Article:
Cosenza, B., Popov, N., Juurlink, B. et al. (5 more authors) (2021) Easy and efficient agent-based simulations with the OpenABL language and compiler. Future Generation Computer Systems, 116. pp. 61-75. ISSN 0167-739X

https://doi.org/10.1016/j.future.2020.10.014

Easy and Efficient Agent-based Simulations with the OpenABL Language and Compiler

Biagio Cosenza[1a], Nikita Popov[b,e], Ben Juurlink[b], Paul Richmond[d], Mozhgan Kabiri Chimeh[f], Carmine Spagnuolo[a], Gennaro Cordasco[c], Vittorio Scarano[a]

[a] *Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, Italy*
[b] *Embedded System Architecture, Technische Universität Berlin, Berlin, Germany*
[c] *Dipartimento di Psicologia, Università degli Studi della Campania, Caserta, Italy*
[d] *University of Sheffield, Sheffield, UK*
[e] *JetBrains, Prague, Czech Republic*
[f] *NVIDIA UK*

Abstract

Agent-based simulations represent an effective scientific tool, with numerous applications from social sciences to biology, which aims to emulate or predict complex phenomena through a set of simple rules performed by multiple agents. To simulate a large number of agents with complex models, practitioners have developed high-performance parallel implementations, often specialized for particular scenarios and target hardware. It is, however, difficult to obtain portable simulations, which achieve high performance and at the same time are easy to write and to reproduce on different hardware. This article gives a complete presentation of OpenABL, a domain-specific language and a compiler for agent-based simulations that enable users to achieve high-performance parallel and distributed agent simulations with a simple and portable programming environment. OpenABL is comprised of (1) an easy-to-program language, which relies on domain abstractions and explicitly exposes agent parallelism, synchronization and locality, (2) a source-to-source compiler, and (3) a set of pluggable compiler backends, which generate target code for multi-core CPUs, GPUs, and cloud-based systems. We evaluate OpenABL on simulations from different fields. In particular, our analysis includes predator-prey and keratinocyte, two complex simulations with multiple step functions, heterogeneous agent types, and dynamic creation and removal of agents. The results show that OpenABL-generated codes are portable to different platforms, perform similarly to manual target-specific implementations, and require significantly fewer lines of codes.

Keywords: agent-based simulation, domain specific language, GPU, parallel and distributed computing, compilers

1. Introduction

The identification and study of common computational patterns is an important way to produce software that is systematically efficient on a variety of similar applications

[1] *Corresponding Author, bcosenza@unisa.it*

and on different target architectures. By focusing on these patterns, we give names to solutions to recurring problems that experts in a problem domain gradually learn and take for granted.

Agent-based simulations (ABS) [1] represents a way to model, through a population of agents, the dynamics of complex phenomena. Such agents follow a few simple principles: they may be of different types (heterogeneous), move in an explicit space such as an multi-dimensional grid, only interact with close agents (locality), and there is no central control over them (autonomous). Emergent behaviors from ABS usually result in computational patterns.

Historically, ABS computational models have been used to model and understand the society by translating social dynamics into computation. Examples are voting behaviors [2], epidemics [3], stock markets [4] and spatial unemployment patterns [5]. Applications go far beyond social sciences, however: the predator-prey equilibrium investigated by ecologists [6]; hazard prevention in evacuation scenarios [7]; keratinocyte colony formation modeled by cellular biologists [8]; the complex behaviors of cooperation simulated by ecologists [6]; crowd simulation interactions in virtual scenes [9] are all notable examples of ABS applications.

All of these ABS applications are characterized by the fact that they require a significant amount of computational power as both a larger number of agents is required to accurately model large populations and, at the same time, models are getting more sophisticated, with complex interactions and strong behavior differentiation between agents in the same simulation, i.e., increasing heterogeneity.

These two aspects have motivated scientists to develop parallel and distributed ABS implementations targeting very different platforms such as Graphics Processing Units (GPUs) [10], High Performance Computing (HPC) clusters [11], and cloud computing systems [12]. Each implementation focuses on a specific agent domain and on platform-specific parallelization methods. Notwithstanding the fact that the core concepts of existing ABS libraries are basically the same, the diversity of both target hardware platforms and application domains has led to very different software infrastructures. Unfortunately, this also led to simulation environments whose comparison and replication is hard to perform.

As a consequence, scientists using ABS face four major challenges:

- **Performance**: The increasing model complexity and the growing number of agents demand for higher computational power, with parallel and distributed implementations, and for efficient algorithms for local agents query. Both are necessary to simulate modern large and complex ABS.
- **Programmability**: Scientists and domain experts often have little expertise on HPC programming; thus, they need high-level tools and programming environments where only domain knowledge is required.
- **Portability**: Existing approaches for writing ABS simulation spans different tool and implementations. Agent simulations should be written in a programming environment that makes it possible to target a variety of parallel and distributed systems without any program changes.
- **Reproducibility**: Simulations must be easy to reproduce, returning *similar* results even if obtained by different implementations and platforms.

In the pursuit of finding a solution to all of these issues, scientists from different fields have joined their forces into the community-driven project OpenAB [13], which provides procedures, data and models for the benchmarking of agent-based simulations on parallel and distributed computing systems. This article gives a complete introduction of OpenABL [14], the *domain-specific language* (DSL) developed by and for the OpenAB community to tackle the four major challenges raised above, and provides an exhaustive evaluation of the OpenABL compiler implementation on complex simulations and several parallel target platforms.

The rest of this article is organized as follows. Section 2 introduces the OpenABL language, a domain-specific and high-level language that allows scientists to write complex agent models with few lines of code and no knowledge about the target platform. The language is designed to easily define space and locality properties of a simulation, and supports advanced features such as heterogeneous agent types and the dynamic creation and removal of agents. Two cornerstones of the language are the step function, which implicitly enable agent parallelism, and the near query, which is the base for local agent query. In this article, each language semantic is discussed in detail, with code excerpt taken from a complex application scenario. *predator-prey*. Additionally, two new language semantics are introduced: sequential step functions and reductions.

A source-to-source compilation infrastructure implements the OpenABL language and maps language semantics such as agent, step function and near query, into a high-level intermediate representation. Thus, a code generation infrastructure based on pluggable backends is capable to efficiently translate those semantics and generate high-performance parallel code targeting a broad range of systems, from multi-core CPUs and GPUs to cloud systems. Section 3 describes in details the parsing, the intermediate representation, the parallel mapping and the limitations of the source-to-source compilation infrastructure.

An extended experimental evaluation is presented in Section 4. We have evaluated the performance of the code generated by the OpenABL compiler on seven applications. In particular, we used two complex applications with both heterogeneous agent types and dynamic agent addition/removal as a test benchmark: predator-prey, an animal ecology simulation where a set of predators hunt for preys; and keratinocyte, a biological simulation that explores the self-organization of *normal human keratinocytes* (NHK) and how the cell-cell and cell-substrate adhesions are critically important to NHK self-organization.
Finally, this article presents a discussion of related work (Section 5) and concluding remarks (Section 6).

2. Language Design and Semantics

The aim of the OpenABL language is to provide a simple and easy-to-use programming environment for agent-based modeling. To this purpose, the language design includes a collections of domain-specific semantics, allowing the users to quickly prototype different models with different tuning parameters. An important language feature is its implicit support for agent parallelism and locality; this is fully exploited by the compilation infrastructure for efficient mapping onto parallel and distributed implementations (details in Section 3).

In this section, we will use excerpts from the *predator-prey* code example[2] to illustrate the general structure of an OpenABL model and its language semantics.

2.1. Agent Declarations, Parameters and Main Function

An OpenABL program is comprised of five parts: the declaration of agent types, the simulation and environment parameters, step function definitions, and the main function. A C-like syntax is used to maintain familiarity with ABS users (mostly adopting C and Java-based ABS frameworks), and supports standard semantics for operators and control-flow, as well as vector types.

```
// Agent declarations
agent Predator {
  position float2 pos;
  float2 dir;
  float2 steer;
  int life;
}
agent Prey {
  position float2 pos;
  float2 dir;
  float2 steer;
  int life;
}
agent Grass {
  position float2 pos;
  int dead_cycles;
  bool avail;
}
// Tunable model parameters
param float REPRODUCE_PREY_PROB = 0.005;
param float REPRODUCE_PRED_PROB = 0.001;
param int GAIN_FROM_FOOD_PRED = 35;
param int GAIN_FROM_FOOD_PREY = 60;
param int GRASS_REGROW_CYCLES = 60;
// Hard-coded model parameters
float PRED_PREY_INTER_RADIUS  = 0.100;
float PREY_GROUP_COH_RADIUS = 0.120;
float SAME_SPECIES_AVOID_RADIUS = 0.035;
// Environment definition
float env_size = sqrt(num_agents/agent_density);
environment { max: float2(env_size) }
```

Listing 1: An OpenABL code excerpt from the predator-prey benchmark.

