A Multi-Process Library for Multi-Agent and Spatial Simulation *

John Emau Timothy Chuang Munehiro Fukuda†
Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011

Abstract

Integrating sensor networks in cloud computing gives new opportunities of using as many cloud-compute nodes as necessary to analyze real-time sensor data on the fly. However, most cloud services for parallelization such as OpenMP, MPI, and MapReduce are not always fitted to on-the-fly sensor-data analyses that are implemented as model-based entity-based, and multi-agent simulations. To address this semantic gap between analyzing algorithms and their actual implementations, we are developing MASS: a library for multi-agent spatial simulation that 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. We are currently implementing a multi-process and a multi-threaded version of the MASS library, both to be combined in a single version in the near future. This paper focuses on an implementation and preliminary performance of the multi-process version.

1 Introduction

The emergent dissemination of wireless sensor networks and the recent popularity of could computing have brought new opportunities of sensor-cloud integration [4] that will facilitate on-the-fly analysis, simulation, and prediction of physical and environmental conditions by feeding real-time sensor data to cloud jobs. For instance in agriculture, frost protection need to predict the overnight transition of orchard air temperature, which can be done by sending temperature data to prediction polynomials [10] and artificial neural network [9] running in the cloud. Another example is accurate car navigation that finds the best route to drive through a busy metropolitan area by feeding the current traffic data to traffic simulation models such as [6].

For on-the-fly analysis, these simulation models need to be accelerated with cloud/grid-provided common parallelization tools such as OpenMP, MPI, a hybrid of these two libraries, and MapReduce, all generally suited well to simple task parallelism. However, of concern is a big semantic gap between sensor-data analyzing models and these software tools. Most model-based simulations [1] apply formula-based, spatial, and/or multi-agent models to sensor data, where model designers would prefer to code their algorithms as focusing on each individual simulation entity [3]. Therefore, the designers feel difficulty in mapping their algorithms to the underlying parallelization tools.

Our research goal is to reduce this semantic gap by providing users with MASS, a new parallelization library for multi-agent and spatial simulation that facilitates: (1) individual-centered programming of multi-agents and simulation spaces, each automatically parallelized over the underlying platforms and (2) utilization of a SMP cluster, (i.e. composed of a collection of communicating multithreaded processes). MASS 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 element or agent, and all communication is scheduled as periodical data exchanges among those entities using their relative indices. In temperature prediction, an orchard is meshed into two-dimensional array elements over which agents move as air flow. This model unleashes an application from accessing entities in for-loops, and thus eases dynamic allocation of entities to multiple computing nodes and multiple CPU

*This research is being conducted with partial support from UW Provost International Grants, Faculty-led Program

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

cores.

We are currently implementing a multi-process and a multi-threaded version of the MASS library, both to be combined in a single version in the near future. This paper focuses on an implementation and preliminary performance of the multi-process version. The structure of papers is as follows: Section 2 gives a brief overview of the MASS programming model; Section 3 explains our library implementation; Section 4 demonstrates the efficiency of the MASS programmability with a traffic simulation model; Section 5 shows some preliminary performance results; Section 6 differentiates MASS from the related work; and Section 7 clarifies our future milestones.

2 Programming Model and Interface

This section explains the MASS programming model and its major library functions to facilitate parallel simulation.

2.1 Programming 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 is called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *places*. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *places* with array indices (thus as duplicating themselves), and interact with other *agents* and *places*.

As shown in Figure 1, parallelization with the MASS library uses a set of multi-threaded communicating processes 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 that of CPU cores per node[1]. Those threads take charge of method call and information exchange among *places* and *agents* in parallel. *Places* are mapped to threads, whereas *agents* are mapped to processes. Unless a programmer indicates a *place*-allocation algorithm, the MASS library partitions *places* into smaller stripes in vertical, each of which is statically allocated to and executed by a different thread (static scheduling). Contrary to *places*, *agents* are allocated to a different process, based on their proximity to the *places* that this process maintains, and are dynamically executed by multiple threads belonging to the process (dynamic scheduling).

