Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU

Mingyu Yang
TUMCREATE Ltd and
Nanyang Technological University, Singapore
yangmy@ntu.edu.sg

Philipp Andelfinger
TUMCREATE Ltd and
Nanyang Technological University, Singapore
pandelfinger@ntu.edu.sg

Wentong Cai
Nanyang Technological University, Singapore
aswtcai@ntu.edu.sg

Alois Knoll
Technische Universität München, Germany
knoll@in.tum.de

ABSTRACT

Graphics processing units (GPUs) have been shown to be well-suited to accelerate agent-based simulations. A fundamental challenge in agent-based simulations is the resolution of conflicts arising when agents compete for simulated resources, which may introduce substantial overhead. A variety of conflict resolution methods on the GPU have been proposed in the literature. In this paper, we systematize and compare these methods and propose two simple new variants. We present performance measurements on the example of the well-known segregation model. We show that the choice of conflict resolution method can substantially affect the simulation performance. Further, although methods in which agents actively indicate their interest in a resource require the use of costly atomic operations, these methods generally outperform the alternatives.

ACM Reference Format:
Mingyu Yang, Philipp Andelfinger, Wentong Cai, and Alois Knoll. 2018. Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU. In *Proceedings of SIGSIM-PADS '18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM (SIGSIM-PADS '18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3200921.3200940

1 INTRODUCTION

Agent-based simulation is widely applied to evaluate systems in domains such as traffic engineering and biology. In contrast to macroscopic simulations, agent-based simulation considers the interactions between the participating entities in detail, incurring substantial computational load. Since most agent-based simulations exhibit a certain degree of locality w.r.t. the simulation space and involve state updates for all agents at the same logical time, graphics processing units (GPUs) have proven to be well-suited for parallelization of agent-based simulations.

If at a given point in logical time, all agent states are updated concurrently, multiple agents may request the same resource, e.g., a location in the simulation space. If the simulation model does not specify how such conflicts are to be resolved, a generic method is required. We suggest that to generate meaningful simulation results, such a method should exhibit three properties: exactly one agent should acquire the resource, the outcome should be deterministic, and no bias should be introduced.

Sequential simulators perform agent state updates one after the other. If considering the actions taken by each agent's predecessors, conflicts are avoided entirely. While some models specify such a one-by-one update of agents (e.g., [11]), the opportunities for parallel processing are severely limited [6]. In the GPU context, conflict resolution is complicated by the indeterminism in the progress of the processing elements. The key challenge is to produce deterministic and unbiased results while still achieving high performance.

In this paper, we make the following contributions: **1.** We systematize the GPU-based conflict resolution approaches from the literature and discuss how bias can be avoided. **2.** We propose two new conflict resolution variants. **3.** We present a performance evaluation of the existing and proposed conflict resolution approaches[1].

2 FUNDAMENTALS

In agent-based simulation, the agent state is represented by a set of variables associated with each agent. In the following, we assume that the simulation proceeds in *cycles*. During each cycle, all agents update their states, accessing only the previous state of other agents.

We define a **conflict** as a situation in which an agent requests a resource that has already been requested by another agent at the same logical time. An example is given by the Sugarscape model [1], in which agents compete for pieces of sugar. In the case of discrete spatial simulations, multiple agents may request the same unoccupied cell on a grid, yet only one of the agents (the "winner") can occupy that cell. The goal of a conflict resolution method is to determine exactly one winner for each conflict. Roughly, conflict resolution involves two steps: first, each agent indicates its interest in a resource. Second, a tie-breaking mechanism determines the winner. All remaining agents then select another resource. A simulation is **deterministic** iff the same result is obtained from repeated simulation runs using the same pseudo-random number generator seed [8]. Determinism is considered important for analyzing simulation results and for debugging [3].

[1] https://github.com/GPUCR

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200940

A model may specify rules for breaking ties. However, we consider the case where the model does not specify a tie-breaking policy. Thus, we postulate that all agents interested in the same resource should have the same probability of acquiring the resource, i.e., given n conflicts with m involved agents on average, the expectation for the number of conflicts won should be n/m for each agent. This requirement implies that the tie-breaking mechanism should not systematically favor certain agents or states, which we refer to as **bias**. As an example, a biased tie-breaking mechanism in a traffic simulation may favor vehicles entering a road from a certain direction, which could introduce behaviors not specified in the model. The required considerations are similar to the handling of simultaneous events in discrete-event simulations [8].

