Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

TRANSLATION OF STRING-AND-PIN-BASED SHORTEST PATH CONSTRUCTION INTO DATA-SCALABLE AGENT-BASED COMPUTATIONAL MODELS

Yun-Ming Shih
Collin Gordon
Munehiro Fukuda

Division of Computing and Software Systems
University of Washington Bothell
18115 Campus Way NE
Bothell, WA 98011, USA

Jasper van de Ven
Christian Freksa

Bremen Spatial Cognition Center
University of Bremen
Enrique-Schmidt-Straße 5
Bremen, 28359, GERMANY

ABSTRACT

From the viewpoint of strong spatial cognition in graph problems, the shortest path can be identified in one physical action using strings and pins that respectively represent graph edges and vertices. By pulling a start and an end pin, we can construct a series of stretched strings as the shortest path. We use agent-based models (ABMs) to translate this action into computational representations. Assuming that a set of strings and pins are hung on a wall with a start pin, agents are disseminated downward to a destination as gravity forces. We implemented three models: a discrete-event, an asynchronous, and an aggregated agent dissemination on top of the MASS (multi-agent spatial simulation) library. To address large-scale network environments, we blended HDFS into MASS so that a graph data set is read over a cluster system in parallel. This paper presents these ABM implementations and performance measurements over a cluster system.

1 INTRODUCTION

Finding the shortest path is a typical problem encountered in traffic simulations and network-routing protocols. The most common algorithm used to address this is Dijkstra's algorithm. Despite its popularity, the algorithm has two drawbacks: (1) representing a graph structure as an adjacency matrix is not intuitive to code as iterating over the matrix involves repetitive traversals, backtracking, or both; and (2) it is not scalable nor is it easily parallelizable because of the nature of dynamic programming.

From the viewpoint of strong spatial cognition in graph problems, the shortest path construction can be solved in one physical action using strings and pins that respectively represent graph edges and vertices (Furbach et al. 2016; Freksa et al. 2016). All one needs is to hold a start pin and pull an end pin so that a series of stretched strings will give the shortest path. This method raises a question for algorithm designers: how can the string-and-pin-based approach be represented in code? One solution is to conduct a contiguous simulation that observes all the movements of pins affected by a human action. However, this would require a tremendous amount of computation as compared to Dijkstra's algorithm.

Instead, we use the concept of agent-based models (ABMs) to offset the computational load. If we consider hanging a set of strings and pins on a wall as holding a start pin, agents can diffuse as gravity forces from a start to an end pin along strings. Three implementations can be conceived. The first is a discrete-event simulation where each agent's movement on a string is scheduled as an event. The event occurs over an interval of time corresponding to the string length. The second is an asynchronous dissemination of agents along all the strings emanating from a start pin where agents with a shorter route overwrite the answer for the shortest path. The third is a special form of asynchronous agent dissemination where agents are aggregated for their computation and migration (named *doall* simulation in the following discussion). Our ABM approach is not only intuitive but also inherently parallelizable as many agents can run in parallel.

Despite that, its execution performance would not yet be comparable to Dijkstra's algorithm in small scale graphs, because of agent creation, migration, and termination overheads.

Given these backgrounds, we are pursuing the following two goals to model and run the string-and-pin-based shortest path construction in an acceptable time: (1) maintaining the ABM intuitive programmability and (2) targeting large-scale and dynamically-changing network environments. For this purpose, we have described and executed the above-mentioned three implementations of string-and-pin approach on top of the MASS (multi-agent spatial simulation) library (Chuang and Fukuda 2013), and have blended HDFS (http://hadoop.apache.org) into MASS so that a large scale graph dataset can be read into a cluster system in parallel. This paper presents our implementations and performance measurements.

The rest of this paper is organized as follows: Section 2 discusses about potential computational representations of string-and-ping-based shortest path construction and applies ABMs to the original algorithm; Section 3 explains the MASS parallelization of the ABM implementations; Section 4 analyzes MASS execution performance; and Section 5 concludes the outcomes.

2 COMPUTATIONAL REPRESENTATIONS OF STRING-AND-PIN-BASED SHORTEST PATH

This section considers four computational representations to run the string-and-pin-based shortest construction that was proposed from the strong spatial cognition's viewpoint in (Freksa et al. 2016). They are a direct computational representation and a gravity-based computational representation, where the latter can be even coded in three different agent migrations: a discrete-event, an asynchronous, and an aggregated (or *doall*) migration.

2.1 An Approach Using Strings and Pins

The original idea of strong spatial cognition in graph problems finds a shortest path using strings and pins, each respectively representing vertices and edges (see Figure 1). Given a start and an end point, the shortest path between them is identified by one physical action of fixing the start pin and pulling the end pin outward or just simply pulling both pins away until they can no longer move apart due to the fully stretched strings between them, in which case we can measure the length of the stretched strings as the shortest path.

