DOI: https://doi.org/10.1007/978-3-319-96983-1_36

OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations

Biagio Cosenza[1(✉)], Nikita Popov[1],
Ben Juurlink[1], Paul Richmond[2],
Mozhgan Kabiri Chimeh[2], Carmine Spagnuolo[3],
Gennaro Cordasco[3], and Vittorio Scarano[3]

[1] TU Berlin, Berlin, Germany
cosenza@tu-berlin.de
[2] University of Sheffield, Sheffield, UK
[3] University of Salerno, Fisciano, Salerno, Italy

Abstract. Agent-based simulations are becoming widespread among scientists from different areas, who use them to model increasingly complex problems. To cope with the growing computational complexity, parallel and distributed implementations have been developed for a wide range of platforms. However, it is difficult to have simulations that are portable to different platforms while still achieving high performance.

We present OpenABL, a domain-specific language for portable, high-performance, parallel agent modeling. It comprises an easy-to-program language that relies on high-level abstractions for programmability and explicitly exploits agent parallelism to deliver high performance. A source-to-source compiler translates the input code to a high-level intermediate representation exposing parallelism, locality and synchronization, and, thanks to an architecture based on pluggable backends, generates target code for multi-core CPUs, GPUs, large clusters and cloud systems.

OpenABL has been evaluated on six applications from various fields such as ecology, animation, and social sciences. The generated code scales to large clusters and performs similarly to hand-written target-specific code, while requiring significantly fewer lines of codes.

1 Introduction

Agent-based simulations (ABS) are a powerful instrument to study a wide range of scientific phenomena. According to Epstein [1], agent-based computational models are well-suited to the analysis of phenomena where agent populations are heterogeneous, there is no central control over individuals (autonomy), the space where the agents work is explicit (e.g., an n-dimensional grid), and agents only have local interactions with neighboring agents. Since SugarScape [2], computational models have been used to interpret society by translating social dynamics into a type of computation. Examples are voting behaviors [3], epidemics [2], and spatial unemployment patterns [4]. Applications go beyond social sciences,

from ecologists studying the predator-prey equilibrium [5] to hazard prevention in evacuations [6].

With an increasing number of applications where agent modeling is used, there is also a growing demand for computational power, due to larger agent populations and increasingly complex models. For this purpose, parallel and distributed implementations targeting different platforms, such as desktop GPUs [7], HPC architectures [8], and distributed cloud systems [9] have been developed, each focusing on a specific class of simulations and particular parallelization issues. While the core concepts of all existing frameworks are fundamentally the same, the variety of both hardware platforms and application contexts has led to very different implementations. Ideally, ABS should be written in a portable environment that can target a variety of parallel and distributed systems without any program modifications.

The necessity of portable solutions to reproduce simulations on different parallel implementations and hardware platforms, has led to the OpenAB initiative[1], a community-driven collaborative project to provide models and procedures for the benchmarking of multi-agent simulations on parallel and distributed computing systems. This work aims at providing an effective and efficient tool to these communities through the design and implementation of a *domain-specific language* (DSL) for portable, parallel and high-performance ABS.

The contributions of this paper are:

1. The design of OpenABL, a novel domain-specific language for agent-based computational modeling and simulation. The language targets the core requirements of these simulations with tailored high-level semantics, and enables parallel processing by explicitly exposing agent-parallelism.
2. A source-to-source compiler implementing the OpenABL language and supporting five different parallel and distributed backends, which are capable of running on diverse platforms such as multi-core CPUs, GPUs, and distributed clouds.
3. A collection of six test simulations from different application fields including biology, ecology and social sciences, and an experimental evaluation and comparison across different platforms.

