**SANDIA REPORT**
SAND2008-6172
Unlimited Release
Printed August 2008

# Parallel Computing in Enterprise Modeling

Rob Armstrong, Benjamin Allan, Michael Goldsby, Zachary Heath, Max Shneider, Keith Vanderveen

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-6172
Unlimited Release
Printed August 2008

# Parallel Computing in Enterprise Modeling

Robert C. Armstrong, 8961, rob@sandia.gov
Benjamin A. Allan, 8961, baallan@sandia.gov
Michael E. Goldsby, 8116, megolds@sandia.gov
Zach Heath, 8964, zheath@sandia.gov
Max Shneider, 8962 msshnei@sandia.gov
Jaideep Ray, 8964 jaray@sandia.gov
Keith B. Vanderveen, 8116, kbvande@sandia.gov

**Abstract**

This report presents the results of our efforts to applly high-performance computing to entity-based simulations with a multi-use plugin for parallel computing. We use the term 'Entity-based simulation' to describe a class of simulaton which includes both discrete event simulation and agent based simulation. What simulations of this class share, and what differs from more traditional models, is that the result sought is emergent from a large number of contributing entities. Logistic, economic and social simulations are members of this class where things or people are organized or self-organize to produce a solution. Entity-based problems never have an *a priori* ergodic principle that will greatly simplify calculations. Because the results of entity-based simulations can only be realized at scale, scalable computing is *de rigueur* for large problems. Having said that, the absence of a spatial organizing principal makes the decomposition of the problem onto processors problematic. In addition, practitioners in this domain commonly use the Java programming language which presents its own problems in a high-performance setting. The plugin we have developed, called the Parallel Particle Data Model, overcomes both of these obstacles and is now being used by two Sandia frameworks: the Decision Analysis Center, and the Seldon social simulation facility. While the ability to engage U.S.-sized problems is now available to the Decision Analysis Center, this plugin is central to the success of Seldon. Because Seldon relies on computationally intensive cognitive sub-models, this work is necessary to acheive the scale necessary for realistic results. With the recent upheavals in the financial markets, and the inscrutability of terrorist activity, this simulation domain will likely need a capability with ever greater fidelity. High-performance computing will play an important part in enabling that greater fidelity..

# Acknowledgment

Many thanks go to Eric Parker and the Seldon crew for the opportunity to extend this work to social systems modeling.

# Contents

**Preface** **8**

**Summary** **9**

**1 Introduction** **11**

**2 Related Work** **13**

**3 Architecture** **15**

3.0.1 Data and Particle Fields . . . . . 16

3.0.2 Communication implementation . . . . . 18

**4 Example Simulations** **21**

4.0.3 Spread of disease in a 2-species population . . . . . 21

4.0.3.1 Disease model . . . . . 21

4.0.3.2 Movement model . . . . . 22

4.0.3.3 Disease Model Implementation and Scaling . . . . . 22

4.0.4 Seldon . . . . . 24

4.0.5 Parallel Social Forces . . . . . 27

**5 Codebase** **31**

5.0.6 Build Tool . . . . . 31

5.0.7 Development Environment . . . . . 32

5.0.8 Java and Parallel Computing . . . . . 32

5.0.9 Launching the PPDM . . . . . 34

**6 Integration** **37**

6.0.10 HLA . . . . . 37

6.0.11 RePast . . . . . 39

**7 Improvements/Limitations** **43**

**8 Conclusion** **45**

**References** **46**

## Appendix

**A Publications** **51**

**B Parallel Particle Data Model** **53**

B.1 Introduction . . . . . 54

B.2 Related Work . . . . . 55

B.3 Architecture . . . . . 56

B.4 Example Simulations . . . . . 59

B.5 Integration . . . . . 65

B.6 Conclusion . . . . . 66

# List of Figures


3.1 Box Domain Model . . . . . 16

3.2 Graph Domain Model . . . . . 17

3.3 Visitor Algorithms . . . . . 18

3.4 Particle Algorithm Output . . . . . 18

3.5 Particle Movement . . . . . 19

3.6 Shadow Data Update . . . . . 19

4.1 Disease Strong Scaling . . . . . 23

4.2 Disease Weak Scaling (Cell Size) . . . . . 23

4.3 Disease Weak Scaling (Num Cells) . . . . . 24

4.4 Disease Performance (Cell Size) . . . . . 25

4.5 Disease Performance (Cell Size log/log) . . . . . 25

4.6 Disease Performance (Num Cells) . . . . . 26

4.7 Disease Performance (Num Cells log/log) . . . . . 26

4.8 Seldon Strong Scaling . . . . . 29

4.9 Seldon Weak Scaling . . . . . 29

4.10 Social Forces Example . . . . . 29

5.1 Launch Diagram . . . . . 35

6.1 Single Machine . . . . . 38

6.2 External Machine/Cluster Connection . . . . . 39

6.3 Cluster . . . . . 40

# Preface

The practice of modeling and simulation, or more specifially, enity modeling is as old as digital computers. Because the most common application is complex systems and emergent behavior from a large number of contributing entities, scale is naturally a concern. There is no natural limit for this type of simulation; if a model that tracks every person in a county is relevant, the model that tracks every person in the state, country or world is also relevant. Because emergent behavior from complex systems is, absent other information, dependent on all scales, the larger the system the more accurate the solution. So there is an implicit need for computational methods and algorithms that provide scalability to entity simulations. The purpose of this work is to create a plugin toolkit that brings scalable high-performance computing to agent-based modeling.

At this point (September 2008) we have two major consumers of our technlogy: the Decision Analysis Center (DAC) framework and the Seldon Social Simulation framework. While the problems that justify high-performance computing for DAC have yet to materialize, the results of this work, the Parallel Particle Data Model (PPDM), is nothing less than enabling for Seldon. It is hard to imagine that entity-based modeling will not require the capabilities of high-performance computing as models require more fidelity and thus become more complex. It is our hope that this work will put Sandia in a better position to take advantage of this future.

# Summary

We present the design and performance of a parallel entity simulation framework called the Parallel Particle Data Model (PPDM). Based loosely on a Particle-In-Cell algorithm, the PPDM orchestrates and supports agent-based simulations on a parallel high-performance platform. The PPDM is targeted at social simulation applications and is designed to be portable to a variety of high-performance platforms. In this paper we show that the PPDM performs well for two agent-based simulations on a clustered platform. We hope that this work will form the cornerstone of a reusable toolkit for modeling and simulation.

# Chapter 1

# Introduction

Enterprise modeling, social simulation, system-of-systems war gaming, and related techniques simulate complex environments and systems (hereafter referred to as Human Dynamical Systems (HDS)) and compute observables which manifest themselves primarily as emergent phenomena. To improve fidelity of these HDS systems, increasingly large computations are needed and are accessible only through high-performance computing (HPC). Computer-based simulation of HDS has long been used within the military for war games, to carry out training, assess new technologies, and evaluate tactics. Recently, the Department of Homeland Security (DHS) and other federal agencies charged with disaster preparedness and response have embraced simulation for modeling a variety of complex phenomena including response to attacks by Weapons of Mass Destruction (WMD) and natural disasters. Sandia National Laboratories has developed a number of simulation applications to support different programs examining WMD countermeasures and concepts of operation, defense applications, terrorist networks, and economic consequences from natural disasters or terrorism affecting critical infrastructures. These simulation applications differ in several respects, yet they share an underlying commonality: all concern themselves with modeling complex ”systems of systems” where the system components are discrete (such as people, autonomous land vehicles, or companies) and interact in highly complex ways with other system components. Historically, modeling of HDS has not taken advantage of parallel processing hardware or techniques. There are at least two reasons for this:

1. The phenomena in question was not modeled at a fidelity that would require the extra complication of parallel computing.

2. The history of simulation of HDS has favored developer friendly environments, use of widely available computing hardware and software, and incorporation of sophisticated graphical user interfaces (GUIs) over speed of execution. This emphasis on rapid deliverables portable to a wide variety of platforms led to the use of virtual machine-based languages such as Java, Python, and Tcl over C and C++, because most programmers are much more productive in these languages. Most programming for high performance parallel processor systems has taken place in languages like C, C++, and (historically) FORTRAN because they give the programmer fine-grained control over allocation of memory and other processor resources.

Increasingly, however, HDS must reproduce significant phenomena/effects at credible fidelity while being fast enough to enable human-in-the-loop and timely batch-oriented analysis. Recently

developed requirements for the next generation of social simulation applications have shown that the current capability, based on conventional computing architectures, falls short in both respects, requiring extension to HPC platforms using parallel processing. Also, improvements to Java and other virtual machine-based languages have made them more attractive in the domain of high performance computing.

We have developed components and tools for HDS simulation methods in a parallel HPC setting. In particular, we are addressing the challenges outlined above:

1. Create a portable, scalable, and general software engine for particle simulations. Referred to as the Parallel Particle Data Model (PPDM), it is a facility for multi-use in agent-based HDS simulations.

2. Ensure the PPDM will be compatible with the developer friendly Java language for which Sandia already has a considerable investment. It bridges the gap between this rapid development language and traditional HPC languages like C/C++.

3. Develop and demonstrate this new capability by prototyping a large-scale homeland security simulation (crowd dynamics, disease propagation, etc.). Show the new capability to possible sponsors and gain visibility in the community.

4. Engage external research community through presentations and publications.

5. Identify synergistic R&D efforts within the laboratory and develop those partnerships. Specific leveraging opportunities include (1) prototype use of the PPDM in existing Decision Analysis Center (DAC) applications, and (2) demonstrate PPDM in a high performance parallel Seldon computation.

The principle research objective of this LDRD was to create a code facility for particle calculations processor/data decomposed using geometry, similar to Particle-In-Cell (PIC) algorithms. The original research goals are accomplished and a paper was accepted and presented to the 2nd World Congress on Social Simulation. This code facility has also been integrated into a number of Sandia projects (Seldon, PopulationDAC, DavisDAC, BioDAC and LMSV).

# Chapter 2

# Related Work

Many simulations have been created to study complex Human Dynamical Systems. Two excellent examples originally developed at Los Alamos National Laboratories and now hosted by Virginia Tech's Network Dynamics and Simulation Science Laboratory include TRANSIMS[3] and EpiSims[4]. TRANSIMS uses detailed urban travel data to model transportation networks and can model the effects that changes to those networks will have on traffic. EpiSims uses that same travel data to model disease epidemics and can be used to test public health mitigation strategies. Another example of work in this arena is Generative Social Science, which uses simulations to help discover the underlying dynamics of complex social systems[12]. In this work Epstein makes a hypothesis of the sets of rules that govern a given dynamic social system. An agent-based model (ABM) implementation of those rules is then created and the results are measured against historical data. The closeness with which the generated data matches the historical data gives credibility to the hypothesized rules. During the span of this LDRD Joshua Epstein along with Jon Parker and fellow colleagues at the Brookings Institution developed the Large-Scale Agent Model [26]. This a flexible, large-scale, distributed agent based epidemic model that is able to simulate the spread of a disease on the scale of several hundred million agents using parallel processing. The model is being further refined to allow the simulation of billions of agents.

As the modeling of HDS has grown in popularity, a number of reusable agent-based modeling toolkits have emerged to simplify the work of the developer. RePast[9] is arguably the most popular framework in the ABM community and provides developers with libraries for agent creation, event scheduling, and data charting and visualization. Mason[22] provides similar functionality but focuses more on light-weight models meant to be run many times for Monte Carlo-type studies. While these toolkits can be run in a parallel batch mode on a clustered environment, neither toolkit lends itself for use developing models that run singularly in parallel across a cluster of computers, thus limiting the size and complexity of such models.

The PPDM in its present form operates in a time-stepped fashion. Discrete event simulation (DES) provides a different time management paradigm for simulation. Instead of advancing time in globally synchronous fixed steps, DES advances the local simulation time to the exact time of the next local event. DES scheduling delivers events in timestamp order at all nodes allowing it to achieve performance gains if the simulated events are irregularly spaced in time or space. The greatest successes of DES to date have been in the areas of network simulation and war gaming. Several frameworks and systems have been created to support this. The High Level Architecture (HLA) standard[16] specifies an API for distributed DES that has seen wide use, particularly in

work done for the Department of Defense. An example of parallel DES (PDES) used for social simulation is the SCATTER traffic simulator[28]. PDES scheduling has also provided time management for an agent-based simulation system[19]. Applications using PDES have scaled to over 1500 nodes[15]. In an exceptional scaling exercise, Perumalla[29] ran a test program on 16384 nodes of a Blue Gene supercomputer. Though the PPDM in its present form makes no use of DES concepts, we envision future DES extensions. To our knowledge, no agent-based simulation system in widespread use provides the distributed synchronization needed for true PDES scheduling.

In a work that highly influenced our design, a more traditional particle-in-cell based, time stepped model was used for simulating the evacuation of large crowds in a building environment[30]. In this work, the Social Forces algorithm was implemented such that the simulated geometry and populations were divided up among a cluster of computers. In [31] a similar approach was used for large crowd simulation but made novel use of the PlayStation 3's high performance Cell architecture.

# Chapter 3

# Architecture

In our work modeling various HDS, we identified a great deal of functional commonality in our simulation codes. We also observed that future simulations in the problem domain would be growing in population size and complexity. This led us to create the Parallel Particle Data Model (PPDM), a Java-based, reusable programming toolkit that supports HDS simulations and can scale to a large number of processors.

The HDS simulated by Sandia are generally composed of large populations of agents (people, vehicles, etc.) spread across a geography of segmented locations such as cities, census tracts, or even households and businesses. Each location contains a set of agents that are able to interact with each other as well as with agents in neighboring locations. The agents can move between locations as time evolves. Various modules of the simulation need to update or gather information from the agent populations in different ways based on the phenomena they model. Examples include infection of people due to a biological hazard plume or detection that a car should trigger a radiation sensor alarm.

We designed the PPDM to accommodate this method of interacting with the simulated populations while operating in a distributed environment. The PPDM uses a particle-in-cell (PIC) data structure which divides the simulated geography into cells that contain populations of agents. The PPDM then distributes these cells across a number of processors. Agents directly affect other agents residing in the same or neighboring cells. Non-neighboring particles can interact through special, less efficient mechanisms.

