Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters

Brandon G. Aaby, Kalyan S. Perumalla and Sudip K. Seal
Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
perumallaks@ornl.gov, sealsk@ornl.gov

ABSTRACT

An effective latency-hiding mechanism is presented in the parallelization of agent-based model simulations (ABMS) with millions of agents. The mechanism is designed to accommodate the hierarchical organization as well as heterogeneity of current state-of-the-art parallel computing platforms. We use it to explore the computation *vs*. communication trade-off continuum available with the deep computational and memory hierarchies of extant platforms and present a novel analytical model of the tradeoff. We describe our implementation and report preliminary performance results on two distinct parallel platforms suitable for ABMS: CUDA threads on multiple, networked graphical processing units (GPUs), and *pthreads* on multi-core processors. Message Passing Interface (MPI) is used for inter-GPU as well as inter-socket communication on a cluster of multiple GPUs and multi-core processors. Results indicate the benefits of our latency-hiding scheme, delivering as much as over 100-fold improvement in runtime for certain benchmark ABMS application scenarios with several million agents. This speed improvement is obtained on our system that is already two to three orders of magnitude faster on one GPU than an equivalent CPU-based execution in a popular simulator in Java. Thus, the overall execution of our current work is over four orders of magnitude faster when executed on multiple GPUs.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance – *Simulation*, *Operational Analysis*; D.4.4 [**Operating Systems**]: Communications Management – *Buffering*, *Message Sending*, *Network Communication*; D.4.8 [**Operating Systems**] Performance – *Operational Analysis*; I.6.1 [**Simulation and Modeling**] General; I.6.3 [**Simulation and Modeling**] Applications

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Agent-based simulation, GPU, Cluster, Threads, MPI, CUDA, Latency hiding, Computational hierarchy, Multi-core

1. INTRODUCTION

The study of human social behavioral systems is finding renewed interest in many applications including military, homeland security, and socioeconomic scenario analyses. Simulation is the most generally applied approach to studying such systems. While computational social science has been actively studied for over three decades, experiments in computational social science so far have only been at small scales – a few thousands of interacting entities [1-4]. Lately, there has been a general surge to represent and capture detailed effects at much larger scale, such as at population counts of cities, states, nations or even the world (10^6-10^9) [5]. Computational aspects that were not prominent at smaller scales are now becoming pronounced at large scales.

1.1 Computational Challenge

Emerging computational platforms are being built with compounds of hierarchical processing elements. For example, clusters of commodity nodes with multiple graphics cards afford multiple levels of tightly and loosely coupled processing elements, with a variety of memory access types and synchronization primitives. Processor clusters, with each processor containing many cores, are another commodity platform that affords high performance albeit with a different type of execution hierarchy.

Inter-element communication latencies are also varied, ranging from nanoseconds to hundreds of microseconds. For example, threads within a block of NVIDIA's Common Unified Data Architecture (CUDA) have very fast access to a shared memory segment, whereas Message Passing Interface (MPI)-based communication across GPU nodes typically consumes hundreds of microseconds.

The challenge is compounded by the fact that computation within each agent's state update in an ABMS can be very fine-grained, taking little more than a few microseconds. When states are decomposed across the hierarchies, synchronization across time-stepped updates to the partitioned states can become a significant source of overhead.

A solution is needed to *simultaneously* address the challenges of latency spectrum, hierarchical organization as well as heterogeneity. Ideally, a single, unified, parameterized solution would be useful that can be easily instantiated, customized, and auto-tuned for any given, specific compound computational platform instance. This paper presents preliminary results from one such attempt.

1.2 Related Work

Several modeling frameworks are available for modeling and simulating social systems such as NetLogo[1], Mason[6], Repast J/.Net[3], Swarm [7]. SPADES [8], JAMES [9], and HLA_AGENT [10]. Also, GPUs have been recently used for ABMS [11, 12].

While parallel execution has not been a major focus of ABMS toolkits in general, a few recent systems have explored parallel/distributed implementations. These include SEAS [13] for disaggregate and aggregate behavioral models interacting with

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

Copyright 2010 ICST, ISBN 78-963-9799-87-5.

actual individuals, and a distributed agent simulator [14]. An agent-based simulation optimized for large shared-memory platforms is described in [15] and a parallel, Java-based agent simulation system is described for disease propagation in [16]. The dynamics of multi-agent based simulation execution on grid environments was analyzed in [17, 18]. By contrast, our focus is on high-performance computing and on heterogeneous platforms, with special emphasis on latency hiding for maximal concurrency.

The problem of performance optimizations of stencil based computations – an area of active research for years – bears resemblance to the parallel execution challenge of ABMS. For example, "ghost cell expansions" (GCE) [19] was proposed for performance improvements for a two dimensional synthetic problem without any time evolution. Automatic parallelization of stencil computation was reported in [20] in the context of a one dimensional Jacobi code on a 32 node (single core) platform. More recently, a detailed empirical study of stencil computation optimization on several multi-core CPU based architecture (but restricted to a single GPU) was reported in [21]. Another related work is a framework for high-order stencil computations [22]. Although similar in some ways, stencil-based computations are distinct from parallel agent-based simulations, as discussed next.

Studies on stencil-based computations are generally based on constant-sized neighborhood dependencies that remain static throughout an execution. Availability of data at neighbor locations is guaranteed at each time step. This is not true for ABMS where the agents are mobile. In ABMS, the region of data dependency is not fixed and can potentially span the whole computational domain. This results in highly non-trivial spatial and temporal data dependencies. Consequently, the communication-computation tradeoff strategies discussed in the stencil computation literature do not necessarily carry over to the needs of ABMS. Existing work on performance optimization of stencil based computation, as detailed above, has largely been empirical in nature [21] and focused on synthetic, reduced-dimensional problems [19, 20].

Stencil optimization strategies have neither focused on nor exploited the full hierarchical organization of current processor platforms and memory architectures. We are also not aware of work that proposed a generalized solution that applies equally well across a variety of architectures (such as CPUs and GPUs); our approach does apply. We are also not aware of prior work that can apply the same template recursively at multiple levels of computational hierarchy, with varying characteristics of memory latencies and capacities, processor speeds, and network latencies and bandwidths. Our focus is on developing a single solution that can be reused despite variations in target platform characteristics due to heterogeneity and hierarchy.

Automatic ghost zone optimization [23] also addresses latency problems in stencil computations. However, unlike our approach, it is not generalized to heterogeneous, deeper hierarchies of computation and communication architectures.

