their migration are not displayed; (2) the interface is still only a proof-of-concept design; (3) break points are not based on computational node values; and (4) analysis and step-through of the developer code itself is not implemented.

Implementation of agent status and migration view is the immediate next step in terms of future development. The interface must be improved for the overall user experience. Allowing break points based on computational node values will give the user more flexibility. However, the goal of the MASS debugger is to provide an easy way to view MASS computations, not to become a commercial debugger with a full feature set. Consequently, certain debugger features: the ability to step through code and to set break points in the code itself, are not prioritized for future development.

## 7 Conclusions

The MASS library eases parallelization of multi-agent individual-based models as well as data-intensive applications. In this paper, we analyzed the current issues in code development and execution with the library, addressed them with our preprocessor approach and debugger design. The MASS library and tools will be made available upon an email request sent to *dslab@uw.edu*.

## References

[1] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering - GPCE'03*, volume LNCS 2830, pages 364–376, Erfurt, Germany, September 2003. Springer-Verlag.

[2] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *Proc. of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting - PVMMPI'99*, volume LNCS 1697, pages 51–58, Barcelona, Spain, September 1999. Springer.

[3] Y. Ichisugi and Y. Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Proc. of the Scientific Computing in Object-Oriented Parallel Environments ISCOPE'97*, volume LNCS 1343, pages 153–163, Marina del Rey, CA, December 1997. Springer-Verlag.

[4] A. Kaminsky. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *Proc. 21st IEEE Int'l Parallel and Distributed Processing Symposium - IPDPS*, pages 1–8, Long Beach, CA, March 2007. IEEE-CS.

[5] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, and D. Chavarria. The global arrays user manual. Technical report number pnnl-13130, Pacific Northwest National Laboratory, Richland, WA, November 2010.

[6] E. Renault and C. Parrot. MPI pre-processor: generating MPI derived datatypes from C datatypes automatically. In *Proc. 2006 Int'l Conf. on Parallel Processing Workshops*, pages 248–256, Columbus, OH, September 2006. IEEE CS.

[7] Rogue Wave Software, Inc. Totalview. User Guide Version 8.9.2, Boulder, Colorado, November 2011.

[8] J. Venner. *Pro Hadoop*, chapter 7, Unit Testing and Debugging, pages 207–237. Apress, New York, NY, 2009.