In Proceedings of the Spring Simulation Multi-Conference, Ottawa, Canada, April 2008

Best Paper Award Winner

Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs

Kalyan S. Perumalla
Brandon G. Aaby
perumallaks@ornl.gov, aabybg@ornl.gov
Oak Ridge National Laboratory

ABSTRACT

Programmable graphics processing units (GPUs) have emerged as excellent computational platforms for certain general-purpose applications. The data parallel execution capabilities of GPUs specifically point to the potential for effective use in simulations of agent-based models (ABM). In this paper, the computational efficiency of ABM simulation on GPUs is evaluated on representative ABM benchmarks. The runtime speed of GPU-based models is compared to that of traditional CPU-based implementation, and also to that of equivalent models in traditional ABM toolkits (Repast and NetLogo). As expected, it is observed that, GPU-based ABM execution affords excellent speedup on simple models, with better speedup on models exhibiting good locality and fair amount of computation per memory element. Execution is two to three orders of magnitude faster with a GPU than with leading ABM toolkits, but at the cost of decrease in modularity, ease of programmability and reusability. At a more fundamental level, however, the data parallel paradigm is found to be somewhat at odds with traditional model-specification approaches for ABM. Effective use of data parallel execution, in general, seems to require resolution of modeling and execution challenges. Some of the challenges are identified and related solution approaches are described.

KEYWORDS

Agent-based Simulation, Parallel Simulation, Emergent Behavior, Graphical Processing Units, Large-scale Simulation

1 INTRODUCTION

1.1 Motivation and Background

Agent-based modeling (ABM) has established itself as one of the leading modeling approaches [1]. Several applications have been shown to benefit from ABM to understand, explain, predict and analyze complex processes occurring in many contexts. While some of the agent behaviors are interesting even in small- to medium-scale (a few hundreds to a few thousands of agents), there is a general desire expressed often by some social scientists to be able to experiment with very large counts of agents. Interesting emergent phenomena, for example, are expected to occur when the counts are to the tune of one million or so. There is also the relatively unchartered territory of experimentation with even larger scenarios with tens of millions of agents, either homogeneous or heterogeneous in their individual behaviors, as at the level of population counts of entire countries. Such a large scale presents many challenges, not the least of which is the modeling complexity itself: developing plausible models that can be reasonably expected to deliver useful results. Besides the great modeling challenge, the other significant hurdle to scaling ABM simulations is the wall-clock time needed to execute a simulation experiment from start to completion. Clearly, execution time should be as small as possible, and, ideally, low enough to enable a large number of simulation runs to perform a coherent multi-run experiment (e.g., by employing parameter sweeps).

Several ABM simulation tools are available today, each better than others in one or more critical aspects, such as ease of programming, visualization support, integrated development capability, rapid prototyping and so on. However, one of the common limiting aspects is their execution speed. Most are limited to sequential execution, and few tools are currently demonstrated to scale to more than the order of 10^4 agents at reasonably low runtimes. Either memory limitations arise, or the runtimes become prohibitively high to be practically useful. Such limitations have not been significant in many useful applications in which the outcomes of interest are satisfied by smaller agent counts. However, it is the authors' understanding that ABM research is reaching a point at which there is interest in pushing the envelope with respect to scale and speed.

Interestingly, recent developments on the computational front point to the possibility of meeting the goals of increased scale and speed. Some of the important platforms relevant to large-scale, high-speed ABM execution include newer shared-memory platforms such as general-purpose programmable graphical processing units (GPUs) [2-5] and multi-core versions of traditional central processing units (CPUs) (e.g., Intel's [6]). These platforms provide on the order of tens to hundreds of processing units packaged within a small form-factor package for a very reasonable market price. For an even larger scale, distributed-memory platforms are relevant, such as many-core installations and supercomputers[7]. In this document, the focus is on the shared-memory platforms, especially on exploiting the data-parallel execution capabilities exhibited by GPUs and multi-core CPUs.

1.2 Related Work

Data parallel execution facilities have seen some recent use in applications that closely resemble ABM simulations. Crowd simulations, with up to 15 thousand "agents" were simulated and visualized at interactive speeds on an older generation of NVIDIA graphical processing units (GeForce 5900) [8, 9]. The authors report a 10- to 30-fold speedup achieved by GPU-based execution, as compared to CPU-based execution. Some early work simulated a probabilistic evolution of a large number of physical system entities, with Monte Carlo-based Ising Spin model simulations [10]. Control of a group of airborne vehicles using "Swarm" type of control [11] was performed on GPUs to meet real-time constraints. In the OneSAF distributed entity simulation framework, the GPU platform was used to improve execution speed, but in a different way [12]. As opposed to speeding up the individual computation of entity (agent) evolution, the GPUs were instead used for speeding up the computationally-intensive collision detection functionality and other global operations.

While all these efforts represent ways of using the parallel execution platforms in agent-like environments, the more generalized ABM-style execution paradigm remains an open area of research. Challenges exist in effectively dealing with the wide variety of aspects [1, 13] in ABM, when viewed in the context of parallel execution. This paper serves to document our insights into such generalized ABM execution on data parallel platforms. In order to gain an understanding of the performance differential afforded by the data parallel platforms, we first undertook an implementation and benchmarking exercise of a few well-known ABM experiments. Based on our experiments, we are able to quantitatively document the tremendous gains in runtime performance that are possible with certain models. Additionally, the exercise helped reveal the relation of typical ABM primitives to their impact on parallelism, exposing some of the fundamental issues underlying data parallel execution of general ABM execution.