2 Background

Many frameworks and libraries for implementing parallel ABS have been proposed; however, each addresses quite different target architectures, with distinct solutions for locality and synchronization. REPAST [10] is an agent-based simulation toolkit written in C++, later extended and parallelized into the REPAST-HPC framework [11], and tested on a Blue Gene/P HPC cluster. Cosenza et al. [8] introduced a distributed load balancing schema for parallel ABS that scales a simulation with one million agents on a cluster with 64 processors. Mason [12] is a popular multi-agent simulation library written in Java. D-Mason [13,14]

[1] More information is available at http://www.openab.org.

provides an effective and efficient way of parallelizing Mason programs for distributed systems, handling communication strategies and load balancing [15], tested on Amazon Web Services [15], and used on several social science scenarios [16]. Flame [7] is an agent-based environment based on an underlying formal model, called the X-Machine, and used in various scenarios such as cell simulations [17] and immune system modeling [18]. FlameGPU [19] is an extension of Flame that executes agent-based models on GPU architectures. Other GPU implementations have focused on bio-inspired visual clustering [20] and on efficient compression of agent direction [21]. Several authors have performed a comparison among ABS toolkits, both sequential [22] and parallel [23].

The idea of assuring portability across parallel implementations through DSLs has been exploited in many application scenarios, in particular to target large-scale computing systems [24]. Liszt [25] is the most similar to our work, with a DSL for constructing mesh-based PDE solvers and capable of targeting clusters, SMPs and GPUs. Liszt applications perform within 12% of hand-written C++ code and scale to large clusters. Other DSL have been designed for stencil computations [26], graph algorithms [27], and image processing pipelines [28].

3 Language Design

The goal of the OpenABL language is to provide a portable, efficient and easy-to-use environment for agent-based modeling. This is achieved by a rich language supporting domain-specific constructs, allowing the users to quickly prototype, reproduce, and compare different models with different parameters. The language also provides implicit support for agent parallelism and locality, so that OpenABL codes can be efficiently mapped onto parallel and distributed implementations.

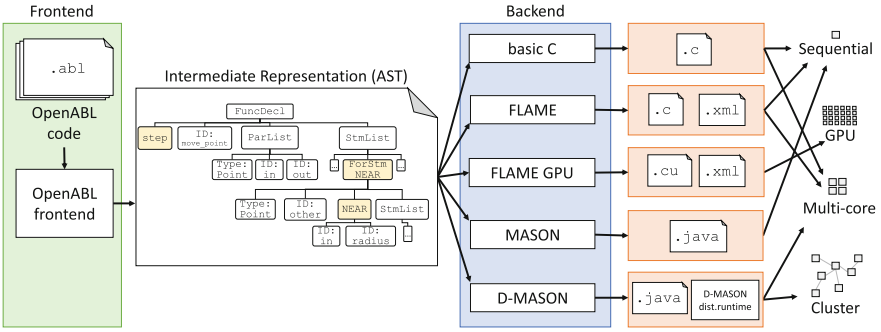

Fig. 1. OpenABL compilation workflow. The input code is translated to an AST-based intermediate representation, from which different backends generate code for specific platforms. The AST shows an excerpt of Listing 1.1.

Listing 1.1 shows a simple OpenABL code that demonstrates the general structure of a simulation. The program is subdivided into multiple top-level sec-

tions: agent declarations, simulation parameters, environment parameters, step functions and the main code. The language uses C-like syntax to maintain familiarity with mostly C and Java based ABS frameworks, and supports standard operators and control-flow statements, as well as vector types.

```
// Agent declarations
agent Point {
  position float3 pos;
}
// Simulation parameters and environment def.
float radius = 5;
float env_size = 100;
param int num_agents = 1000;
param int num_timesteps = 100;
environment { max: float3(env_size) }
// Step function
step move_point(Point in -> out) {
  // Move towards the average direction of the neighbors
  float3 dir = float3(0);
  int num_neighbors = 0;
  for (Point other : near(in, radius)) {
    dir += normalize(other.pos - in.pos);
    num_neighbors += 1;
  }
  out.pos = clamp(in.pos + dir/num_neighbors, float3(
      env_size));
}
// Main code: Initialization and execution
void main() {
  for (int i : 0..num_agents)
    add(Point {pos: random(float3(env_size))});
  simulate(num_timesteps) { move_point }
  save("result.json");
}
```

Listing 1.1. An OpenABL code example implementing a simple agent motion.

