Preliminary Study of Interaction Nets Applicability for Sparse Linear Algebra Based Graph Analysis

Danil Zaripov[1,*], Efim Kubyshkin[1], Nikolai Ponomarev[1], Georgy Sichkar[1], Vladimir Zaikin[1] and Semyon Grigorev[1,*]

[1]Saint Petersburg State University, 7-9 Universitetskaya Embankment, St. Petersburg, 199034, Russia

Abstract

Developing scalable parallel algorithms for graph analysis remains challenging due to the irregular nature of graphs, which makes load balancing non-trivial. Interaction Nets naturally support irregular parallelism at a low level, making them a promising foundation for graph analysis. Using two Interaction Net implementations, Vine and Inpla, we provided an equivalent set of primitives and functions over quad-tree-represented matrices and vectors, and conducted a preliminary performance evaluation. From this, we derived several observations regarding Vine and Inpla, along with directions for future research.

Keywords

Interaction Nets, graph rewriting, graph analysis algorithms, GraphBLAS, sparse linear algebra, quad-tree

1. Introduction

Graph analysis has become fundamental to modern data science, powering applications ranging from recommendation systems in social networks and fraud detection in financial transactions to protein interaction analysis in bioinformatics and static code analysis. As graph sizes grow to billions of vertices and edges, the need for efficient, scalable implementations becomes critical. However, graph analysis algorithms are difficult to parallelize due to their irregular memory access patterns, poor data locality, and load imbalance across computational units.

The GraphBLAS standard [1] addresses these challenges by providing a foundation for high-performance parallel graph analysis, expressing algorithms in linear algebra using generic sparse vectors/matrices and parameterizable operations. However, implementing high-performance generic operations over sparse data remains challenging: irregular access and imbalanced subtasks complicate load balancing and parallelization, while nontrivial chains of operations create intermediate data structures. Kernel fusion could help, but automatic optimization for sparse generic computations is difficult, so it is performed manually in the most common cases.

At the same time, Interaction Nets [2]—a graph-rewriting model introduced by Yves Lafont—offer a computational model that is inherently well-suited for irregular parallelism. In Interaction Nets, computation proceeds through local rewriting rules applied to pairs of connected nodes (so-called active pairs). Crucially, multiple non-interfering active pairs can be reduced simultaneously, making Interaction Nets a promising foundation for parallel irregular computations, including graph analysis. Furthermore, Interaction Nets are relatively close to functional programming languages, making it easy to transfer design approaches and ideas from the functional world.

However, comparisons between existing interaction-net-based evaluators remain very limited and are often performed only on basic benchmarks such as Fibonacci, Ackermann, sorting, and list traversal.

We believe that Interaction Nets can provide a powerful foundation for implementing graph analysis algorithms. The long-term goal of our work is to investigate how Interaction Nets can help solve

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

*Corresponding author.

st129060@student.spbu.ru (D. Zaripov); e.kubyshkin@spbu.ru (E. Kubyshkin); n.ponomarev@spbu.ru (N. Ponomarev); georgy.sichkar@yandex.ru (G. Sichkar); v.a.zaikin@spbu.ru (V. Zaikin); s.v.grigoriev@mail.spbu.ru (S. Grigorev)

0009-0003-1086-9279 (D. Zaripov); 0009-0009-5531-1758 (E. Kubyshkin); 0009-0000-2382-5687 (N. Ponomarev); 0009-0005-4684-9610 (G. Sichkar); 0009-0003-3417-990X (V. Zaikin); 0000-0002-7966-0698 (S. Grigorev)

© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/GCM-paper7.pdf

challenges with irregular parallelism and, possibly, with kernel fusion. In this work, we do not focus on the GraphBLAS API implementation, but we are inspired by it: we implemented a minimal set of generic data structures and operations sufficient to implement several classic graph analysis algorithms.

This work is in progress, and our current contributions are as follows.

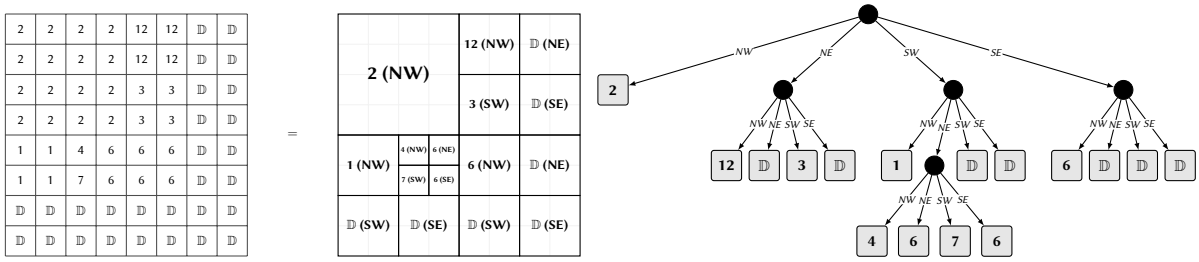
1. We implemented a minimal, functionally equivalent subset of the functions over tree-represented matrices and vectors in F# (a mature functional-first language, chosen for ease of prototyping and testing), Vine, and Inpla. On this foundation, we implemented the following graph analysis algorithms: BFS, SSSP, and triangle counting. Thus, we enable direct comparison of the respective execution models and runtime systems.
2. We evaluated the implemented algorithms on a set of real-world and synthetic graphs and highlight directions for future research.