1.3 Outline

The rest of the document is organized as follows. In Section 2, the benchmark models are described along with their implementation. A performance study using the benchmarks is presented in Section 3, comparing the GPU-based runtime speed with the speed of optimized versions of equivalent CPU-based models. The GPU-based runtime is also compared with equivalent models implemented using ABM toolkits. In Section 0, the challenges with data-parallel execution, in general, of ABM simulations are outlined, and some solution approaches are presented. Results are summarized and future work is outlined in the final Section 5.

2 BENCHMARKS

2.1 Models

The runtime performance of GPU-based ABM execution is evaluated using three different models: (1) Mood Diffusion (2) Game of Life (3) Segregation. These models are described next.

Mood Diffusion: The first is a model of diffusion of mood among interacting people[14, 15]. The "mood" of each agent *i* is modeled with a scalar mood value m_i. The rate of change of mood m_i of agent *i* is dependent on a homeostatic tendency of the agent to a value M^h_i, combined with an influence by the average mood M^a_i of its neighbors, and by the influence of regional/global news events with intensity M^g_r in the neighborhood *r* that includes agent *i*. The equation governing the evolution is given by

$$\frac{dm_i}{dt} = \alpha(M_i^h - m_i) + \beta(M_i^a - m_i) + \gamma(M_r^g - m_i) .$$

The mood values of agents are initialized with random values from the range [0..1]. In our experiments, for simplicity, the homeostatic mood value is set to be the same among all agents. At irregular intervals, mood-altering events are inserted into the grid at random locations with varying reach of influence per event.

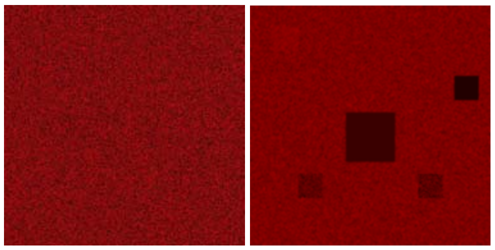

Figure 1. Snapshots of a 512×512 Grid for Mood Diffusion Model. Red Saturation Represents Mood Level

Game of Life: The second model is the well-known Conway's Game of Life [16], with a 2-dimensional spatial grid of cells marked dead or alive. In our experiments, the grid is initialized with a random live cell pattern (a cell is initialized to be alive or dead with equal probability). During evolution, a dead cell with exactly three neighboring cells comes alive, while a live cell dies if fewer than two or more than three of its neighbors are alive.

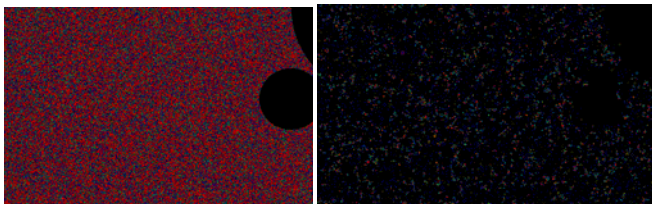

Figure 2. Snapshots of Grid for Game of Life. Black Denotes Empty Cells, Blue Live Cells. Red Cells are those that Died Recently. Green Just Became Alive.

Schelling Segregation: The third model is one of the variants of the Schelling Segregation Model framework [17]. Red and green colored agents are scattered across a 2-dimensional grid. Agents stay put if they are happy, or attempt to move if unhappy with their current position. In our experiments, an agent with a neighborhood of at least 30% like-colored agents is happy; it's unhappy if not. When unhappy, an agent attempts to move from its current cell to a new random vacant location within a certain region of its vision.

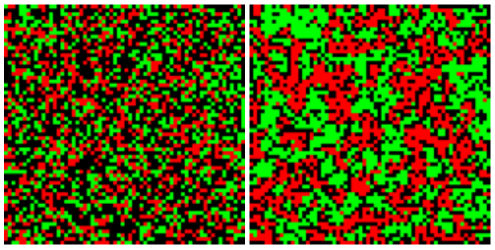

Figure 3. Snapshots of Grid for Segregation of Red and Green Agents. Black Cells are Vacant.

Graphical snapshots from sample executions of the Mood Diffusion, Game of Life and Segregation models are shown in Figure 1, Figure 2, and Figure 3, respectively.

2.2 Implementation

For each model, two distinct implementations were developed: one that executes on a GPU and another equivalent one that executes on a traditional CPU. The CPU-based implementations are carefully developed for optimized execution on the CPU, and hence represent the closest to the best execution possible with a CPU. To ensure equivalence, initialization routines were modified to assign the same initial conditions for agents in GPU, CPU and Repast[18]/NetLogo[19] models.

For the Game of Life model, one additional CPU-based implementation was used in benchmarking. This is the Game of Life model included in the Repast ABM system [18], executed in the maximal performance mode (command-line mode, with graphical output turned off). The Repast .Net (C#) version, as well as Repast J (Java) version, was used for benchmarking. Similarly, for the Segregation model, an additional CPU-based implementation is benchmarked, which is the model included in the NetLogo [19] system (again, executed in maximal performance mode).

