Graphite: A NUMA-aware HPC System for Graph Analytics Based on a new MPI $*$ X Parallelism Model

Mohammad Hasanzadeh Mofrad
Rami Melhem
University of Pittsburgh
Pittsburgh, USA
moh18, melhem@pitt.edu

Yousuf Ahmad
Mohammad Hammoud
Carnegie Mellon University in Qatar
Doha, Qatar
myahmad, mhhamoud@cmu.edu

ABSTRACT

In this paper, we propose a new parallelism model denoted as *MPI $*$ X* and suggest a linear algebra-based graph analytics system, namely, Graphite, which effectively employs it. MPI $*$ X promotes *thread-based partitioning* to distribute computation and communication across threads on a cluster of machines, while eliminating the need for unnecessary thread synchronizations. Consequently, it contrasts with the traditional *MPI + X parallelism model*, which utilizes *process-based partitioning* to distribute data among processes as a way to *scale out* on a cluster of machines (the MPI part), then splits each partition into subpartitions among the threads of each process as a method to *scale up* within a machine (the X part). Besides adopting MPI $*$ X, Graphite is NUMA-aware. In particular, it assigns threads to partitions in a way that exploits CPU and memory affinity, alongside leveraging faster MPI shared memory transport. Moreover, it adopts a variant of the popular GAS (Gather, Apply, and Scatter) computing model, thus decoupling the computation of partitions from the communication of partial results. Lastly, it supports thread-level asynchrony, which does not only overlap the computation with communication, but further interleaves multiple communications. We compared Graphite against GraphPad, Gemini, and LA3 graph analytics systems in an HPC environment using different graph applications. Results show that Graphite is roughly up to 3× faster than these state-of-the-art systems.

PVLDB Reference Format:
Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Graphite: A NUMA-aware HPC System for Graph Analytics Based on a new MPI $*$ X Parallelism Model. *PVLDB*, 13(6): 783-797, 2020.
DOI: https://doi.org/10.14778/3380750.3380751

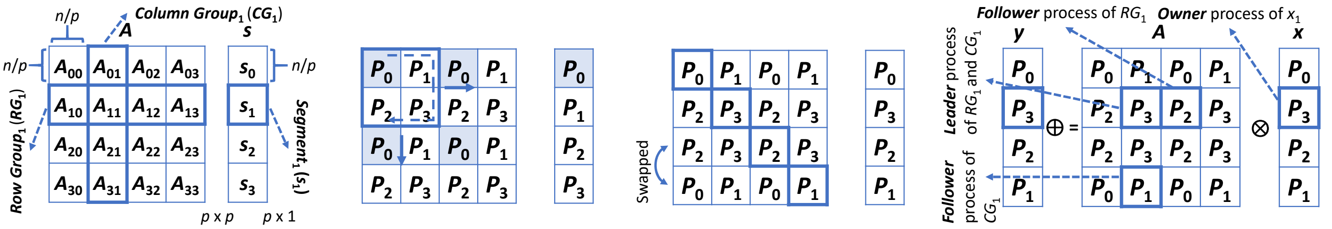
1. INTRODUCTION

High Performance Computing (HPC) and cloud computing are inherently and increasingly geared towards meeting the demands of Big Data and big graph analytics. As data grows, its analysis necessitates scaling HPC and cloud resources. The conventional **MPI + X parallelism model** [4, 52, 59] is a hybrid scheme with: (1) a Message Passing Interface (MPI) [26, 46] used for *horizontal scaling* (or scaling out) across cluster nodes, and (2) a shared memory paradigm, provided typically by a multithreaded library such as OpenMP [45] or Pthread [36], utilized for *vertical scaling* (or scaling up) across the cores of a node. More precisely, the MPI+X model employs *process-based partitioning*, whereby data is first partitioned across MPI processes (or scaled out), *then* each partition is further divided among the number of threads in each process (or scaled up). As such, it can be argued that the units of computation and communication in MPI+X are MPI processes. Moreover, the two scaling directions are orthogonal, where the *out* direction is dictated by the number of processes and the *up* direction is defined by the number of threads per process.

We propose **MPI $*$ X parallelism** that leverages *thread-based partitioning*, wherein data is partitioned directly based on the total number of threads (with $*$ indicating that threads are able to call MPI primitives). The MPI $*$ X model reduces the boundaries between threads and processes via considering threads as direct units of computation and communication. Specifically, it combines data partitioning and scaling directions in a unique way, providing a new paradigm of scaling, which we denote as *diagonal scaling* (or scaling over). With diagonal scaling, the load is distributed across all threads directly and equally, rather than scaling horizontally then vertically. In addition, since the code that is written for computation is inherently tailored for threads, a symmetric code path (control flow) is executed by all threads. Consequently, an MPI $*$ X system yields identical computation and communication patterns, while avoiding unnecessary synchronization points across threads. Afterwards, grouping threads into processes becomes simply an implementation issue, which should exploit the distributed and non-uniform memory access (NUMA) properties of the underlying architecture.

The conventional MPI + X model suggests launching one process per machine, after which shared memory is used for communication across threads inside a process (machine) and MPI over TCP/IP is used for communication across processes [4, 52, 59, 69]. In this paper, we show that with the MPI $*$ X model, grouping threads into a process on each NUMA socket gives superior performance because it allows threads to use the fast MPI shared memory transport for inter-socket communication [33], while avoiding the inefficiency resulting when two threads on different sockets of

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380751



(a) 2D-process-based partitioning **(b)** 2D-Cyclic placement **(c)** 2D-Staggered placement **(d)** Distributed SpMV $y = A \oplus . \otimes x$ (2D-S)

Figure 1: Matrix and vector 2D layouts ($p = 4$). (a) 2D-process-based partitioning of matrix and vector, (b) 2D-Cyclic process placement (e.g. the shaded tiles are assigned to P_0), (c) 2D-Staggered process placement, and (d) 2D-Staggered leader/follower configuration for distributed SpMV $y = A \oplus . \otimes x$.

the same machine share data (shared memory in a machine is physically distributed among sockets).

The MPI + X model has been widely used in graph analytics [1, 2, 11, 39]. In this paper, we focus on linear algebra-based graph analytics systems, especially that they emerged recently as efficient and scalable systems. In particular, we propose Graphite, a new linear algebra-based graph analytics system that adopts MPI $*$ X. It utilizes new 2D-thread-based matrix partitioning and placement approaches to slice the adjacency matrix of a graph into tiles and assign threads to them. Besides, it employs a GAS (Gather, Apply, and Scatter)-like matrix computing model [22], offering thereby a matrix-parallel abstraction for the computation and communication of threads. We compared Graphite against two linear algebra-based graph analytics systems, namely, GraphPad [2] and LA3 [1], and a graph theory-based system, namely, Gemini [69], using four different applications on 10 standard datasets. In short, Graphite is roughly up to 3× faster than these state-of-the-art systems.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 discusses the classical 2D-process-based matrix partitioning and placement approaches. In Section 4, we propose our 2D-thread-based matrix partitioning and placement paradigms. Section 5 discusses NUMA-aware thread placement and Section 6 summarizes features of the MPI $*$ X model. Section 7 puts it altogether and introduces Graphite. Results are reported in Section 8 and Section 9 concludes with some remarks.

2. RELATED WORK

Graph algorithms are traditionally solved over centralized and distributed graph analytics systems, which pursue a combination of fan-in/fan-out operations over vertices and their neighborhoods. Examples of conventional centralized shared memory graph analytics systems are Ligra [51], Galois [44], GraphLab [39], Julienne [18], GraphIt [67], Grazelle [24], TuFast [50], PnP [62], and Radar [3]. Also, examples of traditional distributed shared memory graph analytics systems are Pregel [42], PowerGraph [22], Distributed GraphLab [38], Mizan [29], Presto [60], GraphX [23], Pregelix [9], PowerLyra [14], PowerSwitch [61], GiraphUC [25], ShenTu [34], SHMEMGraph [19], TopoX [32], and Phoenix [16].

Linear algebra primitives such as Sparse Matrix - dense Vector (SpMV) can be used to solve graph algorithms via operating on adjacency matrices that represent input graphs. This powerful form of computation yielded linear algebra-based systems like Pegasus [28], KDT [40], CombBLAS [11], GraphPad [56, 2], Graphulo [20], GraphBLAS [12, 63, 65], SuiteSparse [17], LA3 [1], and GraphTap [43], among others.

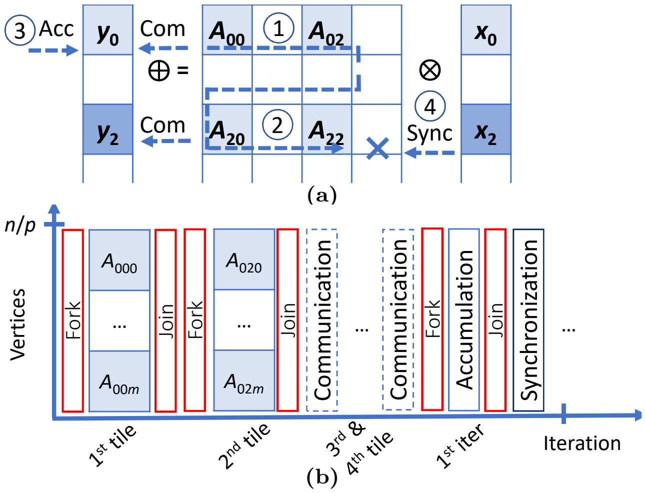
A different line of work, denoted as out-of-core graph analytics systems, attempted to utilize external storage like disks. Examples of these systems are GraphChi [31], GridGraph [70], FlashGraph [68], Graphene [37], Mosaic [41], and RealGraph [27]. In addition, various scalable array-based NoSQL (Not Only SQL) database systems suggested storing 2D adjacency matrices and nD tensors on disks and running complex analytics atop them. Examples of these systems are SciDB [54, 55] and TileDB [47]. Lastly, in the context of graph analytics, cache [5, 6, 66], memory [30], and NUMA optimizations [64, 69] were studied; with NUMA optimizations mostly focusing on CPU/memory affinity.