2. Background

In this section, we introduce tree-based matrix representation, provide the basics of Interaction Nets, and motivate our choice of Interaction Nets implementations used in our evaluation.

2.1. Tree-based Representation for Sparse Matrices and Vectors

To store sparse matrices and vectors efficiently, we employ a quad-tree data structure. The matrix is recursively subdivided into four quadrants—northwest (NW), northeast (NE), southwest (SW), and southeast (SE)—until each block becomes either uniform or empty. Figure 1 shows an example of a matrix in the quad-tree form.



(a) Block decomposition of the matrix (b) Quad-tree representation of the blocks in 1a

Figure 1: Example of the matrix and its quad-tree representation

This recursive partitioning allows large uniform blocks to be represented implicitly, thereby reducing the memory required to store the matrix. Unlike flat array formats such as CSR or CSC, which are standard in libraries like SuiteSparse:GraphBLAS, the quad-tree naturally supports a divide-and-conquer execution model and adapts dynamically to irregular sparsity patterns. This structure simplifies parallelization of element-wise, matrix-matrix, and matrix-vector operations, as independent sub-blocks can be processed recursively without global synchronization.

2.2. Interaction Nets

Interaction Nets is a graph-rewriting computation model proposed by Y. Lafont in 1989 [2] and can be defined as follows.

Definition 1 (Agent). *An agent is a symbol from a fixed alphabet Σ, for which:*

- *an arity is defined—a function $Ar : \Sigma \to \mathbb{N} \cup \{0\}$;*
- *there is one principal port and a number of auxiliary ports equal to its arity.*

Definition 2 (Net). *A net over the alphabet Σ is a labeled undirected multi-graph in which:*

- *each node is labeled by an agent;*
- *edges can only exist between ports;*
- *at most one edge enters each port.*

Let $\mathcal{N}_\Sigma$ denote the set of all possible nets over the alphabet Σ.

We say that port p_a is connected to port p_b if there is an edge between them. We say that a port is *free* if no edge originates from it. A net can also consist simply of a set of edges, in which case each end of an edge is considered a free port.

Definition 3 (Interface). *The interface $I(n)$ is the set of all free ports of a net n.*

Computation in Interaction Nets is defined in terms of *reductions*: rewriting an *active pair* into a net as defined in *reduction rules*. An active pair consists of two agents connected by an edge between their principal ports.

Definition 4 (Reduction Rule). *A reduction rule is a graph rewriting rule. Let R be the set of reduction rules. More formally, a reduction rule is a triplet $r \in \Sigma \times \Sigma \times \mathcal{N}_\Sigma$, for which the following properties hold:*

- ***Symmetry:*** $(l_1, l_2, n) \in R \Rightarrow (l_2, l_1, n) \in R$
- ***Determinism:*** $(l_1, l_2, n_1) \in R \wedge (m_1, m_2, n_2) \in R \Rightarrow (l_1, l_2) \neq (m_1, m_2)$
- ***Interface Preservation:*** $\forall (l_1, l_2, n) \in R, Ar(l_1) + Ar(l_2) = \|I(n)\|$

An example of the computation of the net encoding of the term $(1+0)+(1+0)$ is shown in Figure 2. The set of agents is $\Sigma = \{0, S, Add\}$ with arities $Ar(0) = 0$, $Ar(S) = 1$ and $Ar(Add) = 2$. Reduction rules shown in Figure 3. The left auxiliary ports of the active Add-agents are connected to the net representing 1, encoded in Peano numerals as $S(0)$, and the right auxiliary ports are connected to the third Add-agent. There is one free port—one of the auxiliary ports of the Add-agent. This constitutes the interface.

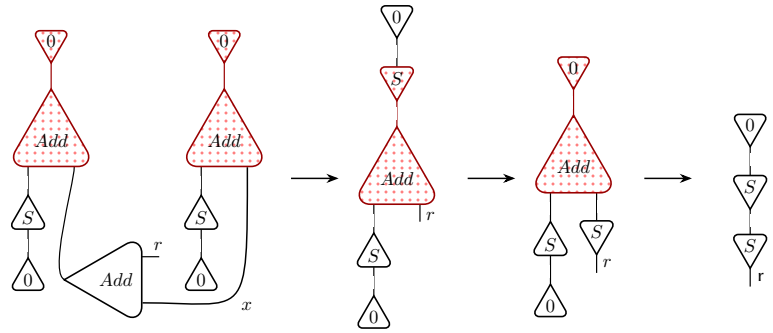


Figure 2: Example of computation in Interaction Nets: reduction of $(1+0)+(1+0)$ encoded in Peano numerals



(a) Rule for $0 + x = x$

(b) Rule for $S(x) + y = S(x + y)$

Figure 3: Reduction rules for Peano numerals

As shown in Lafont's work [2], λ-calculus can be expressed through this formalism, which makes Interaction Nets a general-purpose computation model. Moreover, agents and reduction rules are

not predefined, so one can define custom sets of agents and reduction rules, creating domain-specific (or problem-specific) models. Additionally, reductions can be performed independently, and it has been proven that their order does not matter. This is the primary reason for the natural suitability of Interaction Nets for parallel computations. These properties make Interaction Nets a promising platform for graph analysis tasks, as parallel graph analysis has an irregular nature. Furthermore, if a powerful language for describing graph analysis algorithms (e.g., linear algebra) can be introduced, then creating specific agents and rules may accelerate the final solution.

