PaHiS: A Hierarchical Synchronous Parallel Model for Irregular Workloads

Petros Anastasiadis
petros.anastasiadis@h-partners.com
Computing Systems Laboratory,
Huawei Technologies
Zurich, Switzerland

Denis Jelovina
denis.jelovina@huawei.com
Computing Systems Laboratory,
Huawei Technologies
Zurich, Switzerland

Albert-Jan N. Yzelman
albertjan.yzelman@huawei.com
Computing Systems Laboratory,
Huawei Technologies
Zurich, Switzerland

Abstract

Large-scale solvers are critical in scientific and engineering domains, where achieving portable and near-optimal performance remains a challenge. Existing high-level parallel performance models provide coarse-grain communication cost estimates but fail to capture the hierarchical memory behavior and irregular access patterns prevalent in modern sparse solver workloads. Conversely, sparse modeling techniques focus on low-level kernel details and lack support for solver-level reasoning. To bridge this gap, we introduce PaHiS, a hierarchical cost model that represents hardware and algorithms through a multi-level abstraction of their communication, computation, and synchronization characteristics. We integrate this model into a GraphBLAS library and couple it with an automated microbenchmarking pipeline for hardware characterization, enabling solver cost estimation and thread autotuning for algorithms expressed in GraphBLAS. We evaluate the model on three CPU architectures for two solvers, demonstrating i) strong correlation between predicted and measured performance, ii) high-performance thread-level autotuning in ALP GraphBLAS, and iii) reliable guidance for high-level HW–SW co-design decisions such as hardware selection and algorithm comparison.

CCS Concepts

• **Mathematics of computing** → Solvers; *Mathematical software performance*; • **Computing methodologies** → **Modeling methodologies**; **Model verification and validation**.

Keywords

Performance modeling; Bulk-synchronous models; Autotuning; Sparse linear algebra; GraphBLAS;

ACM Reference Format:
Petros Anastasiadis, Denis Jelovina, and Albert-Jan N. Yzelman. 2026. *PaHiS*: A Hierarchical Synchronous Parallel Model for Irregular Workloads. In *38th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '26), July 6–10, 2026, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3816782.3819197

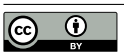
1 Introduction

Large-scale solvers are the core component of many scientific, engineering, and data analysis workloads, where performance and scalability are critical. Commonly, domain experts widely use high-level solver libraries to express their numerical algorithms, to avoid engaging in low-level performance engineering [6, 8, 15, 33, 36, 37, 50]. But, performance is still critical: solvers should achieve near-optimal and scalable performance across systems while being written in high-level abstractions. At the same time, domain experts often need to make algorithmic choices, such as alternative formulations, assumptions, or transformations, and understand how these choices affect end-to-end performance. They also need to assess whether each solver is a good fit for a target architecture, or conversely, which architectures are suitable for a specific class of algorithms. Consequently, addressing these needs requires a mechanism for high-level performance analysis and hardware–software co-design; a bridge between algorithms (software) and hardware.

This idea of high-level parallel performance modeling is not new; there is extensive prior work in this direction from the early days of scientific computing [1, 13, 19, 47]. However, as large-scale solvers and HPC workloads have evolved, performance has increasingly shifted from computation to communication and memory latency, driven by the memory wall and the growing scale of modern clusters. This shift has motivated the adoption of models, the most notable being the BSP family [39, 41, 45, 46], which focus on bandwidth and latency costs while maintaining a high-level, algorithm-centric view of performance. These models are effective at exposing coarse-grain communication costs and supporting high-level reasoning about parallel execution, but remain *too coarse* to capture deep hierarchical memory behavior paired with irregular data movement.

At the same time, many modern solver workloads are dominated by sparse linear algebra, which is fundamentally different from dense: it is memory-bound, highly sensitive to input structure, and dominated by irregular access patterns [38]. Paired with the deep, hierarchical logic and the variability of modern architectures, modeling the behavior of these solvers requires greater detail than what coarse-grain parallel models provide. Additionally, modeling sparse linear algebra is prohibitively complex: performance depends on many interacting factors, such as sparsity structure, workload balance, data layout, and hardware characteristics; as a result, most existing modeling and optimization approaches target individual kernels (e.g., SpMV), are very low-level, and target very specific architectures or problem characteristics. As a result, existing approaches leave a gap between parallel performance models that are too high-level to capture sparse behavior and sparse modeling techniques that are too low-level to support solver-level reasoning.

In this work, we target an intermediate level of abstraction between coarse-grain parallel performance models and low-level sparse kernel analysis. We develop a hierarchical, system-oriented



SPAA '26, London, United Kingdom

ACM ISBN 979-8-4007-2761-0/26/07
https://doi.org/10.1145/3816782.3819197

performance model based on the BSP family that retains a high-level, algorithm-centric view while capturing the dominant effects of bandwidth, latency, and irregular memory access. Rather than requiring algorithm reformulation or domain-expert annotations, our approach relies on an automated procedure to instantiate the model for different *system configurations*, enabling direct cost estimation for solver algorithms expressed in existing high-level frameworks. These resulting cost estimates efficiently distinguish between system configurations, which in turn can be used for solver-level decisions such as algorithm selection, or configuration choice like system selection and thread autotuning.

In summary, we make the following contributions:

(1) A hierarchical synchronous parallel model able to capture bandwidth, latency, and irregular access effects across different system configurations.
(2) An automated microbenchmark suite that instantiates model coefficients empirically for different thread counts and configurations.
(3) A generic algorithm–hardware mapping logic that enables cost prediction for any solver by expressing it as sequences of primitives.
(4) Integration of the model into an ALP/GraphBLAS[50] backend for automated cost estimation, system configuration comparison, and thread-level autotuning for any solver written on GraphBLAS.

2 Background

High-level performance models have been widely used in HPC to reason about scalability, portability, and performance bottlenecks without resorting to low-level, machine-specific analysis. Early hardware performance models primarily focused on computation throughput, with the most common being the PRAM family [19]. However, as multicore and multinode systems became prevalent, data movement [1] increasingly emerged as the dominant performance bottleneck, leading to the development of a new family of data movement-aware models, with the most well-established being the Roofline model [47], the classic alpha-beta model[25] and LogP [13]. Additionally, to accommodate different communication and systems characteristics, these later branched into multiple specializations, including LogGP [1, 27], Parameterized LogP [32], LogGPS [29], LogGOPS [26], LogGPO [11], and others [3, 34]. In parallel with this communication-focused line of work, to model deeper intra-node memory hierarchies, hierarchical memory models for algorithm design and asymptotic complexity analysis were also developed [2, 7, 12, 20, 30], modeling parallel computation as data movement across multi-level cache hierarchies and characterizing algorithms via their I/O complexity and cache utilization. Nevertheless, all of the above models primarily abstract algorithm characteristics, either to identify compute, bandwidth, or latency bottlenecks, or to support asymptotic complexity analysis; they are not designed to serve as an explicit algorithm–hardware bridge[5].