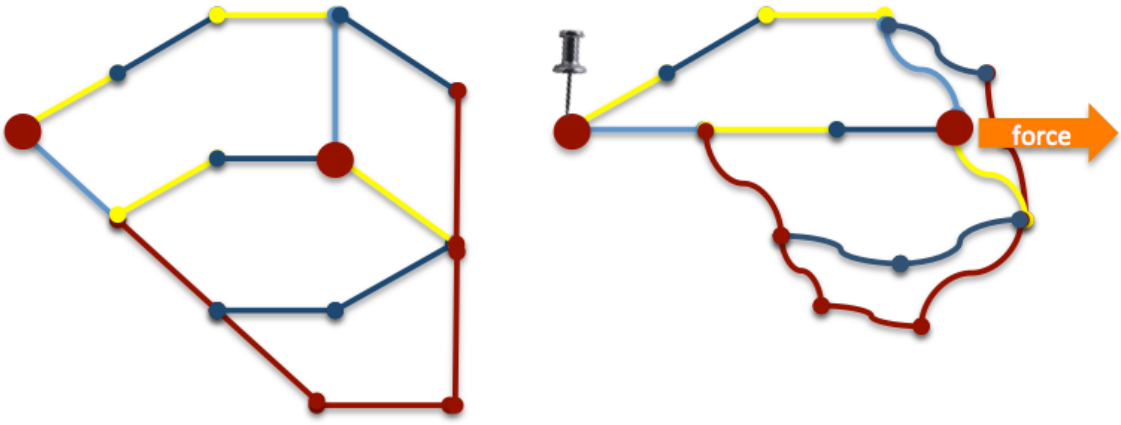


Figure 1: Shortest path construction using strings and pins.

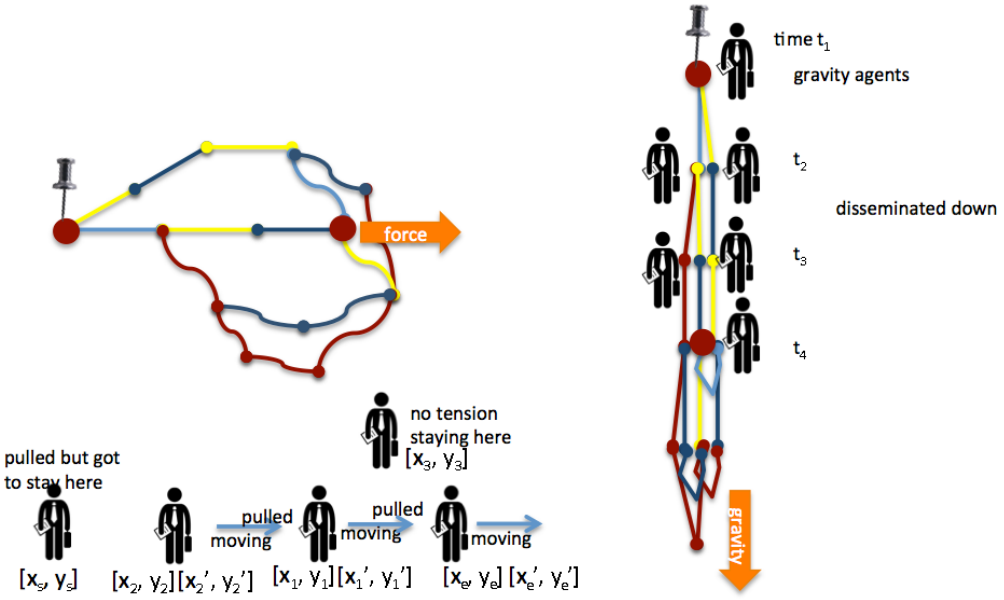
2.2 Direct Computational Representation

While the string-and-pin-based shortest path construction completes in one physical motion (i.e., pulling two pins away at once), its direct simulation would not run in O(1). As illustrated in Figure 2-(a), we

can represent pins as objects or agents that can move over a simulation space. There the agents maintain their communication channels, each representing a string that connects two pins mapped to the agents. The simulation gets started with moving the end pin (or the end agent) rightward gradually, so that the other pins (or agents) can move in response to the dynamic strength of strings pulled by the end pin. This results in contiguous but not discrete-event simulation. Although the computational complexity at a given simulation time will be $O(|V|^2)$ where $V = \#vertices$), a question is how precisely these pins (or agents) should move. Furthermore, regardless of this precision of time advances, the number of time ticks required for this direct simulation will be proportional to the length of the shortest path between a given set of start and end pins. Therefore, the total computational complexity is upper bound to $l|V|^2$ where $l =$ the distance between two mutually furthest pins.

2.3 Gravity-Based Computational Representation

Pulling two pins can be also realized by using gravity in the real world. This modification intends to fix the start pin on a wall and allow gravity to pull the other pins towards the ground – see Figure 2-(b). This effectively stretches the strings and reaches all the other pins including the endpoint. This physical motion can be simulated in discrete events where gravity forces disseminate from the start pin along its emanating strings down to all the other pins in parallel. In an actual implementation, we may represent gravity as agents, each scheduling its next movement as a discrete event whose timing is the same as the edge weight on this agent's direction. We conclude that the very first agent that has reached a given end point carries the shortest path information.