3. 2D-PROCESS-BASED MATRIX TILING & PLACEMENT

A graph can be represented by an adjacency matrix, in which an edge is denoted by a non-zero element in the matrix. Many real-world graphs consist of billions of vertices and tens of billions of edges. Typical linear-algebra based systems use 2D-process-based partitioning (tiling) to decompose a matrix into a 2D grid of tiles and achieve load balancing among processes [1, 2, 11, 56]. Having p processes, 2D tiling partitions the adjacency matrix of a graph G (say, A) with n vertices into a 2D p by p grid of tiles, producing p^2 tiles where each tile covers n/p rows and columns. This tiling creates a 2D layout of p *Row Groups* (*RGs*) and *Column Groups* (*CGs*) of tiles, where A_{ij} denotes the tile placed at i^{th} row group (RG_i) and j^{th} column group (CG_j). Similarly, a vector that can be involved in computation with the matrix (more on this shortly) is also partitioned into p *segments*, where each segment contains n/p elements.

Figure 1a shows the 2D-process-based partitioning of an exemplified matrix and a vector using $p = 4$. After partitioning, a process placement is pursued. To assign p processes to the 2D grid, many 2D-process-based placements put $\sqrt{p}$ processes per row/column group to limit the communication between processes [11]. Examples of this are 2D-Cyclic [2], which assigns processes in a cyclic order (Figure 1b), and 2D-Staggered [1], which further reorders row groups of the 2D-Cyclic to guarantee that each diagonal tile is assigned to a unique process, and thus aligns the assignment of row/column groups to processes (Figure 1c).

Many graph operations can be converted into simple linear algebra primitives. A common linear algebra primitive is SpMV operation $y = A \oplus . \otimes x$, where A is the $n \times n$ adjacency matrix of G, x and y are $n \times 1$ input and output vectors, and $\oplus . \otimes$ is a semiring equipped with $(+, \times)$ operators. Overloading the semiring with operators specific to the application allows SpMV to run different graph applications. The iterative SpMV algorithm repeatedly uses the result vector, y, from one iteration to compute the input vector, x, for the next iteration until convergence. Often, y is transformed to an intermediate vector v, which is then used to compute x.



(a)

(b)

Figure 2: GraphPad tile processing (MPI + X) [2] with $p = 4$ processes and $t = 2$ threads. Tiles are processed in a row-wise order where each tile is split into m smaller sub-tiles where $m >> t$ for balancing load among threads. (a) Steps taken to process tiles/segments by P_0: (1) and (2) are row group SpMVs followed by their communication episodes, (3) is the accumulation of results for the row group owned by P_0, and (4) is P_0's synchronization with other processes. (b) Compulsory forks/joins of t threads while processing each tile.

Figure 1d shows the assignment of tiles to processes in the 2D-Staggered placement. Processes are classified into two distinct categories, *leaders* and *followers*. The leaders (or processes assigned to diagonal tiles) are responsible for aggregating/broadcasting data from/to followers (or processes assigned to off-diagonal tiles) of their row/column group of tiles. In other words, leaders are the *owners* of their corresponding row/column group of tiles. Also, the leader of a row/column group of tile is the owner of the associated x and y vector segments and is responsible for maintaining/updating those segments.

2D-Cyclic and 2D-Staggered are classified as 2D-process-based placements, which can be used by the MPI + X model. Hence, systems relying on this parallelism model such as GraphPad [2], LA3 [1], Gemini [69], and CombBLAS [11] are not well suited for multithreading. As an example, Figure 2 shows how tiles are processed in GraphPad [2]. Specifically, Figure 2a is the sequence of processing tiles executing SpMV and accumulating the results for P_0 instructed by 2D-Cyclic (Figure 1b), while Figure 2b shows the steps taken by P_0 for tile processing. As illustrated in these figures, we identify multiple caveats with this scheme, namely (1) tiles have to be further split based on m which is a multiple of number of threads per process t (m sub-tiles in Figure 2b), (2) there are unnecessary compulsory thread forks and joins before and after the processing of each tile, (3) the main MPI process is responsible for the entire row group communication, (4) there are mandatory thread forks and joins for partial accumulation of results, and (5) the final synchronization point for checking the convergence is offloaded to the main process. These issues are potential sources of performance bottleneck and are applicable to other state-of-the-art graph processing systems such as Gemini [69] and LA3 [1]. In the next section, we show how thread-based tiling can fix them.

4. 2D-THREAD-BASED MATRIX TILING & PLACEMENT

To address the problems of process-based partitioning and placement, including compulsory synchronization points for threads and heavy communication load for MPI processes, we propose **2D-thread-based matrix partitioning and placement**, a scheme implied by MPI $*$ X parallelism, which intrinsically deems threads as the basic units of computation and communication and reduces synchronization points.

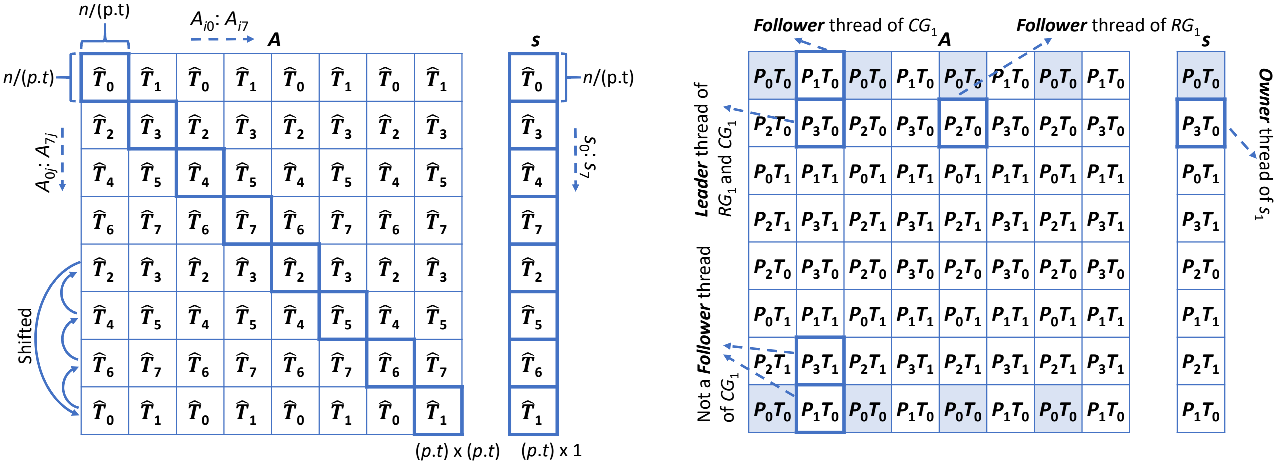
Let A be the n by n adjacency matrix of a graph G with n vertices. To distribute the computation of A to p processes each with t threads, 2D-thread-based partitioning divides A into a grid of $(p \cdot t)$ by $(p \cdot t)$ tiles, each with height/width of $n/(p \cdot t)$. Afterwards, it assigns $p \cdot t$ tiles to each thread and, subsequently, $p \cdot t^2$ tiles to each process. To this end, each row group has $\sqrt{p}$ threads/processes and each column group has $\sqrt{p \cdot t}$ threads and $\sqrt{p/t}$ processes. Also, each thread/process has tiles in $\sqrt{p}$ row groups and each thread has tiles in $\sqrt{p/t}$ column groups. Alongside, each process has tiles in $\sqrt{p \cdot t}$ column groups. These values only hold when both p and $p.t$ are square numbers. In general, however, an integer factorization method [2] shall be used to determine the number of processes and threads per row/column group of tiles (p_r/p_c and t_r/t_c) (Algorithm 1).

Partitioning and placement are two intertwined concepts, whereby partitioning produces the tiles, and placement assigns threads (or processes) to tiles. In this paper, the process-based 2D-Staggered placement [1] is extended to a **thread-based 2D-Staggered (2DT-Staggered) placement**. The input to the 2DT-Staggered is the 2D grid of $(p \cdot t)^2$ tiles produced by 2D-thread-based partitioning.

In 2DT-Staggered, if a diagonal tile, $A_{i,i}$, $i = 0, ..., (p \cdot t) - 1$, is assigned to a thread, then that thread becomes the leader of the i^{th} row group RG_i and i^{th} column group CG_i. Also, that thread renders the owner of the i^{th} segments of the input and output vectors, y_i and x_i. So, before executing the iterative SpMV $y_i = A_{ij} \oplus . \otimes x_j$ (in a right-multiplication fashion), the leader thread of CG_j sends x_j to its follower threads (threads that have tile(s) in CG_j). Later, after executing the SpMV of RG_i by all threads, the leader thread of RG_i receives partial results from the follower threads of that row group and accumulates them in y_i. Next, y_i is used to produce x_i via v_i (a segment of an intermediate vector v that stores results permanently) for the next iteration.

Figure 3a shows the 2DT-Staggered placement for eight global threads ($p = 4$, $t = 2$). From Algorithm 1: line 7 - 8, the 2DT-Staggered is materialized in two steps: (1) $p \cdot t$ thread ids are cyclically assigned to tiles, and (2) these ids are shifted so that each global thread $\hat{T}_k$ is assigned to exactly one diagonal tile. Figure 3b shows the process ids and local thread ids derived from Algorithm 1: lines 9 - 10. Each process P_k is assigned to exactly t diagonal tiles, and each local thread in P_k receives one diagonal tile. RGs are distributed among processes in a staggered way, and then among their local threads in a row-wise way. The staggered property balances the computation/communication among threads, while the row-wise property eliminates the concurrent writes into segments of y by multiple threads.[1]

[1] The uniqueness of diagonal processes/threads ids can be proved by derangement, which is a permutation of elements of a set i.e. no element appears in its original position [21] (Algorithm 1: lines 7-8 creates deranged id permutations).