The GPU-based implementation was developed using a "ping-pong approach" to updating textures with fragment processors, all realized using the Open GL graphics interface[2, 3]. In the models, it was sufficient to map each agent to one pixel value, and map the 2-D grid to a graphics texture. Agent's state is mapped to red, blue, green or alpha channels of a pixel. The software platform is a combination of Microsoft Visual Studio .Net, and the NVIDIA Cg Took Kit [20]. A large fraction of the agent functionality is realized as Cg programs that get invoked for every agent. The Cg programs (also called kernels) manipulate the agent's state encoded in a pixel. Data parallelism is achieved when the GPU streams all texture pixels through multiple (fragment) processors, and each processor invoking the Cg program for every single pixel that is streamed and transformed through it.

We have also previously benchmarked some of the models with the Brook[4, 21] system, and found the performance to be fairly similar to that of the Open GL implementation. The newer CUDA [22] system for GPU-based execution is expected to deliver similar performance as well, with which we have just started to experiment.

The hardware for GPU-based experiments is a recent NVIDIA GeForce 8800 GT unit, and the CPU is an Intel Core2 Duo 2.4 GHz processor with 4 GB memory. These hardware components represent some of the best mainstream commodity units available at a good performance/price ratio. As of this writing, 3.0 GHz "overclocked" CPUs (Intel) are becoming available; however, the performance of these faster CPUs relative to the GPU is about 15% better on these models.

3 PERFORMANCE STUDY

To evaluate runtime performance gains from executing on the GPU, a series of runs were made with the CPU and GPU implementations. The performance is evaluated with increasing number of agents. The grid size is increased in powers of 2, from 16×16 to 512×512. For the Mood Diffusion and Game of Life models, the grid size is increased all the way up to 2048×2048, and then to 3750×3750, giving a maximum of over 14 million agents. The grid size is limited by the maximum texture size supported by the GPU, which, for our platform, was 3750×3750 pixels. A runtime system error in our GPU-based segregation model limits it to 262,144 agents, which we are currently investigating and expect to fix soon, enabling it to scale to 14 million agents similar to our Mood Diffusion and Game of Life models.

3.1 Runtime Speedup

In the following performance charts, speedup is computed as the ratio T_{CPU}/T_{GPU}, where T_{GPU} is the total execution time on the GPU and T_{CPU} is the total execution time on the CPU, for the same number of iterations (time-steps). Thus, a speedup of 30 implies that the simulation completes on the GPU 30 times faster than on the CPU. Also, in all the figures, the amount of variability is negligible (less than 5%) across multiple repeated runs; hence error bars are omitted for readability.

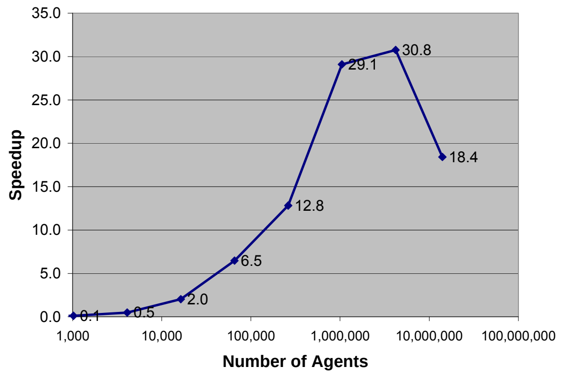

Figure 4. Speedup of GPU compared to CPU for Mood Diffusion Model

For the Mood Diffusion model, the runtime speedup obtained by the GPU-based implementation compared to optimized CPU-based implementation is shown in Figure 4.

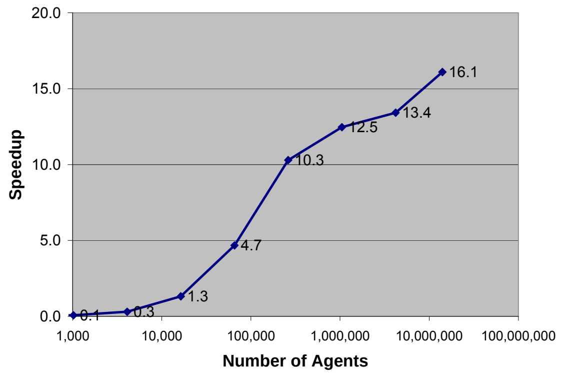

Figure 5. Speedup of GPU compared to CPU for Game of Life Model

The GPU-based model executes significantly faster than the CPU-based model, delivering a peak of over 30-fold reduction in runtime for 4.2 million agents. A drop is seen beyond 4.2 million agents, the cause of which is undetermined as of this writing and is under investigation. Hardware artifact beyond 2048×2048 output texture size is one of the suspects.

The speedup obtained by Game of Life on GPU *vs*. CPU is shown in Figure 5. The execution shows excellent scaling with the number of agents, giving 16-fold speedup for 14 million agents.

