Using Process Calculus for Optimizing Data and Computation Sharing in Complex Stateful Parallel Computations

Zilu Tian
University of Zurich
Zurich, Switzerland
zilu.tian@uzh.ch

Dan Olteanu
University of Zurich
Zurich, Switzerland
olteanu@ifi.uzh.ch

Christoph Koch
EPFL
Lausanne, Switzerland
christoph.koch@epfl.ch

ABSTRACT

We propose novel techniques that exploit data and computation sharing to improve the performance of complex stateful parallel computations, like agent-based simulations. Parallel computations are translated into behavioral equations, a novel formalism layered on top of the foundational process calculus π-calculus. Behavioral equations blend code and data, allowing a system to easily compose and transform parallel programs into specialized programs. We show how optimizations like merging programs, synthesizing efficient message data structures, eliminating local messaging, rewriting communication instructions into local computations, and aggregation pushdown can be expressed as transformations of behavioral equations. We have also built a system called OptiFusion that implements behavioral equations and the aforementioned optimizations. Our experiments showed that OptiFusion is over 10× faster than state-of-the-art stateful systems benchmarked via complex stateful workloads. Generating specialized instructions that are impractical to write by hand allows OptiFusion to outperform even the hand-optimized implementations by up to 2×.

ACM Reference Format:
Zilu Tian, Dan Olteanu, and Christoph Koch. 2025. Using Process Calculus for Optimizing Data and Computation Sharing in Complex Stateful Parallel Computations . In *Proceedings of (SIGMOD '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Complex stateful parallel computations refer to heterogeneous parallel computations on the bulk-synchronous parallel (BSP) machine [63], where each computational unit can execute distinct code and perform in-place updates to its state [61]. The term *complex* emphasizes the distinction with homogeneous stateful parallel computations, like in the vertex-centric paradigm, where all computational units execute the same code. Despite its simplicity, the vertex-centric paradigm is used by many popular distributed systems, including Pregel [33], Giraph [17], and Flink Gelly [20]. In these systems, users specify the behavior of each vertex in an input graph by means of code. Each vertex corresponds to a computational unit. All vertices execute the same code.

SIGMOD '25, June 22–27, 2025, Berlin

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

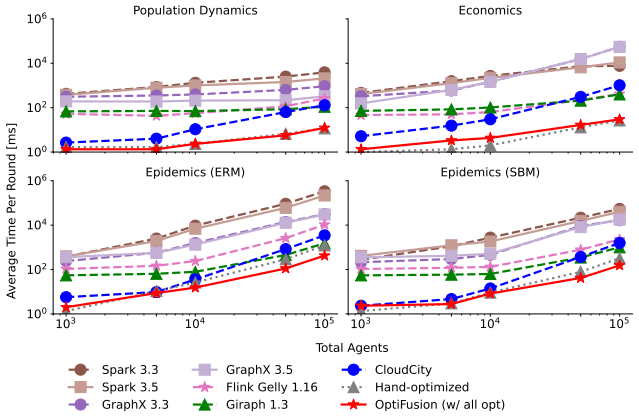


Figure 1: The performance gap between stateless parallel systems (Spark and GraphX) and stateful parallel systems (Flink, Giraph, and CloudCity) when benchmarked using complex stateful workloads remains. The workloads are simulations for population dynamics, economics, and epidemics where the social graph is generated from the Erdős-Rényi random graph model (abbreviated ERM) and the stochastic block random model (abbreviated SBM) respectively. These experiments reproduce the scale-up results in [61] using the latest software versions. By exploiting data and computation sharing optimizations, OptiFusion is over 500× faster than stateless systems and 10× faster than stateful systems. The performance of OptiFusion is comparable with and can even be better than the hand-optimized implementations by 2×.

Agent-based simulations are a prime example of complex stateful parallel computations. These simulations consist of concurrent agents interacting within a virtual world, each with its own state and code. They are highly flexible and have been extensively adopted across diverse fields in the social sciences, such as economics and epidemiology, where they serve as crucial tools for modeling complex phenomena [3, 12, 21, 30, 61].

The computational model of agent-based simulations is based on the BSP model. Agent computations proceed in a sequence of rounds, separated by global synchronizations. Agents interact by sending messages. Per round, each agent independently processes messages received from other agents, updates its local state, and sends messages to other agents. Messages are collected and delivered by the underlying agent-based simulation framework at the end of a round and arrive at the mailbox of the receiving agents at the beginning of the following round.

Prior work [61] showed that stateful parallel systems can outperform stateless parallel systems, such as Spark [70] and GraphX [25], by up to 100× when benchmarked on agent-based simulations, owing to different system design features like in-place updates. At the time of writing, Spark and GraphX have been updated from version 3.3 to 3.5.[1] A natural question is whether the latest versions of stateless parallel systems still suffer from a similar performance degradation compared with stateful systems when benchmarked using the same complex stateful workloads.

In Figure 1, we repeat the scale-up experiments in [61] using the latest software and show the results for Spark (versions 3.3 and 3.5), GraphX (versions 3.3 and 3.5), Flink Gelly (version 1.16), Giraph (version 1.3), and CloudCity (SNAPSHOT-2.0).[2] As the number of agents increases, the previously reported performance gap between stateless and stateful parallel systems remains evident across all workloads in the benchmark.

Figure 1 also shows the performance of our system OptiFusion and the hand-optimized implementations. OptiFusion can be 10x faster than state-of-the-art stateful parallel systems like Giraph, Flink Gelly, and CloudCity. Furthermore, OptiFusion generates specialized instructions for each agent through partial evaluation, which is impractical to write by hand when the number of agents is large. This allows OptiFusion to outperform even hand-optimized implementations by up to 2× as the number of agents increases.

OptiFusion is a program specialization framework that transforms generic programs, which are easy to define, into specialized programs that are efficient to execute. The program transformations are based on a novel formalism called *behavioral equations*, based on a foundational process calculus, π-calculus [34, 36, 37]. We design this language using the following desiderata.

- *Expressing State-based Dynamic Behavior.* State-based dynamic behavior captures how an individual agent's behavior changes concerning states. For this reason, we also refer to the language as a *calculus*. This allows a system to identify points in an agent program where data and computation can be reused or shared across agents.
- *Expressing Non-deterministic Concurrent Behavior.* This allows for optimizations that exploit the non-deterministic execution order of parallel agents in distributed systems for optimizations, such as merging parallel agents to match available hardware parallelism to improve efficiency.
- *Expressing Agent Computations.* The calculus should have high-level primitives that allow users to express the computation of individual agents easily. This simplifies optimizations that exploit algebraic properties, such as associativity and commutativity, to transform computations.
- *Expressing Agent Interactions.* Fine-grained interactions are central to the semantics of parallel programs. This requirement allows for optimizations that exploit and transform the communication patterns of agents, like transforming computations for processing messages into local computations over synthesized message data structures.
- *Expressing Data and Computation Placement.* The calculus should be able to express data and computation placement, enabling locality-based optimizations such as aggregation pushdown, where data is combined locally on each node in a distributed system and partial results are aggregated.

In summary, this paper makes the following contributions.

- We introduce a formalism named *behavioral equations* based on π-calculus for expressing and optimizing complex stateful parallel computations. We also introduce annotations to express data and computation placement. The syntax and semantics of the language are presented in Section 2.
- We demonstrate the generality and usability of behavioral equations by expressing a host of data- and computation-sharing optimizations as transformations of behavioral equations in Section 3.
- We build OptiFusion, a compile-time program specialization framework based on behavioral equations. The optimizer in OptiFusion exploits the aforementioned optimizations to generate specialized programs. We explain the system design and implementation details in Section 4.
- Finally, we evaluate the effectiveness of program specializations. Our experiments show that the performance of the generated programs is on-par with or up to 2× faster than hand-optimized programs for all workloads in an agent-based simulation benchmark. OptiFusion can be 50× faster than other systems like CloudCity, Giraph, and Flink Gelly.

2 BEHAVIORAL EQUATIONS

This section introduces *behavioral equations*, a language based on the π-calculus for modeling complex stateful parallel computations in bulk-synchronous parallel (BSP) systems [63]. To ease the formulation of behavioral equations, we start by explaining the BSP model and presenting a gentle introduction to π-calculus, before detailing the syntax and semantics of behavioral equations. We also introduce partition-annotated behavioral equations, which express data and computation placement by annotations.

2.1 BSP Model

The BSP model is an abstract parallel machine that underpins many distributed frameworks designed for efficient and scalable parallel computing [17, 25, 33, 56, 61, 70]. This abstract machine is made of:

- A set of processors. Each processor is a core-memory pair with private memory. A processor updates values stored in the memory locally. In addition, processors can communicate via sending and receiving messages.
- A synchronization facility to synchronize all processors periodically. Synchronizations divide the parallel computation of processors into a sequence of *supersteps*. Per superstep, a processor performs arbitrarily many *steps*, in the form of updating local values, processing received messages, and sending messages. Messages are delivered at the end of a superstep and arrive at the beginning of a superstep.

For simplicity, we assume messages arrive at the beginning of the next superstep in our discussion below.

[1] Giraph has retired [58], and Gelly only supports Flink up to version 1.16 [57].
[2] The slight performance variation compared with [61] is due to different hardware configurations: the base clock frequency of our server is 2.2 GHz, but 2.7 GHz in [61].

(CHOICE-COMM)	$P + Q \equiv Q + P$
(CHOICE-ASSOC)	$(P + Q) + R \equiv P + (Q + R)$
(CHOICE-IDENT)	$P + 0 \equiv P$
(PAR-COMM)	$P \mid Q \equiv Q \mid P$
(PAR-ASSOC)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
(PAR-IDENT)	$P \mid 0 \equiv P$
(RES-SWAP)	$(va)(vb)P \equiv (vb)(va)P$
(RES-SCOPE)	$(va)(P \mid Q) \equiv P \mid (va)Q \quad$ if $a \notin \text{fn}(P)$
(RES-ANN)	$(va)0 \equiv 0$
(REPLICATION)	$!P \equiv P \mid !P$
(ALPHA-CONV)	$P \equiv P[a/x] \quad$ if $a \notin \text{name}(P), x \in \text{bn}(P)$.