[1]To avoid confusion between bytes sent/received (communication) and bytes read/written (access), we use the term 'data movement' for either

2.1 The BSP family of models

The *Bulk Synchronous Parallel (BSP)* model was the first explicit algorithm–hardware bridging model that captured both computation and communication costs while retaining an algorithm-centric view of execution [24, 45, 49]. In BSP, a parallel computation proceeds as a sequence of *supersteps*, each consisting of three ordered phases: (i) local computation on private data, (ii) communication of data between processors, and (iii) a global synchronization barrier. A BSP machine is characterized by the parameters (p, r, g, ls), where p is the number of processors, r their local computation rate, g the cost per communicated data unit (inverse bandwidth), and ls the synchronization latency. The execution time of a superstep i is modeled as

$$T_i = \max_j (ops_{i,j} \cdot r, g \cdot h_i + ls), \tag{1}$$

where $ops_{i,j}$ is the local work performed by processor j and h_i is the maximum volume of data sent or received by any processor during the communication phase. Summing over all supersteps yields a cost estimation for a parallel algorithm, abstracting away low-level scheduling details from its dominant performance characteristics. This abstraction can be used as the foundation for portable parallel algorithm design, analysis, and performance prediction[5, 45].

Since the original BSP model is primarily designed for multi-node algorithmic analysis, it assumes a flat machine with uniform communication costs, and therefore cannot accurately model deep CPU hierarchies or complex communication patterns. To address this limitation, numerous extensions of BSP have been proposed, each targeting specific communication, architectural, and algorithmic characteristics [4, 10, 21, 31, 40, 43, 44, 46, 48, 51]. The most relevant to our case are *NUMA-aware BSP* models [39, 41, 42], which distinguish between local and remote data movement costs, and the most general and widely used of the hierarchical approaches, *Multi-BSP*c. *Multi-BSP* models a system as a hierarchy of nested BSP components, corresponding to architectural levels (cores, caches, NUMA domains, sockets, and nodes). A Multi-BSP machine is described by a sequence of parameter tuples

$$(p_1, g_1, ls_1, m_1), (p_2, g_2, ls_2, m_2), \ldots, (p_d, g_d, ls_d, m_d),$$

where each level l consists of p_l subcomponents, with data movement cost g_l, synchronization latency ls_l, and local memory capacity m_l. Computation always occurs at the lowest level, while communication and data movement may occur recursively across all levels and must be explicitly defined during algorithm design. In summary, Multi-BSP provides increased modeling accuracy while preserving BSP's bulk-synchronous execution logic, at the cost of a larger parameter space and more complex algorithmic design.

2.2 The case for irregular workloads

Many modern scientific and data-analytic workloads are dominated by sparse linear algebra, including sparse matrix–vector multiplication (SpMV), sparse triangular solves, and graph-based traversals. These problems are inherently memory-bound, and their performance is highly sensitive to sparsity structure, which induces data movement irregularity and imbalance [28, 35, 38]. While there is extensive research on sparse modeling and tuning [17, 18, 22, 23, 28, 35, 38, 52], it typically focuses on low-level, architecture- or problem-specific characteristics and does not map

Table 1: PAHiS hardware parameters.

Parameter	Description
d	Number of hierarchy levels
$r = [r^{\text{scalar}}, r^{\text{vec}}]$	Processing rates for $l = 0$ (scalar and vector)
For $[l = 0 \rightarrow d]$:	
g_l	Inverse bandwidth (seconds per byte)
ls_l	Latency (seconds)
m_l	Available memory
p_l	Number of sub-components for this level
P_l	Total p under level = $p_0 \cdot p_1 \cdot \ldots \cdot p_l$

to BSP's bulk data movement paradigm. On top of this, iterative sparse solvers consist of sequences of both dense and sparse operations, further complicating synchronization, data reuse, and locality properties. Combined with deep memory hierarchies, which complicate data movement level and granularity, sparse solvers do not translate well to Multi-BSP.

In summary, existing high-level performance models fall short of capturing the combined complexity of modern architectures and irregular workloads. Simple performance models, such as Roofline and the LogP family, provide useful first-order insights but are too coarse to model deep hardware hierarchies and, even in their hierarchical variants, do not serve as explicit algorithm–hardware bridges. BSP-based models, on the other hand, address this with their algorithmic formulations. However, they do not accommodate sparse kernels with irregular, structure-dependent data movement patterns, especially for hierarchical models like Multi-BSP. At the other extreme, existing sparse modeling and optimization approaches capture fine-grained behavior in detail but are highly problem- and architecture-specific, and therefore unsuitable as portable, high-level performance models. This motivates the need for a modeling approach that retains the algorithm-centric structure of BSP models, captures hierarchical data movement effects, and can express irregular data movement with relative accuracy.

3 The PAHiS model

This section introduces PAHiS, our proposed extension of bulk parallel synchronous models to capture irregular data movement. It formalizes the model, demonstrates its expressiveness through common algorithmic examples, and concludes with our proposed automated system-level initialization procedure. For the hardware model part, we use a similar notation with Multi-BSP as shown in Table 1. We note that L is renamed to ls to avoid confusion with level l, and we henceforth use zero-based notation for all parameters.

3.1 Introduction of the *stream* notation

First, we extend BSP-like notation to distinguish between regular and irregular data movement. To do this, we split the data movement happening within a superstep to *streams*. As *stream*, we define a *conceptual concurrent construct* that, during a superstep i, performs data movement from/to worker s, of $h_{i,s,k}$ consecutive bytes. In practice, the streams of each superstep ($kn_{i,s}$) express an extra level of detail for its bulk communication phase. Any read/write access to a continuous memory region (for intra-node) or send/receive (for inter-node) is considered to create a stream. For example, accessing

Table 2: Algorithm parameters used in the proposed model.

Parameter	Description
n	Number of supersteps
For $[i = 0 \rightarrow n]$:	
For $[s = 0 \rightarrow P_d]$:	
$w_{i,s} = [w_{i,s}^{\text{scalar}}, w_{i,s}^{\text{vec}}]$	Local work on s for superstep i
l_i	Communication level for i
For $[s = 0 \rightarrow p_{l_i}]$:	
$kn_{i,s}$	Number of streams
For $[k = 0 \rightarrow kn_{i,s}]$:	
$h_{i,s,k}$	Size of the k-th communication

8 consecutive elements of a double vector ($8B/elem$) in a superstep is a single stream ($h = 64B$), accessing only the first is a stream ($h = 8B$), and sending or receiving using a 500 MB serial buffer to another node is also a stream ($h = 500MB$). On the other hand, non-consecutive communication during a superstep creates *an equal amount of stream actions*. I.e., if the 8 elements of the aforementioned double vector were accessed irregularly within a superstep, that would create 8 streams of $h = 8B$.

Table 2 shows the algorithmic notation of the proposed PaHiS model. Relative to Multi-BSP, the definitions of supersteps, hierarchy levels, and local computation w remain unchanged, while send/receive parameters (t, v) are replaced by the stream-based formulation using kn and h_i. For simplicity, the notation abstracts away communication direction (read/write or send/receive) and instead treats all data movement during each superstep as *traffic* at a single hardware level. Consequently, if an algorithm contains communication that happens at various levels (tree reduce, tiling etc), these must be defined as different supersteps.