(a) Direct representation. (b) Gravity-based representation.

Figure 2: Two computational representations of string-and-pin approach.

If we remove the discrete-event concept from this gravity-based computational representation, it will simply behave as information diffusion (i.e., asynchronous gravity diffusion) from a given start to all the other pins. This asynchronous agent migration has advantages and disadvantages particularly when applied to a cluster system. An advantage is that we can get rid of overhead incurred by inter-agent synchronization and event retrieval among cluster nodes. On the other hand, a disadvantage is that a large number of gravity agents will be spawned and flooded chaotically over the underlying cluster system. To mitigate both agent synchronization and flooding overheads, we can aggregate agent computation and migration in a batch,

which we call *doall* migration. Yet, we have to note that, while information diffusion intends to flood the same information over an entire network, discrete-event simulation stops the computation as soon as the first gravity agent arrives at a given destination.

The worst case scenario is having to examine all potential paths regardless of the approach we use. When an agent encounters multiple edges at the current vertex, it must spawn the same number of children as $\#edges - 1$ where it will choose one of the emanating edges for itself to move along. Therefore, we expect that the computational complexity is $O(\#agents)$ where $\#agents$ is the total number of agents spawned.

2.4 Related Work

Since shortest path algorithms are key to route planning in scalable road networks, various improvements have been made so far to perform faster than Dijkstra's algorithm. For instance, highway hierarchies represent a hierarchical structure of streets, boulevards, and highways, which are more realistic and capable of bypassing unnecessary route computation. More generalized are contraction hierarchies that iteratively contract the least important vertex by replacing paths going it through with a shortcut (Geisberger et al. 2012). Time-dependent contraction hierarchies have addressed dynamic changes in traffic. However, contraction hierarchies introduce many additional edges and thus require more memory space. An MPI-based parallelization facilitated distributed memory to contraction hierarchies and accelerated distributed query processing (Kieritz et al. 2010). Although we use agent-based parallelization, our focus is placed on computational representations on strong spatial cognition but not necessarily on pursuit of fast route-planning techniques. In the next section, we will show how the MASS library implements each of the approaches proposed for the string-and-pin-based shortest path algorithm.

3 AGENT-BASED PARALLELIZATION

To describe and parallelize the shortest path search with agents, we use the MASS (multi-agent spatial simulation) library we have implemented in Java, C++, and CUDA. The following discussions focus on the Java version.

3.1 MASS Library

The MASS library distinguishes two classes of objects: *Places* and *Agents*. *Places* are objects in a multi-dimensional array that is mapped over a cluster system. Each array element is called *place*, accessed with a cluster-independent logical index, and capable of exchanging information with others. *Agents* is a group of mobile *agent* objects capable of carrying its internal state from one place to another, communicating with their current place, and using their current place to communicate with other nearby agents.

As illustrated in Figure 3, the MASS library uses a collection of multithread processes, each spawned from JSCH (http://www.jcraft.com/jsch) onto a different cluster node to manage a portion of places and agents residing there. It hides all platform-dependent, parallel-programming constructs. Instead, model designers can perform parallel execution of places and agents, using the method calls summarized in Table 1.

Table 1: A list of MASS methods.

Methods	Specifications
Places.callAll(function):	invokes each place's function in parallel.
Places.exchangeAll(function):	allows each place to collect data from its neighbors' function.
Places.exchangeBoundary( ):	exchanges boundary information among neighboring places.
Agents.callAll(function):	invokes each agent's function in parallel.
Agents.manageAll( ):	performs agent spawning, terminating, and moving operations.

In addition to parallel places/agents execution, the MASS library also facilitates parallel I/O for NetCDF (https://www.unidata.ucar.edu/software/netcdf) and text files. Each place can open an identical file if not yet

opened and thereafter read/write only its corresponding file data, using the built-in functions: *Place.open(fn)*, *read(fd)*, *write(fd, bytes)*, and *close(fd)*, where *fn* is a file name and *fd* is a file descriptor. The MASS parallel I/O (Shih 2018) partially uses HDFS to duplicate a given file as many times as the number of cluster nodes, thus each maintaining at least one replica. Once the file has been automatically made available at each cluster node's */tmp* directory, the MASS parallel I/O allows all places to read their own file data in parallel. When places simultaneously invoke *write()* and *close()*, their data chunks are temporarily stored in */tmp*, thereafter collected back to the master node, and finally assembled as one consistent file.

3.2 Discrete-Event Model

Using the MASS library, we have translated the string-and-pin-based shortest path construction into a discrete-event model of agent migration. Figure 4 describes the agent behavior composed of two functions: *onArrival( )* and *departure( )*. Upon an arrival at a new vertex, the agent deposits its footprint to the current vertex, (showing where it came from) and spawns new children. Thereafter, all the agents including the parent and children migrate to their next vertex.

