Resolving conflicts between multiple competing agents in parallel simulations

Paul Richmond[1]

Department of Computer Science, The University of Sheffield, UK
p.richmond@sheffield.ac.uk,
WWW home page: http://www.paulrichmond.staff.shef.ac.uk.

Abstract. Agents within multi-agent simulation environments frequently compete for limited resources, requiring negotiation to resolve 'conflict'. The negotiation process for resolving conflict often relies on a transactional or serial processes that complicates implementation within a parallel simulation framework. This paper demonstrates how transactional events to resolve competition can be implemented within a parallel simulation framework (FLAME GPU) as a series of iterative parallel agent functions. A sugarscape model where agents compete for space and a model requiring optimal assignment between two populations, the stable marriage problem, are demonstrated. The two case studies act as a building block for more general conflict resolution behaviours requiring negotiation between agents in a parallel simulation environment. Extensions to the FLAME GPU framework are described and performance results are provided to show scalability of the case studies on recent GPU hardware.

Keywords: Agent-Based Simulations, FLAME GPU, Graphics Hardware, CUDA, Conflict Resolution, Multi-Agent Competition

1 Introduction

Agent based modelling provides a natural mechanism for describing complex systems where agents are represented as a set of individuals with behavioural rules. By simulating agents and their interactions over a period of time, emergent behaviour can be observed, giving insight into processes of the system which the model represents. Agent based simulations are typically more computationally expensive than traditional top down equation based models as each individual and their interactions must be simulated. Previous work has shown that agent based simulations can be accelerated and scaled to increasing levels through the use of parallel [5, 10, 1, 7] and distributed methods [11]. In some cases [5, 10, 11] such methods are implemented as part of a simulator providing a high level interface for describing agents with a level of abstraction hiding the complexity of the parallel or distributed architecture from end users (modellers).

A common behaviour within agent based models is for agents to compete for some resource. A simple example of this is reflected by any agent based model consisting of a regular lattice based (grid) environment in which a sequential simulator moves agents between grid cells. To determine a movement strategy, each agent is free to examine the environment to locate free space (unoccupied by any other agent) and decide upon an optimal movement often driven by the availability of some finite resource. Reproducing this same model with a parallel simulation environment is less straightforward. If each agent simultaneously makes a decision to move there is a potential risk that multiple agents will move to the same grid space (especially if it is highly desirable). Translating models built upon serial abstractions in parallel simulators therefore requires that conflicts in movement can be resolved robustly.

Agent movement in lattice environments represents a special case where competition is introduced indirectly via the migration of a serial algorithm to a parallel environment. Agent competition is not however limited to examples involving movement. For example, any form of assignment may result in agents competing against one another. This paper presents a general method for implementing conflict resolution, demonstrated through two case studies implemented within the Flexible Large-scale Agent Modelling Environment for the Graphics Processing Unit (FLAME GPU) simulation framework. The first example is an implementation of the sugarscape model [3] demonstrating how movement collision avoidance can be implemented in parallel. The second case study demonstrates how an iterative approach to conflict resolution can be applied to an agent based implementation of a classic assignment task 'the stable marriage problem' [4]. The paper provides FLAME GPU background and shows how both case studies can be implemented through a series of iterative stages (or rounds) providing results which demonstrate performance characteristics.

2 FLAME GPU

The FLAME framework is an agent based simulation platform with implementations targeting parallel agent simulation on both distributed (FLAME) [5] and GPU architectures (FLAME GPU)[10]. Both implementations use the same underlying mode of an agent, a communicating stream X-machines (a form of extended state machine containing an internal memory). Agents are expressed as a set of states performing some 'behaviour' that updates an agents internal memory when transitioning from one state to another. Agents are able to communicate indirectly through messages stored in globally accessible message lists. After a message is sent it is persistent within the message list as read only data for a single simulation iteration. Ensuring agents never write and read from the same message list during a single agent function gives a natural synchronisation barrier which enforces a stream like paradigm, preventing race conditions and providing a robust format for mapping to parallel architectures.