3.2 The *PAHiS* cost model

We model the execution cost of a parallel algorithm by decomposing each superstep into computation and communication components. The computation cost of superstep i is defined as the maximum local processing time across all sub-components:

$$c_i^{\text{comp}} = \max_{s=0}^{P_d} \left(\max \left(\frac{w_{i,s}^{\text{scalar}}}{r^{\text{scalar}}}, \frac{w_{i,s}^{\text{vec}}}{r^{\text{vec}}} \right) \right), \tag{2}$$

where $w_{i,s}^{\text{scalar}}$ and $w_{i,s}^{\text{vec}}$ denote the number of scalar/vector operations performed by sub-component s during superstep i, while r^{scalar} and r^{vec} are the corresponding processing rates. The cost equation evaluates to the cost of the slowest sub-component, similar to BSP/Multi-BSP.

Similarly, the communication cost of superstep i, which performs communication at a hardware hierarchy level l_i with inverse bandwidth g_{l_i} and latency ls_{l_i}, is modeled as:

$$c_i^{\text{comm}} = \max_{s=0}^{p_{l_i}} \left(\sum_{k=0}^{kn_{i,s}} \left(h_{i,s,k} \cdot g_{l_i} + ls_{l_i} \right) \right), \tag{3}$$

Algorithm 1: SpMV (COO format): Compute $y := Ax$

Model Data: Total workers P_d, worker id s
Data: Sparse matrix A in COO $\{row, col, val\}[nz]$, input vector $x[n]$, in/output vector $y[m]$, dtype size b
$nz_s \leftarrow nz \ / \ P_d$
for $i \leftarrow s \cdot nz_s$ **to** $(s+1) \cdot nz_s$ **do**
  $y[row[i]] \leftarrow y[row[i]] + val[i] \cdot x[col[i]]$
Algo Parameters: atomic $\rightarrow$ sync $\rightarrow$ nz_s supersteps
- $w_{i,s}^{scalar} = 2$
- $kn_{i,s} = 5, h_{i,s,k_{\{0-3\}}} = b, h_{i,s,k_4} = 2 \cdot b$

Cost Model:
- $comp_i = \max_{s=0}^{P_d}(w_{i,s}^{scalar} \ / \ r^{scalar}) = 2 \ / \ r^{scalar}$
- $comm_i = \max_{s=0}^{p_{l_i}} \left(\sum_{k=0}^{4} \left(b \cdot g_{l_i} + ls_{l_i}\right)\right) + 2 \cdot b \cdot g_{l_i} + ls_{l_i} = 6 \cdot g_{l_i} \cdot b + 5 \cdot ls_{l_i}$
- $T = \sum_{i=0}^{nz_s} \left(\max(2 \ / \ r^{scalar}, 6 \cdot g_{l_i} \cdot b + 5 \cdot ls_{l_i})\right)$

where $kn_{i,s}$ denotes the number of streams for sub-component s during superstep i and $h_{i,s,k}$ are the sizes of each of these data movements. Finally, the total cost of a superstep is defined as the maximum between its computation time and communication time, and the overall cost of the algorithm is then the sum across all supersteps:

$$T = \sum_{i=0}^{n} \max\left(c_i^{\text{comp}}, c_i^{\text{comm}}\right) \tag{4}$$

3.2.1 Example: SpMV (COO). To demonstrate the ability to model latency-bound kernels, Algorithm 1 shows an example application on the sparse matrix-vector multiplication (SpMV) kernel using the COO storage format. The sparse matrix is distributed across P_d workers by partitioning the nonzero entries, with each worker s processing a contiguous block of $nz_s = nz \ / \ P_d$ nonzeros. Each nonzero update performs a scalar multiply and add, expressed as scalar work $w_{i,s}^{\text{scalar}} = 2$. We assume a simple implementation where the updates on the output vector $y[row[i]]$ can overlap between threads and thus require atomic access. Due to this implicit synchronization, each loop iteration is modeled as an individual superstep, resulting in nz_s supersteps per worker. For each superstep, communication is modeled at level l_i (more in sec. 3.3) using six streams of size $h_{i,s,k} = b$, corresponding to accesses to the `row`, `col`, and `val` arrays, the read of $x[col[i]]$, and the read and write on $y[row[i]]$. The time of each superstep is determined by the maximum of the computation and communication costs across workers, and the total execution cost T is obtained by summing this cost over all nz_s supersteps, as defined by eq. 4.

3.2.2 Example: SpMV (CSR). On a similar note, to demonstrate the ability to model irregularity and latency-boundness, Algorithm 2 shows an example application on the sparse matrix-vector multiplication (SpMV) kernel using the CSR storage format [16]. The matrix rows are partitioned across P_d workers, with each worker s assigned a contiguous block of $m_s = m \ / \ P_d$ rows. For each local row i_r, the worker accumulates its partial results in a temporary scalar variable by iterating over the nzr_s nonzeros from $row_{pt}[i_r]$ to $row_{pt}[i_r + 1]$, and writes the final value back to $y[i_r]$. Due to the row distribution, each worker writes in separate areas of y without synchronization, and the algorithm can be expressed as a single superstep. But, this approach introduces potential imbalance (both for computation and communication), since the total number of nonzeros each worker processes (nz_s) directly depends on the structure of the input matrix. Regarding computation, the local work performed by workers is labeled as scalar with $w_{i,s}^{\text{scalar}} = 2 \cdot nz_s$.

Algorithm 2: SpMV (CSR format): Compute $y := Ax$

Model Data: Total workers P_d, worker id s
Data: Matrix A in CSR $\{row_{pt}[n+1], col[nz], val[nz]\}$, input vector $x[n]$, in/output vector $y[m]$, dtype size b
$m_s \leftarrow m \ / \ P_d$
for $i_r \leftarrow s \cdot m_s$ **to** $(s+1) \cdot m_s$ **do**
  $y_{temp} = y[i_r]$
  for $j \leftarrow row_{pt}[i_r]$ **to** $row_{pt}[i_r + 1]$ **do**
    $y_{temp} \leftarrow y_{temp} + val[j] \cdot x[col[j]]$
  $y[i_r] = y_{temp}$
Algo Parameters: No synchronization $\rightarrow$ 1 superstep
- $nzr_s[i_r] = row_{pt}[i_r + 1] - row_{pt}[i_r], nz_s = \sum(nzr_s[i_r])$
- $w_{i,s}^{scalar} = 2 \cdot nz_s$
- y, row_{pt}: $kn_{i,s} = 3, h_{i,s,k} = m_s \cdot b$
- col, val: $kn_{i,s} = 2 \cdot m_s, h_{i,s,k} = nzr_s[i_r] \cdot b$
- x: $kn_{i,s} = nz_s, h_{i,s,k} = b$