(a) 2DT-Staggered global thread assignment **(b)** 2DT-Staggered process and local thread assignment

Figure 3: Tile layout for $p = 4$ and $t = 2$. The 2D grid has $(p \cdot t)^2 = 64$ tiles with $p \cdot t = 8$ tiles per thread and $(p \cdot t)^2 = 16$ tiles per process. In (a), rows marked as *shifted* are shifted to guarantee having t diagonal tiles per process. In (b), P_iT_j denotes thread j of process i. Leader threads are at diagonal tiles, and follower threads have the same ids as their leader. Note that each thread is responsible for 8 tiles (e.g., the 8 tiles and 1 segment processed by thread P_0T_0 are shaded in (b)).

Algorithm 1: 2D-thread-based Staggered *tile to process/thread assignment* (See Table 1 for notation).

```
Input: # of processes $p$ and # of threads per process $t$
Output: Assignment of Tiles$[p \cdot t][p \cdot t]$ to global thread ids $\hat{T}_k, k = 0, ..., (p \cdot t) - 1$ (Figure 3a)
            Derivation of process id $P_k, k \in [0, p-1]$, and local thread id $T_k, k \in [0, t-1]$ from global thread id (Figure 3b).
$gcd = \texttt{GCD}(t_r, t_c)$
for $i = 0$ to $p.t$ do
    for $j = 0$ to $p.t$ do
        Tiles$[i][j].\hat{T} = ((i \mod t_c) \cdot t_r) + (j \mod t_r)$
        Tiles$[i][j].\hat{T} += (\lfloor i/(p.t/gcd) \rfloor \cdot r_t) \mod (p \cdot t)$      ▷ Assignment of tiles to global threads
        Tiles$[i][j].P$ = Tiles$[i][j].\hat{T} \mod p$      ▷ Grouping of threads into processes
        Tiles$[i][j].T$ = Tiles$[i][j].\hat{T}/p$      ▷ Derivation of local thread ids
```

Table 1: 2D-process-based tiling versus 2D-thread-based tiling. The function $\texttt{Factorize}(p)$ returns p_r and p_c such that $p_r \cdot p_c = p$ and $|p_c - p_r|$ is minimized.

	2D-process-based	2D-thread-based
# of row/ column group of tiles	p	$p \cdot t$
# of tiles	$p \cdot p = p^2$	$(p \cdot t) \cdot (p \cdot t) = (p \cdot t)^2$
Tile height/ width	n/p	$n/(p \cdot t)$
Tile area	$(n/p)^2$	$(n/(p \cdot t))^2$
# of processes per row/column group (p_r / p_c)	$(p_r, p_c) = \texttt{Factorize}(p)$	$(p_r, p_c) = \texttt{Factorize}(p)$
# of threads per row/column group (t_r/ t_c)		$t_r = p_r, \quad t_c = (p \cdot t)/t_r$
# of row/column groups per process (r_p/ c_p)	$r_p = p/p_c, \quad c_p = p/p_r$	$r_p = (p \cdot t)/p_c, \quad c_p = (p \cdot t)/p_r,$
# of row/column groups per thread (r_t/ c_t)		$r_t = (p \cdot t)/t_c\ , \quad r_c = (p \cdot t)/t_r$

Generally, quick bursts of computation are interleaved with bursts of communication as a result of overlapping computation with communication and, accordingly, achieving scalability. As summarized in Table 1, the area of tiles in 2D-thread-based partitioning is t^2 times smaller than in 2D-process-based partitioning (which is reasonably small and suitable for overlapping). Moreover, the MPI + X model, which uses 2D-process-based partitioning, considers p processes for carrying out communication. Conversely, MPI $*$ X, which uses 2D-thread-based partitioning, utilizes $p.t$ threads to pursue communication. Hence, the MPI $*$ X has t times more communication endpoints and, as such, a better degree of overlapping computation with communication. In summary, MPI $*$ X is a scalable parallelism since it overlaps the computation of reasonably smaller tiles with the communication of fairly smaller messages per thread.

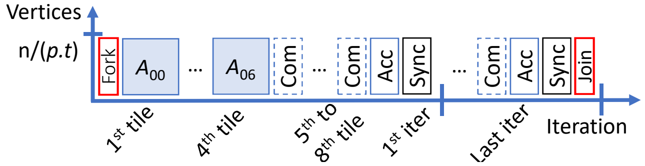


Figure 4: Tiles processed by thread P_0T_0; shaded tiles in Figure 3b (MPI $*$ X). P_0T_0 has a single fork/join, and the synchronization is delayed till the end of an iteration to maximize the overlapping of computation and communication.

Figure 4 demonstrates the advantages of 2D-thread-based over 2D-process-based: (1) 2D-thread-based inherently distributes the computation of tiles among threads, resulting in each thread being only forked/joined before/after the first/last iteration. Clearly, this avoids the overhead of fre-

Table 2: The traditional MPI + X versus the new MPI $*$ X parallelism models.

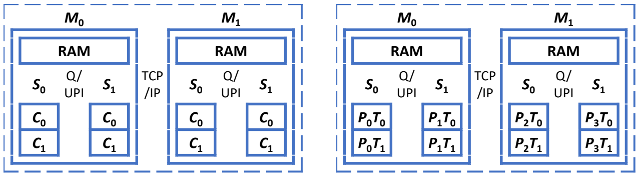
		MPI + X parallelism	MPI * X parallelism	MPI * X advantages
❶	Partitioning & placement	Process-based	Thread-based	More overlapping of computation and communication
❷	Computation & communication units	Processes	Threads	Front-loading the computation and communication patterns
❸	Synchronization	Process-based barriers	Thread-based barriers	Avoids compulsory forks and joins, and minimizes synchronization
❹	Process layout	One process per machine	One process per socket	Enabling NUMA-aware computation and communication, and cache locality
❺	Communication among multiple processes	Process-based inter-machine MPI over TCP/IP	Thread-based inter-socket MPI using shared memory/ Thread-based inter-machine MPI over TCP/IP	Enabling faster MPI shared memory transport via Q/UPI interconnect
❻	Communication among threads within a process	Intra-/Inter-socket shared memory	Intra-socket shared memory	Avoid inter-socket communication
❼	Scaling	Horizontal **then** vertical scaling	Horizontal **and** vertical (diagonal) scaling	Removing the unnecessary boundaries between processes and threads

quent thread creation/termination and enables cooperative thread synchronization at the end of each iteration. (2) 2D-thread-based evenly splits the row/column group communication among threads, eliminating thereby the communication bottleneck caused by offloading communication to only MPI processes in the process-based variant. (3) 2D-thread-based offers a higher degree of overlapping between computation and communication because of its smaller tiles and larger number of MPI endpoints.

5. NUMA-AWARE PLACEMENT IN 2D-THREAD-BASED TILING

The 2DT-Staggered placement assigns $P_0, ..., P_{p-1}$ process ids to tiles in a staggered way. Also, it assigns local thread ids of a process, e.g., $P_0T_0, ..., P_0T_{t-1}$ for P_0 to different rows. In Figure 3b, threads placed in the same row/column group, but belong to different processes (e.g., P_0T_0 and P_1T_0), use MPI to communicate with each other. Also, threads placed in the same row/column group, but belong to the same process (e.g., P_0T_0 and P_0T_1 in P_0), use shared memory to communicate. MPI has two transports, TCP/IP transport with 4 GB/s speed [48] used for inter-machine communication, and shared memory transport with 60 GB/s speed [48] used for intra-machine communication. Thus, a NUMA-aware assignment of MPI endpoints to tiles will benefit from the faster shared memory transport.

The linear order of process/thread ids does not necessarily imply assignments of processes to machines/NUMA sockets and threads to cores. These assignments are done by the MPI/threading environments before launching an MPI application and do not necessarily follow an expected order such as a sequential assignment order. However, knowing these assignments, processes/threads can be reordered before populating the 2D grid to efficiently exploit MPI's shared memory transports. To reorder processes, we first gather five pieces of information, namely, the topology of the cluster (using MPI [26]), the microarchitecture of machines (using NUMActl [35]), the number of processes per machine (using MPI [26]), the number of threads per process (using OpenMP [45]), and the number of processes/threads per row/column group of tiles (using the integer factorize method [2]). Finally, tapping into these, we reorder processes to maximize the MPI shared memory communication.



(a) (b)

Figure 5: (a) A cluster with two dual-core dual-socket NUMA machines, and (b) NUMA-aware assignment of threads to cores with $p = 4$ and $t = 2$.

For instance, consider a simple cluster consisting of two NUMA machines as shown in Figure 5a, and its NUMA-aware assignment of processes/threads to machines/sockets/ cores as shown in Figure 5b. Combining the information of thread assignment to tiles (from Figure 3b) with the information of thread assignment to cores from Figure 5b), a NUMA-aware assignment of threads to cores can maximize the usage of the MPI shared memory transport for inter-socket communication among different MPI endpoints. Furthermore, from our experiments, we found out that assigning one MPI process per socket provides faster MPI communication as the usage of integrated memory controller of NUMA sockets like Intel's QuickPath Interconnect (QPI) or Ultra Path Interconnect (UPI) with 16 GB/s speed [48] is only limited to the threads of the two processes placed in different sockets of a machine. Alongside, the controller is not used for shared memory accesses of threads inside the same process (because the shared memory communication of threads inside each process is limited to a single socket.)

6. SUMMARY OF MPI $*$ X FEATURES

In Table 2 we outlined the main characteristics of the MPI $*$ X parallelism model and contrast them with those of the classical MPI + X model. The key advantages of MPI $*$ X stem from the 2D-thread-based partitioning which elevates threads to first-class citizens across the computation, communication, and synchronization. Specifically partitioning the input matrix into smaller tiles based on the total number of threads, thus evenly balancing the computation load among threads in a fine-granular way (❶ in Table 2). Moreover, threads are communicating endpoints which provides a finer degree of computation and communication overlapping when using MPI asynchronous primitives (❷ in Table 2). In addition, in MPI $*$ X, threads are persistent throughout computation, and synchronization is performed directly among threads at the end of each iteration. Thus, MPI $*$ X has less synchronization overhead (❸ in Table 2). Conversely, in MPI + X, threads are *forked* and *joined* at every iteration and synchronization is applied between threads in each MPI process *then* among MPI processes.