FLAME for the GPU (or the FLAME GPU) is an implementation of FLAME which utilises graphics card hardware to accelerate FLAME agent models by automatically translating the model to optimised GPU code. Rather than using a custom model parser, FLAME GPU uses XSLT templates to generate a complete simulation in NVIDIA CUDA C code. The advantage of a code generation process is that performance overheads of having a programming API are avoided lending to extremely high performance. Whilst based on the same principles as FLAME, FLAME GPU has a number of key differences which are important to understand the features and limitations with respect to parallel simulation.

From a simulation perspective, FLAME GPU utilises the Single Program Multiple Data (SPMD) architecture of GPUs to map agent functions as individual GPU kernels operating over agent (and message) memory, stored as arrays of linearly offset data. FLAME GPU makes a distinction between mobile and non mobile agents (Cellular Automaton) so that performance can be optimised in each case. In order to optimise performance for common communication strategies a distinction is made between three common types of message communication, discrete (for cellular automaton), brute-force (for all to all communication) and spatial partitioning (for limited range interactions). In each case, message data is cached using an efficient algorithm to reduce the number of global memory accesses and improve performance.

In certain cases, FLAME GPU provides a massive performance increase over FLAME however it is best suited to large populations of relatively simple (in terms of agent memory requirements) agents. When large number of agents are simulated many GPU threads can be spawned providing an effective mechanism for hiding memory latencies through context switching. If low numbers of threads are used then hiding memory latency becomes more difficult. Similarly if large (complex) agents are used performance is impacted due to the the limited availability of registers. As such methods for solving conflicts and assignments within FLAME GPU must be highly parallel and not for example rely on a single 'large' agent that resolves conflict by having a global overview of the entire population.

3 Competition between agents on a lattice

Lattice based agent simulations are a common abstraction within serial agent based simulation software. In such simulation software the serial order of processing agents is either randomised to ensure fairness or based on a priority scheme which gives preference to agents according to some trait. In translating the movement of agents into a parallel architecture it is important that many agents can perform movement simultaneously creating the potential for conflict through competition at highly desirable locations. If the model is built upon the principle that a discrete spatial cell of the environment may be occupied by only a single agent [EA96] then this conflict must be resolved. Within a parallel simulator such as FLAME GPU there are a number of ways to address this competition in parallel. The simplest is to simply to relax the rule preventing grid cells from being occupied by multiple agents. Whilst computationally inexpensive this simplification ultimately changes the nature of the model and will produce significantly different results to the model which relies on sequential processing to avoid conflict. If the rule to relax cells from containing multiple agents is not appropriate then the alternative approach is to serialise some parts of the movement processes where conflicts are observed.

To demonstrate how to address the serialisation of movement conflicts between agents, this paper considers a model of an artificial society (the sugarscape model) proposed by Epstein and Axtell [3]. The sugarscape model in its simplest form it consists of a population of agents (a society) distributed across an environment with a renewable resource (i.e. sugar). Agents require sugar to survive and sequentially move to empty cells within the lattice environment (without breaking the constraint of a single agent per cell) to consume sugar. Each agent has a sugar store which is incremented by accumulating sugar from the environment. During each simulation step an agent is required to use up part of its sugar store to survive at a rate determined by its randomly assigned metabolism. Epstein and Axtell describe a number of more advanced rules including pollution, reproduction, seasonal environments, cultural connections and combat. For the purposes of this paper these additional rules are omitted as the purpose of this model is to demonstrate the use of transactional movement techniques in FLAME GPU agent simulations.