The code declares three agent types. Agent types are declared similarly to structs in C. Additionally, each agent may have a designated `position` member of type `float2` or `float3`, which provides the position of the agent for spatial queries and may also be used for visualization; later in Section 2.3 it will be shown how this field is used to provide locality information to the OpenABL compiler.

The parameters of the simulation are declared as global constants. The `param` keyword, used as prefix for constants, indicates a parameter that may also be overridden

[2]The full code is available at `https://github.com/OpenABL/OpenABL/blob/master/examples/predator_prey.abl`.

from the command line. Overridden parameters are specified during code generation rather than at runtime, because they may be used in contexts that some backends cannot modify at runtime.

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 data 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.

An OpenABL simulation starts with the `main` function, which is the entry point typically used to set up agents, either from a file or randomly generated, to invoke the step functions implementing the simulation logic, and to finally save the results of the simulation. The following code snippet shows the main function of *predator-prey*.

```
void main() {
  int num_pred = int(pred_fraction * num_agents);
  int num_prey = int(prey_fraction * num_agents);
  int num_grass = num_agents - num_pred - num_prey;
  // Agent initialization
  for (int i : 0..num_grass) {
    add(Grass {
      pos: random(float2(env_size)),
      dead_cycles: 0, avail: true
    });
  }
  for (int i : 0..num_prey) {
    add(Prey {
      pos: random(float2(env_size)),
      dir: randomDirection(),
      steer: float2(0),
      life: GAIN_FROM_FOOD_PREY + randomInt(10)
    });
  }
  for (int i : 0..num_pred) {
    add(Predator {
      pos: random(float2(env_size)),
      dir: randomDirection(),
      steer: float2(0),
      life: GAIN_FROM_FOOD_PRED + randomInt(10)
    });
  }
  // A simulation loop invoking 13 step functions for a certain number of
      time steps
  simulate(num_timesteps) {
    pred_follow_prey, prey_avoid_pred,
    prey_flock, pred_avoid,
    prey_move, pred_move,
    prey_eat_or_starve, pred_eat_or_starve,
    grass_eaten, prey_eaten,
    pred_reproduction, prey_reproduction,
    grass_growth
  }
  // Final agent properties are written into a file
  save("agents.json");
}
```

Listing 2: A main function from the Predator-prey code.

Agents are typically initialized in a for loop, and are added to the simulation using the `add` function. While adding an agent to the simulation, the OpenABL framework internally keeps track of the agent type, so that agents can be easily selected by agent type. Once the agents are initialized, the main function typically uses the `simulate` statement to invoke a sequence of step functions. In this case, the simulation consist of 13 step functions, repeatedly called for a specified number of time steps.

After simulating all the step functions, a simulation usually ends with a call to the `save` function, which stores the final agent properties into a JSON file.

2.2. Agent Parallelism and Step Functions

Agent parallelism, i.e., the fact that agents' behaviours can be updated independently and in parallel, is an important aspect that allows ABS implementation to reach high performance. Therefore, it is crucial to design a language that implicitly supports agent parallelism and enables its parallel mapping on different target platforms.

OpenABL supports agent parallelism with different semantics. The most important are *step functions*, which take an input agent of some type and returns a (optionally modified) output agent. For instance, prey's movement behaviour is implemented with the following step function:

```
step prey_move(Prey in -> out) { ... }
```

where `in` is an input agent of type `Prey` whose state is the result of the timestep `t-1`, and `out` will be the output agent i.e. the result of the current timestep `t`. For synchronization, the output agents will only become available once the step function has been invoked for all agents of that type. This implicitly suggests a double-buffering agent implementation: a read-only agent buffer for the input; a write-only agent buffer for the output. The strong separation between `in` and `out` produces deterministic, order-independent simulations. Unfortunately, many existing sequential agent libraries (e.g., Mason) do not implement a similar double-buffering mechanism; therefore, their results depend on the agent-update order and are not deterministic.

To invocate a step function, a `simulate` statement is required. A `simulate` statement should specify a list of step functions will be executed in the given order, and a number of timesteps. To preserve the step function ordering, a step function is executed for all agents (of the specified type) before the next step function is executed.

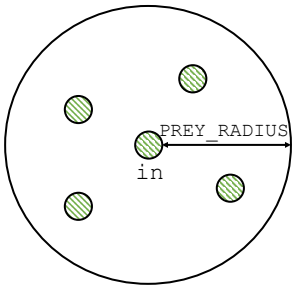
For instance, in the *predator-prey* model (for brevity, we do not list it here), 13 step functions are called: the first, `pred_follow_prey`, on all predator agents; the second, `prey_avoid_pred`, on all prey agents; and so forth until the `grass_growth` step, called on all grass agents.

2.3. Locality and Neighborhood Queries

A fundamental concept of agent-based modeling is locality, because interactions are usually limited to agents in a specified neighborhood. Currently, in OpenABL, the locality is expressed by either a two- or three-dimensional Euclidean topology. Therefore, an agent declaration must have a field, marked with the `position` keyword, either of

type `float2` or `float3`, which indicates the position of the agent for locality queries. The language semantic to query the agents within a radius (the neighborhood) is a combination of `for` loop and `near()` query. For instance, a prey accesses all other prey within a specified radius with the code in Figure 1.

```
step prey_flock(Prey in->out) {
  // ...
  for(Prey py : near(in,PREY_RADIUS)) {
    // ...
    cohesion_velocity += ...;
  }
  // ...
  out.steer += cohesion_velocity;
}
```



Prey py : near(in,PREY_RADIUS)

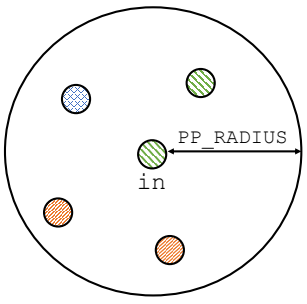
Figure 1: Example of `near()` query with homogeneous agent types, from the prey's flocking behavior.

The agent types used as input and output of the `near()` query are used to support both homogeneous and heterogeneous agent simulations. The OpenABL compiler is in charge of delivering an efficient implementation of the `near()` function, typically using data structures such as a grid [15] or a k-d tree [16].

2.4. Agent Heterogeneity

Heterogeneous simulations are typically implemented with different agent types. Two language semantics provide a support for such simulations: `near()` queries can specify different input and return type (see Figure 2); `simulate` invocation can list step functions operating on different agent types. Language features such as polymorphism and runtime type checks are not supported, because they cannot be easily supported on some backends and, generally, introduce significant overhead in a computationally-intensive part of the code.

```
step prey_avoid_pred(Prey in->out) {
  // ...
  for(Predator pt : near(in,PP_RADIUS))  {
    // ...
    avoid_velocity += ...;
  }
  // ...
  out.steer += avoid_velocity;
}
```



Predator pt : near(in,PP_RADIUS)

Figure 2: Example of `near()` query with heterogeneous agent types, from the prey's fleeing behavior: it iterates all agent of type Predator within the specified radius from the Prey position.

2.5. Dynamic Agent Creation and Removal

Most simulations create a number of agents before any step function is executed. A few of more sophisticated simulations, however, require to dynamically create or remove agents during the step function execution, thus allowing for a dynamically-sized population. For instance in *Predator-Prey*, agents are both created (because of preys and predators reproduction) and removed (when a prey is eaten by a predator). Such simulations require a way to dynamically add and remove agents while the simulation is in progress.

```
step prey_reproduction(Prey in -> out) {
  if (random(1.0) < REPRODUCE_PREY_PROB) {
    add(Prey {
      pos: in.pos, // same position
      dir: -in.dir, steer: -in.steer, // oppos. dir.
      life: in.life/2 // life split btwn parent-child
    });
    out.life = in.life/2;
  }
}
```

Listing 3: Example of probabilistic agent creation with `add()`, from prey's reproduction behavior. The newly created prey has the same position of the father but opposite direction; the life is split between father and child.

To support this feature on as many backends as possible, the language enforces an additional constraint: during the execution of a step function, at most one new agent may be added. This limit is particularly useful to decrease the complexity of GPU implementations (e.g., FlameGPU has a similar limitation). In the first release of OpenABL [14], an additional constraint was that the position of the new agent must be the same as the input position of the current agent (e.g., `in.pos`). Such constraint was important for distributed implementations, where creating an agent on a different position may actually mean to physically allocate that agent on another distributed node, therefore requiring extra communication. However, in this updated version, we have been working together with the developer of D-Mason to solve this issue, thus this language constraint has been removed.

2.6. Sequential Step Functions and Reduction

Step functions have an inherently parallel semantic: they are executed for each agent of a specified type. However, there are cases where it is necessary to perform a sequential task, e.g., to collect statistical information about the whole simulation or aggregate statistics about all agents of a given type.