Figure 2: Structural congruence rules for π-calculus.

2.2 π-Calculus Primer

The *π-calculus* provides a mathematical framework for modeling interactions in concurrent systems. There are two basic entities, names and processes.[3] A *name* represents a communication channel (abbreviated as channel) or value. A *process* interacts with other processes by sending and receiving names along channels. Processes can be composed in parallel with other processes, create new names, and spawn new processes. Below, we present the syntax and semantics of π-calculus, followed by a concrete example.

2.2.1 Syntax. Let X be the set of names, $x, a \in X$. A process is built from names, using *prefix actions* of the form $a(x)$ and $\overline{a}x$, and operators for *choice*, *parallel composition*, *restriction*, and *replication*. These operators are represented by symbols +, |, v, and !, respectively. More concretely, the syntax of a process expression is summarized in the following BNF grammar:

$$P, Q ::= 0 \mid \overline{a}x.P \mid a(x).P \mid P + Q \mid P|Q \mid (va)P \mid !P$$

where

- 0 represents the inactive process.
- $\overline{a}x.P$ is a process waiting to send name x on the output channel $\overline{a}$ (bar represents the outgoing direction) before continuing as process P.
- $a(x).P$ is a process waiting to receive a name on the input channel a before binding the received name to x in P and proceeding as process P. If the name x is not free in P, then alpha-conversion is required.
- $P + Q$ is a process that can take part in either P or Q for communication. The process does not commit to any alternative until communication happens; once it does, the occurrence precludes the other alternative.
- $P|Q$ is a process that consists of P and Q that run in parallel.
- $!P$ replicates arbitrarily many copies of process P.
- $(va)P$ creates a new name a that is restricted to, that is, known only by, the prefixed process P.

[3]Many versions of π-calculus exist [34, 36, 38]. This introduction is based on [36].

2.2.2 Equivalence. To simplify the presentation of semantics, we introduce an equivalence relation (denoted ≡) between processes – structural congruence [36] – defined by the rules in Figure 2.

Observe that Rules **RES-SCOPE** and **ALPHA-CONV** contain references to $fn(P)$, $bn(P)$, and $name(P)$. The first two are functions that return the set of names in process P that are *free* and *bound* respectively. A name x is bound if it appears in the restriction operator vx or as the name for the received value of an input channel $a(x)$ in the prefix action; otherwise, it is free. The function $name(P)$ returns the set of all names, $name(P) = fn(P) \cup bn(P)$.

We briefly explain the rules in Figure 2. Operators + and | are commutative and associative (**CHOICE-COMM**, **CHOICE-ASSOC**, **PAR-COMM**, **PAR-ASSOC**). Interacting with the inactive process 0 does not change the behavior of the other process (**CHOICE-IDENT**, **PAR-IDENT**). New names generated for the same prefixed process can be swapped (**RES-SWAP**). The scope of a new name for two parallel processes can be restricted to one process if the name is bound in the other process (**RES-SCOPE**). Creating a new name in the inactive process has no effect (**RES-ANN**). The replication operator can create a new process (**REPLICATION**). Finally, two processes are structurally congruent if they only differ by a change of bound names (**ALPHA-CONV**). $P[a/x]$ denotes the process P in which name a is substituted for the bound name x.

2.2.3 Semantics. We explain the meaning of π-calculus processes with reduction rules. We use $P \rightarrow Q$ to denote that *process P can perform a computation step and be transformed into process Q*. Every computation step in π-calculus requires the interaction between two processes, captured by the communication rule [36]:

(Communication) $(\ldots + \overline{a}x.P) \mid (\ldots + a(y).Q) \rightarrow P \mid Q[x/y]$

The communication rule has two key aspects. First, communication occurs between two complementary parallel processes, where one is waiting to send a name x along the output channel $\overline{a}$ and the other is waiting to receive a name along the input channel with the same name a. Second, once the communication occurs, other possible communications – shown as ... in the rule – are discarded.

The communication rule is the only axiom. Other reduction rules are inference rules, shown below:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \frac{P \rightarrow P'}{(vx)P \rightarrow (vx)P'} \qquad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

From left to right, the rules state that (a) reductions can occur under parallel composition and (b) restriction. (c) Structurally congruent processes have the same reduction.

2.2.4 Example. For a minimal example, we model the abstract behavior of a DRAM cell: a cell can store one value; reading is destructive and erases the cell's current value. The cell behavior can be modelled as a process B:

$$B \stackrel{def}{=} i(x).(\overline{o}x.B + B)$$

By the convention of π-calculus, RHS is a process expression and LHS is a process identifier. A process identifier is purely syntactic and is substituted by the RHS expression during reductions.

Process B waits to receive a name along the input channel i and substitutes the received name for the bound name x in the prefixed expression. Once it receives the value, B has two options: (a) sends

the received name along the output channel $\overline{o}$ before continuing as B, and (b) continues as B.

We now demonstrate how interactions transform π-calculus processes through a sequence of reductions. Assume a user process that writes 5 and 6 sequentially to an empty cell and reads from it. A system that consists only of the user process and an empty cell is modelled as follows. The ν operator ensures that names i and o are known only to the user process and the cell. Reduction arrows are labelled with interacting actions for clarity.

$$
\begin{aligned}
&(\nu i)(\nu o)(i(x).(\overline{o}x.B + B) \mid \overline{i}5.\overline{i}6.o(x).0) \\
&\equiv i(x).(\overline{o}x.B + B) \mid \overline{i}5.\overline{i}6.o(x).0 \\
&\xrightarrow{i(x)|\overline{i}5} (\overline{o}5.B + i(x).(\overline{o}x.B + B)) \mid \overline{i}6.o(x).0 \\
&\xrightarrow{i(x)|\overline{i}6} (\overline{o}6.B + B) \mid o(x).0 \\
&\xrightarrow{\overline{o}6|o(x)} B \mid 0 \equiv B
\end{aligned}
$$

The example demonstrates how to express concurrent behavior and fine-grained interactions using π-calculus. However, expressing functions that describe high-level computations is nontrivial and often results in convoluted expressions [35]. We address this by behavioral equations, which lets the users program using high-level abstractions like *states* and *functions*. A behavioral equation is a macro that generates π-calculus processes.

2.3 Syntax of Behavioral Equations

A *state* p represents a core-memory pair in a BSP processor. It consists of a value and a unique state name. The basic building block for programming states is called *behavioral equation*.

Intuitively, for a state p, users should be able to specify how to transit to another state declaratively, based on values of p and optionally other states. We capture this programming model using *behavior equations*, expressed as:

$$
p \xrightarrow{f\{i_1,\ldots,i_n\}} q, \tag{1}
$$

where p, q are state names, f is a user-defined function for computing the value of q, and $\{i_1, \ldots, i_n\}$ is a reference set that contains state names i_1 to i_n needed for computing q. Equivalently, we can express Equation (1) in the alternative representation below:

$$
p := f\{i_1, \ldots, i_n\}.q, \tag{2}
$$

following the π-calculus convention. The symbol := means *is defined as*. Unlike a process identifier, a state name in the LHS cannot be substituted by the expression on the RHS during reductions. The equation reads "*The behavior of state p is to obtain values from states $i_1, \ldots, i_n$, apply function f over the obtained values and its value, initialize state q with the computed result, and behave like state q.*" The interactions among states in behavioral equations are made precise using π-calculus. Behavioral equations let users program with high-level abstractions like *functions* and *states* in π-calculus.

2.4 Semantics of Behavioral Equations

We translate behavioral equations into π-calculus process expressions to make precise the *interaction* among concurrent states. Let $\mathcal{B}$ be the set of behavioral equations and $\mathcal{E}$ be the set of π-calculus process expressions. We define a meta-algorithm $\beta : \mathcal{B} \to \mathcal{E}$ that translates a behavioral equation into a process expression.

2.4.1 Semantics of Non-recursive Behavioral Equations. We first consider non-recursive behavioral equations. Let $b \in \mathcal{B}$ be a non-recursive behavioral equation like in Equation (2), $p \neq q$. Translating b to π-calculus is straightforward, shown below using the syntax of the programming language Scala[4]:

$$
\begin{aligned}
&b \textbf{ match case } p := f\{i_1, \ldots, i_n\}.q \textbf{ if } p \neq q \Rightarrow \\
&p(x).(\nu d)(\nu m)(&& (3)\\
&i_1(m).\overline{d}m.0 \mid \ldots \mid i_n(m).\overline{d}m.0 \mid && (4)\\
&d(m_1)\ldots d(m_n).(\nu y)[\![y = f(m_1, \ldots, m_n, x)]\!].!\overline{q}y.0) && (5)
\end{aligned}
$$

For clarity, we show the translated π-calculus expression in three lines, labeled as equations (3), (4), and (5),[5] explained below.

- Equation (3) initializes the value of state p and creates new names d and m for processes in Equation (4) and Equation (5) to communicate privately;
- Equation (4) receives values from states in the reference set non-blockingly by parallel composition, and forwards received values to Equation (5) on private channels; and
- Equation (5) computes f and sends the computed value to state q on the RHS. The expression $[\![y = f(m_1, \ldots, m_n, x)]\!]$ denotes a π-calculus process that applies f to values m_1, ..., m_n, x, binding the result to y. The encoding details are in [35]. The process $!\overline{q}y$ sends y on the channel $\overline{q}$ many times to initialize the state q on the RHS of b and inform other processes of the value of q.

Non-recursive behavioral equations can be readily composed to model iterative parallel computations of a BSP system. Though formal proof is beyond the scope of this paper, we illustrate how to model computations in a BSP system by composing behavioral equations through the following example.

Example 2.1. Consider a minimal BSP system with two BSP cores, labelled core 1 and 2, initially in states p_1 and p_2 with values 5 and 6 respectively. Per superstep, each core sends its value to the other core and applies functions f and g respectively to its value and received values.

