Multi-Stage Vertex-Centric Programming for Agent-Based Simulations

Zilu Tian
zilu.tian@epfl.ch
EPFL
Lausanne, Switzerland

Abstract

In vertex-centric programming, users express a graph algorithm as a vertex program and specify the iterative behavior of a vertex in a `compute` function, which is executed by all vertices in a graph in parallel, synchronously in a sequence of supersteps. While this programming model is straightforward for simple algorithms where vertices behave the same in each superstep, for complex vertex programs where vertices have different behavior across supersteps, a vertex needs to frequently dispatch on the value of supersteps in `compute`, which suffers from unnecessary interpretation overhead and complicates the control flow.

We address this using meta-programming: instead of branching on the value of a superstep, users separate instructions that should be executed in different supersteps via a staging-time `wait()` instruction. When a superstep starts, computations in a vertex program resume from the last execution point, and continue executing until the next `wait()`. We implement this in the programming model of an agent-based simulation framework CloudCity and show that avoiding the interpretation overhead caused by dispatching on the value of a superstep can improve the performance by up to 25% and lead to more robust performance.

CCS Concepts: **• Software and its engineering → Domain specific languages**; **• Computing methodologies → Simulation languages**; **Parallel algorithms**.

Keywords: agent-based simulations, vertex-centric programming, distributed systems, staging, metaprogramming

ACM Reference Format:
Zilu Tian. 2023. Multi-Stage Vertex-Centric Programming for Agent-Based Simulations. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3624007.3624057

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10…$15.00
https://doi.org/10.1145/3624007.3624057

1 Introduction

The vertex-centric programming paradigm is first proposed and popularized by Google's Pregel [21] for the efficient execution of graph algorithms over large, distributed graphs. The core of its computational model is based on the bulk-synchronous parallel (BSP) processing model [37], which is an abstract machine model consisting of parallel processors that communicate by sending messages. Computations proceed synchronously in a sequence of supersteps,[1] separated by global synchronization. Messages between parallel processors are sent at the end of a superstep and are available for processing at the beginning of a superstep. BSP achieves high scalability in distributed systems by performing computations in parallel. Additionally, the BSP model ensures that a parallel program is free of deadlocks and data races, due to its synchronous, shared-nothing architecture, hence making parallel programming easy.

In vertex-centric programming, users specify an input graph and express a graph algorithm as a vertex program [21]. The vertex program contains a vertex's state variables and a `compute` function that describes the behavior of a vertex, such as processing messages, performing local computations, and sending messages to other vertices. Computations proceed in a sequence of supersteps, in each of which vertices execute `compute` in parallel for exactly once.

To clarify, let s_0 be the value of the initial state of an arbitrary vertex in a graph and s_k be the value of this vertex's state after executing `compute` in the k-th superstep of a graph algorithm, then we can describe the change of this vertex's state using the following state transition, where each arrow is labeled with the transition name, that is, executing `compute`:

$$s_0 \xrightarrow{\texttt{compute}} s_1 \xrightarrow{\texttt{compute}} s_2 \xrightarrow{\texttt{compute}} s_3 \ldots.$$

For a complex vertex program that executes different instructions across supersteps, interpreting `compute` is inefficient. We demonstrate it with the following example written in Scala.[2] Let A, B, and C represent three blocks of instructions. In every superstep, a vertex needs to check whether

[1] In the BSP model, a step refers to a primitive operation like reading or writing the processor's local memory, similar to the PRAM model. The term *superstep* emphasizes that a processor can perform arbitrarily many steps before synchronizing with other processors.

[2] We will use Scala syntax throughout this paper.

```
1  var color: Int = 0
2  def compute(): Unit = {
3    if (superstep == 0) {
4      color = 1
5      // syntactically correct
6      // but semantically wrong
7      if (superstep == 1)
8        color = 2
9      if (superstep == 2)
10       color = 3
11   }
12 }
```

(a) Dispatching on the value of a superstep is error-prone.

```
1  var color: Int = 0
2  @lift
3  def compute(): Unit ={
4    color = 1
5    wait()
6    color = 2
7    wait()
8    color = 3
9    wait()
10 }
```

(b) A staged vertex program expressed with `wait()`.

```
1  var color: Int = 0
2  var idx: Int = 0
3  var yield: Boolean = false
4  val sc = Array(
5   ()=>{color=1; yield=true; idx=1},
6   ()=>{color=2; yield=true; idx=2},
7   ()=>{color=3; yield=true; idx=0})
8  def compute(): Unit = {
9    yield = false
10   while (!yield) {
11     sc(idx)()
12   }
13 }
```

(c) Vertex program generated from the staged vertex program in (b).

Figure 1. In a vertex program, describing different vertex behavior across supersteps is commonly achieved by branching on a superstep value. However, this suffers from interpretation overhead and permits erroneous programs like (a), where instructions that ought to be executed in superstep 1 (line 8) can still be expressed as instructions for superstep 0 (lines 4–10) instead. We eliminate the need to dispatch on the superstep value by introducing a staging-time coroutine-like instruction `wait()` in (b), which transforms vertex behavior into an array of closures in the generated vertex program in (c).

the current superstep value is 0 (line 2) or 1 (line 4), which is unnecessary and inefficient for a long-running program.

```
1 def compute(): Unit = {
2   if (superstep == 0) {
3     A  // arbitrary computation, same for B, C
4   } else if (superstep == 1) {
5     B
6   } else {
7     C
8   }
9 }
```

The need for complex vertex programs where vertices have different behavior across supersteps arises for agent-based simulations, where each vertex is regarded as an agent. Per superstep, an agent performs arbitrary local computations, sends messages, and processes received messages.[3] In such applications, agents often engage in diverse operations across supersteps, and multiple types of concurrent agents exhibit distinct behaviors within the same superstep. For example, an epidemics simulation can model both hospitals and people as agents, which have different behaviors. People agents also behave differently across supersteps as the disease progresses.

We would like a vertex to only execute instructions that are *useful*. Intuitively, in superstep 1, a vertex only needs to execute line 5, without having to evaluate the conditions on lines 2 and 4 first. We capture the intuition of useful instructions for a given superstep in `compute` using notations from partial evaluation [14]. Let α be a meta-algorithm that partially evaluates `compute` when the variable `superstep` has value t. We use $\alpha(\texttt{compute}, t)$ to denote the residual program generated after the partial evaluation of `compute` with respect to the variable `superstep` at value t, which is also referred to as *useful* instructions for superstep t in `compute`. The value of a vertex's state when only executing useful instructions in each superstep changes as below:

$$s_0 \xrightarrow{\alpha(\texttt{compute},0)} s_1 \xrightarrow{\alpha(\texttt{compute},1)} s_2 \xrightarrow{\alpha(\texttt{compute},2)} s_3 \ldots.$$

In our example, $\alpha(\texttt{compute}, 0) = \texttt{A}$, $\alpha(\texttt{compute}, 1) = \texttt{B}$.

