Agent Programmability Enhancement for Rambling over a Scientific Dataset

Matthew Sell[0000−0002−5369−7599] and Munehiro Fukuda[000−0001−7285−2569]

Computing and Software Systems, University of Washington Bothell WA 98011, USA
{mrsell,mfukuda}@uw.edu

Abstract. Agent-based modeling (ABM), while originally intended for micro-simulation of individual entities, (i.e., agents), has been adopted to operations research as biologically-inspired algorithms including ant colonial optimization and grasshopper optimization algorithm. Observing their successful use in traveling salesman problem and K-means clustering, we promote this trend in ABM to distributed data analysis. Our approach is to populate reactive agents on a distributed, structured dataset and to have them discover the dataset's attributes (e.g., the shortest routes and the best cluster centroids) through agent migration and interaction. We implemented this agent-based approach with the multi-agent spatial simulation (MASS) library and identified programming features for agents to best achieve data discovery. Of importance is ease of describing when and how to have agents traverse a graph, ramble over an array, and share the on-going computational states. We have responded to this question with two agent-descriptivity enhancements: (1) event-driven agent behavioral execution and (2) direct inter-agent broadcast. The former automatically schedules agent actions before and after agent migration, whereas the latter informs all agents of up-to-date global information, (e.g., the best slate of centroids so far). This paper presents our design, implementation, and evaluation of these two agent descriptivity enhancements.

Keywords: agent-based modeling · programmability · distributed data analysis.

1 Introduction

Agent-based modeling (ABM) gained its popularity to observe an emergent collective group behavior of many agents by simulating and gathering their microscopic interactions. Nowadays ABM is being applied to not only micro-simulation but also operations research. This extension is known as biologically-inspired algorithms such as ant colonial optimization (ACO) [1] and grasshopper optimization algorithm (GOA) [13], heuristically effective to compute NP-hard problems: traveling salesman problem (TSP) and K-means clustering problems. Our focus is to promote this trend in agent-based computation to broader and scalable data-science problems. Obviously this agent-based approach is not applicable to all problems nor always competitive to major tools in big-data computing including MapReduce, Spark, and Storm[1]. To seek for killer applications, we should first point out that biologically-inspired algorithms walk agents over a given dataset for discovering attributes of the data space, (e.g., the shortest routes and the best cluster centroids). This is distinguished from data streaming in most big-data tools that examine every single data item to compute statistics of the entire dataset, (e.g., sum and average). Based on this observation, we have aimed at facilitating agent-based data discovery by constructing a big data structure over distributed memory and populating agents to ramble over the dataset in parallel.

For the feasibility study, we used our multi-agent spatial simulation (MASS) library where *main()* serves as a data-analyzing scenario while *Places* and *Agents*, two major classes of the MASS library, construct a distributed dataset and populate agents on it respectively. Through our former work [4,7,15], the MASS library demonstrated two programming advantages: (1) enforcing weak consistency among agent computation and (2) allowing users to code agents from the vehicle driver's viewpoint. The former guarantees an automatic barrier synchronization among all agents for their state transitions and migrations each time *main()* gets control back from a parallel function invocation on *Agents*. The latter intuitively navigates agents along graph edges or diffuses them to adjacent array elements. However, we also encountered two challenges in describing agent behaviors where users need (1) to give agents step-by-step actions to take – more specifically what to do before and after their migration to a new vertex or an array element and (2) to emulate inter-agent message broadcast with *main()* that passes arguments to *Agents* functions. This emulation makes it tedious for agents to share on-going computational states, e.g. the shortest route so far in ACO-based TSP or the best centroids in GOA-based K-means.

These challenges are big burdens to non-computing specialists who are interested in simply dispatching agents into their datasets but not coding every single agent behavior. Therefore, we are addressing them by automating parallel invocation of *Agents* functions and facilitating inter-agent message broadcast. This paper presents our design, implementation, and evaluation of these two agent descriptivity enhancements. The rest of this paper consists of the following sections: Section 2 details the current challenges in agent descriptivity for pursuing agent-based data discovery; Section 3 presents our implementation of event-driven agent behavioral execution and direct inter-agent message broadcast; Section 4 evaluates our implementation both from programmability and execution performance; and Section 5 concludes our discussions.