[1]The multi-process version uses only the main thread within each process to run an application

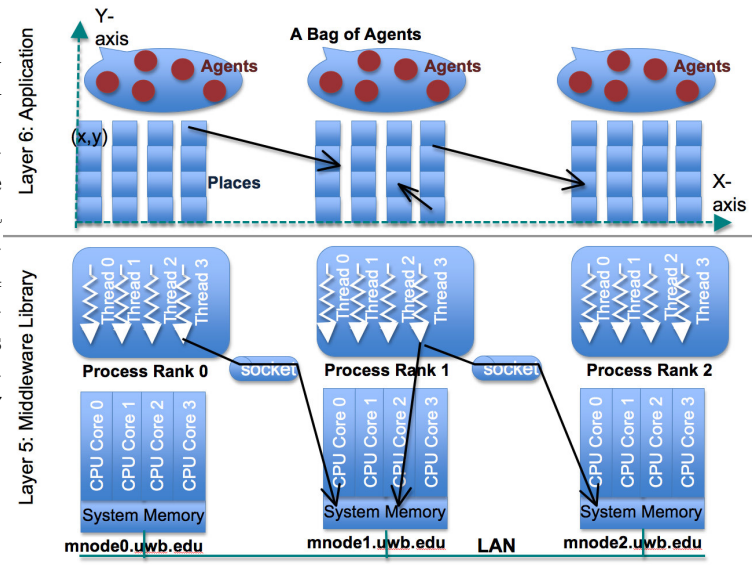

Figure 1. Parallel execution with MASS library

2.2 Programming Interface

A user designs a behavior of a *place* and an *agent* by extending the *Place* and *Agent* respectively. The entities are populated through the *Places* and *Agents* based classes. Actual computation is performed between *MASS.init( )* and *MASS.finish( )*, using the following methods.

MASS Class

- ***public init(String[] args, int nProc, int nThr)*** uses *nProc* processes, each with *nThr* threads.
- ***finish( )*** finishes computation.

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 a non-negative number, it indexes a particular element, a row, or a column. If *index[i]* is a negative number, say $-x$, it indexes every x element. Calls are performed in parallel.
- ***public void exchangeAll(int handle, String functionName, Vector<int[]> destinations)*** calls from each of all elements to 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 from 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 the *map( )* method 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( )*.
- ***public void sortAll(boolean descending)*** sorts agents within each place in the *descending* or *ascending* order of their *key* data member.

3 Library Implementation

In this section we will discuss implementation for the major components of *MASS*. Figure 2 describes the relationship between classes and machines in a cluster by example of the inter-process communication. After a message containing a command with parameters is created it is transported to the slave process through the *mNode* class communication pipeline, established with Java Secure Channel (JSCH). Once received, the slave process parses the message and executes the command.

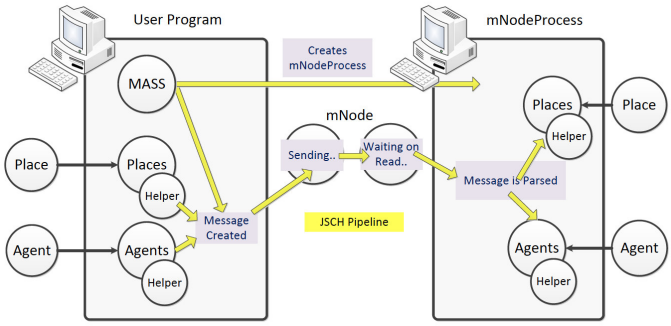

Figure 2. MASS-internal classes over a cluster

3.1 MASS, mNode, and mNodeProcess Implementation

1. **MASS:** is responsible for the construction and deconstruction of a cluster and maintains references to all *Places*, *Agents* and *mNode* instances. *init( )* identifies all remote hosts in the host file and through JSCH, a *mNodeProcess* is launched on the remote hosts, while a *mNode* instance is created locally as a wrapper for the JSCH connection. *MASS* can be initialized on any host running a SSH Server and Java Virtual Machine. Throughout the application lifecycle *MASS* provides useful methods for retrieving any previously created Places or Agents instance.*MASS.finish( )* 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 the connections.
2. **mNode:** works as a wrapper for all inter-process connections and includes several useful methods for process communication. Each *mNode* pair represents a communication pipeline between nodes and a single collection of *mNodes* exists on every process in the cluster. The *mNodes* primary function is to write and read objects passed between process through Java's ObjectInput and ObjectOutput streams that sit ontop of a JSCH or Sockect based connection.
3. **mNodeProcess:** runs as a slave process on a remote host. The *mNodeProcess* facilitates all commands invoked by Process 0, managing program flow on behalf of Process 0. The *mNodeProcess* has a three states in its lifecycle, initiations, running, and deconstruction. During initiation the *mNodeProcess* establishes *mNode* communication pipelines with Process 0 and all other *mNodeProcess* in the cluster. After the *mNodeProcess* has

finished initialization it will sit in an infinite loop waiting for commands from Process 0, execute the command, and return to waiting for the next command. Once the command to terminate is received the *mNodeProcess* closes all connections and self-terminates.