To illustrate the principles of conflict resolution on a simple example, we consider the **segregation model** by Schelling [11], in which agents compete for locations in a two-dimensional cellular simulation space. However, the considered conflict resolution methods are also applicable to more complex models, e.g., when agents move on a graph or compete for resources other than locations.

In the segregation model, agents are assigned one of two types. The "happiness" of each agent is calculated based on the number of agents of the same type in the eight adjacent cells. In each simulation cycle, each unhappy (*moveable*) agent moves to a random position in a configurable neighborhood. Conflicts occur whenever multiple agents attempt to move to the same position. We consider cells containing moveable agents as unoccupied. Thus, there are at least as many unoccupied cells as moveable agents.

3 CONFLICT RESOLUTION METHODS

A simple way of resolving conflicts is used in the MatSim traffic simulator [12]: agents attempt to obtain a resource by atomically writing to a variable associated with each resource. The earliest agent successfully obtains the resource, whereas the other agents fail. The atomic write operation ensures that exactly one agent obtains the resource. However, determinism is not among the design goals of MatSim [5]. Although the method identifies a winner for each conflict, if no additional action is taken, the results depend on the execution order among the processing elements. In the following, we only consider deterministic approaches.

We propose a classification of the existing approaches into two categories, *push* and *pull*, which are differentiated by the manner in which potential assignments between agents and resources are written to memory. In push approaches, agents actively try to obtain the resources by writing to a variable associated with the desired resource. Generally, push approaches require the use of atomic operations to control concurrent accesses to the resources. In pull approaches, possible assignments are stored locally by the active entities. For instance, if agents take the active part, each agent stores the determined assignment in a per-agent variable. Thus, no atomic operations are required. Subsequently, scanning is performed by the resources to determine the interested agents.

We further differentiate among tie-breaking methods: with *incremental tie-breaking*, predefined priorities are applied as the agents register their interest, so that a winner has been identified once the last agent has registered. With *postponed tie-breaking*, the interested agents are explicitly stored in a list. Subsequently, the list is sorted and a pseudo-random number is drawn to determine the winner.

Algorithm 1 Iterative Push

```
1:  while A ≠ ∅ do
2:    for each a ∈ A in parallel do
3:      a.r ← SelectResource(R)
4:      AtomicMax(registry[a.r], a.priority)
5:    Synchronize()
6:    for each r ∈ R in parallel do:
7:      priority ← registry[r]
8:      if priority ≠ nil then
9:        a ← GetAgent(priority); Assign(a, r)
10:       A ← A \ {a}; R ← R \ {a.r}
11:   Synchronize()
```

For brevity, we describe all methods using incremental tie-breaking, however, our performance evaluation covers both approaches.

3.1 Push

In push approaches, agents actively indicate their interest in a resource by writing to a variable associated with the resource.

Iterative Push: Lysenko et al. [2] proposed a conflict resolution method based on atomic operations. In the first stage, each agent attempts to atomically write a unique priority to a per-resource variable. The assignment of suitable priorities will be discussed in Section 4. An atomic maximum operation ensures that the final result holds the value written by the agent with the highest priority. After performing a global synchronization to guarantee that all results of the first stage have been written to memory, in the second stage, each resource checks whether it has been selected by an agent. If that is the case, the agent is assigned the resource. The two stages are repeated for a number of *iterations* until all agents have acquired a resource. Iteration here is defined as a series of kernel calls that complete the assign and acquire stages. Pseudo-code is provided in Alg. 1 where A is the set of agents intending to obtain a resource, and R is the set of resources.