Cost Model:
- $comp_i = \max_{s=0}^{P_d}(\frac{2 \cdot nz_s}{r^{scalar}})$
- $comm_i = \max_{s=0}^{p_{l_i}}($
  $\sum_{k=0}^{3} \left(m_s \cdot b \cdot g_{l_i} + ls_{l_i}\right) + \sum_{k=3}^{3+2 \cdot m_s} \left(nzr_s[i_r] \cdot b \cdot g_{l_i} + ls_{l_i}\right) +$
  $\sum_{k=3+2 \cdot m_s}^{3+2 \cdot m_s + nz_s} \left(b \cdot g_{l_i} + ls_{l_i}\right)$)
  $= \left(\max_{s=0}^{P_d} nz_s\right) \cdot (ls_{l_i} + 3 \cdot g_{l_i} \cdot b) +$
  $m_s \cdot (2 \cdot ls_{l_i} + 3 \cdot g_{l_i} \cdot b) + 3 \cdot ls_{l_i}$
- $T = \max(comp_i, comm_i)$

Communication is much more complex for CSR, since access patterns on the sparse data structures and the vectors vary; consequently, each must be modeled separately. y and row_{pt} are accessed contiguously on the external loop only, represented as 3 streams of size $m_s \cdot b$. Then, col and val are accessed contiguously on *each* internal loop, represented by $2 \cdot m_s$ streams of size $nzr_s[i_r] \cdot b$. Finally, the input vector x is accessed irregularly, which is represented by nz_s streams of size b. These individual stream costs are summed up to calculate the algorithm's total communication cost $comm_i$, and then used to calculate the final cost T in combination with $comp_i$.

Based on these costs, we can show the model's ability to highlight the performance trade-offs of SpMV in COO and CSR formats. First, in both cases, the models confirm the low operational intensity/ memory-boundness of SpMV. For COO, the model infers balanced computation and communication across workers, since work and data movement scale uniformly with nz_s. However, the implicit synchronization makes the kernel strongly latency-bound. For CSR, the model correctly shows fewer memory accesses and lower latency due to the lack of shared writes. At the same time, both computation and communication costs depend on nz_s, which

is structure-dependent, exposing CSR's load imbalance. The stream formulation also captures the known impact of many short rows on CSR performance, since having many small nzr_s values increases the relative latency. Finally, irregular accesses to x remain the main source of latency as expected. Overall, the cost expressions confirm that CSR is generally preferable due to lower footprint and latency, except for cases of severe imbalance or very low nonzeros per row.

3.3 Model instantiation

On modern CPUs, algorithms utilize multi-threaded execution to achieve optimal performance, and both the placement and number of threads directly affect available memory per tier, computational throughput, effective bandwidth, and synchronization costs. But, adding thread count and placement as model parameters would considerably expand the parameter space and complexity for the hardware model. To avoid this, we keep the hardware abstraction static, and instead treat each combination of thread count ($thread\_count$) and thread placement ($thread\_config$) as distinct *system configurations*, represented by separate hardware models Consequently, each system is modeled with multiple PaHiS hardware models corresponding to different thread counts.

3.3.1 Model coefficient initialization. Next, we provide the methodology for initializing the hardware coefficients for different hardware, for which we have developed an open-source, portable Python suite for automatic model initialization[2]. Since we view thread counts/placements as different *system configurations*, in practice we have to initialize $thread\_count_{num} \times thread\_config_{num}$ different (p_l, g_l, ls_l, m_l) sets, each with a depth of d levels, *for each system*. The suite is invoked on a new system given a high level hierarchy baseline (d, p_l, m_l) extracted via hwloc [9], and a target set of $thread\_counts$, and performs C++ microbenchmarks to calculate g_l, ls_l level-by-level for the given thread counts and two thread placement policies; *close* (bind consecutively on cores) and *spread* (spread round-robin across NUMA domains).

To measure the latency and bandwidth coefficients per-level, we iterate over the d hierarchy levels. For each level l, we use five kernel microbenchmarks (ker): four basic kernels that operate on consecutive data for bandwidth (`copy`, `daxpy`, `triad`, and `stream`), plus a `random pointer chase` kernel for latency. We use $m_l \cdot \frac{P_d}{p_l}$ as the maximum working set (WS_{max}), and $m_{l-1} \cdot 2 \cdot \frac{P_d}{p_l}$ as the minimum working set (WS_{min}). Before each ker benchmark, we run a smaller ker of working-set WS_{min} / 2 to dirty lower-level caches. Then, each ker is timed over many working-sets (default = 1000), normally distributed in $[WS_{min}, WS_{max}]$, each with total communication volume H_{ker}. The resulting timings t_{ker}, along with H_{ker}, are used to fit a linear model $t_{ker} = G_{ker} \cdot H_{ker} + L_{ker}$ with least-squares regression. g_l is calculated from the first five kernels with $g_l = \sum(G_{ker})$ / 5, while ls_l from all of them with $ls_l = \max(L_{ker\{1-4\}}, G_{ker\{chase\}} + L_{ker\{chase\}})$. Finally, after estimating all levels, the suite optionally adds a *global_sync* level on top of the hierarchy for global synchronization with $g_{global\_sync} = 0$ and ls_{global_sync} estimated by timing a global barrier across the chosen thread count + placement. After microbenchmark completion, the resulting model coefficients for all *system configurations* of each machine are stored in a C++ header file. The suite's backend currently runs OpenMP-based microbenchmarks [14], but can be easily extended by injecting other base kernels (for GPUs or other architectures) without changes to the workflow.

[2] https://github.com/p-anastas/PaHiS/tree/legacy-pahis1

4 GraphBLAS integration

With the hardware coefficients initialized for each system configuration, the remaining requirement for applying the cost model is an algorithmic representation of the target workloads. Manually defining such a representation for each solver, however, is impractical due to the large number of solvers, the fact that most are not designed to map to the Multi-BSP model, and the lack of scalability to algorithmic variations, which would require separate manual representations. To address this, we integrate the cost model into ALP GraphBLAS [50], a C++ implementation of the GraphBLAS standard. Iterative solvers, graph algorithms, and data-analytic workloads are expressed in GraphBLAS as a sequence of algebraic operations on vectors and matrices. This primitive-oriented execution model is an ideal fit for model application; solvers use predefined building blocks that are easier to model, and solver variants are expressed as different series of blocks.

Following this logic, we model solvers as sequences of primitives, with each being treated as an individual superstep and evaluated independently. We construct an algorithmic representation for each primitive, assuming all its internal supersteps share a communication level l_i. l_i is selected based on the primitive's memory footprint: we choose the minimum level l_i whose $\text{total\_mem}_{l_i} = m_{l_i} \cdot \frac{P_d}{p_{l_i}}$ is larger than the working set of the primitive. When the invocation or completion of a primitive introduces synchronization (barriers, synchronous kernel calls, etc.), an additional global synchronization latency cost is added, modeled using the *global_sync* level. The solver cost is then obtained by aggregating the costs of all its constituent primitives. This approach allows primitive representations to be defined only once, and then reused for arbitrary solver compositions without additional modeling effort.