2.3. Implementations of Interaction Nets

Currently, we focus on Interaction Nets implementations that provide their own runtime. Consequently, we omit solutions that embed Interaction Nets into other languages (e.g., OCaml [3] or Haskell) or those that merely provide an interpreter. To the best of our knowledge, three such projects are actively developed: Inpla[1] [4], Vine[2], and HVM[3].

Although the HVM project is promising—offering a high-level language (Bend) that compiles to an interaction-net-based intermediate representation and an optional GPGPU-accelerated runtime—it is not yet stable enough for our purposes[4]. Therefore, we selected Inpla and Vine for evaluation.

Inpla provides a low-level Interaction Nets programming language with a parallel runtime. Programming in Inpla requires defining a custom set of rewriting rules for a chosen set of agents. The Inpla program for the interaction net from Figure 2 is shown in Listing 1. Computation runs on its own parallel virtual machine, implemented in C.

```
(* Defining custom rules *)
Add(r, x) >< Z => r ~ x; (* Rule presented in Figure 3a *)
Add(r, x) >< S(y) => Add(w, x) ~ y, r ~ S(w); (* Rule presented in Figure 3b *)
(* (1 + 0) + (1 + 0): two active pairs (Fig 2) *)
Add(x, S(Z)) ~ Z, Add(Add(r, x), S(Z)) ~ Z;
(* Printing the result and freeing the `r` name *)
r; free r;
```

Listing 1: Peano numerals addition in Inpla

Vine, in contrast to Inpla, is a high-level Rust-like language with a rich standard library (also implemented in Vine). It compiles to a low-level interaction-net-based language called IVY, which consists of a fixed set of agents and reduction rules — specifically, a slightly extended version of the standard basis for λ-calculus. The compiled program then runs on a parallel virtual machine implemented in Rust.

3. Implementation Details

In this section we describe our implementations: in F# (used as a prototype and reference for other implementations), in Inpla (a low-level language), and in Vine (a high-level language). We present the data structures and functions, list graph analysis algorithms (with short descriptions), and discuss details related to the implementation of the described structures and functions in selected languages.

3.1. Basic Data Structures and Functions

Matrix representation, as described in the F# code, is shown in Listing 2. Our representation is generic, meaning it is parameterized by the matrix cell type. The structure is not limited to matrices whose

[1]Inpla project: https://github.com/inpla/inpla.
[2]Vine project: https://github.com/VineLang/vine.
[3]HVM project: https://github.com/HigherOrderCO/HVM2.
[4]For example, open issues remain regarding failed tests (https://github.com/HigherOrderCO/HVM2/issues/437) and the failure to accelerate quad-tree-based computations (https://github.com/HigherOrderCO/Bend/issues/738).

```
type treeValue<'value> = | Dummy | UserValue of 'value

(* | x1 | x2 |
   ----------
   | x3 | x4 |  *)
type qtree<'value> =
    | Node of qtree<'value> * qtree<'value> * qtree<'value> * qtree<'value>
    | Leaf of treeValue<'value>

[<Struct>]
type Storage<'value> =
    // Storage is always size-x-size square.
    val size: uint64<storageSize>
    val data: qtree<'value>
    new(_size, _data) = { size = _size; data = _data }

[<Struct>]
type SparseMatrix<'value> =
    val nrows: uint64<nrows>
    val ncols: uint64<ncols>
    val nvals: uint64<nvals>
    val storage: Storage<Option<'value>>
```

Listing 2: Matrix representation

dimensions are powers of two. A user-defined matrix of arbitrary size is aligned to a square storage block whose size is the next power of two using `Dummy` blocks. Since the representation is compressed, `Storage` stores the aligned size (as a power of two) together with the corresponding tree. Finally, `SparseMatrix` stores the matrix dimensions and the number of non-zero entries (nnz) together with the data represented as `Storage`. Data structures for a sparse vector are described in a similar way.

We use `Option<'T>` to explicitly represent cells with no values. This approach is natural for graph representation: a cell of the adjacency matrix of a graph with edge labels of type `'T` has type `Option<'T>`—either there exists an edge with a label of the respective type (`Some(lbl)`), or there is no edge at all (`None`). Thus, for a sparse matrix with filled cells of type `'T`, the storage has type `Option<'T>`.

We implemented a minimal set of operations over matrices and vectors required to implement the selected graph analysis algorithms. The list of basic functions includes `map`, `fold`, and `reduce`. Element-wise operations are based on `map2`. In particular, `mask` is implemented as a special case of `map2`. Linear algebra functions include `vxm` (generic vector-matrix multiplication) and `mxm` (generic matrix-matrix multiplication). We also support transpose and extraction of the triangular part of a matrix (required for triangle counting). Conversion to and from coordinate list format is provided to simplify graph input construction. All operations take sparsity into account and preserve compression. We do not implement fused versions or manually parallelized versions of these functions.

Traditional exception handling is unlikely to be possible in Interaction Nets due to their highly parallel and irregular nature. Moreover, the selected languages (Inpla and Vine) do not provide any specific mechanisms for exception handling. We focus on realistic code, so we handle not only correct cases but also erroneous ones (e.g., inconsistent matrix or vector sizes). For this, we use the `Result` type (even in the F# version)[5] which explicitly represents two cases: `Ok<'R>` for successfully completed computations and `Err<'E>` for errors. This error handling pattern is supported by many languages, such as Rust and F#.