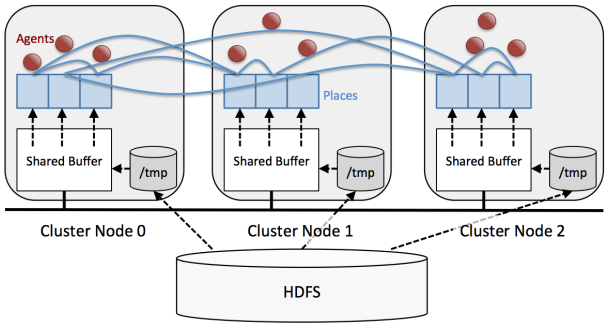


Figure 3: The MASS (multi-agent spatial simulation) architecture.

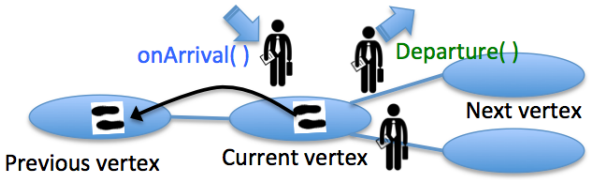


Figure 4: Agent behavior on arrival at and departure from a vertex.

Figure 5 shows a code snippet of the *main( )* function that starts MASS (line 3), creates a street map (lines 4-5) and populates the very first gravity agent (line 7). The *Argument(10, 0, 0)* in line 6 is passed to the agent's constructor, denoting *place[10]*, *place[0]*, and *time 0* as its destination pin, start pin, and first migration time respectively. Thereafter, the execution goes into a *for*-loop in line 9 that repetitively calls gravity agents' *departure( )* and *onArrival( )* functions (lines 10 and 13). When an agent has reached the end pin, it returns -1 from *onArrival( )* (line 14), which finishes the simulation (line 18).

Figure 6 describes the gravity agent's behavior. When initialized, each agent receives the final destination, the next place to hop, and the time to go (line3). Upon an *onArrival( )* call (line 11), an agent deposits its footprint to the current vertex and spawns new children (lines 16 and 19). In *departure( )* (line 5), when the simulation time has come to its departure time (line 7), the agent migrates to the next vertex (line 9).

3.3 Asynchronous Model

The discrete-event model shown above can be converted into an asynchronous model by allowing gravity agents to migrate over a given map freely without waiting for the next event to be fired. However, the first agent that arrived at a given destination is no longer guaranteed to find the shortest path from the start pin. Therefore, as shown in Figure 7, *main( )* must collect all the agents that have finished their travel (i.e., #agents should be 0 or *gravity.nAgents( ) == 0*). Figure 8 shows the changes to the agent behavior. Agents no longer have to be concerned with timing (lines 6-7). Instead, each *place.footprint* should be used to record the current travel cost from the start pin (line 17). Only agents whose travel cost is shorter than the current footprint can continue migrating to the next place, otherwise they will be terminated (line 13).

```
1 public class DiscreteEventModel {
2   public static void main(String[] args) {
3     MASS.init( );
4     Places map = new Places(1, Map, nVertices);
5     map.callAll(Map.init_);
6     Argument arg = new Argument(10, 0, 0);
7     Agents ag = new Agents(2, Gravity, arg, map,
            1);
8     int nextEvent = −1;
9     for (int t = 0; ; time = nextEvent) {
10      ag.callAll(Gravity.departure_; time);
11      ag.manageAll( );
12      Integer[] allEvents
13        = ag.callAll(Gravity.onArrival_);
14      if ((nextEvent = minInt(allEvents)) == −1)
15        break;
16      ag.manageAll( );
17    }
18    MASS.finish( );
19} }
```

Figure 5: Main in the discrete event model.

```
1 public class Gravity extends Agent {
2   public Gravity(Argument arg) {
3     dest = arg.dest; next = arg.next; dept = arg.
          dept;
4   }
5   public void departure(Integer time) {
6     int currTime = time.intValue( );
7     if (currTime == dept) { // my departure time?
8       prevVertex = place.index;
9       migrate(nextNode);
10  } }
11  public Integer onArrival(Integer time) {
12    if (place.footprint == −1 || place.index ==
          dest) {
13      dept = −1
14      kill( );
15    } else {
16      place.footprint = prevVertex;
17      Argument[] args
18        = createArg(place.neighbors−1, dept);
19      spawn(args);
20      dept = time + place.neighbors[0].weight;
21    }
22    return dept;
23} }
```

Figure 6: Agents in the discrete event model.

3.4 DoAll Model

The *doall* model is a special form of asynchronous model where we aggregate agent computation and migration. Figure 9 shows the aggregation in *main()*. The *doAll()* function receives a list of agent methods (line 10), each called simultaneously as in *callAll()*. It differs from *callAll()* in that *doAll()* does not synchronize among cluster nodes. Furthermore, each time *doAll()* completes a given agent method, it automatically invokes *manageAll()* that handles agent creation, termination, and migration. The advantages of *doall* are two-fold: (1) mitigating the number of cluster-wide synchronizations and (2) controlling chaotic agent flooding over a graph.