3.2 Performance Effects due to Conditional Statements

The lower speedup of Game of Life compared to Mood Diffusion (16× *vs*.30×) deserves some explanation. The data parallel execution mechanism in the GPU suffers from lowered performance when conditional statements are executed by the fragment processors of the GPU. Data parallelism artifacts force all processors to incur the combined cost of the true and false branches of the conditional statement, instead of the cost of the actual branch taken alone. The CPU on the other hand is better able to deal with conditional statements. Branch/value prediction is another aspect in which the CPU and GPU differ; the CPU seems to employ a prediction framework and hardware support, which apparently helps deliver better performance.

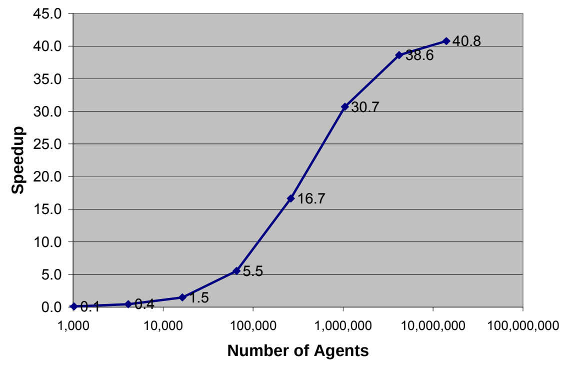

Figure 6. Speedup of GPU compared to CPU for "Unconditional GOL" Test

The fact that the conditionals are the primary contributor to the lower amount of GPU/CPU speedup is verified by the speedup graph shown in Figure 6. The Game of Life model is modified such that no conditional statement is executed (by inserting an "early return" into the model). As expected, the performance of the modified model increased to over 40×. Although the modified model is not very useful as a model itself, it nonetheless serves to verify the performance penalty imposed by the conditional statements.

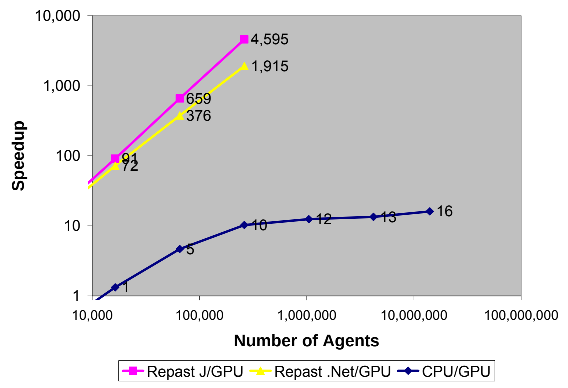

Figure 7. Speedup of GPU compared to Repast for Game of Life Model

In order to gauge the performance differential between the GPU-based model and ABM tool kit-based model, we benchmarked the GPU execution against an equivalent version implemented in the Repast tool kit. The speedup delivered by the GPU-based model is shown in Figure 7. As expected, our GPU-based model ran three to four orders of magnitude faster than the Repast versions. The Repast runtimes for agent populations larger than 512×512 are omitted in the figure as they were very large (extrapolated runtime being in millions of seconds).

3.3 Performance Effects due to Locality

The increase in speedup with increasing agent count is noteworthy. While the GPU exhibits relatively constant runtime cost with increasing texture size (and hence, the number of agents), the ABM tool kits typically suffer from at least linear (and sometime quadratic) increase in runtime cost.

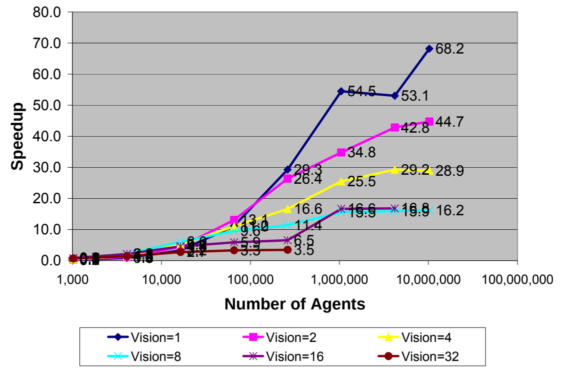

Figure 8. Speedup of GPU model compared to CPU for Segregation Model

The speedup chart for the Schelling Segregation model is shown in Figure 8, giving the ratio of GPU model speed and CPU model speed. Figure 9 shows the speedup of the GPU-based model compared to its equivalent model written in NetLogo. For the GPU-based implementation, a lock-free data-parallel algorithm called Select-Backoff (described later in Section 0) is used to parallelize this model. The Segregation model includes a free parameter, which is the area of vision of each agent. A vision value equal to *v* represents the number of cells in each direction to which the agent is willing to move if the agent is unhappy with its current location. Thus, given a vision *v*, the number of cells that the agent must consider for relocation is up to $(2v+1)^2-1$. For the largest vision value shown (v=32), each unhappy agent potentially considers up to 4223 cells around it.

Clearly, the larger the vision value, the greater is the processing requirement of each agent per iteration. Perhaps more importantly, the amount of locality decreases quadratically with increasing vision distance. It is well known that GPUs perform extremely well when the application exhibits good locality, and delivers significant speedup compared to a traditional CPU. The better performance of GPUs in such cases is due to various optimizations specific to GPUs, including asynchronous memory operations, and better system tuning for 2-dimensional locality. However, when locality decreases, the performance edge of GPUs over CPUs decreases; this is partly due to the relatively larger sizes of L1 and L2 caches of CPUs. These effects are clearly seen with the segregation model, with increasing agent vision distance. On scenarios with low vision distance (consequently, high locality), a speedup of almost 70× is observed. Towards the poorer end, a vision of 16 brings the speedup down to 16×. It is nonetheless noteworthy that a speedup, rather than slowdown, is still observed even on such a large vision value that requires the agents to consider a very large number of neighboring agents. Note also that the performance effects of vision distance are not so pronounced until the large scenarios containing 0.25 million agents are considered.