3.2. Implemented Graph Analysis Algorithms

We implemented three basic graph analysis algorithms.

[5]We do this to be as close to the interaction-net-based implementations as possible; F# has traditional exception mechanisms.

- Level-based BFS: for the given graph and start vertex, it returns a sparse vector `result` such that if `result[i] = Some(k)`, then vertex `i` was visited at step `k`. If `result[i] = None`, then `i` is unreachable.
- SSSP: for the given graph and start vertex, it returns a sparse vector `result` such that if `result[i] = Some(k)`, then the shortest path from the start vertex to `i` has length `k`. If `result[i] = None`, then `i` is unreachable. To prevent an infinite loop in the case of negative cycles, we check that the number of iterations should not be greater than the number of vertices in the input graph.
- Sandia triangle counting algorithm [5] that for the given undirected graph returns total number of triangles in it.

All algorithms were implemented using operations over matrices and vectors similarly to how they are described in GraphBLAS-related materials. We provide basic implementations without any optimizations such as the *push-pull* optimization for BFS or SSSP.

3.3. Interaction Nets Related Details

The translation from F# to Vine and Inpla is almost "as-is" without any interaction-nets-specific optimizations, code practices, or techniques.

In the case of Inpla, we translate algebraic type constructors from high-level languages into interaction-net agents. Functions are represented as agents with a single port and an associated rewriting rule, as shown in Listing 3. Calling such a function involves connecting the function agent to its argument (which may be represented as a tuple agent). The result is then obtained on the agent's port after a sequence of rewritings. This defines an interface function that is typically rewritten as a single implementation agent with more complex interaction rules.

```
(* Simple interaction rule; real implementation omitted *)
SMMap2(r) >< (f, m1, m2) =>
    SMMap2Decons_1(r, f, m2) ~ m1;
(* ...  *)
(* Operations on generic data are also agents and rules *)
op_add(r) >< (x, y) =>
    op_add_1(r, x) ~ y;

(* Calling the interface function with real data *)
m1 ~ SparseMatrix(4, 4, 16, 4, QTreeLeaf(UserValue(Some(1)))),
m2 ~ SparseMatrix(..., QTreeNode(...)),
SMMap2(r) ~ (%op_add, m1, m2);
r; free r;
```

Listing 3: An example of top-level functions and constructors translation result

Inpla has no I/O operations, so input data structures must be generated in advance. Error handling is minimal and relies on the virtual machine's "no interaction rule" exception for unforeseen situations, as implementing comprehensive checks would introduce significant cognitive overhead in such a low-level language.

Since we do not use any .NET-specific or F#-specific features in our reference implementation, translation to Vine is rather straightforward: this language provides algebraic data types, pattern matching, and standard `List`, `Option`, and `Result` types. We used the last for error handling. Vine supports file reading, so data loading was implemented. However, this makes performance comparison less accurate because, due to the nature of Interaction Nets, separation of steps is difficult.

In both Vine and Inpla implementations, weights in SSSP are represented as integers, not floats because Inpla does not support floats, and Vine supports floats but without comparison operators (possibly due to issues with NaN comparison).

4. Evaluation

The goal of this evaluation is to assess the applicability of runtimes based on Interaction Nets to real-world problems, identify bottlenecks and issues, and highlight directions for further investigation.

4.1. Experimental Setup

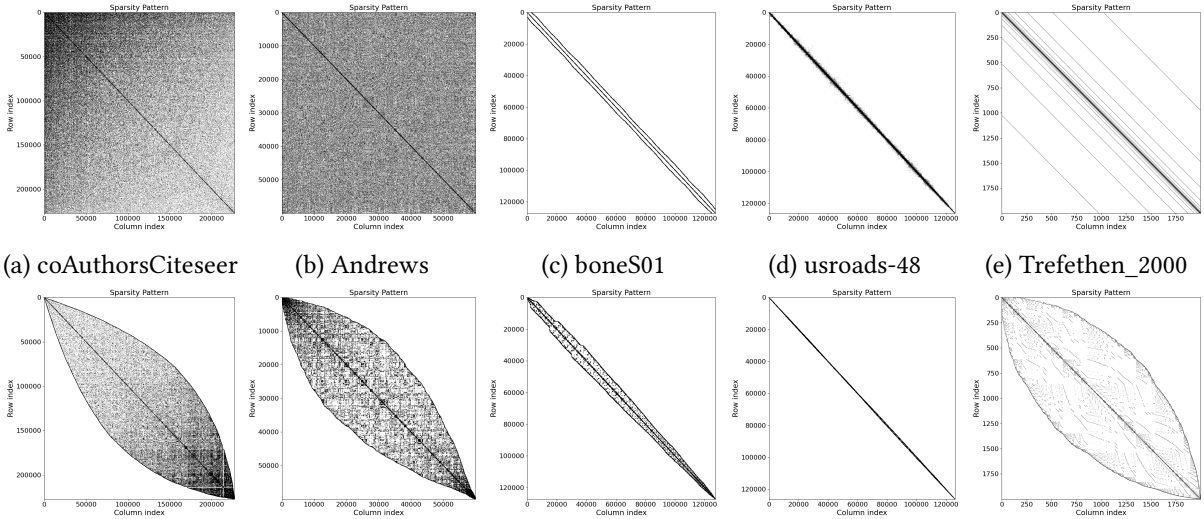
Hardware. All experiments were conducted on a system with an AMD Ryzen 7 5800X 8-core processor (16 threads with hyper-threading enabled, base clock 3.8 GHz) and 128 GiB of DDR5 RAM.