3.1 Implementing Sugarscape in FLAME GPU

It has already been shown by Lynsenko and D'Souza [6] that the sugarscape model is suitable for simulation on a GPU by scattering agents to collision map to resolve movement collisions. Within Lynsenkos work the use of atomic operations ensures that only the highest priority agent is able to occupy a single space. Agents which are superseded by higher priority agents attempt to move in a subsequent step which is repeated until all collisions are resolved. The implementation for the sugarscape agent model within FLAME GPU uses this same principle of iterative rounds of agent movement to resolve conflicts. The FLAME GPU implementation differs however in that atomic operations are replaced with a series of transactional bids (negotiation) placed through messages between agents. This approach of agent negotiation makes the processes applicable to any parallel simulator or to parallel architectures where atomic operations are not supported. More specifically the process to allow agent movement uses the following sequence of events which are defined within FLAME GPU as individual agent functions (an agent in this case represents a spatial cell which may or may not contain an agent);

1. Cells containing agents read in messages from the environment to determine the best place to move to. Once a target location is identified, they output a request to move as a message containing the targets location identifier and the agents identifier information.
2. Unoccupied cells read all request messages to determine if any neighbouring sugarscape agents would like to move to the location. If multiple agents request to move to the same cell then the cell uses the agents priority to determine which agent should move. After all requests have been considered the cell becomes occupied by the agent with the highest priority. It then

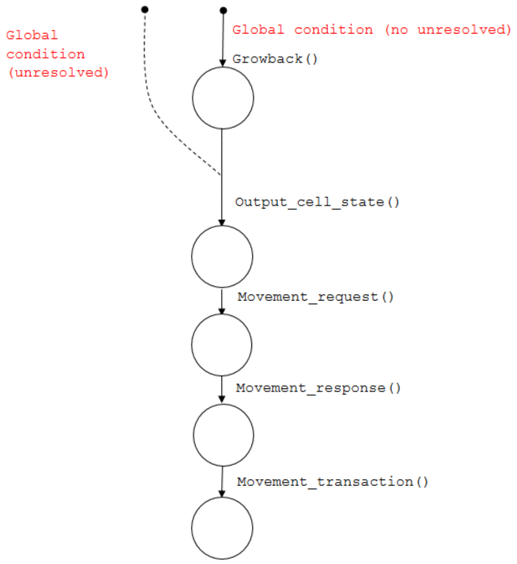


Fig. 1. The sugarscape model consists of a *Growback* agent function which performs the ‘normal’ agent behaviour e.g. consumption of sugar, etc. This function is only triggered if all agents are in a resolved state indicating that all agents have moved. If any agents are unresolved then a simulation step consists of agents performing an iterative round of bidding where only the highest priority agent is allowed to move should conflict occur.

sends a confirmation response with the agents identifier to notify the old cell that the agent has moved.

3. Cells containing agents which previously requested to move, check the confirmation response messages. If a confirmation is found the agent knows it has been relocated and updates the cell to an unoccupied state. If no confirmation is found then the agent remains at the same location.

The above steps represent a single iteration of agent movement and do not guarantee that all agents with the desire to relocate will actually do so. To overcome this problem, it is essential that the process is repeated until every agent has moved. To guarantee movement of all agents potentially requires the process to be repeated for every possible location that an agent could move into. With a simple vision radius of 1, 8 potential repetitions represent the worst case scenario. In many cases all agent movement can be resolved in far fewer iterations so a FLAME GPU global function condition is used to determine the global state of agent movement. A global function condition allows agents to perform a state transition (and perform the necessary agent function in doing so) only when a conditional statement is met by all agents (for which the function is applied). In this case agents are assigned two states, resolved and unresolved (with respect to their movement) and the global function condition only allows the population of agents to move to a resolved state when all agent movements are resolved. Only within simulation steps where all agents are resolved does a function performing the 'normal sugarscape behaviour', e.g. environment grow back, removal of sugar from the environment and feeding (according to the metabolic rate) take place. Following the normal sugarscape behaviour all agents enter the unresolved state resulting in the next simulation iteration performing only the collision resolution steps (Figure 1).