2 Challenges in Applying ABM to Distributed Data Analysis

This section emphasizes ABM's superiority over conventional approach to structured data analysis, summarizes our previous endeavor with the MASS library,

[1] http://{hadoop,spark,storm}.apache.org

and clarifies the challenges in agent descriptivity for ABM to be smoothly adopted to distributed data discovery.

2.1 Conventional Approach to Analysis of Structured Dataset

Most big-data computing tools benefit statistical analysis of data continuously streamed as flat texts from social, business, and IoT environments. To extend their practicability to analysis of structured datasets, they provide users with additional services to partition, flatten, and stream multi-dimensional NetCDF data[2] into MapReduce with SciHadoop [2]; to describe and process graphs with GraphX[3] on top of Spark; and to schedule repetitive MapReduce processing over a structured dataset with Tez[4]. Furthermore, some textbooks [8,11] introduce how to process a graph problem with MapReduce and Spark in an edge-oriented approach that reads a list of graph edges, narrows down candidate edges by examining their connectivity, and eventually identifies sub-graphs, (e.g., triangles) in the graph [5]. While these big-data computing tools offered simple programming frameworks for parallel computing and interpretive execution environments, their extension to structured datasets does not always achieve the best programmability and execution performance of data discovery [4]. This is due to their nature of data streaming. For instance, data streaming does not allow data to stay in memory and thus cannot analyze different data relationships through the same streaming operation. Repetitive MapReduce invocations or many Spark transformations would transfer back and forth or swap in and out data between disk and memory, which slows down the execution speed.

As observed above, the key is allowing different analyses to be applied to an in-memory structured dataset. GraphLab followed by Turi [9] maintains a graph structure on Amazon EC2, has each graph vertex access its neighboring vertices' state, and prepares a variety of graph functions in Python. NetCDF4-Python[5] is a Python interface to the NetCDF parallel I/O that is implemented on top of HDF5[6] and MPI-I/O, so that a NetCDF dataset is loaded over a cluster system and accessed in parallel by Python programs. Although these tools handle graphs or multi-dimensional arrays in Python, (i.e., a high-level interpretive environment), they still focus on statistical analysis of graphs and arrays such as vertices/edges counting, summation, and max/min values as well as major machine-learning functions that can be done with data streaming, too.

In contrast to them, our agent-based approach loads a structured dataset in parallel, maintains the structure in memory, dispatches reactive agents onto the dataset, and discovers user-designated data attributes in an emergent group behavior among these agents.

[2] https://www.unidata.ucar.edu/software/netcdf/
[3] http://spark.apache.org/graphx
[4] http://tez.apache.org
[5] https://github.com/Unidata/netcdf4-python
[6] https://support.hdfgroup.org/HDF5/

2.2 Previous Work with MASS

We have evaluated the feasibility of and challenges in agent-based data discovery, using the MASS library. MASS instantiates a multi-dimensional distributed array with *Places* and initializes it with an input data file in parallel [14]. For a graph construction, we create a 1D array of vertices and initialize the vertices with a file that contains an adjacency list. MASS populates agents on a given *Places* object from the *Agents* class. MASS performs parallel function call to all array elements and agents with *Places.callAll(fid)* and *Agents.callAll(fid)*; data diffusion across array elements with *Places.exchangeAll()*; and agent migration, termination, and additional population with *Agents.manageAll()*. Note that these agent behaviors are scheduled as *migrate()*, *kill()*, and *spawn()* in *Agents.callAll()* and thereafter committed with *Agents.manageAll()*. Listing 1.1 shows *main()* in a typical MASS-based data discovery, which loads a structured dataset into memory (line 4), populates a crawler agent (line 5), lets it start from place[0] (line 6), and schedules its dissemination over the dataset (lines 7-9). Weak consistency or barrier synchronizations are enforced between each statement of *callAll()* and *manageAll()*. Listing 1.2 describes each agent's behavior in *walk()* that examines all edges emanating from the current vertex (line12) and disseminates its copies along each edge (lines 13-16).