3.5 Logical Network Construction Using HDFS

In the previous sub-sections, we have focused on only the descriptions of agent behavior. Obviously before agents start their travels, *Places* must construct a street map over a cluster system. This corresponds to *map.callAll(Map.init_)* in Figure 5's line 5. We prepare a text file that has described a street map in an adjacency matrix where each row lists distances from a given vertex to the others. When each place calls *init()* as shown in Figure 10, it is guaranteed to simultaneously capture only the corresponding row of the matrix file (line 10) so that it can internally maintain its neighbors and their distances (see Figure 11).

The MASS parallel file I/O addresses two typical problems in allocating spatial data over a cluster system. One is to use HDFS for the purpose of preventing the master node or NFS from behaving as a bottleneck of data distribution. The other is to allow a user to generate a random graph in a file in advance rather than create a graph in a simulation program on the fly. The latter case is familiar to many graph applications that need to generate a different random graph repeatedly to find a particular graph relation such as a network motif (Andersen et al. 2016). The problem is how to generate an identical graph between sequential and parallel settings. A cluster system must maintain the same sequence of random number

```
9  for (int t = 0; ; time = nextEvent) {
10   ...
11   ...
12   ...
13   ...
14   if ((nexstEvent = minInt(allEvents)) == −1)
```

should be replaced with:

```
9  while (gravity.nAgents( ) > 0) {
```

Figure 7: Main in the asynchronous model.

```
6    int currTime = time.intValue( );
7    if (currTime == dept) { // my departure time?
```

are no longer needed. The footprint should record the shortest cost (i.e., ***dept***) from the start pin:

```
12  public Integer onArrival(Integer time) {
13    if(place.footprint<=dept || place.index==dest
          ){
14      dept = −1
15      kill( );
16    } else {
17      place.footprint =
            dept; // instead of prevVertex
```

Figure 8: Agents in the asynchronous model.

```
10      ag.callAll(Gravity.departure_; time);
11      ag.manageAll( );
12      Integer[] allEvents
13        = ag.callAll(Gravity.onArrival_);
14      ...
15      ...
16      ag.manageAll( );
```

should be replaced with

```
10      ag.doAll( new int[] {Gravity.departure_, Gravity.onArrival_} );
```

Figure 9: Main in the doall model.

generations over all nodes as the sequential version generates. A pair of vertices must be synchronized if they create a bidirectional edge, which results in cluster-node communication if these two vertices reside on a different node. Therefore, it is relatively easy to generate a graph file first and thereafter to let each cluster node read a different portion of the file, using the MASS parallel I/O.

4 EXECUTION PERFORMANCE

Given these three ABMs to represent the string-and-pin-based shortest path construction, we have evaluated their execution performance, using the University of Washington Bothell's shared Linux cluster: 16 Dell Optiplex 710 desktops, each with an Intel i7-3770 Quad-Core CPU at 3.40 GHz and 16 GB RAM. Our evaluation covers (1) our ABMs' parallel execution over the cluster system, (2) the effect of multithreaded execution, (3) the performance of MASS parallel I/O, and (4) execution overheads in discrete-event, asynchronous, and *doall* agent migrations. Regarding evaluation item 4, discrete-event simulation finishes its computation when the very first gravity agent reached the destination, whereas both asynchronous and *doall* agent migrations must wait for all agents to get terminated (i.e., until all nodes are thoroughly examined).

4.1 Parallel Execution over a Cluster System

Figure 12 presents the computation time of shortest path search in the discrete-event model. The model did not show any CPU scalability with up to eight cluster nodes while the graph could be stored in a single

```
1 public class Vertex extends Place {
2   public int[] neighbors = null;
3   public int[] distances = null;
4   public int footprint = −1; // cost from the
        source
5   public Place(Integer arg) {
6     footprint = arg;
7   }
8   public void init(String filename) {
9     int f = open(filename, 0);
10    Scanner data = new Scanner(new String(read(
          f));
11    Scanner values = new Scanner(data.nextLine()
          );
12    for (int i = 0; values.hasNextInt(); i++) {
13      int distance = values.nextInt();
14      if (distance >= 0) {
15            neighborsList.add(new Integer(i));
16            distances.add(distance);
17 } } } }
```

Figure 10: Places initialization.

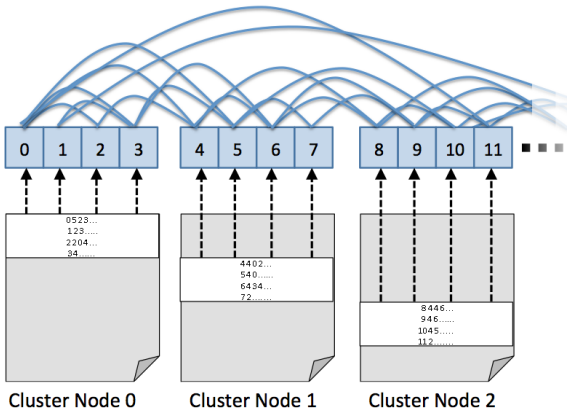


Figure 11: A logical network construction with MASS Places.

computing node (i.e., while the graph size stays below 2,048 vertices). We can infer that this is because of agent migration overheads. However, as we further increased a graph size, its entire graph could no longer fit to a smaller number of cluster nodes. Shortest path search over a graph of 16,382 vertices was able to complete only with eight computing nodes. This in turn means that our ABM demonstrated good memory scalability.