Competitors. We compare five implementations presented below.

- **F#**: Our quad-tree-based single-threaded implementation[6] that written in F# and compiled for the .NET 10.0 runtime. This serves as a baseline representing a conventional functional language approach and prototype for porting to Interaction Nets. We use BenchmarkDotNet[7] for benchmarking automation (e.g., for JIT warmup).
- **Inpla (version 0.13.4, patched)**[8]: Our interaction-net-based implementation[9] in Inpla low-level language, executed on the Inpla runtime system. Our patches of Inpla allow one to use arbitrary heap size configuration with per-thread adjustments via scripting. The patch also increases several virtual machine compile-time constants to support very large trees (thus, large graphs). Additionally, a bottleneck in lists parsing was discovered during testing, and the fix has since been merged into the main Inpla repository[10]. We perform three or five runs for each configuration, depending on data size.
- **Vine (commit #41e5ec67)**[11]: Our interaction-net-based implementation[12] in Vine high-level language, executed on the Vine runtime system with 110 GB of heap. We perform five runs for each configuration.
- **LAGraph[6] (version 1.2.1)**: A collection of graph algorithms built on SuiteSparse:GraphBLAS (version 10.3.1), providing state-of-the-art, linear-algebra-based reference implementations of BFS, SSSP, and triangle counting. We use the provided algorithms as a SOTA baseline for high-performance parallel graph analysis. Additionally, we use LAGraph as a baseline to analyze scaling across available threads. We perform 20 runs for each configuration.
- **NetworkX (version 3.6.1)**: A popular Python library for graph analysis. We use the algorithms implemented in NetworkX as a well-known baseline, without any special backends or settings. Evaluation is performed using pytest-benchmark, which automatically adjusts the number of runs according to execution time.

Dataset. We evaluate our implementations on several real-world and synthetic graphs with `Int` weights from the SuiteSparse Matrix Collection [7]. The graphs were chosen to represent a variety of sizes, sparsity patterns, and application domains, considering the capabilities and limitations of the selected competitors. Table 1 summarizes the number of vertices ($|V|$) and edges ($|E|$) for each selected graph. Weights were shifted relative to the minimum weight per graph to eliminate negative cycles. All graphs were converted to undirected form and have exactly one connected component. Thus, any vertex can serve as the start for BFS and SSSP. We selected vertex 0 for all runs.

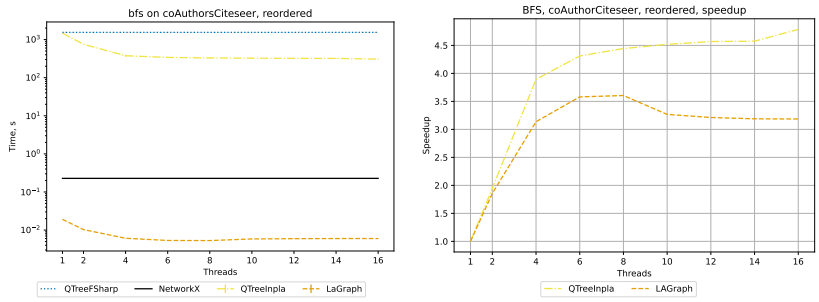
[6] Quad-tree-based linear algebra in F#: https://github.com/Lamagraph/QTreeFSharp.
[7] BenchmarkDotNet project: https://github.com/dotnet/BenchmarkDotNet.
[8] Patched version of Inpla: https://github.com/Lamagraph/inpla.
[9] Quad-tree-based linear algebra in Inpla: https://github.com/Lamagraph/QTreeInpla.
[10] Change request to Inpla with lists parsing fix: https://github.com/inpla/inpla/pull/4.
[11] Installed using instructions from documentation: https://vine.dev/docs/starting/installation.
[12] Quad-tree-based linear algebra in Vine: https://github.com/Lamagraph/QTreeVine.



(a) coAuthorsCiteseer (b) Andrews (c) boneS01 (d) usroads-48 (e) Trefethen_2000

(f) coAuthorsCiteseer (g) Andrews (h) boneS01 (i) usroads-48 (j) Trefethen_2000

Figure 4: Matrices shapes: original and after reordering



(a) Execution time (b) Scaling

Figure 5: Evaluation results for BFS on *coAuthorsCiteseer* (reordered)

Table 1: Dataset properties

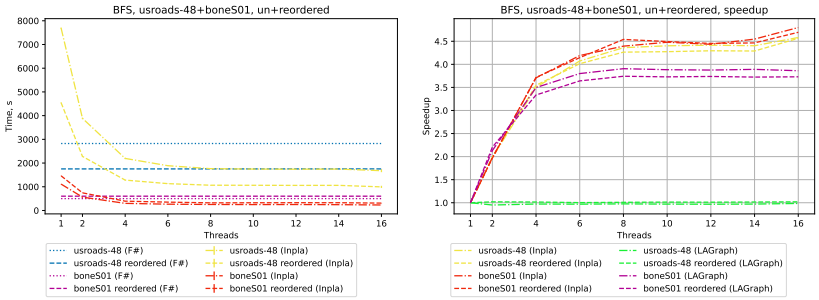
Graph	$\|V\|$	$\|E\|$
coAuthorsCiteseer	227,320	1,628,268
boneS01	127,224	5,516,602
usroads-48	126,146	323,900
Andrews	60,000	760,154
Trefethen_2000	2000	41,906
gr_30_30	900	7,744