Additionally, eliminating the need to branch on the value of supersteps has the benefit of preventing erroneous programs like Figure 1a, where instructions that describe behavior in superstep 1 (line 8) are specified in the scope of instructions to be executed in superstep 0 (lines 3–11).

In this work, we describe how we ensure that vertices only execute useful instructions by making the superstep-dependent structure in a vertex program explicit through a coroutine-like [7] staging-time instruction `wait()`.[4] Just like coroutines, computations of a vertex function yield when executing `wait()` and later resume from the last saved execution point when the next superstep begins, avoiding the interpretation overhead caused by dispatching on the value of a superstep. The previous pseudocode can be expressed as follows using `wait()`:

[3]For simplicity, we assume that the semantics of an agent-based simulation is precisely the BSP model, where computations of parallel agents proceed iteratively in a sequence of supersteps. An agent-based simulation can be implemented as a vertex program.

[4]As pointed out by a reviewer, the term `wait` may cause confusions with similarly named primitives in *asynchronous* programming. Although a vertex program is single-threaded and sequential, `wait()` is a blocking instruction that delays the execution of later instructions until the next superstep starts, not unlike asynchronous programming.

```
1  def compute(): Unit = {
2    A  // arbitrary computation, same for B, C
3    wait()
4    B
5    wait()
6    while(true){
7      C
8      wait()
9    }
10 }
```

We implement wait() in the programming model of a distributed agent-based simulation system CloudCity [35] and evaluate the performance benefit of staging compared with unstaged vertex programs. Our results show that for an agent with 5 to 20 branches, staging reduces the total execution time by 10% – 25%. As we increase the number of agents, the overall hardware utilization efficiency decreases as resource contention worsens. For 1000 agents, the speedup of staging is around 10%. Though the speedup is modest in both cases, staging greatly improves the robustness of the performance. We repeat each experiment three times: the standard deviation is 10× lower for staged programs than unstaged ones, for both one and 1000 agents.

The rest of the paper is structured as follows. We start by explaining in more detail what we mean by vertex-centric programming and agent-based simulations in Section 2. Afterward, we describe the limitations of the current approach in Section 3 and possible solutions. In Section 4, we examine the staging-time wait() instruction closely. We then discuss how we implement wait() in CloudCity in Section 5 and evaluate the performance impact in Section 6. Finally, we discuss related work in Section 7 and end this paper with conclusions and future works in Section 8.

2 Background

Vertex-centric programming was first proposed and popularized by Pregel [21]. Later other frameworks like GraphX [39] and Flink [9] have also adopted this paradigm for parallel graph processing, implementing the vertex-centric paradigm via a Pregel-like operator. Though there are differences among such frameworks concerning details such as whether the graph topology is mutable, all these frameworks share the same assumption as Pregel, where vertices interpret the same user-defined function iteratively in every superstep.

In this section, we use Pregel as an example and explain its syntax and semantics to familiarize users with vertex-centric programming. More concretely, we show how to implement a classic graph algorithm, PageRank [3], which represents the target applications that Pregel is designed for. We also explain what agent-based simulations are and how they can be expressed as complex vertex programs.

```
1  abstract class Vertex[VertexValue, EdgeValue,
     MessageValue] {
2    var value: VertexValue
3    val vertexId: String
4    val numVertices: Int
5
6    def superstep: Int = 0
7
8    def compute(msgs: Iterator[MessageValue]):
       Unit
9    // type Edge is built-in
10   def getOutEdgeIterator(): Iterator[Edge]
11   def sendMessageTo(dest: String, message:
       MessageValue): Unit
12   def voteToHalt(): Unit
13 }
```

Figure 2. Core Pregel DSL.

2.1 Pregel syntax

Pregel is a domain-specific language (DSL) for vertex-centric computing embedded purely in the host language C++ [21]. For demonstration, we present Pregel using Scala pseudocode, summarized in Figure 2. Each vertex has a unique id (line 3) and can obtain the current superstep (line 6). Users need to define compute (line 8), which specifies the behavior of a vertex that is executed in each superstep, such as updating local states and sending messages to neighbors (lines 2, 10–11). If there is no further work until new messages arrive, a vertex votes to halt (line 12).

Users define a vertex program by creating a subclass that extends the Vertex class. In particular, users provide types for VertexValue, EdgeValue, and MessageValue (line 1) and override the compute method (line 8) with vertex behavior. A graph algorithm can only define one vertex program.

2.2 Pregel semantics

The semantics of Pregel closely follows the BSP processing model [21], which is an abstract parallel machine that is commonly used by distributed frameworks for parallel computing. This abstract machine contains:

- a set of processors. Each processor can be viewed as a core-memory pair, where the memory is private to the core. A core can perform arbitrary computations over values stored in its memory. Processors communicate by sending messages, which arrive at the beginning of a superstep;
- a synchronization facility to synchronize all processors periodically.

A Pregel program proceeds in a sequence of supersteps separated by global synchronization. Initially, all vertices are *active*. Per superstep, every active vertex executes its compute method in parallel. A superstep ends when all active

vertices have completed executing compute. An active vertex becomes *inactive* by voting to halt (through the Pregel instruction voteToHalt), suggesting that it has no more work to do until new messages arrive. An inactive vertex becomes active if it has received at least one message at the beginning of a superstep, and remains inactive otherwise. Messages are sent at the end of a superstep and are available for processing at the beginning of the next superstep. In Pregel, a program terminates when all vertices have become *inactive* and there is no more message in the system.

Example 2.1 (PageRank)**.** The PageRank algorithm computes the importance of web pages based on the assumption that a page is more important if it receives links from other important pages [3]. Each page has a PageRank value, where a higher value suggests that a page is more important. This algorithm proceeds in the following two phases:

- Initialization: Assign an initial PageRank value to each page in the system. Initially, all pages have equal PageRank values;
- Iterative Update:
  - For each page, calculate its new PageRank value based on the incoming links from other pages.
  - Distribute a fraction of the current page's PageRank to the pages it links to. The amount distributed is proportional to the current page's PageRank and inversely proportional to the number of its outgoing links.
  - Update the PageRank values for all pages.

  Repeat the above steps until the PageRank values converge (no significant changes).

To account for user behavior, PageRank introduces a damping factor that models the probability of a user continuing to click on links instead of jumping to a random page, typically set to 0.85. The damping factor adjusts the distribution of a page's PageRank during the iterative update.