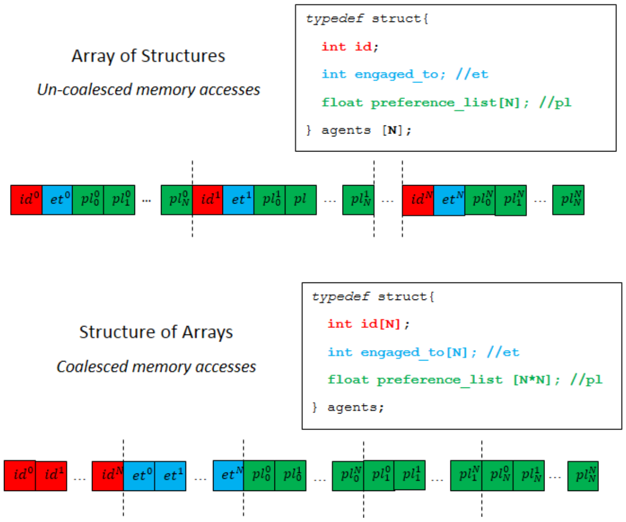
4 The stable marriage problem

Within agent based modelling the assignment of agents to resources (including other agents) is a common construct occurring within many models. For example within economics it is common to match sets of workers with firms [2]. To demonstrate how to implement assignment behaviour within parallel agent based models, the stable marriage problem is considered. The stable marriage problem is a example a classical two sided matching problem that can be applied not only to matching agents (i.e. marriage) but (through small variations) is equally applicable to other assignment problems. The stable marriage problem defines two sided matching as a matching between two equal sized sets of n men and n woman where each man and woman has a personalised ranking of all member of the opposite sex. The goal is to find a stable solution of matches where stability is defined as a set of matches where there are no two couples that would prefer to swap with each others partners.

The Gale-Shapley algorithm [4] is an iterative algorithm which guarantees to find a solution where everyone is married and marriages are stable. It does not guarantee optimality from the perspective of both sides but is in fact optimal from the perspective of the proposer. The algorithm works by iterating through a number of rounds of proposals. During each round, single men (men who are not engaged) propose to the woman that they have highest preference for and which they have not previously proposed. Each woman then considers all proposals and accepts the proposal of highest preference rejecting all others. At this stage a provisional engagement between a couple is formed. This can only be broken if the woman receives a a proposal from a man which she prefers in subsequent rounds.

4.1 Implementing the Gale-Shapley Algorithm in FLAME GPU

In order to handle the storage requirements of the stable marriage problem within FLAME GPU an extension to the existing framework has been made to allow agents to hold fixed length arrays within the internal memory of an agent. This effectively allows both a man and woman agent to hold a list of length n (the number of men and woman) corresponding to the rank of their preferred partners. Fixed length array agent memory variables have been implemented in such as way as to preserve the coherence of memory accesses for a group of agents simultaneously access the ith value from the array. In order to ensure coherent memory accesses array items are organised as a Structure of Arrays (SoA) rather than an Array of Structures(AoS). Figure 2 demonstrates how agent variables including (newly supported) arrays are assembled in memory. Array items for each agent are separated in memory by a stride of n and as such agent memory array access functions have been incorporated into the FLAME GPU code generation templates to handle accessing variables with the appropriate strides.


Array of Structures
Un-coalesced memory accesses
id^0 et^0 pl_0^0 pl_1^0 ... pl_N^0 id^1 et^1 pl_0^1 pl ... pl_N^1 ... id^N et^N pl_0^N pl_1^N ... pl_N^N
Structure of Arrays
Coalesced memory accesses
id^0 id^1 ... id^N et^0 et^1 ... et^N pl_0^0 pl_0^1 ... pl_0^N pl_1^0 pl_1^1 ... pl_1^N pl_N^0 pl_N^1 ... pl_N^N

```
typedef struct{
  int id;
  int engaged_to; //et
  float preference_list[N]; //pl
} agents [N];
```

```
typedef struct{
  int id[N];
  int engaged_to[N]; //et
  float preference_list [N*N]; //pl
} agents;
```

Fig. 2. Figure shows the memory layout differences of agent memory variables and agent memory variable arrays when using an Array of Structures (AoS) vs a Structure of Arrays (SoA) for data storage. Superscript indicates agent index, subscript indicates array index. When using a SoA agent variables and indices of arrays are grouped sequentially for each agent, allowing coalesced memory access.