Step Functions and Agent Parallelism. It is important to incorporate agent parallelism into the language in a way that can be supported with the same semantics by all backends. In OpenABL, this is accomplished using *step functions*, which take an input agent of some type and yield a modified output agent. For example, in

```
step move_point(Point in -> out) { ... }
```

the input agent `in` of type `Point` is the result of the last timestep, while the output agent `out` will be the result of the current timestep. The output will only become available once a step function has been called for all agents of that type. Conceptually, this corresponds to a double buffering mechanism: an input buffer

of read-only agents and an output buffer of write-only agents, which are swapped at the end of a discrete simulation step. The strict `in`/`out` separation is required in order to produce deterministic, order-independent simulations. Surprisingly, we found that many sequential agent libraries such as Mason do not provide a native double-buffering mechanism. Therefore their results are not deterministic, as they depend on the updating order of the agents. OpenABL overcomes this limitation and always produces order-independent models.

The `simulate` statement invokes a simulation for a certain number of timesteps, during which a list of step functions will be executed in the given order. First one step function is executed for all agents (of the applicable type), before the next step function is run. Between step functions, the `out` parameter becomes the new `in` parameter. For instance, in the *Ants* model

```
simulate(num_timesteps) { ant_act1, pheromone_deposit,
    ant_act2 }
```

three step functions are called: the first on ant agents; the second on pheromone agents; the last again on ant agents.

The whole simulation starts at the `main` function, which is used to set up agents (typically from a file or randomly generated), to invoke the simulation and save the simulation results. Simulation parameters are declared as global constants. If a constant is prefixed with the `param` keyword it may also be overridden from the command line.

Locality. A fundamental concept of agent-based modeling is locality, because interactions are usually limited to nearby agents. In general, this may be governed by arbitrary topologies, but OpenABL is currently limited to the common case of two- and three-dimensional Euclidean topologies. Each agent declares a designated `position` member of type `float2` or `float3`, which provides the position of the agent for spatial queries. The agent neighborhood can then be accessed through the combination of a `for` loop and a radius-based `near()` query:

```
for (AgentT neighbor : near(in, radius)) { ... }
```

The type of the input agent and the type of the neighboring agents that are fetched does not necessarily have to match. The query is performed using a backend-specific spatial acceleration data structure such as grid [19] and kd-tree [29].

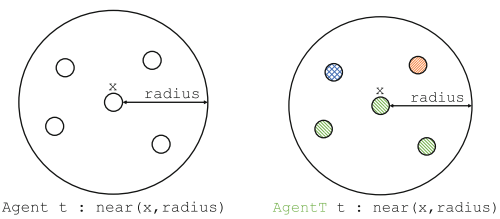

Fig. 2. `near()` queries with homogeneous and heterogeneous agent types.

Simulations with *heterogeneous* agents, i.e. with multiple agent types, are implemented with multiple step functions on different agent types, and by specifying different return types on `near()` queries, as shown in Fig. 2.

Environment properties are specified using an `environment` declaration which includes the environment dimensionality and bounds (`min` and `max`). Agent positions must stay within these bounds. For performance reasons, this is not automatically enforced by the language, but functions to perform the necessary clamping or wrap-around are provided. The radius used for spatial acceleration structures is usually determined automatically, but may also be explicitly given here. The standard library also provides commonly used functions for geometric and trigonometric operations.

Dynamic Agent Creation and Removal. Typically, the agents are created at the beginning of the simulation and are not removed until the end; however, some simulations require a dynamic mechanism for the creation and removal of agents. For example, in the *Predator-Prey* model, predators pursue and eat prey, who reproduce at a given rate. As a result, the agent populations periodically increase and decrease during the course of a simulation. The language enforces a number of additional constraints for backend compatibility: in a step function, each agent may add at most one new agent. This means that in a single step, the number of agents can at most double. The new agent position must be the same as the generating agent position (e.g., `in.pos`). An agent can remove itself by using the `removeCurrent()` function, but cannot remove a different agent.

4 Implementation

Compilation Process. OpenABL is a source-to-source compiler employing a classical three-stage pipeline: First, Flex and Bison are used to parse the source code into an *abstract syntax tree* (AST), which acts as our primary intermediate representation. Then, target-independent analysis is performed, which validates the code and enforces semantic constraints, while also annotating the AST with necessary type and dependency information. Finally, different backends emit source code (and other auxiliary files) for the target platform based on the annotated AST.