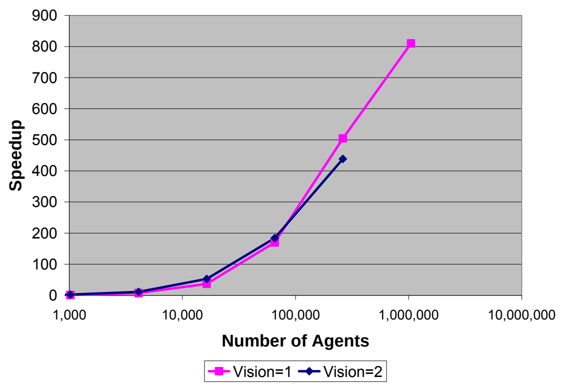

Figure 9. GPU speedup compared to NetLogo for Segregation Model

3.4 Frame Rates and Interactivity

The previous performance charts serve to illustrate the speedup that could be achieved when ABM simulations are executed on GPUs as opposed to on CPUs, either with optimized CPU execution or with traditional CPU-based ABM toolkits. Let us now consider absolute speed of these simulations, for an understanding of their relation to real-time needs. The metric of interest is the number of iterations (time steps) that could be executed within one wall-clock second.

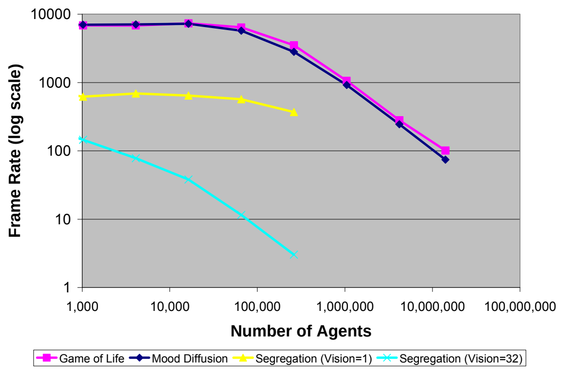

Figure 10. Number of iterations per wall-clock second (absolute frame rate) achieved with the models on GPU

Clearly, a frame rate of 30 frames per second (fps) is sufficient for interactivity. However, faster than 30fps is desirable for models in which not every frame needs to be visualized (*i.e.*, flushed to the screen), or in which periodic snapshots are to be displayed to verify satisfactory progress towards the desired goals for emergent phenomena. Such faster-than-real-time speed is clearly achieved with models such as Game of Life and Mood Diffusion, even on the largest grid sizes. Figure 10 shows the frame rates achievable with GPU-based and CPU-based executions. With the GPU, the largest configurations of 14 million agents are seen to be delivered at the surprisingly high rate (over 7000fps with 1024 agents, to over 70fps with 14 million agents).

It is important to note that frame rate is limited by the concurrency afforded by the model. In the Segregation model, for example, variation in the vision distance is seen to significantly affect the frame rate, from super-interactive rates (620fps–370fps for the best case of vision=1) down to relatively slower rates (145fps–3fps for the worst case of vision=32).

3.5 On-line and Off-line Visualization

For off-line visualization purposes, we make use of a tool called imdbg, which is quite useful for debugging and testing the texture values between iterations. We utilized the EasyBMP tool for continuous animations which are created as a series of bitmaps; the bitmaps themselves are stitched together into a movie by using another tool called BMPtoAVI.

In large-scale, long-time simulations, it is not necessary to visualize each and every iteration that is executed. If the specific evolution of interest is slow, a high rate of state evolution can be sustained by computing at GPU rates, while the state is only periodically paused to be flushed to the screen, say, once every 100 iterations. Since the graphics processor is already built for rendering, it is relatively straightforward to obtain a first-order visualization of state evolution using a suitable encoding that maps state values to pixel colors.

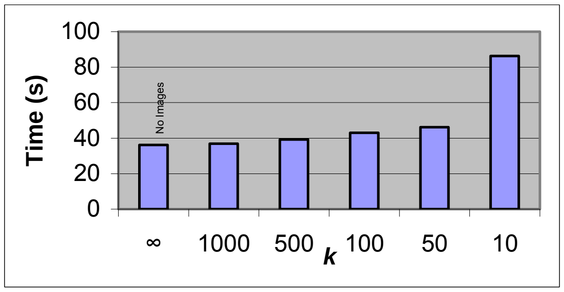

Figure 11. Time to Generate Infrequent Snapshot Images once every *k* Iterations

A natural question that arises is: how frequently can the GPU-based state be offloaded from the GPU to either the display and/or the CPU memory while still maintaining speedup by GPU-based simulation. To answer this, we performed experiments that read the current state of the simulation state every *k* iterations, where *k* is varied from 10 to 1000. Figure 11 shows the speedup when *k* is varied, where the runtime includes the cost of reading the pixel values from GPU to CPU memory. The data shows that infrequent on-line visualization can be used if needed, without any significant loss of speed to native GPU computation rates. The data shows that generating an image once every 50 or 100 iterations cost little more than not generating any image snapshots at all.