Even though Java is not renowned for its high performance computing capabilities we chose it as the PPDM's implementation language. The primary reasons were to support legacy code, to utilize existing programmer skill sets and to provide ease of development. Evolving improvements to the JVM's performance and extensive open source code libraries, including several new high performance communication libraries, have also increased its attractiveness. The end result is that Java tools can provide a portable and easy to use framework for high performance computing.

To make use of the PPDM, the user must first define the geometry of the problem domain. Currently, we have implemented two types of geometry. A user can specify either an n-d lattice (Figure 3.1) or a graph of cell locations (Figure 3.2). Along with the geometrical structure, the user must define the number of functional patches into which they would like the geometry broken. A patch represents a grouping of cell locations that are guaranteed to be collocated on the same processor. Each processor is allocated a number of patches to process. Modelers write their

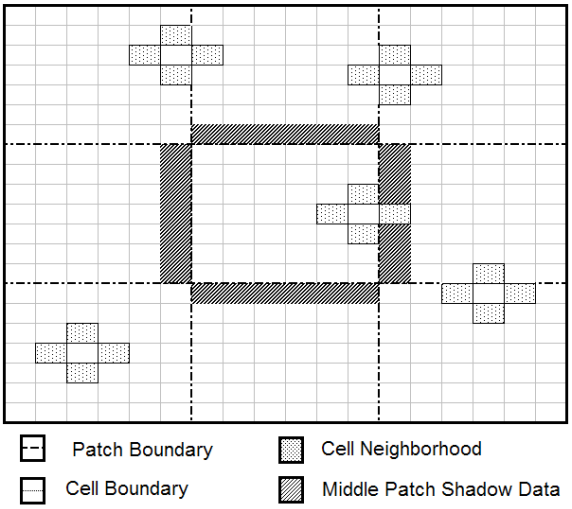


**Figure 3.1.** Box Domain Model

algorithms and agent update code to work on a patch-by-patch or cell-by-cell basis which enables the PPDM to distribute the user's program across many processors.

### 3.0.1 Data and Particle Fields

After specifying the domain geometry, modelers define multiple data and particle fields across this geometry. A data field holds a single unit of data (such as an int or float) for each cell in the domain, while a particle field allocates to each cell in the domain a list-like data structure called a particle set. The particle set data structure allows the modeler to store and organize agents and provides other useful utility functions. In creating a data or particle field, the modeler specifies a locality stencil for that field. This defines each cell's local data neighborhood within the overall defined geometry. In the case of 2-d lattices, a modeler will commonly specify a 4 point stencil in which a cell's neighbors include adjacent cells to the north, east, south, and west. In a graph geometry, on the other hand, the graph structure directly determines cell neighborhoods. The local neighborhood defines a region of shadow data that each patch must make available to the cells it maintains. Shadow data is a read-only copy of neighboring cell's data. After doing a shadow update for a particle field, each patch will have the data for the cells it maintains as well as a read-only copy of data from its neighboring cells that are maintained by other patches. If a patch has neighboring cells that are located on another processor, data communication must take place. The transfer of shadow data enables the simulation to update agents' state using data from their own cell as well as neighboring cells. This method is similar to that used in a parallel implementation of the social forces model [30]. The shadow data construct is optional. If a particular domain does not need to have neighbor cell interactions this feature can be disabled. To summarize, a processors will be allocated a number of patches. Each patch is composed of a number of cells. Each cell contains a number of particles or agents.

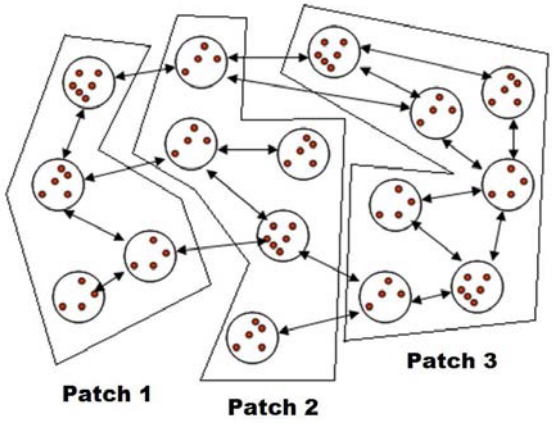


**Figure 3.2.** Graph Domain Model

While processing the agents, modelers also have the ability to schedule the movement of agents from cell to cell. This cell movement can span patches and processors. The PPDM takes care of the actual movement of the agents and transfer of shadow data. If the modeler guarantees that particle movement only occurs between neighboring cells, the PPDM can make optimizations that reduce the amount of communications needed between processors, thereby allowing greater speedup. The PPDM's capabilities hide a lot of parallel processing complexity from the modeler.

To better enable parallel processing, the PPDM limits the modeler's interaction to the data and particle fields by allowing only cell-by-cell or patch-by-patch access to the data. A modeler's algorithms have full access to all agents owned by the patch they process, as well as read access to agents of neighboring cells of that patch. The PPDM's feature set encourages modelers to build their algorithms with locality in mind. This allows the algorithms and data to be more easily and efficiently distributed across multiple processors. From our experience in prior simulations, we consider this restriction acceptable. We also offer various convenient yet less efficient mechanisms to handle global communication between processors.

Several patterns for interacting with the particle populations are available. The most common method of particle access is to submit particle algorithms to the PPDM (Figure 3.3). Particle algorithms are data structures that implement the visitor pattern. When particle algorithms are submitted to the PPDM, the PPDM duplicates them across all processors and schedules them to be executed at regular intervals. The algorithms visit each cell or patch owned by the processors on which they are located and perform their needed operations on the contained agents. The particle algorithms can optionally coordinate with their copies on other processors and send back aggregated data to their parent modules or other subscribing particle algorithms (Figure 3.4).

In our simulations, various modules work together in a decoupled manner by submitting particle algorithms that enforce their desired behavior or area of concern on the agents. This deviates from standard agent-based techniques, which initialize the agents with a given behavior and state and allow the agents to evolve their state. This difference arises because the PPDM grew out of our experience with prior simulations composed of separate modules that ran on their own Java

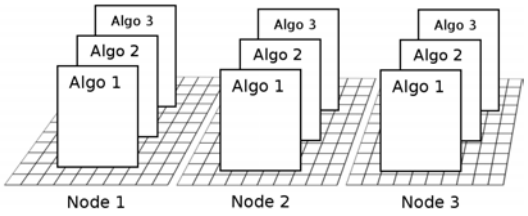


**Figure 3.3.** Visitor Algorithms

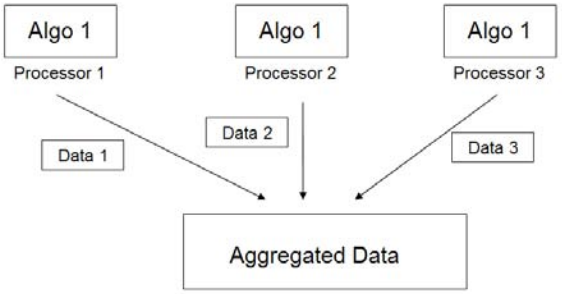


**Figure 3.4.** Particle Algorithm Output

virtual machine. Each of these modules needed to interact with the population. Rather than moving the population data to each of the modules, we allowed the modules to move their code to the population data. This is similar to how distributed systems interact with a common database through sql or stored procedure calls. One can duplicate the more traditional agent update behavior in the PPDM through the use of a simple particle algorithm that calls an update method on all or a selected number of the agents at scheduled intervals. The agents themselves would contain all of the logic needed to interact with neighboring data and agents and perform all necessary updates and movement. Another alternative would be to have the various modules inject differing behavior strategy objects into the agents upon initialization which will eventually be called during updates. Dynamic languages like Ruby would provide an ideal environment for this technique.

We are also looking into incorporating patch-based event queues into the PPDM to allow for algorithms or agents themselves to make use of the discrete event simulation programming model[16]. The queues would allow the PPDM to dynamically adapt its updates to the resolution needed by the particular model being studied and could result in great compute time savings. However, since the PPDM operates in a distributed fashion such an addition would require some additions for synchronization.

### 3.0.2 Communication implementation

The PPDM uses the MPI model for parallel data communication. Communication occurs between copies of particle algorithms running on differing patches, when agents move between cells owned

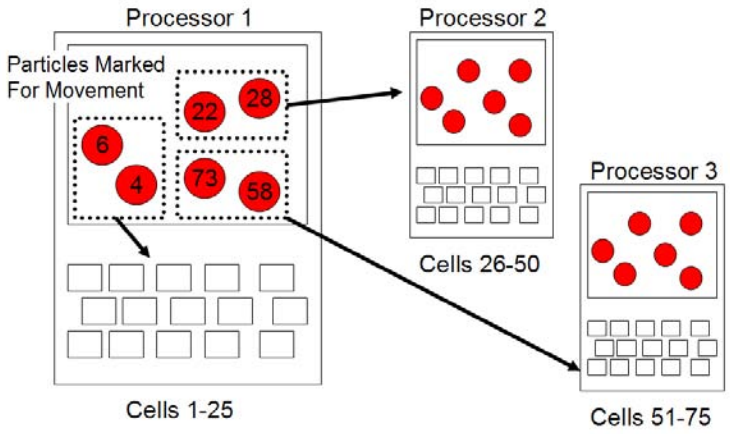


**Figure 3.5.** Particle Movement

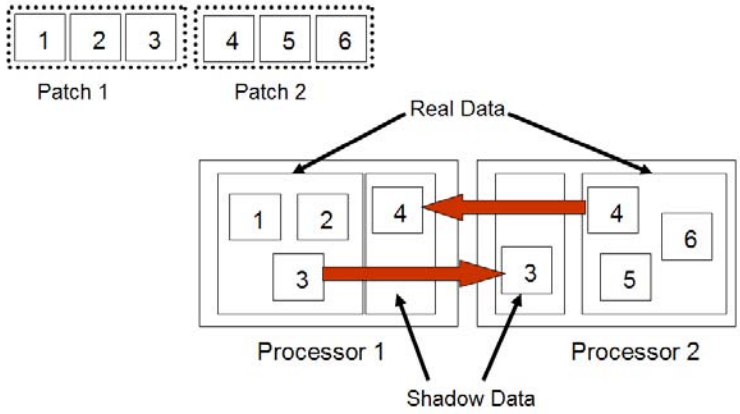


**Figure 3.6.** Shadow Data Update

by different processors, and to update shadow data located on neighboring processors.

When necessary, the particle algorithms communicate and coordinate with each other across patches using standard parallel processing messaging such as the MPI broadcast and reduction methods. We have provided modelers with simplified interfaces to these functions. We have also implemented a messaging service to allow agents to communicate with other agents that are not located on their patch. The PPDM organizes and sends messages using a regularly occurring batch update process. The messaging service is optional and when in use it requires additional overhead because it must track agent locations to enable routing.

The PPDM handles particle movement during a regularly scheduled batch update where all processors work together to redistribute agents to the correct locations (Figure 3.5). We handle copying of shadow cell data for neighboring cells in a similar manner. This requires the various processors to remain synchronized in time (Figure 3.6).

Modelers can configure the PPDM to make use of mpiJava [2] or MPJ Express [23] for the MPI services. Both of these are object-oriented implementations of the MPI protocol using the Java programming language. They differ in that mpiJava provides a JNI wrapping over a native MPI environment (such as MPICH or LAM), while MPJ Express is a complete implementation of the MPI protocol using only Java. We found MPJ Express to be far more portable and easier to use since it removes the native MPI compilation and integration steps. In Java, serialization of object data to byte data is a major cpu cost. We hope that custom serialization code will reduce some of this cost and are looking into the serialization and communication libraries from the Ibis project[7].

# Chapter 4

# Example Simulations

In this section we demonstrate the use of PPDM in three different examples and conclude with an illustration of its parallel capabilities. The problems have been formulated in an agent-based fashion, but carry a PIC flavor in the sense that agents/particles are collated into "bags"/cells with in-cell (in-bag/near-field) behavior modeled at a far greater fidelity than the far-field (out-of-bag) behavior.

### 4.0.3 Spread of disease in a 2-species population

This example approximates the spread of a disease in two separate species, humans and (non-human) primates, with primates acting as a reservoir for the pathogen. The pathogen, modeled loosely on Ebola, is communicable within each of the species and is also capable of primate-to-human jumps. This example models both humans and primates as individual agents. A number of primates and humans are collocated into settlements where humans may come in contact with primates and contract the disease. Both humans and primates may move between cells, though primates' movements are significantly more circumscribed than humans.

#### 4.0.3.1 Disease model

In this work we use a compartmental disease propagation model. Both the humans and the primates go through five compartments/states of health - Susceptible(S), Exposed (E), Mildly Symptomatic (MS), Severely Symptomatic (SS), and Recovered/Dead (R/D). The residence time in each of these states of health is a random variable, which follows a distribution (usually log-normal). The process of infection of humans is governed by their contact with other humans, cadavers and primates, i.e. it depends on their closeness of contact and the probability of meeting an infected individual, primate or cadaver. Transmission within the primate population is entirely by contact. We will assume that both the Mildly Symptomatic and Severely Symptomatic stages are contagious, for both humans and primates.

#### 4.0.3.2 Movement model

We assume that there exist $M$ human settlements, each containing a number of primates. Interactions between human and primate occur between those present in the same settlement. The human and primate populations in each settlement are drawn from the log-normal distribution $\mathscr{L}(N_{median}, S)$, where $N_{median}$ is the median and $S$ the standard deviation. In each settlement we use a median of 1000 and standard deviation of 500 for the human population and a median of 50 and standard deviation of 12.5 for the monkey population.

For each settlement we define a mobility factor $Mh$ and $Mm$ for the human and monkey populations. This mobility factor determines the percentage of each population that will relocate on a given movement step. The indexing of settlements reflects geographical proximity. Primates travel to other human settlements primarily because of proximity. Once the connection routes are determined, a settlement's mobile population is distributed across its connecting settlements using the above probabilities as a weighting factor.

The model can be configured with two modes of population movement. In the first, agents are assigned a fixed day-night routine. In the second mode, agents move randomly several times a day to connected settlements using the above probabilities and weights, and at night they return to their home settlement. Mildly symptomatic individuals move, but severely symptomatic ones do not.