Backends. OpenABL currently supports five backend implementations targeting different agent models, acceleration data structures and platforms.

A *basic C* backend serves as a reference implementation and basis for other C-based backends. It does not use acceleration structures, implements double-buffering using two arrays of agents swapped after each step, and uses OpenMP for trivial parallelization.

Flame [7] models agents using X-Machines, which are state machines that support sending messages between agents. A step function can modify the memory of the current agent, send messages and iterate over messages sent in the

previous step. To support neighborhood queries, we determine which members of neighboring agents are used inside a for-near loop and generate one step function that sends a message containing all the used members. A second step function processes the messages falling into the specified neighborhood. A side-effect of this process is that no explicit double buffering of the agent memory is necessary: because messages are generated in a previous step, changes to agents in the current timestep are not observable. The backend generates the three parts of a Flame model: an XML model specification, an XML initial agent state file, and the C step function definitions. Flame does not support adding or removing agents at runtime and does not use spatial acceleration structures.

FlameGPU [19] is based on Flame, but targets execution on the GPU using CUDA. Our FlameGPU backend is structurally similar to the Flame backend. It supports grid-based spatial acceleration, which requires a specification of the environment dimensions and partitioning radius in the XML model. The environment bounds must be adjusted upwards to be multiples of the radius. Runtime agent removal/addition is supported, but only the current agent may be removed and one added per step function. Both restrictions are enforced by the language.

Mason [12] is an ABS and visualization library written in Java. The two main components are an environment, which supports grid-based neighborhood queries, and a schedule, which executes the step functions. Mason does not have native support for double-buffering: simulations are fundamentally order-dependent, based on the assumption that for most models it does not make a significant difference if the state from the current (rather than previous) timestep is used for some agents. To support our execution semantics, agents hold two state objects, which are used alternately and swapped at the end of a step function. In Mason, each agent has only a single step function; however, our execution model may require multiple step functions, executed for all agents and in a specific order.[2] We solve this with a cyclic counter for each agent indicating which step function to execute. Mason supports both removal and addition of agents at runtime. This backend also produces code for the visualization of the simulation.

D-Mason [13,14] is a distributed extension of Mason that allows the distribution of the simulation across multiple, even heterogeneous machines. It is based on a Master/Workers paradigm where the master partitions the simulation environment into regions. All the agents in a region are assigned to a machine, which performs the simulation, handles the migration of agents, and manages the synchronization between neighboring regions. The D-Mason communication mechanism is based on the Publish/Subscribe pattern. Unlike Mason, D-Mason requires environments to use only positive coordinates. D-Mason supports agent removal/addition at runtime; however, new agents must be positioned in the current space partition.

Compiler flags are provided for further backend-specific configuration. For example, the `float` data-type used by OpenABL is mapped to double-precision

[2] While Mason itself supports multiple step functions in the form of anonymous Steppables, this is not supported by D-Mason, so a different solution is required.

floats by default, because this is the only type supported by all backends, but Flame and FlameGPU can switch to single-precision through a compiler flag.

5 Experimental Evaluation

OpenABL has been evaluated on six applications in terms of programmability of the language and the performance of the code generated for the five backends, including single-node performance on CPUs and GPUs, scalability on a cluster, and a comparison against hand-written target-specific code.

Reference Simulation Models. The evaluation uses six agent-based models from different domains, for which reference implementations were available for at least one of our targets. Table 1 lists general properties of these models.

Circle is a standardized benchmark part of the OpenAB initiative, for assessing the performance of fixed-radius near neighbor lookups, formally defined by Chisholm et al. [30]. *Boids* [31] is a steering behavior for autonomous characters in animation and games, which simulates the flocking behavior of birds. The agent motion is derived from three components: separation, alignment, and cohesion. Conway's *Game-of-life* is a cellular automaton model, implemented with one agent per cell and an `alive` boolean status variable. *Sugarscape* is a social science model where agents move on a grid of a regrowing resource (sugar), which they must consume to survive. We implement it using a stationary grid of agents. The *Ants Foraging* model simulates ants that, when they discover a food source, establish a trail of pheromones between the nest and the food source. The model uses two pheromones, which set up gradients and evaporate after some simulation steps, to the nest and to the food source respectively. We parallelized the original Mason model [32]: sequential access to global data structures, which is not suitable for parallelization, has been replaced by two step functions that handle the deposition and evaporation of (grid) pheromones, and one for the ant movement. *Predator-Prey* is our largest model, which involves three different agent types (prey, predator and grass), 13 step functions, and utilizes dynamic agent creation and removal. Both predators and prey implement short-range collision avoidance, a mid-range flocking, and can reproduce at different rates. Each predator follows the closest prey, which is eaten if it is too close. Conversely, prey avoids predators and eats grass, which regrows after a fixed time interval. All simulations have been executed with a different number of agents. The environment is scaled with the square root (for two-dimensional simulations) of the agent number, so that the agent density remains constant.