4 DATA PARALLEL CHALLENGES

We now identify some of the challenges that arise when ABM execution in general is mapped onto data-parallel execution platforms such as GPUs and multi-core processors: The challenges arise due to the potential conflict between semantics of ABM primitives and model execution on parallel platforms. Here we focus only on shared-memory platforms. We abstract the challenges into four categories: (1) Random Affect (2) Scheduling Policies (3) Aggregation Operations (4) Asynchrony. Each of these challenges is described next, together with some solution approaches.

4.1 *Random Affect*

One of the biggest challenges in ABM execution on data parallel platforms concerns the issue of a modeling paradigm that we will call "Random Affect." In this paradigm, in general, agents randomly select other agents and/or other grid cells to affect the behavior in some fashion. The nature of affecting the chosen agents/cells (here we will refer to cells and agents collectively on an equal footing) varies with the particular model under question, but the underlying fundamental paradigm remains the same. The type of affect, for example, could be movement, spawning, tagging and so on.

In many agent models, for example, agents move from one grid cell to another over the course of simulation. Most typically, the movement is randomized in some fashion (e.g., moving to a random destination cell chosen from among all vacant neighboring cells). Agent movement is an example of such a random affect operation. Another example of random affect activity is agents spawning other agents (e.g., Predator-Prey models). To realize such activity, several agents must, at each iteration, choose randomized destination cells at which new agents are spawned. Yet another example of random affect is a "tagging" type of activity, in which agents randomly select other agents to tag them in some way, such as infected agents marking other selected agents as infected in an epidemiological model.

The random affect paradigm in general often calls for such overall randomized activity to be (a) simultaneously performed by many agents (b) randomized among all possible source-destination pairs without bias/advantage for/against any particular agents or cells (c) successful if the activity is indeed possible. While these conditions are intuitively obvious and are also easily coded in a sequential execution context, they are not as easy to ensure and realize outside sequential execution. Each condition in fact imposes its own nature of difficulty of realizing in a data parallel execution context. The principal difficulty is coming up with algorithms that satisfy all the conditions together.

Randomized Bi-Partite Mapping

An abstraction of the core problem is to create a random bi-partite graph of the grid elements. This is illustrated in Figure 12, in which eight sources S1...S8 are being mapped to six potential destinations D1...D6. In this example, while most sources are assigned a random destination, sources S4, S6 and S8 remain unassigned to any destination, and also no source is mapped to destination D5. However, the property that is maintained in the graph is that each source is assigned to at most one destination, and vice versa, and, more importantly, the pairings are random.

The difficulty of the bi-partite mapping problem arises from the facts that (a) the compositions of the source agent/cell set and destination agent/cell set vary dynamically from one iteration to the next, and (b) the mapping from sources to destinations must be randomized across iterations, to avoid introducing any artificial bias into the model. For example, in the segregation problem, the two partitions are: occupied cells on one partition, and vacant cells on the other partition. In fact, a more specific partition would include in the first partition only those occupied cells that are unhappy, and, consequently, would like to move from their current position.

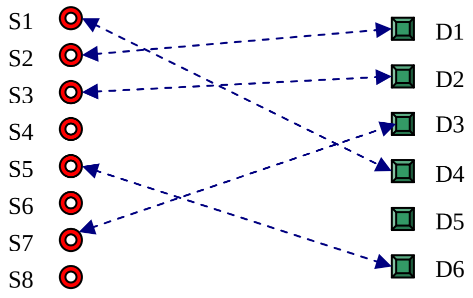

Figure 12. Example of a bi-partite assignment between sources and destinations

In the data-parallel execution context, the problem of creation of a random bi-partite graph at every iteration is comprised of two sub-problems:

1. Exclusion: Ensure that exactly one destination is assigned to any given source. In a data parallel context, a perfect algorithm is impossible to ensure this, so this problem must sometimes be relaxed to one that ensures that each source is mapped to at most one (instead of exactly one) destination (cell).

2. Information Propagation: For a given source-destination pair, convey the destination identifier to the source cell and the source identifier to the destination cell.

Noteworthy is that these two considerations hold even when only immediate neighborhood is considered for random movement. In other words, the bi-partite problem is independent of connectivity among source and destination cells.

The first sub-problem, in fact, is the more difficult one to resolve. The latter concern (information propagation) is somewhat less problematic in data-parallel executions which have both scatter and gather memory operation support (e.g., CUDA[22]). On platforms with only gather (but no scatter) support, it is still possible to couple the information sharing with one-to-one mapping solution.

While traditional multi-threaded data-parallel programs solve the information propagation problem using synchronization primitives such as locks and semaphores, the very fine-grained nature of computation in ABM simulation seems to indicate that lock-free solution would be much better suited for this mapping problem; lock-based solutions will incur significant overhead for updating data structures and are also prone to initiating non-local writes that are detrimental to cache performance.