3.2 Places Implementation

Places manages all *place* elements in the simulations space. Every process maintains a collection of *Places* instances, each *Places* instance created by a user program on Process 0 has a corresponding instance on a number of slave processes in the cluster. There are two major methods for *place* manipulation in the *Places* class, *callAll* and *exchangAll*. The *Places* class utilizes Helper threads to assist in the implementation of the *exchangeAll* algorithm.

1. **callAll/Some( ):** Is for issuing commands and sending data to all or some *place* elements. Each *mNodeProcess* receives a method identifier, *place* element indices, and arguments from Process 0, and invokes the given method of the specified *place* elements. The returning value of this method maybe returned to Process 0 after all executions have completed. Process 0 waits to receive the confirmation or return message from all *mNodeProcesses*.
2. **exchangeAll/Some( ):** Is a complete exchange algorithm for exchanging data among *place* elements. The three steps to *exchangeAll* are send request, exchange local data, and process responses. Each *mNodeProcess* receives a method identifier, and a collection of invoking destinations from Process 0. If a destination is outsides the bounds the local process a request is generated to the corresponding slave process (Node). Shown in Figure 3, each process carries out this data exchange asynchronously, using one sender and as many Helper threads as the slave processes. The sender thread sends each Node a message over the secondary connection that specifies what method needs to be invoked by which *place* element in the Node, at which point the sender thread may exchange data among local elements while waiting for the response. Helper threads are each in charge of a different Node and always waits for a new message. Once received the Helper threads process the message by invoking the specified method of the requested *place* element and sends the turning value as a response to the corresponding sender thread. Synchronization is used to coordinate *place* element updates among threads.

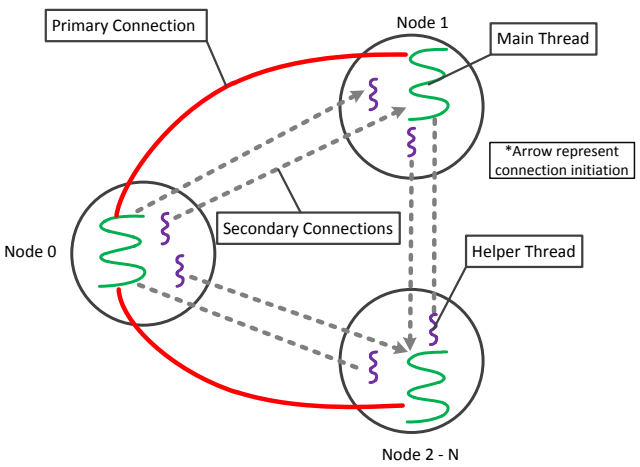

Figure 3. An implementation of exchangeAll

4 Programmability

We consider the MASS programmability, using a simple traffic simulation program whose model and code are presented in Figures 4 and 5 respectively. The program simulates the traffic status of a street map (constructed as a two-dimensional *Places* array) over which a number of cars, (each controlled by an individual agent and denoted as a red dot in the map) drive to a given destination.

The steps to develop this traffic simulation program and any program using MASS are as follows. First, create a class that inherits from the Place and Agent classes, in this example *MeshedStrees* and *Car* respectively. Second, create call methods within the classes and assign them function ids. Lastly, create a driver program that will run simulation.

The simulation starts with three parameters such as a map *size*, a given *nCars* number of cars, and *maxTime* to finish the simulation (lines 8-10). Then, it invokes the MASS library (line 13), creates a *streets* array (line 16), and distributes *car* objects on it (line 19). It also sets up communication links from each array element to its four neighbors (lines 24-27). Thereafter, the program enters a cyclic simulation where each iteration exchanges the traffic status among all array elements (line 32), decides each car's next destination (lines 34-37), and moves it there (line38).

The real-time geographical and traffic data (to be provided by traffic sensor systems) can be reflected to the *map( )* method of both the *Place* and *Agent* classes that create user-defined roads and populates cars at run time.