NUMA effects in PaHiS are modeled with different hierarchical levels, so selecting l_i for a primitive (in addition to the aforementioned capacity condition) should also account for NUMA effects when threads span multiple NUMA domains. To this end, we define three NUMA modeling policies (`NUMA_opt`) representing different degrees of data locality optimization: an optimistic, a balanced, and a pessimistic. The *optimistic* policy assumes that all data accessed by a thread is available in its local NUMA domain, and all data movement is therefore modeled at the local-NUMA level. The *balanced* policy assumes partial locality optimization, where regular accesses are served from local NUMA memory, while irregular accesses incur inter-NUMA costs. Finally, the *pessimistic* policy assumes no locality, with all accesses modeled as inter-NUMA data movement. In our experiments, we use the optimistic policy to reflect a well-optimized library layer.

4.1 Implementation

Our implementation is split into two parts: a cost-model layer that defines cost models for GraphBLAS primitives via C++ classes, and an integration layer for ALP/GraphBLAS that connects these models to compute solver-level costs.

Table 3: Evaluated CPU systems and key hardware characteristics.

ID	CPU	Freq. (GHz)	Cores	LLC (MB)	NUMA	Mem. (GB)	CPU Affinity	Thread Configurations
I	Huawei Kunpeng 920	2.60	96	96 (4 × 24)	4(2)	512	close, spread	1, 2, 4, 8, 16, 24, 32, 48, 64, 96
II	Intel Xeon Gold 6238	2.10	88	64 (2 × 32)	2(2)	200	close, spread	1, 2, 4, 8, 12, 22, 32, 44, 64, 88
III	AMD EPYC 9634	2.25	168	768 (24 × 32)	8(2)	600	close, spread	1, 2, 4, 7, 8, 14, 21, 42, 84, 168

The cost model layer is implemented as a header-only C++ library that provides cost prediction functions for a set of GraphBLAS primitives, and is structured into two main components: (i) a *hardware model* namespace and (ii) *algorithm model* namespaces. The hardware model namespace defines a data structure `HW_params` with the parameters listed in Table 1 and a `Sys_config` storing all different system configurations of a machine (initialized by the microbenchmarks in Section 3.3.1), along with functions to interface with them. The algorithm model namespace defines a data structure `Algo_params` corresponding to the parameters in Table 2 and a `predict(HW_params, Algo_params, NUMA_opt)` function that computes the cost of an algorithm for a given system configuration and NUMA_opt. Finally, for each GraphBLAS primitive `prim`, we define a `get_params_prim(args)` function that takes the primitive's arguments (problem dimensions, operand types, and datatypes) and instantiates a corresponding `Algo_params` representation. Combining the above, we can evaluate the cost of any primitive for any system configuration (number of threads, placement policy) during runtime.

To inject PaHiS into GraphBLAS, we use a redefinition mechanism: we inject a C++ header (`cost_backend.hpp`) that provides function wrappers for the original GraphBLAS primitive definitions, capturing primitive input parameters. These wrappers add `get_params_prim` and `predict` calls before the actual primitive execution, with compilation flags controlling whether prediction, execution, or both are enabled. When prediction-only mode is used, the system acts as a model aggregator: all primitives invoked by a workload or solver are captured automatically, and their individual costs are used to estimate the total solver cost. This offers offline cost prediction for arbitrary compiled GraphBLAS workloads. The main limitation of this approach is that it cannot evaluate branch conditions dependent on runtime values, so it requires static workflows, such as solvers with a preset number of iterations, or user input on these cases. On the other hand, when both prediction and execution are enabled, GraphBLAS workflows execute identically to the original implementation, including branches and data-dependent decisions, and the online costs can be utilized for runtime autotuning. However, depending on the autotuning target, this introduces prediction overheads affected by many factors, such as cost-model complexity, the number of `predict` invocations across system configurations and even implementation efficiency. Since this work focuses on model feasibility and validation, the exploration and optimization of runtime usage is left for future work.

5 Evaluation

In this section, we validate the proposed cost model's ability to capture relative performance behavior across different problem instances, hardware platforms, system configurations, and algorithmic choices. We first present a global validation of the correlation between predicted costs and measured performance, comparing against two model baselines: Multi-BSP [46] and a Hier-Roofline. We then evaluate the model on different model application scenarios: HW–SW co-design, autotuning, and algorithm selection.

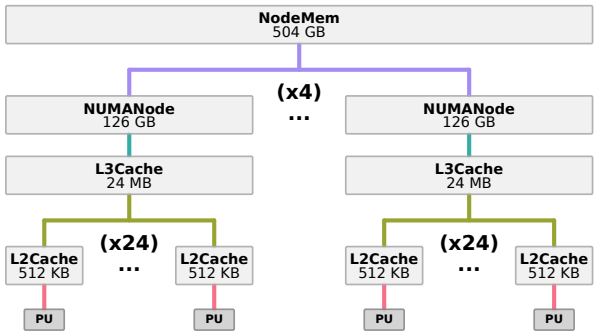


Figure 1: PaHiS hierarchical hardware model for T_I. The leafs ($l = 0$) correspond to the PUs where computation takes place. Each level above them represents a memory level l, annotated with its capacity (m_l), and the multiplier label below it indicates its number of subcomponents (p_l).

5.1 Experimental Setup

Table 3 summarizes the three CPU testbeds used in the evaluation, covering the ARM, Intel, and AMD processor architectures (henceforth T_I, T_{II}, and T_{III}, respectively). For each system, the table reports the processor model, rate, total core count, last-level cache (LLC) capacity, NUMA organization (and number of actual sockets), and main memory size. In addition, it lists the CPU affinity policies and the set of thread-count benchmarks for each platform, which together define the available *system configurations*. For each system, we consider ten thread counts which, in combination with affinity, capture execution at the L2, LLC, inter-NUMA, inter-socket, and full-node levels. We note that T_I and T_{II} correspond to earlier-generation systems with comparable peak performance, but different focus: T_I emphasizes higher thread counts and a deeper hierarchy, while T_{II} favors a simpler NUMA and cache organization with fast caches. In contrast, T_{III} represents a more recent, high-end platform, featuring a much higher core count and LLC capacity, a deeper and more complex hierarchy, and higher overall hardware capabilities. For modeling the hardware of all three testbeds, we use a five-level structure ($d = 5$, *levels[1-5] = [L2, L3, NUMA, NodeMem, global_sync]*); they only differ in per-level capacities m_l (Table 3), subcomponent count p_l and in the underlying coefficient values (g_l, ls_l). Figure 1 visualizes this hierarchy for T_I, and bellow we list the p_l values of each system (the synthetic *global_sync* level

Table 4: Distribution of validation matrices based on solver workload composition (SpMV vs vector/sync percentage).