1.3 Contributions

Here, we present a latency-hiding mechanism designed to exploit and seamlessly adapt to the hierarchical organization and heterogeneity of emerging high performance computing platforms. We call it the "*B+2R latency-hiding scheme*." It is based on the well-known principle of computation *vs.* communication tradeoff (or, the duplication of some computation to gain some concurrency to offset communication latencies).

While being simple to articulate, it is rather complex to implement in heterogeneous platforms. For example, while concurrency considerations require larger cached-block sizes, memory limitations constrain the cached block-size; this conflict of considerations needs to be addressed in implementation carefully. As an example, we had to address this conflict in our CUDA-based implementation, in which traditional ping-pong approach of read-write buffer swaps across iterations limited the size of blocks that could be handled by each thread or block within the limits of shared memory. Once implemented, despite implementation complexity, however, it is relatively easy to fine tune for optimal performance on a variety of platforms. The scheme also affords excellent performance even in the most challenging ABMS scenarios characterized by very fine computation granularity.

To the best of our knowledge, the work reported in this article is among the first to execute ABMS on multiple GPUs communicating over a network. It is based on a novel analytical model (discussed in a latter section) that is applicable to arbitrary levels of computational and memory hierarchy.

The analytical model proposed here reduces to previous models on computation vs. communication tradeoff on stencil-based computations [19] while validating, both analytically and empirically, the degradation of the payoffs [24] with increasing expansion levels. We also believe that this is among the first ABMS to execute multiple regular (CPU-based) threads over distributed memory platforms, optimized to sustain fine granularity.

We present our preliminary findings in the context of a well-known ABMS benchmark application as well as a complex model of current interest in social sciences [25], to demonstrate significant runtime improvements via latency hiding.

2. LATENCY HIDING SCHEME

2.1 Latency Problem

ABMS toolkits typically provide an interface in which agents are organized in a grid, and agents interact with each other, typically within some specific distance of reach in a neighborhood region. As with other grid-based models, due to partitioning of the global state across processors, the state of adjacent cells in the neighborhood of some cells may be remotely located outside of that processor. In time-stepped parallel execution of agent-based simulations, copies of off-processor neighbor states are fetched and used within a time step. A synchronization primitive such as parallel barrier is used to align all processing elements after every time step. The problem in scaling this approach is that the communication and synchronization costs can become quite large when hierarchical, heterogeneous computing elements are used, resulting in large slowdowns as opposed to speedups for fine-grained agent models. As will be seen from performance data in later sections, the naïve approach of synchronizing after every iteration is vastly sub-optimal. A technique is needed to offset this cost, and hide the large inter-element latencies.

2.2 Our Solution Approach

Given a grid of agents, we can logically separate the grid into dependent *blocks* allotted to independent processing elements. These blocks clearly have data dependencies across each other as agent state updates bordering a given block depend upon the current state of agents in neighboring blocks. Here, we use an approach in which sub-grids of this grid, *B*, can be padded on the sides by *R* layers of surrounding data. These *R* layers of surrounding data encapsulate remote agents to be simulated by neighboring blocks allotted to other independent processing elements. Computation on local agents can then be increased by *R* iterations before having to re-synchronize with off-processor cells. Since a given *B+2R* block captures all surrounding data for local simulation, communication between nodes is also decreased. The *R* layers induce resilience to error locally, and thus offer latency hiding, both in terms of communication (exchanging data fewer number of times, albeit a larger amount of data per exchange) and synchronization (synchronizing less often, only once every *R* iterations).

As a simple illustrative example, Figure 1 shows a *3×3* grid separated into blocks to be processed by *P* processing elements in two different contexts, with and without the latency hiding scheme implemented. $\text{Block}_{1,1}$ of the conventional approach (Figure 1) simulates *B×B* agents with communication between boundary agents necessary at every simulation time step. By contrast, $\text{Block}_{1,1}$ of the latency hiding scheme (Figure 2) simulates a larger number of agents *(B+2R)×(B+2R)*, but requiring less frequent communication with neighboring blocks, only once every *R* time steps.

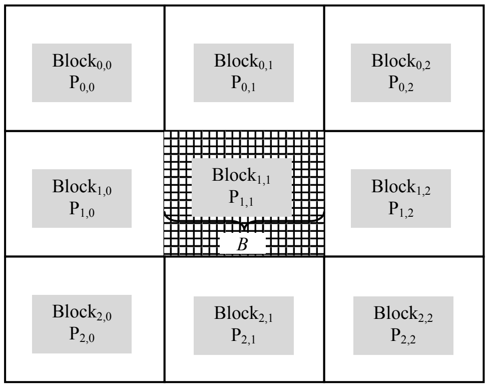


Figure 1: Traditional approach with synchronization between every time-step

Figure 3 illustrates error propagation at successive (up to *R*) simulation time steps. Error propagates inward, one layer per iteration, but never enters the central *B×B* block that is mapped to this processing element. Thus, after *R* iterations, a valid *B×B* block remains at the center, evolved by *R* iterations, for subsequent synchronization with neighboring blocks.

For a given grid split into *N×N* logical blocks, this scheme is implemented homogenously across all blocks. Once again, only after exactly *R* iterations is synchronization necessary. This synchronization is the gathering or scattering of agent state information to neighboring blocks

Data allotted to and surrounding a given block's valid *B×B*, *i.e.* $((B+2R)^2-B^2)$, is refilled with state information from neighboring blocks' *B×B*. Subsequently, execution can continue for another *R* time steps before this synchronization is required once again.

With this conceptual framework, we present a simple algorithm in Figure 4 by which simulation continues. Referring to Figure 3, it is necessary to only update the largest data square containing valid, correctly-simulated agents. After *i* iterations, we are required to update a square of size *(B+2(R-i))×(B+2(R-i))*. Note that in this algorithm *update* and *communicate* are implementation-specific. These will be further discussed in our implementation and benchmarks.