To overcome this problem, we have introduced a new type of step function, called sequential step function, that is executed only once, contrary to normal step function that are executed once for each agent of a specified type. Sequential step function can be used as parameter in a simulate statement, in the same way of other step functions. We have also introduced an early support of reduction operation, which currently can only be used within a sequential step. The following code illustrate an example of both semantics:

```
sequential step gather_stats() {
  int num_prey = count(Prey);
```

```
    int num_predator = count(Predator);
    int num_avail_grass = sum(Grass.avail);
    float exec_time = getLastExecTime();
    log_csv(num_prey, num_predator, num_avail_grass, exec_time);
}
```

2.7. Language Limitations

Currently, the language only supports Euclidean topologies. While some extensions, such as a support for toroidal topologies, would be straightforward, more general cases like network-based interactions do not fit well into the current language design. In particular, the support of graph-based topologies (i.e., network of agents) poses great challenges in terms of parallel and distributed implementations. Network-based simulations are often based on complex algorithms such as betweenness centrality, which requires complex techniques to scale [17, 18] and are currently not supported by any of the ABS backends.

OpenABL does not support event-driven simulations: the language exploits a conservative synchronization approach to ensure a consistent integration of distributed simulations. Unfortunately, conservative synchronization does not map easily onto event-driven simulations.

3. Compilation Infrastructure

A compiler implementing the OpenABL language can exploit the high-level information provided by language to generate efficient code, to take advantage of the inherent parallelism, and to applying a sequence of code transformations enabled by the simplified computational model. This work also provides an OpenABL compiler implementation that shows how high-level semantic can be easily mapped into parallel and distributed architectures, ranging from multi-core CPU and GPUs to cloud-based systems.

This section illustrates the compilation workflow of our implementation, but the language design is generic enough to allow other programmers to implement their own compiler. Within our compilation framework, it is also possible to implement new backends supporting a specific target with native and customized high-performance implementations.

3.1. Compilation Workflow

Figure 3 shows the compilation infrastructure of the OpenABL compiler, which follows a classical source-to-source compilation approach. Starting from an `abl` input script, the lexical analysis (based on Flex) and the syntax analysis (based on Bison) translate the source code into an *abstract syntax tree* (AST), which is the main intermediate representation. The AST encapsulates language semantic such as simulate invocation and step function with a specific node type. The analyses are performed to validate and enforce the language semantics, and to annotate the syntax tree with type information and dependency.

After this step, the backend infrastructure is responsible of translating the IR through code generation. This comprises a set of (pluggable) backends generate the output source code (and other required files) for the target platform. Depending on the target platform, a backend may decide specific way to implement agent parallelism and new query.

Thee next sections provide details of the single step involved in the process.

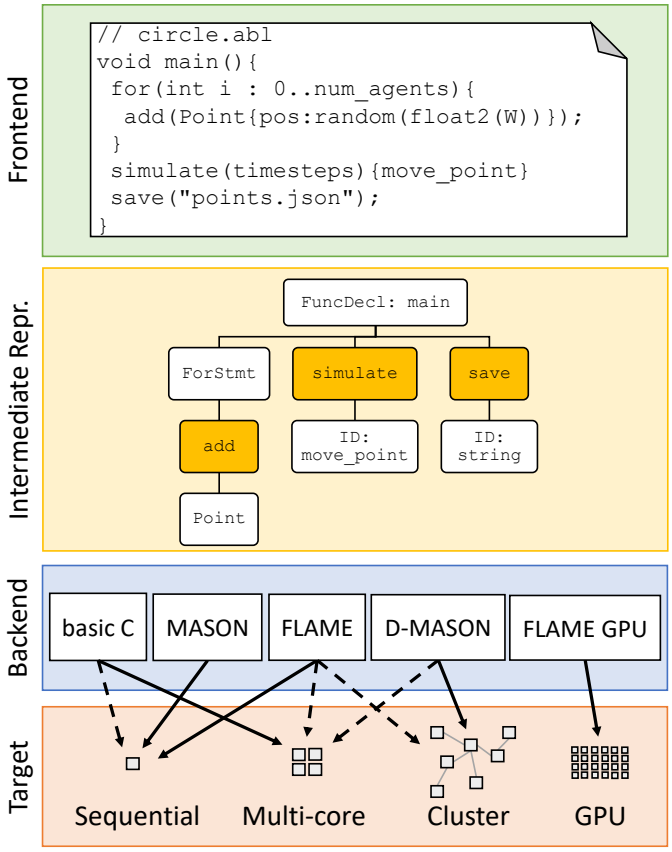


Figure 3: The OpenABL compilation workflow. Top-to-bottom, the `.abl` input code is translated into an AST-based intermediate representation, from which different backends generate code for specific platforms. Both code and IR show an excerpt of the circle test benchmark, with highlighted domain-specific semantics. Bold lines are backend-target combinations evaluated in this article.

3.2. Parsing and Checking

The frontend is responsible of parsing and checking the validity of the input `.abl` script. At first, it perform lexical and syntax analysis; this part is implement with Flex and Bison, and returns an intermediate representation of the code. Once the AST is built, an analysis pass visits the AST, performs type checking and generic checks such as array initialization and initialization of global constant with constant expression.

In addition, the analysis pass also verifies the validity of the domain-specific semantics. This includes a large list of checks that applies at different levels, e.g.: step functions (the proper use of `in` and `out` values, sequential step function semantic, cannot directly call step function without a `simulate` keyword), for-near loops (the specified type should be an agent, multiple loops in a single step function cannot use them, loop on agent without position member), agent (redefinition of agent, unknown agent type, agent without members), parameter declaration (must be assigned to constants), environment declaration (there can be only one single specification, environment member must be a constant

expression, min and max bound must be `float2` or `float3`, granularity must be a number, cannot access environment prior to its declaration, unknown environment member), simulate (a script can only contain a single simulate statement), timestep (number of timesteps must be an integer) add/remove (`removeCurrent()` can only be used inside a step function, argument of `add()` must be an agent-creation expression).

3.3. Intermediate Representation

After semantical check, the resulting AST is used as *intermediate representation* (IR) for the compiler. The IR is similar to an AST generated for the C99 language. However, it adds special domain-specific nodes for domain specific semantics. Example of those nodes are `AgentDeclaration` and `Param`. `SimulateStatement` and `step` functions are particularly important fir efficient code generation, as they represent the key semantics to express agent parallelism in the IR.

OpenABL also provides a small transformation framework that is capable to perform high-level code transformations on the AST. Similarly to other source-to-source compiler such (e.g., Rose [19]) AST transformations are designed to enable optimizations at IR level; however, unlike Rose, it does not need preprocessor directive or other code annotations, as important language features such as agent parallelism and near query are already specified in the domain-specific input code, and kept as specific node in the AST. An example of transformation is *step fusion*: two steps operating on the same agents type, after a dependency check, can be potentially fused in a single one. However, this part is still experimental and is not discussed in this article.

The IR is later processed by one of the available backends, which performs a traversal of the tree and prints the appropriate target code.

3.4. Parallel Backends

OpenABL currently supports five backend implementations targeting different platforms, i.e., ABS libraries and their supported spatial data structures. The backend implementation is in charge of translating the IR semantics into efficient target code, including mapping and scheduling for those backends that support parallelism.

3.4.1. Simple reference backend

A first, simpler backend implemented in basic C provides a reference implementation for other backends. This backend is trivially parallelized with OpenMP using `#pragma omp parallel for`.

It does not use any spatial data structures, thus any near query has quadratic complexity with the the number of agents. This backend implements a trivial double-buffering mechanism with two agent arrays swapped after each timestep.

3.4.2. Flame

The *Flame* [10] backend is based on state machines supporting sending messages between agents. Therefore, a step function is implemented to modify the current agent memory, send messages to other agents, and iterate over messages received in the previous step. OpenABL neighborhood lookups are supported as follows: first, inside each for-near loop, we calculate which members of neighboring agents are used; thus, we generate one function that sends a message containing all of those members. A second step

function loops over the messages sent and identifies the ones falling into the specified neighborhood. As a side-effect, no explicit double buffering is necessary: because messages are sent in a previous timestep, and changes to agents are performed in the current step. Flame models consist of three parts: (1) an XML model specification, which determines the agent types, their members and states, as well as step functions and messages sent or received by them; (2) a function file containing the C code of the step functions; (3) an XML file containing the initial state of the agents. Flame does not provide any means to execute our `main` agent initialization code; for this reason, OpenABL generates additional code based on the custom C backend, which executes the initialization code, exports the agent state into Flame's XML format and then invokes Flame to perform the actual simulation. Unfortunately, adding or removing agents at runtime is not supported in Flame.

FLAME supports an MPI-based parallelization method that can theoretically support multi-node parallelization, as well as multi-core parallelization by mapping each MPI process to a core.

However, FLAME does not have the concept of spatial messaging, i.e. does not have a spatial data structure that efficiently implement near agents query such as kd-tree or grid. Instead, it uses the message board library (libmboard) to filter messages between nodes on an MPI cluster. This is good for large models with communication between different agent types but no so good for large homogeneous populations which only require communication over short spatial distances (i.e., the number of messages is quadratic with the number of agents). For these scalability issues, FLAME cannot be realistically used on the complex and large simulations evaluated in this article, therefore we only tested FLAME for the single node scenarios.