This example code demonstrates two programming

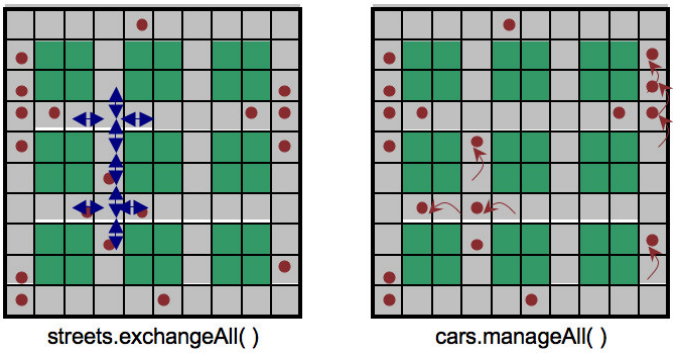

Figure 4. Traffic Simulation Space

advantages with the MASS library. One is clear separation of the simulation scenario from simulation models. The *main( )* function in Figure 5 works as a scenario that introduces necessary models, instantiates/constructs entities, and controls their interaction. This separation allows model designers to focus on each model design. The other advantage is automatic parallelization. The MASS library constructs the *streets* array over multiple computing nodes, populates *cars* on it as maintaining the *cars*-to-*streets* proximity, and calls their functions in parallel. These advantages can be applied to other multi-agent spatial simulations such Schroedinger's wave simulation, Fourier's heat simulation, and artificial societies.

5 Preliminary Performance

Performance evaluation was conducted for *callAll* and *exchangeAll* functions separately to observe their overheads as computational granularity changes from fine-grained to coarse-grained. The computational granularity increases from 10 through to 200 iterations of a floating point multiplication. *exchangAll* was setup with four adjacent neighbors to the north, east, south, and west. The number of participating slave processes (nodes) increases from one to eight. In this evaluation all machines are equiped with similarly configured 3.2Ghz Xerons with 512MB memory connected to Giga-Ethernet.

Figure 6 shows *callAll*'s performance results. The results show us an expected pattern as the number of floating point operations increases so does overall time to completion. We also see an expected drop in time to completion as the number of nodes increases within an individual grouping. Concluding that, the gains in performance are substantial between the number of slave processes as the computation granularity increases.

When analyzing exchangeAll's performance in Figure 7 we notice a pattern emerges as the computation granularity increases. For 10 floating point operations the time to completion increases, for 100 floating point operations, the time to completion tends to be similar regardless of the node count, and for 200 floating point operations, the time to completion decreases as we add nodes. This can be explained by the overhead associated with the *exchangeAll* algorithm. Increasing the number of computing nodes increases the inter-process communication overhead. The advantages of parallel computation must overcome this overhead before performance can be improved with additional nodes. We conclude that performance improvements only exists for coarse computation granularity.

In summary, we see the potential for MASS to improve performance of parallel programs. This is true for algorithms that include both *callAll* at fine computation granularity and *exchangAll* at coarse computation granularity.

```
1  import MASS.*; // Library for Multi-Agent Spatial Simula-
   tion
2  import java.util.Vector;
3
4  // Simulation Scenario
5  public class TrafficSimulation {
6    public static void main( String[] args ) {
7      // validate the arguments
8      int size = Integer.parseInt( args[0] );
9      int nCars = Integer.parseInt( args[1] );
10     int maxTime = Integer.parseInt( args[2] );
11
12     // start MASS
13     MASS.init( args );
14
15     // create streets in a distributed array
16     Places streets = new Places( 1, ``MeshedStrees'',
17                                  null, size, size );
18     // populate agents that behave as cars on the streets
19     Agents cars = new Agents{ 2, ``Car'', null, streets,
20                               nCars}
21
22     // define the four neighbors of each cell
23     Vector<int[]> neighbors = new Vector<int[]>( );
24     int[] north = { 0, -1 }; neighbors.add( north );
25     int[] east  = { 1,  0 }; neighbors.add( east );
26     int[] south = { 0,  1 }; neighbors.add( south );
27     int[] west  = { -1, 0 }; neighbors.add( west );
28
29     now go into a cyclic simulation
30     for ( int time = 0; time < maxTime; time++ ) {
31       // exchange #cars with four neighbors
32       streets.exchangeAll( 1, MeshedStreets.exchange,
33                            neighbors );
34       streets.callAll( MeshedStreets.update );
35
36       move cars to a neighbor if space is available
37       cars.callAll( Car.decideNewPosition );
38       cars.manageAll( );
39     }
40
41     // finish MASS
42     MASS.finish( );
43   }
44 }
```