Non-Iterative Push: We propose a variant of Alg. 1 that requires only one iteration per simulation cycle: each agent attempts to store its priority in a per-resource variable using an atomic maximum operation. In contrast to Alg. 1, each agent considers the previously stored priority returned by the atomic operation to determine whether its priority is the current maximum. If another agent already registered a higher priority, the current agent immediately attempts to obtain another resource. Otherwise, if the agent has displaced a previously registered agent, the current GPU thread takes control of the displaced agent and repeats the procedure until it has registered the highest priority for a resource. Since displaced agents are moved immediately, a simulation cycle concludes in one iteration. With non-iterative push, the overall number of conflicts is larger than with iterative push, where agents that have already obtained a resource do not take part in subsequent iterations. However, due to a reduction in memory accesses and synchronization points, our approach still outperforms iterative push (cf. Section 5).

3.2 Pull

In pull approaches, write accesses by agents and resource are limited to local variables. Thus, atomic operations are avoided. In an early work on GPUs, a pull approach was proposed by Perumalla et al. [6]: first each resource selects a random neighboring agent and stores its identifier locally. Each agent then scans for resources that have selected the respective agent. If the number of resources is

Algorithm 2 Iterative Pull

```
1:  while A ≠ ∅ do
2:    for each a ∈ A in parallel do
3:      a.targetResource ← SelectResource(R)
4:    Synchronize()
5:    for each r ∈ R in parallel do
6:      A_r ← ∅
7:      for each a in the neighborhood of r do
8:        if a.targetResource = r then
9:          A_r ← A_r ∪ {a}
10:     if |A_r| ≥ 1 then
11:       a_r ← MaxPriority(A_r); Assign(a_r, r)
12:       A ← A \ {a_r}; R ← R \ {r}
13:   Synchronize()
```

exactly one, there is no conflict and the agent can obtain the target resource. Otherwise, the resource is not obtained by any agent. Since this approach does not resolve conflicts competing for the same resource, it is not included in our evaluation.

An iterative pull approach is used in FLAME GPU [7]. First, each agent selects a resource and stores a resource identifier in a per-agent variable. Then, each resource scans for interested agents. The winner of a conflict is determined based on the agents' priorities. This process is repeated until all agents have obtained resources. The approach avoids atomic operations. However, scanning for interested agents incurs substantial overhead if a large neighborhood is considered. The iterative pull approach is shown in Alg. 2.

3.3 Sampling and Permutation

We propose a simple conflict resolution approach targeting models where agents compete for resources selected uniformly at random from a global set. With unrestricted agent movement, Schelling's segregation model is an instance of this model class. The approach is based on the observation made previously by Perumalla et al. [6] that the desired result of a simulation cycle is a random injective mapping between agents and resources. We directly determine such a mapping by a two-step approach. First, a random sample is drawn from the resources. Second, a random permutation of the agents is computed to determine the mapping to the resources (cf. Fig. 1). Pseudo-code is provided in Alg. 3. For random sampling and permutation, we rely on parallel algorithms by Sanders et al. [9, 10]. For random sampling, we ported existing CPU code[2], whereas random permutation was implemented from scratch. For Schelling's segregation model with limited neighborhoods, the approach is not applicable: since agents compete for overlapping sets of resources, the probability of selecting a given resource varies among agents.

4 AVOIDING BIAS

Clearly, the requirement of unbiased conflict resolution (cf. Sec. 2) is satisfied by *postponed* tie-breaking: once the set of interested agents has been identified, the agents are sorted and a random number is drawn to select the winner of the conflict. However, with *incremental* tie-breaking, care must be taken not to introduce undesired bias into the simulation. We illustrate the issue using Schelling's segregation model: it may seem natural to use the agents' current position as their priorities. However, if the neighborhood considered for movement is limited, this choice of priorities introduces a bias into the movement directions of agents. Consider the situation

[2]https://github.com/sebalamm/DistributedSampling

Algorithm 3 Sampling and Permutation

```
1: R_S ← RandomSample(R, |A|)
2: A_P ← RandomPermute(A)
3: Synchronize()
4: for each a_i ∈ A_P in parallel do
5:   Assign(a_i, R_S[i])
```

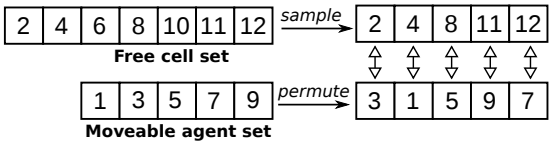

Figure 1: Sampling and Permutation.