Listing 1.1. The main program

```
 1 import MASS.*;
 2 public class Analysis {
 3   public void main(String[] args) {
 4     Places dataset = new Places( ... );
 5     Agents crawlers = new Agents(“Crawler’’, dataset, 1);
 6     crawlers.callAll(ClawerAgent.init_, 0); // start from place[0]
 7     while ( crawlers.hasAgents() ) {
 8       crawlers.callAll(ClawerAgent.walk_);
 9       crawlers.manageAll();
10 } } }
```

Listing 1.2. Agent behavior

```
 1 public class Crawler extends Agent {
 2   public static final int init_ = 0; // fid 0 linked to init()
 3   public static final int walk_ = 1; // fid 1 linked to walk()
 4   public void init( Object arg ) {
 5     migrate( ( Integer )arg ); // let it start from place[arg]
 6   }
 7   public void walk( ) {
 8     if ( place.visited == true ) {
 9       kill( );
10       return; }
11     place.visited = true;
12     for ( int i = 0; i < place.neighbors.length; i++ ) {
13       if ( i == 0 )
14         migrate( place.neighbor[0] );
15       spawn( place.neighbors[i] );
16 } } }
```

Using MASS, we have so far developed two practical applications and one benchmark test set in data discovery: (a) global-warming analysis based on NetCDF climate data [15], (b) biological network motif search [7], and (c) comparison with MapReduce and Spark in six benchmark programs in graphs, optimizations, and data sciences [4]. Through the development work, we encountered two performance issues in agent management. One is an explosive increase of agent population that consumed a cluster system's memory space. For instance in biological network motif search, 5.5 million agents were spawned in total over $16MB \times 8$ compute instances to find all motifs with size five. The other is too many barrier synchronizations incurred between *Agents.callAll()* and *manageAll()*. To address these problems, we have developed two additional MASS features to improve the execution performance:

1. **Agent population control**: does not keep all *agents* active when they migrate over a dataset, thus allows users to specify the max cap of *agent* population in *MASS.Init(cap)*, serializes agents beyond this cap, and deserialize them when the population goes down; and
2. **Asynchronous and automatic agent migration**: unlike typical ABM simulation based on synchronous agent execution, will reduce synchronization overheads through *doAll(fid[ ],iterations)*, which automates asynchronous *iterations* of *callAll(fid)* and *manageAll()* invocations.

In [4,5,14], we demonstrated the MASS library's substantial performance improvements with these two new features. However, we have not yet addressed any challenges in agent programmability to pursue agent-based data discovery. Below we summarize two challenges we have identified as well as our solutions to them:

1. **Manual descriptions of agent decision-making logic**: this drawback requires users to precisely give agents step-by-step actions to take. For better descriptivity, agents should invoke their behavioral function automatically before their departure, upon their arrival at a new destination, and when receiving an inter-agent message. As a solution, we annotated *@OnDeparture*, *@OnArrival*, and *@OnMessage* to agent functions to invoke automatically.
2. **Emulation of inter-agent broadcast**: while *Places* can serve as an asynchronous mailbox shared among agents, the synchronous system-wide communication must be emulated at a user level via *main()* that becomes a focal point of collecting return values from and sending arguments to agent functions. We implemented direct inter-agent broadcast in the *Agents.exchangeAll()* function. It allows agents to smoothly share on-going computational states, e.g. the shortest route so far in ACO-based TSP.

3 Agent Behaviors to Support Data Discovery

This section describe our enhancement of agent descriptivity that supports event-driven agent behaviors and inter-agent message broadcast. We also differentiate our implementation techniques from other ABM systems.

3.1 Event-Driven Agent Behaviors

While the MASS library maintains weak consistency that guarantees a barrier synchronization of all agents upon executing *callAll()*, *manageAll()*, or *doAll()*, it is a big burden for users to repetitively invoke agent/place function calls from the *main()* program. Instead, such functions should be automatically invoked when their associated events are fired. Since agents duplicate themselves and ramble over a structured data, their major events are three-fold: agents' departure from the current place, their arrival at a new place, and their duplication. We allow users to associate agent functions with these three events, each annotated with *@OnDeparture*, *@OnArrival*, and *@OnCreation*. We also extend the *doAll(fid[ ],iterations)* function to *doWhile(lambda)* and *doUntil(lambda)*. They fire these events and commit the annotated functions while a given lambda expression stays true or until it sets true. With these features, users can focus on describing event-driven agent behavior rather than orchestrating their invocations.