#### 4.0.3.3 Disease Model Implementation and Scaling

The disease model initializes the PPDM with a 2-d lattice geometry. The model represents each settlement by a cell in the lattice where a cell is given an x and y location to determine its distance from other cells. We initialize two separate particle fields for the human and monkey populations. We submit four configurable particle algorithms to the PPDM to evolve the disease agents. These algorithms create the initial population distributions, initialize the agent movement patterns, perform agent movement at a set rate, and evolve the disease state of the agents. The algorithms are modular and can be replaced with different implementations of varying complexity to study how changes to the population or disease dynamics effect the outcome of the epidemic. For this model, two types of inter processor communication take place. During agent movement some agents will need to relocate to neighboring processors, and after each disease update the master disease algorithm coordinates with its neighbor algorithms to output the aggregated disease statistics for that time.

We performed experiments testing both the strong and weak scalability of this model on the PPDM. In the strong scaling test, we initialized the population with a fixed 2,048 settlements, each containing an average population size of 1,000 for a total population of 2 million agents. We then varied the number of processors allocated to the PPDM and ran the simulation for a simulated 2 months. Figure 4.1 shows the results of the strong scaling test. The results show near-ideal scaling up to 16 processors, after which the particular problem setup begins to scale poorly. This poor scaling results because we keep the problem size fixed. As we increase the number of processors, each processor has less to do and communication and synchronizations costs take over.

In the weak scaling test, we initialized the population with a fixed 2,048 settlements. We then scaled the population size up linearly with the number of processors allocated to the PPDM so that each processor would maintain a fixed number of agents. We ran the model for a simulated 2 months. The Estimated Single-Node plot line represents the expected time that a single node would take to perform the problem size if it had infinite memory and did not succumb to non-linear effects. Under the weak scaling case where the problem size grows with the number of allocated processors, we achieve fairly good scaling with the 64 node case within twenty-five percent of the ideal result. We see this good scaling because the communication load for each processor is balanced by the increasing computational load. These results show that we can scale up our modeled size by increasing the available number of processors. Figure 4.4 shows a further exploration of the scaling data space. For a given processor allocation we plot the timing response for problems of increasing scale. The problem size represents a linear scaling of the population allocated to each cell while the number of cells remains fixed. Missing data points exist when a problem size is too large for the number of allocated processors.

To further test the weak scaling of the PPDM, the disease model's problem growth strategy was modified. In this case as the number of processors was increased the population per cell was kept constant but the number of total cells allocated was increased linearly. This led to a fixed number of cells and people allocated per processor. The simulation run time was also increased to 3 months to mask some additional initialization costs. Figure 4.3 shows the resultant weak scaling numbers. Figure 4.6 shows additional plots of the cell based scaling of the disease model. Surprisingly, the scaling results are quite a bit worse for this case. Even though the scaling results show that the problem complexity does scale linearly for a given processor allocation, the run times for a given problem size are much greater than ideal between processor allocations. We have yet to determine the cause for this deviation from the previous weak scaling case as the communication loads between processors are similar. Despite the worse results, this level of scaling is still useful because it allows one to run problem sizes much larger than what can be run on a single processor.

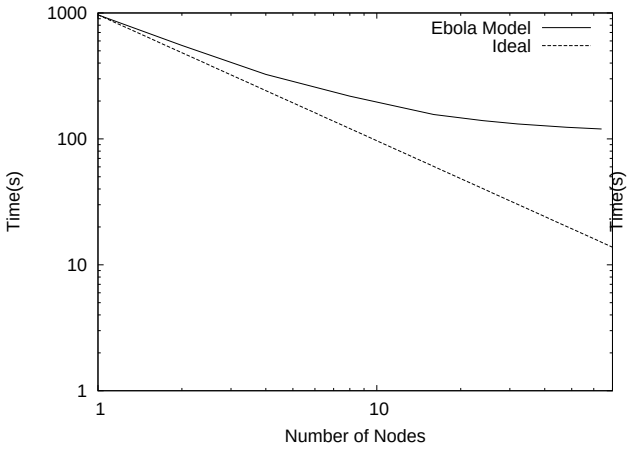


**Figure 4.1.** Disease Strong Scaling

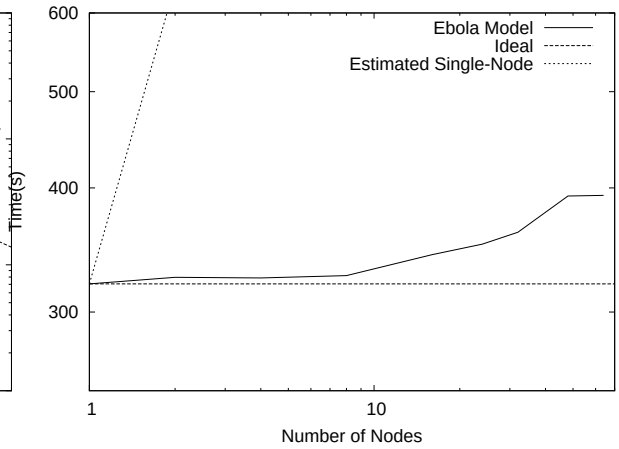


**Figure 4.2.** Disease Weak Scaling (Cell Size)

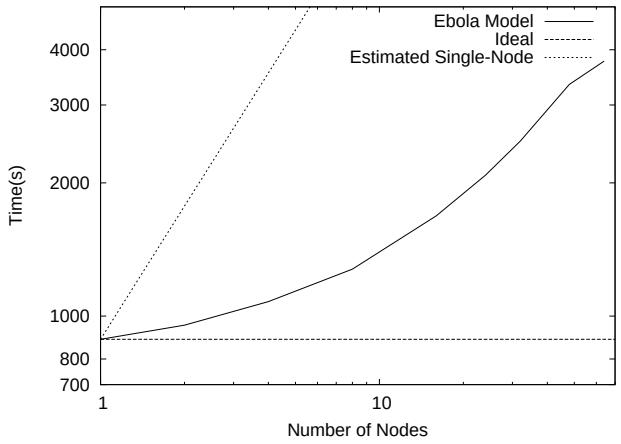


**Figure 4.3.** Disease Weak Scaling (Num Cells)

### 4.0.4 Seldon

Cognitive Seldon is a software toolkit that combines technology and concepts from a variety of different research areas, including psychology, social science, cognitive science, and agent-based modeling and simulation. It has been used to study urban gang recruitment and terrorist network recruitment. This second example demonstrates the utility of Cognitive Seldon and the PPDM in analyzing the effect of media on populations.

Previous uses of Seldon involved rather few ($O(10^3)$) agents, and serial implementations generally sufficed. However, large agent populations were required to study the effect of media, which in turn required parallel computing. Further, the individual agent models were enhanced to include a cognitive model. The cognitive models are essentially semantic graphs of concept activations and edge weights that allow for realistic processing of media information. They lead to an increase in the computational intensity of individual agents, further spurring the need for parallelization.

Cognitive Seldon has two types of agents: individuals and abstract. Abstract agents represent social or institutional concepts that can influence an individual (e.g. schools and mosques). Since they contain a set of individual agent members, they can be highly-connected nodes in the overall structure. For this reason, we have distributed the abstract agents across the processors to avoid bottlenecks. They interact with the local individuals and then communicate across processors to synchronize their state.

An interaction between two agents involves significant processing. Sets of attributes are exchanged and modified according to linear attraction and reinforcement rules. Concept activation vectors are also exchanged, causing nodes to fire in the semantic graphs, and thereby changing their cognitive states.

The simulation commences with an unconnected set of agents. Homophily is used as the basis

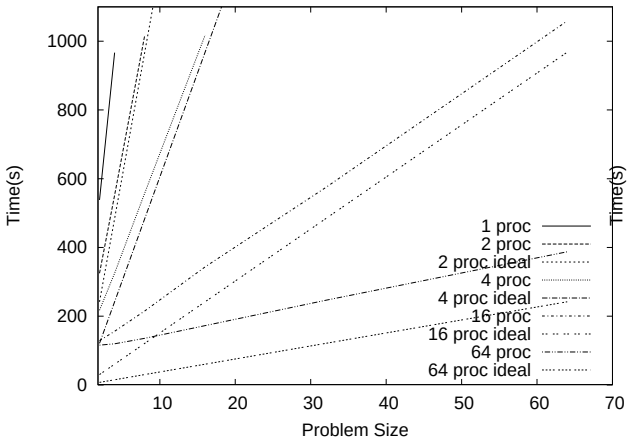


**Figure 4.4.** Disease Performance (Cell Size)

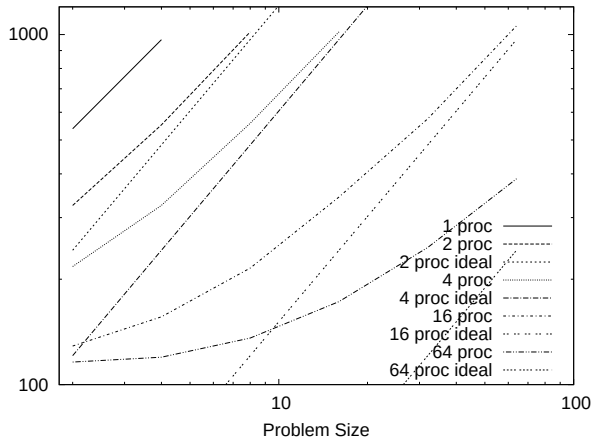


**Figure 4.5.** Disease Performance (Cell Size log/log)

of attraction, and relationships form as interactions proceed. Each agent has a maximum amount of relationship energy, thus providing a flexible cap for either a large number of weak relationships or a small number of strong relationships. Agents also have personality factors, which affect their interactions. For instance, an extroverted agent might interact more than an introverted agent. As relationships evolve, social networks form, ranging from acquaintances to cliques. These, in turn, drive subsequent interactions, so that agents are more likely to interact with close friends in cliques than acquaintances.

Each timestep in the algorithm consists of a couple of steps. First, the individual agents determine their membership with the abstract agents, and then the interactions occur between them. Next, the individual agents identify other individual agents with which to interact. The interaction procedure consists of three parts: *send*, *receive*, and *respond*. Agent A *send*s a subset of its information to Agent B, who then *receive*s the information, compares it to its own, and *respond*s to Agent A. The procedure is transactional, so both agents change their emotional state, or both stay the same. Interactions occur between agents through the creation of message objects, and large numbers of messages are routed and delivered concurrently. Since *receive* messages can create *response* messages, processing continues until there are no more messages. This barrier synchronization ensures that each step finishes completely before the next step begins.

We parallelized Seldon by decomposing the problem across processors in a load-balanced manner. We also maximized the likelihood of intraprocessor communication by using Zoltan [11], a load-balancing library, to invoke graph-partitioning algorithms in ParMETIS [21]. Zoltan uses the social network structure (with relationship strengths as edge weights) to calculate the optimal agent-to-processor mapping. It also provides a distributed directory capability to track these mappings for routing. The data migration is performed separately and involves packing and unpacking agents at the source and target processors, similar to the process of message delivery.

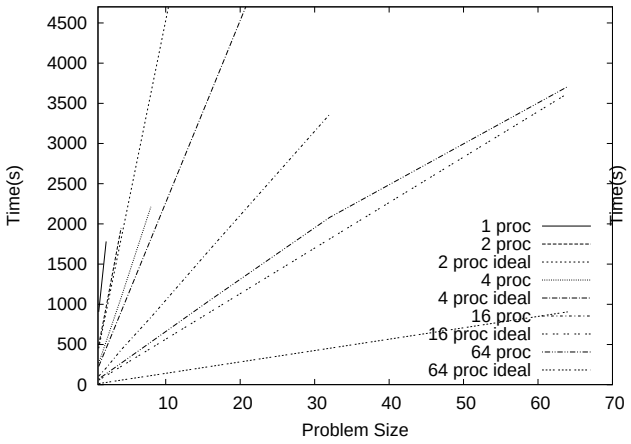


**Figure 4.6.** Disease Performance (Num Cells)

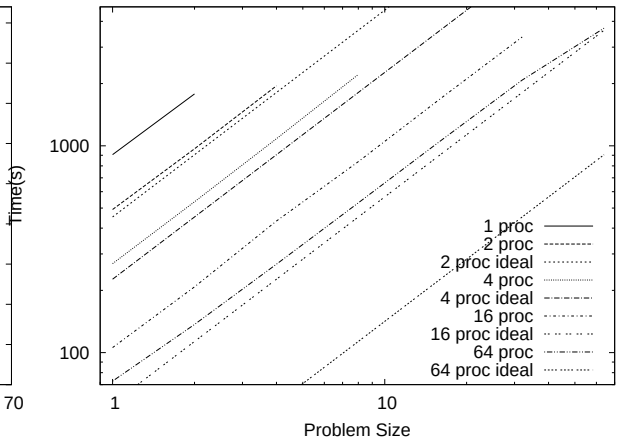


**Figure 4.7.** Disease Performance (Num Cells log/log)

When repartitioning, Zoltan exploits the current decomposition to reduce data migration. An additional difficulty involved multi-language integration, since Zoltan is implemented in C/C++ and Seldon is implemented in Java. We created a JNI wrapper around Zoltan using Swig [5], which then provided access from Seldon.

To study scaling characteristics, we used a cognitive model with agents for individuals and media outlets to simulate the shift in public opinion in Amman, Jordan, after the November, 2005 bombings. The individual agents were provided with cognitive information automatically generated from Jordanian newspapers published before and around the time of the bombings. The weak scaling runs held the number of individuals constant at 1000; the strong scaling runs used $1000P$ individuals, where $P$ is the number of processors. There were 40 media agents in all runs.

In Figures 4.8 and 4.9 we plot results from strong and weak scalability studies. It is clear from the weak scalability analysis (Figure 4.9) that there is little locality in the agent-interaction pattern - cross-processor communication costs increase as the processors (and the total problem size) are increased. The strong scalability run is somewhat more promising, following a quasi-ideal convergence till around 10 nodes; thereafter divergence from ideal is abrupt, as the high communication costs (observed in the weak scalability analysis) overwhelm the steadily decreasing computational costs. At a point (around 30 nodes), the communication costs dominate and a follow the trend observed in the weak scalability study (Figure 4.9) where communication costs are roughly proportional to the number of processors.