3.4.3. FlameGPU for NVIDIA GPU

FlameGPU [15] is an extension of the Flame, which targets GPU through an implementation based on CUDA (i.e., each agent is mapped on a CUDA thread). As such, the FlameGPU backend is similar to the one targeting Flame. As a spatial data structure supporting *near* queries, FlameGPU provides a grid, which additionally requires an XML model with partitioning radius and the environment bounds specification (those must be adjusted upwards so that they are multiples of the radius). Agents removal and addition at runtime is supported by Flame GPU; however, it comes with one limitation: only the current agent can be removed and only one agent can be added per step function. The OpenABL language design enforces both restrictions. It should be noted that, even if the collaborative use of CPU and GPU can be supported by the OpenABL language and IR, this backend completely offload agents' computation to the GPU and therefore does not support CPU-GPU collaborative computation. Additionally, this would also require the allocation and mapping of tasks based on the computing characteristics of heterogeneous processors and careful load balancing, as suggested by recent research results [20, 21, 22, 23].

3.4.4. Mason

Mason [24] is a Java library implementing the simulation and visualization of agent models. A Mason simulation is composed by an environment, which supports neighborhood queries based on a grid data structure, and a schedule, which executes step functions at given times. There are two issues on supporting our language semantics on

Mason: Firstly, Mason Mason simulations are fundamentally order-dependent, i.e., no double-buffering. To enforce our deterministic, order-independent semantic, each agents stores two states (i.e., old and new), which are swapped at the end of each step function. Secondly, in idiomatic Mason code, only a single step function is provided for each agent, while our execution model support a list of step functions[3]. This issue is solved with a cyclic counter for each agent, which selects the step function to execute. As such, the number of timesteps for a Mason simulation has to be multiplied by the number of step functions. Dynamic agents addition and removal is already supported in Mason: if an agent is removed, all agents are rescheduled. In addition, the backend for Mason generates extra code for the visualization of the simulation. Regarding parallelism, Mason is limited to a single thread; however, the D-Mason backend provides a parallel (and distributed) implementation.

3.4.5. D-Mason for multi-core and cluster

D-Mason [25, 26] is a Mason extension allowing the distribution of the simulation across distributed nodes. The master-workers paradigm is used for space partitioning: a master node partitions the simulation environment into regions, each region contains a set of agents, and is assigned to a node, with each node being in charge of simulating the agents that belong to the assigned region. Extra work is required by the node to handle agents migration and managing the synchronization between neighboring regions. D-Mason supports different communication layers [27]: (1) pureActiveMQ, which uses Apache ActiveMQ as message broker for both the management and the synchronization messages; (2) pureMPI, which is based on an MPI-based decentralized communication strategy; (3) hybrid, which uses a combination of the two modes depending on the communication type used. While OpenABL support all three modes, in this article the nodes communicate using the first approach, e.g., with a publish-subscribe pattern: a multicast channel is assigned to each region; the nodes subscribe to the topics associated with neighboring regions to receive message updates. D-Mason implements a static partitioning policy. In OpenABL the agent initialization occurs sequentially (in the `main` function), while in D-Mason the initialization is distributed. This is supported by running the same initialization code on all worker nodes, but discarding all the agents whose position does not fall into the assigned region. D-Mason imposes a few additional restrictions over Mason: the environment definition only support positive coordinates; the handling of environment bounds is more restricted than in other backends (e.g., agents may not be placed exactly on the environment boundary). Finally, dynamic agents removal and addition is supported; however, added agents must be positioned in the current space partition. This is the reason why OpenABL requires the position of the added agent to be the same as that of the creating agent.

3.4.6. Differing Backend Capabilities

We already mentioned that certain parts of the OpenABL language are not supported by all backends. For example, Flame does not support dynamic additional/removal of agents. While this makes the language not fully portable, we want to avoid making the

[3]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.

language overly restrictive for the sake of a single backend. A set of compilation flags are provided to support backend-specific configurations. E.g., by default, the `float` data-type is mapped onto double-precision floating point (since is the only fp type supported by all backends). However, a specific compiler flag allows Flame and FlameGPU to adopt single-precision for floats.

4. Experimental Evaluation

In order to evaluate OpenABL a collections of seven applications, coming from different fields and described in the next section (4.1) has been deployed. We have evaluated both the programmability of the language (4.2) and the performance of the code generated for all the considered backends, including single-node performance on CPUs and GPUs (4.3). In Section 4.4 we provide the analysis of OpenABL scalability on a cluster while in Section 4.5 we present a comparison against manually tuned target-specific code.

4.1. Reference Simulation Models

The evaluation is based on seven agent-based models from different domains. The choice of the models was due to the fact that, for each of them, at list a reference implementations was available for at least one of our targets. Table 1 depicts some properties of these models, such us the number of different agent types, the number of step functions required to implement the simulations and whether addition/removal of agent is required for the considered model.

Circle is a baseline benchmark behavior for assessing the performance of fixed-radius near neighbor look-ups. Circle is part of the OpenAB [13] initiative. A formal definition is given by Chisholm et al. [28] and a reference implementation is available in FlameGPU. This benchmark only has a single agent type (Point) and a step function, which combines a short-range repulsion force with an attractive force operating at a larger radius (see reference implementation [29]).

Boids [30] is a steering behavior, originally developed to replicate the flocking of birds, which enable the simulation of an entire group of autonomous characters in animation and games, by exploiting only few simple rules. The agent behaviour is obtained by the combination of three components: separation, to avoid near flockmates; alignment, to match the average direction of local flockmates; cohesion, to move toward the average position (center of mass) of local flockmates. The behaviour is implemented by one agent type and one step function, which performs a near query for the interaction and apply the three rules based on agents' distance (see reference implementation [31]).

Conway's *game-of-life* [32] is a well-known cellular automaton model, developed on top of a grid of boolean square cells. Each cell represents an agent, which at each simulation step, can be *dead* or *alive*, according to the status of neighbors cells at the preceding simulation step (see reference implementation [33]).

Sugarscape is a social science model introduced to Epstein and Axtell [3], where agents move on a grid. Some unequally distributed, regrowing resources (sugar), are available on the field and the goals of agents is to consume in order to survive. Similarly to *game-of-life*, *Sugarscape* is based on a stationary grid of square cells. It is implemented by an agent type with many fields including sugar level, metabolism and other stat values, and for step functions implementing the four sugarscape behaviours: (1) metabolise and

growback, (2) movement request, (3) movement response and (4) movement transaction. Our implementation is strictly following the original model, where the each grid position is only occupied by one agent and necessitates a negotiation process. (see reference implementation [34]).

The *Ants Foraging* model simulates the behaviour of ants. Ants are continuously looking for a food. Once they discover a food source, they establish a trail of pheromones between the nest and the food source in order to indicate to the other ants the food-to-nest and nest-to-food paths. The model uses two pheromones, one for each path and simulates the evaporation of pheromones after some simulation steps. The OpenABL implementation is based on the existing Mason code implementing Panait and Luke's model [35]. The original (sequential) implementation is based on the access to global data structures, which clearly is not suitable for scope. We solve this by defining two agent types: one that emulate the ants, and a grid of pheromone agents that emulate the deposition and evaporation of pheromones. The two pheromones are actually implemented as a single Pheromone agent with two different values for the home and food trails. Overall, it includes three step functions: (1) `ant_deposit` (agent step), where each ants looks for the nearby home and food pheromone, and pheromones are only read; (2) `pheromone_deposit` (pheromone step), where it is implemented the actual deposit of home and food pheromones, and each pheromone is written/updated; (3) `ant_act` (ant step), which implement ants move according to the nearby pheromones, or alternatively does a random move. (see reference implementation [36]).

Predator-Prey [37] is a more complex model, which involves three different agent types (prey, predator and grass) and 13 step functions. Moreover it requires dynamic agent creation and removal . In this model agents implements different set behaviours, e.g., predators and prey implement collision avoidance, flocking, and reproduction. The goal of each predator is to reach and eat the closest prey. Preys try to escape from predators and at the same time eat grass to survive. As can be seen in the reference implementation [38], the simulate loop calls the step function int he following order:

```
simulate(num_timesteps) {
    pred_follow_prey, prey_avoid_pred,
    prey_flock, pred_avoid, prey_move, pred_move,
    prey_eat_or_starve, pred_eat_or_starve,
    grass_eaten, prey_eaten,
    pred_reproduction, prey_reproduction, grass_growth,
    gather_stats
}
```

As seen in Section 2, this simulation contains many heterogeneous behaviours (follow and avoid steps), agent creation (pred and prey reproduction), and removal (eat or starve). Here, we additionally added one exemplary step function for statistics gathering.