A question comes up on how to trigger, to keep, and to terminate this event-processing sequence. To get started with agent-based data discovery, users are supposed to first populate agents on a structured dataset through *new Agents(AgentClassName, places)* that schedules their very first invocation of *@OnCreation* function, say *funcA*. They will then trigger *Agents.doWhile(lambda)* just only one time where *lambda* in many cases would be *()→agents.hasAgents()*, which repeats annotated agent functions until all agents are gone. If *funcA* schedules *migrate()* in it, *doWhile()* initiates the agent migration, before and after of which it invokes all *@OnDeparture* and *@OnArrival*-annotated functions, each named *funcB* and *funcC* respectively. If *funcC* schedules *spawn()*, *kill()*, and/or *migrate()*, the new events are continuously fired by *doWhile()*.

Listing 1.3 shows the simplification of the *main()* program with *doWhile*. The main program is completely relieved from agent behavioral orchestration. All it has to do is to initiate their repetitive data analysis (line 6). As we are currently implementing an interactive version of the MASS library with JShell [10], data-science specialists (who do not care of agent implementation) will be able to simply inject off-the-shelf agents into their dataset in an interactive fashion. On the other hand, model designers (who are interested in developing agent-based algorithms) can now clarify about which event will fire a given agent behavior, as shown in lines 2 and 6 in Listing 1.4.

The MASS library is a collection of Places/Agents functions, each called one by one from the main program but executed in parallel over a cluster system and with multithreading. Among them, the center of agent management is *Agents.manageAll()*. It examines all agents that have changed their status in the last *callAll* method through *spawn()*, *kill()*, or *migrate()*. How the *manageAll* function processed them is based on batch processing: (1) spawning all new children, thereafter (2) terminating those that called *kill()*, and finally (3) moving all that called *migrate()*. This in turn means that a barrier synchronization is carried out between each of these three actions, which thus enforces weak consistency. While agent annotation provides users with an option of event-driven programming, the underlying MASS execution model still maintains weak consistency

Listing 1.3. The main program with doWhile

```
  crawlers.callAll(ClawerAgent.init_, 0); // start from place[0]
  while ( crawlers.hasAgents() ) {
    crawlers.callAll(ClawerAgent.walk_);
    crawlers.manageAll();
```

should be replaced with

```
  crawlers.doWhile( ()→ crawlers.hasAgents() );
```

Listing 1.4. Agent behavior with agent annotations

```
public class Crawler extends Agent {
  @OnCreation
  public void init( Object arg ) {
    ...
  }
  @OnArrival
  public void walk( ) {
    ...
```

by the *manageAll* function that handles each of *OnCreation*, *OnDeparture*, and *OnArrival* annotations in the order summarized in Table 1. The *eventDispatcher* class in this table uses Java reflection to find each agent's method associated with a given annotation. In an attempt to reduce the effect of using Java reflection, the eventDispatcher caches methods associated with each annotation. Our initial testing has shown, as compared with the use of integers representing functions with *callAll* (see lines 2-3 in Listing 1.2), this caching method reduces the impact to an extent where it is of statistical insignificance. Since we focus on data scalability in data discovery rather than speed-up in ABM simulation, this implementation would not negate the advantages that MASS provides.

Table 1. Annotation handling in manageAll

actions	annotation handling

(1) spawn	for (Agent agent : agents)
	  eventDispatcher.invoke(OnCreation.class, agent);

(2) kill	no annotation handling

(3) migrate	for (Agent agent : agents)
	  eventDispatcher.invoke(OnDeparture.class, agent);
	move agents to their new place.
	... barrier synchronization among all cluster nodes ...
	for (Agent agent : agents)
	  eventDispatcher.invoke(OnArrival.class, agent);

3.2 Inter-Agent Message Broadcast

In general, it is not easy to deliver a message directly from one to another moving agent. As a solution to inter-agent messaging in MASS, we have originally chosen via-place indirect communication among agents that reside on the same place. In other words, agents use each place's data members as a mailbox. Rather than make a rendezvous, senders deposit their messages to a given place, whereas receivers read or pick them up later by visiting the same place. In many cases when agents disseminate over a graph, most frequent information exchanged among agents are run-time attributes of each place, regarding if the current place, (i.e., a vertex) has been visited or not, if the currently recorded travel distance to this vertex is longer than a new agent's distance traveled, and which vertex is the current vertex's predecessor. Therefore, we believe that the MASS indirect communication can cover many agent-based graph analyses.