### 4.0.5 Parallel Social Forces

A parallel implementation of the social forces algorithm for pedestriam simulation [17] was built using the PPDM. The social forces algorithm is used for simulated the movement behavior of crowds. Each person's movement is directed by the summation of all social forces acting upon them. Repulsive forces come from nearby people, obstructions such as walls, and dangers such as fire. Attractive forces come from a person's desired exit or someone that person wishes to follow. A person tries to reach their goal while keeping a 'comfortable' distance from near by people or obstacles. When properly calibrated using these forces can realistic replicate crowd behavior and produces emergent behavior observed in real life such as lane formation and exit bottlenecks. This application space provided an ideal situation for the PPDM since the problem fits nicely with the particle-in-cell processing paradigm. In fact, parts of the PPDM's design were inspired by another parallel implementation of the social forces algorithm which used C and MPI [30].

The test social forces application uses a 2D domain simulating a very large room. This room is divided into a grid of cell locations. This grid is further subdivided horizontally into a series of patches designating one patch per processor. For our test application we populate the room with a configured number of people put in random locations. Each person is also randomly assigned one of four goal locations. When a person reaches a their goal they are reassigned a new random goal location. A person is color coded based on their current goal location.

At each time step the location of each person in the simulation needs to be calculated. All of the processors first receive the updated location of people in their shadow cells from neighboring processors. This is necessary because people in a cell are affected by the other people in their cell as well as people in neighboring cells, which may be managed by external processors. Next the processor must now update the people in the cells that it manages. For a given cell each person is updated by calculating all of the forces acting on that person. First the social forces of nearby people are applied. This is done by iterating through the people in the cell and neighboring cells. Next the forces of obstructing objects is applied. Finally the goal seeking force is applied. Once all of the forces have been added the person's next location may be set. Finally all people must be relocated to their new cell location. This may require moving the person to a new processor. After all updates on all processors have been performed a collective location update is performed by all processors which results in the master processor holding a list of people's location. This location list is used to update the display.

Such code could be used as the base of a crowd evacuation simulation tool. By using the PPDM, such an application could be distributed across a large number of processors and simulate very large crowds, with complicated behavior, in faster than real-time. This simulation could be used to launch a batch of jobs to test different evacuation strategies in the face of an impending emergency and could be used to redirect crowds as emergency conditions changed. However, we have only a toy implementation of the social forces algorithm and such a crowd evacuation tool would need significant additions over the current toy implementation. These additions include the ability to build the simulated crowd environment including building floor plans and obstructions as well as the tools necessary to properly load balance the population and geometry across the allocated processors. It would also need enhanced crowd behaviors such as panic and leadership,

and a path search component.

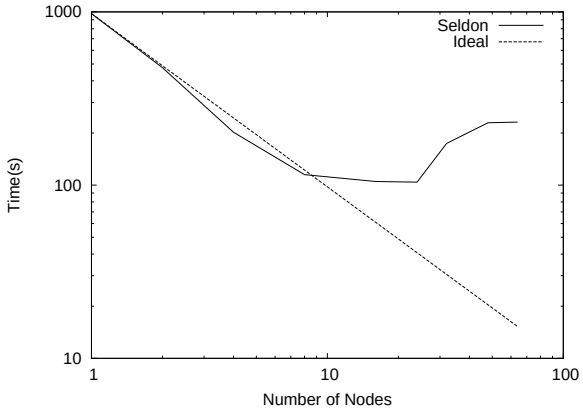


**Figure 4.8.** Seldon Strong Scaling

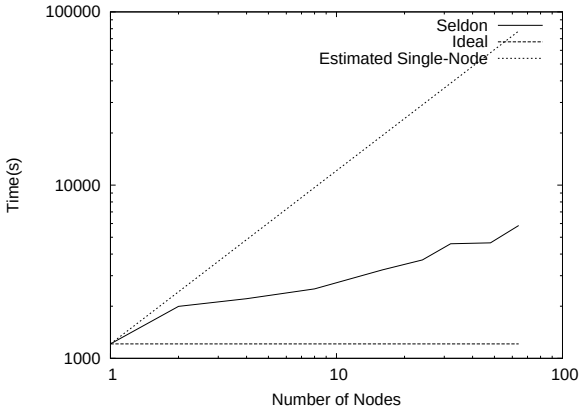


**Figure 4.9.** Seldon Weak Scaling

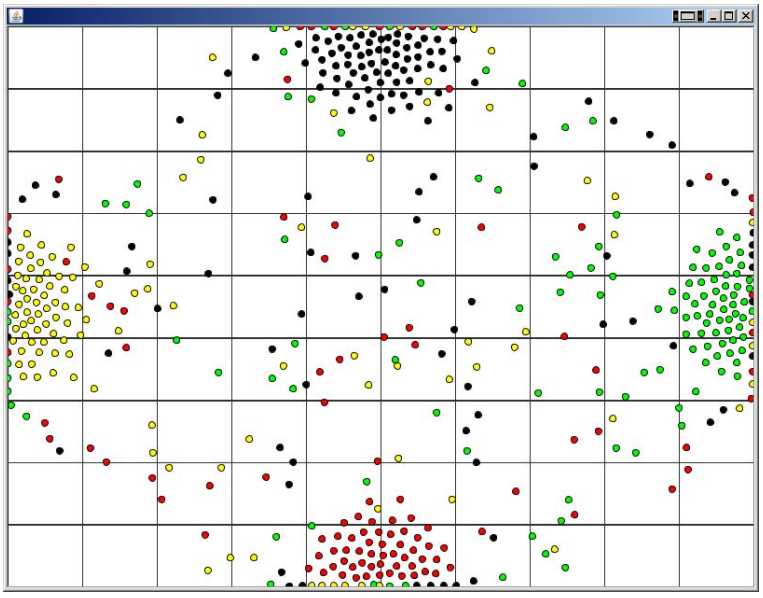

**Figure 4.10.** Social Forces Example

# Chapter 5

# Codebase

The ideas behind, and some of the code for the PPDM came from Sandia's BioDAC project. BioDAC simulated a biological attack on an urban environment. We wished to refactor BioDAC's population simulation code to create a reusable toolkit for large agent calculations that could be used in other projects. We also wanted to extend this code to allow it to run in parallel across a number of machines. This is the primary reason behind the choice of Java as the implementation language. Java was also ideal in this case because of the large amount of Java developing experience found in the teams that would be creating and using this tool. Java is also widespread outside of this community and many code libraries are available for use in creating and extending the PPDM. The Java community is also rich with many development tools for building, testing, debugging, and profiling Java code.

### 5.0.6 Build Tool

At the beginning of the project we experimented with a new Java project management and build tool named Maven [37]. The key feature of Maven is build automation through convention. It boasts advantages over the standard Java build tool Ant [13] becuase it provides a standardized project layout and automates the most common build processes such as compiling, testing, and deploying. This standardization could save developers a lot of time when switching between projects that all use Maven. Maven also has the advantage of automatically downloading dependency library jar files from local or remote repositories. This allows for easy library upgrading and allows different projects to share library files.

However, in practice Maven ended up costing our team more time than it saved and caused many head-aches when compared to Ant. To be fair, the developers of the project had much experience with Ant and no experience with Maven. Maven has also made improvements over the years that may have solved our original problems. The automated build processes were nice but we found adding additional build tasks to be much more difficult than in Ant. We also did not reap the benefits of the standardized layout since no other projects were using Maven. The automated downloading of jar files also caused problems. This required setting up a local Maven repository for us to host libraries not found on remote repositories. We were plagued with proxy problems and the process for uploading and downloading these libraries proved much more complicated than the Ant standard of simply keeping all jar files in a lib directory. We eventually switched back to using Ant for our build process. The Maven project seems to be an excellent idea but in practice it

would need to be adopted across an organization and include team training to see its benefits.

### 5.0.7 Development Environment

The PPDM projects used Subversion [20] for its revision control software and had no troubles. Developers were free to use any development tools they chose but most used the Eclipse IDE [14] for its rich Java development feature set. A powerful component of Eclipse is the many available plug-ins such as Subclipse [35] which made the functionality of Subversion available from Eclipse. A DokuWiki [32] based Wiki was set up as a collaboration tool for the development team to document the build process, project progress, and any common problems. Unfortunately the Wiki quickly fell out of use. Wiki's can be very useful to a project and aid in the documentation and sharing process. Unfortunately they also require discipline on the development teams part and can suffer from a drop in momentum. Another important component of the development environment was the use of JUnit [6] for unit testing. The JUnit framework makes creating and launching standardized test cases for Java code very easy. Developers are highly encouraged to write and run test cases for all functionality. The testing process helps speed up the process of creating correct code by being able to quickly determine when and where there are errors. It also leads to the creation of code that is easier to test. Another benefit of a large suite of tests is that one can refactor the code base and feel more confident that they did not create any new bugs if all of the tests pass.

### 5.0.8 Java and Parallel Computing

On top of working with existing and future Java based Sandia simulation programs, a second requirement of the PPDM was to perform large, complex simulations across a cluster of computers. Unfortunately, Java has not been the language of choice in the high performance computing community. This community typically prefers C, C++, and Fortran which offer faster and more direct access to memory and io devices and consequently has a large body of tools to support parallel and distributed computation. Fortunately, the stigma of Java being a low performance language has abated with improvements in its Just In Time compilers and improved garbage collector implementations, as well as new advanced IO and communication libraries. Java libraries such as Java Fast Sockets (JFS) [34] also exist to allow users to take advantage of advanced high performance communication interconnects that were previously unavailable to Java developers. This has lead to the creation of new parallel communication tools for Java. Some studies are finding that such improvements and libraries are making distributed Java based application's performance comparable to and in some cases even better than those programmed in C and Fortran [1].

We selected the Message Passing Interface (MPI) for Java for the PPDM's parallel communication. MPI is a well established protocol for parallel computation with a long history. We felt that the maturity of this protocol and the previous experience held by some of the developers made it the best choice. In the process of building and testing the PPDM two different Java based MPI libraries were used.

mpiJava provides an object-oriented Java interface to the MPI library. It uses the Java Native Interface to wrap a natively compiled MPI library. In effect all Java calls are forwarded onto the native MPI library (usually written in C) that was compiled for that machine. With mpiJava, the user still needs to be able to compile a native version of MPI that will work with their machine. Their version of Java and mpiJava also need to work correctly with the native library. Much time was spent getting such a setup working and difficulties still occur when moving the PPDM to new machines. However, mpiJava's use of a native MPI library should help boost performance over alternatives since the native library should be optimized for the machine and have direct access to the most efficient communication devices. We had the most success in using MPICH2 as the native MPI library. We were unsuccessful in getting mpiJava on our machines to work with LAM/MPI and Open MPI. But even with mpich2 we still experienced program freezes and crashes. Most of these would occur in the native code making debugging incredibly difficult.

Seldon is an agent-based social simulation that uses the PPDM for message passing and agent migration. Seldon also uses Zoltan, a load-balancing library that is based on MPI as well. Unfortunately, MPI is not set up to run multiple instances at the same time. If more than one library is statically linked against MPI, the instances wait for each other and the program hangs. To avoid this, we needed to build MPI with shared libraries so that both Zoltan and the PPDM could dynamically link to them. The following command was used to build MPI (specifically, mpich2 v1.0.5p4) with shared libraries:

$ ./configure –prefix=⟨MPI install path⟩ –enable-sharedlibs=gcc –disable-cxx –with-rsh=ssh

We then added MPI's shared library directory to LD_LIBRARY_PATH, and set the MPICH_USE_SHLIB variable to "yes":

$ export LD_LIBRARY_PATH=⟨MPI install path⟩/lib
$ export MPICH_USE_SHLIB=yes

Once you set MPICH_USE_SHLIB, you can configure mpiJava in the standard way ($ ./configure –with-MPI=mpich) and it will use the MPI shared libraries. For Zoltan, we first compiled it with "-fPIC", and then we pulled it and the MPI shared libraries into the Seldon shared library with "gcc -shared".

MPJ Express is the second library we tested. This is a pure Java implementation of the MPI protocol. Its interfaces also match those of mpiJava allowing us to easily swap between the two libraries. The advantage of MPJ Express is that no native MPI implementation needs to be compiled for the machine. This allowed us to easily run the PPDM on many different architectures. Errors were also much easier to track down using Java's exception handling capabilities. This proved to be a much more stable platform for running the PPDM. Unfortunately because it is a pure Java implementation it can not make use of the cluster's high speed interconnects. This provides a tradeoff of stability and speed. MPJ Express is definitely the choice when prototyping new parallel applications.

A third Java implementation of MPI that exists is contained within the Ibis project [7]. Unfortunately even through the classes, functions, and overall functionality are the same as the previous

two libraries, the Ibis project chose to use a different package structure. Code changes to the PPDM would be needed in order to utilize Ibis. ProActive [8] is an additional pure Java based distributed programming environment that is gaining popularity and has advanced features that could be used by the PPDM. This points out that the PPDM could benefit by abstracting its use of the underlying communication library. Such an abstraction could allow users to try out different communication libraries with the PPDM. A user could use the framework the best matches their application's convenience and performance needs.

### 5.0.9 Launching the PPDM

Much effort was needed in debugging the launch process of the PPDM. This is simple in serial mode where PPDM can be treated as a normal Java component and adds no requirements to the program launch process. Difficulty arose when using the PPDM in parallel mode.

1. MPI Libraries require special launch configurations

   The two MPI Java implementations provide similar yet different ways of launching programs that use their library. They are similar in that both provide a series of batch scripts and supporting programs. Both require that daemon applications be running on all compute nodes. A starter program is launched from the master node. This takes in a series of arguments needed for the MPI configuration as well as the actual program. Once the starter program is launched it connects to the daemon programs running on the supporting compute nodes. The start program sends the needed launch arguments and the daemon programs launch the the actual Java application. Proper care must be taken to insure the program codebase is available to the compute nodes. A shared file system eases this requirement but results in slower speeds. Both MPI libraries provide scripts for their starter and daemon applications, however their arguments differ and in some cases need to be changed for a given deployment. This necessitated the creation of separate PPDM launching scripts for the different library types. This script put together the proper Java class path for the application and constructed the arguments and calls for the library used and helped hide many of the problems previously faced by the user.

2. Batch Job Scheduling

   Batch scheduling of jobs is an important feature of the PPDM. This allows one to submit many jobs to a cluster to be run in parallel. We used the qsub scheduling tool found on Sandia's compute clusters for this purpose. Unfortunately again, different cluster environments require different qsub arguments. We again created scripts for the different environments to hide these complications from the user. Special care is needed to be sure the daemon and starter programs get launched correctly. We made use of the interactive option (-I) for much of our debugging when running parallel jobs. Care and some debugging is still necessary when moving the PPDM to new parallel environments.