Figure 13 measured the entire execution of discrete-event model including graph creation over the cluster system. No CPU scalability was observed with up to 1,024 vertices. Beyond that number, the entire execution performed better with more cluster nodes. Similar to the computation-only performance, the entire execution shows better memory scalability. These results support the efficiency of the MASS parallel file I/O.

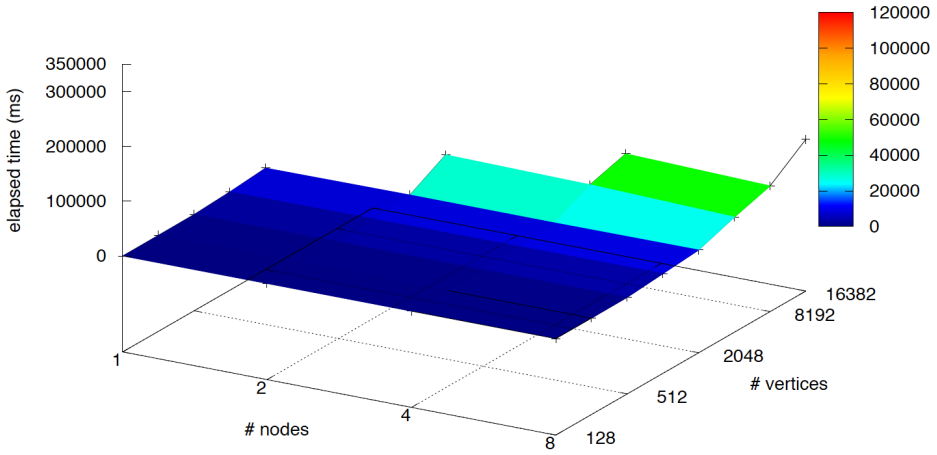


Figure 12: Computation time of discrete-event simulation.

Table 2 compares execution performance between Dijkstra's algorithm and eight-way parallel discrete-event simulation when they create a 2,048-node network and compute the shortest path. The parallel simulation can create an on-the-fly network so quickly, while performing much slower, even using eight

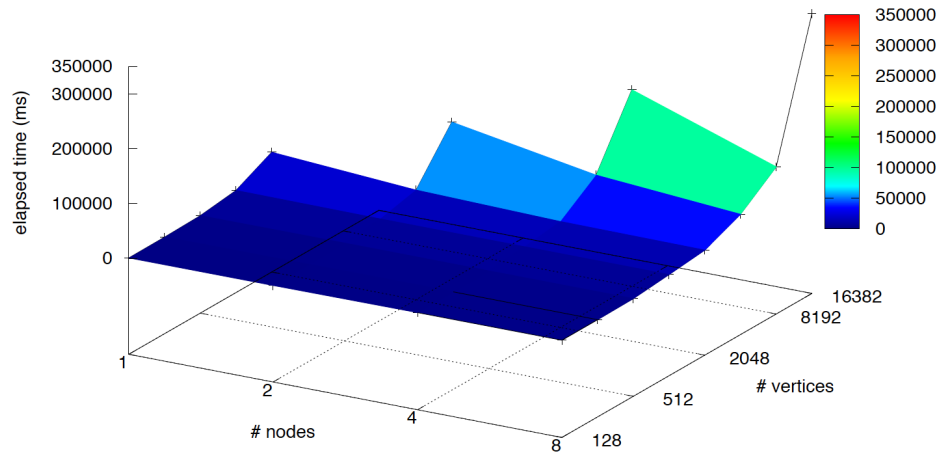


Figure 13: Total execution of discrete-event simulation including graph creation.

computing nodes, than Dijkstra's algorithm. Beyond 2,048 nodes, Dijkstra's sequential execution consumes the entire memory space in one computing nodes, and therefore parallelization using distributed memory enables scalable shortest path computation.

Table 2: Sequential Dijkstra's algorithm versus eight-way parallel simulation (milliseconds).

Measures for a 2048-node network	Dijkstra's algorithm	8-way parallelization
Network creation from file	2,430	2,427
On-the-fly network creation	25,942	1,201
Computation time	19	3,435

4.2 Multithreaded Execution

Since each cluster node includes four CPU cores, we also evaluated the effect of multithreaded execution. Table 3 compares single, two-, and four-threaded executions of the discrete-event model. No performance improvements were observed. Since agents disseminate over a graph like a wave, they move to and reside on closer or even identical vertices (or places), multiple threads would compete for accessing the same memory space and thus increase cache thrashing.