Keratinocyte Cellular biologist use agent model to simulate and understand the rules of keratinocyte colony formation. This approach has been originally proposed by Sun et al. [8] and has been first implemented into FlameGPU [15]. Our implementation [39] follows FlameGPU's code, and is made of only one agent type with 13 fields modeling bonds, motility and movements, and six step functions. The steps implement the biological functions such as life cycle, cell differentiation, death signal, migration, and force resolution.

All the considered simulations have been executed on several backends (namely, basic

C, Mason, D-Mason, Flame and FlameGPU) varying some simulation parameters, such as for instance the number of agents. The environment size is scaled so that the agent density remains constant (e.g., the number of agents is the square root of the environment for two-dimensional simulations).

Application	Types/Steps/AR	OpenABL/FlameGPU/D-Mason
circle	1 / 1 / -	36 / 184 (×5.1) / 537 (×14.9)
boids	1 / 1 / -	82 / 240 (×2.9) / 767 (×9.4)
game of life	1 / 1 / -	48 / 133 (×2.8) / 477 (×9.9)
sugarscape	1 / 4 / -	154 / 345 (×2.2) / n.a.
ants foraging	2 / 3 / -	191 / n.a. / 967 (×5.1)
predator-prey	3 / 13 / AR	248 / 858 (×3.5) / n.a.
keratinocyte	1 / 6 / AR	306 / 1172 (×3.8) / n.a.

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

4.2. 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 from FlameGPU and D-Mason.[4] As depicted in the right side of Table 1, the FlameGPU implementations are 2-5 times larger, while the D-Mason models are 5-15 times larger. We acknowledge that eLOC is not a very reliable measure of programmability, but on the other hand the data clearly shows that OpenABL models are significantly more compact than manual implementations.

4.3. Single-node Performance Comparison

We deployed and analyzed the effectiveness of the code generated by the OpenABL compiler for the basic C, Mason, Flame and FlameGPU backends. The seven models were simulated for 100 timesteps varying the number of agents from 250 to 10^6. We notice that the *Predator-prey* and *keratinocyte* models have been evaluated only on backends which supports the the addition and removal of agents at execution time. Single-node Performance benchmarks have been performed on a machine equipped with an Intel Core i5-4690K CPU (4 cores at 3.50GHz), 16GB of memory, running Ubuntu 16.04. The basic C backend has been configured to exploit multiple threads using OpenMP. The FlameGPU backend has been evaluated on an NVIDIA Titan Xp (Pascal architecture) having 12GB of memory.

Figure 4 shows the performance comparison. Both Flame and basic C scale quadratically with respect to the number of agents. This was expected, since they do not use data structure, which optimizes neighbourhood queries. Mason provides better performances compared to Flame, and seems to be the best implementation to simulate a small

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

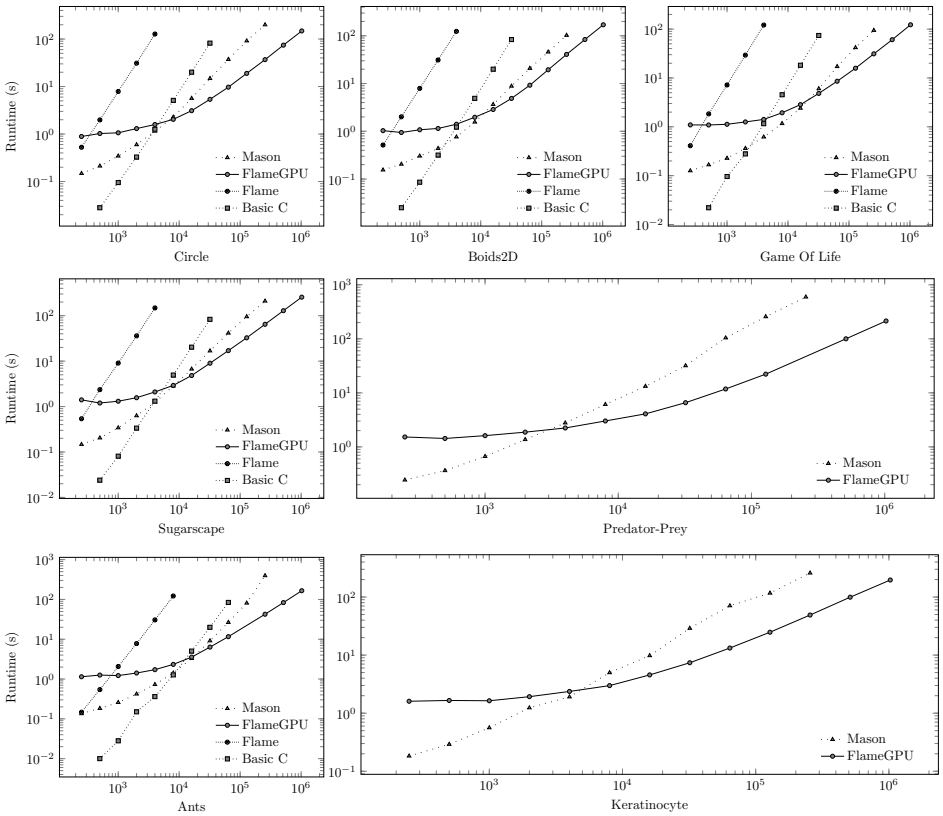


Figure 4: Performance (y-axis, in sec) of OpenABL-generated codes for Mason, FlameGPU, Flame and basic C, with a growing number of agents (x-axis, in log scale). More details are shown for the most complex predator-prey and keratinocyte, which have dynamic agents creation and removal.

number of agents. Indeed, FlameGPU leads to a high overhead for small-sized simulations, due to the data transfer to the GPU. On the other hand FlameGPU provides the best-performing solution for simulations having a larger number of agents($> 10^4$ agents). For the models that have a balanced distribution of agent on the space, both Mason and FlameGPU scale roughly linearly varying the size of the population. One notable exception is *ants*, where the performance of Mason degenerates to a quadratic behavior, because the *ants* simulation starts with all the agent is a specific position (the nest) and this unbalanced distribution harms the performance of Mason.

4.4. Cluster Scaling

To evaluate the scalability of the OpenABL D-Mason backend, several simulations have been executed on a cluster of 12 nodes equipped with two Intel Xeon E5-2430 (six cores) with hyper-threading disabled and connected by a Gigabit network. D-Mason exploits a master-workers paradigm and therefore a specific node has been used to coordinate the simulation, while the others provides the computation power. Each worker allocates one D-Mason logical processor, for each available core, running on Oracle JVM

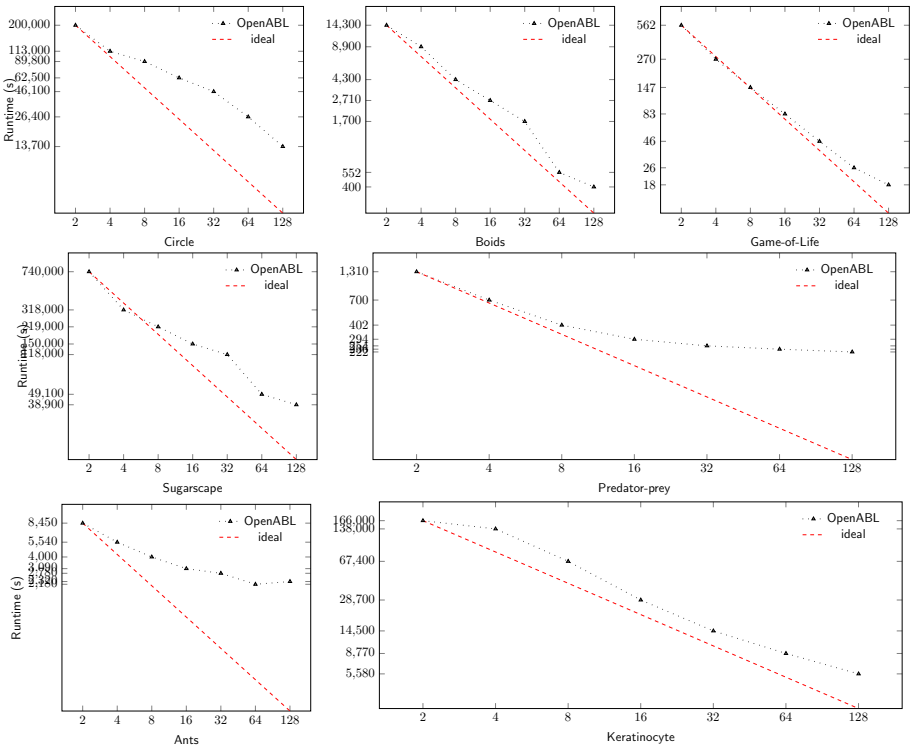


Figure 5: OpenABL D-Mason strong scaling with different number of cores (x-axis) on seven test benchmarks. More details are shown for the more complex simulations with agents removal and creation.

1.8. D-Mason exploits Apache ActiveMQ as message broker for the communication among logical processors. The broker is allocated on the master node. Figure 5 depicts the strong scalability of the seven models. Each model simulates 10^6 agents for 1000 timesteps. The plots show the runtime in seconds varying the number of logical processors (cores) involved.