MPI $*$ X leverages NUMA where this micro-architectural property provides the following benefits (❹ through ❻ in Table 2). Specifically, by launching one process per socket, MPI $*$ X's threads enjoy processor/memory affinity where threads are bound to unique processors and, subsequently exploit L1 cache locality. Also, threads' memory accesses are local to their host sockets, restricting the shared memory communication of threads to those sockets which also avoids overloading the QPI/UPI interconnect. Moreover, combining the 2D-thread-based tiling with the micro-architectural information allows MPI $*$ X to take advantage of the MPI shared memory transport for inter-socket communication within a machine. Accordingly, MPI $*$ X offers fast inter-socket communication using the QPI/UPI interconnect.

Finally, MPI $*$ X incorporates diagonal scaling (❼ in Table 2), which blurs the boundaries between processes and threads, and front-loads computation, communication, and synchronization among threads. The diagonal scaling is possible because the abstraction model, the library specification, and hardware properties are seamlessly integrated.

7. THE GRAPHITE

In this section, we discuss Graphite, a new linear algebra-based distributed graph analytics system that employs the MPI $*$ X parallelism model. Graphite uses 2D-thread-based partitioning and placement (,i.e., 2DT-Staggered placement) to equally break the computation and communication of a sparse matrix among threads, while avoiding non-compulsory synchronizations. It scales diagonally and treats threads as basic units of computation and communication. Internally, Graphite utilizes MPI's `MPI_THREAD_MULTIPLE` option in conjunction with splitting the MPI communicator to enable collective and point-to-point communication between computing threads.

7.1 Multithreaded MPI Input Processing

Graphite supports distributed reading of plaintext and binary unweighted/weighted edge lists (which represent input graphs). For unweighted edge lists, it only stores the source and destination of each edge without a weight. Graphite has a built-in graph converter to manipulate an input graph based on problem constraints such as transposing it, making it acyclic, removing self-loops, or removing parallel edges. The 2D-thread-based partitioning used in Graphite instructs threads to collectively read edges from an edge list and insert them in their associated tiles. Tiles are compressed using Triply Compressed Sparse Column (TCSC) [43], a new sparse matrix compression format offering a compact representation of given sparse matrix and vectors. In addition, TCSC supports a new optimized variant of the SpMV primitive that takes advantage of the sparsity distribution of the matrix and vectors. This variant is called SpMSpV^2 (Sparse Matrix - Sparse input and output Vectors) which filters the empty rows and columns of a sparse matrix and vector.

7.2 Distributed SpMSpV^2 using 2D-thread-based Tiling & Placement

As pointed out earlier, in Graphite, tiles are compressed using TCSC [43], which enables distributed execution of SpMSpV^2 at scale. Rendering an n by n matrix A that represents a graph G with n vertices, a graph operation can be translated into an SpMSpV^2 primitive $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$; where $\bar{\bar{A}}$ is a $nzr \times nzc$ compressed matrix holding no empty rows and columns, and $\bar{x}$ and $\bar{y}$ are $nzc \times 1$ and $nzr \times 1$ compressed input and output vectors, with nzc and nzr standing for the numbers of nonzero columns and rows, respectively. Graphite is a *vertex-centric* system that abstracts the iterative computation of a large graph from the standpoint of a vertex. A vertex has a *value* (or state) containing some information about the problem being solved. Hence, there is a value (state) vector v of length n, which is divided into multiple segments like the $\bar{x}$ and $\bar{y}$ vectors. To run an application, first, v is interpolated to construct the new compressed input vector $\bar{x}$. Second, the SpMSpV^2 primitive $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$ is executed to produce the compressed vector $\bar{y}$. Finally, $\bar{y}$ is expanded to an uncompressed value vector v to interpolate and store the results permanently; as it will be used to construct the new $\bar{x}$ in the next iteration. We next formalize this sequence as an iterative matrix computing model where this model closely works with the new 2D-thread-based partitioning and placement.

7.3 Matrix Computing Model

Many Vertex-centric systems [1, 2, 15, 39, 42] assume graph-parallel abstractions such as *vertex programs* for encapsulating the operations executed on vertices of a graph. To collect and disseminate information, the GAS (Gather, Apply, and Scatter) model [22] adds fan-in/fan-out operations to a vertex program and characterizes the differences between vertex and edge computations. The Gather operation collects information about adjacent vertices and edges via a centralized sum. The Scatter operation propagates the new value of a central vertex through its adjacent edges. Finally, the Apply operation updates the value of the central vertex. Graphite adopts a similar model and iterates through *Broadcast* (Scatter in GAS), *Combine* (Gather in GAS), and *Apply* operations.

Graphite's computing model suggests a vertex program that can be overloaded with the desired code for the Broadcast, Combine, and Apply operations. Before running the vertex program, tiles of $\bar{\bar{A}}$ and segments of $\bar{x}$, $\bar{y}$ and v are distributed among threads using 2DT-Staggered, where $\bar{\bar{A}}_{ij}$ is the tile placed at the intersection of i^{th} and j^{th} row and column groups of tiles. In 2DT-Staggered, each thread is the leader of a unique row/column group of tiles and their corresponding segments of $\bar{y}/\bar{x}$ vectors (although it may have tiles in multiple row/column groups). Therefore, the k^{th} thread $T_k \mid k \in [0, t-1]$ of a process is the leader of the k^{th} uniquely owned row/column group and the associated vectors segments.

Algorithm 2 demonstrates the pseudocode of Graphite's GAS-like matrix computing model. Also, Figure 6 sketches the operations of the matrix computing model of Graphite which is used in conjunction with 2D-thread-based partitioning and placement for matrix parallel computations. In the following, we delve deeper into Graphite's computing model and discuss Broadcast, Combine, and Apply operations in details.

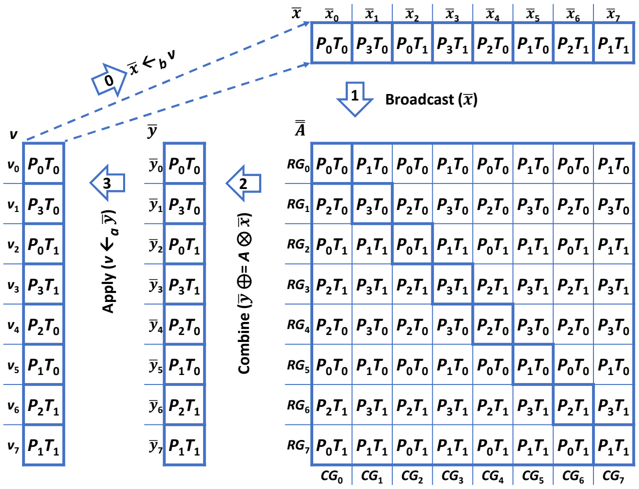
Algorithm 2: Matrix Computing Model

```
1: Input: Tiles of matrix $\bar{\bar{A}}$ and $\bar{x}$, $\bar{y}$ and $v$ vectors
2: Input: Overloaded functions to implement the operators
   for combine $(\otimes, \oplus)$, apply $(\leftarrow_a)$ and broadcast $(\leftarrow_b)$
3: Temporary vector: $\hat{\bar{y}}$
4: for $k = 0$ to $t$ do fork($T_k$)    ▷ Pin $T_k$ to a unique core
5: Initialize $v_k$    ▷ Every thread $T_k$ executes the following:
6: do
7:     $\bar{x}_k \leftarrow_b v_k$                              ▷ Broadcast
8:     for $\forall j \in CG$ do
9:         MPI_Ibcast($\bar{x}_j$, $leader_j$, MPI_COMM_COL$_j$)
10:
11:    for $i,\ j \in \bar{\bar{A}}$ do                             ▷ Combine
12:        $\bar{y}_i \oplus= (\bar{\bar{A}}_{ij} \otimes \bar{x}_j)$
13:        if $RG_i$ tiles are processed then
14:            if $T_k$ is $leader_i$ then
15:                for $T_l \in RG_i$ followers do
16:                    MPI_Irecv($\hat{\bar{y}}_{il}$, $T_l$, MPI_COMM_ROW$_i$)
17:            else
18:                MPI_Isend($\bar{y}_i$, $T_k$, MPI_COMM_ROW$_i$)
19:    $\bar{y}_i \oplus= \sum_l \hat{\bar{y}}_{il}$
20:
21:    $v_k \leftarrow_a \bar{y}_k$                                  ▷ Apply
22: while Not CONVERGED($T_k$)    ▷ Check convergence
```

7.3.1 Broadcast Operation

At the beginning of each iteration, each k^{th} leader thread (the leader of the k^{th} owned column group) calls the *Broadcast* operation (Algorithm 2: lines 7 - 9, where $\leftarrow_b$ is the broadcast operator) to produce the new input segment $\bar{x}_k$ from an interpolation of v_k values (e.g., new ranks in PageRank). This transformation is marked by arrow 0 in Figure 6. Later, as signified by arrow 1, each T_k thread broadcasts the new input segment $\bar{x}_k$ to its followers in its column group, which enables every thread to receive new inputs required for the Combine operation. This is equivalent to the GAS's Scatter operation, which is utilized to fan-out on outgoing edges and send new inputs to neighboring vertices.