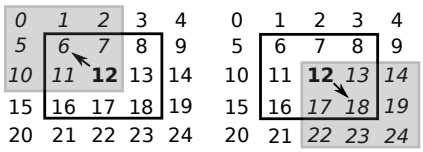

Figure 2: Illustration of bias caused by fixed priorities.

in Figure 2, where agent priorities are chosen by their current positions. Agents may move to locations in a 3×3 neighborhood. We show two possible target positions chosen by the agent at position 12. The gray rectangles denote the source position of agents that may compete for the same position. If the agent intends to move to position 6, any competing agent will have a lower priority, i.e., the agent will always win the conflict and be able to move. In contrast, if the agent intends to move to position 18, any competing agent will have a higher priority, i.e., the agent will lose the conflict and will have to select a new target position. Since this type of asymmetry affects the other reachable target positions as well, the general tendency for agents is to move to the top left of the simulation space. The bias is more pronounced at small neighborhood sizes.

Independence between winning probabilities and agent states can be achieved by choosing random priorities for each simulation cycle. We extended the push approach by generating a random permutation and assigning the results as the agents' priorities, which is similar to shuffling the agents' movement order in sequential agent-based simulations. It is necessary to generate a new permutation at the beginning of each cycle to avoid introducing a randomized but consistent bias pattern. When permuting the priorities at each cycle, the relative priorities of the competing agents are chosen without favoring certain agents or agent states systematically. Thus, as with postponed tie-breaking, the results are unbiased.

5 PERFORMANCE EVALUATION

We measured the performance of the different conflict resolution approaches using the segregation model on a system equipped with a NVIDIA Tesla K20Xm with CUDA 7.5. 99% confidence intervals are plotted but are too small to be visible.

Figure 3 compares the execution times of simulations using iterative push, non-iterative push and sampling and permutation when agents move without spatial restrictions. Since in the pull approach, resources scan the entire grid for interested agents, it is about 2000 times slower than the other approaches and thus excluded from the

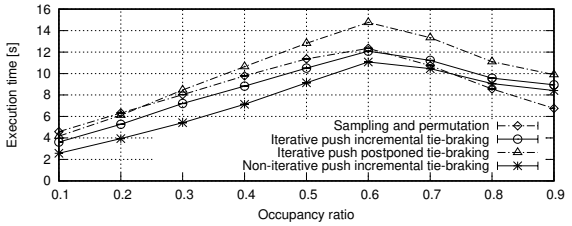

Figure 3: Simulation runtime with global movement.

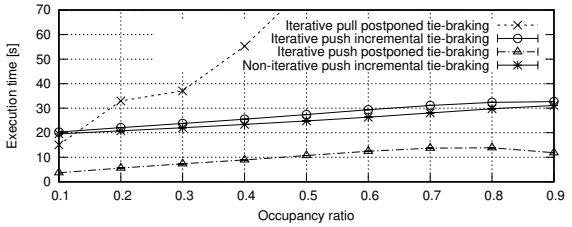

Figure 4: Simulation runtime with limited neighborhood.

figure. As the non-iterative approaches depend on incremental tie-breaking, combinations with postponed tie-breaking are excluded as well. We varied the occupancy ratio, i.e., the ratio of populated cells, from 0.1 to 0.9. The simulation space was a grid of about 16 million (4096×4096) cells. The agents' happiness threshold was set to 5. The simulation was terminated after 100 simulation cycles. The results show a minor runtime reduction of non-iterative push over iterative push. The sampling and permutation approach performs worse under low occupancy ratios. However, for occupancy ratios above 0.7, sampling and permutation outperforms the push approaches. The reason is that at high occupancy ratios, with the push approaches, agents will perform many retries until an open cell is found and potential conflicts are won. Sampling and permutation avoids these situations by directly mapping agents to the open cell set. At a smaller grid size of 1024×1024 cells, the relative performance of the push approaches remained roughly the same. However, sampling and permutation was outperformed by a factor of about 2.9 to 5.2. Generally, the performance is affected by the number of moveable agents and conflicts. The largest number of conflicts was generated at an occupancy ratio of 0.6, coinciding with the longest observed execution times.