	T_I	T_{II}	T_{III}
SpMV-dominated (> 50% SpMV)	15/42	13/42	2/42
Vector ops/sync overhead-dominated	9/42	8/42	19/42
Mixed (varies per thread count)	18/42	21/42	21/42

introduced in Section 4 with $p_5 = 1$ is omitted):

$$T_I : \text{PU} \xrightarrow{\times 1} \text{L2} \xrightarrow{\times 24} \text{L3} \xrightarrow{\times 1} \text{NUMA} \xrightarrow{\times 4} \text{NodeMem}$$

$$T_{II} : \text{PU} \xrightarrow{\times 2} \text{L2} \xrightarrow{\times 22} \text{L3} \xrightarrow{\times 1} \text{NUMA} \xrightarrow{\times 2} \text{NodeMem}$$

$$T_{III} : \text{PU} \xrightarrow{\times 1} \text{L2} \xrightarrow{\times 7} \text{L3} \xrightarrow{\times 3} \text{NUMA} \xrightarrow{\times 8} \text{NodeMem}$$

Finally, in Tables 7, 8, 9 (Appendix), we report the full per-tier bandwidth (g_l) and latency (ls_l) coefficients used in this work, together with the system-wide compute rates (r^{scalar}, r^{vec}).

For our evaluation, we use two iterative solvers, Conjugate Gradient (CG) and BiCGstab, implemented in ALP/GraphBLAS, which accept sparse matrices in Matrix Market (`.mtx`) format. We use a real-matrix dataset of 42 matrices from the SuiteSparse Matrix Collection from a mix of diverse application families (e.g., structural mechanics, circuit simulation, PDE discretizations, meshes/graphs, and web/social networks). Table 10 (Appendix) shows the full list of matrices used in the evaluation. Matrix selection was based on prior SpMV modeling and autotuning work [17, 18, 22, 23, 28, 38], extended intentionally with smaller/sparser matrices to stress-test mixed workloads where SpMV execution time is less dominant. The selected matrices cover a wide range of memory footprints (3.6 KB − 3.6 GB), matrix dimensions(67 − 50 M), and nonzero (nnz) counts (300 − 300 M) and patterns. Table 4 summarizes the coverage of our validation dataset across SpMV-dominated workloads (capturing memory-bound and irregular sparsity behaviors), vector-operation-dominated workloads (capturing synchronization overheads and solver cost aggregation effects), and mixed workloads where bottlenecks overlap for different thread counts.

We run benchmarks for each combination of system, configuration, algorithm, and matrix ($3\times(2\times10)\times2\times42 = 5040$ combinations). First, we run a per-primitive tracing benchmark (prediction disabled), which uses the injection header to run, record, and time all primitive invocations. The results are aggregated by primitive invocation variation, grouping all primitive calls with identical input parameters and filtering only primitives called during the iterative phase. Then, we run a *raw* benchmark for the solver using the ALP GraphBLAS implementations without injection, to measure end-to-end runtime without tracing overhead. All benchmarks execute 10 warmup iterations followed by 10 timed iterations, and are repeated 5 times per benchmark.

5.2 Model Validation

First, we evaluate the model's ability to capture performance trends. Given the high-level nature of the cost model, absolute prediction accuracy is not the primary objective; instead, we assess how the predicted costs correlate with measured execution time. We compare PaHiS against two baselines: Multi-BSP, which PaHiS extends,

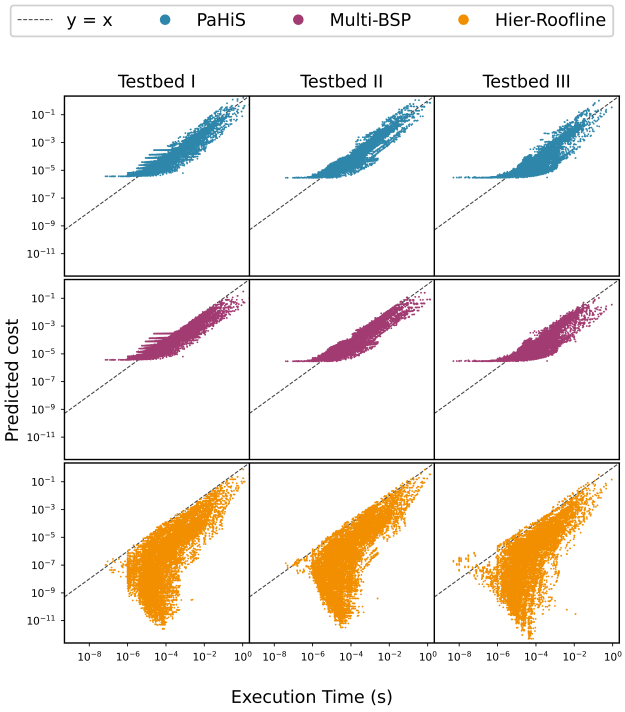


Figure 2: Correlation of predicted cost ($y-axis$) versus measured execution time ($x-axis$) for each primitive, grouped per testbed (*columns*) and comparing Hier-Roofline, Multi-BSP, and PaHiS (*rows*). The dashed line denotes perfect correlation ($y = x$). In all testbeds, our model outperforms the other two, with the Hier-Roofline completely ignoring latency effects and Multi-BSP clustering below $y = x$ due to the lack of irregularity modeling.

and hierarchical roofline [47] (henceforth Hier-Roofline). Both baselines use the coefficients from Section 3.3, so the comparison reflects the cost-model logic itself rather than differences in coefficient estimation. We also note that Multi-BSP is parametric: its latency term l is normally bound manually by the algorithm designer and also encodes no NUMA information. Consequently, to produce a comparable *cost-time prediction*, we evaluate it within the PaHiS pipeline (Sections 3.3 and 4), replacing only the cost-model logic. The evaluation is structured in two parts: first, we consider per-primitive cost prediction, and second, we evaluate the solver-level cost aggregation.

5.2.1 Primitive level. Figure 2 shows the predicted cost against measured execution time for all primitive invocation variations observed in CG and BiCGStab (10 variations, 50,400 data points), grouped by testbed (columns) and model (rows). Across all testbeds, the Hier-Roofline model shows high deviation from execution time, with a strong bias toward under-prediction. This is most visible in short executions - corresponding to small problems, lightweight primitives, and highly irregular access patterns - where ignoring latency leads to overly optimistic estimates. Multi-BSP, which is equivalent to PaHiS for regular problems, captures well the latency

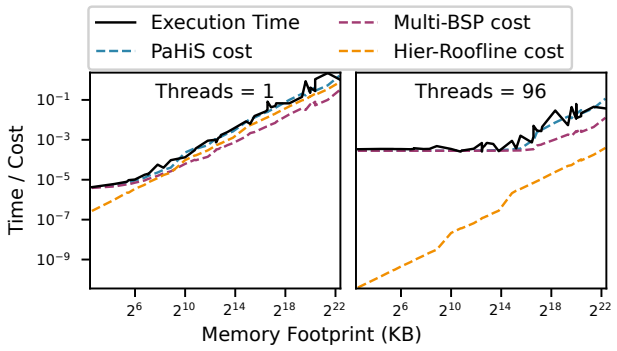


Figure 3: Execution time and cost ($y-axis$) of sparse matrix-vector multiplication ($SpMV-CSR$) on T_1, for different memory footprints ($x-axis$) and comparing serial (left) vs parallel execution with max threads (right). All three models track the serial trend, but Multi-BSP and the Hier-Roofline under-predict. Under parallel execution, the Hier-Roofline fails outright due to its lack of latency modeling, while Multi-BSP tracks the trend but with widening under-prediction.