Implementing PageRank in Pregel is straightforward, shown in Figure 3 (adapted from [21]). Each vertex has a local state that denotes its current PageRank value (type double). In superstep 0, the PageRank value of each vertex is the same, given in a separate input graph file (not shown here). Per superstep, vertices send messages to their neighbors (lines 14–16), where each message contains its current PageRank value divided by the number of outgoing edges. Since messages take one superstep to arrive, starting from superstep 1, each vertex also processes all received messages and updates its value of PageRank before sending messages to neighbors (lines 7–11). In this example, the PageRank algorithm runs only for 30 supersteps (line 13); vertices vote to halt afterward (line 18).

```
1  class PageRankVertex extends Vertex[Double,
     Unit, Double] {
2    // Attributes value, vertexId, numVertices,
3    // getOutEdgeIterator are initialized,
     omitted
4
5    override def compute(msgs: Iterator[Double])
     : Unit = {
6      if (superstep >= 1){
7        var sum: Double = 0
8        while (msgs.hasNext) {
9          sum += msgs.next()
10       }
11       value=0.15/numVertices+0.85*sum
12     }
13     if (superstep < 30) {
14       val n: Int = getOutEdgeIterator().
     toIterable.size
15       // A syntactic sugar for sending
     messages to neighbors
16       sendMessageToAllNeighbors(value / n)
17     } else {
18       voteToHalt()
19     }
20   }
21 }
```

Figure 3. PageRank implementation.

2.3 Agent-based simulations

Agent-based simulations are simulations in which a number of agents, each with their own thread, code, and state, interact in a virtual environment. These simulations enable users to make changes to the micro behavior of agents and observe the collective impact of such changes at the macro scale. An agent may execute arbitrary code, affecting its own state as well as the state of the virtual world and any other agents living within.

There are two natural flavors of agent-based simulations, synchronous and asynchronous. Here we only consider synchronous agent-based simulations, where computations proceed in a sequence of locksteps, just like supersteps in the BSP model. An agent can be viewed as a vertex, computing locally and communicating with other agents synchronously; an agent-based simulation can be expressed as a graph algorithm using a vertex program.

Example 2.2 (Traffic light simulation)**.** We consider a minimal traffic simulation that models a pedestrian and a traffic light. The traffic light repeatedly iterates over three colors: green, yellow, and red. By default, the traffic light waits for three supersteps for each color. Additionally, the traffic light has a pedestrian crossing button, which causes the current signal to change to green when activated. Initially, the traffic light is in a random color and notifies the pedestrian of the signal. Whenever the traffic signal changes, the traffic light notifies the pedestrian of the new color. The pedestrian wants to cross and waits patiently for the green light. If the light is yellow, the pedestrian presses the crossing button. If red, the

pedestrian waits for up to two supersteps before pressing the crossing button. The traffic light turns green when receiving an activation signal from the pedestrian button.

This example highlights the need to define different vertex behavior within the same superstep in agent-based simulations, which is achieved by dispatching on the vertex id in the current vertex-centric programming, incurring unnecessary interpretation overhead.

3 Limitations of Vertex-Centric Programming

Vertex-centric programming has gained wide popularity due to its simple yet flexible programming model. Still, there are several limitations to this approach, which we describe in this section.

3.1 Interpretation overhead caused by dispatching on the value of a superstep

In Pregel, users dispatch on the value of supersteps to express different vertex behavior across supersteps. Our example in Section 2 also illustrates this problem. The code snippet in Figure 3 (lines 6, 13) clearly demonstrates the overhead of dispatching on the value of supersteps.

One standard approach to eliminate such interpretation overhead is to generate a specialized program that is more efficient to execute by partially evaluating `compute` with respect to a superstep value t.

Since the value of the current superstep t is only changed by the Pregel runtime system and cannot be reassigned by a vertex program, we can lift `compute` to pass the superstep value t as an additional input argument:

$$\overline{\texttt{Compute}}^{\texttt{t}}(t : \texttt{Int},\ msgs : \texttt{Iterator[Message]}),$$

and replace function applications of `superstep` in `compute` in a vertex program with t via β-reduction.

Generating a specialized program with respect to t by partially evaluating $\overline{\texttt{Compute}}^{\texttt{t}}$ with respect to t can be described using the following transformation [14]:

$$\overline{\texttt{Compute}}^{\texttt{t}}(t, msgs) = \alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t)(msgs),$$

where α is a partial evaluation algorithm that takes the program $\overline{\texttt{Compute}}^{\texttt{t}}$ and a superstep value t as partial evaluation variables and produces a residual program that is specialized for the superstep t. For our analysis, the details of α are not important. We assume that α executes $\overline{\texttt{Compute}}^{\texttt{t}}$ symbolically on a given value of t.

Assuming that the test expression of a control flow that contains t can be evaluated at specialization time [17], the partial evaluation approach requires precomputing a specialized program $\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t)$ for each possible value of t from 0 to K for a Pregel program that runs for K supersteps, as shown in Figure 4 for a vertex program that executes for

```
1  var idx: Int = 0
2  val instructions = Array(
3    () => {α(Compute^t,0); idx = 1},
4    () => {α(Compute^t,1); idx = 2},
5    () => {α(Compute^t,2); idx = 3},
6    ...,
7    () => {α(Compute^t,29); idx = 30},
8    () => {idx = 30}
9  )
10 def compute(msgs: Iterator[Message]): Unit = {
11   instructions(idx)()
12 }
```

Figure 4. A Pregel program with partial evaluation.

30 supersteps. The attribute `instructions` is an array of generated programs $\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t)$ indexed by t. At each superstep, only useful instructions in the generated program $\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t)$ are interpreted, hence more efficient than executing `compute`.

But precomputing residual programs for t may not always be feasible, since the test expression in a control flow that contains t can be dynamic, where t is compared to a dynamic variable whose value is known only at runtime. In addition, this approach makes it difficult to exploit *locality* across supersteps, where a vertex program repeatedly executes the same instructions in different supersteps. For instance, in the PageRank example,

$$\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, 1) = \alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t),\ \text{for } 1 < t < 30.$$

Instead of repeatedly computing $\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, t)$ for $1 \leq t < 30$, we would like to only compute $\alpha(\overline{\texttt{Compute}}^{\texttt{t}}, 1)$.

3.2 Lack of binding constructs for instructions executed in different supersteps

In vertex-centric programming, there is no binding construct for instructions executed in different supersteps. In particular, this permits erroneous programs like in Figure 1a, where it is syntactically correct to express instructions that should be executed in different supersteps within the same superstep, which are semantically incorrect.