Programmability Evaluation. To evaluate the programmability and ease of use of OpenABL, we compare the eLOC (effective lines of code, ignoring comments and blank lines) of OpenABL models with available reference models

Table 1. Simulation benchmarks with the number of agent types, number of step functions, whether dynamic agent addition/removal is used (AR), effective lines of code (eLOC) of the implementations in OpenABL, FlameGPU and D-Mason.

Application	Area	Model properties			Implementation size in eLOC		
		Types	Steps	AR	OpenABL	FlameGPU	D-Mason
Circle	Micro-benchmark	1	1		36	184 (×5.1)	537 (×14.9)
Boids	Animation	1	1		82	240 (×2.9)	767 (×9.4)
Game of Life	Cellular automaton	1	1		48	133 (×2.8)	477 (×9.9)
Sugarscape	Social science	1	4		154	345 (×2.2)	n/a
Ants Foraging	Animal ecology	2	3		191	n/a	967 (×5.1)
Predator-Prey	Animal ecology	3	13	✓	248	858 (×3.5)	n/a

from FlameGPU and D-Mason.[3] As seen in Table 1, the FlameGPU implementations are 2–5 times larger, while the D-Mason models are 5–15 times larger. While eLOC is not a very reliable indicator of programmability, it is clear that OpenABL models tend to be significantly more compact than manual implementations.

Single-Node Performance Comparison. For single-node performance evaluation, we compared the performance of the code generated by the OpenABL compiler for the basic C, Mason, Flame and FlameGPU backends. The six test models were run for 100 timesteps with population sizes ranging from 250 to 10^6 agents. The *Predator-Prey* model was only evaluated on backends supporting dynamic addition/removal of agents. The benchmarks were performed on a system with an Intel Core i5-4690K CPU (4 cores at 3.50 GHz), 16 GB of memory, running on Ubuntu 16.04. For FlameGPU, we used an NVIDIA Titan Xp (Pascal architecture) with 12 GB of memory. The basic C backend was configured to use multiple threads using OpenMP.

The results in Fig. 3 show that Flame and basic C scale quadratically with the number of agents. This is expected, as they do not use any spatial acceleration structure. Mason is much faster than Flame, and is the best implementation for small-sized simulations. FlameGPU pays a high overhead for small-sized simulations, because of the data transfer from the host to the GPU; however, it is the best-performing solution for larger simulations with more than 10^4 agents. For most models, both Mason and FlameGPU scale approximately linearly at large population counts. One notable exception is *Ants*, where Mason degenerates to quadratic behavior, because of a very dense agent clustering at the start of the simulation.

[3] The used reference models are available at https://github.com/FLAMEGPU/FLAMEGPU, https://github.com/FLAMEGPU/Tutorial and https://github.com/isislab-unisa/dmason.