Reordering. Sparse matrix reordering techniques can improve the performance of matrix operations [8]. In quad-tree-based representations, reordering can also reduce memory consumption. We apply the reverse Cuthill–McKee algorithm for reordering. Figure 4 shows the sparsity patterns of the selected matrices (the pattern for *gr_30_30* resembles that of *boneS01*). In our evaluation, we use both original and reordered matrices, except for *coAuthorsCiteseer*, where we use only the reordered version, as the original causes out-of-memory (OOM) error in Inpla.

Research Questions. Our evaluation addresses the following research questions.

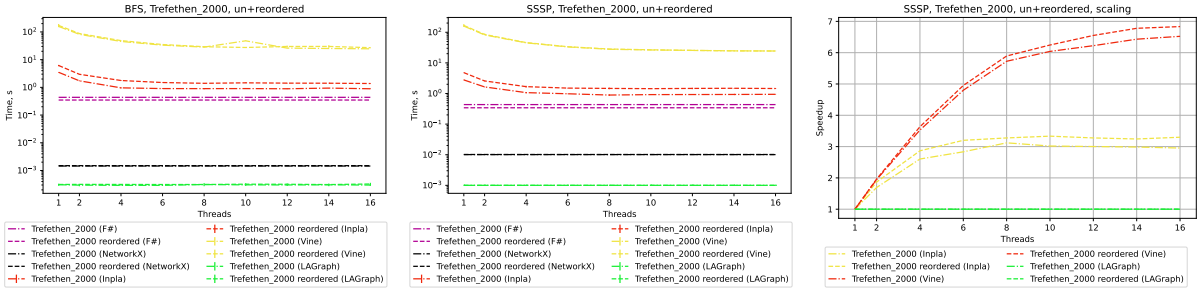
RQ1 (Performance). How does the execution time of Inpla and Vine compare to LAGraph, NetworkX and F# on BFS, SSSP, and triangle counting?

RQ2 (Scaling). How well do the Inpla, Vine, and LAGraph implementations scale with the number of available CPU cores? We measure speedup from 1 to all available cores for BFS, SSSP, and triangle counting.



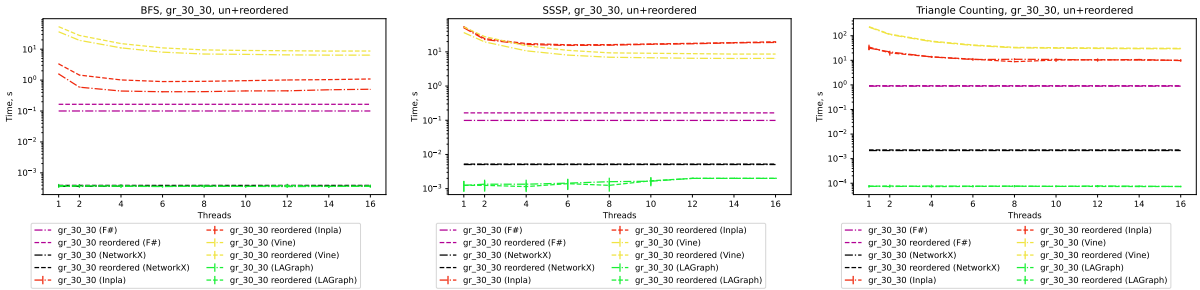
(a) Execution time (b) Scaling

Figure 6: Evaluation results for BFS on *boneS01* and *usroads-48* (both original and reordered)



(a) BFS (b) SSSP (c) SSSP scaling

Figure 7: Evaluation results for *Trefethen_2000* (both original and reordered)



(a) BFS (b) SSSP (c) Triangle counting

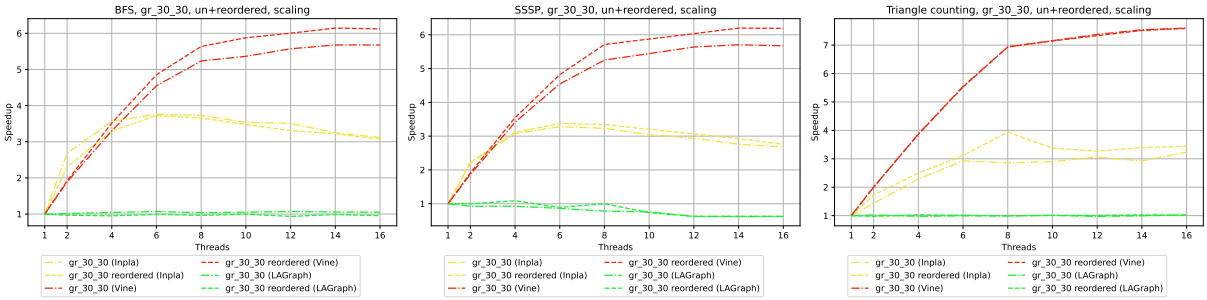
Figure 8: Evaluation results for *gr_30_30* (both original and reordered)

RQ3 (Impact of Matrix Reordering). How does reordering affect the performance of competitors?