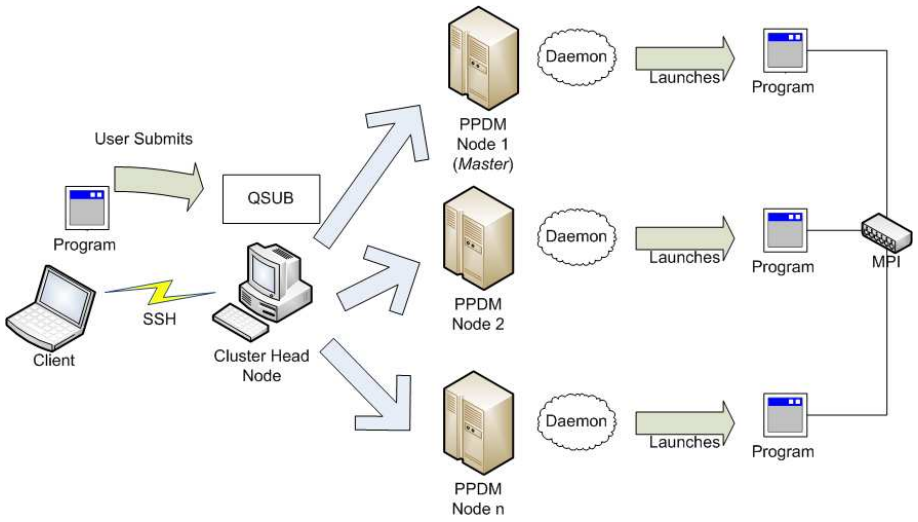


**Figure 5.1.** Launch Diagram

# Chapter 6

# Integration

We integrated the PPDM with several existing simulation frameworks on an experimental basis. This was used to see how well the PPDM could work with Sandia's existing simulation codebase as well as how it might work with future simulation programs.

### 6.0.10 HLA

We first used the PPDM in conjunction with an HLA federation [16] to simulate a moving population in a large metropolitan area. This was an important test because the primary purpose of the PPDM was to add parallel support to the Weapons of Mass Destruction Decision Analysis Center (WMD-DAC) suite of simulation tools. These tools consist of a collection of interacting submodels that simulate different parts of the overall system under study. Each submodel consists of a HLA Federate. By incorporating the PPDM into one of these Federates, we allow that particular submodel to scale in size and complexity beyond what could be processed on a single computer. In this integration we tested the PPDM in three modes of operation.

1. Single Machine Figure 6.1

   In the first mode of operation all of the HLA federation was run on a single Windows machine. Part of this federation was the Population Federate which contained an instance of the PPDM running in serial mode. The PPDM was used to maintain and update the people agents in the simulation. The other federates could submit particle algorithms as well as other supporting data objects through HLA to the Population Federate. The algorithms were then run by the PPDM to update the state of the agents and generate output statistics. Output data was passed from the PPDM to the Population Federate and then out to all subscribing Federates via HLA. This integration was the least complicated due to the fact that a good portion of the source of the PPDM came from a previous version of the Population Federate used in the BioDAC program and there were no parallel processing or distributed computing issues. This integration was successful and several current Sandia applications are using the PPDM in this manner.

2. External Machine/Cluster Connection Figure 6.2

   In the second mode of operation the HLA federation was again run on a single Windows machine to be further referred to as the client. As opposed to above, the PPDM was started

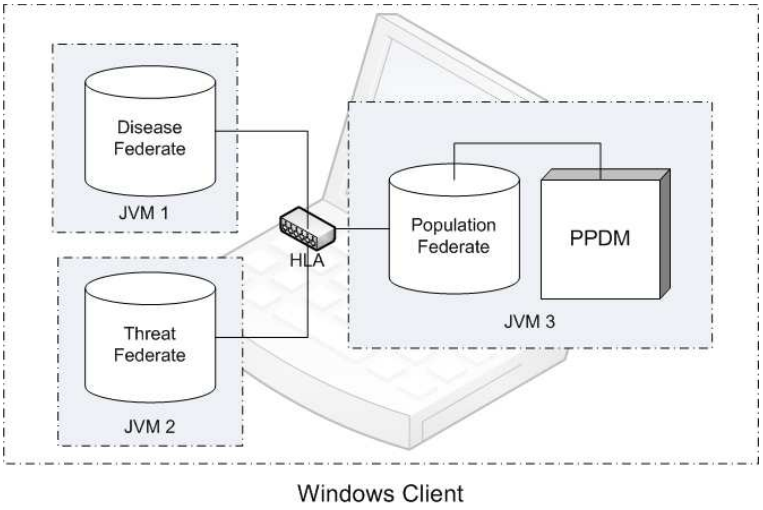


**Figure 6.1.** Single Machine

separately on a Linux compute cluster in parallel mode utilizing a configured number of compute nodes. The launching of the PPDM can be done interactively or through a job submission client. Once both the Population Federate on the client and the PPDM on the cluster are started a connection is made between them using a tunnel built using a custom implementation of Java's Remote Method Invocation (RMI) functionality. The HLA federation on the client runs as above but makes calls to a RMI proxy to the PPDM. These calls are forwarded to the PPDM on the compute cluster using the RMI tunnel. Setting up the RMI tunnel between the client machine and cluster proved to be complicated. Cluster nodes are not addressable from outside of the cluster and the client machine is not addressable from inside the cluster. A daemon application was setup on the cluster's head node to solve this problem. The head node of a cluster is generally used for launching jobs as well as accessing resources on the compute nodes. When launching the parallel job on the cluster the daemon application must also be launched on the head node. When the parallel job starts up one of the processors is set to be the master node and provides coordination between all of the worker nodes. This master node connects to the daemon application on the head node via RMI. When the Population Federate starts up on the client machine it also connects to the daemon application on the head node. RMI traffic can then pass from the client application to the head node, then from the head node to the master PPDM node. If necessary the master PPDM node then distributes the data to the PPDM compute nodes. Overall, this mode of operation is quite complicated and the tunnel between the client node and the cluster nodes adds a lot of communication overhead. However such a setup can pay off if the size and workload of the population processing is high. This setup was also necessary in the case where the application could only run on a Windows machine. This setup allows the heavy processing and size of the population code to be distributed across the cluster allowing the Windows client to process larger and more complicated scenarios.

3. Cluster Figure 6.3

   In the third mode of operation both the PPDM and the HLA Federation ran on the cluster.

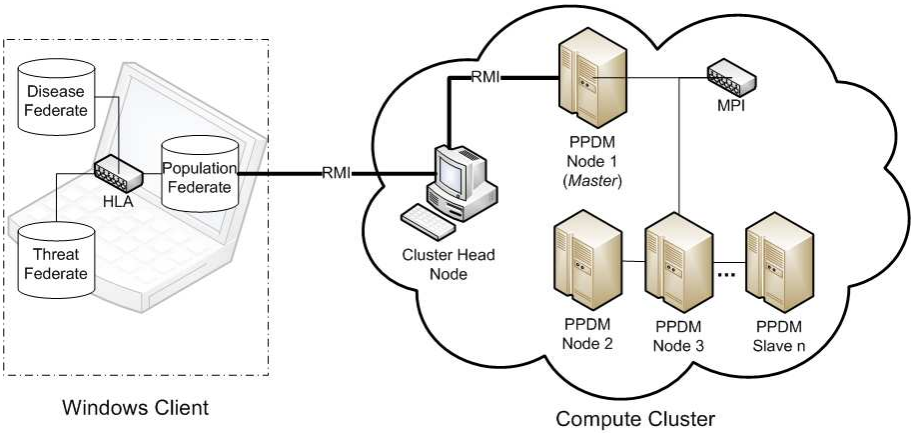


**Figure 6.2.** External Machine/Cluster Connection

However, the test application in question was only a simple toy program since the larger HLA federation program only worked on Windows. The cluster's job submission tool was used and upon launching each test federate was started on its own compute node. An instance of the PPDM was also launched in parallel across a number of compute nodes. The master node of the PPDM was located on the same node as the Population Test Federate. Communication between the PPDM and the test federate occurred again through RMI even though both were on the same machine. This use of RMI and associated overhead could be removed if both the Population Test Federate and the master node of the PPDM were started in the same Java Virtual Machine. This is just a mater of solving some complications in the launch code for Federates and the PPDM. Running everything inside of the cluster represents a more sane and less cumbersome mode of operation. Federates can be distributed over a number of processors and the Federate to PPDM connection is much more direct. This mode represents the best case for non interactive, batch data analysis.

### 6.0.11 RePast

We also used the PPDM as the underlying data structure for one of the demo application models of the RePast Agent-Based Modeling toolkit [9]. RePast is a free and open source agent based modeling toolkit and is very popular in the social simulation community. The tool was originally developed at the University of Chicago but has subsequently become an open source project receiving much development support and direction from Argonne National Laboratory. This toolkit provides many utilities for agent creation, visualization, geometry, scheduling, and data analysis. It incorporates the use of dynamic scripting languages such as Groovy and also provides tools for some concurrent event processing, allowing the utilization of multi-processor/multi-core systems as well as clustered batch processing mechanisms. However, it is limited in that that it currently does not support distributed, fine grained parallelism of a model. To clarify, a single model can not be run distributed across multiple computers. This limits the size and complexity of a model to what can be stored and processed on a single machine. This is the very task that the PPDM was

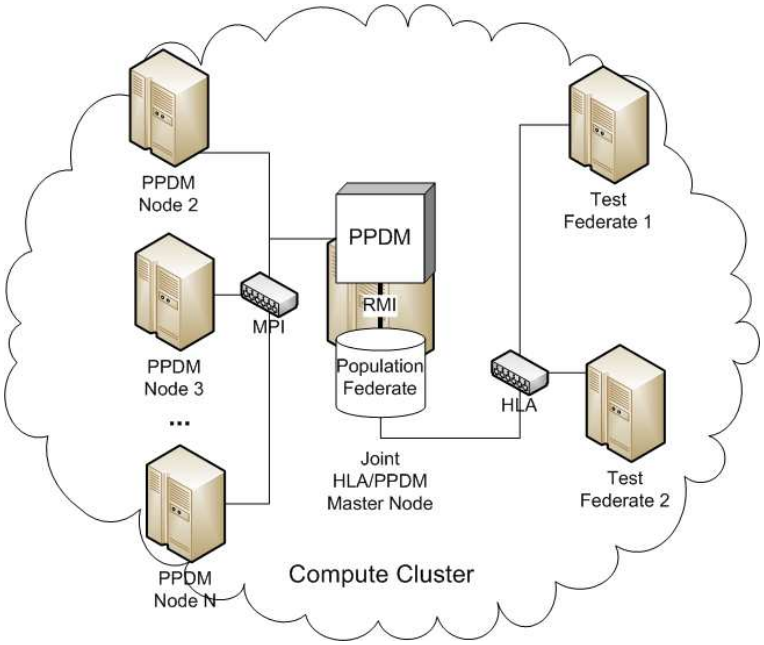


**Figure 6.3.** Cluster

designed to handle.

An ad-hoc integration between the two tools was undertaken to prove that a RePast model could be run distributed using the PPDM. A similar approach to adding parallelism to RePast can be found in [27]. To provide the integration, the 2d landscape of the model was replaced by a PPDM Domain with one Particle Field. The PPDM was run in Single Program, Multiple Data (SPMD) mode and each processor was in charge of creating a segment of the agent population based on its id. No changes were made to the RePast model's agents or agent update code. We did have to make some changes to the RePast `Schedule` class to maintain synchronization between the PPDM worker nodes. At each time step synchronization took place to allow for agent movement and shadow agent updates between processors. We scheduled particle movement and update at fixed intervals on all nodes.

This was a proof-of-concept integration and a more detailed analysis and code changes would need to be undertaken if the PPDM were to be effectively integrated into the RePast codebase. Such an integration could be advantageous to RePast, the social simulation community, as well as Sandia and the PPDM project itself. The social simulation community would benefit by being able to run their existing and new RePast models in a distributed fashion by using the PPDM. Sandia and PPDM user's would benefit by allowing use of RePast's library of agent based modeling utilities. There would also be the benefit of a larger user base for finding and fixing bugs as well as a pool of people willing to make continued improvements to the framework. The 2nd World Congress on Social Simulation afforded us contact with one of the core developers and project life planners of RePast. The project is very welcoming of outside contributions but would require the PPDM to be open-sourced and be made available with one of their licenses. The RePast project itself

also has several side projects, not available at the time of this paper, working to make distributing computing available to RePast modelers.

# Chapter 7

# Improvements/Limitations

The following is a list of limitations of the PPDM as well as suggestions for further improvement.

1. Load Balancing

   The PPDM currently comes with no load balancing tools. Patches are evenly distributed across the available processors. This may not be ideal as patch loads may very greatly. Load balancing was not in the scope of this LDRD but, to be effective, the PPDM should be able to make use of load balancing tools or at least provide the user with ways of choosing their own load balancing scheme. Seldon, which uses the PPDM, uses the Zoltan library for load balancing and provides an example of how other applications may do the same.

2. Parallel Discrete Event Processing Support (PDES)

   All PPDM applications are currently time stepped. A given time is reached, all of the patches are processed, the patch processors synchronize, time is advanced and the process repeats. This time stepping behavior is wasteful in cases where the work load varies in time and space across the simulation domain. The PPDM could be made compatible with the PDES paradigm so better support such simulations. In this paradigm each patch would be treated as its own processing unit (logical process) and would only need to synchronize with neighboring patches if there were a time or data dependence between them. This allows time to advance at rates that match the actual model needs. Such an integration would take some work but could greatly expand the powers of the PPDM.

3. Simulation Support Tools

   Simulation frameworks such as RePast provide for the integration of a lot of different simulation support tools for visualization and statistics. Such a capability should also be available to users of the PPDM. The actual implementation of these tools would be out of the scope of the project. However, it would be very beneficial to search out existing libraries that would be compatible with the PPDM and provide examples of how to use them in conjunction with the PPDM. This could greatly increase the functionality and popularity of the PPDM.

4. Dynamic Language Support

   Dynamic languages such as Python [36], Groovy [33], and Ruby [24] have been growing quickly in popularity due to the increase in programmer productivity that they can achieve. Sophisticated support for these languages using Java's JVM (Jython [18], JRuby [25]) also

continues to make progress. Support for these language inside of the PPDM may increase the PPDM's ease of use, user base, and even functionality.