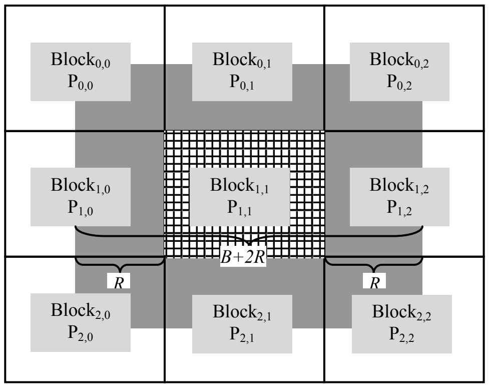
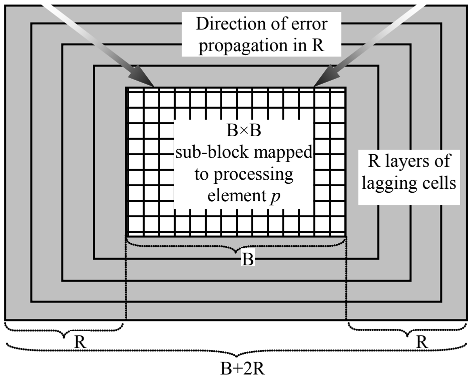


Figure 2: Our B+2R scheme for latency-hiding to sustain multiple time-steps per synchronization



Direction of error propagation in R
B×B sub-block mapped to processing element *p*
R layers of lagging cells
B
R
R
B+2R

Figure 3: Error propagation at consecutive simulation time-steps

```
Let $T_e$ be total number of iterations in the simulation
1 For all blocks $\text{Block}_{ij}$ in the given agent grid G
1.1 Let ($t_{li}$, $t_{lj}$) be the top left index of $\text{Block}_{ij}$
1.2 Let ($b_{ri}$, $b_{rj}$) be the bottom right index of $\text{Block}_{ij}$
1.3 For t=0 to $T_e$/R
1.4         For r=R-1 down to 0
1.5                 Update( $t_{li}$-r, $t_{lj}$-r, $b_{ri}$+r, $b_{rj}$+r )
1.6         Communicate( $t_{li}$, $t_{lj}$, $b_{ri}$, $b_{rj}$, r )
1.7         Barrier()
```

Figure 4: Generalized latency-hiding scheme

2.3 Analytical Model

Let *F* be the total run time for a logical block of size *B*. Then, *F* is the sum of computation time F_c and communication time F_m for *R* simulation time steps:

$$F \equiv F_c + F_m$$

For *R* iterations, each subsequent iteration needs to only update remaining valid data, as previously discussed. Thus, total computation time is the aggregate time to update progressively smaller blocks. Note that f_c is the computation cost function for a given block size of agents.

$$F_c \equiv f_c(B+2R) + f_c(B+2\overline{R-1}) + f_c(B+2\overline{R-2}) + \cdots + f_c(B+2\overline{R-R})$$

The computational time increases as the square of the grid size. Therefore, if a is an implementation-specific computation constant,

$$f_c(x) = ax^2 \Rightarrow F_c \equiv \sum_{i=1}^{R} f_c(B+2\overline{R-i})$$

$$= a\sum_{i=1}^{R} (B+2\overline{R-i})^2 = a[\frac{4}{3}R^3 + (2B+2)R^2 + (B^2+2B+\frac{2}{3})R + B^2]$$

Now, the communication cost required after R time steps must be expressed separately for CPU and GPU platforms due to configuration-specific details. For example, in the case of the CUDA environment for GPU, block's shared memory is flushed immediately after kernel invocations. We represent these as F_{m_1} and F_{m_2} respectively (CPU and GPU) where f_w and f_r are general write and read representations for communication cost between blocks for both CPU and GPU. Let b and c be platform-specific communication overhead constants (typically determined empirically). Communication cost is expressed (either read or write) as f_{wr} below.

$$f_{wr}(y) = by + c$$

$$F_{m_1} \equiv f_w(B^2 - \overline{B-2R}^2) + f_r(\overline{B+2R}^2 - B^2)$$
$$F_{m_2} \equiv f_w(B^2) + f_r(\overline{B+2R}^2 - B^2)$$

After some algebra, we finally obtain the final solution as a cubic equation. For CPU-based parallelism, this is expressed as F_{CPU}, and for GPU-based parallelism F_{GPU}.

$$F_{CPU} = a[\tfrac{4}{3}R^3 + (2B+2)R^2 + (B^2+2B+\tfrac{2}{3})R + B^2] + b[8BR] + c$$

$$F_{GPU} = a[\tfrac{4}{3}R^3 + (2B+2)R^2 + (B^2+2B+\tfrac{2}{3})R + B^2] + b[\overline{B+2R}^2] + c$$

The remarkable part of this equation is the cubic nature of dependence of the runtime on *R*, which indicates two traits. The first is that data-parallel execution on all platforms shall experience a decrease in overall execution time as *R* increases to some finite integer. The second is that there will also exist an *R* value at which the platform no longer favors computation over communication; in other words, there will be a fixed *R* for a given *B* for which optimal performance is achieved. Later, in the performance study, we in fact observe the fall and rise of runtime with *R*, as predicted by the analytical model. These inferences are in fact in line with the observation and empirical findings in stencil-based computations as well, although our model is more general in nature.

2.4 Latency Parametric Range

Let us define a platform *level* as a computation and communication interface in a parallel computing hierarchy. Two examples of such a hierarchy are shown in Figure 5 and Figure 6. The value range of *R* for a given block size is constrained by *B* at any given level. The restriction is that, at any level, the range of *R* is limited such that a given block cannot encroach upon a neighboring block's execution. This gives $1 \le R_i \le B_i/2$ for any given level *i*. Furthermore, at level *i*, it must be enforced for correct execution that level *i+1* not update more iterations than its parent level's: $R_{i+1} \le R_i$.

2.5 Mobility and Neighborhood Reach > 1

Mobility of agents in the grid is modeled by copying the state of the agent from the source grid cell to the destination grid cell to which the agent moves. The selection of the destination is usually based on vacancy determination procedures combined with some randomization. The reach within the neighborhood of an agent is the extents to/by which the influence of the agents actions extends. Both the neighborhood reach as well as the mobility aspects share the notion of bounding box of influence of some *D* cells around a given grid cell. Both of these aspects are easily accommodated by the *B*+2*R* scheme with one constraint on *R*, namely, R must be a integral multiple of the neighborhood reach or mobility extent *D*. This constraint accommodates all local state update functions as well as remote movement functions.

Thus, the scheme is generalized, and can support "stencils" that reach more than one cell in any direction, and it is not limited to a neighborhood of one cell away.

3. ALGORITHMIC IMPLEMENTATION FRAMEWORK

We have implemented this scheme in two hierarchical parallel processing platforms that are the most commonly available. The first is the multi-GPU platform and the second is the multi-core platform. The null hypothesis is that communication and/or synchronization costs are high on these platforms, and that these costs can be hidden by using our latency hiding scheme. We will first discuss the implementation frameworks for both platforms, followed by performance studies on both.