For demonstration, we only show computations in two supersteps. Additionally, we assume that cores send and receive messages at the beginning of the first superstep for simplicity. Computations in this BSP system can be modelled as the following process S:

$$
\begin{aligned}
S \stackrel{def}{=} (\nu p_5)(\nu p_6)((\nu p_3)(\nu p_4)((\nu p_1)(\nu p_2)(!\overline{p_1}5 \mid !\overline{p_2}6 \mid [\![p_1 := f\{p_2\} \\
.p_3]\!] \mid [\![p_2 := g\{p_1\}.p_4]\!]) \mid [\![p_3 := f\{p_4\}.p_5]\!] \mid [\![p_4 := g\{p_3\}.p_6]\!])).
\end{aligned}
$$

The restriction operator in νp_i ensures the state p_i is visible only in the corresponding superstep. The process $(!\overline{p_1}5 \mid !\overline{p_2}6)$ initializes states p_1 and p_2 with values 5 and 6. $[\![p_1 := f\{p_2\}.p_3]\!]$ denotes the π-calculus expression translated from the enclosed non-recursive behavioral equation.[6] Each update to a BSP core is captured by a behavioral equation. Equations $p_1 := f\{p_2\}.p_3$ and $p_2 := g\{p_1\}.p_4$

[4] Scala keywords are highlighted in **boldface**.

[5] An alternative presentation is to name the expressions in Equation (4) and Equation (5) with process identifiers L and M, and showing Equation (3) as $p(x).(\nu d)(\nu m)(L \mid M)$. These two presentations are identical.

[6] We overload the meaning of the notation $[\![\,]\!]$ compared with its usage in Equation (5).

model state updates in the first superstep in cores 1 and 2 respectively. Similarly for the second superstep.

2.4.2 Semantics of Recursive Behavioral Equations. Recursive behavioral equations are of the form $p := f\{i_1, \ldots, i_n\}.p$, where p is on both the LHS and RHS. This causes problems when composing equations in parallel to model a BSP system. For instance, substituting non-recursive behavioral equations in Section 2.4.1 with recursive behavioral equations for p_1 and p_2 results in process S':

$$S' \stackrel{def}{=} (\nu p_1)(\nu p_2)(!\overline{p_1}5 \mid !\overline{p_2}6 \mid [\![p_1 := f\{p_2\}.p_1]\!] \mid [\![p_2 := g\{p_1\}.p_2]\!])$$

which no longer models computations of the BSP system in Example 2.1. For starters, S' cannot be reduced to an irreducible form – a process expression where no reduction rules defined in Section 2.2 can be applied – after finitely many reductions. S' models computations that run indefinitely, instead of two supersteps.

In addition, S' introduces undesirable non-determinism, allowing states p_1 and p_2 to receive obsolete state values from past supersteps, as opposed to the most recent value: In Equation (5), $!\overline{q}y$ sends the computed value y many times on $\overline{q}$, to initialize the state of After computing the updated value of p_1, the result y is again sent out on the same channel $\overline{p_1}$, in parallel with the previous value a sent out on the same channel.

To address this, we let the system synchronize the BSP states by introducing new actions *yield* and *resume*. Per superstep, the BSP system sends messages to each state on the channel $\overline{resume}$ with values needed by each state. At the end of a superstep, each state sends its name p, computed value y, and names that it wants to obtain values of, to the system on the channel $\overline{yield}$.[7] The meta-algorithm is updated with the following case expression:

case $p := f\{i_1, i_2, \ldots, i_n\}.p \Rightarrow P$,

$$P \stackrel{def}{=} resume(m_1, \ldots, m_n).p(x).$$
$$(\nu y)[\![y = f(m_1, \ldots, m_n, x)]\!].(\overline{yield}(p, y, i_1, \ldots, i_n).P \mid \overline{p}y.0)$$

The notation $[\![y = f(m_1, \ldots, m_n, x)]\!]$ has the same meaning as in Equation (5): a process expression that encodes the function application of f and binding the result to y. The process $\overline{p}y.0$ stores the calculated value to initialize the value of p in the next superstep after receiving *resume* from the system.

2.5 Annotations of Behavioral Equations

Expressing physical properties like data and computation placement allows the system to distinguish between different copies of a value on different machines but sharing the same state name, which is needed for expressing optimizations that exploit data locality.

We distinguish states on the same machine from those on different machines by introducing the *partition* abstraction. Conceptually, a partition is another BSP machine with a set of BSP cores. In this regard, our computational model where states are separated into different partitions, can be described as the slightly generalized *partitioned or hierarchical BSP model* [10, 11, 16, 45].

Each partition has a unique identifier. We annotate each state with its partition identifier p_i, expressed as $p@p_i$, and refer to such equations as *partition-annotated* or *annotated* behavioral equations.

[7] *yield* and *resume* are *polyadic* names – channels that can send and bind multiple values. Interested readers can refer to [36] for a detailed explanation.

Note that partition-annotated behavioral equations achieve all the requirements in the desiderata in Section 1: We can now express state-based dynamic behavior, non-deterministic concurrent behavior, agent computations, agent interactions, and data and computation placement, using annotated behavioral equations.

3 OPTIMIZATION EXAMPLE

In this section, we demonstrate the generality and usability of behavioral equations by expressing various data-sharing and computation-sharing optimizations as transformations over behavioral equations, through a concrete example.

Example 3.1. Let A_1 be an agent that aggregates values from agents A_2 to A_4 and updates its value v to the minimum of v and one plus the minimum of the received values, as described in op:

$$op(v, m_1, m_2, m_3) = min(v, 1 + min(m_1, m_2, m_3)).$$

Assume A_1 is in state x_1, and its neighbors are in states x_2, x_3, and x_4. The behavioral equation for x_1 is expressed as:

$$x_1 := op\{x_2, x_3, x_4\}.x_1'. \tag{6}$$

Figure 3a illustrates Equation (6) as a *computation tree*. The root node is $x'_{1_{op}}$, where op is the user-defined function in Equation (6). The leaves consist of names in the reference set together with the state name on the LHS of Equation (6). The operators $remote()$ and $local()$ are code generators inserted automatically by the system. Data flows from the bottom up. Nodes at the same height are unordered and can be evaluated independently. Each node evaluates to a state name, allowing such operators to be nested and composed. Below we present various data-sharing and computation-sharing optimizations, and show how they transform Figure 3a.

Compile Local Messages Away. Users do not make assumptions about the placement of an agent, defining a generic program applicable to all agents. By default, the system assumes remote communication between agents, as shown in Figure 3a. This optimization enables a system to analyze the partition information of agents and rewrite $remote()$ to $local()$ when possible, as illustrated in Figure 3b.

Merging. Multiple computation graphs can be merged into one computation graph. Duplicated nodes are removed and their edges are redirected to the unified node, preserving connectivity. This can greatly improve data sharing among nodes in the graph.

Figure 3c shows how to merge computation trees for x_1 and x_2. The behavioral equation for x_2 is defined like follows:

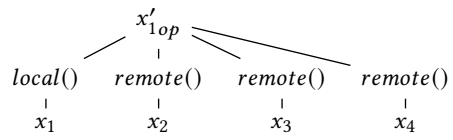
$$x_2 := op\{x_3, x_4\}.x_2'$$

State x_2 aggregates values from x_3 and x_4 in the same way as x_1, before updating its state.

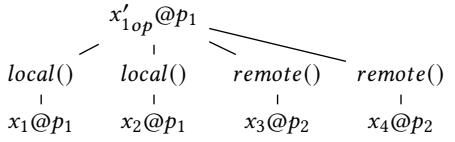
Aggregation pushdown. This optimization aggregates messages locally at remote machines and sends processed results to an agent, to reduce the amount of intermediate data shuffled in the network and to distribute computations for a better load balance, shown in Figure 3d.[8] The operator $d_{1_{op}}@p_2$ creates a state d_1 dynamically[9]

[8] Applying this optimization requires the computation for processing messages to be commutative and associative, which is satisfied by $min()$ in op.

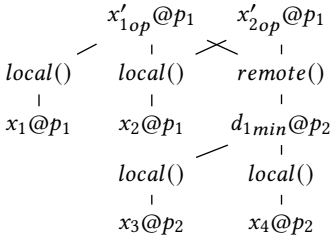
[9] The term "dynamic" refers to the fact that the state is created by the system during the optimization phase rather than defined by the user in behavioral equations.



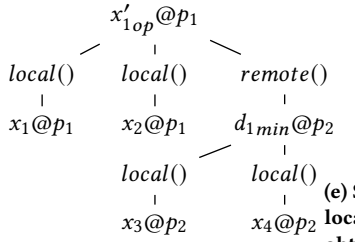
(a) Visualize $x_1 := op\{x_2, x_3, x_4\}.x'_1$ as a computation tree, where $op(v, m_2, m_3, m_4) = min(v, 1 + min(m_2, m_3, m_4))$. x_1 is the reference of state x_1 and x'_1 is the reference of state x'_1. x_2 to x_4 are state references of neighboring agents.



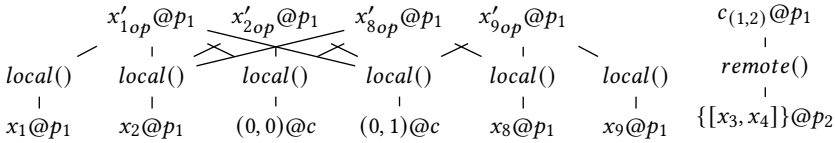
(b) Knowing the graph partition after loading an input graph allows the system to compile local messages away. Here states referenced by x_1 and x_2 are in graph partition p_1, and states referenced by x_3 and x_4 are in partition p_2.



(c) Merging agents.



(d) Aggregation pushdown.



(e) Synthesize message data structures and transform computations for remote communication into local computations over the synthesized data structures. Compared with (b), the *remote* operators for obtaining values of x_3 and x_4 when constructing x'_1 are changed into *local*, to interact with a locally synthesized message data structure c instead.