The results show that the *boids*, *game-of-life* and *sugarscape* models exhibits good scalability. Despite having similar behavior, *Circle*'s scaling is slightly worse due to different parametrization. In fact, the larger interaction radius largely extends the amount of agents falling in the halo zone, which implies that a large amount of agents need to be transferred between nearby nodes after each step function.

On the other hand, *ants* and *predator-prey* do not scale very well. In both simulations, we have a load balancing problem arising from the highly dynamic behaviour of the agents, which unfortunately impacts the D-Mason static partitioning strategy.

In the *ants* model, in terms of agents distribution, the simulation starts with a dense concentration of the ant agents near the home; then, ants randomly and evenly spread on random walks; finally, after a pheromone trail is established, all ants follow the same path between home and food. These three distribution patterns result in an uneven and dynamic distribution of the workload, which cannot be handled efficiently by a static partitioning policy.

In *Predator-prey*, the dynamic behaviour is mostly due to the addition and removal of agents. When a predator agent enters an area with many prey agents, a large number of prey agents are removed (predator eats prey). In addition, prey agents also move in different areas and flee from predators. This results on sudden and dynamic decrease of the agents. Clearly, more advanced load balancing strategies may substantially improve this aspect [40].

Despite being multi-step, *keratinocyte* shows a very good scaling; this because cell agents are distributed evenly, are not too dynamic, and the agent population grows slowly.

4.5. Comparison Against Manually-tuned Code

To evaluate any potential overhead, we compared the OpenABL-generated codes against manual implementations of the same test benchmark. Unfortunately, most codes are not available on all ABS libraries. We selected the *boids* model for this comparison, because it is well-known (most libraries provide an implementation), is simple to validate, and scales easily with the number of agents.

We omitted Flame from the comparison because of its very poor scalability, as shown in Section 4.3, which makes it impractical for simulations with more than 5000 agents.

The code generated for Mason is 9% slower than the manual implementation written directly in Mason. The reason for this overhead is the double buffering mechanism introduced by OpenABL to ensure order-independent correctness, not support in standard Mason.

For FlameGPU, the performance of the code generated by OpenABL has similar performance than the manual implementation directly using FlameGPU: the semantics of the language map very well in the GPU model, without any noticeable overhead.

The overhead analysis of D-Mason shows some knowns challenges of ABS on distributed systems: the overhead over a the native implementation is 30%. We found that the major reason is the synchronization mechanism for each step function. An improvement of the synchronization mechanism of the step function may potentially reduce such overhead. In complex multi-step applications, in particular, a read-write buffer analysis that avoids unnecessary synchronizations may be helpful to reduce the communication of the synchronization of each timestep.

4.6. Analysis of Complex Simulations

Complex simulations are particularly challenging for parallel ABS implementations. In this context, we assume that the complexity is related to multiple factors. First, the number of step functions: we have seen how *circle* (1 step) scales better than *predator-prey* (13 steps), in Figure 5. A second aspect is heterogeneity: simulations with multiple agent types may expose a different granularity of parallelism, depending of the agent type. For instance, a *predator-prey* simulation may have thousands of preys but only a few dozens of predators. Finally, an important aspect is the possibility to dynamically create or remove agents in the simulation. E.g., the number of preys and predators grows and shrinks over the time, exhibiting a cyclic behavior.

The evaluate how complex simulations behave on different OpenABL backends, we took as reference *predator-prey* and *keratinocyte*. In particular, we analyzed how the simulations evolve over the time (among step and timestep), how the number of agents

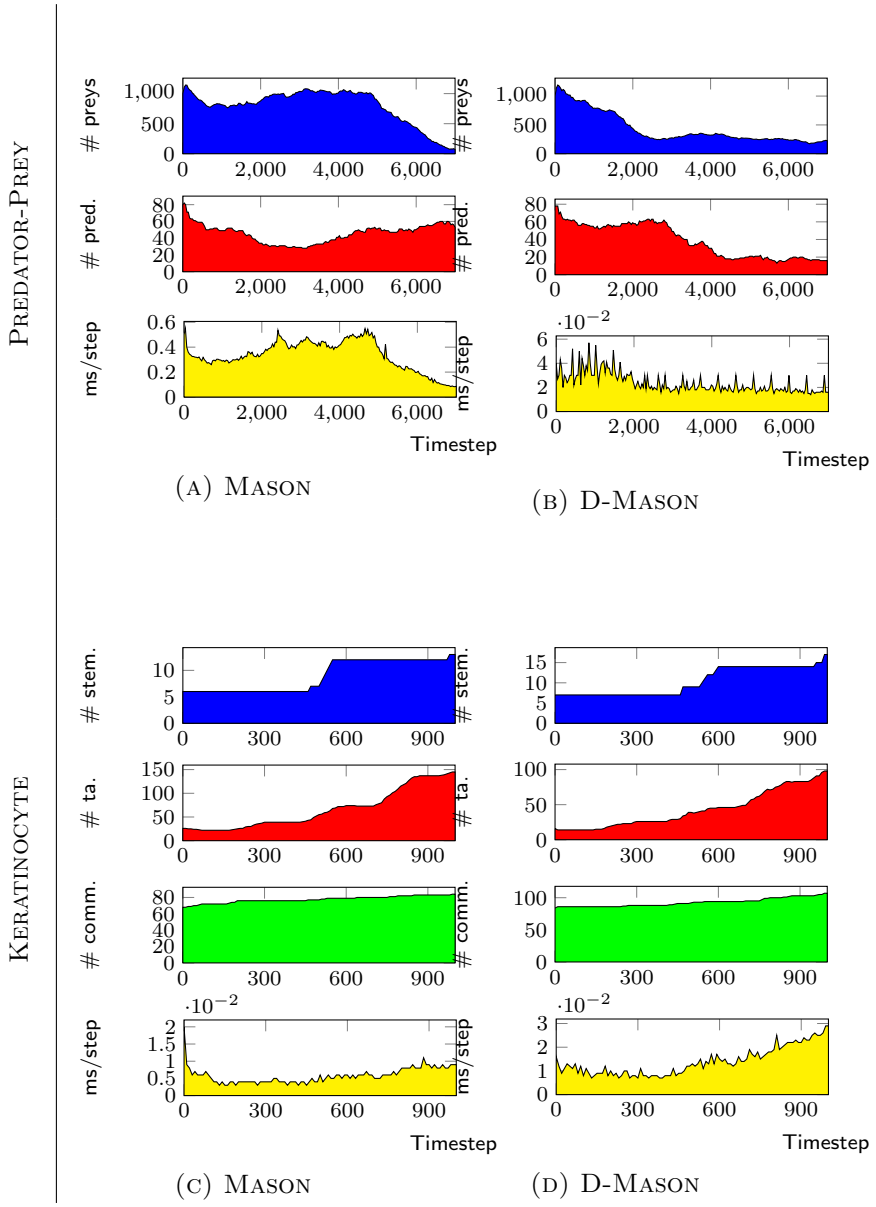


Table 2: Analysis of two complex simulations on two backends. For all simulations, time series with the performance/step and the number of agents are presented. In *predator-prey*, the number of preys and predators are shown (grass cell are omitted because constant over the time). In *keratinocyte*, we show the three used types of keratinocyte cell agent, i.e., ta, comm and stem.

varies, and how this affects the performance/step. In Table 2 we show a selection of four simulations on two backends: the simplest Mason, and the most complex D-Mason.

In `(A)`, we run the *predator-prey* simulation starts with about 3200 agents on Mason:

1000 predators, 80 preys, and 2120 grass (the number of grass agents is fixed, but they may be marked as *eaten*; for this reason we do not show them in the figure). After predators eating many preys, their number drastically drops down. After some steps, the scarcity of preys also reduces the number of predators. This cyclic behavior may end up in a situation where all preys dye, reducing the total number of agents to zero. This dynamic behavior does not significantly affect the performance of the Mason sequential backend, as the runtime/step is linear with the number of agents.

However, in distributed backends such as D-Mason (B), this dynamic behavior incurs in extra communication overhead. The performance/step analysis shows several peaks, which corresponds to communication overhead associated to the highly-dynamic nature of the simulation.

The *keratinocyte* simulation, instead, has a different behavior. Instead of a cyclic increase/decrease of agents, the total number of agents tends to gradually increase; also, cells are rather stationary and their *motility* is low. This more predictable behavior also affects the performance of the parallel backends. In (C) and (D) we show the number of agents and runtime per step for two simulations running on D-Mason/OpenABL and FlameGPU/OpenABL. In this case, both backends have a similar behavior the few observed peaks are related to the cost of creating new cell agents, rather than communication cost due to excessive agent movement.

5. Related Work

Many researchers have investigated the different aspects of agent-based modeling, from the development of more sophisticated models to the engineering of trustworthy simulations [41]. This section focuses on the research investigating specifically parallel ABSs and DSLs.