3.1 Implementation on Hierarchical Multi-GPU, Multi-core Platforms

At the lowest level in a multi-CPU multi-GPU configuration are GPU threads (e.g., NVIDIA's CUDA threads). Even at this granularity, the *B+2R* scheme can be implemented at multiple levels: the CUDA block level and the CUDA thread level. A given thread can operate over a block of data in shared memory. Assigning threads to a 2-D space of *B+2R* allows for *R* correct iterations before communication between threads. At the block level, we once again overlap computation to avoid unnecessary communication between blocks. This follows for multi-GPU configurations whereby synchronization occurs via successive kernel calls. Once again, a large domain can be split up across GPUs for computation while employing this latency tolerant scheme across networked nodes. Further up the tree the methodology is still useful.

At the level of each core in a multi-core platform, we can exercise speed improvements by latency-hiding. This logically continues to the socket level such that a given node can employ these techniques at every computational level providing latency tolerance over the entire data domain.

These hierarchical configurations illustrated in Figure 5 and Figure 6.

Prior to implementing the scheme on GPUs on multiple nodes, we first investigate the implementation of the scheme on a single GPU. Once we establish a sound framework by which GPU exhibits this latency hiding scheme, we use MPI for inter-node GPU communication. This will be discussed in the performance study.

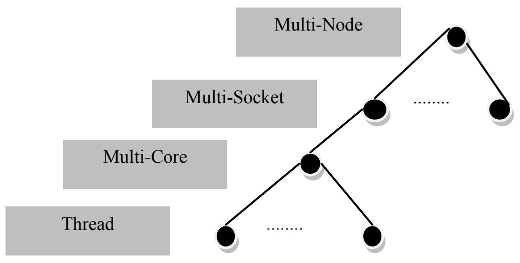


Figure 5: Multi-node, multi-core hierarchy

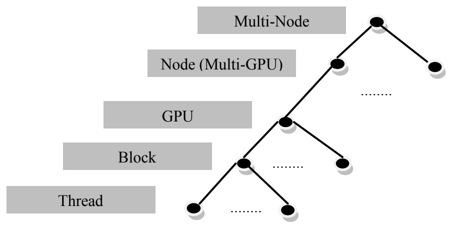


Figure 6: Multi-node, multi-GPU hierarchy

3.2 Latency Hiding per GPU

3.2.1 Block Level

The simplest implementation on the GPU is remaining one level higher from the deepest in the hierarchy, the CUDA block level. In this configuration, a physical dataset is split into $b \times b$ blocks containing an equal number of threads. Then, each agent is mapped to a single thread. This affords a simple one-to-one mapping at the thread level within the CUDA kernel. Maximum concurrency is therefore determined by physical block size.

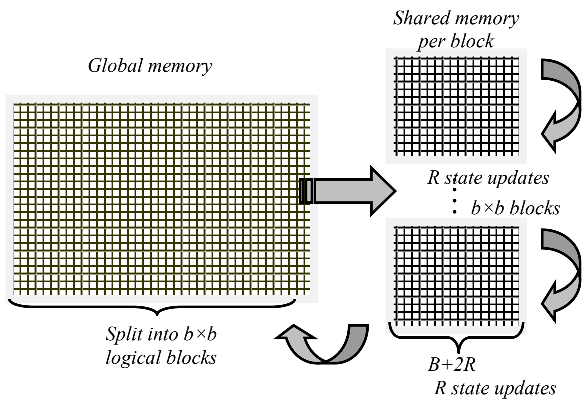


Figure 7: GPU block-level latency-hiding workflow

Simulation data provided either by a parent level or initialized on the GPU is linearly stored in CUDA global memory. Subsequently, the computation kernel is invoked *global iterations/R* times. Within this kernel, blocks are allotted $(B+2R) \times (B+2R)$ shared memory for computation. Following a read from global memory and successive thread synchronization, each thread updates its assigned agent and synchronizes with other threads R times before a write back of size $B \times B$ to global memory ensuring correct execution. This organization is depicted in Figure 7

3.2.2 Block to Thread Level

The second way in which we implement latency hiding is by letting the block level be a logical intermediary for latency hiding at the thread level. Given the workflow in Figure 7, we append latency hiding on the thread level by further dividing block shared memory into logically smaller thread blocks. B_b is the block size mapped to CUDA blocks and R_b is the corresponding padding layer width for each such block. In this scenario, threads contrastingly operate over multiple data. Kernel invocations still serve as block level synchronization; however, individual threads access a third physical data structure for inter-thread communication.

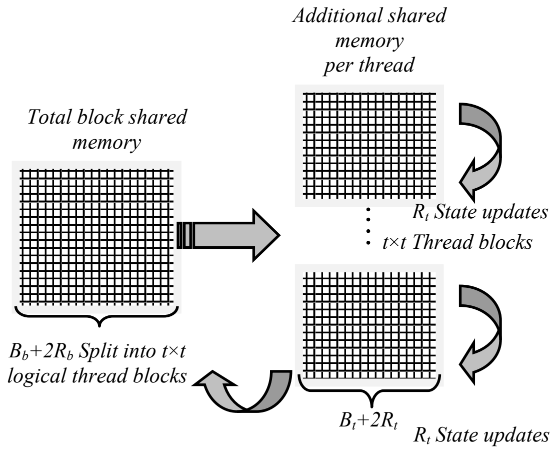

Additional shared memory per thread
Total block shared memory
R_t State updates
t×t Thread blocks
B_b+2R_b Split into t×t logical thread blocks
B_t+2R_t
R_t State updates

Figure 8: Thread-level latency-hiding workflow

Let R_t be the value of R used at the thread level. Within the CUDA kernel, an initial copy from global memory into block shared memory is succeeded by a copy from block shared memory to thread shared memory. Note that if $R_t > 1$, it is required that the total amount of thread shared memory per CUDA block exceeds the amount of block shared memory. As a result, we encounter current GPU hardware limitations preventing full hierarchical latency hiding. Importantly, we have exercised the ability to implement the latency hiding scheme even at the lowest hierarchical level. Figure 8 illustrates this extension.

3.3 Multi-Core Multi-GPU Implementation with MPI

On top of the implementation for latency hiding on a single GPU, we build the remaining framework around this for both multi-GPU and, also substitute the CUDA thread implementation with a pthreads-based CPU thread implementation for multi-core configurations. Both the multi-GPU and multi-core frameworks utilize MPI for inter-node communication.

On a cluster of nodes, a socket allocates memory according to its subsection of the whole computation domain. For instance, if the whole computation size is 1024×1024 agents and 4 nodes of a cluster are utilized, each node would accordingly allocate memory for 256×256 agents. From here, we either execute the simulation via a single GPU by passing this memory to GPU global memory, or we execute on the CPU by using POSIX pthreads. After R_{CPU} or R_{GPU} iterations, we post non-blocking MPI_Irecvs and MPI_Isends to tasks according to their logical 2D rank. Each node then repopulates its $((B+2R)^2-B^2)$ data, and execution continues.