4.3 Graph Creation Performance

In addition to the MASS parallel file I/O's CPU and memory scalability, we also measured its competence in graph creation. Figure 14 determines the thresholds for the MASS parallel I/O to outperform on-the-fly random graph creation. Although it could never perform faster than on-the-fly creation in single execution, the MASS parallel I/O worked faster on a graph beyond 1,024 vertices, when using two through to eight computing nodes.

Table 3: Multithreaded execution of discrete-event simulation (milliseconds).

# vertices	1 thread	2 threads	4 threads
128	692	715	701
2,048	43,586	43,264	43,076

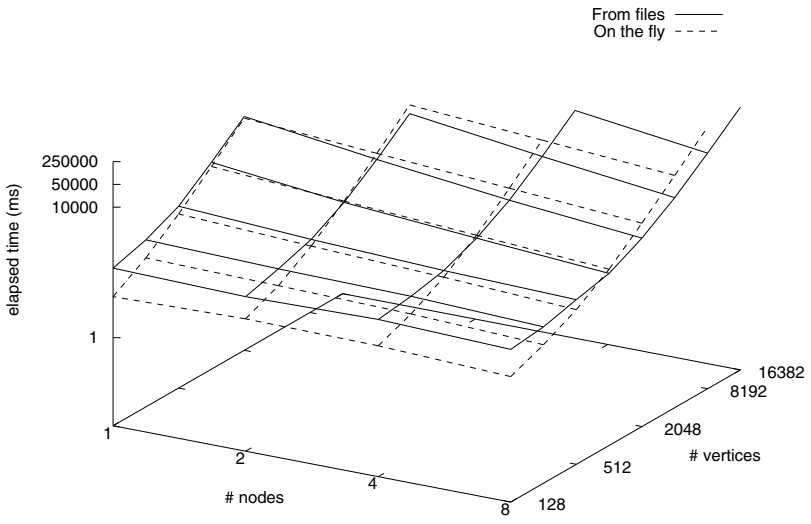


Figure 14: Graph creation from file data versus on the fly.

4.4 Overheads in Discrete-Event, Asynchronous, and DoAll Migrations

Our performance evaluation revealed that the discrete-event model always performed better than asynchronous and *doall* models. Table 4 compares their execution performance when using eight cluster nodes. As scaling up the graph size, the difference in their execution performance even grew larger. To make matters worse, neither asynchronous or *doall* model could complete its computation in five minutes beyond 2,048 vertices.

Table 4: Computation time (in milliseconds) of discrete-event, asynchronous, and *doall* models.

# vertices	Discrete event	Asynchronous	Doall
128	803	1,138	1,216
2,048	12,438	60,569	59,113

Figures 15 and 16 give us hints about why they performed much slower than the discrete-event model. Figure 15 counts the total number of simulation cycles, each migrating all agents from one vertex to another during a simulation. For a smaller graph, the discrete-event model could walk an agent to the final destination in less cycles than the other two models. On the other hand, for a larger graph, the asynchronous and *doall* models completed their simulation in a fewer cycles. The problem is not the number of simulation cycles but the number of agents.

In Figure 16, solid lines observe the growth of cumulative number of agents spawned during a simulation. Both asynchronous and *doall* models exponentially increased the number of agents to walk over a graph. The dotted lines indicate the number of agents to walk per each cycle (i.e., $cumulative\ \#agents/\#cycles$). We can infer that this number in both asynchronous and *doall* models would jump up beyond 800,000 agents when searching for 4,096 vertices. This is the reason behind various run-time errors including using up memory, causing disk thrashing, and never completing computation.

In summary, although the asynchronous model can walk agents freely over a graph with less synchronizations and in a fewer simulation cycles, it causes the explosive growth of agent population. The *doall* model can slightly mitigate agent-spawning overheads, however this is not sufficient enough to yield better performance than the discrete-event model.

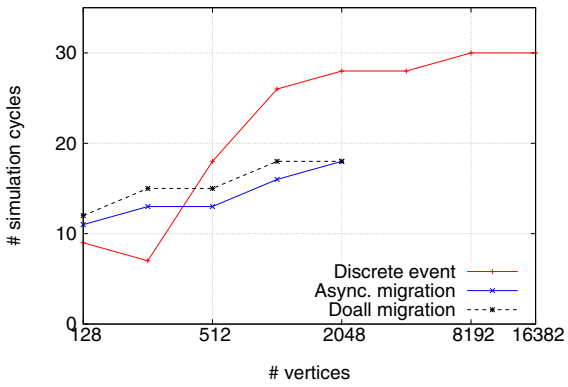


Figure 15: Total number of simulation cycles needed in discrete, asynchronous, and *doall* simulation.