Lock-free Select-Backoff Algorithm

We designed a novel two-pass algorithm for generating such a bi-partite mapping that is randomized across iterations in an ABM execution. The pseudo code for the algorithm is shown in Figure 13. In the first pass, all destinations select a source at random. In the second pass, every source checks how many destinations have selected itself as their source. If the source detects that exactly one destination has selected this source, then the source accepts that assignment and keeps it. If less than one destination or more than one destination, selected this source, the source rejects all assignments (*i.e.*, stays unmapped). The destinations also, in the second pass, need to determine if their selected source has accepted or rejected their selection. Each destination determines such acceptance/rejection of its selection by evaluating the same conditions as the source does, essentially duplicating the computation of the selected source but within the context of the destination.

```
Pass 1: For each agent A_ij
    If A_ij is a source
        Do nothing
    Else (A_ij is a destination)
        Within the neighborhood of vision v,
         Randomly select a source
             S (tentative)
Pass 2: For each agent A_ij
    If A_ij is a source
        Within the neighborhood of vision v,
         Find the number of destination agents
          who have picked A_ij as their source
        If the number is exactly equal to 1
            Mark self as mapped to that
            Unique destination
    Else (A_ij is a destination)
        If this agent has a source selected
            S(tentative)
           Examine the neighborhood of S to
             verify that A_ij is the only
             destination that selected S
           If A_ij is unique in selection of S
             Mark self as mapped to that S
```

Figure 13. Lock-free randomized bi-partite algorithm

We used this algorithm to model movement in the data parallel execution of segregation model on the GPU, with varying values of vision, as described in Section 2.

4.2 Scheduling Policies and Behavioral Semantics

Often, agent-based models are specified with the notion of sequential execution and/or synchronized execution. For example, the segregation model contains an inherently serialized mode of migration across cells. The randomization procedure of mapping between empty cells to cells that desire to relocate is sequential in nature. If this specification is relaxed in favor of a data parallel setting, the overall emergent phenomenon materializes at very low rates. For example, if agent movement is restricted to a vision of 1, segregation is seen to occur at a greatly reduced rate. Segregation is observed to occur much faster when the vision encompasses a good fraction of the larger grid. However, the farther the vision, the more inherently sequential the specification becomes. The tight dependency among agents in Segregation model has already been previously mentioned in Ref [1].

For models with such serial execution semantics built into their behavior specification, one cannot hope to obtain satisfactory parallel speedup. Ideally, for effective parallelization, behavioral models have to be expressed in an execution-independent fashion. Perhaps the source of serial nature of conventional ABM is historical, due to social scientists experimenting manually by hand with various possible models (e.g., pebble games). Now that ABM is computer-aided, execution-decoupled specifications of asynchronous behavior might be more appropriate.

One of the advantages of data parallel execution is that all agents can be simultaneously updated without artificial bias/advantage towards any agents. However, the same feature can become a disadvantage if some specific scheduling policy is prescribed the model specification. Any scheduling policy other than random, independent invocations of agent activity at each step is difficult to realize efficiently. One solution is to redefine the model, if necessary, in terms of independent update semantics while still retaining the semantics of the original emergent behavioral phenomenon.

4.3 Aggregation Operations

Another challenge that is harder in data parallel (as opposed to sequential) execution is the implementation of aggregation operations. Aggregation operations include statistics (e.g., averages, counts), instrumentation (e.g., number alive/dead), termination condition detection (e.g., stop when number alive reaches equilibrium). While such operations are straightforward to implement sequentially, new scalable algorithms are required to implement the same in parallel.

An important, almost universal, parallel operation is the so-called “parallel scan” or “parallel reduction” operation (e.g., algorithms optimized for GPUs to run in $O(\log(n))$ time for n agents [23]). However, care must be taken to ensure that not too many such operations are invoked for every iteration.

In the Schelling Segregation model, for example, we needed to perform a parallel reduction operation in order to determine if/how many agents remain that are unhappy. This count was used to determine when a reasonably stable situation was reached. Since agent state is scattered in general, and not all GPU processors can access all data equally fast, aggregation operations have to be carefully included into the model for optimal runtime. Too liberal a use can result in loss of speedup. The tension between the need for aggregate operations in the model and the cost of executing aggregate operations on the data parallel platform is one of the important challenges.

4.4 Asynchrony: Semantics and Execution

ABM simulations are commonly defined in terms of time-stepped execution, in which time is advanced globally in fixed increments, and all agents are updated at each time step. However, some ABM simulations are either inherently asynchronous in their formulation, or amenable to an alternative, asynchronous execution style for faster evolution. In such asynchronous execution models, updates to agents are processed via staggered timestamps across agents (e.g., [24] and [25]). While it is relatively straightforward to map synchronous (time-stepped) execution to data parallel platforms, the mapping of asynchronous execution is not so obvious. Efficient execution of asynchronous activity requires much more complex handling. The fundamental issue at hand is the preservation of correct causal dependencies among agents across time instants. Correctness requires that agents be updated in time-stamp order [26]. A time synchronization algorithm is needed to resolve these challenges. We are currently developing such an algorithm (e.g., [27, 28]) to ABM specification and execution.