Figure 3: Illustration of how various optimizations transform computation trees.

to store the partial result obtained after processing messages locally in partition p_2 and sends the value to x_1.

Synthesize Cross-Partition Message Data Structures. This optimization minimizes messaging overhead by aggregating communications at the partition level. By analyzing cross-partition edges,[10] the optimizer can merge boundary agents for each remote partition and create a fixed data structure, called "cache", that serves as a placeholder for the values of boundary agents adjacent to a given partition. These agents are ordered by their ids within the cache to enable efficient offset-based lookups.

In Figure 3e, the operator $c_{(1,2)}@p_1$ creates a cache data structure with a cache reference c in the graph partition p_1, as a placeholder for remote messages from partition p_2 of size 1×2 with message schema $[x_3, x_4]$, shown as $\{[x_3, x_4]\}@p_2$. The first value stored in c is received from x_3, and the second is from x_4.

Transform Remote Communication into Local Computation. Finally, a system can transform instructions for remote communication, such as sending or processing messages to and from agents in other partitions, into local computations over locally synthesized message data structures for better efficiency.

For instance, in Figure 3b, constructing x'_1 needs the value of x_3 and x_4, which are on different partitions, through the *remote* operators. In Figure 3e, the *remote* operators are transformed into *local* when constructing x'_1 and interact with the locally synthesized message data structure c, which is called *local computation*.

More generally, after synthesizing message placeholders based on the partition structure for the value of cross-partition agents, an optimizer rewrites the computation trees of the boundary agents to interact with the synthesized data structure. A *boundary* agent sends or receives at least one message from agents in other partitions.

In Figure 3e, we introduce two additional behavioral equations,

$$x_8 := op\{x_2, x_3\}.x'_8 \quad x_9 := op\{x_8, x_4\}.x'_9$$

Since x_3 and x_4 are shared by multiple states and processed differently, we do not push computation to the sender and simply synthesize a cross-partition message data structure c and rewrite the behavior of states x_1, x_2, x_8, x_9 to interact with the synthesized message data structure c: the behavior of x_1 is transformed to look up the value of c at offset $(0, 0)$, shown as $(0, 0)@c$, rather than receiving from x_3. The rewrites for x_8 and x_9 are similar.

4 OPTIFUSION

To demonstrate the application of behavioral equations in optimizing data and computation sharing for real stateful parallel systems, we developed OptiFusion, a Scala-based prototype integrated as a library in CloudCity [61], a stateful parallel system designed for distributed agent-based simulations

Figure 4 shows the overall system architecture of OptiFusion, which has three layers: frontend, optimizer, and backend. In the frontend, users specify agent behaviors using behavioral equations and partition structures. The optimizer then transforms behavioral equations and partitions to improve performance through various optimizations. The optimized partitions are mapped to the agents in CloudCity in the backend for parallel executions.

4.1 Frontend

The frontend leverages both non-recursive and recursive behavioral equations. Users specify the behavior of an agent in one superstep,

[10] While this approach uses fixed data structures to manage messages based on a static graph structure, it does not restrict the system to static communication. The system supports dynamic updates, allowing agents to change their references to connected neighbors and request values from new ones. In such cases, it reverts to default agent-to-agent messaging, where messages are exchanged directly between agents.

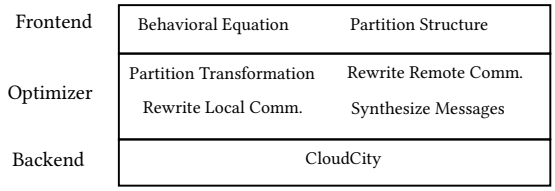


Figure 4: System architecture of OptiFusion.

which is applied repeatedly.[11] The behavior of an agent is decomposed into fine-grained combinator functions, which synthesize *non-recursive behavioral equations* that describe computations of an agent in a superstep. This allows our system to apply optimizations in Section 3. Iterative computations across supersteps are modelled by *recursive behavioral equations*, again synthesized by our system, where states are updated in place. Behavioral equations are represented as types and optimizations that transform behavioral equations are expressed as type-level operations, thanks to Scala language features such as abstract type members, explicit self-types, and mixin compositions [41].

Behavioral Equation. At the user level, we name our interface using the more familiar term `BSP`, shown below. A `BSP` instance represents a recursive behavioral equation $p := f\{i, j\}.p$. The name and initial value of the state p are shown on lines 2 and 3 respectively. The reference set is shown on line 4.

```
1  trait BSP { this: ComputeMethod =>
2    val ref: Ref
3    var value: Value
4    val receiveFrom: Iterable[Ref]
5    def run(ms: Set[InMessage]): Unit
6  }
```

The user-defined function f in a behavioral equation is modelled as an instance of `ComputeMethod` in our programming model. On line 1, the explicit self-type annotation `this : ComputeMethod` means that an instance of `BSP` needs to be mixed in [41] with an stance of `ComputeMethod`. The abstract type members and combinators defined in `ComputeMethod` are accessible within `BSP`.

The `run` method on line 5 is *generated* by the system and goes through different transformations during the optimization. The agent state value is updated in-place in the generated `run` method.

User-defined Function. We model a user-defined function as an instance of `ComputeMethod`, using combinators (lines 6–10) to capture the computation structure of the function and abstract type members (lines 2–4) to provide a uniform interface that hides the heterogeneity of type members.

```
1  trait ComputeMethod {
2    type Value
3    type InMessage
4    type OutMessage
5
6    def stateToMessage(s: Value): OutMessage
7    def partialCompute(ms: Set[InMessage]): Option[
       InMessage]
8    def updateState(s: Value, m: Option[InMessage]):
       Value
9    def deserialize(OutMessage): InMessage
10   def run(s: Value, ms: Set[InMessage]): Value = {
11     updateState(s, partialCompute(ms))
12   }
13 }
```

The `stateToMessage` combinator (line 6) transforms the value of an agent state into a message before sending it to other agents, to avoid redundant computations by pushing message-processing computations on the sender side. The `partialCompute` combinator (line 7) enables aggregation pushdown where locally received messages can be aggregated locally into a new message before being sent through the network. The combinator `updateState` (line 8) computes the updated value based on the current value and aggregated value of received messages declaratively without updating the agent's value in-place, to allow for later optimizations. The `deserialize` combinator on line 9 allows users to decouple the serialized format of a message from an input value type used for the computation. The `run` function (line 10) expresses how to compute the new state value, with a default implementation (line 11).

Example. We illustrate how to use our new programming model for complex stateful computations via a concrete example, the population dynamics simulation based on Conway's Game of Life. Each agent represents a grid cell and the world (2D grid) is represented implicitly by a collection of agents. The state of an agent is a Boolean value that denotes whether a cell is alive. The state of an agent evolves in response to the state of its eight adjacent neighbors:

- if an agent has less than 2 or more than 3 alive neighbors, then it dies due to under-population or over-population respectively;
- if an agent has exactly three alive neighbors, then it is alive due to reproduction;
- otherwise, the state of an agent remains unchanged.

Per superstep, each agent receives messages from adjacent neighbors that contain their current states and updates its state according to the aforementioned rules.

Figure 5 shows how to implement this example in OptiFusion using `BSP` (lines 1–4) and `ComputeMethod` (lines 6–28). On line 1, the trait `GoLCompute` that model the function for updating the state of an agent is mixed-in. On line 3, `FixedCommunication` is an annotation in OptiFusion that indicates the communication pattern remains fixed across supersteps, which allows for optimizations.

The combinators in `GoLCompute` allow the system to easily transform and optimize the program. In this example, each agent needs to count the number of alive neighbors before applying its state update rule. Hence, each agent transforms the value of a neighbor from a Boolean into an integer (1 for true and 0 for false) and sums up the received values. In `stateToMessage` (lines 26–28), we factor out the transformation from a Boolean into an integer. The `partialCompute` combinator (lines 11–15) specifies how to partially aggregate the number of alive neighbors. The `updateState` combinator defines how to apply the update rule. The `deserialize` combinator is defaulted to the identity function when the `InMessage` and `OutMessage` are the same, as is the case here.

The core of OptiFusion is to transform a generic agent program like in Figure 5 into specialized programs that are efficient to execute

[11] This is a standard programming model in the vertex-centric paradigm, used by frameworks like Pregel and Giraph.

```
1  class Cell(ref: Ref, neighbors: Seq[Ref]) extends BSP
       with GoLCompute {
2    var value: Boolean = Random.nextBoolean()
3    val receiveFrom = FixedCommunication(neighbors)
4  }
5
6  trait GoLCompute extends ComputeMethod {
7    type Value = Boolean
8    type InMessage = Int
9    type OutMessage = Int
10
11   def partialCompute(m1: Iterable[Int]):Option[Int]=
12     m1 match {
13       case Nil => None
14       case _ => Some(m1.fold(0)(_+_))
15     }
16
17   def updateState(s: Boolean, m: Option[Int]) =
18     m match {
19       case None => s
20       case Some(n) =>
21         if (n == 3) true
22         else if (n < 2 || n > 3) false
23         else s
24     }
25
26   def stateToMessage(s: Boolean): OutMessage =
27     if (s) 1 else 0
28 }
```

Figure 5: Population dynamics example in OptiFusion: Agent definition using `BSP` and `ComputeMethod` abstractions.

by exploiting the partition structure of an input data graph. Next, we explain how to represent the partition structure.

Partition Structure. In addition to generic agent definitions, users specify an input data graph and its partition structure when initializing a simulation using the abstraction `Partition`, shown below.

```
1  trait Partition {
2    type NodeId
3    type Member
4
5    val id: PartitionId
6    val topo: Graph[NodeId]
7    val members: Set[Member]
8  }
```

A partition has a unique id (line 3) and a set of members (lines 3, 7). The graph structure is captured in `topo` (lines 2, 6), where the `Graph` abstraction contains cross-partition edges.