The Gale-Shapley algorithm has been implemented within FLAME GPU as two agents (men and woman) with the following five agent functions which represent the behaviour of a single round of the algorithm;

1. *make_proposals*(): Only men who are not engaged propose to their highest ranked woman who they have not already proposed to. To select the next highest ranking woman, men select from an ordered list of indices representing the rank of woman they prefer. Proposal messages contain the index of the woman who is being proposed to and the index of the man who is proposing.
2. *check_proposals*(): All woman iterate all proposals to check if any are directed towards them (by considering the indices stored in the proposal mes-

sages). If they have received a proposal from any men in which they have a higher preference for than their current provisional partner (or if they do not have partner) then the details of this 'suitor' are saved and the woman becomes provisionally engaged.

3. *notify_suitors*(): An agent function filter determines woman who are provisionally engaged and allows these woman to notify suitors by sending a notification message containing her unique index and the index of the suitor that she has chosen.
4. *check_notifications*(): All men check all notification messages. If a man sees a notification that a woman has accepted his proposal then he becomes provisionally engaged. Men who were previously engaged but do not receive a notification have been replaced by a more desirable suitor and become free.
5. *check_resolved*() A global function condition checks the engagement state of each male agent. If all agents are engaged then a stable solution has been found and all male agents move into an 'engagement resolved' state where no further proposals will take place.

The *check_resolved* function plays an important role in determining the terminal state of the simulation. Once all Men are engaged the simulation ends (in at worst $n^2 + 2n - 2$ steps).

5 Experimental Results

The purpose of this section is to demonstrate the performance results of the two case studies presented within this paper. The results obtained give insight into the expected performance results which can be obtained by either migrating lattice based serial movement models or implementing assignment resolution within a FLAME GPU model. The two case studies have been implemented in FLAME GPU 1.3 for CUDA 6.0 (available online for free at http://www.flamegpu.com) which contains the new functionality to support array value agent memory variables. Both models are available within the updated framework to ensure results to be producible. Results have been obtained from an Intel Core i7-2600K Machine using an NVIDIA K40 GPU with CUDA 6.0. A fixed number of 64 threads per block is applied to all FLAME GPU kernels. FLAME GPU has a one to one mapping of agent functions and GPU kernals, additional GPU kernels provide background management of agent and message data for additional information readers are directed to previous publications on these methods [8, 9].

The sugarscape model is configured by randomly selecting 50 percent of available cells to contain sugarscape agents. The environment is also initialised with a random sugar distribution. Figure 3 (**A**) shows that over a million FLAME GPU agent cells are able to perform a single simulation step in just over 5ms, a single simulation step with over 16 million cells takes on average 76 ms. Each simulation step represents a single iteration of the movement conflict resolution processes described in section 3.1 and as such agent behaviour such as consumption (of sugar) and environmental grow-back is only performed within simulation steps where movement conflicts are fully resolved. Figure 3 (**B**) shows simultaneously; a breakdown of simulation step performance of 1048576 cells over the first 100 simulation steps and the total number of complete (fully resolved) simulation steps. The performance fluctuates between simulation steps in a fairly repeatable pattern as a simulation step where movement is resolved takes longer to process than a simulation step performing only movement resolution. Performance is also shown to improve as agents trend towards a fully resolved state as subsequent resolution steps are required to perform less work (e.g. simulation steps 2-5 in the figure). On average it can be observed that it takes roughly 5 iterations of the simulation to resolve all movement conflicts.