3.4 CUDA Restrictions and Memory Considerations

The current state of GPU hardware and software configurations imposes some restrictions on the flexibility of our scheme at both the block and thread levels The two most important factors limiting maximum concurrency not present in recent CPU cluster configurations are the number of threads per block, and the maximum amount of memory that can be allocated on a per block basis. For the 8800 GTX GPU, the maximum number of threads per block can be as large as 512. However, given that we are operating on a logically 2-dimensional grid, this number is reduced to the smallest square less than 512, 484. Thus, we can operate with at most 22×22 threads per block. For our block level implementation, this was a limiting factor unexpressed in CPU implementations.

Implementation of the scheme down to the thread level, however, presented the largest barrier. It is known that operations in the shared memory space of a GPU are much faster than operations on the global address space. For maximal computational speed, therefore, we attempt to perform most computation on shared memory. Referring back to Figure 8 and prior discussion, we observe that extra shared memory is required for thread level implementation. Here, we now qualify this restriction.

For a physical data grid split into blocks of size B_b with block level reach R_b, each CUDA block requires $(B_b+2R_b)^2$ bytes of shared memory. In addition, for thread blocks of size B_t with thread level reach R_t and t^2 threads per block, we require $t^2(B_t+2R_t)^2$ bytes of shared memory. Finally, typically employed methods for intra-block computation require separate read and write memory spaces. In other words, it is common to "ping-pong" computation between two memory spaces. If we employed this configuration (requiring another $t^2(B_t+2R_t)^2$ bytes of shared memory) we would have minimal concurrency (number of threads per block less than or equal to 4) and a maximum thread reach, R_t, of 2. This would not be sufficient enough to investigate latency hiding at the thread level. We therefore implemented a method by which typical ping-pong fashion is not required, discussed in the next section. After this optimization, the number of threads could be increased up to 16 threads per block, giving a maximum R_t of 4.

3.5 Minimizing Memory Requirements

In shared memory units, a larger value of R results in increased concurrency, since the communication cost is negligible within the shared memory unit. However, increasing R also increases the amount of additional memory used for latency hiding. Thus, it is important to find ways to minimize the memory usage while still increasing R.

Such as problem arises in a CUDA-based implementation, in which the shared memory size is limited, and hence must be carefully organized for the threads to perform their concurrent computation. Traditional update schemes employ a read buffer and a write buffer for evolving an N×N grid, requiring $2N^2$ memory variables. Instead, if an in place update scheme exists, it can be used to avoid another copy of the entire grid. We developed such an in-place update scheme, as shown in Figure 9, and used it to reduce the memory needs, and consequently increase the concurrency afforded by the latency hiding scheme, which reduces the temporary storage from N^2 down to N+1.

Given a 2D grid of cells for parallel update, we can use $(N+1)\times V$ registers for complete state update instead of using $N\times N$ extra registers. This method is depicted in Figure 9 where white cells are "to be updated" and yellow are already updated. These updates occur in linear fashion and we store data as needed in additional registers.

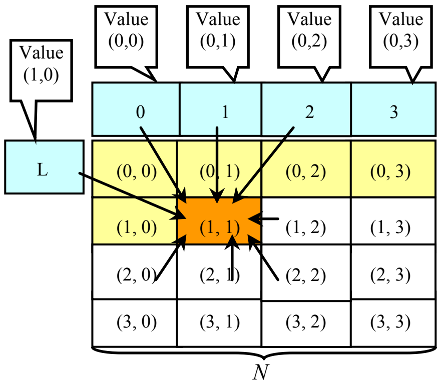


Figure 9: Memory minimization for thread-level computation. Blue cells are temporary registers, yellow are already updated, white are to be updated, and orange is currently being updated

4. EXPERIMENTATION PLATFORM

4.1 Hardware

GPU and CPU experiments have been run on the National Center for Computational Science (NCCS) LENS cluster. The platform was suitable because, as a data analysis cluster, not only were we able to access multi-node and multi-core functionality employed on many clusters, but it also afforded the use of recent NVIDIA 8800 GTX (2 per node) GPUs. Each offers 768MB of onboard memory, 128 stream processors, and a core clock speed of 575MHz. In regard to CPU experiments, each node contains four quad-core 2.3 GHz AMD Opteron processors with 64 GB of memory.

4.2 Software

For our single and multi-GPU runs, we use the NVIDIA Toolkit and SDK (`nvcc` compiler). Concurrent execution on the multi-CPU level is obtained through POSIX pthreads (16 per node with 16 cores per node). Finally, inter-node communication is handled through MPI (Open MPI specifically). All runs were conducted on a 64-bit Linux cluster.

5. APPLICATIONS AND SCENARIOS

With the goal to reduce communication latency at the cost of increased computation, we choose scenarios that are fine- to medium-grained in computation. The first benchmark is a relatively well known model, namely, John Conway's Game of Life. The second is a recent, more complex model, called Leadership. The details of both models are discussed next.

5.1 Game of Life

The Game of Life (GOL) is a scenario in which a 2-dimensional spatial grid of cells is initially marked dead or alive. At each simulation time step, cells gather information from surrounding neighbors and make a Boolean choice. Cells that are occupied and surrounded by two or three neighbors remain occupied, otherwise, remove themselves from the grid. Unpopulated cells with exactly three neighbors become occupied.

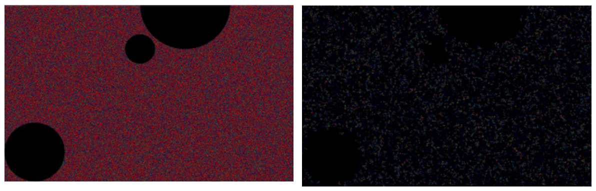


Figure 10. Snapshots of grid for Game of Life: empty cells are in black; live cells are blue; recently-dead cells are red; green just became alive.