More concretely, we summarize a simplified grammar for vertex-centric programs in Figure 5, which contains two data types, Booleans (line 1) and Integers (line 2), and selected operations (line 5) over these data types. The `superstep` (line 3) is a constant expression whose value is that of the current superstep. A program defined in `compute` is an expression in *Exp* (line 6). This allows for incorrect programs like Figure 1a, where instructions that should be executed in superstep 1 are captured in the scope of instructions that should be executed in superstep 0 instead.

We observe that this problem can be addressed by restricting the usage of `superstep` in *Exp* (line 6). We introduce an

```
1  Booleans b ∈ B = {true, false}
2  Integers n ∈ N = {..., -1, 0, 1, ...}
3  ConstSymbol s = {superstep}
4  VariableSymbols v ∈ Sym
5  Operations op ∈ {+, ≥, ==}
6  Expr e ∈ Exp ::= n | b | s | v := e | e op e |
        if e then e else e | e; e | while e do e
```

Figure 5. A simplified grammar for vertex-centric programs.

explicit binding construct:

$$\texttt{sc}(step : \texttt{Int},\ inst : () \texttt{=>}\ \texttt{Unit}) : \texttt{Unit},$$

which allows programmers to express instructions that should be executed in different supersteps. The *step* is an integer that denotes the value of a superstep, and *inst* denotes a block of instructions that should be executed in the given superstep.

The `compute` method contains a sequence of `sc` instructions. In each superstep, `compute` evaluates the closure specified in the `sc` with the current superstep value, defaulting to no action if no such bindings are defined.

The example in Figure 1a can be expressed as below.

```
1  val sc = Mutable.Map[Int, () => Unit]()
2  var color: Int = 1
3  def compute(): Unit = {
4    sc(0, () => {color = 1})
5    sc(1, () => {color = 2})
6    sc(2, () => {color = 3})
7
8    // a syntactic sugar for
9    // sc.getOrElseDefault(superstep,()=>Unit)()
10   sc(superstep)()
11 }
```

In this example, lines 4–6 specify a sequence of closures in the `compute` method. Per superstep, the closure defined for the current superstep is evaluated (line 10).

3.3 Code duplication

While introducing a binding construct `sc` can eliminate incorrect programs like Figure 1a, this can lead to undesirable code duplication when a vertex has shared instructions across supersteps, such as in intricate branching patterns. To demonstrate, we show how the vertex program in Figure 3 can be expressed using the new binding construct below.

```
1  val sc = Mutable.Map[Int, () => Unit]()
2  def compute(msgs: Iterator[Double]): Unit = {
3    sc(0, () => {
4      val n: Int = getOutEdgeIterator().
       toIterable.size
5      sendMessageToAllNeighbors(value / n)
6    })
7    Range(1, 30).foreach(idx => {
8      sc(idx, () => {
9        var sum: Double = 0
10       while (msgs.hasNext) {
11         sum += msgs.next()
12       }
13       value=0.15/numVertices+0.85*sum
14       val n: Int = getOutEdgeIterator().
     toIterable.size
15       sendMessageToAllNeighbors(value / n)
16     })
17   }
18   sc(superstep)()
19 }
```

Compared with Figure 3, readers may notice that there is no longer a `voteToHalt` instruction in the body of `compute`. We point out that this is because we have assumed a different termination condition than that of naive Pregel, to make it more suitable for agent-based simulations: a user specifies a fixed number of supersteps `total` that a vertex program should run. This is desirable for agent-based simulations, where users may want to express that an agent waits for some supersteps before performing another action. Without such changes, vertices that are waiting to perform the next action will be considered *inactive* and Pregel simply terminates when all vertices are waiting.[5]

In the code above, we see that lines 4–5 and lines 14–15 are duplicated, which is not the case for the Pregel implementation in Figure 3. To address this, we want to adjust the granularity of the closures and let each superstep execute multiple closures instead of one. In particular, we bind shared instructions into closures and reference them using the De Bruijn index [6], specifying the index of a continuation at the end of each closure. To illustrate, the previous code snippet can be refactored as below, where `idx` (line 1) denotes the De Bruijn index of the continuation and `sc` is an array of closures (lines 2–15). Now there is no more duplicated code in the closures on lines 4–6 and lines 9–14.

```
1  var idx: Int = 0  // De Bruijn index
2  val sc: Array[() => Unit](
3  () => {
4    val n: Int = getOutEdgeIterator().toIterable
       .size
5    sendMessageToAllNeighbors(value / n)
6    idx = 1
7  },
8  () => {
9    var sum: Double = 0
10   while (msgs.hasNext) {
11     sum += msgs.next()
12   }
13   value=0.15/numVertices+0.85*sum
14   idx = 0
15 })
```

[5]Assume that there are no messages in the system.

What is missing from the code above is how to determine which closures to execute in a given superstep. In this example, superstep 0 should execute closure 0, and each of the later supersteps should execute closures 1 and 0. We observe that the computation of a new superstep resumes precisely from the state where computations have yielded in the last superstep, similar to coroutines [7]. Hence we introduce a `yield` flag that is similar to the `yield` coroutine instruction. In any superstep, a vertex program continues evaluating the indexed closures until `yield` becomes `true`. The full vertex program for the PageRank example can be described below.

```
1  var idx: Int = 0  // De Bruijn index
2  var yield: Boolean = false
3  val sc: Array[() => Unit](
4  () => {
5    val n: Int = getOutEdgeIterator().toIterable
       .size
6    sendMessageToAllNeighbors(value / n)
7    yield = true
8    idx = 1
9  },
10 () => {
11   var sum: Double = 0
12   while (msgs.hasNext) {
13     sum += msgs.next()
14   }
15   value=0.15/numVertices+0.85*sum
16   idx = 0
17 })
18
19 def compute(): Unit = {
20   yield = false
21   while (!yield) {
22     sc(idx)()
23   }
24 }
```

It is easy to verify that in superstep 0, only closure 0 (lines 6–9) is executed; in later supersteps, both closures 1 (lines 12–17) and 0 are executed, just as expected.

3.4 Explicit dependency on the value of vertex ids

In Pregel, users define only one vertex program for a graph algorithm. To express different behavior in the same superstep, as illustrated in the traffic light simulation example, users need to branch on the value of vertex ids. Similar to the problem of branching on the value of supersteps as we have just discussed, conditioning on the value of vertex ids is error-prone and inefficient. This can be addressed by allowing polymorphic vertices, where users define multiple vertex classes that correspond to different types of vertices, which is straightforward.