In systems like GraphPad [2] and LA3 [1], GAS's Scatter operation is implemented using point-to-point primitives (`MPI_Isend()`/`MPI_Irecv()`) on the global MPI communicator. In Graphite, the two exemplified MPI primitives are merged into a single `MPI_Ibcast()` function (which is faster than point-to-point primitives due to using a tree-based communication algorithm [33]) via splitting the MPI communicator. To enable broadcasting messages inside a column group, we first create independent MPI column group communicators MPI_COMM_COL$_j$ for the threads in the same column group, CG_j, of tiles. While `MPI_COMM_WORLD` enables broadcast and collective communication among *all* processes, splitting the communicator into subgroups enables broadcast and collective communication among threads hosted by the same column group. As a matter of fact, broadcasting across column group communicators mitigates the pressure on the global communicator and avoids potential delays and contentions. Also, since each thread T_k is the leader in only one of its column groups (the root process of the `MPI_Ibcast()`) and a follower in the rest, a nonblocking broadcast can overlap communication among threads. As such, threads can simultaneously send/receive different input segments associated with different column groups and interleave the communication of *sends* with *receives*.



x̄ x̄0 x̄1 x̄2 x̄3 x̄4 x̄5 x̄6 x̄7
P0T0 P3T0 P0T1 P3T1 P2T0 P1T0 P2T1 P1T1
0 x̄ ←b v
1 Broadcast (x̄)
v
v0 P0T0
v1 P3T0
v2 P0T1
v3 P3T1
v4 P2T0
v5 P1T0
v6 P2T1
v7 P1T1
3 Apply (v ←a ȳ)
ȳ
ȳ0 P0T0
ȳ1 P3T0
ȳ2 P0T1
ȳ3 P3T1
ȳ4 P2T0
ȳ5 P1T0
ȳ6 P2T1
ȳ7 P1T1
2 Combine (ȳ ⊕= A ⊗ x̄)
Ā̄
RG0 P0T0 P1T0 P0T0 P1T0 P0T0 P1T0 P0T0 P1T0
RG1 P2T0 P3T0 P2T0 P3T0 P2T0 P3T0 P2T0 P3T0
RG2 P0T1 P1T1 P0T1 P1T1 P0T1 P1T1 P0T1 P1T1
RG3 P2T1 P3T1 P2T1 P3T1 P2T1 P3T1 P2T1 P3T1
RG4 P2T0 P3T0 P2T0 P3T0 P2T0 P3T0 P2T0 P3T0
RG5 P0T0 P1T0 P0T0 P1T0 P0T0 P1T0 P0T0 P1T0
RG6 P2T1 P3T1 P2T1 P3T1 P2T1 P3T1 P2T1 P3T1
RG7 P0T1 P1T1 P0T1 P1T1 P0T1 P1T1 P0T1 P1T1
CG0 CG1 CG2 CG3 CG4 CG5 CG6 CG7

Figure 6: Integrating the matrix computing model (Broadcast, Combine, and Apply) with 2D-thread-based tiling to run SpMSpV^2 ($p = 4$ and $t = 2$).

7.3.2 Combine Operation

As marked by arrow 2 in Figure 6, after broadcasting the new input $\bar{x}$, the *Combine* operation runs the SpMSpV^2 kernel. This is similar to GAS's Gather operation, which fan-ins and calculates a generalized sum over a neighborhood of a vertex. In the Combine operation (Algorithm 2: lines 11 - 19), each thread T_k iterates over its tiles in a *row order fashion* and executes the SpMSpV^2 kernel on its edges (i.e., $\bar{y}_i \oplus= (\bar{\bar{A}}_{ij} \otimes \bar{x}_j)$, where i and j are the indices of i^{th}/j^{th} row/column group of tiles, or i^{th}/j^{th} segments of $\bar{y}_i/\bar{x}_j$). After consuming tiles related to i^{th} row group, follower threads post their sends to the leader thread of the i^{th} row group, while leader threads post receives for partial output segments $\hat{\bar{y}}_i$'s from their followers. Afterwards, all threads move on to their next row group of tiles asynchronously. Once, all tiles are consumed, each T_k adds the partial result segments of $\hat{\bar{y}}_i$ to its $\bar{y}_i$ segment, which later will be used to update the i^{th} segment of value vector v_i.

To expedite the communication of the Combine operation, we split the global communicator into row group communicators, whereby threads inside the same row group of tiles, RG_i, use the same row group communicator MPI_COMM_ROW$_i$ for sending/receiving partial results. Combine uses the row group communicator to send/receive partial accumulation results. Moreover, for row group communication, the number of communicators is set equal to the number of row groups per process in order to provide concurrent race-free communication for all threads. Furthermore, Combine uses the MPI asynchronous communication routines, including `MPI_Isend()` and `MPI_Irecv()` to overlap the computation of tiles with the communication of partial $\hat{\bar{y}}$ segments. Hence, at the end of each row group, follower threads post their sends and leader threads post their receives. Subsequently, all threads carry on independently with processing their next row group of tiles, while MPI buffers are still being sent/received in the background. Since the communication is only performed when the last tile of a row group is consumed, only a single pair of send/receive is required to transfer the partial results from a follower to the leader thread of that row group. Lastly, the accumulation of the segment owned by each leader, T_k, is done when all receives are completed as T_k's receives are sufficient for accumulation.

7.3.3 Apply Operation

Marked by arrow 3 in Figure 6, in the *Apply* operation, each leader thread T_k interpolates its owned output segment $\bar{y}_k$ and constructs the new vertex values v_k (Algorithm 2: line 21, where $\leftarrow_a$ is the apply operator). This is similar to the GAS's Apply operation, which updates the state of the central vertex.

Finally, our matrix computing model operations, including Broadcast, Combine, and Apply are followed by a check for convergence, which is also run concurrently by all threads. Depending on the application requirements, this sequence repeats until executing a certain number of iterations or reaching convergence.

7.4 Leveraging NUMA in Graphite

7.4.1 NUMA-aware Shared Memory Communication

As discussed in Section 7.3.2, our computing model relies on point-to-point primitives for the Combine operation, which can be effectively accelerated using the MPI shared memory transport. Guided by the MPI $*$ X model, which suggests launching one MPI process per socket, we place threads that belong to two processes launched at the same machine in the same row group of tiles in the 2D grid of tiles. Therefore, the inter-socket communication can be highly optimized using the MPI shared memory transport (see Section 5). Having this setting, the communication of the Combine operation is overlapped with its computation of tiles, which further alleviates the use of point-to-point MPI primitives. Contrarily, column group communication cannot benefit from the MPI shared memory transport because column group processes run mostly on different machines. In this case, however, `MPI_Ibcast()` already offers swift TCP/IP communication which mitigates the lack of having a better transport.

7.4.2 Processor & Memory Affinity

Processor/memory affinity avoids excessive migrations of processes/threads, thus allowing them to benefit from hot caches and NUMA. GraphPad [2] and Gemini [69] leverage MPI [26, 46] and OpenMP [45] to control CPU and memory affinity at runtime. In contrast, Graphite explicitly controls affinity by launching one MPI process per socket and pinning threads to cores. In particular, the *processor affinity* forces threads to be launched at the same NUMA socket as the MPI process. This translates to fewer context switches, TLB flushes, and L1 cache invalidations. Also, it offers efficient L2/L3 cache accesses because an access to L2 is limited to the working thread pinned on a core and an access to L3 is limited to the working threads running on that cores' socket. Moreover, *memory affinity* enforces contiguous allocation of memory for matrix tiles and vector segments on a NUMA node when the MPI process uses `numa_alloc_onnode` [35]. Memory affinity avoids overloading the memory interconnect across sockets such as QPI/UPI and offers faster main memory accesses. Also, binding a core to a thread allows all data structures of tiles and segments to be allocated at the same NUMA socket of the core. All in all, threads can subsequently enjoy hot caches while running SpMSpV^{2}s on $\bar{x}$ and $\bar{y}$ segments. Moreover, within a process, there is only one $\bar{x}$ segment per column group from which all threads can safely read in parallel.

7.5 Enabling Compiler Optimization

Based on our experience with multithreaded programming, compiler optimizations are not fully supported (or are degraded– e.g., from `-O3` to `-O2`) while developing programs with cross-function and/or cross-file invocations by threads. The loss from the absence of compiler optimizations and the presence of sandboxing (where programs are sandboxed in multithreading runtimes, thus inducing overhead) may completely offset the gain from multithreaded programming [7, 57, 58]. As such, we keep the iterative compute-intensive SpMSpV^2 kernels concise and overload them locally with basic mathematical operators instead of using expensive cross-function calls. In addition, we avoid using `virtual` methods because they enforce each function call to go through a virtual table to look up and invoke the callee method. To this end, we utilize `inline` methods, which allow the compiler to see the majority of code in advance and, accordingly, exploit vectorization and loop unrolling. Lastly, to effectively break the code and computation among threads, we use Pthread instead of OpenMP, which is about 20% faster [53].

7.6 Activity & Computation Filtering

Graph applications are divided into *stationary applications*, where *all* vertices remain active during execution, and *non-stationary applications*, where the number of active vertices varies during runtime.

Activity filtering is a technique used in non-stationary applications to remove unnecessary computation and communication of inactive vertices [1, 2, 69]. In Graphite, if less than 60% of vertices render inactive, only a list of (index, value) pairs representing active vertices are used for communication, precluding thereby any traffic data that pertains to inactive vertices. Otherwise, Graphite falls back to sending the original arrays of nonzero elements, which encompass actual values for active vertices and dummy values for inactive ones. When activity filtering is enabled, a SpMSpV^2 kernel only executes the received list of (index, value) pairs, skipping naturally the computations of inactive vertices. When activity filtering is disabled (i.e., when Graphite falls back to sending original arrays), a SpMSpV^2 kernel skips the computations of inactive vertices using the dummy placeholders.

Besides activity filtering, *computation filtering* is used in stationary applications [1] to skip the computation of unnecessary edges of a sparse matrix. First, computation filtering classifies vertices into four categories: 1) *regular vertices*, which are vertices with both ingoing and outgoing edges, 2) *source vertices*, which are vertices with only outgoing edges, 3) *sink vertices*, which are vertices with only ingoing edges, and 4) *isolated vertices*, which are vertices with no edges. Next, it leverages these types of vertices to avert unnecessary computations as follows: 1) regular vertices are executed in all iterations because their values are used by other vertices via the input vector, 2) source vertices are only executed in the first iteration because their values will not be changed afterwards, 3) sink vertices are only executed in the last iteration because their values are not used in earlier iterations, and 4) isolated vertices are discarded completely from the execution loop because their values are never used in any iteration. Graphite adopts computation filtering for directed graphs, only since these four types of vertices exist only in them. For undirected graphs, all vertices are regular or isolated, rendering computation filtering less effective.