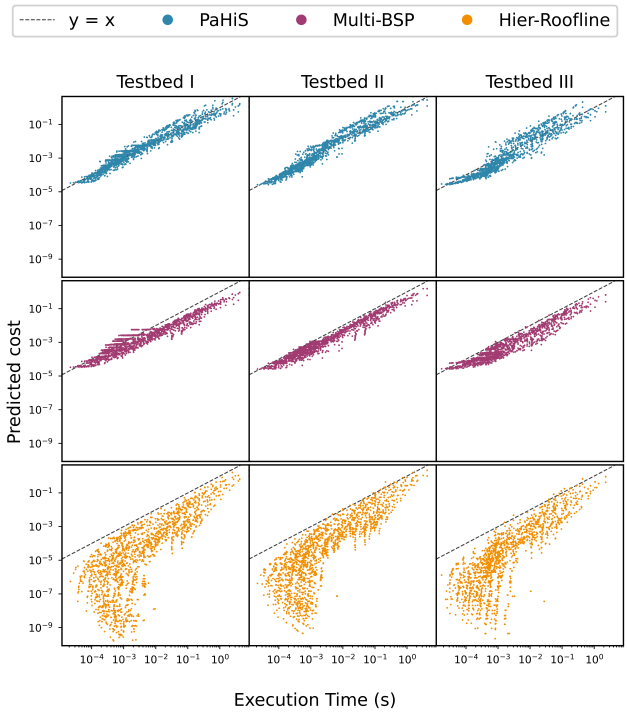


Figure 4: Correlation of predicted cost ($y-axis$) versus measured execution time ($x-axis$) for a solver iteration (CG and BiCGStab), grouped per testbed ($columns$) and comparing Hier-Roofline, Multi-BSP, and PaHiS ($rows$). The dashed line denotes perfect correlation ($y=x$). Models follow their primitive-level patterns after solver aggregation: the Hier-Roofline retains its wide scatter, Multi-BSP's offset below $y=x$ widens due to SpMV dominating solver-iteration time, and PaHiS clusters even more around $y=x$.

overheads of the dense-vector primitives, which in this figure correspond to 90% of the points (mxv is the only irregular one). However, on the remaining 10% of mxv points, it considerably under-predicts, resulting in a visible displacement below $y=x$ across all testbeds. In contrast, the PaHiS costs cluster around the $y=x$ line, with only a small bias towards under-prediction, which is consistent with the optimistic assumptions we use for NUMA access.

To better understand the observed divergence between the three models, we give an example for sparse matrix–vector multiplication (SpMV), the dominant kernel in sparse iterative solvers. Figure 3 shows execution time and predicted cost as a function of memory footprint for serial (left) vs parallel (right) execution on T_1. In serial execution, SpMV is mostly bandwidth-bound, so all models capture the *trend* as the footprint increases. However, unlike PaHiS, both the Hier-Roofline and Multi-BSP under-predict *performance*, because their cost expressions do not account for the latency of irregular accesses. Parallel execution introduces additional latency–bandwidth tradeoffs, arising from irregular data movement, partially over NUMA, and thread overheads/synchronization. Here, the Hier-Roofline fails outright due to its lack of latency modeling, while Multi-BSP captures the *trend* due to its *sync* step (overheads/synchronization between supersteps) but misses the internal latency of the *comm* step, leading to a widening *performance* under-prediction. PaHiS, in contrast, closely tracks the *trend* and *performance* across the whole range of problem sizes.

5.2.2 Solver level. We next extend the evaluation to cost prediction for the entire solver iteration. Figure 4 presents the correlation between predicted cost and measured execution time, similar to Figure 2, but for the entire set of solvers (CG and BiCGStab). Cost aggregation into a solver-iteration prediction (Sec. 4) works consistently across all three models, with each model's solver-level predictions being a weighted sum of its per-primitive behavior. This results in similar correlation patterns, with the exception that SpMV has a large impact on the trend since it accounts for a considerable percentage of solver time. Consequently, the Hier-Roofline carries its low correlation due to second-order errors, and Multi-BSP's offset below $y=x$ increases because of the aforementioned gap in SpMV prediction. On the other hand, PaHiS shows similar correlation at the solver level, with predictions actually clustering more tightly around the $y=x$ line.

To further analyze this behavior, Table 5 reports the Spearman correlation (ρ) and geometric mean ratio (GMR) between predicted cost and measured execution time, per primitive and for the total solver. We make three core observations. First, the Hier-Roofline performs worst on both metrics: it has low correlation ($\rho \in [0.65, 0.86]$) and very low $GMR (\leq 0.02)$ due to its lack of latency modeling. Second, PaHiS and Multi-BSP have identical ρ and GMR on every dense-vector primitive since the two cost models are equal when no irregularity is present; the only primitive on which they diverge is SpMV, where Multi-BSP considerably underestimates cost (PaHiS $GMR = 0.82$ vs Multi-BSP $GMR = 0.22$) due to no *comm*-irregularity modeling. Third, the SpMV kernel has the

Table 5: Spearman correlation (ρ) of cost vs time, and the geometric mean (GMR) of their $rate = \frac{cost}{time}$, grouped by primitive. $GMR < 1$ = under-prediction bias, > 1 = over-prediction bias.

	PaHiS		Multi-BSP		Hier-Roofline	
Function	ρ	GMR	ρ	GMR	ρ	GMR
set	0.834	0.42	0.834	0.42	0.725	0.002
foldr	0.909	1.12	0.909	1.12	0.648	0.008
foldl	0.933	1.11	0.933	1.11	0.680	0.009
dot	0.960	0.62	0.960	0.62	0.665	0.004
eWiseMul	0.962	1.44	0.962	1.44	0.693	0.010
mxv($SpMV$)	0.972	0.82	0.951	0.22	0.864	0.015
Solver	0.967	0.94	0.948	0.44	0.795	0.01

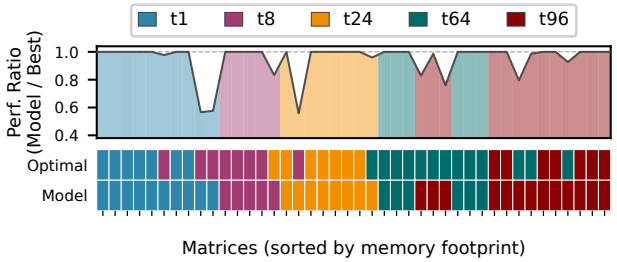


Figure 5: Comparison of model-achieved vs optimal performance for CG on T_I with close affinity. The top subplot shows the performance of model-selected thread counts, normalized to the optimal performance (1.0) for each matrix to assess the cost of misspredictions. The bottom shows the optimal vs model-selected thread count at each matrix (colour mismatch = misprediction).

highest ρ for every model, which happens because the dense primitives operate on much smaller vectors (effectively one dimension of SpMV), where launch, synchronization, and ALP-implementation overheads (critical sections, atomics, barriers) dominate the per-call time and add noise to their correlation. As a result, since SpMV accounts for a good majority of solver-iteration time, the solver-level ρ and GMR closely reflect those of SpMV: PaHiS achieves slightly higher correlation than Multi-BSP ($\rho = 0.967$ vs 0.948), but their GMR values diverge greatly ($GMR = 0.94$ vs 0.44); Multi-BSP results in geomean $> 2\times$ under-prediction inherited from its SpMV gap. Overall, PaHiS is the only model that captures both the *trend* (high ρ) and *performance* (GMR close to 1) for sparse workloads with its irregularity-aware additions over plain Multi-BSP.