4 Staging-Time `wait()` Instruction

We have discussed various limitations of the existing vertex-centric programming and how such limitations can be addressed. In particular, we have shown how users can specify the behavior of a vertex as a sequence of closures using the binding construct `sc` to avoid:

- interpretation overhead caused by dispatching on the value of a superstep;
- incorrect programs where instructions that should be executed in superstep t are specified in the body of the branch for superstep k instead ($t \neq k$); and
- duplicated code that arises from executing a single closure in a superstep.

Still, defining such closures and their continuation directly by users is low-level and error-prone. We want to provide a high-level user-friendly interface and let the system automatically generate such closures.

To this end, we introduce a staging-time instruction `wait()` that separates instructions in a vertex program into different supersteps, making it easy to automatically generate a sequence of closures based on static analysis at staging time.[6]

The PageRank example can be expressed below using `wait()`. Users no longer need to create different closures or specify the index of the continuation that points to the closure that should be executed next.

```
1  @lift
2  def compute(msgs: Iterator[Double]): Unit = {
3    while (true) {
4      val n: Int = getOutEdgeIterator().
       toIterable.size
5      sendMessageToAllNeighbors(value / n)
6      wait()
7      var sum: Double = 0
8      while (msgs.hasNext) {
9        sum += msgs.next()
10     }
11     value=0.15/numVertices+0.85*sum
12   }
13 }
```

This program is a staged program – as suggested by the `@lift` annotation on line 1 – that is lifted into some intermediate representation for preprocessing before generating instructions. In the generated code, an array of closures that specifies the behavior of a vertex is created outside the scope of `compute`, which is executed only once during initialization, like other state variables of the vertex.

[6]For the ease of presentation, we only discuss `wait()` with arity 0. In practice, we implement a 1-arity `wait(`n : `Int)`, which blocks until n supersteps have passed. This makes it straightforward to specify a vertex with idle behavior across supersteps. For example, a vertex can perform computations only in supersteps 0 and 100. In this case, users can use `wait(99)` to clarify that a vertex is idle and has no behavior for 99 supersteps. Here we focus on explaining how `wait()` separates instructions into different supersteps.

5 CloudCity

We implement the staging-time `wait()` instruction in CloudCity [35], a distributed agent-based simulation system. An *agent* is single-threaded and communicates by sending messages. For this purpose, each agent has a mailbox and is uniquely addressable. Incoming messages are buffered in the mailbox, waiting passively to be processed by the agent. An agent-based simulation consists only of such agents.

The software stack of CloudCity is shown in Figure 6. There are four layers in total: users interact with the core indirectly via the programming model in the *frontend*; *core* refers to the distributed engine for agent-based simulations; *optimizations* contain system optimizations, both at compile-time and runtime; *backend* refers to the low-level distributed library used in the implementation.

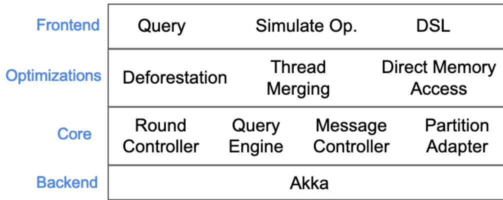

Figure 6. Software stack of CloudCity. The *core* refers to the distributed agent-based simulation engine. Users interact with the core indirectly via the programming model in the *frontend*. There are different runtime and compile-time system optimizations, summarized in the layer *optimizations*. The *backend* refers to the low-level distributed library used in the implementation.

Our discussion here concerns only the DSL in the frontend. For simplicity, we assume that computations proceed in a sequence of supersteps, exactly as described in BSP, and all messages take one superstep to arrive.[7]

The DSL is embedded in the host language Scala, shown in Figure 7: `wait()` is a synchronization instruction since it plays the role of a barrier instruction that synchronizes all agents at the end of a superstep.[8]

The DSL also contains communication instructions that let agents communicate in a distributed environment. To send a message, an agent calls

$$\texttt{send}(rid : \texttt{Long}, m : \texttt{Message}) : \texttt{Unit},$$

where *rid* is the id of the receiver agent and *m* is a message object. `Message` class is defined in our library and can be extended. Retrieving a message is done via

$$\texttt{receive}() : \texttt{Option[Message]},$$

which returns `None` if the mailbox is empty.

Together, `send` and `receive` implement the basic message-passing protocol, but this protocol places no restriction on what messages contain and how messages are processed. Specifically, a message can be ill-formed and does not contain all arguments that are required for processing. In practice, it is desirable to ensure that messages are well-formed. Hence, CloudCity also supports remote procedure calls (RPCs), a special type of message-passing protocol that limits senders to only send valid messages that can be processed by receivers. RPCs have two types of messages, requests and replies.

We view public methods in an agent program as *RPC methods*. An RPC request message contains the identifier of an RPC method defined in the receiver. When processing an RPC request, the receiver looks up the corresponding RPC method and invokes it locally. For performance reasons, we differentiate two types of RPC requests in our DSL, depending on whether a receiver sends an RPC reply message: a *two-sided* RPC request requires the receiver to send an RPC reply message back to the sender, which contains the return value of the local call; a *one-sided* RPC request does not have an associated reply.

Agents send a two-sided RPC request message with

$$\texttt{asyncCall}(() \Rightarrow receiver.API(args^*) : \texttt{T}) : \texttt{Future[T]},$$

which returns a future object used by the sender to check whether the RPC reply has arrived and to retrieve the return value in the RPC reply. A future object has type `Future[T]`, which is defined in the CloudCity library, but with a similar usage as that in the standard Scala library. `T` is a type variable that denotes the type of return value in an RPC reply. Sending a one-sided RPC request can be achieved using

$$\texttt{callAndForget}(() \Rightarrow receiver.API(args^*) : \texttt{T}) : \texttt{Unit},$$

which does not return a future object.

To process RPC request messages, an agent can repeatedly call `receive`, parse the message, and invoke the corresponding RPC methods. Alternatively, DSL allows a receiver to retrieve and process RPC requests in batch via

$$\texttt{handleRPC}() : \texttt{Unit},$$

which traverses received messages in an arbitrary order. Per RPC request, the receiver calls the corresponding method and sends a reply when applicable. An RPC reply has the same `delay` and `sessionId` as the corresponding request.

RPC instructions like `asyncCall` and `callAndForget` are convenient for users by isolating them from low-level implementation details such as how to package the corresponding RPC message. CloudCity automatically derives available RPC methods from agent class definitions and assigns these methods a unique method identifier at specialization time before generating the corresponding message. In other words, such instructions are *code generators* that emit different instructions during staging time. Similarly, `handleRPC` is also a

[7] Compare with [35], we eliminate *delay* from the signature of RPC instructions `callAndForget` and `asyncCall` for simplicity.