5. More data access patterns

   Currently the PPDM provides only a few particle algorithm types for iterating through the simulated population of agents. The algorithms mimic the observer design pattern and give the user access to the population one agent, cell, or patch at a time. After doing all of their processing they can submit a collective operation that allows them to synchronize their output data. Additional accessor algorithm types could be created to provide new and useful access patterns. One such useful access pattern would use the MapReduce [10] algorithm. This algorithm would provide access to the agent population one agent, cell, or patch at a time. But instead of a final collective output the user would emit intermediate outputs as they process the population. When defining the algorithm the user also submits a corresponding gather algorithm that will be used to process all of the outputs that they emit across all processors. We would hope that as the PPDM gets further used additional useful access patterns would emerge that would eventually be folded back into the PPDM code base.

6. Better support for multi core/processor machines

   Being a Java multithreaded application, the PPDM is already natively able to take some advantage of multi core/processor machines. This is particularly applicable in garbage collecting operations that can run in the background. However, computer architectures are increasingly being pushed to multi-core systems. For the current implementation of the PPDM to fully take advantage of the power of a core a separate JVM instance must be launch for each core. This creates a separate memory space, additional overhead, and the added complication of parallel launch scripts. This overhead could be eliminated if the PPDM were refactored to run in a multithreaded manner for each patch. This would require duplicating all processing requests across the patches and would need additions to properly handle the collective operations between patches. The multiple cores could also be put to use in the synchronization states of shadow data updates and agent movements. Such implementations are definitely warranted and would greatly improve performance if the trend toward multi-core architectures continues.

# Chapter 8

# Conclusion

Agent-based modeling and discrete event simulation are arguably the standard tools for understanding and making predictions about complex systems. To some degree both ABM and DES make the argument that, unlike statistical methods, simulations must be performed at scale. For such social systems, there is no *a priori* idea of a scaling law or average that would predict their emergent behavior. Because the phenomena simulated are usually large and complex, and because they must be computed at scale, parallel high performance computing is required to enable successful simulations of social systems.

We hope that our work in developing the Parallel Particle Data Model (PPDM) instigates the development of a library of general purpose components for large-scale entity modeling. From a software engineering point of view, separable components for ABM/DES will aid repeatability and methodical experimentation. Beyond reusable software to speed software development, the ability to change out entity models and even the framework and schemes for updating without changing the parallelization scheme is an important contribution of the PPDM. Often when comparing two different models that purport to achieve the same result, the phenomenological particulars of the solutions differ in so many ways that comparisons are difficult. Separating out the parallel implementation, as we have done in the PPDM, hopefully will make comparison between different models easier in the future.

# References


[1] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, and Guillermo L. Taboada. Current state of java for hpc. http://hal.inria.fr/inria-00312039/en, 2008.

[2] Mark Baker and Bryan Carpenter. mpiJava. http://www.hpjava.org/mpiJava.html.

[3] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, , and M.V. Marathe. Transims: Transportation analysis simulation systems. http://ndssl.vbi.vt.edu/transims.html.

[4] C. Barrett, S. Eubank, V.S. Anil Kumar, and M. Marathe. The epidemiological simulation system (episims). http://ndssl.vbi.vt.edu/episims.html.

[5] David Beazley. Swig homepage. http://www.swig.org.

[6] Kent Beck and Erich Gamma. JUnit - a unit testing framework for the java programming language. http://junit.sourceforge.net/.

[7] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. Ibis: Efficient java-based grid computing. http://projects.gforge.cs.vu.nl/ibis/.

[8] Denis Caromel. Proactive - parallel, distributed, multi-core solutions with java. http://proactive.inria.fr/.

[9] Nick Collier. Repast agent simulation toolkit. http://repast.sourceforge.net/index.html.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, 2004.

[11] Karen Devine. Zoltan homepage. http://www.cs.sandia.gov/Zoltan/.

[12] Joshua M. Epstein. *Generative Social Science*. Princeton University Press, New Jersey, 2007.

[13] Apache Software Foundation. Ant - a software tool for automating software build processes. http://ant.apache.org/.

[14] Eclipse Foundation. Eclipse - an open development platform. http://www.eclipse.org/.

[15] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-scale network simulation: How big? How fast? In M. Calzarossa and E. Gelenbe, editors, *11th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2003)*, volume 2965 of *Lecture Notes in Computer Science*, pages 116–123. IEEE Computer Society, 2006.

[16] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, New York, 2000.

[17] D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000.

[18] Jim Hugunin. Jython - java implementation of the python interpreter. http://www.jython.org.

[19] M. Hybinette, E. Kraemer, X. Yinand G. Matthews, and J. Ahmed. Sassy: A design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In B. Lawson, F. Perrone, J. Liu, and F. Wieland, editors, *Proceedings of the 2006 Winter Simulation Conference*, pages 926–933. IEEE Computer Society, 2006.

[20] CollabNet Inc. Subversion - a version control system. http://subversion.tigris.org/.

[21] G Karypis, K Schloegel, and V Kumar. Parmetis homepage. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[22] Sean Luke, Gabriel Catalin Balan, and Liviu Panait. Mason: Multi-agent simulator of neighborhoods. http://cs.gmu.edu/ eclab/projects/mason/.

[23] Bryan Carpenter Mark Baker and Aamir Shafi. Mpj express project. http://acet.rdg.ac.uk/projects/mpj.

[24] Yukihiro Matsumoto. Ruby programming language. http://www.ruby-lang.org/en/.

[25] Charles Nutter, Thomas Enebo, Ola Bini, , and Nick Sieger. Jruby - java implementation of the ruby interpreter. http://jruby.codehaus.org/.

[26] Jon Parker. A flexible, large-scale, distributed agent based epidemic model. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 1543–1547, Piscataway, NJ, USA, 2007. IEEE Press.

[27] Hazel R. Parry, Andrew J. Evans, and Alison J. Heppenstall. Millions of agents: Parallel simulations with the repast agent-based toolkit. In *International Symposium on Agent Based Modeling and Simulation*, 2006.

[28] K. S. Perumalla. A systems approach to scalable transportation network modeling. In B. Lawson, F. Perrone, J. Liu, and F. Wieland, editors, *Proceedings of the 2006 Winter Simulation Conference*, pages 1500–1507. IEEE Computer Society, 2006.

[29] K. S. Perumalla. Scaling Time Warp-based discrete event execution to $10^4$ processors on a Blue Gene supercomputer. In U. Banerjee, J. Moreira, M. Dubois, and P. Stenström, editors, *Proceedings of the 4th Conference on Computing Frontiers, 2007*, pages 69–76. Association for Computing Machinery, 2007.

[30] M. J. Quinn, R. A. Metoyer, and K. Hunter-Zaworski. Parallel implementation of the social forces model. *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, 2003.

[31] C. Reynolds. Big fast crowds on ps3. In *Sandbox (an ACM Video Games Symposium), Boston, Massachusetts*, 2006.

[32] splitbrain.org. Dokuwiki - standards compliant, simple to use wiki. http://www.dokuwiki.org/dokuwiki.

[33] James Strachan. Groovy programming language. http://groovy.codehaus.org/.

[34] G. L. Taboada, J. Touriño, and R. Doallo. Efficient Java Communication Protocols on High-speed Cluster Interconnects. In *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, pages 264–271, Tampa, FL, 2006.

[35] Tigris. Subclipse - subversion eclipse plug-in. http://subclipse.tigris.org/.

[36] Guido van Rossum. Python programming language. http://www.python.org/.

[37] Jason van Zyl. Maven - software tool for java project management and build automation. http://maven.apache.org/.

# Appendix A

# Publications

The following paper appeared in the proceedings of the 2008 World Congress on Social Simulation at George Mason University in Farfax, VA in July 2008.

# Appendix B

# Parallel Particle Data Model

Zachary J. Heath, Max S. Shneider, Jaideep Ray, Benjamin A. Allan,
Keith B. Vanderveen, Michael E. Goldsby and Robert C. Armstrong
{*zheath, msshnei, jairay, baallan, kbvande, megolds, rob*}*@sandia.gov*
*Sandia National Laboratories*
*7011 East Avenue, MS 9915, Livermore CA, 94550-0969*

**Abstract**

We present the design and performance of a parallel entity simulation framework called the Parallel Particle Data Model (PPDM). Based loosely on a Particle-In-Cell algorithm, the PPDM orchestrates and supports agent-based simulations on a parallel high-performance platform. The PPDM is targeted at social simulation applications and is designed to be portable to a variety of high-performance platforms. In this paper we show that the PPDM performs well for two agent-based simulations on a clustered platform. We hope that this work will form the cornerstone of a reusable toolkit for modeling and simulation.

## B.1 Introduction

Enterprise modeling, social simulation, system-of-systems war gaming, and related techniques simulate complex environments and systems (hereafter referred to as Human Dynamical Systems (HDS)) and compute observables which manifest themselves primarily as emergent phenomena. To increase fidelity of these HDS systems, increasingly large computations are needed and are accessible only through high-performance computing (HPC). Computer-based simulation of HDS has long been used within the military for war games, to carry out training, assess new technologies, and evaluate tactics. Recently, the Department of Homeland Security (DHS) and other federal agencies charged with disaster preparedness and response have embraced simulation for modeling a variety of complex phenomena including response to attacks by Weapons of Mass Destruction (WMD) and natural disasters. Sandia National Laboratories has developed a number of simulation applications to support different programs examining WMD countermeasures and concepts of operation, defense applications, terrorist networks, and economic consequences from natural disasters or terrorism affecting critical infrastructures. These simulation applications differ in several respects, yet they share an underlying commonality: all concern themselves with modeling complex ”systems of systems” where the system components are discrete (such as people, autonomous land vehicles, or companies) and interact in highly complex ways with other system components. Historically, modeling of HDS has not taken advantage of parallel processing hardware or techniques. There are at least two reasons for this:

1. The phenomena in question was not modeled at a fidelity that would require the extra complication of parallel computing.

2. The history of simulation of HDS has favored developer friendly environments, use of widely available computing hardware and software, and incorporation of sophisticated graphical user interfaces (GUIs) over speed of execution. This emphasis on rapid deliverables portable to a wide variety of platforms led to the use of virtual machine-based languages such as Java, Python, and Tcl over C and C++, because most programmers are much more productive in these languages. Most programming for high performance parallel processor systems has taken place in languages like C, C++, and (historically) FORTRAN because they give the programmer fine-grained control over allocation of memory and other processor resources.

Increasingly, however, HDS must reproduce significant phenomena/effects at credible fidelity while being fast enough to enable human-in-the-loop and timely batch-oriented analysis. Recently developed requirements for the next generation of social simulation applications have shown that the current capability, based on conventional computing architectures, falls short in both respects, requiring extension to HPC platforms using parallel processing. Also, improvements to Java and other virtual machine-based languages have made them more attractive in the domain of high performance computing.

We have developed components and tools for HDS simulation methods in a parallel HPC setting. In particular, we are addressing the challenges outlined above by creating a portable, scalable, and general engine for particle simulations as a facility for multi-use in agent-based simulations.

## B.2 Related Work

Many simulations have been created to study complex Human Dynamical Systems. Two excellent examples originally developed at Los Alamos National Laboratories and now hosted by Virginia Tech's Network Dynamics and Simulation Science Laboratory include TRANSIMS[3] and EpiSims[4]. TRANSIMS uses detailed urban travel data to model transportation networks and can model the effects that changes to those networks will have on traffic. EpiSims uses that same travel data to model disease epidemics and can be used to test public health mitigation strategies. Another example of work in this arena is Generative Social Science, which uses simulations to help discover the underlying dynamics of complex social systems[12]. In this work Epstein makes a hypothesis of the sets of rules that govern a given dynamic social system. An agent-based model (ABM) implementation of those rules is then created and the results are measured against historical data. The closeness by which the generated data matches the historical data gives credibility to the hypothesised rules.

As the modeling of HDS has grown in popularity, a number of reusable agent-based modeling toolkits have emerged to simplify the work of the developer. RePast[9] is arguably the most popular framework in the ABM community and provides developers with libraries for agent creation, event scheduling, and data charting and visualization. Mason[22] provides similar functionality but focuses more on light-weight models meant to be run many times for Monte Carlo-type studies. While these toolkits can be run in a parallel batch mode on a clustered environment, neither toolkit lends itself for use developing models that run singularly in parallel across a cluster of computers, thus limiting the size and complexity of such models.

The PPDM in its present form operates in a time-stepped fashion. Discrete event simulation (DES) provides a different time management paradigm for simulation. Instead of advancing time in globally synchronous fixed steps, DES advances the local simulation time to the exact time of the next local event. DES scheduling delivers events in timestamp order at all nodes allowing it to achieve performance gains if the simulated events are irregularly spaced in time or space. The greatest successes of DES to date have been in the areas of network simulation and war gaming. Several frameworks and systems have been created to support this. The High Level Architecture (HLA) standard[16] specifies an API for distributed DES that has seen wide use, particularly in work done for the Department of Defense. An example of parallel DES (PDES) used for social simulation is the SCATTER traffic simulator[28]. PDES scheduling has also provided time management for an agent-based simulation system[19]. Applications using PDES have scaled to over 1500 nodes[15]. In an exceptional scaling exercise, Perumalla[29] ran a test program on 16384 nodes of a Blue Gene supercomputer. Though the PPDM in its present form makes no use of DES concepts, we envision future DES extensions. To our knowledge, no agent-based simulation system in widespread use provides the distributed synchronization needed for true PDES scheduling.

In a work that highly influenced our design, a more traditional particle-in-cell based, time stepped model was used for simulating the evacuation of large crowds in a building environment[30]. In this work, the Social Forces algorithm was implemented such that the simulated geometry and populations were divided up amongst a cluster of computers. In [31] a similar approach was used for large crowd simulation but made novel use of the PlayStation 3's high performance Cell archi-

tecture.

## B.3 Architecture

In our work modeling various HDS, we identified a great deal of functional commonality in our simulation codes. We also observed that future simulations in the problem domain would be growing in population size and complexity. This led us to create the Parallel Particle Data Model (PPDM), a Java-based, reusable programming toolkit that supports HDS simulations and can scale to a large number of processors.