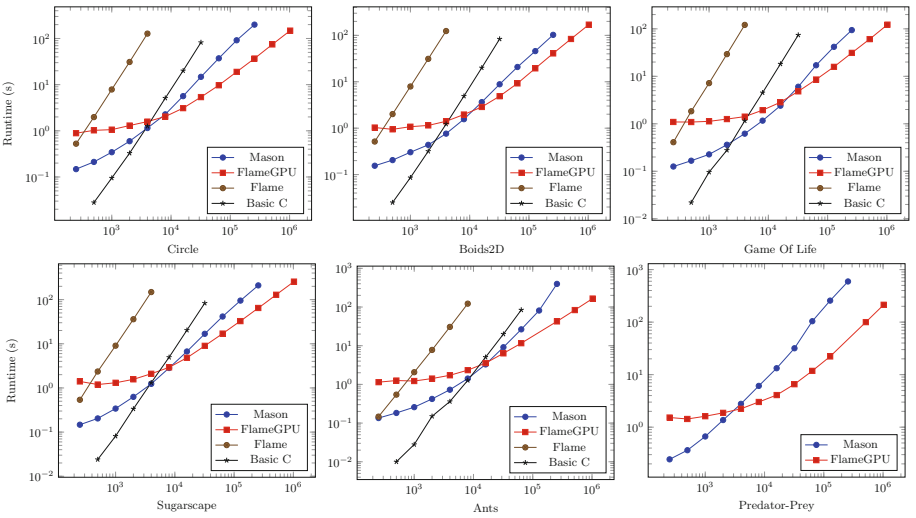

Fig. 3. Performance of generated Mason, FlameGPU, Flame and basic C code with a different number of agents (x-axis).

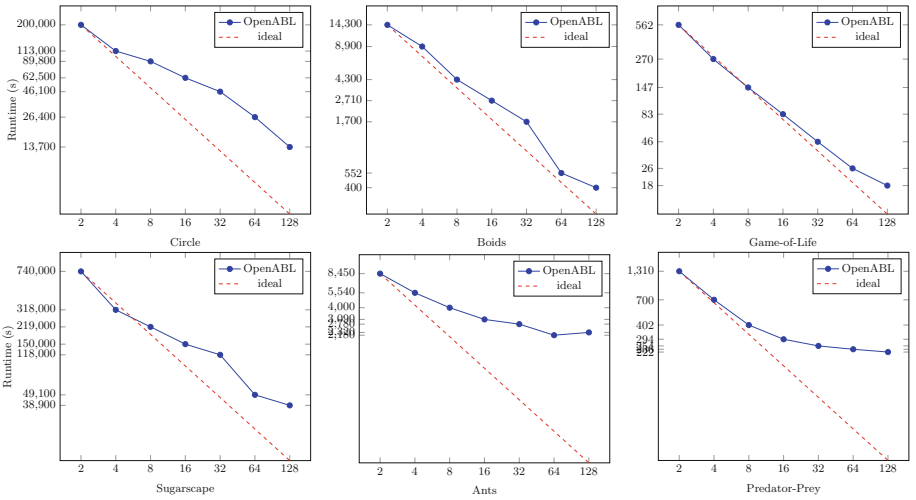

Fig. 4. OpenABL D-Mason strong scaling with different number of cores (x-axis).

Cluster Scaling. To evaluate the performance scaling of the OpenABL D-Mason backend, we use a cluster of 12 nodes equipped with two Intel Xeon E5-2430 (six cores) with hyper-threading disabled, connected by I350 Gigabit network adapters. One node is used for coordinating the simulation, while the others allocate one D-Mason logical processor for each core, running on Oracle JVM 1.8 and exploiting Apache ActiveMQ as message broker for communica-

tion/synchronization (the broker is allocated on the coordinating node). Figure 4 shows the strong scalability of the six applications, with each model simulating 10^6 agents for 1000 timesteps. The plots show the runtime in seconds for an increasing number of logical processors (cores).

The *Boids* model exhibits good scalability. Despite having similar behavior, *Circle*'s scaling is slightly worse due to different parametrization, e.g., a wider interaction radius. In *Game-of-Life* and *Sugarscape*, agents are distributed evenly in the space (on a grid) and are stationary, resulting in high scalability. *Ants* and *Predator-Prey* are the most complex simulations. The *Ants* model suffers from a dense concentration of the ant agents, especially at the start of the simulation, resulting in an uneven distribution of the workload. *Predator-Prey* is highly dynamic because of the addition/removal of agents, which drastically affects prey-crowded areas after the arrival of predators. This represents a challenge for distributed memory system, leading to bad scalability. We believe that more advanced load balancing strategies may substantially improve this aspect [33].