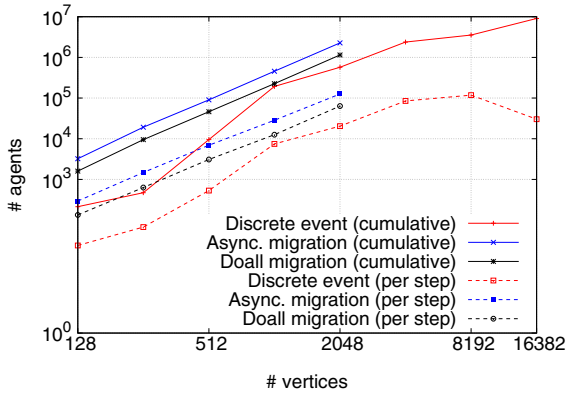


Figure 16: The number of agents created in discrete, asynchronous, and *doall* simulation.

5 CONCLUSIONS

We examined ABMs to represent the string-and-pin-based shortest path construction that can complete in one human action by pulling both start and end pins apart and stretching strings between them. Our models walk agents over a graph as gravity forces and distinguishes three different implementations: discrete-event, asynchronous, and *doall* agent migration models. Our discussions and experiments have clarified that, while the discrete-event model needs logical-time management, thus yielding more complicated programmability than the other two, it demonstrates the best execution performance by pacing the agent dissemination and keeping its lower population. We also confirmed the efficiency of the MASS parallel file I/O that manages all */tmp* disks in a cluster system using HDFS. Our future work includes the following two tasks:

1. *Agent population control:* This new feature temporarily freezes an excessive number of agents, using Java serialization and resumes their execution when the population gets decreased sufficiently. We will apply it to both asynchronous and *doall* agent migration to see their improvements in both CPU and space scalability.
2. *GPU computing:* As we have developed MASS CUDA, we will port our ABMs to this platform and compare their execution performance with the C++ native execution of Dijkstra's algorithm.

Finally, the MASS library and its sample applications including our shortest-path search program are all available at the following website: http://depts.washington.edu/dslab/MASS.

ACKNOWLEDGMENTS

We acknowledge funding through the DFG CRC 1320 EASE - Everyday Activity Science and Engineering (subproject P3 - Spatial reasoning in everyday activity).

REFERENCES

Andersen, A., W. Kim, and M. Fukuda. 2016. "MASS-Based NemoProfile Construction for an Efficient Network Motif Search". In *Proceedings of the 2016 IEEE International Conference on Big Data and Cloud Computing in Bioinformatics*, 601–606. Atlanta, GA: IEEE.

Chuang, T., and M. Fukuda. 2013. "A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems". In *Proceedings of the 2013 International Conference on Computational Science and Engineering*, 143–150. Sydney, Australia: IEEE.

Freksa, C., A. Olteteanu, A. L. Ali, T. Barkowsky, J. van de Ven, F. Dylla, and Z. Falomir. 2016. "Towards Spatial Reasoning with Strings and Pins". *Advances in Cognitive Systems* 4:1–15.

Furbach, U., F. Furbach, and C. Freksa. 2016. "Relating Strong Spatial Cognition to Symbolic Problem Solving – An Example". *arXiv preprint arXiv:1606.04397*.

Geisberger, R., P. Sanders, D. Schultes, and C. Vetter. 2012. "Exact Routing in Large Road Networks using Contraction Hierarchies". *Transportation Science* 46(3):388–404.

Kieritz, T., D. Luxen, P. Sanders, and C. Vetter. 2010. "Distributed Time-Dependent Contraction Hierarchies". In *Proceedings of the 2010 International Conference on Experimental Algorithms*, 83–93. Naples, Italy: Springer-Verlag.

Shih, Y.-M. 2018. "MASS HDFS: Multi-Agent Spatial Simulation Hadoop Distributed File System". MS Capstone Final Report, MS in Computer Science & Software Engineering, Univ. of Washington Bothell.

AUTHOR BIOGRAPHIES

YUN-MING SHIH holds an MS in Computer Science and Software Engineering from University of Washington Bothell. Her graduate research centered around design and implementation of parallel file I/O to support agent-based models. She is working as a software engineer at Security Invocation. Her email address is shihy4@uw.edu.

COLLIN CORDON holds an MS in Computer Science and Software Engineering from University of Washington Bothell. During his graduate study, he conducted research on agent-based machine learning as Research Assistant. He is working as a software engineer at JBT FoodTech. His email address is colntrev@uw.edu.

MUNEHIRO FUKUDA is Professor of Computing Software and Systems Division at University of Washington Bothell. He holds a Ph.D. in Information and Computer Science from University of California, Irvine. His research interests include multi-agent systems, agent-based simulation, and parallel computing. His email address is mfukuda@uw.edu.

JASPER VAN DE VEN is Researcher of Cognitive Systems Group/Bremen Spatial Cognition Center at University of Bremen. He holds a Ph.D. in Informatics from University of Bremen. His research focuses on methods to apply ambient intelligence and qualitative spatial and temporal reasoning. His email address is jasper.vandeven@uni-bremen.de.

CHRISTIAN FREKSA is Professor of Informatics at University of Bremen and Director of Bremen Spatial Cognition Center. He holds a Ph.D. in Artificial Intelligence from University of California, Berkeley. His research interests focus on knowledge representation, cognitive science, reasoning, and spatial cognition. His email address is freksa@uni-bremen.de.