[8] For simplicity, here we consider `wait()`, as we have discussed in the previous section. This is different from `wait(n) : Unit` in [35], which takes an integer parameter. The semantics of `wait()` is the same as `wait(1)`.

code generator that can produce different instructions for different agent class definitions.

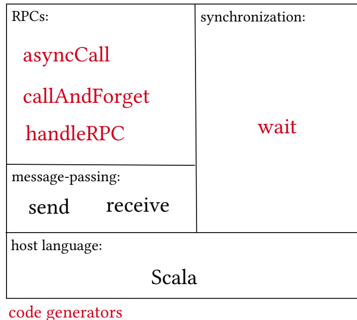

Figure 7. DSL stack. CloudCity DSL is embedded in the host language Scala and contains three components: message-passing, RPCs, and synchronization. While message-passing instructions can be implemented as Scala functions directly, instructions in RPCs and synchronization (shown in red) are code generators that require binding-time analysis to dynamically generate corresponding Scala instructions.

An agent program defined using DSL is a metaprogram since it contains code generators. The metaprogram is lifted using Squid library [25] into A-normal form (ANF) [13] as an intermediate representation (IR), before generating the corresponding instructions and being transformed into a Scala program. The ANF names intermediate results to avoid code duplication, generating one or more closures for each user instruction in the vertex program, which are later combined to reduce the number of closures. The T-diagram of CloudCity is shown in Figure 8.

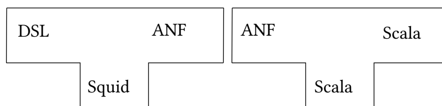

Figure 8. An agent program in CloudCity written in DSL (left) which contains `wait()` is lifted using Squid into ANF before being transformed into a Scala program (right).

More concretely, we show how to implement PageRank in CloudCity below. Line 1 is the annotation for using the class-lifting macro in Squid. The `neighborRank` (lines 4–6) is an RPC method that can be specified in RPC requests (line 12). The behavior of an agent is defined in the `main` method (lines 7–15). The `outNeighbors` is an attribute defined in the `Agent` class in CloudCity, which contains a collection of neighboring agents that an agent can send messages to.

```
1  @lift
2  class Vertex(var value: Double) extends Agent{
3    var sum: Double = 0
4    def neighborRank(s: Double): Unit = {
5      sum += s
6    }
7    def main(): Unit = {
8      while (true) {
9        sum = 0
10       handleRPC()
11       value = 0.15/numVertices+0.85*sum /
       outNeighbors.size
12       outNeighbors.foreach(i => callAndForget
       (()=>i.neighborRank(value)))
13       wait()
14     }
15   }
16 }
```

After defining a vertex program, users lift it using Squid into an ANF IR and compile the IR into a Scala program:

```
1  // Squid representation of the lifted class
2  val cls = Vertex.reflect(IR)
3  // CloudCity compiler that translates Squid IR
4  // to generate (write) a target Scala program
5  compile(cls)
```

CloudCity compiler rewrites the ANF IR using Squid. For example, the rewrite rule for `callAndForget` using code pattern matching in Squid is:

```
1  cde match {
2    case code"callAndForget[$mt](()=>${
         m@MethodApplication(msg)}:mt, $t:Int)"
         =>
3      val receiverActorVar = msg.args.head.head
4      val argss = msg.args.tail.map(a => a.map(
         arg => code"$arg"))
5      val methodId = methodIdMap(msg.symbol.
         asTerm.name)
6      Send[T](receiverActorVar, methodId, t,
         argss)
7  }
```

On line 1, `cde` is the lifted agent definition in IR. `Send` (line 6) is a CloudCity compiler IR that is later translated to Scala. Similarly for other code generator instructions.

CloudCity compiler generates an array of closures that describe the agent behavior, as we have discussed in Section 4. This is achieved by walking the abstract syntax tree of the agent behavior defined in the `main` method, where each node of the tree is transformed into a closure. Closures that correspond to a sequence of straight-line instructions that do not contain branches or `wait()` are later merged into one closure to reduce the number of closures.

6 Evaluation

In this section, we quantify the performance benefit of eliminating the need to dispatch on the value of supersteps empirically. For hardware, we use a server that has 24 cores (two Intel Xeon E5-2680 v2, 48 hardware threads), 128GB of RAM, and 200GB of SSD.

We evaluate both approaches – with and without eliminating dispatching overhead – using a microbenchmark with nested branching, where if-statements are nested within other if-statements, for up to two levels. In the innermost body of a condition, we model simple computations where an agent performs some calculations and assigns the result. For instance, the pseudocode for a vertex program with three branches is shown below, for one and two levels respectively.

```
1  // one level
2  def compute(): Unit = {
3    if (time % period == 0)
4        color += Random.nextInt() + 1
5    else if (time % period == 1)
6        color += Random.nextInt() + 2
7    else if (time % period == 2)
8        color += Random.nextInt() + 3
9    ...
10 }
11
12 // two levels
13 def compute(): Unit = {
14   if (Random.nextBoolean()) {
15     if (time % period == 0)
16         color += Random.nextInt() + 1
17     else if (time % period == 1)
18         color += Random.nextInt() + 2
19     else if (time % period == 2)
20         color += Random.nextInt() + 3
21     ...
22   }
23   ...
24 }
```

Per experiment, we measure the total execution time when running 1000 supersteps, both with and without staging, and repeat three times after warming up the cache. We first consider a single agent and increase the number of the innermost branches from 5 to 20, for both one and two levels, shown in Figure 9. The x-axis denotes the number of branches and the y-axis denotes the total execution time of each experiment, averaged over three runs. The standard deviation is shown using error bars. The blue bars that are shaded with slashes (on the left) show the results for experiments without staging, and the orange bars filled with circles (on the right) represent the results after applying staging.

Figure 9 shows that for a single agent, staging improves the overall average execution time by 10% to 25% for one level branch, and 10% to 15% for nested branches of two levels. In addition, experiments with staging behave much more robust than those without staging, as evident from significantly lower standard deviations for experiments with staging: 5-14× lower for one level and 3-10× lower when there are two nested levels of branches.