5.2 Leadership

The Leadership (LDR) model (developed as part of a US DARPA project) in , is a computationally involved model in which each agent computes an objective function for every iteration. In the model, reproduced below, each agent maximizes its utility in order to decide on the best behavior to adopt at any moment in the simulation:

$$Order = O \in \{-1,0,1\}$$
$$Behavior = B \in \{-1,0,1\}$$
$$Propensity = P \in \{-1,0,1\}$$
$$Loyalty = L = \lambda \frac{|O-B|}{2}$$
$$Lambda = \lambda = \lambda_{previous}(1-\delta) + M_l\delta$$
$$Coercion = C = R\frac{|O-B|}{2}$$
$$Idealogy = I = \frac{|P-B|}{2}$$
$$Utility = U = 1 - w_l L^2 - w_c C^2 - w_i I^2$$

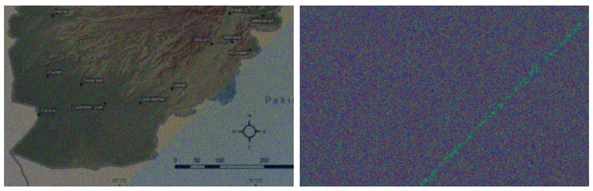
Given an order O, of interest is the variation of behavior B that is chosen by each individual to maximize the individual's utility U. Lambda's time dependence induces variation of B over time. When M_l is defined as the mean loyalty of neighbors, the variation of B is less interesting, as lambda follows some sort of a diffusion process which can be expected to converge to an overall average across all individuals. To accommodate some dynamics, we make one change, namely, M_l is defined as the maximum loyalty, instead of mean loyalty, among neighbors. The rationale behind this variation is that the neighbor with the largest loyalty, even if there is only one, potentially has an overbearing influence on all its neighbors. Our implementation is initialized with constants: O=1, R=0.25, W_l=0.33, W_c=0.33, W_i=0.34, and δ=0.01. P is uniformly randomized across the population.

6. PERFORMANCE STUDY

In our performance study, we use "improvement level" as the metric to observe the decrease in run time that our latency-hiding provides over traditional technique with no latency-hiding. This is given by the following equation.

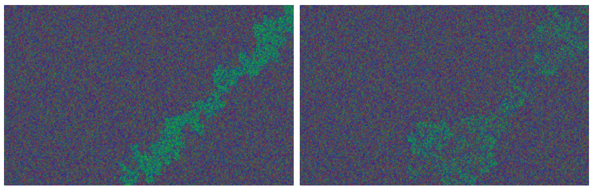
$$Improvement\,Level(\%) = \frac{Run\,Time_{no\ latency\ hiding}}{Run\,Time_{latency\ hiding}} \times 100$$

Note that an improvement level L=100% implies that the run times with and without latency hiding are the same, and any level L>100% implies a reduction in run time of latency hiding over that of no latency hiding by a factor of L/100.



(a) Initial behavior map divided along a country border; loyal behaviors are below the diagonal (blue)

(b) Behavior smoothens after a few time steps, but neutral behaviors emerge along diagonal



(c) Fluctuations and growth of neutral behavior outward from the diagonal is observed

(d) Neutrality waves are regenerated despite intermediate ebbs

Figure 11: A simulation of the leadership model: blue shows loyalty to leadership, green shows neutrality and blue shows anti-order stance. Sustained waves to/away from neutrality indicate prolonged "unrest"

6.1 Single GPU

To initially conduct our performance study, we benchmarked the GOL scenario on a single GPU on a single node of the LENS cluster. For both block- and thread-level schemes, we ran a range of agent populations, with multiple R values, and varying number of threads per block, T. Observed phenomena are generally static for varying populations, i.e. the only observed performance difference as population increases is expected and observed runtime increase. All benchmarks presented in this subsection are for approximately one million agents, and simulated for 256 global iterations. This equates to 256/R kernel invocations for varying R.

6.1.1 Thread Level Latency Hiding

We started by investigating latency hiding at the deepest hierarchical level, the CUDA thread level, followed by empirically uncovering both the nature of the latency hiding scheme at this level and restrictions inherent to the GPU CUDA architecture. At the thread level, for a fixed number of threads (T^2 threads in operation for a given T), in line with our hypothesis, we observe a decrease in runtime as R increases from 1 (essentially no latency hiding) to 2. We also see communication cost decrease with increasing R as expected. As R increases beyond 2, however, execution time increases and levels off. This indicates multiple phenomena. For the GPU architecture, at *R=2*, we quickly reach the point at which trading communication for computation affords speedup. Also, we would expect that as R is increased, computation cost would eventually overtake the reduced communication cost, resulting in an increase in runtime with no upper bound. Given our restrictions on shared memory and the ability of the GPU to quickly perform arithmetic, we do not observe this expected increase. At *R=2* with 4 threads per block *(2x2)* in operation, we observe the most efficient execution. Finally, after measuring synchronization cost amongst GPU block threads, we find it to be negligible in comparison to either memory reads/writes or computation. Thus, the additional overhead incurred by implementing thread-level latency hiding does not afford additional speedup.

6.1.2 Block Level Latency Hiding

At the block level, we see communication cost decrease and overall runtime decrease up to a given *R* (see Figure 12). We also notice, importantly, the stark contrast between overall runtime when comparing block and thread level latency hiding. As previously stated, the increased overhead incurred by implementation of the scheme at the thread level hinders performance. This manifests itself as an order of magnitude difference. We therefore conclude that thread level implementation is not useful for optimal speedup, and for our subsequent benchmarks on multiple nodes of the LENS cluster, we implement the scheme only at the block level.

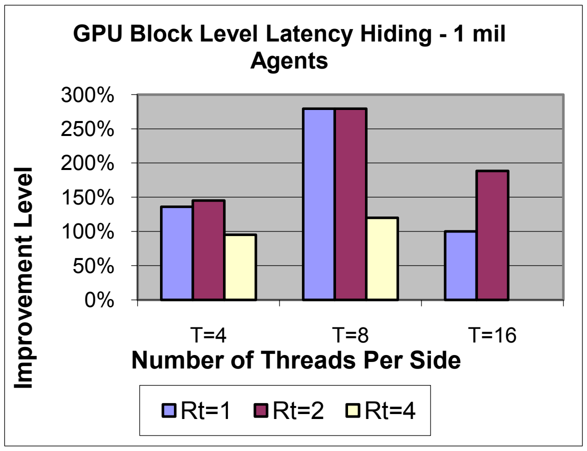


Figure 12: Improvement of GPU block level latency-hiding compared to traditional (no latency-hiding)

With this initial single GPU study, we turn to our benchmarks of both the GOL and LDR models on a larger, multi-GPU and multi-CPU scale (16 nodes of the LENS cluster). The first set of these benchmarks, discussed next, uses a single GPU on each of the 16 nodes.