5 SUMMARY AND FUTURE WORK

The data parallel execution capabilities of GPUs are clearly useful for fast simulation of agent-based models. The empirical results from our experiments show that high simulation speeds can be achieved even with very large agent populations. In some of the largest configurations we tested, more than 14 million agents can be executed at hundreds of iterations per wall-clock second. The speeds on the GPU are roughly two to three orders of magnitude higher than those of extant ABM tool kits. Such phenomenal performance differential makes it a very compelling argument to perform further research in making the GPU-based execution more accessible to ABM researchers. Automated translation mechanisms and/or new programming interfaces are some of the potential approaches that could help move from existing models to scalable platforms.

While the performance gains are promising in representative, explored models, additional research is needed to address data parallel issues for more generalized ABM execution. Issues of correctness and performance have to be resolved, especially to prevent artificial biases from being introduced due to performance optimizations. We believe bias would be among the most important factors that need to be considered when parallelizing ABM execution. Alternatively, classes of models must be identified that are resilient to bias and/or "approximated execution" of agent rule sets. Aside from runtime speedup, other considerations arise, such as programmability and usability, which have to be addressed before the GPU platform can be used in the mainstream for ABM to realize the speed gains. To this end, we are currently developing a GPU-based ABM framework called GARFIELD to help mitigate the usability concerns. GARFIELD currently is able to receive customized input models and present interactive runtime animation of the evolution of the system.

ACKNOWLEDGEMENTS

Constructive comments by ORNL internal reviewers have helped improve the presentation. 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.

REFERENCES

1. North, M.J. and C.M. Macal, *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. 2007: Oxford University Press
2. GPGPU. *General Purpose Computation Using Graphics Hardware*. 2005; http://www.gpgpu.org.
3. Pharr, M. and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005: Addison Wesley Professional
4. Buck, I., et al., *Brook for GPUs: Stream Computing on Graphics Hardware.* ACM Transactions on Graphics, 2004. **23**(3): p. 777-786.
5. Owens, J.D., et al. *A Survey of General-Purpose Computation on Graphics Hardware*. in *Eurographics*. 2005.
6. Intel-Corporation. *Intel Multi-core Technology*. 2007; http://www.intel.com/multi-core.
7. *Top 500 Supercomputing Sites*. 2007; http://top500.org.
8. Courty, N. and S.R. Musse. *Simulation of Large Crowds in Emergency Situations Including Gaseous Phenomena*. in *IEEE Computer Graphics International*. 2005..
9. Reynolds, C. *Big Fast Crowds on PS3*. 2006; www.research.scea.com/pscrowd.
10. Tomov, S., et al., *Benchmarking and Implementation of Probability-based Simulations on Programmable Graphics Cards.* Computers and Graphics, 2005. **29**(1).
11. Walter, B., et al. *UAV Swarm Control: Calculating Digital Phermone Fields with the GPU*. in *IITSEC*. 2005.
12. Verdesca, M., et al. *Using Graphics Processor Units to Accelerate OneSAF: A Case Study in Technology Transition*. in *IITSEC*. 2005.
13. Moya, L. and A. Tolk, *Towards A Taxonomy of Agents and Multi-Agent Systems*, in *Agent-Directed Simulation Symposium*. 2007, ACM: Norfolk, VA, USA.
14. Neumann, R. and F. Strack, *The Automatic Transfer of Mood between Persons.* Journal of Personality and Social Psychology, 2000. **79**(2): p. 211-223.
15. Hess, J.D., J.J. Kacen, and J. Kim, *Mood-management Dynamics: The Interrelationship between Moods and Behaviors.* British Journal of Mathematical and Statistical Psychology, 2006. **59**(2): p. 347-378.
16. Gardner, M., *Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"*, in *Scientific American*. 1970. p. 120-123.
17. Schelling, T., *Micromotives and Macrobehavior*. 1978: W. W. Norton
18. North, M.J., N.T. Collier, and J.R. Vos, *Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit.* ACM TOMACS, 2006. **16**(1): p. 1-25.
19. Wilensky, U., *NetLogo*. 1999, Center for Connected Learning and Computer-Based Modeling, Northwestern University: Evanston, IL.
20. Fernando, R. and M.J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. 1 ed. 2003: Addison Wesley Professional
21. Perumalla, K.S. *Discrete Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)*. in *IEEE/ACM/SCS PADS*. 2006.
22. NVIDIA. *NVIDIA CUDA*. 2007; http://developer.nvidia.com/cuda.
23. Sengupta, S., et al., *Scan Primitives for GPU Computing*, in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 2007, Eurographics Association: San Diego, California.
24. Epstein, J., *Modeling Civil Violence: An Agent-based Computational Approach.* PNAS, 2002. **99**(3): p. 7243-7250.
25. Nutaro, J., *Parallel Discrete Event Simulation with Application to Continuous Systems*, in *Department of Electrical and Computer Engineering*. 2003, University of Arizona: Tucson, AZ. p. 182.
26. Perumalla, K.S., *Model Execution*, in *Handbook of Dynamic System Modeling*. 2007, CRC Press.
27. Fujimoto, R.M., *Parallel and Distributed Simulation Systems*. 2000: Wiley Interscience
28. Perumalla, K.S. *Parallel and Distributed Simulation: Traditional Techniques and Recent Advances*. in *Winter Simulation Conference*. 2006. INFORMS.