To see how to use the `Partition` abstraction, we continue with the population dynamics example and show the end-to-end initialization program in Figure 6.[12] OptiFusion provides a graph library for generating different graphs, including 2D torus and stochastic block random graph models. On line 1, `graph` creates an in-memory data graph corresponding to the social network graph of the population dynamics example. After this, users transform each vertex in the data graph into a `Cell` agent (line 2). The `partition` function (line 3) is a library function in OptiFusion that separates the data

[12]The initialization program is for a single-machine multi-threaded simulation. In the distributed setting, the initialization program is different.

```
1  val graph = GraphFactory.torus2D(width, height)
2  val cells: Map[Int, BSP with ComputeMethod] = graph.
       adjacencyList.map(i => (i._1, new Cell(i._1, i.
       _2)))
3  partition(graph, components).zipWithIndex.par.map(i
       => {
4    val part = new Partition {
5      type Member = BSP with ComputeMethod
6      type NodeId = Ref
7
8      val id = i._2
9      val topo = i._1
10     val members = i._1.vertices.map(j => cells(j)).
         toList
11   }
12   Optimize.default(part)
13 }).map(i => bspToAgent(part)).seq
```

Figure 6: Population dynamics example in OptiFusion: Specifying the partition structure using `Partition`.

graph into a given number of partitions, where each partition can be easily mapped to a `Partition` structure (lines 4 – 11).[13]

Our optimizer exploits the partition structure to generate specialized programs that are efficient to execute. This can be seen on line 12: `Optimize.default` transforms a partition through the default optimization phases, explained in Section 4.2. Finally, on line 13, each transformed specialized partition is mapped into an agent in CloudCity through the connector function `bspToAgent`.

4.2 Optimizer

The optimizer rewrites partitions through a sequence of type-safe transformations, as shown below.

```
1  trait Optimizer[T <: Partition, V <: Partition] {
2    def transform(part: T): V
3  }
```

The optimizer is parameterized with two types, `T` and `V`, both are constrained to extend `Partition` (as seen from T <: Partition on line 1). The `transform` function takes a partition of type `T` as input and returns a partition of type `T`. This design ensures flexibility in handling different partition types while maintaining type safety and facilitating reusable and composable partition transformations.

While users are welcome to define their own partition transformations, the system provides several built-in transformations, including rewriting local and remote communications and synthesizing message data structures based on partitions. The overall optimization pipeline is shown in Figure 7.

In the first phase in Figure 7, the system refines the communication for members in a partition structure to identify what communication can be compiled away, namely fixed communication repeated in every round. OptiFusion provides an annotation to let users specify which references in `receiveFrom` are fixed. The output of this phase is a partition whose members are pairs, where the first element is the original member of type `BSP with ComputeMethod`, except that its `receiveFrom` is updated to not contain references that are static. The static references are captured in the second

[13]Note that we show the full definition of `Partition` (lines 4 – 11) for clarification. In practice, we can also define an implicit function that automatically transforms partitions returned by `partition` into `Partition` objects.

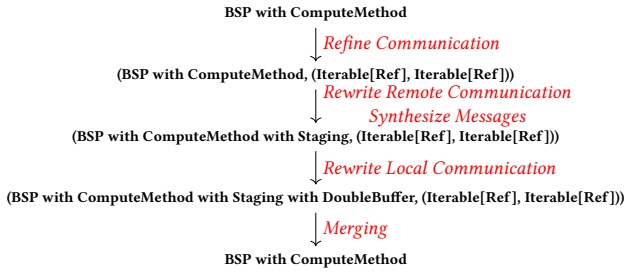


Figure 7: Partition transformations in the default optimizer.

element, which is again a pair $(localStatic, remoteStatic)$, whose names are self-explanatory.

In the next phase, the system synthesizes cross-partition message structures based on the cross-edges specified in the topology of the partition structure. Additionally, the system specializes for references in the *remoteStatic* and rewrites instructions for remote communication into instructions that operate over the synthesized message structures. The computations for partially aggregating the instructions are stored as `StagedExpr` defined in the trait `Staging`, which is mixed in as the output type.

Similarly, the system specializes for *localStatic* by storing computations as staged expressions that are evaluated later. Before doing this, however, the system needs to additionally mix-in the `DoubleBuffer` trait, which keeps an additional copy of the state value, to ensure that concurrent local accesses should read the correct version of the value, as explained in Section 2.

Finally, the system can merge a partition, consolidating its members into a single element of a collection type like `Vector`. This can reduce the degree of parallelism and improve data locality, when mapping members of a partition into parallel agents in the distributed backend.

4.3 Backend

Integrating with CloudCity is achieved easily through library functions. As seen previously in Figure 6, OptiFusion provides connector functions like `bspToAgent` that convert a specialized partition into an agent object in CloudCity. Figure 8 shows the definition of `bspToAgent`. The attributes `id` (line 1) and `connectedAgentIds` (line 2) are built-in attributes for a CloudCity agent. The `run` method in CloudCity agents [61] is a co-routine that yields the control to the system at the end of each superstep. The functions `toInMessage` (line 7) and `toOutMessage` (line 9) are library functions that transform values in OptiFusion into messages in CloudCity.

5 EXPERIMENTS

OptiFusion is a compile-time program specialization framework that exploits a wide range of data-sharing and computation-sharing optimizations. Our goal is to bridge the performance gap between generic programs, which are easy to develop, and specialized programs, which are efficient to execute through compile-time optimizations. We evaluate the effectiveness of OptiFusion's optimizations using a benchmark for complex stateful workloads described in [61]. More specifically, we compare the best performance of

```
1  def bspToAgent(bsp: BSP with ComputeMethod): Actor =
2    new Actor {
3      id = bsp.id
4      connectedAgentIds = bsp.receiveFrom
5
6      override def run(): Unit = {
7        bsp.run(receivedMessages.map(i => bsp.
       deserialize(toInMessage[bsp.OutMessage](i))))
8        receivedMessages.clear()
9        val outMessage = toOutMessage(bsp.
       stateToMessage(bsp.vaue))
10       connectedAgentIds.foreach(n => {
11         sendMessage(n, outMessage)
12       })}}
```

Figure 8: BSP instances in OptiFusion are transformed into agents in CloudCity via functions like `bspToAgent`.

OptiFusion with all optimizations enabled, with four reference implementations, and show that OptiFusion achieves:

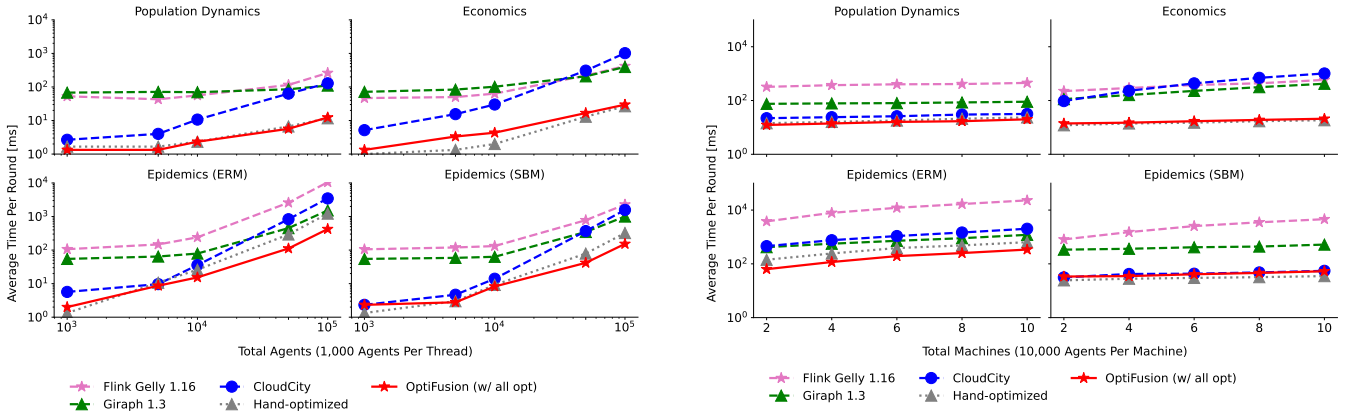
- on par or better performance than the hand-optimized implementations with hard-coded data-sharing and computation-sharing optimizations, and
- over 10× faster than the vertex-centric systems without data or computation sharing optimizations, like Giraph, Flink Gelly, and CloudCity.

Additionally, we provide a detailed breakdown of how each optimization in OptiFusion contributes to overall performance and report the time spent on the optimizer.

5.1 Configuration

Our experiments use 10 servers, each configured with an Intel Xeon Silver 4214 processor, which features 24 cores, 48 hardware threads, and a clock frequency of 2.2GHz. Every server is equipped with two 10Gbps Ethernet network interface cards (NIC), configured to use Link Aggregation Control Protocol (LACP). The servers are distributed across multiple racks within the same network. Switches across racks are connected to each other via switches using two 100Gbps Ethernet links. The operating system is Debian 12. Software dependencies include OpenJDK 11, Scala 2.12.18, and CloudCity 2.0-SNAPSHOT.

We benchmark different implementations using the agent-based simulation benchmark introduced in [61]. This benchmark contains a diverse set of workloads: population dynamics, economics, and epidemiology. Details of the workloads can be found in [61]. The population dynamics simulation is the game of life, where each agent is connected to eight adjacent agents. The economics simulation models an evolutionary stock market, where the stock price is seen as the emerging property of traders' buy and sell actions. There is one market agent and the rest are trader agents. The market and traders are connected. Traders do not communicate among themselves. For the epidemics example, two random graph models are considered: the Erdős-Rényi model (ERM), where each edge is included in the generated graph with probability $p = 0.01$, and the stochastic block model (SBM), where vertices are separated into 5 balanced blocks. Two vertices in the same block are connected with probability $p = 0.01$. Vertices in different blocks are not connected.



(a) Increase threads (1,000 agents per thread, 1 machine).

(b) Increase machines (10,000 agents per machine).