Performance Comparison Against Manually-Tuned Code. The potential overhead of the generated code has been evaluated against manual implementations of the *Boids* benchmark, because it is available for most libraries and is simple to validate. Results are summarized in Table 2. The generated code for Mason is 9% slower than the manual implementation; the reason is the double buffering mechanism introduced by OpenABL to ensure order-independent correctness, not supported in standard Mason. For FlameGPU, both generated code and manual implementation have similar performance: the semantics of the language map very well without any noticeable overhead. The overhead for D-Mason is 30%, motivated essentially by an improvement of the synchronization mechanism for each step function (agent buffer analysis may potentially reduce such overhead by avoiding unnecessary synchronizations). We omitted Flame from the comparison because of its very poor scalability (impractical with >5000 agents).

Table 2. Overhead of OpenABL generated code for *Boids* model compared to manually tuned code.

Backend	Overhead	Main reason
Mason	9%	Double-buffering
D-Mason	30%	Double-buffering and additional synchronization
Flame	n/a	(Too slow to compare)
FlameGPU	0%	Perfect programming model match

6 Conclusion

We present OpenABL, a domain-specific language designed for agent modeling on high-performance parallel and distributed architectures. It comprises an easy-to-program language that relies on high-level abstractions for programmability and explicitly exploits agent parallelism to deliver high-performance. It supports a wide range of context-specific semantics such as order-independent step functions, neighborhood queries, heterogeneous agents, and dynamic agent addition and removal. A source-to-source compiler translates the input OpenABL code into an AST-based intermediate representation exposing parallelism, locality and synchronization at the agent level. Subsequently, a collection of pluggable backends generate target codes for multi-core CPUs, massively parallel GPUs, large clusters and cloud systems. The OpenABL generated codes have been tested on a collection of six applications from various fields. While a program written in OpenABL is much smaller than one written for non-portable platform-specific libraries, its performance is very close to manual implementations.

OpenABL is an open source project available at https://github.com/OpenABL/OpenABL, with the goal of becoming an open research platform. The used code and instructions to reproduce our benchmarking results are available in a figshare repository [34].

This research has been partially funded by the DFG project *CELERITY* (CO 1544/1-1) and by the EPSRC fellowship *Accelerating Scientific Discovery with Accelerated Computing* (EP/N018869/1).

References

1. Epstein, J.M.: Agent-based computational models and generative social science. Complexity **4**(5), 41–60 (1999)
2. Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. The Brookings Institution, Washington, D.C. (1996)
3. Kollman, K., Miller, J.H., Page, S.E.: Adaptive parties in spatial elections. Am. Polit. Sci. Rev. **86**(4), 929–937 (1992)
4. Topa, G.: Social interactions, local spillovers and unemployment. Rev. Econ. Stud. **68**(2), 261–295 (2001)
5. Haynes, T., Sen, S.: Evolving behavioral strategies in predators and prey. In: Weiß, G., Sen, S. (eds.) IJCAI 1995. LNCS, vol. 1042, pp. 113–126. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60923-7_22
6. Pelechano, N., Allbeck, J.M., Badler, N.I.: Controlling individual agents in high-density crowd simulation. In: EG Symposium on Computer Animation, pp. 99–108 (2007)
7. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: FLAME: simulating large populations of agents on parallel hardware architectures. In: Conference on Autonomous Agents and Multiagent Systems, pp. 1633–1636 (2010)
8. Cosenza, B., Cordasco, G., Chiara, R.D., Scarano, V.: Distributed load balancing for parallel agent-based simulations. In: International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP, pp. 62–69 (2011)