8. RESULTS

8.1 Experimental Settings

8.1.1 Cluster Configuration

Experiments are conducted on a HPC cluster of 20 nodes, each with 28-core (14 cores per socket) Broadwell Processor (2.60GHz) and 192GB RAM. The cluster has Intel Omnipath interconnect (10 Gb/s speed) and all nodes are connected to an OFA network fabric. Nodes run Red Hat Enterprise Linux Server 7.6. The cluster uses Slurm workload manager for batch job queuing [49]. We use Intel MPI [26] with multithreading support for communication across machines and Pthread [36] for launching threads inside an MPI process. We utilize OpenMP [45] to collect information of allocated cores within a process, Pthread to provide CPU affinity, NUMActl [35] to enable memory affinity, and Linux `sysconf` to get the cache information at runtime.

Experiments on the cluster follow two settings. *Weak scaling* where the number of machines (and cores) used for processing is proportional to the size of the graphs and *Strong scaling* where the graph size is fixed and the number of machines (cores) is varied. At scale, we use all 20 nodes of the cluster (560 cores). Finally, any reported number is the average of multiple individual runs.

8.1.2 Counterpart Systems

Graphite[2] has been tested against two linear algebra-based systems GraphPad [2] and LA3 [1], and one graph theory-based system Gemini [69]. For all systems, we fine-tune the number of processes per machine π and the number of threads per process t, and pick the configuration that demonstrates the best runtime. Thus, we run GraphPad with $\pi = 2$ and $t = 14$, LA3 with $\pi = 14$ and $t = 2$, and Gemini with $\pi = 1$ and $t = 28$. Similarly, we run Graphite with different combinations of π and t and use $\pi = 2$ and $t = 14$ as it delivers the best results. Note that GraphPad, LA3, and Gemini crashed for some experimental settings because of limitation on memory size or number of processes.

8.1.3 Graph Datasets

Table 3 shows graphs used in the experiments including six real-world graphs (web crawls and social network from LAW [8]), and four synthesized graphs (RMAT 26 - 30 graphs from the Graph 500 challenge [13].[3]). In Table 3, the last column, *Node* reports the number of nodes used to process a graph dataset in our experiments (unless otherwise stated). Last, in order to provide the weak scaling property, starting from four nodes up to 20 (our custer size), the number of nodes are increased relative to the graph size.

8.1.4 Graph Applications

Graphite system has an extensible API supporting different graph analytics applications. We implemented PageRank (PR) for unweighted directed graphs as a prime stationary application. PR includes degree application as well. In addition, we implemented three well-known non-stationary applications including Single Source Shortest Path (SSSP) for weighted directed graphs, Breadth First Search (BFS) and Connected Component (CC) for unweighted undirected graphs. Similar to the setting used in GraphPad [2], LA3 [1], and Gemini [69] we ran PR for 20 iterations and SSSP, BFS, and CC until convergence.

Table 3: Datasets used for experiments (including web crawl, social network and synthetic types), and the number of nodes used to process them.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Type	Nodes
UK'05 (UK5) [8]	39.4 M	0.93 B	Web	4
IT'04 (IT4) [8]	41.2 M	1.15 B	Web	4
Twitter (TWT) [8]	41.6 M	1.46 B	Soc	8
GSH'15 (G15) [8]	68.6 M	1.80 B	Web	8
UK'06 (UK6) [8]	80.6 M	2.48 B	Web	16
UK Union (UKU) [8]	133 M	5.50 B	Web	20
Rmat26 (R26) [13]	67.1 M	1.07 B	Syn	4
Rmat27 (R27) [13]	134 M	2.14 B	Syn	8
Rmat28 (R28) [13]	268 M	4.29 B	Syn	16
Rmat29 (R29) [13]	536 M	8.58 B	Syn	20

8.2 Multithreading Spectrum

In this experiment, we show how the number of processes per machine π and the number of threads per process t affect the scalability of GraphPad [2] and LA3 [1] (two MPI + X systems), and Graphite (an MPI $*$ X system). Figure 7 shows the results of GraphPad, LA3, and Graphite with different configurations of π and t (x-axis), i.e., $\pi \cdot t = 28$ (total number of cores per machine) using PageRank (PR) and Connected Component (CC). For Graphite, certain observations can be made from its Double-u (W) shaped trends of Figure 7. The optimal configuration for Graphite is $\pi = 2$ and $t = 14$ where we launch one process per socket and leverage faster inter-socket communication. Also, there is a spike at runtime for $\pi = 7$ which is due to having an odd number of processes; with this configuration there is a process in each machine that has threads on both sockets, therefore stressing the QPI/UPI interconnect for shared memory communication among threads. Moreover, neither the pure multithreading ($\pi = 1$) nor the pure multi-processing ($\pi = 28$) per machine produces good results. We believe from the viewpoint of a single machine, pure multithreading imposes communication overhead on QPI/UPI for accessing input vector segments across sockets, and pure multiprocessing imposes communication overhead on QPI/UPI for inter-process communication.

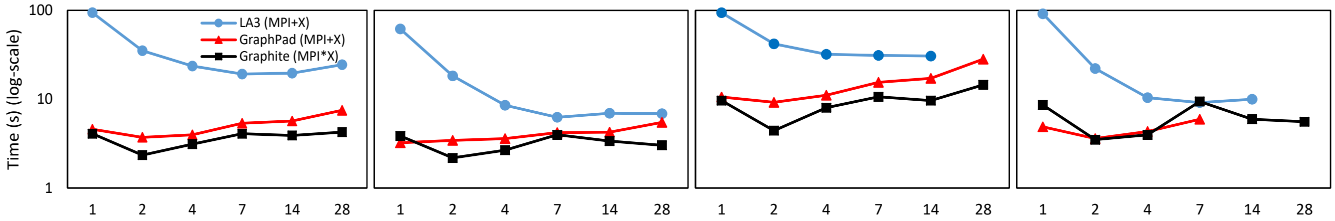
From Figure 7, GraphPad has comparable performance when launched with one or two processes per machine and its performance drops as it moves to more processes per machine (perfect multiprocessing). Also, LA3 cannot utilize threads effectively, and therefore as it utilizes more processes than threads its performance first improves (up to 14 processes per machine) and then drops (for 28 processes) which roots in LA3's poor work distribution among threads. Comparing with GraphPad and LA3, Graphite has a decent runtime difference across the majority of configurations.

8.3 Sensitivity to Different Optimizations

Graphite offers a set of features for scalable graph processing including NUMA-aware shared memory MPI communication (NUMA), *compiler optimization* (COMP-OPTI), *computation filtering* (CMPT-FLTR), and *activity filtering* (ACTY-FLTR). From Figure 8a, on PageRank (PR) (a stationary application), NUMA, compiler optimization, com-

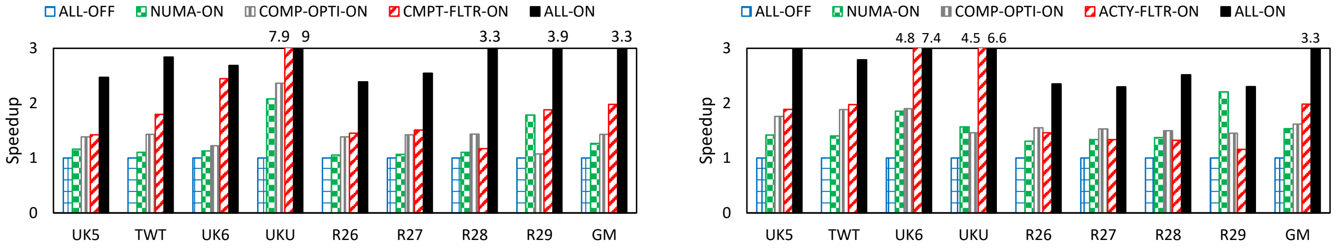
[2] Graphite's source code: https://github.com/hmofrad/Graphite

[3] RMATs follow power of two growth rate, i.e., RMATn has 2^n vertices and 2^{n+4} edges.



(a) PR on TWT (8 machines) **(b)** CC on TWT **(c)** PR on R28 (16 machines) **(d)** CC on R28

Figure 7: Runtime of Graphite and others with (π, t) = (1, 28), (2, 14), (4, 7), (7, 4), (14, 2), and (28, 1).



(a) PR **(b)** CC

Figure 8: Normalized speedup (weak scaling) of NUMA, COMP-OPTI, CMPT-FLTR, and ACTY-FLTR with ALL-OFF and ALL-ON as baseline and headline. GM is the grand geometric mean.

putation filtering, and the combination of these features give 27%, 43%, 2×, and 3.3× speedups, respectively. From this figure, smaller graphs benefit more from compiler optimization whereas larger graphs benefit more from NUMA and computation filtering. Also, from Figure 8b, on Connected Component (CC) (a non-stationary application), NUMA, compiler optimization, activity filtering, and the combination of them give 53%, 62%, 2×, and 3.3× speedups, respectively. From this figure, NUMA and compiler optimization are more effective in synthetic graphs (which has uniform distribution with constant edge factor), whereas activity filtering is more effective in real-world graphs (which follows power-law distribution with high variance in number of edges per vertex). From the last bars of each dataset of Figure 8, enabling all features results in a better speedup which shows the effect of these features are cumulative.

Figure 8 shows that **NUMA** is more effective for larger graphs which stems from leveraging memory and processor architecture to maximize the usage of MPI shared memory transport. Also, enabling the **compiler optimization** to its fullest extent is vital for running an iterative compute-intensive SpMSpV^2 kernel, because this kernel includes the bulk of computation done by threads, and any optimization that can slightly improve on this kernel, will largely improve the overall runtime. Finally, the **computation filtering** advantage comes from passing over the computation of subsets of unnecessary vertices in stationary applications like PR, and the **activity filtering** advantage comes from skipping the communication and computation of inactive vertices in non-stationary applications like CC.