Figure 5. Traffic Simulation Code

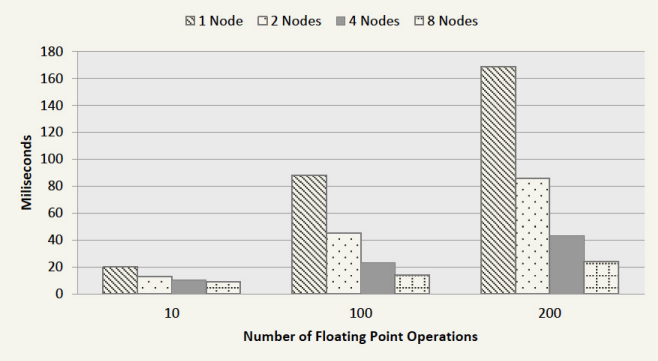

Figure 6. Performance results of callAll

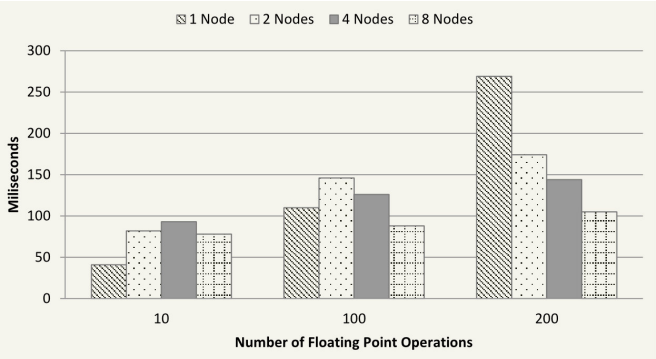

Figure 7. Performance results of exchangeAll

6 Related Work

This section differentiates MASS from its related work in two major aspects: (1) distributed shared arrays and (2) parallel multi-agent simulation environments.

Example systems supporting distributed shared arrays include UPC: Unified Parallel C [11], Co-Array Fortran [8], and GlobalArray [7]. UPC allocates global memory space in the sequential consistency model, which is then shared among multiple threads running even on different computing nodes. Co-Array Fortran allows "so-called" images, (i.e., different execution entities including ranks, processes, and threads) to co-allocate, to perform one-sided operations onto, and to synchronize on shared arrays. GlobalArray facilitates not only one-sided but also collective operations onto global arrays that are shared among different MPI ranks. Although MASS has a similarity as these language-based runtime systems in 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 system-predefined operations. In particular, exchangeAll/Some operations in MASS do not invoke a method call to each array element but rather invoke a parallel call from each to other elements, (in other words, inter-element parallel calls).

Most multi-agent simulation environments such as PDES-MAS [5] and MACE3J [2] focus 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 multicast 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 mapped them over a collection of computing nodes for parallelization. MASS is different from them in handling fine-grain reactive agents that sustain a partial view of their entire space and interact with other agents in their neighborhood. Although an array element in MASS can be considered as an interest manager in PDES-MAS and MACE3J, MASS instantiates a large number of array elements, (i.e., interest managers), and define their logical connection with exchangeAll/Some functions.

In summary, MASS is unique in facilitating user-defined inter-element communication in distributed arrays and realizing fine-grain reactive agents, each interacting with others through numerous array elements (i.e., interest managers).

7 Conclusions

The MASS library is intended to facilitate entity-based simulation for on-the-fly sensor-data analysis. In this paper, we demonstrated the programming advantages in using the MASS library for such simulation as well as its competitive performance in parallel execution of fine-to-medium grain computation. Our next step is to extend this multi-process version to a multi-process multi-threaded version to utilize a cluster of multi-core computing nodes.

References

[1] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *The VLDB Journal*, Vol.14(No.4):417–443, November 2005.

[2] L. Gasser and K. Kakugawa. MACE3J: fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proc. of the first international joint conference on Autonomous agents and multiagent systems - AAMAS-2002*, Bologna, Italy, November 2001. ACM.

[3] V. Grimm and S. F. Railsback. *Individual-based Modeling and Ecology*. Princeton Univesity Press, Princeton, NJ, 2005.

[4] M. M. Hassan, B. Song, and E.-N. Huh. A framework of sensor-cloud integration opportunities and challenges. In *Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 618–626, Suwon, Korea, January 2009. ACM.

[5] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. *Proceedings of the IEEE*, Vol.89(No.2):174–186, January 2001.

[6] Multi-Agent Transport Simulation - MATSim. http://www.matsim.org/.

[7] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, Vol.20(No.2):203–231, 2006.

[8] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, Vol.17(No.2):1–31, August 1998.

[9] B. A. Smith, G. Hoogenboom, and R. W. McClendon. Artificial neural networks for automated year-round temperature prediction. *Computers and Electronics in Agriculture*, Vol.68(Issue.1):52–61, August 2009.

[10] UC Davis Biometeorology Program - Frost Protection. http://biomet.ucdavis.edu/frost-protection.html.

[11] UPC Consortium. UPC Language. Language specifications 1.2, The High Performance Computing Laboratory, George Washington University, http://upc.gwu.edu/, May 2005.