Figure 9: Cross-comparing OptiFusion with other systems and Hand-optimized when increasing: (a) threads, and (b) machines.

5.2 Reference Approaches

Recall that in Figure 1 in Section 1, we presented the experiment results for repeating the scale-up experiments in [61] with the latest software when applicable. We reproduced the result in [61] and demonstrated that the performance gap between stateless BSP systems and stateful BSP systems remains when executing agent-based simulations. Here we only compare against stateful BSP systems, namely Giraph [58], Flink Gelly [20], and CloudCity [61]. Additionally, we hand-optimize each workload for comparison.

The system separates the input graph into K balanced components.[14] The partitioning strategy affects the number of cross-component messages and is performance-crucial. We evaluate the following strategies:

- Random partitioning: Agents are randomly partitioned into components of a target size.
- Hash partitioning: An agent is assigned to a component according to a hash function. We consider the following two self-explanatory hash functions:

$$\text{hash}(id) = (id \,/\, partitionSize).\text{toInt}$$
$$\text{hash}(id) = (id \,\%\, partitionSize).\text{toInt}.$$

- Greedy partitioning: Each component is initialized with a randomly selected agent. We then iteratively add unplaced neighbors of the agents to the component using a breadth-first strategy, continuing until either all neighboring agents are placed or the component reaches its target size.

We evaluate each partitioning strategy for the benchmark workloads. Our experiment results[15] showed that the greedy strategy resulted in the best overall performance across all workloads. Since our main goal is to investigate the effectiveness of program transformations that exploit optimizations in OptiFusion, not how different partitioning strategies affect the performance, we set greedy partition as the default partitioning strategy whenever applicable.

5.3 Performance Evaluation

For the experiments described next, each experiment is repeated three times, and we report the mean of the average time per round for each workload in the benchmark. Each workload is tuned in the same way as documented in [61]. We run the population dynamics and the economics example for 200 rounds, and the epidemics examples for 50 rounds.[16]

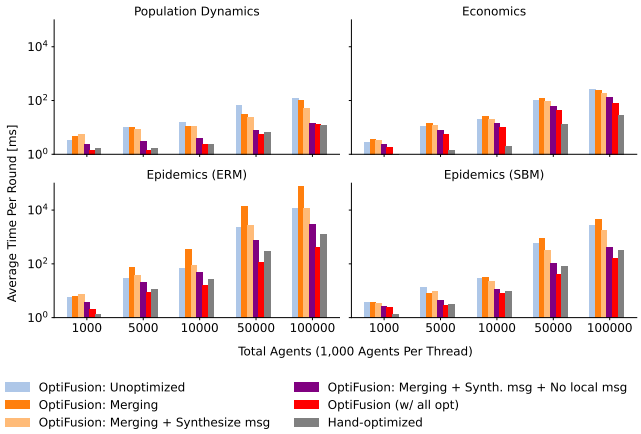
To compare the scalability of OptiFusion with other systems, we fix 1,000 agents per thread and increase the number of threads from 1 to 100 and the number of machines from 1 to 10. Agents are partitioned using the greedy strategy into balanced components that each contain 1,000 agents. Every graph component has a thread.

Figure 9 shows how the average time per round for each workload changes for each system as we increase the number of agents, both by increasing the number of threads and the number of machines. The y-axis is the average time per round measured in milliseconds in the log scale. OptiFusion achieves comparable or better performance than hand-optimized implementations, both when increasing the number of agents on one machine and when increasing the number of machines. OptiFusion can be 10× faster than CloudCity, Giraph and Flink Gelly. The impact of system design on the benchmark performance of baseline systems has been analyzed thoroughly in [61] and omitted here.

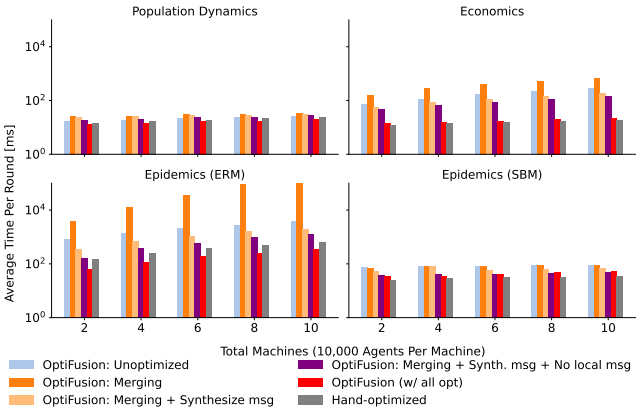
Figure 9a shows the scale-up experiments when increasing the number of threads from 1 to 100. Per thread, the number of agents is fixed to 1,000. The x-axis denotes the number of agents in the log scale. As the number of agents increases, the speedup of OptiFusion and the hand-optimized implementations over other baseline systems increases from 2-4× to 8-15× across all workloads. A detailed breakdown of how different optimizations affect the performance is shown in Figure 10a and Figure 10b, examined in Section 5.4.

Figure 9b shows the scale-out experiments when increasing the number of machines up to ten. We fix 10,000 agents per machine and shows the number of machines in the linear scale on x-axis.

[14] The value of K is determined by the number of threads and the number of machines.
[15] Omitted here due to space limitations.
[16] As explained in [61], the difference in the number of rounds is to capture "interesting" computations in the presence of different communication patterns.



(a) Optimization breakdown when increasing threads.



(b) Optimization breakdown when increasing machines.

Figure 10: Analyzing different optimizations in OptiFusion when increasing: (a) threads and (b) machines.

Compared with other workloads, the economics experiments witness the highest speedup: hand-optimized and OptiFusion are over 50× faster than other systems. This is due to aggregation pushdown, where an aggregator agent processes values from traders locally before sending the result to the market agent, which drastically reduces the number of messages and balances computations. For population dynamics, the average time per round is nearly the same as the number of machines increases, since computations and remote messages per machine remains unchanged for all systems.

In some workloads like ERM, we see that OptiFusion can outperform even hand-optimized implementations by up to 2×. This is due to code specialization. While OptiFusion generates instructions specialized for each agent, the hand-optimized implementations resemble a generic, interpreted approach. The difference between them is highlighted in the pseudocode below:

```
1  // hand-optimized
2  adjList: Map[Id, List[Id]]
3  values: Map[Id, Value] // overhead when looking up
4  adjList(0).map(i => values(i))
5
6  // generated from OptiFusion
7  values: Array[Value]
8  List(2, 3, 9).map(i => values(i))
```

In hand-optimized implementations, an agent looks up neighbor ids from `adjList` and obtains their values from `values` (line 4). OptiFusion generates instructions shown in lines 7–8. The neighbor ids of the agent 0 are inlined and transformed to the local offset of the array `values`. Writing code like line 8 for each agent in the graph by hand is impractical for a large number of agents.

The interpretation overhead in hand-optimized implementations increases as the number of neighbors for an agent increases. This is evident in Figure 9a. As the number of agents increases from 1,000 to 100,000, the number of neighbors (on average) for an agent in ERM increases from 10 to 1,000, since the edge probability of the ERM model is set to 0.01, and the speedup of OptiFusion over hand-optimized also increases to 2×. Similarly for the SBM experiments. For the population dynamics workload, each agent has exactly eight neighbors, regardless of the total number of agents. As a result, OptiFusion and hand-optimized have similar performance even as the number of agents increases. For the economics example, OptiFusion and hand-optimized both implement the aggregation pushdown optimization, as we have explained before.

5.4 Optimization Analysis

In the unoptimized OptiFusion, a behavioral equation is transformed directly into a CloudCity agent. Below we analyze the performance impact of different optimizations in OptiFusion; each transformation is applied after the previous one.

Merging. This transformation merges a collection of agents in the unoptimized OptiFusion into one agent. After merging, agents within a graph component can obtain the scope information dynamically, such as which neighbors are local (in the same graph component). This sets the stage for later optimizations, which exploit the locality information to specialize agent computations.

Merging unoptimized agents can worsen the performance of unoptimized agents, as seen in Figure 10. There are two main reasons. First, to deliver messages to a neighbor that is fused in a remote component, an agent needs to prepend the component id to a cross-component message, in addition to specifying the id of the receiver agent. This causes more data to be serialized compared with the unoptimized implementation. Second, merging can worsen the computation imbalance among graph components, causing skewed tail latency that slows down the overall average time per round. Luckily, we can mitigate the overhead of merging with optimizations that are enabled, as we will explain shortly.

Synthesize remote message data structures. After merging, we evaluate the performance impact of synthesizing data structures for messages between graph components. This optimization addresses the messaging inefficiency of merging, where an agent needs to prepend a component identifier in addition to the agent id to every message. In Figure 10, we see that this optimization mitigates the overhead of merging and achieves similar or slightly better performance than unoptimized across all workloads, due to more efficient message data structures. The performance improvement is

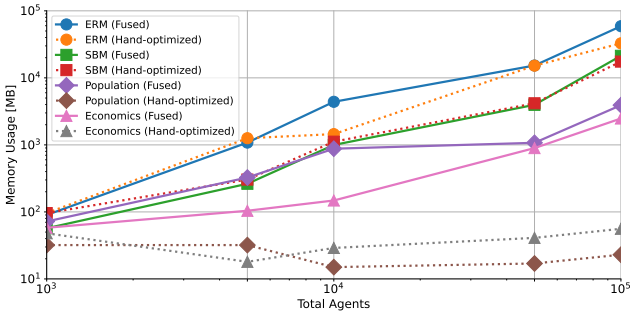


Figure 11: Memory consumption of the partition structure.