4.2. Evaluation Results

First, we discuss the limitations that prevented evaluating all (graph, algorithm, tool) combinations. The primary limitation is the substantial memory consumption of Inpla and Vine. Since our implementations rely on a recursive divide-and-conquer computation pattern, aggressive duplication occurs during computations. While BFS and SSSP require only the `vxm` operation, triangle counting requires the `mxm` operation, which involves more complex recursive calls. As a result, triangle counting is feasible only on relatively small graphs. Notably, Vine encounters out-of-memory (OOM) errors even on *Trefethen_2000* when performing triangle counting. Second, Vine is excessively slow. Therefore, we can execute it within a reasonable time only on small matrices, namely *Trefethen_2000* and *gr_30_30*. Additionally, Inpla has data races that lead to instability[13]. We verified that the results are correct, but this does not guarantee that the races do not affect performance patterns.

[13]Some have already been fixed by our team (e.g., https://github.com/inpla/inpla/pull/8); others remain to be fixed.



(a) BFS scaling (b) SSSP scaling (c) Triangle counting scaling

Figure 9: Scaling analysis on *gr_30_30*

Performance analysis shows that in all cases (Figures 5a, 6a, 7a, 7b, 8a, 8b), interaction-net-based implementations are several orders of magnitude slower (up to 10^5) than both NetworkX and LAGraph. Note that in almost all cases, LAGraph is faster than NetworkX. Thus, for all selected cases—even for small graphs—the linear-algebra-based approach demonstrates reasonable performance. We also observe that when the graph is sufficiently large (*coAuthorsCiteseer*, *boneS01*, *usroads-48*, *Andrews*), Inpla can outperform the F# implementation, whereas on small graphs (*Trefethen_2000* and *gr_30_30*), F# outperforms Inpla by up to several orders of magnitude. This suggests that as graph size grows, the performance gap with NetworkX and LAGraph may decrease. However, this is difficult to verify due to Inpla's memory consumption problems.

Even on small graphs, Vine is slower than Inpla by up to an order of magnitude, as shown in Figures 7a, 7b, and 8a. However, Inpla is slower in the case of SSSP on *gr_30_30*, as presented in Figure 8b. Moreover, Inpla scales very poorly in this case, which may be caused by manual tuning of virtual machine parameters aimed at performance on large graphs. Nevertheless, this case should be investigated more deeply in the future.

Scaling analysis reveals that, in almost all cases (see Figures 5b, 6b, 7c, and 9c), the scaling behavior of all interaction-net-based implementations is straightforward: execution time decreases as the number of threads increases. Furthermore, scaling is often nearly linear for up to 8 of the 16 available threads (typically on 4–6 threads), a pattern that may be attributable to hardware configuration (e.g., hyperthreading or cache sizes). The exceptions are BFS and SSSP in Inpla on *gr_30_30* (Figures 9a and 9b), where scaling degrades beyond 6 threads: additional threads decrease performance. Figures 5b and 6b show that Inpla scales better than LAGraph on large graphs. Inpla's scaling depends primarily on matrix size rather than on the number of non-zero elements. This is evident in Figure 6b: *boneS01* and *usroads-48* have similar dimensions but vastly different numbers of non-zero elements. This observation is notable because LAGraph's scaling—in contrast to Inpla's—depends mostly on the number of non-zero elements. The difference suggests that, for tree-based representations, storage traversal dominates over element-wise computation. Vine scales better than Inpla in all cases, except on two threads for the *gr_30_30* graph. Consequently, a deeper examination of the Vine and Inpla virtual machines may yield insights into design better load balancing strategies.

Reordering analysis shows that in cases where the initial matrix has no specific pattern (*coAuthorsCiteseer* and *Andrews*), reordering helps reduce memory consumption and improve performance for all quad-tree-based implementations (including F#). In cases where the matrix has a diagonal-like initial pattern, the contribution of reordering is less straightforward. For both Inpla and F#, for matrices of equal size but significantly different sparsity, the effect of reordering differs: *usroads-48* is faster when reordered, while *boneS01* is slower when reordered. For small matrices, in almost all cases, Vine and Inpla process reordered matrices more slowly or see no effect. However, for F#, handling the reordered *Trefethen_2000* is faster than handling the original. Finally, for the smallest matrix *gr_30_30*, reordering degrades performance even for F#. Note that in some cases for small matrices (Figures 7c, 9a, 9b), while handling the reordered matrix is slower, it scales better. At the same time, for *usroads-48*, handling the reordered version scales slightly worse while performing better. Thus, data layout has a non-trivial

effect on performance and scaling and should be investigated more deeply.

We further analyze the total number of interactions performed by the virtual machines during evaluation, with results presented in Figure 10. As expected, Vine performs significantly more interactions, as it compiles code to a fixed set of low-level basic agents, whereas Inpla employs higher-level custom agents. This suggests that the Inpla approach may be better suited for optimizing domain-specific computations. In particular, it could benefit future hardware-based implementations of Interaction Nets by providing a means to extend a fixed basic set of agents and rules with custom ones tailored to specific computational tasks. An anomaly in the interactions count was detected for SSSP on *gr_30_30*: the gap between Vine and Inpla is significantly smaller than in the other cases. This observation correlates with the performance anomaly for the same case discussed above. Consequently, this case warrants detailed analysis.