9. Carillo, M., Cordasco, G., Serrapica, F., Spagnuolo, C., Szufel, P., Vicidomini, L.: D-Mason on the cloud: an experience with amazon web services. In: Desprez, F., et al. (eds.) Euro-Par 2016. LNCS, vol. 10104, pp. 322–333. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58943-5_26
10. North, M.J., Collier, N.T., Vos, J.R.: Experiences creating three implementations of the repast agent modeling toolkit. Trans. Model. Comp. Sim. **16**(1), 1–25 (2006)
11. Collier, N., North, M.: Parallel agent-based simulation with repast for high performance computing. SIMULATION **89**(10), 1215–1235 (2013)
12. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: a multi-agent simulation environment. SIMULATION **81**(7), 517–527 (2005)
13. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A framework for distributing agent-based simulations. In: Alexander, M., et al. (eds.) Euro-Par 2011. LNCS, vol. 7155, pp. 460–470. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29737-3_51
14. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON. SIMULATION **89**(10), 1236–1253 (2013)
15. Cordasco, G., Chiara, R.D., Raia, F., Scarano, V., Spagnuolo, C., Vicidomini, L.: Designing computational steering facilities for distributed agent based simulations. In: SIGSIM Principles of Advanced Discrete Simulation, pp. 385–390 (2013)
16. Lettieri, N., Spagnuolo, C., Vicidomini, L.: Distributed agent-based simulation and GIS: an experiment with the dynamics of social norms. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 379–391. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_31
17. Oliveira, A.P., Richmond, P.: Feasibility study of multi-agent simulation at the cellular level with FLAME GPU. In: FLAIRS Conference, pp. 398–403 (2016)
18. Tamrakar, S., Richmond, P., D'Souza, R.M.: PI-FLAME: a parallel immune system simulator using the FLAME graphic processing unit environment. SIMULATION **93**(1), 69–84 (2017)
19. Richmond, P., Walker, D., Coakley, S., Romano, D.: High performance cellular level agent-based simulation with FLAME for the GPU. Brief. Bioinform. **11**(3), 334 (2010)
20. Erra, U., Frola, B., Scarano, V.: A GPU-based interactive bio-inspired visual clustering. In: Symposium on Computational Intelligence and Data Mining, pp. 268–275 (2011)
21. Cosenza, B.: Behavioral spherical harmonics for long-range agents' interaction. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 392–404. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_32
22. Macal, C.M., North, M.J.: Tutorial on agent-based modeling and simulation. In: 37th Conference on Winter Simulation, pp. 2–15 (2005)
23. Rousset, A., Herrmann, B., Lang, C., Philippe, L.: A survey on parallel and distributed multi-agent systems. In: Lopes, L., et al. (eds.) Euro-Par 2014. LNCS, vol. 8805, pp. 371–382. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14325-5_32
24. Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: A uniform approach for programming distributed heterogeneous computing systems. J. Parallel Distrib. Comput. **74**(12), 3228–3239 (2014)
25. DeVito, Z., et al.: Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Conference on High Performance Computing Networking, Storage and Analysis, pp. 9:1–9:12 (2011)

26. Christen, M., Schenk, O., Cui, Y.: Patus for convenient high-performance stencils: evaluation in earthquake simulations. In: Conference on High Performance Computing, Networking, Storage and Analysis, SC, pp. 11:1–11:10 (2012)
27. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: ASPLOS, pp. 349–362 (2012)
28. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S.P., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. **31**(4), 32:1–32:12 (2012)
29. Kofler, K., Steinhauser, D., Cosenza, B., Grasso, I., Schindler, S., Fahringer, T.: Kd-tree based N-body simulations with volume-mass heuristic on the GPU. In: 2014 IEEE International Parallel and Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014, pp. 1256–1265 (2014)
30. Chisholm, R., Richmond, P., Maddock, S.: A standardised benchmark for assessing the performance of fixed radius near neighbours. In: Desprez, F., et al. (eds.) Euro-Par 2016. LNCS, vol. 10104, pp. 311–321. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58943-5_25
31. Reynolds, C.W.: Flocks, herds and schools: a distributed behavioral model. ACM SIGGRAPH **21**(4), 25–34 (1987)
32. Panait, L., Luke, S.: A pheromone-based utility model for collaborative foraging. In: Conference on Autonomous Agents and Multiagent Systems, pp. 36–43 (2004)
33. Cordasco, G., Cosenza, B., De Chiara, R., Erra, U., Scarano, V.: Experiences with mesh-like computations using prediction binary trees. Scalable Comput.: Pract. Exp. Sci. Int. J. Parallel Distrib. Comput. (SCPE) **10**(2), 173–187 (2009)
34. Cosenza, B., et al.: OpenABL: a domain-specific language for parallel and distributed agent-based simulations, figshare. Fileset (2018). https://doi.org/10.6084/m9.figshare.6384413