The stable marriage model has been initialised by assigning each man and woman with a randomly ordered preference for agents of the opposite sex. Within Figure 4 (**A**) performance has been measured by averaging simulation time over the total number of rounds required to reach a stable solution. Given the random preference of agents the average number of rounds to reach a stable solution is roughly 3.2 times the population size of a single sex. For example 1024 agents (of each sex) are resolved in 3166 rounds in a total of 19 seconds and 16384 agents are resolved in 49168 steps taking just over 2 hours. Figure 4 (**B**) shows simultaneously; the cumulative timing performance of each of the agent functions and the total number of engaged men for a population size of 1024 (men and woman). As the number of engaged men increases the performance of a simulation step improves. This is mainly a result of performance improvements in the Woman agents *check_proposals* function which has fewer proposal messages to iterate per round. It should be noted that as the number of engaged men increases the number of agents performing the *make_proposals* agent function decreases to the point that the level of parallelism is lower than the amount required to fully utilise the GPU. To avoid underutilising the GPU a potential solution would be to transfer the execution of this agent function to the CPU. The disadvantage of using the CPU for simulation would be that in this case the transfer cost would outweigh the underutilisation of the hardware until very small agent population sizes were observed. To provide optimal GPU and CPU sharing of agent functions for general cases would require dynamic load balancing, an area being explored for FLAME and FLAME GPU in future work.

6 Conclusion

Two cases studies have been demonstrated in which ‘conflict’ resolution is required in order to manage competition between agents. The first case study demonstrated a common problem which arises from the translation of a lattice based agent system from a serial to a parallel simulation framework. The second demonstrates a classic assignment problem. In each case the FLAME GPU framework has been shown to be a suitable parallel simulator to demonstrate implementation of an iterative method to resolve conflicts, allowing high performance agent interactions to be exploited to provide negotiation for a conflict resolution process. The case studies provide a building block for more complex