However, this indirect communication is expensive to maintain message ordering or memory consistency, and thus cannot cover biologically-inspired optimizations such as ACO, GOA, and particle swarm optimization (PSO) [6]. After every migration of agents, they need to identify the agent temporarily closest to the optimal solution, for the purpose of having other agents follow the best. The details are given in Listings 1.5 and 1.6. The MASS main program (in Listing 1.5) collects the latest agent states from the first *Agent.callAll()* as the return values (line 8) and then scatters the best agent information to the second *Agent.callAll()* as the arguments (line 10). Since the main behaves as the focal point of this collective communication and the best agent sorting operation, the entire agent management code in lines 5-10 cannot be condensed into *doAll(max)*. Therefore, none of agent behavior such as *getState* and *approachToBest* functions in Listing 1.6 can be annotated with *@OnDeparture* nor *@OnArrival*, either.

Listing 1.5. MASS main of PSO

```
 1 main() {
 2   public void main(String[] args) {
 3     Places dataset = new Places( ... );
 4     Agents swarm = new Agents( "Particle", dataset, Integer.intValue(args[0]));
 5     Data[] data = new Data[swarm.nAgents()];
 6     for (int i = 0; i < max; i++) {
 7       swarm.manageAll();
 8       data = swarm.callAll(Particle.getState_); // gather each agent's value
 9       Data[] best = min( data ); // find the best value so far
10       swarm.callAll(Particle.approachToBest_, best); // scatter the best value
11 } }
```

Listing 1.6. MASS agent behavior of PSO

```
public class Particle extends Agent {
  int[] bestValue, myValue; // globally best or my current value
  public Object getState(Object arg) {
    myValue = {index[0], index[1], place.value};
    return myValue; // return my current value
  }
  public Object approachToBest(Object arg) {
    bestValue = (int[])arg;
    x = place.index[0] + updateVelocity( bestValue );
    y = place.index[1] + updateVelocity();
    migrate(x, y); // approach to the best agent so far
} }
```

To address this problem, we have implemented inter-agent message broadcast, so that each agent can collect the states from all the others directly and identify the best agent so far independently from the main program. The *MASSMessaging* class allows agents to broadcast a message to the other agents or places with *sendAgentMessage(message)* or *sendPlaceMessage(message)* respectively.

Using agent annotations, we deliver a message directly to each agent's or place's method that is annotated with *@OnMessage*. This relieves the main program from collective communication. As demonstrated in Listings 1.7, all agent behavioral coordination in the main is simplified into *doAll(max)* that repeats the *max* times of PSO agent migration. Listings 1.8 describes event-associated PSO agents. The logic to find the best value is moved from the main into the PSO agent (line 11). All the agent logic is clarified with arrival and message events.

Listing 1.7. Simplified PSO (main)

```
5      Data[] data = new Data[swarm.nAgents()];
6      for (int i = 0; i < max; i++) {
7        data = swarm.callAll(Particle.getState_);
8        ...
9        ...
10       swarm.callAll(Particle.approachToBest_, best);
```

should be replaced with

```
5      swarm.doAll(max);
```

Listing 1.8. Annotated PSO (agents)

```
1  public class Particle extends Agent {
2    int[] bestValue, myValue; // globally best or my current value
3    @OnArrival
4    public Object getValue(Object arg) {
5      myValue = {index[0], index[1], place.value};
6      MASSMessaging.sendAgentMessage(myValue);
7    }
8    @OnMessage
9    public Object approachToBest(Object best) {
10     bestValue = (int[])arg;
11     if ( myValue[2] > bestValue[2] )
12       // the same logic as lines 9−11 in Listing 1.6
```

Our initial implementation of agent messaging uses Hazelcast [7], an in-memory distributed data grid. Hazelcast was selected for performance, maturity, and ease of connecting nodes together. Our use of Hazelcast for messaging allows for agents to send messages (Java Objects) to either a specific agent, a collection of agents, or all agents. An agent may transmit a message by calling *sendAgentMessage* and providing either an enumerated value for a broadcast or one

[7] https://hazelcast.com/

or more agent ID numbers. The payload is serialized and the destination agents receive the message via a callback that is registered upon agent creation or migration to a different node. An agent method annotated with *@OnMessage* with either no arguments or a single argument with data type matching the payload is invoked and the message is delivered. Message delivery callbacks are queued using the aforementioned *eventDispatcher* so that messaging events are handled at the appropriate time.