8.4 Execution Time Analysis

Graphite's matrix computing model iterates over Broadcast, Combine, and Apply operations. In addition, Graphite checks for convergence and enforces synchronization among threads at the end of each iteration. Figure 9 shows the breakdown of execution time of 20 iterations of PR on R28. It is clear that Broadcast and Combine operations are both computation and communication intensive, and constitutes about 90% of the runtime. **Broadcast time** (46%) consists of the time for preparing the new input segments (Bcast-comp), plus the overlapped communication time (Bcast-comm). **Combine time** (41%) constitutes the time spent for running the SpMSpV^2 (Combine-SpMSpV^2+comm), and the accumulation time of the partial output segments which is overlapped with background communication (Combine-accu+comm). **Apply time** is the time for interpolating and updating the values of the vertices. Note Combine-accu+comm and Apply times are roughly equal as both are operating on similar vector segments. Finally, **Convergence time** represents the total synchronization time of threads at the end of each iteration which also includes the time for checking the convergence.

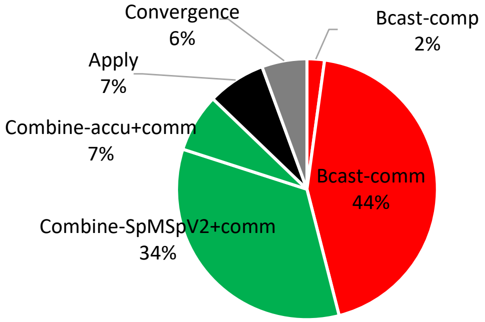


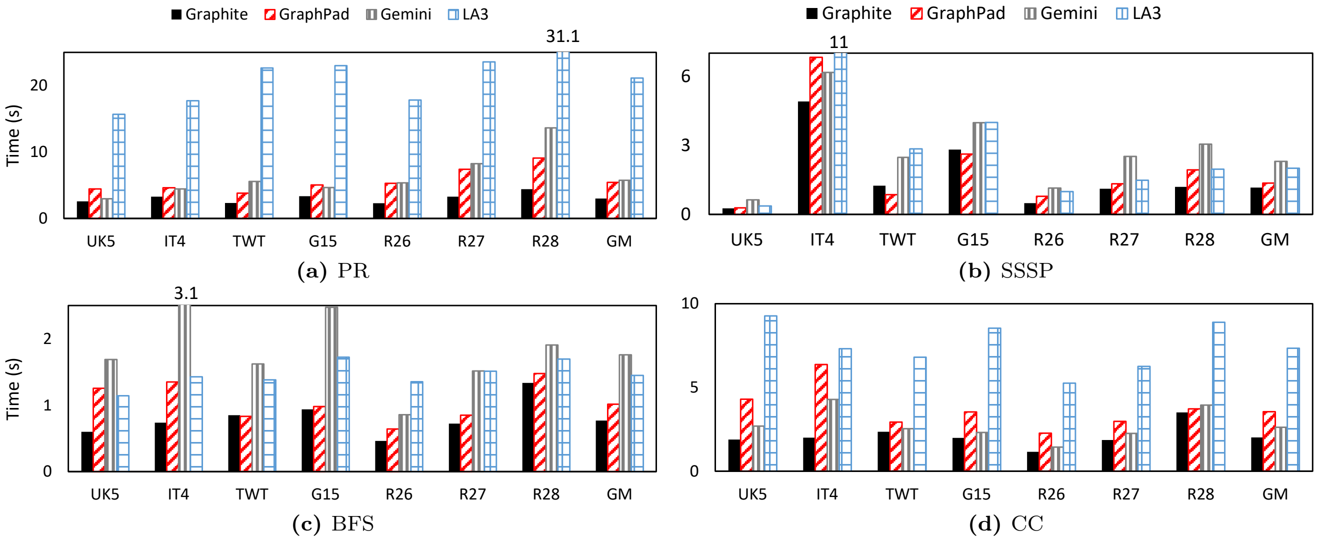
Figure 9: Graphite Execution time breakdown in seconds from running PR on R28 using 16 machines.

8.5 Comparisons with other Systems

8.5.1 Weak Scaling Comparison

Weak scaling of Graphite versus GraphPad, Gemini, and LA3 are reported in Figure 10. Based on the grand geometric mean of results (geometric mean of geometric mean of each subfigure), Graphite is 2.9×, 60%, 80%, and 2.1× faster than these systems in PR, SSSP, BFS, and CC.

From Figure 10a, in **PageRank (PR)**, Graphite is on average (geometric mean) 81%, 91%, and 7.1× faster than GraphPad, Gemini, and LA3. PR is a computation- and



(a) PR

(b) SSSP

(c) BFS

(d) CC

Figure 10: Runtime of Graphite and others (weak Scaling). GM is the grand geometric mean over all datasets.

communication-intensive non-stationary application which needs to visit all vertices and their associated edges in order to rank them. For PR, computation filtering helps Graphite to skip the computation of subsets of vertices[4]. Also, compared to others, LA3 does not perform good on PR because it has rigorous communication optimizations which are not effective in an HPC cluster with fast interconnect.

Having a look at Figure 10b, Graphite is 18%, 2×, and 73% faster than GraphPad, Gemini, and LA3 on average in **Single Source Shortest Path (SSSP)**. Running SSSP on a directed graph, the source vertex is an important factor regardless of the size of graph. Starting from the source, SSSP traverses all vertices connected to the source with an incoming link from the source. So, if source is sampled from a small connected component, all vertices of that component will be visited quickly and that is why for some graphs like UK5 or R26 the runtime is small compared to other graphs. Graphite performs the best in SSSP except for TWT because the complex structure of the largest component of TWT causes a huge load imbalance among threads[5]. In addition, GraphPad outperforms Gemini and LA3 because of its better communication and compression optimizations.

On BFS (Figure 10c), Graphite outperforms GraphPad, Gemini, and LA3 with 33%, 2.3×, and 90% better runtime on average. Given **Breadth First Search (BFS)** uses undirected graphs, unlike SSSP, it eventually visits all vertices of the connected component where the source is chosen from. Therefore, BFS deals with more communication and computation than SSSP. Gemini is relatively slow because it does not have a good communication strategy and relies on a single thread per process to communicate messages of a row of tiles which works fine only for small number of nodes e.g. 8 nodes. LA3's communication optimizations work better in BFS (and SSSP) because communication pattern of BFS (and SSSP) include(s) small bursts of data transfer which can quickly be compressed per destination process in LA3.

As shown in Figure 10d, on average Graphite performs 77%, 31%, and 3.7× faster than GraphPad, Gemini, and LA3 for **Connected Component (CC)**. CC tries to find a set of vertices that are connected to each other by paths (a strongly connected subgraph) and accomplishes this task by iteratively visiting all vertices inside components. Gemini outperforms GraphPad and LA3 because it uses NUMA-aware partitioning which offers faster local memory access and higher cache utilization. Both GraphPad and Gemini use a pair of dense vectors accompanied by a bitvector for fast random access of compressed vectors. However, GraphPad is slower than Gemini in CC because for this application Gemini can effectively switches between its sparse and dense representations using its push/pull model, whereas, GraphPad compression threshold is ineffective here. On the other hand, Graphite's decision for switching between sparse and dense representations is made in Broadcast operation and reused in Combine operation. Although this approach poses a small overhead to the Broadcast operation, altogether it results in a better performance for CC as it skips the computation of activities in the Combine operation.

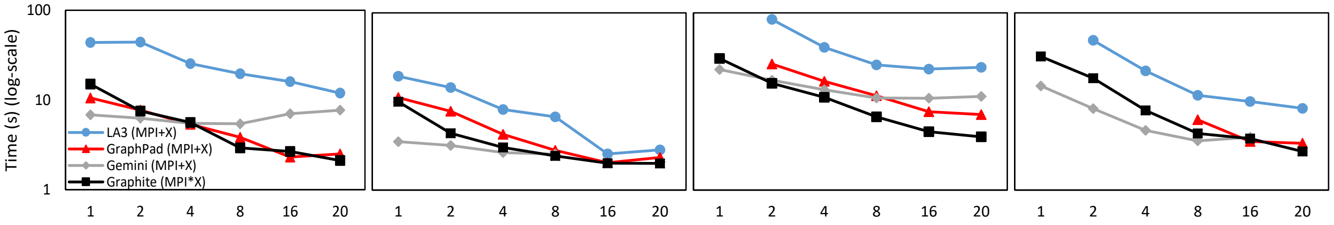
From Figure 10, Graphite outperforms GraphPad, Gemini, and LA3 systems, where this outperformance is largely due to the usage of MPI $*$ X parallelism model and 2D-thread-based partitioning and placement. Conversely, other systems follow MPI + X parallelism and process-based partitioning that underperform in iterative applications. For example, GraphPad and Gemini use process-based 2D-Cyclic and 1D-Row placements, which are less scalable than thread-based 2D-Staggered placement used in Graphite.

8.5.2 *Strong Cluster Scaling Comparison*

Figure 11 shows the runtime of different systems for different number of machines for PR and CC on TWT and R28. Overall, Graphite scales very well on both TWT (real-world) and R28 (synthetic). It can effectively leverage the added processing power and improve the runtime. This scalability is highly due to MPI $*$ X parallelism model which balances the computation and communication of tiles among threads. Moreover, GraphPad which follows the MPI + X parallelism model exhibits comparable scalability on TWT and poorer scalability on R28. Next, Gemini starts with a good performance, but fails to scale for larger clusters due to the limitations of MPI + X parallelism, e.g., only MPI processes carry out communication. Last, LA3 does not scale well as its communication strategy is not suitable for HPC clusters.

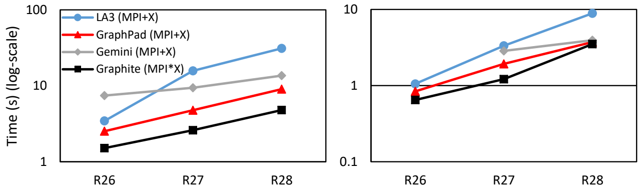
[4] In R29, computation filtering skips 5% of SpMSpV^2 ops.

[5] The largest component of TWT includes 80% of its edges.



(a) PR on TWT **(b)** CC on TWT **(c)** PR on R28 **(d)** CC on R28

Figure 11: Strong cluster scaling of Graphite and other systems on R28. X-axis is the number of machines.



(a) PR **(b)** CC

Figure 12: Strong data scaling (R26-28 with 16 nodes).