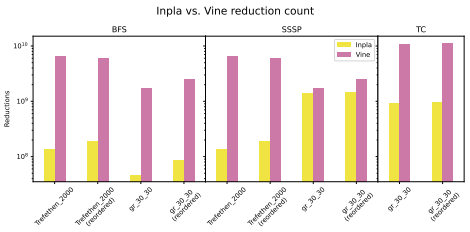


Figure 10: Total number of interactions

5. Conclusion and Future Work

Summarizing our evaluation, we highlight the following directions for further investigation. We note that our Inpla and Vine implementations are almost direct ports of the F# implementation, without heavy interaction-net-specific optimizations. Therefore, some of the observed bottlenecks may be resolved by utilizing appropriate programming patterns and practices. Formulating such practices remains a task for future research.

1. **Memory consumption.** Interaction Nets programming requires duplicator agents for data reuse. Inpla eagerly copies entire subnets during duplication, incurring substantial memory overhead relative to the F# prototype. Vine adopts a different strategy but still demands significant memory. Can coding patterns, evaluation strategies, and memory management mechanisms be co-designed to substantially reduce memory consumption?
2. **Alternative matrix representations.** Tree-based representations looks suboptimal: the recursive divide-and-conquer pattern induces aggressive duplication, traversal dominates cell handling, and performance is highly sensitive to reordering. Could coordinate list (COO) formats—operations on which rely on sorting, a task that parallelizes well in Interaction Nets—offer better memory efficiency and performance?
3. **Load balancing and scaling.** The current configuration of Inpla scales poorly on small graphs but well on large ones. Vine generally scales better than Inpla. Scaling and performance of both tools remain sensitive to matrix reordering. Can the load balancing strategy be improved to address these issues?
4. **Advanced fusion techniques.** Optimization techniques from functional programming, such as supercompilation [9] or distillation [10], could automate the fusion of operation sequences, analogous to manual kernel fusion for masks and other constructs in SuiteSparse:GraphBLAS. While partial evaluation has been shown possible for Interaction Nets [11], whether these more advanced techniques are adaptable remains an open question. In particular, such techniques can remove intermediate data structures, so it is necessary to investigate whether they can help solve memory consumption problems in some cases.

Finally, we believe our work motivates comparisons among different Interaction Nets implementations—in particular, by extending our evaluation to other systems such as HVM—and helps make Interaction Nets applicable to real-world problems.

Acknowledgments

This research has been supported by the St. Petersburg State University, grant id 116636233.

Declaration on Generative AI

During the preparation of this work, the authors used DeepSeek in order to: Grammar and spelling check. After using these tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] B. Brock, A. Buluç, T. G. Mattson, S. McMillan, J. E. Moreira, Introduction to graphblas 2.0, in: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021, pp. 253–262. doi:10.1109/IPDPSW52791.2021.00047.

[2] Y. Lafont, Interaction nets, in: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90, Association for Computing Machinery, New York, NY, USA, 1989, p. 95–108. URL: https://doi.org/10.1145/96709.96718. doi:10.1145/96709.96718.

[3] N. Huber, W. Yi, An encoding of interaction nets in ocaml, Electronic Proceedings in Theoretical Computer Science 417 (2025) 1–16. URL: http://dx.doi.org/10.4204/EPTCS.417.1. doi:10.4204/eptcs.417.1.

[4] I. Mackie, S. Sato, In-place graph rewriting with interaction nets, in: A. Corradini, H. Zantema (Eds.), Proceedings 9th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2016, Eindhoven, The Netherlands, April 8, 2016, EPTCS, 2016, pp. 15–24. URL: https://doi.org/10.4204/EPTCS.225.4. doi:10.4204/EPTCS.225.4.

[5] M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, S. Rajamanickam, KKTri: Fast Linear Algebra-Based Triangle Counting with KokkosKernels., Technical Report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2017.

[6] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, C. Yang, Lagraph: A community effort to collect graph algorithms built on top of the graphblas, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 276–284. doi:10.1109/IPDPSW.2019.00053.

[7] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, R. Sandstrom, The suitesparse matrix collection website interface, Journal of Open Source Software 4 (2019) 1244. URL: https://doi.org/10.21105/joss.01244. doi:10.21105/joss.01244.

[8] O. Asudeh, S. Mahdipour Saravani, F. Rastello, G. Sabin, P. Sadayappan, How effective is matrix reordering for improving performance of sparse matrix-vector multiplication?, in: Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC Workshops '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 823–827. URL: https://doi.org/10.1145/3731599.3767441. doi:10.1145/3731599.3767441.

[9] V. F. Turchin, Supercompilation: Techniques and results, in: D. Bjørner, M. Broy, I. V. Pottosin (Eds.), Perspectives of System Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 227–248.

[10] G. Hamilton, The next 700 program transformers, in: E. De Angelis, W. Vanhoof (Eds.), Logic-Based Program Synthesis and Transformation, Springer International Publishing, Cham, 2022, pp. 113–134.

[11] D. Béchet, Partial evaluation of interaction nets, in: M. Billaud, P. Castéran, M. Corsini, K. Musumbu, A. Rauzy (Eds.), Actes WSA'92 Workshop on Static Analysis (Bordeaux, France), September 1992, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings, Series Bigre, Atelier Irisa, IRISA, Campus de Beaulieu, 1992, pp. 331–338.