3.3 Related Work

RepastHPC [12] and FLAME [3] are two representative platforms to run ABM simulations in parallel on top of MPI.

RepastHPC populates and moves agents over a shared space named *Projection*, which is similar to MASS. It allows users to code agent behavior as a collection of methods and to schedule their invocation events in their execution environment named *Context*. The main program then initiates an entire execution. However, RepastHPC's event scheduling is an enumeration of agent methods to be invoked along a repetitive time sequence. We feel that this scheduling strategy is weaker than MASS that associates each agent method with a named event. RepastHPC has its agents communicate with each other indirectly through *Project*. However, they have no message-broadcast feature.

FLAME views an ABM simulation as a collection of communicating, state-transiting agents statically mapped over MPI ranks. FLAME distinguishes two different languages: XML and C. XML declares agent interfaces and schedules their execution events, while C describes agent behaviors. In similar to RepastHPC, FLAME's event scheduling makes a list of agent methods to invoke. On the other hand, FLAME provides agents with a message box at each MPI rank, so that each agent can broadcast its state to all the others. However, message retrievals must be done within a code block specified in MESSAGE_LOOP. Therefore, FLAME can't invoke a given method upon a message broadcast rather than keep polling messages.

Besides event scheduling and inter-agent message broadcast, RepastHPC and FLAME are not a good choice for distributed data discovery rather than ABM simulation. The reasons are: RepastHPC needs to handle I/O in the main and cannot control agent population; and FLAME needs to enclose all spatial data in each agent [4].

4 Evaluation of Agent Behaviors

We compared the conventional and annotated versions of two MASS applications from the viewpoints of their code descriptivity and execution performance. These applications are agent-based BFS (breadth-first search) and PSO programs.

Our descriptivity comparison focuses on quantitative analysis of their MASS-Java code, in particular regarding their total lines of code (LoC) and boilerplate code (BP). The latter counts the lines of code that is irrelevant to the algorithms but needed to use the MASS library for their parallelization. Table 2 shows our analysis. As expected, both BFS and PSO annotated versions show a 37% to 58% reduction of LoC in the main program due to their simplification of agent coordination into a *doWhile()* or *doAll()* statement. In some cases, however, the number of agent BP lines is not reduced or may even increase slightly because of the addition of annotations.

Table 2. BFS and PSO Descriptivity

BFS					PSO				
		Total	Main	Agent			Total	Main	Agent
Conventional	LoC	160	27	43	Conventional	LoC	139	52	57
	BP	23	10	9		BP	25	15	6
Annotated	LoC	148	17	36	Annotated	LoC	102	22	57
	BP	21	9	9		BP	21	7	11
Reduction	LoC in %	7.5	37	16	Reduction	LoC in %	26.6	57.7	0
	BP in %	9	10	0		BP in %	16	53	-83

Our performance measurements compared both conventional and annotated version of BFS and PSO programs executed on top of the MASS library. Figure 1 visualizes their execution time. In BFS, its annotated version yielded 29% and 9% slow-down as compared to the conventional version when executed with a single thread and four threads respectively. In PSO, we observed a significant performance increase, most likely attributable to a large reduction in message transmission. This reduction in messaging overhead helped to reduce execution time by 83%. Contrary to our expectations, increasing the number of threads for PSO execution actually increased execution time, which we attribute to cache thrashing and memory contention.

While our previous performance evaluations on MapReduce and Spark in [4] used different problem sizes in BFS and PSO, each with 3000 vertices and 600×600 places respectively, we estimated their execution performance with 2048 vertices and $3K \times 3K$ places as in Figure 1. MapReduce and Spark would take 17.4 and 6.9 seconds in BFS and 79.5 and 562.4 seconds in PSO respectively. These estimates indicate that the annotated version of MASS performs 2 times slower in BFS but 8+ times faster in PSO than MapReduce and Spark. We are further improving MASS performance for solving graph problems faster.