The performance of iterative push is affected by the number of iterations required to resolve all conflicts, which can be computed by iteratively determining the number n of agents that lose a conflict according to the birthday paradox [4]: $n = |A| - |C| + |C|(1 - \frac{1}{|C|})^{|A|}$, where $|A|$ is the number of agents and $|C|$ is the number of cells.

Figure 4 shows the execution times of the various approaches when the agent movement is limited to a 3×3 neighborhood on a grid of 4096×4096 cells. For incremental tie-breaking, the approach described in Section 4 to avoid bias is used. As discussed in Section 3.3, sampling and permutation cannot be apply to restricted neighborhoods. In contrast to the unrestricted movement, the incremental tie-breaking approaches require the computation of random permutations in order to avoid bias. If an agent cannot find an unoccupied cell, its neighborhood is enlarged in steps of 3 cells after 10 trials each. We observed neighborhoods up to 63×63 cells at occupancy ratio 0.9. Since in the pull approach, the cells scan for interested agents, they have to consider the largest neighborhood used by any agent in the current iteration. Accordingly, iterative pull was substantially slower than the push approaches, requiring up to 1127 seconds at an occupancy ratio of 0.9. Given these results, we did not implement iterative pull with incremental tie-breaking. Iterative and Non-iterative push with incremental tie-breaking requires the computation of a random permutation for each simulation cycle. Since postponed tie-breaking does not require this step, it consistently achieved the lowest execution times.

6 CONCLUSIONS AND OUTLOOK

We systematized and evaluated conflict resolution approaches for agent-based simulation on GPUs from the literature and proposed two new variants. Our measurements indicate that if agents compete for resources globally without restriction to a certain neighborhood, a non-iterative approach achieves best performance. If the numbers of agents and conflicts are both large, a direct computation of a random mapping between agents and resources performed the best. If agents consider resources within a limited neighborhood, a postponed tie-breaking between competing agents substantially outperformed the alternatives. We further discussed ways to avoid bias in the conflict resolution. Our current observations were made only based on the classic segregation model, which is simple but often used to illustrate the power of agent-based modeling. Future work could extend our observations to more practical applications such as traffic simulation.

7 ACKNOWLEDGMENTS

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

REFERENCES

[1] Joshua M Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom up.* Brookings Institution Press.
[2] Mikola Lysenko and Roshan M. D'Souza. 2008. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation* 11, 4 (2008), 10.
[3] Thom McLean and Richard Fujimoto. 2000. Repeatability in Real-Time Distributed Simulation Executions. In *Proceedings of the Workshop on Parallel and Distributed simulation.* IEEE, 23–32.
[4] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press.
[5] Kai Nagel, Dominik Grether, Ulrike Beuck, Yu Chen, Marcel Rieser, and Kay W Axhausen. 2008. Multi-Agent Transport Simulations and Economic Evaluation. *Jahrbücher für Nationalökonomie und Statistik* 228, 2-3 (2008), 173–194.
[6] Kalyan S Perumalla and Brandon G Aaby. 2008. Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs. In *Proceedings of the Spring Simulation Multiconference.* SCS, 116–123.
[7] Paul Richmond. 2014. Resolving Conflicts Between Multiple Competing Agents in Parallel Simulations. In *European Conf. on Parallel Processing.* Springer, 383–394.
[8] Robert Rönngren and Michael Liljenstam. 1999. On Event Ordering in Parallel Discrete Event Simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation.* IEEE, 38–45.
[9] Peter Sanders. 1998. Random Permutations on Distributed, External and Hierarchical Memory. *Inform. Process. Lett.* 67, 6 (1998), 305–309.
[10] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. 2018. Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online. *ACM Trans. Math. Softw.* 44, 3 (2018), 29:1–29:14.
[11] Thomas C Schelling. 1971. Dynamic Models of Segregation. *Journal of Mathematical Sociology* 1, 2 (1971), 143–186.
[12] David Strippgen and Kai Nagel. 2009. Using Common Graphics Hardware for Multi-Agent Traffic Simulation with CUDA. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques.* ICST, 62.