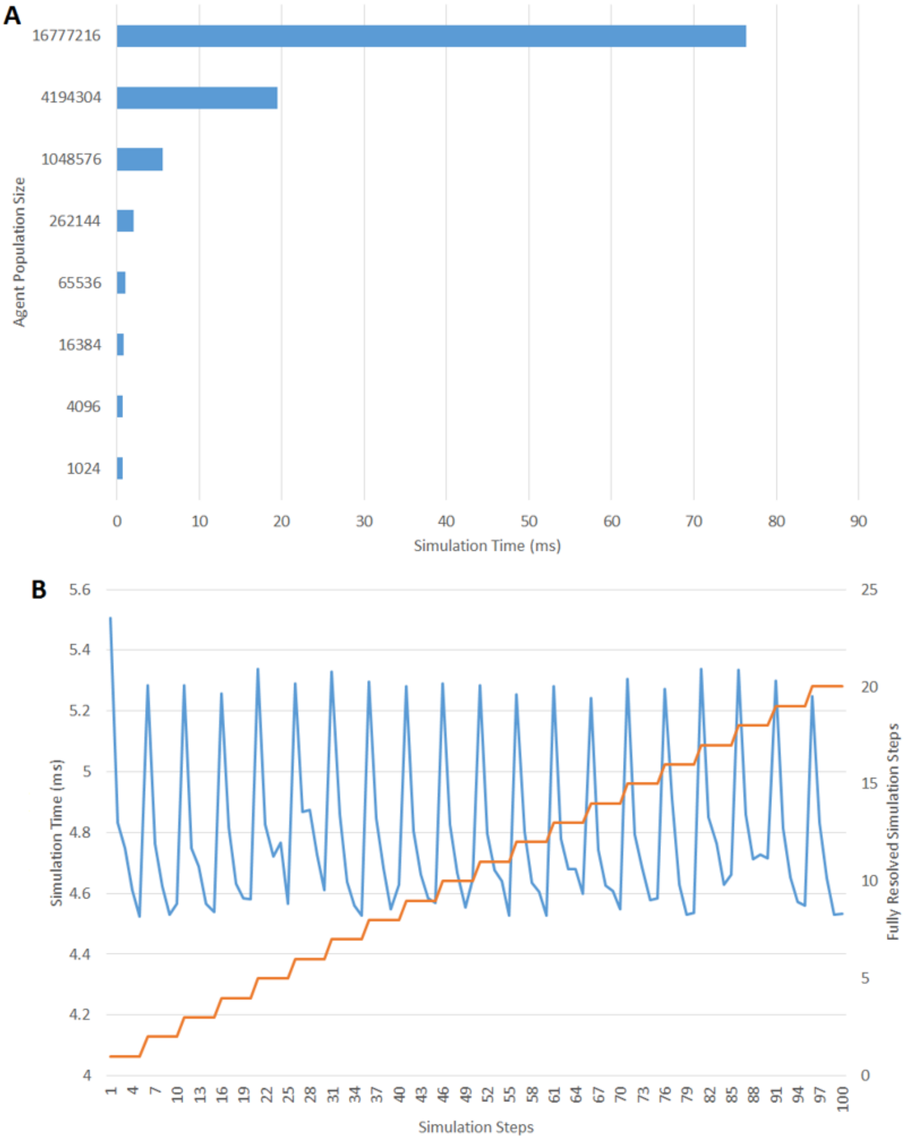


Fig. 3. Performance results of the sugarscape simulation showing; **A**: The performance scaling of the simulation by increasing the agent population size. Timings (in milliseconds) are measured over 100 simulation steps and then averaged. **B**: A breakdown of simulation performance over the first 100 steps of simulation. Primary (left hand) axis corresponds with the blue simulation timings (in milliseconds), the secondary (right hand) axis shows the number of fully resolved resolution steps (where all agents have resolved movement collision conflicts).