5.1. Parallel Agent-Based Simulations

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 [42] is an agent-based simulation toolkit written in C++, later extended and parallelized into the REPAST-HPC framework [43], and tested on a Blue Gene/P HPC cluster. Cosenza et al. [11] introduced a distributed load balancing schema for parallel ABS that scales a simulation with one million agents on a cluster with 64 processors. Mason [24] is a popular multi-agent simulation library written in Java. D-Mason [25, 26] provides an effective and efficient way of parallelizing Mason programs for distributed systems, handling communication strategies and load balancing [44]. D-Mason has been tested also on Amazon Web Services [44], and used on several social science scenarios [45]. Flame [10] is an agent-based environment based on an underlying formal model, called the X-Machine, and used in various scenarios such as cell simulations [46] and immune system modeling [47]. FlameGPU [15] extends Flame enabling the execution of agent-based models on GPU architectures. Other GPU implementations have focused on bio-inspired visual clustering [48] and on efficient compression of agent direction [49]. Piccione et al. [50] presented an API for Parallel Discrete Event Simulations.

The variety of implementations has led researchers to investigate and compare their functionalities and performance. Macal and North [51] identify methods and toolkits for

ABS. Berryman [52] conducted a broad review of sequential ABS toolkits: BactoWars, EINSTein, MANA, Mason, NetLogo, Repast, Swarm and WISDOM-II. Rousset et al. [53] provided a comparison among parallel and distributed ABS toolkits. This comparison has been performed at two levels: a qualitative analysis of the provided functionality, showing a deep heterogeneity among the considered platforms; and a quantitative performance evaluation based on a reference model implementation. We notice that this quantitative evaluation is strongly influenced by the approach used to implement the model. Our methodology here is different, we provide here a unique ABM programming environment, while the choice for the best approach to implement the model on each specific backend is deferred to the compiler.

5.2. Domain-Specific Languages and Parallelism

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 [54].

In the following, we report the description of some interesting DSLs that exploit parallelism at different levels: PATUS [55] is a code generation and auto-tuning framework for stencil computations targeting modern multi- and many-core processors; PATUS's stencil specification language is independent of hardware-specific details and has been extended with machine learning-based autotuning methods [56]. Green-Marl [57] is a DSL whose high-level language constructs allow developers to describe their graph analysis algorithms intuitively, but expose the data-level parallelism inherent in the algorithms. Halide [58] is a language and optimizing compiler for image processing pipelines. Recently, Halide has been extended to support distributed memory systems [59]. *PIPES* [60] is a macro-dataflow programming environment for distributed-memory clusters, based on the Intel Concurrent Collections (CnC) runtime. The Refactoring Pattern Language (*RPL*) [61] is a high-level domain-specific language that represents the parallel structure of an application and generates semantically equivalent, parallelizations targeting OpenMP, Intel TBB and FastFlow. Rajbhandari et al. [62] present a layered domain-specific compiler to support *MADNESS* (Multiresolution ADaptive Numerical Environment for Scientific Simulation), a high-level software environment for the solution of integral and differential equations in many dimensions, using adaptive and fast harmonic analysis methods with guaranteed precision. Vasista et al. [63] introduces a DSL based on the PolyMage framework for geometric multigrid methods. Similarly, Karol et al. [64] shows a DSL for parallel particle methods.

The most similar to our work is Liszt [65], which provide a DSL for constructing mesh-based PDE solvers able of targeting clusters, SMPs and GPUs. Liszt applications perform within 12% of hand-written C++ code and scale to large clusters. To the best of our knowledge, no similar DSL has been defined for devising ABSs exploiting parallelism provided by different backends.

6. Conclusion and Future Works

We present OpenABL, a framework consisting of a domain-specific language designed for agent modeling, and a compiler implementation that maps input codes into high-performance parallel and distributed architectures.

The OpenABL language is a DSL that relies on high-level abstractions for programmability exploiting explicitly agent parallelism to deliver high-performance. OpenABL supports a wide range of context-specific features such as order-independent step functions, neighborhood queries, heterogeneous agents, and dynamic agent addition and removal. OpenABL language has been implemented as a source-to-source compiler, which translates the input OpenABL code into an AST-based intermediate representation exposing parallelism, locality and synchronization at the agent level. This intermediate representation is then used by a collection of pluggable backends, which generate target codes for different target platforms such as multi-core CPUs, massively parallel GPUs, large clusters and cloud systems. The proposed framework has been evaluated on a collection of seven applications from various fields: micro-benchmarks (circle), animation (boids), animal ecology (ants foraging and predator-prey), cellular automata (game of life), social science (sugarscape) and biology (keratinocyte). Results shows that a program written in OpenABL is much compact than one written for non-portable platform-specific libraries while 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.

OpenABL opens to new interesting research directions. On the language side, new language features may help scientists on easily designing their simulations. The compiler can support optimizing code transformation that improve the performance of the generated code. In addition, code generation can be extended to support alternative architectures, or more efficient native implementation that bypass current agent-based modeling libraries.

Acknowledgments

This research has been supported 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] J. M. Epstein, Agent-based computational models and generative social science, Complexity 4 (5) (1999) 41–60.

[2] K. Kollman, J. H. Miller, S. E. Page, Adaptive parties in spatial elections, American Political Science Review 86 (4) (1992) 929–937.

[3] J. M. Epstein, R. Axtell, Growing Artificial Societies: Social Science from the Bottom Up, The Brookings Institution, 1996.

[4] P. Bak, M. Paczuski, M. Shubik, Price variations in a stock market with many agents, Physica A: Statistical Mechanics and its Applic. 246 (3-4) (1997) 430–453.

[5] G. Topa, Social interactions, local spillovers and unemployment, The Review of Economic Studies 68 (2) (2001) 261–295.

[6] T. Haynes, S. Sen, Evolving behavioral strategies in predators and prey, in: Int. Joint Conf. on Artificial Intelligence, 1995, pp. 113–126.

[7] N. Pelechano, J. M. Allbeck, N. I. Badler, Controlling individual agents in high-density crowd simulation, in: EG Symp. on Computer Animation, 2007, pp. 99–108.

[8] T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood, S. MacNeil, An integrated systems biology approach to understanding the rules of keratinocyte colony formation, Journal of the Royal Society Interface 4 (17) (2007) 1077–1092.

[9] R. Narain, A. Golas, S. Curtis, M. C. Lin, Aggregate dynamics for dense crowd simulation, ACM Trans. Graph. 28 (5) (2009) 122:1–122:8. doi:10.1145/1618452.1618468.

[10] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, C. Greenough, Flame: Simulating large populations of agents on parallel hardware architectures, in: Conf. on Autonomous Agents and Multiagent Systems, 2010, pp. 1633–1636.

[11] B. Cosenza, G. Cordasco, R. D. Chiara, V. Scarano, Distributed load balancing for parallel agent-based simulations, in: Int. Euromicro Conf. on Parallel, Distributed and Network-based Processing, PDP, 2011, pp. 62–69. doi:10.1109/PDP.2011.22.

[12] M. Carillo, G. Cordasco, F. Serrapica, C. Spagnuolo, P. Szufel, L. Vicidomini, D-Mason on the Cloud: An Experience with Amazon Web Services, in: Euro-Par Parallel Processing Workshops, PADABS, 2016, pp. 322–333. doi:10.1007/978-3-319-58943-5_26.

[13] OpenAB Members, The OpenAB Initiative (2015).
URL http://www.openab.org

[14] B. Cosenza, N. Popov, B. Juurlink, P. Richmond, M. K. Chimeh, C. Spagnuolo, G. Cordasco, V. Scarano, Openabl: A domain-specific language for parallel and distributed agent-based simulations, in: International European Conference on Parallel and Distributed Computing (Euro-Par), 2018.

[15] P. Richmond, D. Walker, S. Coakley, D. Romano, High performance cellular level agent-based simulation with Flame for the GPU, Briefings in Bioinformatics 11 (3) (2010) 334. doi:10.1093/bib/bbp073.

[16] K. Kofler, D. Steinhauser, B. Cosenza, I. Grasso, S. Schindler, T. Fahringer, Kd-tree based n-body simulations with volume-mass heuristic on the GPU, in: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 1256–1265. doi:10.1109/IPDPSW.2014.141.

[17] E. Solomonik, M. Besta, F. Vella, T. Hoefler, Scaling betweenness centrality using communication-efficient sparse matrix multiplication, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017, 2017, pp. 47:1–47:14. doi:10.1145/3126908.3126971.

[18] M. Bernaschi, G. Carbone, F. Vella, Scalable betweenness centrality on multi-gpu systems, in: Proceedings of the ACM International Conference on Computing Frontiers, CF'16, Como, Italy, May 16-19, 2016, 2016, pp. 29–36. doi:10.1145/2903150.2903153.
URL http://doi.acm.org/10.1145/2903150.2903153

[19] D. Quinlan, C. Liao, The ROSE source-to-source compiler infrastructure, in: Cetus users and compiler infrastructure workshop, in conjunction with PACT, Vol. 2011, Citeseer, 2011, p. 1.