The HDS simulated by Sandia are generally composed of large populations of agents (people, vehicles, etc.) spread across a geography of segmented locations such as cities, census tracts, or even households and businesses. Each location contains a set of agents that are able to interact with each other as well as with agents in neighboring locations. The agents can move between locations as time evolves. Various modules of the simulation need to update or gather information from the agent populations in different ways based on the phenomena they model. Examples include infection of people due to a biological hazard plume or detection that a car should trigger a radiation sensor alarm.

We designed the PPDM to accommodate this method of interacting with the simulated populations while operating in a distributed environment. The PPDM uses a particle-in-cell (PIC) data structure which divides the simulated geography into cells that contain populations of agents. The PPDM then distributes these cells across a number of processors. Agents directly affect other agents residing in the same or neighboring cells. Non-neighboring particles can interact through special, less efficient mechanisms.

Even though Java is not renowned for its high performance computing capabilities we chose it as the PPDM's implementation language. The primary reasons were to support legacy code, to utilize existing programmer skill sets and to provide ease of development. Evolving improvements to the JVM's performance and extensive open source code libraries, including several new high performance communication libraries, have also increased its attractiveness. The end result is that Java tools can provide a portable and easy to use framework for high performance computing.

To make use of the PPDM, the user must first define the geometry of the problem domain. Currently, we have implemented two types of geometry. A user can specify either an n-d lattice or a graph of cell locations. Along with the geometrical structure, the user must define the number of functional patches into which they would like the geometry broken. A patch represents a grouping of cell locations that are guaranteed to be collocated on the same processor. Each processor is allocated a number of patches to process. Modelers write their algorithms and agent update code to work on a patch-by-patch or cell-by-cell basis which enables the PPDM to distribute the user's program across many processors.

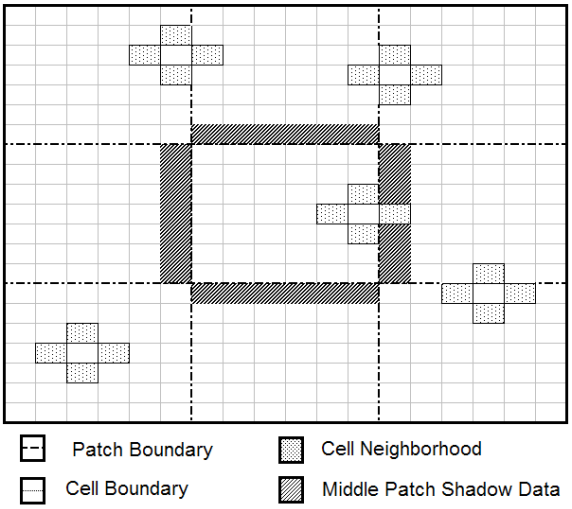


**Figure B.1.** Domain Model

### B.3.1 Data and Particle Fields

After specifying the domain geometry, modelers define multiple data and particle fields across this geometry. A data field holds a single unit of data (such as an int or float) for each cell in the domain, while a particle field allocates to each cell in the domain a list-like data structure called a particle set. The particle set data structure allows the modeler to store and organize agents and provides other useful utility functions. In creating a data or particle field, the modeler specifies a locality stencil for that field. This defines each cell's local data neighborhood within the overall defined geometry. In the case of 2-d lattices, a modeler will commonly specify a 4 point stencil in which a cell's neighbors include adjacent cells to the north, east, south, and west. In a graph geometry, on the other hand, the graph structure directly determines cell neighborhoods. The local neighborhood defines a region of shadow data that each patch must make available to the cells it maintains. Shadow data is a read only copy of neighboring cell's data. After doing a shadow update for a particle field, each patch will have the data for the cells it maintains as well as a read-only copy of data from its neighboring cells that are maintained by other patches. If a patch has neighboring cells that are located on another processor, data communication must take place. The transfer of shadow data enables the simulation to update agents' state using data from their own cell as well as neighboring cells. This method is similar to that used in a parallel implementation of the social forces model [30]. The shadow data construct is optional. If a particular domain does not need to have neighbor cell interactions this feature can be disabled. To summarize, a processors will be allocated a number of patches. Each patch is composed of a number of cells. Each cell contains a number of particles or agents.

While processing the agents, modelers also have the ability to schedule the movement of agents from cell to cell. This cell movement can span patches and processors. The PPDM takes care of the actual movement of the agents and transfer of shadow data. If the modeler guarantees that particle movement only occurs between neighboring cells, the PPDM can make optimizations that reduce

the amount of communications needed between processors, thereby allowing greater speedup. The PPDM's capabilities hide a lot of parallel processing complexity from the modeler.

To better enable parallel processing, the PPDM limits the modeler's interaction to the data and particle fields by allowing only cell-by-cell or patch-by-patch access to the data. A modeler's algorithms have full access to all agents owned by the patch they process, as well as read access to agents of neighboring cells of that patch. The PPDM's feature set encourages modelers to build their algorithms with locality in mind. This allows the algorithms and data to be more easily and efficiently distributed across multiple processors. From our experience in prior simulations, this restriction is acceptable. We also offer various convenient yet less efficient mechanisms to handle global communication between processors.

Several patterns for interacting with the particle populations are available. The most common method of particle access is to submit particle algorithms to the PPDM. Particle algorithms are data structures that implement the visitor pattern. When particle algorithms are submitted to the PPDM, the PPDM duplicates them across all processors and schedules them to be executed at regular intervals. The algorithms visit each cell or patch owned by the processors on which they are located and perform their needed operations on the contained agents. The particle algorithms can optionally coordinate with their copies on other processors and send back aggregated data to their parent modules or other subscribing particle algorithms.

In our simulations, various modules work together in a decoupled manner by submitting particle algorithms that enforce their desired behavior or area of concern on the agents. This deviates from standard agent-based techniques, which initialize the agents with a given behavior and state and allow the agents to evolve their state. This difference arises because the PPDM grew out of our experience with prior simulations composed of separate modules that ran on their own Java virtual machine. Each of these modules needed to interact with the population. Rather than moving the population data to each of the modules, we allowed the modules to move their code to the population data. This is similar to how distributed systems interact with a common database through sql or stored procedure calls. One can duplicate the more traditional agent update behavior in the PPDM through the use of a simple particle algorithm that calls an update method on all or a selected number of the agents at scheduled intervals. The agents themselves would contain all of the logic needed to interact with neighboring data and agents and perform all necessary updates and movement. Another alternative would be to have the various modules inject differing behavior strategy objects into the agents upon initialization which will eventually be called during updates. Dynamic languages like Ruby would provide an ideal environment for this technique.

We are also looking into incorporating patch-based event queues into the PPDM to allow for algorithms or agents themselves to make use of the discrete event simulation programming model[16]. To date, our simulation needs have not warranted the added synchronization and bookkeeping issues that DES will require.

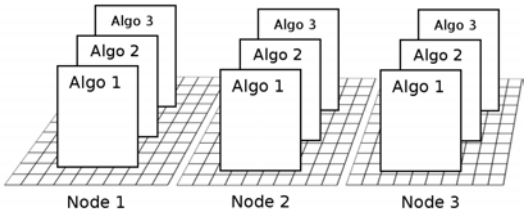


**Figure B.2.** Visitor Algorithms

### B.3.2 Communication implementation

The PPDM uses the MPI model for parallel data communication. Communication occurs between copies of particle algorithms running on differing patches, when agents move between cells owned by different processors, and to update shadow data located on neighboring processors.

When necessary, the particle algorithms communicate and coordinate with each other across patches using standard parallel processing messaging such as the MPI broadcast and reduction methods. We have provided modelers with simplified interfaces to these functions. We have also implemented a messaging service to allow agents to communicate with other agents that are not located on their patch. The PPDM organizes and sends messages using a regularly occurring batch update process. The messaging service is optional and when in use it requires additional overhead because it must track agent locations to enable routing.

The PPDM handles particle movement during a regularly scheduled batch update where all processors work together to redistribute agents to the correct locations. We handle copying of shadow cell data for neighboring cells in a similar manner. This requires the various processors to remain synchronized in time.

Modelers can configure the PPDM to make use of mpiJava [2] or MPJ Express [23] for the MPI services. Both of these are object-oriented implementations of the MPI protocol using the Java programming language. They differ in that mpiJava provides a JNI wrapping over a native MPI environment (such as MPICH or LAM), while MPJ Express is a complete implementation of the MPI protocol using only Java. We found MPJ Express to be far more portable and easier to use since it removes the native MPI compilation and integration steps. In Java, serialization of object data to byte data is a major cpu cost. We hope that custom serialization code will reduce some of this cost and are looking into the serialization and communication libraries from the Ibis project[7].

## B.4 Example Simulations

In this section we demonstrate the use of PPDM in two different examples and conclude with an illustration of its parallel capabilities. The problems have been formulated in an agent-based

fashion, but carry a PIC flavor in the sense that agents/particles are collated into "bags"/cells with in-cell (in-bag/near-field) behavior modeled at a far greater fidelity than the far-field (out-of-bag) behavior.

### B.4.1 Spread of disease in a 2-species population

This example approximates the spread of a disease in two separate species, humans and (non-human) primates, with primates acting as a reservoir for the pathogen. The pathogen, modeled loosely on Ebola, is communicable within each of the species and is also capable of primate-to-human jumps. This example models both humans and primates as individual agents. A number of primates and humans are collocated into settlements where humans may come in contact with primates and contract the disease. Both humans and primates may move between cells, though primates' movements are significantly more circumscribed than humans.

#### B.4.1.1 Disease model

In this work we use a compartmental disease propagation model. Both the humans and the primates go through five compartments/states of health - Susceptible(S), Exposed (E), Mildly Symptomatic (MS), Severely Symptomatic (SS), and Recovered/Dead (R/D). The residence time in each of these states of health is a random variable, which follows a distribution (usually log-normal). The process of infection of humans is governed by their contact with other humans, cadavers and primates, i.e. it depends on their closeness of contact and the probability of meeting an infected individual, primate or cadaver. Transmission within the primate population is entirely by contact. We will assume that both the Mildly Symptomatic and Severely Symptomatic stages are contagious, for both humans and primates.

#### B.4.1.2 Movement model

We assume that there exist $M$ human settlements, each containing a number of primates. Interactions between human and primate occur between those present in the same settlement. The human and primate populations in each settlement are drawn from the log-normal distribution $\mathscr{L}(N_{median}, S)$, where $N_{median}$ is the median and $S$ the standard deviation. In each settlement we use a median of 1000 and standard deviation of 500 for the human population and a median of 50 and standard deviation of 12.5 for the monkey population.

For each settlement we define a mobility factor $Mh$ and $Mm$ for the human and monkey populations. This mobility factor determines the percentage of each population that will relocate on a given movement step. The indexing of settlements reflects geographical proximity. Primates travel to other human settlements primarily because of proximity. Once the connection routes are determined, a settlement's mobile population is distributed across its connecting settlements using the above probabilities as a weighting factor.

The model can be configured with two modes of population movement. In the first, agents are assigned a fixed day-night routine. In the second mode, agents move randomly several times a day to connected settlements using the above probabilities and weights, and at night they return to their home settlement. Mildly symptomatic individuals move, but severely symptomatic ones do not.

#### B.4.1.3 Disease Model Implementation and Scaling

The disease model initializes the PPDM with a 2-d lattice geometry. The model represents each settlement by a cell in the lattice where a cell is given an x and y location to determine its distance from other cells. We initialize two separate particle fields for the human and monkey populations. We submit four configurable particle algorithms to the PPDM to evolve the disease agents. These algorithms create the initial population distributions, initialize the agent movement patterns, perform agent movement at a set rate, and evolve the disease state of the agents. The algorithms are modular and can be replaced with different implementations of varying complexity to study how changes to the population or disease dynamics effect the outcome of the epidemic. For this model, two types of inter processor communication take place. During agent movement some agents will need to relocate to neighboring processors, and after each disease update the master disease algorithm coordinates with its neighbor algorithms to output the aggregated disease statistics for that time.

We performed experiments testing both the strong and weak scalability of this model on the PPDM. In the strong scaling test, we initialized the population with a fixed 2,048 settlements, each containing an average population size of 1,000 for a total population of 2 million agents. We then varied the number of processors allocated to the PPDM and ran the simulation for a simulated 2 months. Figure B.3 shows the results of the strong scaling test. The results show near-ideal scaling up to 16 processors, after which the particular problem setup begins to scale poorly. This poor scaling results because we keep the problem size fixed. As we increase the number of processors, each processor has less to do and communication and synchronizations costs take over.

In the weak scaling test, we initialized the population with a fixed 2,048 settlements. We then scaled the population size up linearly with the number of processors allocated to the PPDM so that each processor would maintain a fixed number of agents. We ran the model for a simulated 2 months. The Estimated Single-Node plot line represents the expected time that a single node would take to perform the problem size if it had infinite memory and did not succumb to non-linear effects. Under the weak scaling case where the problem size grows with the number of allocated processors, we achieve fairly good scaling with the 64 node case within twenty-five percent of the ideal result. We see this good scaling because the communication load for each processor is balanced by the increasing computational load. These results show that we can scale up our modeled size by increasing the available number of processors. Figure B.6 shows a further exploration of the scaling data space. For a given processor allocation we plot the timing response for problems of increasing scale. The problem size represents a linear scaling of the population allocated to each cell while the number of cells remains fixed. Missing data points exist when a problem size is to large for the number of allocated processors.

To further test the weak scaling of the PPDM, the disease model's problem growth strategy was modified. In this case as the number of processors was increased the population per cell was kept constant but the number of total cells allocated was increased linearly. This led to a fixed number of cells and people allocated per processor. The simulation run time was also increased to 3 months to mask some additional initialization costs. Figure B.5 shows the resultant weak scaling numbers. Figure B.8 shows additional plots of the cell based scaling of the disease model. Surprisingly, the scaling results are quite a bit worse for this case. Even though the scaling results show that the problem complexity does scale linearly for a given processor allocation, the run times for a given problem size are much greater than ideal between processor allocations. We have yet to determine the cause for this deviation from the previous weak scaling case as the communication loads between processors are similar. Despite the worse results, this level of scaling is still useful because it allows one to run problem sizes much larger than what can be run on a single processor.