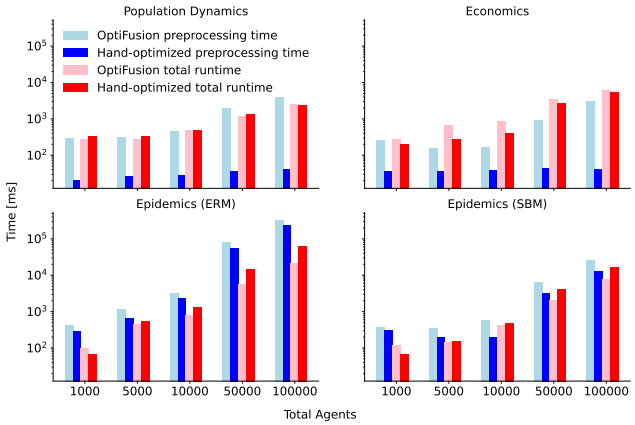


Figure 12: Comparing end-to-end performance: preprocessing time (fixed, one-time cost) and total runtime (increases when increasing the number of rounds).

more evident in the scale-out experiments in Figure 10b than in the scale-up experiments in Figure 10a, since network messages are serialized. For ERM experiments on 10 machines, this optimization improves the performance of merging by over 100×.

Compile away local messages. We further consider compiling local messages away, assuming that an agent sends the same value to all neighbors. Instead of materializing and sending these local messages, we let a sender agent generate one message that stores the value of its state. Other agents in the same graph component look up this message locally. Note that we have separated transforming communication instructions into local computation instructions as another optimization. Figure 10 shows that this technique results in up to 2× improvement over the previous optimization.

Rewrite communication instructions into local computations. This optimization inlines the value of neighbor ids into specialized instructions generated for each agent after compiling away local messages, improving the performance by up to 5×, finally allowing OptiFusion to achieve on par or even better performance than the hand-optimized implementations.

Aggregation Pushdown. The final optimization that we evaluate is aggregation pushdown. As explained before, this optimization is only applied to the economics workload. We introduce an aggregator agent in each graph component to combine the action of traders locally before sending the aggregated value to the market agent. This drastically reduces network messages and results in over 10× speedup for the economics example.

5.5 Optimization Overhead

We have shown that OptiFusion can match and even surpass the performance of hand-optimized implementations. Here we examine the resource consumption of our approach and end-to-end time.

5.5.1 Memory Footprint. Our optimizations transform a generic program into a specialized program through several phases of rewriting. This can consume more memory than the hand-optimized implementation, due to the auxiliary partition structure as well as the intermediate data structures used in the optimization phase. The memory overhead can be addressed via metaprogramming that modifies the underlying abstract syntax tree of a program, instead of generating a new closure with the transformed program.

We measure the memory consumption of the optimizer in OptiFusion and compare it with that of the hand-optimized implementation, as the number of agents increases from 1,000 to 100,000 on one machine, shown in Figure 11. The x-axis shows the number of agents in the log scale and the y-axis shows the memory consumption measured in MB in the log scale.

Figure 11 illustrates that OptiFusion consumes more memory than the hand-optimized implementation. Though each workload has the same number of agents, their memory consumption varies, closely related to the average number of neighbors per agent. In both OptiFusion and the hand-optimized implementation, the amount of memory consumed by the epidemics examples (ERM, SBM) are greater than other workloads.

Figure 11 also shows that the memory consumption can actually decrease slightly when increasing the number of agents for hand-optimized experiments: from 5,000 to 10,000 for population dynamics, and from 1,000 to 5,000 for economics. This is due to just-in-time (JIT) compilation. To see this, in the population dynamics experiments, the dimension of the grid is 50×100 and 100×100 for 5,000, and 10,000 agents respectively. Hence, the code that describes the behavior of a row of agents is repeated for 50 and 100 times, which can cause the JIT compiler in the Java Virtual Machine to optimize at different levels: the JIT compiler optimizes more aggressively when a code is repeated 100 times than 10 or 50 times. Similarly for the economics experiments. Such effects are more noticeable when the overall memory usage is low.

5.5.2 End-to-End Performance. We also measure the end-to-end time, comparing the preprocessing time and total runtime of OptiFusion with hand-optimized. The preprocessing time refers to the time spent before simulations start, including constructing the underlying social graph with agents. For OptiFusion, the preprocessing time also includes the optimization time.

In Figure 12, the preprocessing time of epidemics examples are longer than their runtime in both OptiFusion and hand-optimized. In other words, constructing a social network graph according to the random graph models ERM and SBM can be more costly than

running the simulation itself, since the epidemics simulations only run for 50 rounds. In these examples, the optimizer overhead is less than 10% of the overall preprocessing time; the preprocessing time of OptiFusion is only slightly longer than that of hand-optimized. For the population dynamics and the economics examples, however, where the social graphs are trivial to generate for hand-optimized implementations, although the percentage of the optimizer overhead remains approximately the same, the overall preprocessing time of hand-optimized is much less than OptiFusion.

We also measured the time breakdown for each optimization phase described in Section 4. In the first phase, the optimizer refines the `receiveFrom` values of each agent to identify which communication can be removed, which takes 5% of the optimizer time. After this, the optimizer synthesizes message data structures for cross-partition messages and rewrites remote communication instructions into local computations that interact with the synthesized message data structures, which accounts for 8%–10% of the time. Finally, the optimizer fuses agents in a partition and rewrites local communication into local computations.

The preprocessing time can be shortened by exploiting more efficient algorithms for constructing random graphs. Alternatively, one can also read an input data graph from a file, which is also supported by OptiFusion. The optimizer can also be improved by exploiting low-level parallel primitives rather than the default Scala parallel primitives like *par* and *seq*, as shown in Figure 8. Additionally, the current optimizer uses trait mixins to decompose and rewrite programs, which can generate unnecessary data structures. This issue can be improved using metaprogramming, which rewrites the abstract syntax tree of a program directly without generating the intermediate closure objects, but at the cost of adopting a less user-familiar metaprogramming interface.

6 RELATED WORK

The design features listed in the desiderata at the beginning of this paper in Section 1 have been explored partially in existing distributed systems. Dedalus [6] is a Datalog-like language that emphasizes the model-theoretic perspective of distributed programs, featuring the logical aspects of this desideratum, including state-based dynamics and simple interactions like asynchronous communication. In the Boom project [5], Overlog [18] is another Datalog-like language that allows users to specify computation and data placement via location specifiers, but not state-based dynamics or interactions. While these Datalog-like languages focus on *declarative* programming that specifies top-down transformations of a parallel collection, our language is grounded in the π-calculus, a widely used concurrency formalism that can model fine-grained *interactions* between items in a parallel collection bottom-up. Another key difference is that these languages are for asynchronous systems while our language targets BSP systems.

Existing optimizations for BSP systems can be coarsely classified as follows: varying the degree of synchrony and leveraging partitions [24, 27, 32, 72], optimizing for different hardware [31, 40, 52, 53, 71], specializing for target applications [26, 48, 60, 66], and proposing variants of the vertex-centric model [25, 47, 54, 59, 67–70, 73]. While it is common knowledge that data and computation sharing – an umbrella term that refers to various optimization techniques – can improve performance, there is no existing work that investigates a principled foundation for such techniques in BSP systems. For example, Spark [70] users share computation by materializing the intermediate result for part of a computation graph, while Giraph++ [59] users share computation by programming boundary vertices with explicit sharing behavior.

Here we propose a novel approach of using a process calculus to model the concurrent behavior of BSP programs for system optimizations. Process calculi, such as the π-calculus, are formal frameworks designed to model and analyze the intricate interactions in parallel systems through a rigorous mathematical language. These calculi have been widely utilized across various domains, including programming language design [8, 19, 44, 46, 62], concurrent constraint programming [39, 49, 50, 55, 64], network mobility and security [2, 28, 51]. Despite their extensive application in these fields and others [4, 7, 9, 13, 14, 22, 43, 65], process calculi have yet to be exploited for system optimization purposes.

The π-calculus has inspired numerous language variants. The *Spi Calculus* [2] and *Applied π-Calculus* [1] extend its capabilities to model cryptographic protocols and verify security properties, while the *Join Calculus* [23] introduces join patterns for distributed programming. The *Fusion Calculus* [42] generalizes π-calculus with symmetric communication, and the *Ambient Calculus* [15] focuses on mobile computation through bounded spaces. *Psi-Calculi* [9] provides a highly customizable framework for diverse applications, and the *Session π-Calculus* [29] formalizes structured communication via sessions. Behavioral equations are designed to model and transform distributed computations for system optimizations. To the best of our knowledge, our work is the first to demonstrate that process calculi are a valuable and practical optimization tool for system designers. We showed that behavioral equations can be used as a uniform framework to express a wide range of practical data and computation sharing techniques.

7 CONCLUSIONS AND FUTURE WORK

In this work, we have introduced a new language called *behavioral equations* based on the π-calculus and showed how various data-sharing and computation-sharing optimizations can be expressed as transformations of behavioral equations. We have also built the OptiFusion system based on behavioral equations and demonstrated the effectiveness of the optimizations empirically.

The optimizations we have exploited only begin to tap into the potential of behavioral equations and process calculus. For example, the full flexibility of using π-calculus, in particular the replication operator, allows the system to distinguish values that are available exactly once from those that are always available. Though not leveraged by the optimizations here, such features are important for expressing privacy and security constraints.

We believe that using process calculus, which models fine-grained interactions between processes, provides a clean, generic, and flexible foundation for parallel programming. Our formalism paves the way for future work like analyzing the theoretical properties of stateful systems, such as the correctness condition, the equivalence relation, the complexity of states, and the relationship with other stateless or asynchronous distributed systems.

ACKNOWLEDGMENTS

This work was supported by a Postdoc Grant at the University of Zurich, project number FK-24-020.

REFERENCES

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. 2018. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *J. ACM* 65, 1 (2018), 1:1–1:41. https://doi.org/10.1145/3127586

[2] Martín Abadi and Andrew D. Gordon. 1997. A calculus for cryptographic protocols: the spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security* (Zurich, Switzerland) *(CCS '97)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/266420.266432

[3] David Adam. 2020. Special report: The simulations driving the world's response to COVID-19. https://www.nature.com/articles/d41586-020-01003-6

[4] Michael Alexander and William Gardner. 2008. *Process algebra for parallel and distributed processing*. CRC Press, U.S.A.