6.2 Multi-Node, One-GPU per Node

We ported the single GPU latency hiding scheme to one hierarchically higher level on the LENS cluster of GPUs. Within this new framework, we effectively increase maximum number of agents simulated on a single GPU (approximately 16 million in our studies) multiplied by 16 (nodes).

Both the LDR and GOL scenarios were benchmarked. Once again, because of shared memory limitations, we here present data for each GPU simulating approximately 1 million agents. These benchmarks are represented in Figure 13 and Figure 14 respectively.

The speedup bars clearly highlight the dramatic gains afforded by the latency hiding scheme when multiple GPUs are used across MPI. It is evident here that communication latency hiding represented by R_m (for the parameter *R* at the MPI/node-level) is the dominant factor in speedup for both scenarios. As expected, inter-node communication is much more expensive with respect to wall time. Also we observe strong performance benefits when applying this scheme across nodes (up to two orders of magnitude on GOL).

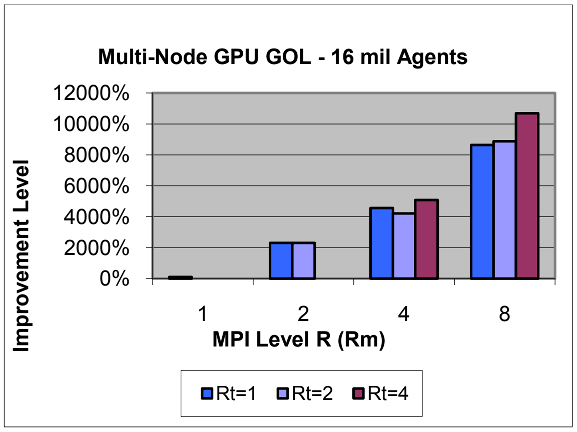


Figure 13: Improvement of latency-hiding compared to no latency-hiding for GOL simulated on 16 GPUs of the LENS cluster

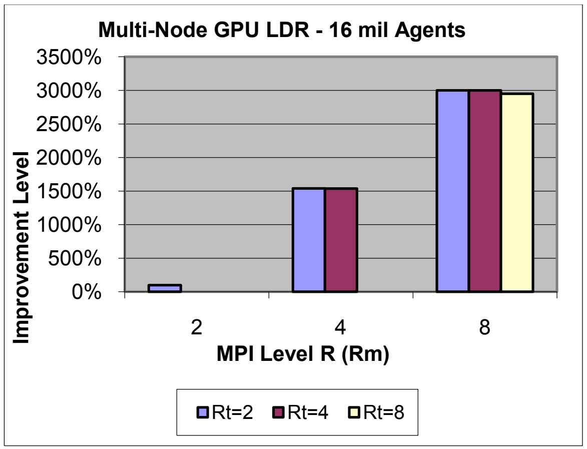


Figure 14: Improvement of latency-hiding compared to no latency-hiding for LDR simulated on 16 GPUs of the LENS cluster

With this relatively small dataset, however, we do not observe the point at which computation catches up with communication. A key hypothesis is that there will be a fixed point at which this tradeoff becomes unfavorable to runtime, i.e., we no longer benefit from the scheme. This will be discussed in the next section.

6.3 Multi-Node Multi-Core

Not constrained by the memory configurations of the 8800 GTX GPU, on multi-core platform, we are able to scale simulation size to over 10^9 agents. In similar fashion to GPU benchmarks, we test both the LDR and GOL models. These are represented in Figure 15 and Figure 16. Most notable in this configuration is large scalability of our scheme. Similar to multi-GPU execution, latency hiding at the MPI level is most dominant factor in model speedup. In Figure 17 we observe the point at which the communication vs. computation continuum no longer affords increased speedup. For *R>256* at the MPI task level, computation costs finally offset communication costs. This was predicted via the analytical model, and is now quantified.

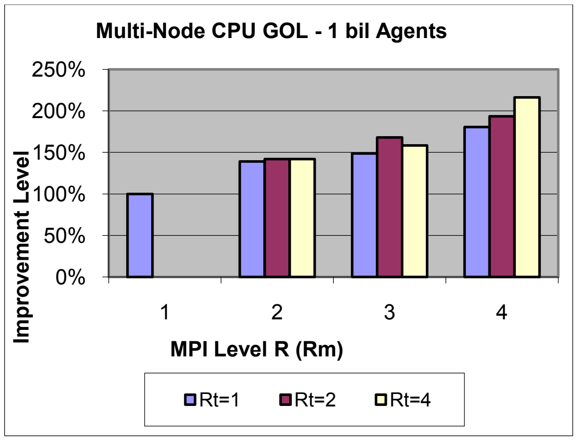


Figure 15: Improvement of latency-hiding compared to no latency-hiding (R_t=1) when GOL is simulated on multi-core multi-node platform

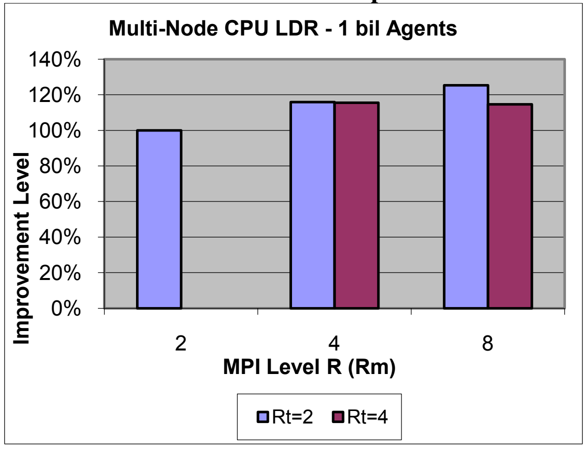


Figure 16: Improvement of latency-hiding compared to no latency-hiding (R_t=2) when LDR is simulated on multi-core multi-node platform

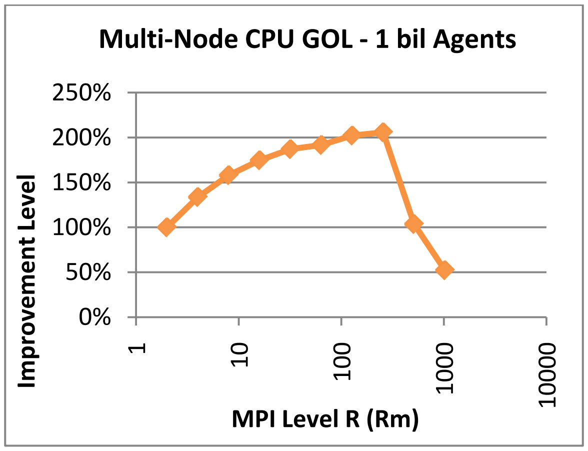


Figure 17: Improvement of latency-hiding compared to no latency-hiding; shows reduction in runtime with increasing *R*, reducing by more than half in the best case before extra computation costs more than gains from decreased communication

7. SUMMARY AND FUTURE WORK

We presented a way to scale ABMS on extant multi-CPU and multi-GPU systems while retaining both model fidelity and high execution speed of fine to medium granularity models. We are able to scale these simulations to over one billion agents, to aid in exploration of emergent phenomena in certain agent models. We have also presented a flexible way to exploit emerging computing resources. Here, we are able to utilize up to 256 CPU-cores and/or 16 GPUs concurrently.