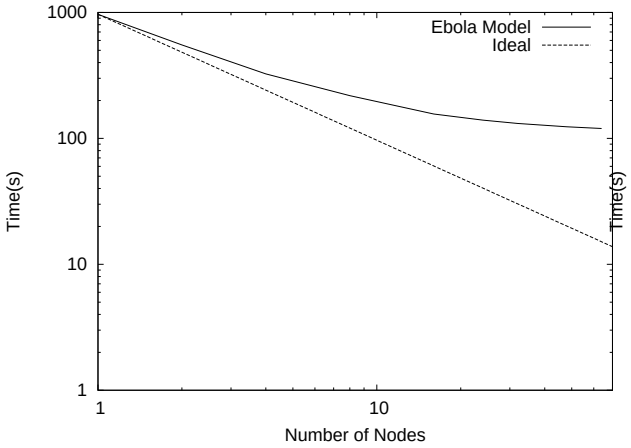


**Figure B.3.** Disease Strong Scaling

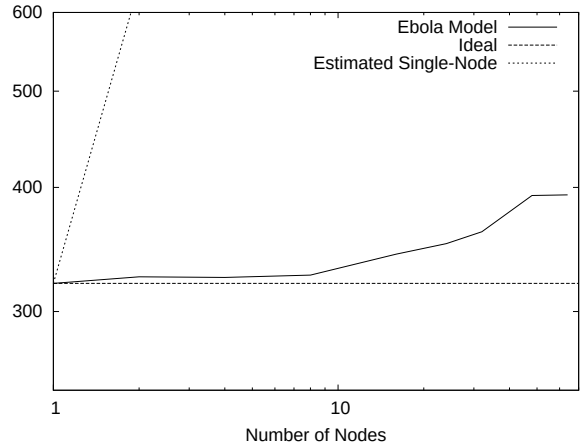


**Figure B.4.** Disease Weak Scaling (Cell Size)

## B.4.2 Seldon

Seldon is a software toolkit that combines technology and concepts from a variety of different research areas, including psychology, social science, and agent-based modeling and simulation. It has been used to study urban gang recruitment and terrorist network recruitment. This second example demonstrates the utility of Seldon and the PPDM in analyzing the effect of media on populations.

Previous uses of Seldon involved rather few ($O(10^3)$) agents, and serial implementations generally sufficed. However, large agent populations were required to study the effect of media, which in turn required parallel computing. Further, the individual agent models were enhanced to include a cognitive model (essentially a semantic graph of concept activations and edge weights), allowing a realistic processing of media information. This led to an increase in the computational intensity

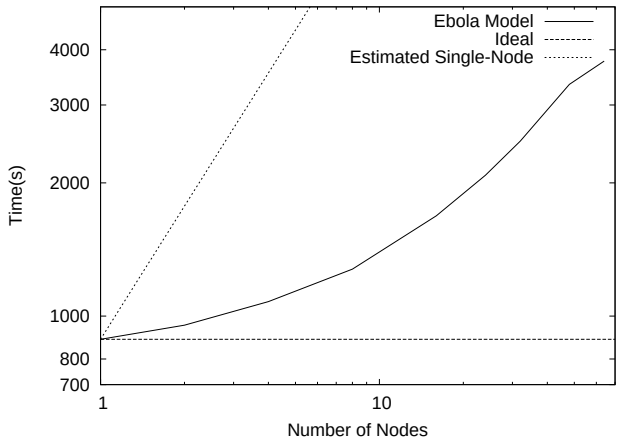


**Figure B.5.** Disease Weak Scaling (Num Cells)

of individual agents, further spurring the need for parallelization.

Seldon has two types of agents: individuals and abstract. Abstract agents represent social or institutional concepts that can influence an individual (e.g. schools and mosques). Since they contain a set of individual agent members, they can be highly-connected nodes in the overall structure.

An interaction between two agents involves significant processing. Sets of attributes are exchanged and modified according to linear attraction and reinforcement rules. Concept activation vectors are also exchanged, causing nodes to fire in the semantic graphs, and thereby changing their cognitive states.

The simulation commences with an unconnected set of agents. Homophily is used as the basis of attraction, and relationships form as interactions proceed. Each agent has a maximum amount of relationship energy, thus providing a flexible cap for either a large number of weak relationships or a small number of strong relationships. Agents also have personality factors, which affect their interactions. For instance, an extroverted agent might interact more than an introverted agent. As relationships evolve, social networks form, ranging from acquaintances to cliques. These, in turn, drive subsequent interactions, so that agents are more likely to interact with close friends in cliques than acquaintances.

Each timestep in the algorithm consists of a couple of steps. First, the individual agents determine their membership with the abstract agents, and then the interactions occur between them. Next, the individual agents identify other individual agents with which to interact. The interaction procedure consists of three parts: *send*, *receive*, and *respond*. Agent A *send*s a subset of its information to Agent B, who then *receive*s the information, compares it to its own, and *respond*s to Agent A. The procedure is transactional, so both agents change their emotional state, or both stay the same. Interactions occur between agents through the creation of message objects, and

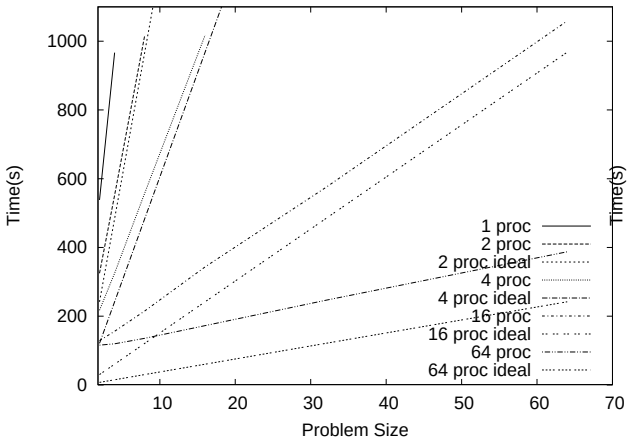


**Figure B.6.** Disease Performance (Cell Size)

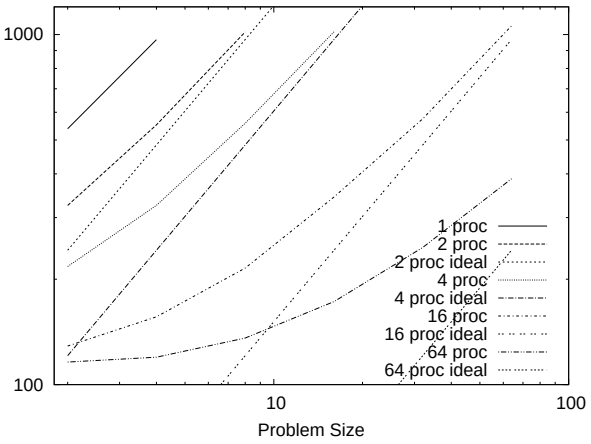


**Figure B.7.** Disease Performance (Cell Size log/log)

large numbers of messages are routed and delivered concurrently. Since *receive* messages can create *response* messages, processing continues until there are no more messages. This barrier synchronization ensures that each step finishes completely before the next step starts.

We parallelized Seldon by decomposing the problem across processors in a load-balanced manner. We also maximized the likelihood of intraprocessor communication by using Zoltan [11], a load-balancing library, to invoke graph-partitioning algorithms in ParMETIS [21]. Zoltan uses the social network structure (with relationship strengths as edge weights) to calculate the optimal agent-to-processor mapping. It also provides a distributed directory capability to track these mappings for routing. The data migration is performed separately and involves packing and unpacking agents at the source and target processors, similar to the process of message delivery. When repartitioning, Zoltan exploits the current decomposition to reduce data migration. An additional difficulty involved multi-language integration, since Zoltan is implemented in C/C++ and Seldon is implemented in Java. We created a JNI wrapper around Zoltan using Swig [5], which then provided access from Seldon.

To study scaling characteristics, we used a cognitive model with agents for individuals and media outlets to simulate the shift in public opinion in Amman, Jordan, after the November, 2005, bombings. The individual agents were provided with cognitive information automatically generated from Jordanian newspapers published before and around the time of the bombings. The weak scaling runs held the number of individuals constant at 1000; the strong scaling runs used 1000$P$ individuals, where $P$ is the number of processors . There were 40 media agents in all runs.

In Figures B.10 and B.11 we plot results from strong and weak scalability studies. It is clear from the weak scalability analysis (Figure B.11) that there is little locality in the agent-interaction pattern - cross-processor communication costs increase as the processors (and the total problem size) are increased. The strong scalability run is somewhat more promising, following a quasi-

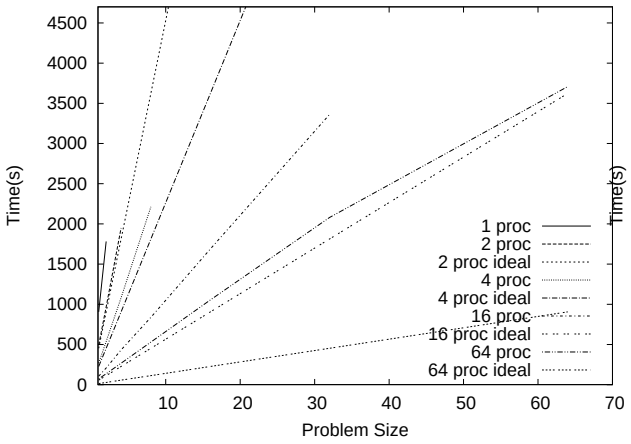


**Figure B.8.** Disease Performance (Num Cells)

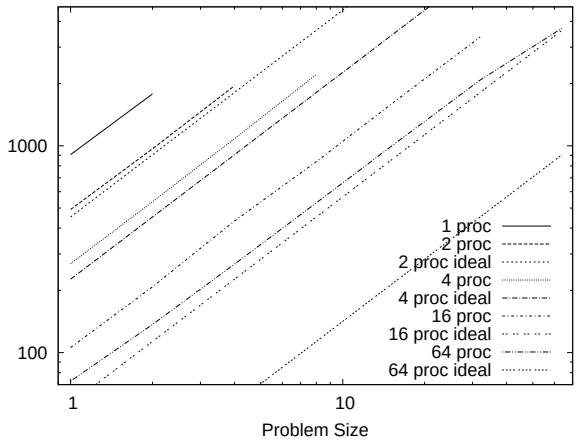


**Figure B.9.** Disease Performance (Num Cells log/log)

ideal convergence till around 10 nodes; thereafter divergence from ideal is abrupt, as the high communication costs (observed in the weak scalability analysis) overwhelm the steadily decreasing computational costs. At a point (around 30 nodes), the communication costs dominate and a follow the trend observed in the weak scalability study (Figure B.11) where communication costs are roughly proportional to the number of processors.

## B.5 Integration

We integrated the PPDM with several existing frameworks on an experimental basis. We used the PPDM as an extension of an HLA federate [16] to simulate a moving population in a large metropolitan area. The PPDM ran on a cluster, the HLA federation ran outside the cluster, and the two communicated via remote method invocation (RMI). We separated the components in this way because of differing platform requirements. We also used the PPDM as the underlying data structure for one of the demo application models of ths RePast Agent-Based Modeling framework [9]. A similar approach to adding parallelism to RePast can be found in [27]. We had to make some changes to the RePast `Schedule` class to maintain synchronization, allow for agent movement, and shadow agent updates between processors. We scheduled particle movement and update at fixed intervals on all nodes.

## B.6 Conclusion

Agent-based modeling and discrete event simulation are arguably the standard tools for understanding and making predictions about complex systems. To some degree both ABM and DES make the argument that, unlike statistical methods, simulations must be performed at scale. For such social systems, there is no *a priori* idea of a scaling law or average that would predict their emergent behavior. Because the phenomena simulated are usually large and complex, and because they must be computed at scale, parallel high performance computing is required to enable successful simulations of social systems.

We hope that our work in developing the Parallel Particle Data Model (PPDM) instigates the development of a library of general purpose components for large-scale entity modeling. From a software engineering point of view, separable components for ABM/DES will aid repeatability and methodical experimentation. Beyond reusable software to speed software development, the ability to change out entity models and even the framework and schemes for updating without changing the parallelization scheme is an important contribution of the PPDM. Often when comparing two different models that purport to acheive the same result, the phenomenological particulars of the solutions differ in so many ways that comparisons are difficult. Separating out the parallel implementation as we have done in the PPDM, hopefully will make comparison between different models easier in the future.

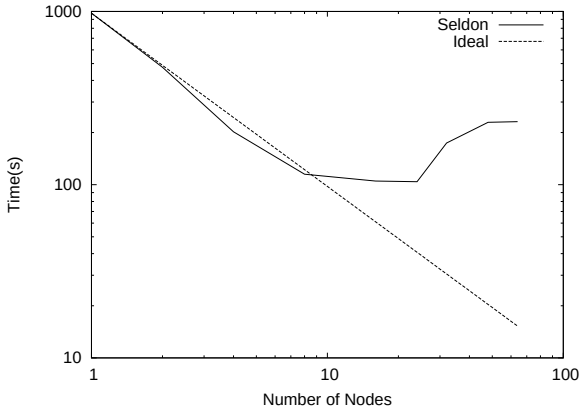


**Figure B.10.** Seldon Strong Scaling

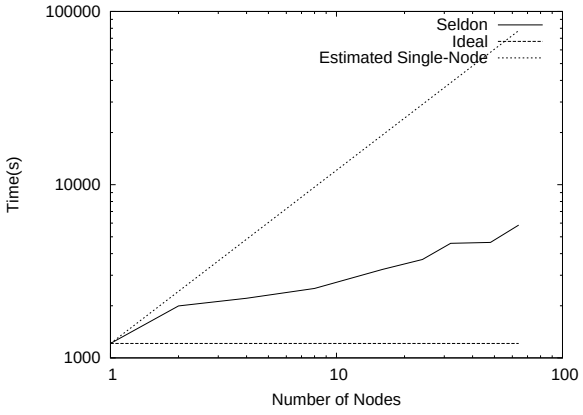


**Figure B.11.** Seldon Weak Scaling

# DISTRIBUTION:

| | | |
|---|---|---|
| 1 | MS 0763 | Donna L. Chavez, 0123 |
| 1 | MS 9151 | Howard H. Hirano, 8960 |
| 1 | MS 9159 | Heidi R. Ammerlahn, 8962 |
| 1 | MS 0899 | Technical Library, 9536 |
| 1 | MS 0899 | Technical Library, 9536 (electronic) |

Sandia National Laboratories