8.5.3 *Strong Data Scaling Comparison*

Figure 12 shows the runtimes of different systems on R26 - R28 using 16 machines for PR and CC. From Figure 12a, Graphite, which follows MPI $*$ X exhibits a better data scaling with minimal changes in the runtime for PR. In this figure, Graphite outperforms GraphPad, Gemini and LA3 which all follow MPI + X. Additionally, a similar trend for CC can also be seen in Figure 12b.

8.5.4 *Discussion of Evaluated Systems*

Table 4 thoroughly reports different features of the studied systems. From this table, GraphPad [2], Gemini [69], and LA3 [1] use MPI + X model with processes being the basic units of computation and communication, whereas Graphite uses MPI $*$ X model where threads are the basic units of computation and communication. GraphPad, Gemini, and LA3 use process-based 2D-Staggered, 2D-Cyclic, and 1D-Row placements, whereas Graphite uses 2DT-Staggered that is devised for threads. Moreover, although all these systems utilize GAS-like computing models, Graphite carefully incorporates asynchronous collective MPI primitives in its model enabling faster communication. Also, Graphite leverages NUMA for both computation (CPU/memory affinity) and communication (MPI shared memory communication) purposes, whereas Gemini internally supports memory affinity but relies on OpenMP for processor affinity. In addition, Graphite carefully follows strict programming guidelines to completely enable compiler optimizations for multithreaded SpMSpV^2 kernels. Last, all systems use activity filtering, but only Graphite and LA3 use computation filtering.

The performance difference between Graphite and LA3 is due to three design decisions made in LA3: (1) Communication strategy: LA3 is designed for *cloud environments* (not HPC) with low-bandwidth and high-latency interconnection networks. It has an extensive communication optimization that tailors input messages per tile to reduce the communication volume at the expense of more computation overhead. In a cloud environment, this strategy works well because the communication delay is more expensive than the time spent for constructing the optimized messages. In contrast, in an HPC environment, this strategy is not productive because of fast interconnects. (2) Parallelism model: LA3 follows the MPI + X model. It relies on OpenMP runtime to distribute the computation of tiles across threads while bounding the communication to only MPI processes. Thus, compared to an MPI $*$ X system like Graphite, LA3 has less MPI communication endpoints and larger tiles, which reduces the overlapping of computation with communication. (3) Matrix compression: LA3 uses Doubly Compressed Sparse Column (DCSC) [10], whereas Graphite uses Triply Compressed Sparse Column (TCSC) [43], which is more cache friendly.

Table 4: Summary of features of the studied systems (parallelism model (PARA-MDL), compiler optimization (COMP-OPTI), computation filtering (CMPT-FLTR), and activity filtering (ACTY-FLTR). GP stands for GraphPad. S and C in 2DT-S and 2D-C stand for staggered and cyclic.

#	Feature	Graphite	GP [2]	LA3 [1]	Gemini [69]
1	Model	MPI$*$X	MP+X	MP+X	MP+X
2	Unit	Thread	Process	Process	Process
3	Tiling	2DT-S	2D-C	2D-S	1D-Row
4	PARA-MDL	GAS	GAS	GAS	Push/pull
5	NUMA	Full	No	No	Mem
6	COMP-OPTI	Targeted	Default	Default	Default
7	CMPT-FLTR	Yes	No	Yes	No
8	ACTY-FLTR	Yes	Yes	Yes	Yes

9. CONCLUSIONS

In this paper, we introduced Graphite, a new linear algebra based graph analytics system that uses the MPI $*$ X parallelism model with 2D-thread-based partitioning and placement. In Graphite, threads are treated as first-class citizens of a distributed system where computation and communication are fairly distributed among all threads while minimizing the synchronization points. Graphite utilizes a GAS-like matrix computing model for fast execution of iterative analytics that takes advantage of MPI and distributed shared memory capabilities. It exploits NUMA for both computation (CPU/memory affinity) and communication (MPI shared memory communication). Compared against GraphPad, Gemini and LA3 analytics systems, the proposed Graphite achieves a speedup of roughly up to 3× due to its thread-level asynchronous communication and computation, high degree of concurrent communications, and NUMA-ware computation and communication.

10. ACKNOWLEDGMENTS

This publication was made possible by National Science Foundation Grant Number CCF-1725657, and NPRP grant #7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). This research was supported in part by the University of Pittsburgh Center for Research Computing through the resources provided. Finally, we thank the VLDB2020 reviewers for their valuable comments.

11. REFERENCES

[1] Y. Ahmad, O. Khattab, A. Malik, A. Musleh, M. Hammoud, M. Kutlu, M. Shehata, and T. Elsayed. La3: a scalable link-and locality-aware linear algebra-based graph analytics system. *PVLDB*, 11(8):920–933, 2018.

[2] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 313–322. IEEE, 2016.

[3] V. Balaji and B. Lucia. Combining data duplication and graph reordering to accelerate parallel graph processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 133–144. ACM, 2019.

[4] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. T. Pedretti. Toward an evolutionary task parallel integrated mpi+ x programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 30–39. ACM, 2015.

[5] S. Beamer, K. Asanovic, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *2015 IEEE International Symposium on Workload Characterization*, pages 56–65. IEEE, 2015.

[6] S. Beamer, K. Asanović, and D. Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831. IEEE, 2017.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In *ACM Sigplan Notices*, volume 40, pages 261–268. ACM, 2005.

[8] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *13th ACM WWW*, pages 595–601, 2004.

[9] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[10] A. Buluc and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[11] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[12] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the graphblas api for c. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652. IEEE, 2017.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[14] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[16] R. Dathathri, G. Gill, L. Hoang, and K. Pingali. Phoenix: A substrate for resilient distributed graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 615–630. ACM, 2019.

[17] T. Davis. Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra. *Submitted to ACM TOMS*, 2018.

[18] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304. ACM, 2017.

[19] H. Fu, M. G. Venkata, S. Salman, N. Imam, and W. Yu. Shmemgraph: efficient and balanced graph processing using one-sided communication. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 513–522. IEEE, 2018.

[20] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner. Graphulo: Linear algebra graph kernels for nosql databases. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 822–830. IEEE, 2015.

[21] I. M. Gessel and C. Reutenauer. Counting permutations with given cycle structure and descent set. *Journal of Combinatorial Theory, Series A*, 64(2):189–215, 1993.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2. Usenix, 2012.

[23] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, Oct. 2014. USENIX Association.

[24] S. Grossman, H. Litz, and C. Kozyrakis. Making pull-based graph processing performant. In *ACM SIGPLAN Notices*, volume 53, pages 246–260. ACM, 2018.

[25] M. Han and K. Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[26] Intel. Intel mpi library. https://software.intel.com/en-us/mpi-library.

[27] Y.-Y. Jo, M.-H. Jang, S.-W. Kim, and S. Park. Realgraph: a graph engine leveraging the power-law distribution of real-world graphs. In *The World Wide Web Conference*, pages 807–817. ACM, 2019.

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system

implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, pages 229–238. Washington, DC, USA, 2009.

[29] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[30] V. Kiriansky, Y. Zhang, and S. Amarasinghe. Optimizing indirect memory references with milk. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 299–312. IEEE, 2016.

[31] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[32] D. Li, Y. Zhang, J. Wang, and K.-L. Tan. Topox: topology refactorization for efficient graph partitioning and processing. *PVLDB*, 12(8):891–905, 2019.

[33] S. Li, T. Hoefler, and M. Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.

[34] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu, et al. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 56. IEEE Press, 2018.

[35] Linux. Numa - numa policy library. http://man7.org/linux/man-pages/man3/numa.3.html.

[36] Linux. Posix thread (pthread) library. http://man7.org/linux/man-pages/man7/pthreads.7.html.

[37] H. Liu and H. H. Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, Feb. 2017. USENIX Association.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[39] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[40] A. Lugowski, A. Buluç, J. R. Gilbert, and S. Reinhardt. Scalable complex graph analysis with the knowledge discovery toolbox. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5345–5348. IEEE, 2012.

[41] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[43] M. H. Mofrad, R. Melhem, Y. Ahamd, and M. Hammoud. Efficient distributed graph analytics using triply compressed sparse format. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[44] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[45] OpenMP. The openmp api specification for parallel programming. https://www.openmp.org/.

[46] OpenMPI. Open mpi: Open source high performance computing. https://www.open-mpi.org/.

[47] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The tiledb array data storage manager. *PVLDB*, 10(4):349–360, 2016.

[48] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[49] Schedmd. Slurm workload manager. https://slurm.schedmd.com/.

[50] Z. Shang, J. X. Yu, and Z. Zhang. Tufast: A lightweight parallelization library for graph analytics. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 710–721. IEEE, 2019.

[51] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146. ACM, 2013.

[52] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. Mt-mpi: multithreaded mpi for many-core environments. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 125–134. ACM, 2014.

[53] A. Stamatakis and M. Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with mpi, pthreads, and openmp: A performance study. In *IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 424–435. Springer, 2008.

[54] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[55] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54, 2013.

[56] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[57] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan. Difftrace: Efficient whole-program trace analysis and diffing for debugging. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[58] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher. Parlot: Efficient whole-program call tracing for hpc applications. In *Programming and Performance Visualization Tools*, pages 162–184. Springer, 2017.

[59] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff. Mpi at exascale. *Procceedings of SciDAC*, 2:14–35, 2010.

[60] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 197–210. ACM, 2013.

[61] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *ACM SIGPLAN Notices*, volume 50, pages 194–204. ACM, 2015.

[62] C. Xu, K. Vora, and R. Gupta. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 587–600. ACM, 2019.

[63] C. Yang, A. Buluc, and J. D. Owens. Implementing push-pull efficiently in graphblas. *arXiv preprint arXiv:1804.03327*, 2018.

[64] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. *ACM SIGPLAN Notices*, 50(8):183–193, 2015.

[65] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan. Gbtl-cuda: Graph algorithms and primitives for gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–920. IEEE, 2016.

[66] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[67] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121, 2018.

[68] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, Feb. 2015. USENIX Association.

[69] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, Savannah, GA, Nov. 2016. USENIX Association.

[70] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.