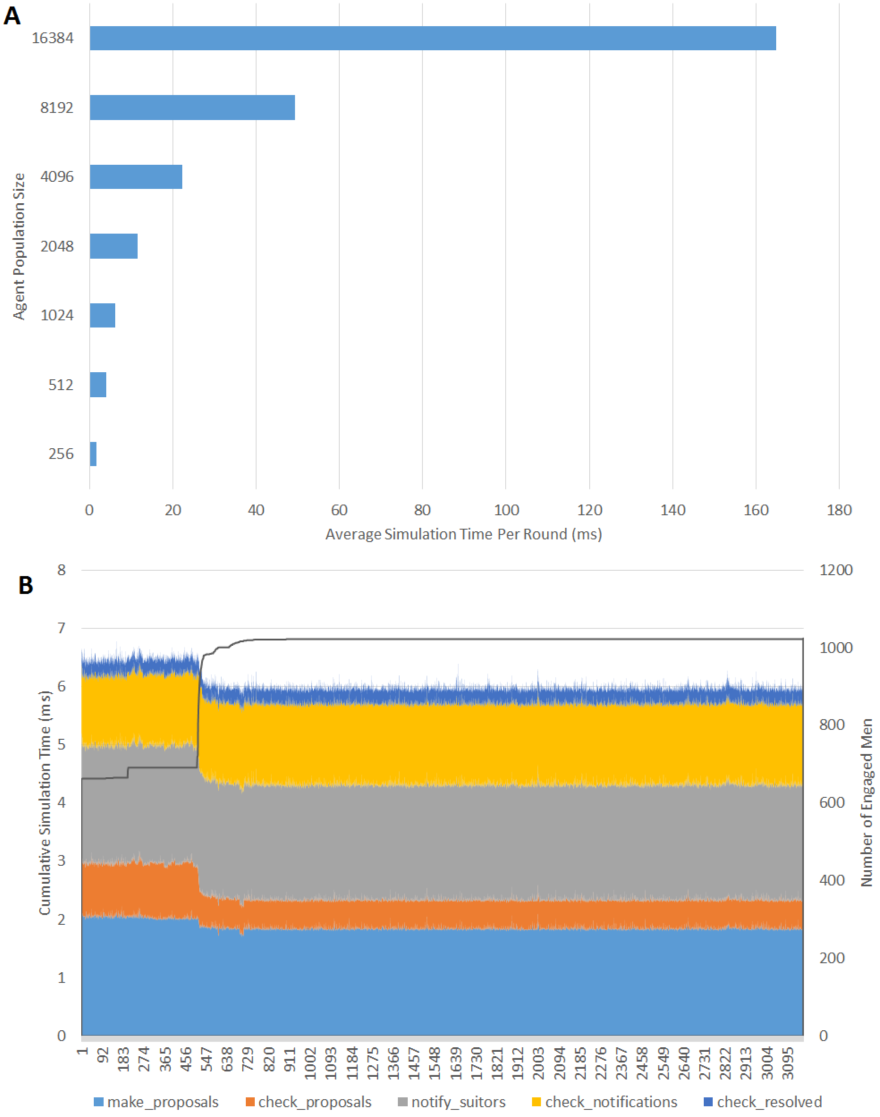


Fig. 4. Performance results of the Gale-Shapely simulation showing; **A**: The performance scaling of the simulation where the agent population size represents half the total number of agents (i.e. the population size of either the men or woman). Timings show the time required to simulate an average round of the Gale-Shapely algorithm. **B**: A breakdown of the cumulative timings (primary left hand axis) for each of the agent functions. The total number of engaged men is shown on the secondary (right hand) axis.

models to be implemented using FLAME GPU, allowing improved scaling and performance. Further more they act as a guide in the translation of serial models to parallel simulators where negotiation can be used to resolve movement or assignment conflicts. In future work the process of resolving competitive assignment will be applied to economic and biological models.

References

1. Aaby, B.G., Perumalla, K.S., Seal, S.K.: Efficient simulation of agent-based models on multi-gpu and multi-core clusters. In: proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. p. 29. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2010)
2. Crawford, V.P., Knoer, E.M.: Job matching with heterogeneous firms and workers. Econometrica 49(2), 437–450 (1981)
3. Epstein, J.M.: Growing artificial societies: social science from the bottom up. Brookings Institution Press (1996)
4. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. American Mathematical Monthly pp. 9–15 (1962)
5. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: Flame: Simulating large populations of agents on parallel hardware architectures. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1. pp. 1633–1636. AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2010), http://dl.acm.org/citation.cfm?id=1838206.1838517
6. Lynsenko, M., D'Souza, R.M.: A framework for megascale agent based model simulations on graphics processing units. Journal of Artificial Societies & Social Simulation 11(4) (2008)
7. Moser, D., Riener, A., Zia, K., Ferscha, A.: Comparing parallel simulation of social agents using cilk and opencl. In: Distributed Simulation and Real Time Applications (DS-RT), 2011 IEEE/ACM 15th International Symposium on. pp. 88–97. IEEE (2011)
8. Richmond, P.: FLAME GPU Technical Report and User Guide (CS-11-03). Tech. rep., Department of Computer Science, University of Sheffield (2011)
9. Richmond, P., Romano, D.: Template-driven agent-based modeling and simulation with cu da. GPU Computing Gems Emerald Edition p. 313 (2011)
10. Richmond, P., Walker, D., Coakley, S., Romano, D.: High performance cellular level agent-based simulation with flame for the gpu. Briefings in Bioinformatics (2010), http://bib.oxfordjournals.org/content/early/2010/02/01/bib.bbp073.abstract
11. Scarano, V., Cordasco, G., Chiara, R., Vicidomini, L.: D-mason: A short tutorial. In: Mey, D., Alexander, M., Bientinesi, P., Cannataro, M., Clauss, C., Costan, A., Kecskemeti, G., Morin, C., Ricci, L., Sahuquillo, J., Schulz, M., Scarano, V., Scott, S., Weidendorfer, J. (eds.) Euro-Par 2013: Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 8374, pp. 490–500. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54420-0_48