[20] K. Li, W. Yang, K. Li, Performance analysis and optimization for spmv on gpu using probabilistic modeling, IEEE Transactions on Parallel and Distributed Systems 26 (1) (2015) 196–205.

[21] P. Guo, L. Wang, P. Chen, A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus, IEEE Transactions on Parallel and Distributed Systems 25 (5) (2014) 1112–1123.

[22] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned spmv on gpus and multicore cpus, IEEE Transactions on Computers 64 (9) (2015) 2623–2636.

[23] H. Li, K. Li, J. An, K. Li, Msgd: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus, IEEE Transactions on Parallel and Distributed Systems 29 (7) (2018) 1530–1544.

[24] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, Mason: A multiagent simulation environment, SIMULATION 81 (7) (2005) 517–527. doi:10.1177/0037549705058073.

[25] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, C. Spagnuolo, A framework for distributing agent-based simulations, in: Euro-Par Parallel Processing Workshops, HeteroPar, 2011, pp. 460–470. doi:10.1007/978-3-642-29737-3_51.

[26] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, C. Spagnuolo, Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason, SIMULATION 89 (10) (2013) 1236–1253. doi:10.1177/0037549713489594.

[27] G. Cordasco, V. Scarano, C. Spagnuolo, Distributed mason: A scalable distributed multi-agent simulation environment, Simulation Modelling Practice and Theory 89 (2018) 15–34, cited By 1. doi:10.1016/j.simpat.2018.09.002.

[28] R. Chisholm, P. Richmond, S. C. Maddock, A standardised benchmark for assessing the performance of fixed radius near neighbours, in: Euro-Par Parallel Processing Workshops, PADABS, 2016, pp. 311–321. doi:10.1007/978-3-319-58943-5_25.

[29] Circle test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/circle.

abl.
[30] C. W. Reynolds, Flocks, herds and schools: A distributed behavioral model, ACM SIGGRAPH 21 (4) (1987) 25–34.
[31] Boids test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/circle.abl.
[32] M. Gardner, Mathematical games, Scientific American 222 (6) (1970) 132–140.
[33] Game-of-life test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/game_of_life.abl.
[34] Sugarscape test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/sugarscape.abl.
[35] L. Panait, S. Luke, A pheromone-based utility model for collaborative foraging, in: Conf. on Autonomous Agents and Multiagent Systems, 2004, pp. 36–43. doi:10.1109/AAMAS.2004.25.
[36] Ants test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/ants.abl.
[37] U. Wilensky, Netlogo wolf sheep predation model, Center for Connected Learning and Computer-Based Modeling, Northwestern University () (1997) .
[38] Predator prey test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/predator_prey.abl.
[39] Keratinocyte test benchmark, https://github.com/OpenABL/OpenABL/blob/master/examples/keratinocyte.abl.
[40] G. Cordasco, B. Cosenza, R. De Chiara, U. Erra, V. Scarano, Experiences with Mesh-like computations using Prediction Binary Trees, Scalable Computing: Practice and Experience, Scientific international journal for parallel and distributed computing (SCPE) 10 (2) (2009) 173–187.
[41] S. Stepney, F. A. C. Polack, Engineering Simulations as Scientific Instruments: A Pattern Language, Springer, 2018. doi:10.1007/978-3-030-01938-9.
URL https://doi.org/10.1007/978-3-030-01938-9
[42] M. J. North, N. T. Collier, J. R. Vos, Experiences creating three implementations of the repast agent modeling toolkit, Trans. Model. Comp. Sim. 16 (1) (2006) 1–25. doi:10.1145/1122012.1122013.
[43] N. Collier, M. North, Parallel agent-based simulation with repast for high performance computing, SIMULATION 89 (10) (2013) 1215–1235. doi:10.1177/0037549712462620.
[44] G. Cordasco, R. D. Chiara, F. Raia, V. Scarano, C. Spagnuolo, L. Vicidomini, Designing computational steering facilities for distributed agent based simulations, in: SIGSIM Principles of Advanced Discrete Simulation, 2013, pp. 385–390. doi:10.1145/2486092.2486147.
[45] N. Lettieri, C. Spagnuolo, L. Vicidomini, Distributed agent-based simulation and GIS: an experiment with the dynamics of social norms, in: Euro-Par Parallel Processing Workshops, PADABS, 2015, pp. 379–391. doi:10.1007/978-3-319-27308-2_31.
[46] A. P. Oliveira, P. Richmond, Feasibility study of multi-agent simulation at the cellular level with FLAME GPU, in: FLAIRS Conf., 2016, pp. 398–403.
[47] S. Tamrakar, P. Richmond, R. M. D'Souza, PI-FLAME: A parallel immune system simulator using the FLAME graphic processing unit environment, SIMULATION 93 (1) (2017) 69–84. doi:10.1177/0037549716673724.
[48] U. Erra, B. Frola, V. Scarano, A GPU-based interactive bio-inspired visual clustering, in: Symp. on Comp. Intelligence and Data Mining, 2011, pp. 268–275. doi:10.1109/CIDM.2011.5949300.
[49] B. Cosenza, Behavioral spherical harmonics for long-range agents' interaction, in: Euro-Par Parallel Processing Workshops, PADABS, 2015, pp. 392–404. doi:10.1007/978-3-319-27308-2_32.
[50] A. Piccione, M. Principe, A. Pellegrini, F. Quaglia, An agent-based simulation api for speculative pdes runtime environments, in: Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS '19, ACM, New York, NY, USA, 2019, pp. 83–94. doi:10.1145/3316480.3322890.
URL http://doi.acm.org/10.1145/3316480.3322890
[51] C. M. Macal, M. J. North, Tutorial on agent-based modeling and simulation, in: 37th Conf. on Winter Simulation, 2005, pp. 2–15.
[52] M. Berryman, Review of software platforms for agent based models, Technical report dsto-gd-0532, Australian Government, Department of Defense (2008).
[53] A. Rousset, B. Herrmann, C. Lang, L. Philippe, A Survey on Parallel and Distributed Multi-Agent Systems, in: Euro-Par Workshops, PADABS, 2014.
[54] I. Grasso, S. Pellegrini, B. Cosenza, T. Fahringer, A uniform approach for programming distributed heterogeneous computing systems, J. Parallel Distrib. Comput. 74 (12) (2014) 3228–3239.
[55] M. Christen, O. Schenk, Y. Cui, Patus for convenient high-performance stencils: Evaluation in earthquake simulations, in: Conf. on High Performance Computing, Networking, Storage and Anal-

ysis, SC, 2012, pp. 11:1–11:10.

[56] B. Cosenza, J. J. Durillo, S. Ermon, B. H. H. Juurlink, Autotuning stencil computations with structural ordinal regression learning, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 287–296. doi:10.1109/IPDPS.2017.102.

[57] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-marl: A DSL for easy and efficient graph analysis, in: ASPLOS, 2012, pp. 349–362. doi:10.1145/2150976.2151013.

[58] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, in: ACM SIGPLAN PLDI, 2013, pp. 519–530. doi:10.1145/2491956.2462176.

[59] T. Denniston, S. Kamil, S. Amarasinghe, Distributed halide, in: ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP, 2016, pp. 5:1–5:12. doi:10.1145/2851141.2851157.

[60] M. Kong, L.-N. Pouchet, P. Sadayappan, V. Sarkar, Pipes: A language and compiler for task-based programming on distributed-memory clusters, in: Conf. for High Performance Computing, Networking, Storage and Analysis, SC, 2016, pp. 39:1–39:12.

[61] V. Janjic, C. Brown, K. Mackenzie, K. Hammond, M. Danelutto, M. Aldinucci, J. D. García, RPL: A domain-specific language for designing and implementing parallel C++ applications, in: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016, 2016, pp. 288–295. doi:10.1109/PDP.2016.122.

[62] S. Rajbhandari, J. Kim, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, R. J. Harrison, P. Sadayappan, A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment, in: Conf. on High Performance Computing, Networking, Storage and Analysis, SC, 2016, pp. 40:1–40:12.

[63] V. Vasista, K. Narasimhan, S. Bhat, U. Bondhugula, Optimizing geometric multigrid method computation using a dsl approach, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, 2017, pp. 15:1–15:13. doi:10.1145/3126908.3126968.

[64] S. Karol, T. Nett, J. Castrillon, I. F. Sbalzarini, A domain-specific language and editor for parallel particle methods, ACM Trans. Math. Softw. 44 (3) (2018) 34:1–34:32. doi:10.1145/3175659.
URL http://doi.acm.org/10.1145/3175659

[65] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a domain specific language for building portable mesh-based PDE solvers, in: Conf. on High Perfor. Computing Networking, Storage and Analysis, 2011, pp. 9:1–9:12. doi:10.1145/2063384.2063396.