With our preliminary implementation and performance study, on multi-CPU architectures, we have shown large decreases in runtime by trading communication for computation. This same method, when applied to multi-GPU systems, allows for speed increases of over two orders of magnitude. This is all achieved through a single, unified, parameterized model, applicable on multiple architectures. Importantly; the method can be used on many hierarchical computational levels and their combinations, from CUDA threads to inter-node communication.

It is important to note that the B+2R speedups are relative to no latency hiding scheme, and that the no latency hiding schemes are already highly optimized for a single (non-networked) GPU. Our earlier work [11] on a single GPU already demonstrated three orders of magnitude faster agent simulations on a single GPU. Thus, actual (absolute) speedups of our current multi-GPU work when compared to a CPU-based implementation are over 30×, and speedups compared to existing CPU-based systems in Java are over 1000×. In the best case (Figure 13), the speedup of our scheme over the existing straightforward scheme is over 150×, which represents over four orders of magnitude improvement over existing Java-based agent simulations.

This leaves many areas for future work. One of the limitations in our implementation was coding separately for each platform. Ideally, it would be useful to have a seamless interface for all platform levels. OpenCL is one current technology designed to do this, our future experiments in latency hiding may investigate this technology. Another desirable feature is to dynamically and automatically tune *R* for each level. This would decrease trial and error methods for finding the most suitable *R*. Also, with newer GPU configurations, levels of performance and scale could be expected to increase. Finally, incorporation of additional ABM-specific features such as agent mobility and large agent neighborhoods will be investigated.

ACKNOWLEDGEMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] U. Wilensky, "NetLogo," ed. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, 1999.

[2] J. Epstein, "Modeling Civil Violence: An Agent-based Computational Approach," *PNAS,* vol. 99, pp. 7243-7250, 2002/05/14 2002.

[3] M. J. North*, et al.*, "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit," *ACM Transactions on Modeling and Computer Simulation,* vol. 16, pp. 1-25, 2006/01/01 2006.

[4] M. J. North and C. M. Macal, *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*: Oxford University Press, 2007.

[5] DARPA. (2009, *US Defense Advanced Research Projects Agency - Technologies for the Applications of Social Computing (TASC)* [Web]. Available: http://www.darpa.mil/ipto/solicit/baa/RFI-SN-09-20_PIP.pdf

[6] S. Luke, *et al.*, "MASON: A New Multi-Agent Simulation Toolkit," in *SwarmFest Workshop*, 2004.

[7] B. Walter*, et al.*, "UAV Swarm Control: Calculating Digital Phermone Fields with the GPU," in *Interservice/Industry Training, Simulation and Education Conference (IITSEC)*, Orlando, FL, 2005.

[8] P. Riley, "SPADES: A System for Parallel-Agent, Discrete-Event Simulation," *AI Magazine,* vol. 24, 2009.

[9] A. M. Uhrmacher and K. Gugler, "Distributed, parallel simulation of multiple, deliberative agents," Proceedings of the fourteenth workshop on Parallel and distributed simulation, Bologna, Italy, 2000.

[10] M. Lees*, et al.*, "Distributed simulation of agent-based systems with HLA," *ACM Trans. Model. Comput. Simul.,* vol. 17, p. 11, 2007.

[11] K. S. Perumalla and B. Aaby, "Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs," Agent-Directed Simulation Symposium, 2008.

[12] R. D'Souza*, et al.*, "SugarScape on Steroids: Simulating Over a Million Agents at Interactive Rates," AGENT 2007 Conference on Complex Interaction and Social Emergence, Evanston, IL, 2007.

[13] A. Chaturvedi*, et al.*, "Bridging Kinetic and Non-kinetic Interactions over Time and Space Continua," in *Interservice/Industry Training, Simulation and Education Conference*, Orlando, FL, USA, 2005.

[14] M. Hybinette*, et al.*, "A Design for a Scalable Agent-based Simulation System using a Distributed Discrete Event Infrastructure," Winter Simulation Conference, 2006.

[15] J. Parker, "A Flexible, Large-scale, Distributed Agent-based Epidemic Model," Winter Simulation Conference, Piscataway, NJ, 2007.

[16] R. C. Armstrong*, et al.*, "Parallel Computing in Enterprise Modeling," Sandia National Laboratory, Techincal Report SAND2008-6172, 2008/08/01 2008.

[17] M. Dawit, "Performance Optimization for Multi-agent Based Simulation in Grid Environments," IEEE International Symposium on Cluster Computing and the Grid, 2008.

[18] M. Ripeanu*, et al.*, "Cactus Application: Performance Predictions in Grid Environments," ed, 2001, pp. 807-816.

[19] C. Ding and Y. He, "A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems," Supercomputing, 2001.

[20] S. Krishnamoorthy*, et al.*, "Effective Automatic Parallelization of Stencil Computations," Programming Languages Design and Implementation (PLDI), San Diego, California, USA, 2007.

[21] K. Datta*, et al.*, "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures," Supercomputing, Austin, Texas, 2008.

[22] H. Dursun*, et al.*, "A Multilevel Parallelization Framework for High-Order Stencil Computations," in *Lecture Notes in Computer Science*. vol. 5704/2009, ed: Springer Berlin / Heidelberg, 2009, pp. 642-653.

[23] J. Meng and K. Skadron, "Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs," 23rd international Conference on Supercomputing, Yorktown Heights, NY, USA, 2009.

[24] Z. C. Rojas and M. Hoemmen. (2004, *Communication Savings with Ghost Cell Expansion for Domain Decompositions of Finite Difference Grids* [Project Report]. Available: http://www.cs.berkeley.edu/~ejr/GSI/cs267-s04/final-projects/mhoemmen-rojas/report.pdf

[25] P. Brecke*, et al.*, "Actionable Capability for Social and Economic Systems (ACSES)," Seedling Project - Defense Advanced Research Projects Agency, Project Report2008/05/01 2008.