[5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, France, 223–236. https://doi.org/10.1145/1755913.1755937

[6] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.), Vol. 6702. Springer, UK, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

[7] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press, USA.

[8] Fred Barnes and Peter Welch. 2024. occam-pi: blending the best of CSP and the pi-calculus. https://www.cs.kent.ac.uk/projects/ofa/kroc/ Accessed: 2024-08-29.

[9] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. 2009. Psi-calculi: Mobile Processes, Nominal Data, and Logic. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science (LICS '09)*. IEEE Computer Society, USA, 39–48. https://doi.org/10.1109/LICS.2009.20

[10] Martin Beran. 1999. Decomposable Bulk Synchronous Parallel Computers. In *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings (Lecture Notes in Computer Science)*, Jan Pavelka, Gerard Tel, and Miroslav Bartosek (Eds.), Vol. 1725. Springer, Milovy, Czech Republic, 349–359. https://doi.org/10.1007/3-540-47849-3_22

[11] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte, and Ingo Rieping. 1999. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*. IEEE Computer Society, San Juan, Puerto Rico, 99–104. https://doi.org/10.1109/IPPS.1999.760442

[12] Mark Buchanan. 2009. Economics: Meltdown modelling. *Nature* 460, 7256 (2009), 680–682.

[13] Peter R. Calvert. 2015. *Architecture-neutral parallelism via the Join Calculus*. Technical Report UCAM-CL-TR-871. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-871

[14] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5 (2020), 28:1–28:94. https://doi.org/10.1145/3402457

[15] Luca Cardelli and Andrew D. Gordon. 2000. Mobile ambients. *Theor. Comput. Sci.* 240, 1 (2000), 177–213. https://doi.org/10.1016/S0304-3975(99)00231-5

[16] Hojung Cha and Dongho Lee. 2001. H-BSP: A Hierarchical BSP Computation Model. *J. Supercomput.* 18, 2 (2001), 179–200. https://doi.org/10.1023/A:1008113017444

[17] 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

[18] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. 2008. Evita raced: metacompilation for declarative networks. *Proc. VLDB Endow.* 1, 1 (2008), 1153–1165. https://doi.org/10.14778/1453856.1453978

[19] Robert De Simone. 1985. Higher-level synchronising devices in Meije-SCCS. *Theoretical computer science* 37 (1985), 245–267.

[20] 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.

[21] J. Doyne Farmer and Duncan Foley. 2009. The economy needs agent-based modelling. *Nature* 460, 7256 (2009), 685–686. https://doi.org/10.1038/460685a

[22] Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-Calculus. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, USA, 372–385. https://doi.org/10.1145/237721.237805

[23] Cédric Fournet and Georges Gonthier. 2000. The Join Calculus: A Language for Distributed Mobile Programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (Lecture Notes in Computer Science)*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.), Vol. 2395. Springer, 268–332. https://doi.org/10.1007/3-540-45699-6_6

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, USA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[25] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[26] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/2588555.2610511

[27] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.

[28] Matthew Hennessy. 2007. *A distributed Pi-calculus*. Cambridge University Press, UK.

[29] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science)*, Eike Best (Ed.), Vol. 715. Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[30] Imperial College COVID-19 Response Team. 2020. *Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand*. Technical Report. Imperial College.

[31] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, Canada, 239–252. https://doi.org/10.1145/2600212.2600227

[32] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727. https://doi.org/10.14778/2212351.2212354

[33] 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, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[34] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall, UK.

[35] Robin Milner. 1990. Functions as Processes. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings (Lecture Notes in Computer Science)*, Mike Paterson (Ed.), Vol. 443. Springer, England, UK, 167–180. https://doi.org/10.1007/BFB0032030

[36] Robin Milner. 1993. *The polyadic π-calculus: a tutorial*. Springer, UK.

[37] Robin Milner. 1999. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, UK.

[38] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

[39] Rubén Monjaraz and Julio Mariño. 2012. From the π-calculus to flat GHC. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming* (Leuven, Belgium) *(PPDP '12)*. Association for Computing Machinery, New York, NY, USA, 163–172. https://doi.org/10.1145/2370776.2370797

[40] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014, Boston, MA, USA, May 11-13, 2014*. IEEE Computer Society, USA, 25–28.

https://doi.org/10.1109/FCCM.2014.15

[41] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 41–57. https://doi.org/10.1145/1094811.1094815

[42] Joachim Parrow and Björn Victor. 1998. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*. IEEE Computer Society, 176–185. https://doi.org/10.1109/LICS.1998.705654

[43] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. https://doi.org/10.1145/3280984

[44] Benjamin C. Pierce and David N. Turner. 2000. Pict: a programming language based on the Pi-Calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, USA, 455–494.

[45] Pilar de la Torre and Clyde P. Kruskal. 1996. Submachine Locality in the Bulk Synchronous Setting (Extended Abstract). In *Euro-Par '96 Parallel Processing (Lecture Notes in Computer Science)*, Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert (Eds.), Vol. 1124. Springer, Lyon, France, 352–358. https://doi.org/10.1007/BFb0024723

[46] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

[47] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[48] Semih Salihoglu and Jennifer Widom. 2014. Optimizing Graph Algorithms on Pregel-like Systems. *Proc. VLDB Endow.* 7, 7 (2014), 577–588. https://doi.org/10.14778/2732286.2732294

[49] Vijay A Saraswat and Patrick Lincoln. 1992. *Higher-order linear concurrent constraint programming*. Technical Report. Technical report, Xerox PARC.

[50] Vijay A. Saraswat and Martin Rinard. 1989. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 232–245. https://doi.org/10.1145/96709.96733

[51] Alan Schmitt and Jean-Bernard Stefani. 2003. The m-calculus: a higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/604131.604136

[52] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, China, 135–146. https://doi.org/10.1145/2442516.2442530

[53] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer (Eds.). IEEE, USA, 403–412. https://doi.org/10.1109/DCC.2015.8

[54] Yogesh Simmhan, Alok Gautam Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings (Lecture Notes in Computer Science)*, Fernando M. A. Silva, Inês de Castro Dutra, and Vítor Santos Costa (Eds.), Vol. 8632. Springer, Portugal, 451–462. https://doi.org/10.1007/978-3-319-09873-9_38

[55] Gert Smolka. 1994. A Foundation for Higher-order Concurrent Constraint Programming. In *Constraints in Computational Logics, First International Conference, CCL'94, Munich, Germany, September 7-9, 1994 (Lecture Notes in Computer Science)*, Jean-Pierre Jouannaud (Ed.), Vol. 845. Springer, Germany, 50–72. https://doi.org/10.1007/BFB0016844

[56] The Apache Software Foundation. 2024. Apache Flink. https://flink.apache.org/. Accessed: 2024-10-05.

[57] The Apache Software Foundation. 2024. Apache Flink Gelly. https://mvnrepository.com/artifact/org.apache.flink/flink-gelly. Accessed: 2024-10-05.

[58] The Apache Software Foundation. 2024. Apache Giraph. https://attic.apache.org/projects/giraph.html. Accessed: 2024-10-05.

[59] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (2013), 193–204. https://doi.org/10.14778/2732232.2732238

[60] 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 2023, Cascais, Portugal, October 22-23, 2023*, Coen De Roover, Bernhard Rumpe, and Amir Shaikhha (Eds.). ACM, Portugal, 100–112. https://doi.org/10.1145/3624007.3624057

[61] Zilu Tian, Peter Lindner, Markus Nissl, Christoph Koch, and Val Tannen. 2023. Generalizing Bulk-Synchronous Parallel Processing for Data Science: From Data to Threads and Agent-Based Simulations. *Proc. ACM Manag. Data* 1, 2, Article 151 (June 2023), 28 pages. https://doi.org/10.1145/3589296

[62] Daniele Turi and Gordon D. Plotkin. 1997. Towards a Mathematical Operational Semantics. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*. IEEE Computer Society, Poland, 280–291. https://doi.org/10.1109/LICS.1997.614955

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

[64] Björn Victor and Joachim Parrow. 1996. Constraints as Processes. In *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings (Lecture Notes in Computer Science)*, Ugo Montanari and Vladimiro Sassone (Eds.), Vol. 1119. Springer, Italy, 389–405. https://doi.org/10.1007/3-540-61604-7_66

[65] Lucian Wischik and Philippa Gardner. 2005. Explicit fusions. *Theor. Comput. Sci.* 340, 3 (2005), 606–630. https://doi.org/10.1016/J.TCS.2005.03.017

[66] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3447786.3456247

[67] Wenlei Xie, Guozhang Wang, David Bindel, Alan J. Demers, and Johannes Gehrke. 2013. Fast Iterative Graph Computation with Block Updates. *Proc. VLDB Endow.* 6, 14 (2013), 2014–2025. https://doi.org/10.14778/2556549.2556581

[68] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: a block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.* 7, 14 (oct 2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[69] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast Iterative Graph Computation: A Path Centric Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, Trish Damkroger and Jack J. Dongarra (Eds.). IEEE Computer Society, USA, 401–412. https://doi.org/10.1109/SC.2014.38

[70] 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

[71] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Albert Cohen and David Grove (Eds.). ACM, USA, 183–193. https://doi.org/10.1145/2688500.2688507

[72] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. 2011. PrIter: a distributed framework for prioritized iterative computations. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, Jeffrey S. Chase and Amr El Abbadi (Eds.). ACM, Portugal, 13. https://doi.org/10.1145/2038916.2038929

[73] Xianke Zhou, Pengfei Chang, and Gang Chen. 2014. An Efficient Graph Processing System. In *Web Technologies and Applications - 16th Asia-Pacific Web Conference, APWeb 2014, Changsha, China, September 5-7, 2014. Proceedings (Lecture Notes in Computer Science)*, Lei Chen, Yan Jia, Timos K. Sellis, and Guanfeng Liu (Eds.), Vol. 8709. Springer, China, 401–412. https://doi.org/10.1007/978-3-319-11116-2_35