5 Conclusions

To apply ABM to distributed analysis of structured dataset, we enhanced the MASS library with method annotations and inter-agent message broadcast, which allows data scientists to simply inject off-the-shelf agents from the main program, whereas developers of agent-based algorithms can focus on describing event-driven agent behaviors. Our analysis of agent programmability and execution performance demonstrated a drastic reduction in both LoC and in execution time for messaging-based applications. For non-messaging applications we observed less than a 10% negative performance effect. Additionally, we feel that this new event-driven behavior model more clearly encapsulates agent behavior, thus making the MASS applications easier to understand and easier to maintain.

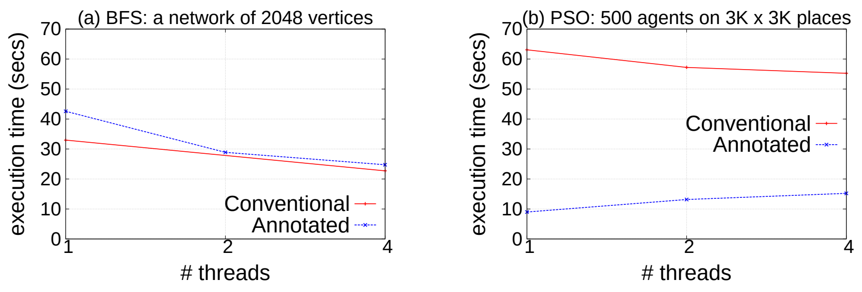

Fig. 1. BFS and PSO Execution Performance

References

1. Blum, C.: Ant colony optimization: introduction and recent trends. Physics of Life Reviews **2**(4), 353–373 (2005)
2. Buck, J., et al.: SciHadoop: Array-based Query Processing in Hadoop. In: Proceedings of SC'2011 (2011), doi:10.1145/2063384.2063473
3. FLAME: http://www.flame.ac.uk
4. Fukuda, M., Gordon, C., Mert, U., Sell, M.: Agent-Based Computational Framework for Distributed Analysis. IEEE Computer **53**(3), 16–25 (2020). https://doi.org/doi:10.1109/MC.2019.2932964
5. Gordon, C., et al.: Implementation techniques to parallelize agent-based graph analysis. In: Int'l Workshops of PAAMS 2019, Highlights of Practical Applications of Survivable Agents and Multi-Agent Systems. pp. 3–14. Avila, Spain (June 2019)
6. Kennedy, J., et al.: Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks IV. pp. 1942–1948 (1995)
7. Kipps, M., et al.: Agent and Spatial Based Parallelization of Biological Network Motif Search. In: 17th IEEE Int'l Conf. HPCC. pp. 786–791. New York (2015)
8. Lin, J., et al.: Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers (2010)
9. Low, Y., et al.: Distributed GraphLab: A Framework fro Machine Learning and Data Mining in the Cloud. In: Proc. of the 38th Int'l Conf. on Very Large Data Bases, Vol. 5, No. 8. pp. 716–727. Istanbul, Turkey (August 2012)
10. Oracle: Java Platform, Standard Edition, Java Shell User's Guide, Release 9. Tech. Rep. E87478-01 (2017)
11. Parsian, M.: Data Algorithms: Recipes for Scaling Up with Hadoop and Spark. O'Reilly (2015)
12. RepastHPC: https://repast.github.io/repast_hpc.html
13. Saremi, S., Mirjalili, S., Lewis, A.: Grasshopper optimization algorithm: Theory and application. Advances in Engineering Software **105**, 30–47 (2017)
14. Shih, Y., et al.: Translation of String-and-Pin-based Shortest Path Search into Data-Scalable Agent-based Computational Models. In: Proceedings of Winter Simulation Conference. pp. 881–892. Gothenburg, Sweden (December 2018)
15. Woodring, J., et al.: A Multi-Agent Parallel Approach to Analyzing Large Climate Data Sets. In: 37th IEEE ICDCS. pp. 1639–1648. Atlanta, GA (June 2017)