As we increase the number of agents from one to 1000, Figure 10 shows that staging still results in modest speedup, improving the overall average execution time by around

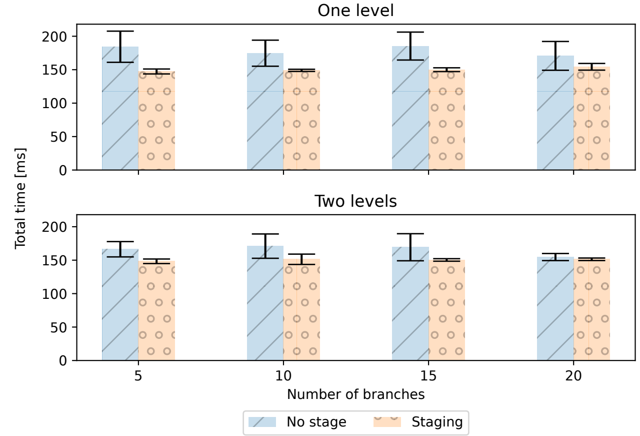

Figure 9. Increase the number of branches (one agent).

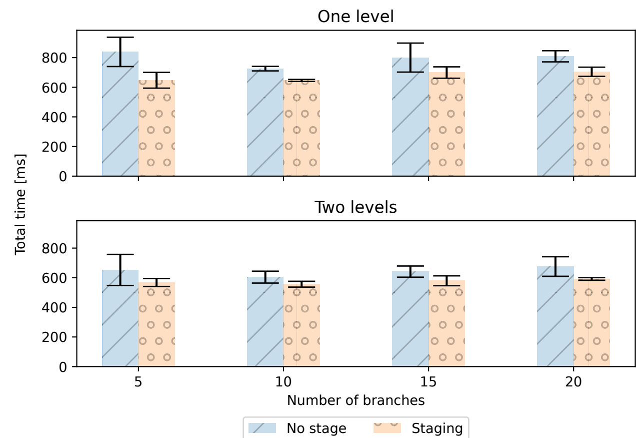

Figure 10. Increase the number of branches (1000 agents).

10%, for both one and two levels of branches. Similar to Figure 9, staging also leads to more robust performance for 1000 agents, having significantly lower standard deviation than branching, up to 10×. We note that for 1000 agents, the performance benefit of applying staging is slightly less compared with that of a single agent. This is due to the presence of other performance factors, such as increased cache contention, as the number of agents increases.

Ideally, the total time should remain unchanged (flat) as the number of branches increases with staging, after we have eliminated the overhead of dispatching. However, we see that the total time for experiments with staging increases slightly as the number of branches increases, in both Figures 9 and Figure 10. This can be caused by the overhead of locating the corresponding instructions in the array. Nevertheless, experiments with staging have overall better performance than those without staging for all experiments.

7 Related Work

Our work was initially motivated by the question of how to scale an agent-based simulation up (increase total agents) and out (increase total machines). Agent-based simulations have a long history [31] and a diverse set of simulation frameworks have been developed by domain scientists [24], but these frameworks are either single-machine [20, 22, 36] or do not scale well. To illustrate, we explain the scalability of one of the best-known distributed agent-based simulation frameworks, Repast HPC [5]. A simulation in Repast HPC is a discrete event simulation at its core: users schedule various events at different time ticks in a global schedule queue [27]. These events can be generated from agents, but can also be global events. At any time tick, a simulation proceeds by executing the scheduled events sequentially. After all such events have been processed for a given time tick, the simulation continues to the next time tick. In discrete event simulations, events that are scheduled for the same time tick are added sequentially with possible ordering constraints, hence cannot be easily parallelized.

In computer science, the bulk-synchronous parallel (BSP) processing model [37] has become standard for parallel programming [1, 8, 21, 40]: computations are performed locally over distributed data and intermediate data are combined to advance computations. This computational model also underpins vertex-centric programming, a programming paradigm for distributed graph processing: per superstep, all vertices execute computations embarrassingly parallel; the intermediate data shuffled across supersteps takes the form of messages, which are sent between vertices.

Vertex-centric programming as a programming paradigm was proposed by Google [21] and has gained wide popularity due to its simplicity and demonstrated performance. Many state-of-the-art distributed frameworks support this paradigm, which we briefly describe below.

Giraph [4] is an open-source implementation of Pregel and has been widely used by industry, including tech companies like Facebook. GraphX [39] is a graph library built on top of Spark [40], a general-purpose data flow processing framework that provides a resilient distributed dataset (RDD) abstraction that makes it easy to perform computations over distributed data. Since graph processing is an important workload in distributed analytics, GraphX provides built-in implementations of commonly used graph algorithms, which have been highly optimized to improve their performance, compared with naive implementations on Spark. One of the graph operators supported by GraphX is `Pregel`, which lets users program in a vertex-centric way just like Pregel. Flink [9] supports both graph and stream processing; Flink Gelly provides a `Pregel` operator, similar to that in GraphX, which supports vertex-centric programming for graphs. While differing in their target workloads, all such frameworks support vertex-centric programming as defined in Pregel, where users define a single `compute` function that is executed by every active vertex in each superstep.

A natural question is that given all existing graph processing frameworks that already support vertex-centric programming, why is there a need for another framework for agent-based simulations? This question is addressed in [35], where our experiments showed that the performance of these existing frameworks could differ by up to three orders of magnitude when executing a benchmark consisting of representative agent-based simulation workloads selected from population dynamics, economics, and epidemics, due to different design choices of these systems.

While applying vertex-centric programming to agent-based simulations by modeling each agent as a vertex, it became clear to us that the existing programming model in the vertex-centric paradigm is tailored for graph algorithms which often have simple vertex behavior. The `compute` method becomes tangled with branches that dispatch on the value of supersteps as the complexity of a vertex behavior increases, such as performing different actions across supersteps, which is characteristic of agent-based simulations.

The problem that a vertex program becomes convoluted and error-prone when the control flow of the vertex program gets complex has also been examined in Fregel [12, 16], a functional library for specifying vertex behavior. A Fregel program provides a functional abstraction that hides the complex control flow from users, but is then compiled to generate a vertex program with complex control flows for existing frameworks like Giraph, hence still suffering from sources of inefficiency as we have discussed in this paper.

Our work is the first to introduce a staging-time instruction to make superstep-dependent program structures explicit in an iterative vertex program, which avoids complex control flows that dispatch on the value of supersteps and mitigates sources of inefficiency in vertex programs.

We note that staging is a standard technique used to address sources of program inefficiency like dispatching overhead, and is commonly used in areas such as designing domain-specific languages (DSL) [2, 10, 11, 15, 19, 28, 32, 33] and constructing compilers to generate more efficient code [18, 23, 26, 29, 30]. A classic example that illustrates the use case of staging is the generic power function `power`(x, n) [34]:

```
1  def power(x: Int, n: Int): Int = {
2    if (n==0)
3      1
4    else
5      x * power(x, n-1)
6  }
```

This high-level generic function provides a nice abstraction that makes defining other special functions straightforward:

```
1  def square(x: Int): Int = power(x, 2)
2  def cube(x: Int): Int = power(x, 3)
```

However, this abstraction comes at the cost of performance. In particular, a low-level implementation of `square` that multiplies x with itself will be faster than invoking `power`, which computes the result recursively. Not all hope is lost though. On line 1, the body of `square` has a constant argument 2 that is known statically. Therefore, we can apply staging to rewrite the body of `square` with the partially evaluated program generated from `power`$(x, 2)$, making it possible to achieve both the high-level programming abstractions and the high-performance of low-level implementations.

The parallelism between how staging is applied in the classic `power` example and our work is clear: we similarly exploit the fact that when dispatching on the value of supersteps in vertex programs, the value of supersteps in the conditions are often known statically, which allows us to avoid such overhead by rewriting the original program with more efficient instructions that only compute useful code via staging.

8 Conclusions and Future Work

In this work, we explained the limitations of existing vertex-centric programming and described how to address such limitations by making the superstep-dependent structure in a vertex program explicit using a staging-time instruction `wait()`. We implemented this approach in an agent-based simulation framework CloudCity and demonstrated empirically that staging improves the overall performance of our experiments by 10%-25% while reducing the standard deviation of multiple runs by over 5×.

We have only scratched the surface of how staging can be applied in this work. Other use cases of staging include providing a platform-independent programming interface while emitting platform-specific instructions for heterogeneous hardware [19], generating a DSL implementation from a declarative specification [33], enabling a just-in-time compiler to call back into the running program to perform compile-time computations that define custom optimizations [30]. Each of these possible use cases directs to future works that can enable more aggressive compiler optimizations and alternative approaches for designing agent-based simulations. There is also potential for other optimizations in the future, such as applying program slicing [38] to dynamically resize the array of closures in a generated vertex program at runtime and discard closures that are no longer relevant.

More generally, vertex-centric programming is only a special case of the BSP paradigm. It is interesting to see whether the insight of introducing a staging-time barrier instruction that makes static scheduling explicit in vertex programs, as we have done in this work, can be generalized to applications in other BSP systems, especially for latency-critical systems where robust performances are crucial.

References

[1] Apache Hadoop Developers. 2006. Apache Hadoop. https://hadoop.apache.org/

[2] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*. IEEE Computer Society, Grenoble, France, 169–178. https://doi.org/10.1109/MEMCOD.2010.5558637

[3] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[4] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (8 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[5] Nicholson T. Collier and Michael J. North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *Simul.* 89, 10 (2013), 1215–1235. https://doi.org/10.1177/0037549712462620

[6] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.

[7] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31. https://doi.org/10.1145/1462166.1462167

[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, San Francisco, California, USA, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html

[9] Flink Gelly Developers. 2022. Source code of Pregel operators in Flink Gelly. https://nightlies.apache.org/flink/flink-docs-master/api/java/org/apache/flink/graph/pregel/. Accessed: 2023-03-10.

[10] JetBrain MPS Developers. [n. d.]. JetBrain MPS. https://www.jetbrains.com/mps//.

[11] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 105–116. https://doi.org/10.1145/2491956.2462166

[12] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. 2016. Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, Nara, Japan, 200–213. https://doi.org/10.1145/2951913.2951938

[13] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations (with retrospective). In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, Kathryn S. McKinley (Ed.). ACM, USA, 502–514. https://doi.org/10.1145/989393.989443

[14] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *High. Order Symb. Comput.* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

[15] Christoph Armin Herrmann and Tobias Langhammer. 2006. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.* 62, 1 (2006), 47–65. https://doi.org/10.1016/j.scico.2006.02.002

[16] Hideya Iwasaki, Kento Emoto, Akimasa Morihata, Kiminori Matsuzaki, and Zhenjiang Hu. 2022. Fregel: a functional domain-specific language for vertex-centric large-scale graph processing. *J. Funct. Program.* 32 (2022), e4. https://doi.org/10.1017/S0956796821000277

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall, USA.

[18] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/512644.512652

[19] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (2011), 42–53. https://doi.org/10.1109/MM.2011.68

[20] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Catalin Balan. 2005. MASON: A Multiagent Simulation Environment. *Simul.* 81, 7 (2005), 517–527. https://doi.org/10.1177/0037549705058073

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, Indianapolis, Indiana, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[22] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark J. Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adapt. Syst. Model.* 1 (2013), 3. https://doi.org/10.1186/2194-3206-1-3

[23] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 125–134. https://doi.org/10.1145/2517208.2517228

[24] Constantin-Valentin Pal, Florin Leon, Marcin Paprzycki, and Maria Ganzha. 2020. A Review of Platforms for the Development of Agent Systems. *CoRR* abs/2007.08961 (2020). arXiv:2007.08961 https://arxiv.org/abs/2007.08961

[25] Lionel Parreaux and Amir Shaikhha. 2020. Multi-Stage Programming in the Large with Staged Classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) *(GPCE 2020)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/3425898.3426961

[26] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[27] Repast HPC developers. 2023. Repast HPC API. https://repast.github.io/docs/api/hpc/repast_hpc/classrepast_1_1_schedule.html#details Accessed: 2023-03-02.

[28] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[29] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510. https://doi.org/10.1145/2429069.2429128

[30] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 41–52. https://doi.org/10.1145/2594291.2594316

[31] Thomas C Schelling. 1969. Models of segregation. *The American economic review* 59, 2 (1969), 488–493.

[32] Tim Sheard, Zine-El-Abidine Benaissa, and Emir Pasalic. 1999. DSL implementation using staging and monads. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*, Thomas Ball (Ed.). ACM, 81–94. https://doi.org/10.1145/331960.331975

[33] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, Indiana, USA) *(GPCE '13)*. Association for Computing Machinery, New York, NY, USA, 145–154. https://doi.org/10.1145/2517208.2517220

[34] Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 3016)*, Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky (Eds.). Springer, Germany, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[35] Zilu Tian, Peter Lindner, Markus Nissl, Christoph Koch, and Val Tannen. 2023. Generalizing Bulk-Synchronous Parallel Processing Model: From Data to Threads and Agent-Based Simulations. In *Proc. ACM. Management of Data (PACMMOD)*. ACM, USA, 439–480. https://doi.org/10.1145/3589296

[36] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, Vol. 21. Citeseer, Boston, MA, 16–21.

[37] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[38] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[39] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, Peter A. Boncz and Thomas Neumann (Eds.). CWI/ACM, USA, 2. https://doi.org/10.1145/2484425.2484427

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, San Jose, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Received 2023-07-14; accepted 2023-09-03