5.3 Thread autotuning application

Following validation, we apply the model to a more practical use-case: thread autotuning in ALP GraphBLAS. Here, the optimization problem boils down to recognizing the optimal *performance regions* for different thread counts and matching them to problem characteristics. In the following subsection, we assess the model's ability to accurately solve this optimization problem and the performance improvements achieved in ALP GraphBLAS as a result.

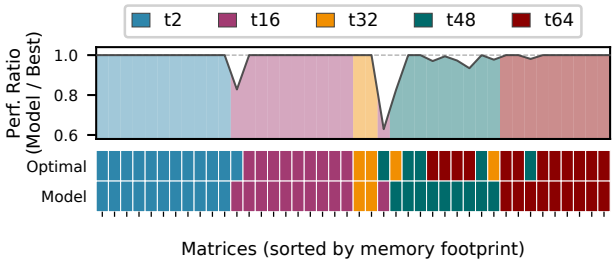


Figure 6: Comparison of model-achieved vs optimal performance for CG on T_I with spread affinity. The top subplot shows the performance of model-selected thread counts, normalized to the optimal performance (1.0) for each matrix to assess the cost of misspredictions. The bottom shows the optimal vs model-selected thread count at each matrix (colour mismatch = misprediction).

5.3.1 Thread selection. Figure 5 illustrates thread selection for CG execution in T_I, for close CPU affinity. It shows the five best-performing (on average) thread counts for the whole dataset, normalized by the maximum performance achieved by any of these threads, per matrix. The model effectively identifies the distinct performance regions, resulting in optimal thread selection for most cases (30 out of 42). Additionally, for the remaining 12 cases, the model selects a good thread count, with mispredictions occurring near region boundaries and for most cases, causing only minor performance deviations from the maximum, with 3 cases around 60% and the rest above 80% of the optimal. Finally, we note that half of the mispredictions happen between 64 and 96 threads, where performance is similar (64 threads are close to saturating full bandwidth for T_I). Figure 6 shows the same plot, but for spread CPU affinity, which results in different tradeoffs (less L3 sharing but more NUMA traffic). The model performs even better in this scenario (34 out of 42 optimal choices), with all mispredictions resulting in very low performance deviation - all within 90% of the optimal performance with only one exception (63%).

Finally, we consider selection with the models from section 5.2. The hierarchical roofline has no latency term, so it is completely unsuitable for thread selection: it always selects the bandwidth-maximizing thread count (max threads in ~95% of cases), scoring only 14/42 and 9/42 on T_I for close and spread, respectively. Multi-BSP, on the other hand, models launch/synchronization costs and therefore performs better, scoring 24/42 and 23/42 on T_I close and spread. Still, it remains strictly inferior to PaHiS because it mispredicts on boundary cases where irregularity decides the optimum.

5.3.2 Autotuning Performance. After showing that the model selects near-optimal thread counts, we evaluate the performance impact of applying this autotuning across the full dataset. Table 6 reports the performance improvements for each benchmark scenario (system, algorithm, matrix), for both close and spread affinity, when selecting the thread count that minimizes the cost model, compared to using fixed static thread counts. On average, autotuning achieves performance improvements of 39 – 48% across all

Table 6: Performance improvement of autotuning versus static thread configurations, per testbed and policy.

T_I	Close	Spread	T_{II}	Close	Spread	T_{III}	Close	Spread
Threads	Perf Impr %	Perf Impr %	Threads	Perf Impr %	Perf Impr %	Threads	Perf Impr %	Perf Impr %
1	+69.1%	+59.1%	1	+66.2%	+63.7%	1	+50.0%	+51.0%
2	+62.6%	+61.5%	2	+60.3%	+51.0%	2	+53.7%	+56.5%
4	+48.6%	+58.9%	4	+51.6%	+49.1%	4	+44.9%	+49.4%
8	+39.3%	+45.3%	8	+38.9%	+38.3%	7	+31.1%	+48.1%
16	+42.1%	+37.2%	12	+34.1%	+34.5%	8	+45.5%	+47.2%
24	+39.8%	+33.3%	22	+28.4%	+24.1%	14	+43.7%	+37.0%
32	+40.6%	+34.0%	32	+35.5%	+25.3%	21	+44.0%	+32.4%
48	+45.1%	+36.4%	44	+36.8%	+36.2%	42	+43.7%	+33.5%
64	+43.3%	+42.8%	64	+45.2%	+35.8%	84	+47.6%	+40.3%
96	+53.3%	+53.9%	88	+53.8%	+44.7%	168	+56.2%	+49.0%
avg	+47.8%	+45.5%	avg	+44.1%	+39.4%	avg	+45.7%	+44.0%

systems and affinity policies. It's also interesting to note that the selection problem is non-trivial; the optimal static thread count varies significantly across systems and affinity settings (e.g., 8 and 24 threads for T_I, 22 and 22 threads for T_{II}, and 7 and 21 threads for T_{III}, for close and spread affinity, respectively). Even compared to these best static thread counts, model-based autotuning offers $24-40\%$ better performance, and these already assume prior knowledge/tuning, making this baseline very optimistic in practice. In summary, these results show that PaHiS enables effective thread autotuning for ALP GraphBLAS, delivering strong performance improvements across all tested CPUs.

5.4 HW-SW codesign

Finally, we evaluate the model as a tool for high-level HW–SW co-design decisions. Specifically, we assess its ability to support (i) hardware selection, by predicting which system delivers the best performance for a given problem and algorithm, and (ii) algorithm selection, by choosing between alternative algorithms for solving a problem on a given system.

5.4.1 Hardware comparisson. First, we evaluate whether the model can predict which system delivers the best performance for a given problem. Among the evaluated platforms, T_I and T_{II} are of similar generation and are roughly comparable. On the other hand, T_{III} is a more recent, higher-end system with substantially higher core count ($\sim 2x$), memory bandwidth ($\sim 2x$ per-core, $\sim 4x$ total), and cache capacity ($\sim 8x$). Consequently, in a direct comparison, T_{III} consistently outperforms the other systems, making model-based hardware selection trivial. To enable a more meaningful evaluation, we therefore compare a single NUMA node of T_I and T_{II} against the full node of T_{III}, emulating three hardware scenarios with distinct characteristics: T_I favors lower latency and fewer threads, T_{II} favors higher parallelism and bandwidth at the cost of increased latency, and T_{III} represents a very high-throughput but high-latency configuration.

Figure 7 shows the normalized CG performance across these configurations for each matrix. The model accurately identifies the performance regions where each testbed excels, with T_I favored for latency-sensitive cases, T_{III} for bandwidth-dominated ones, and

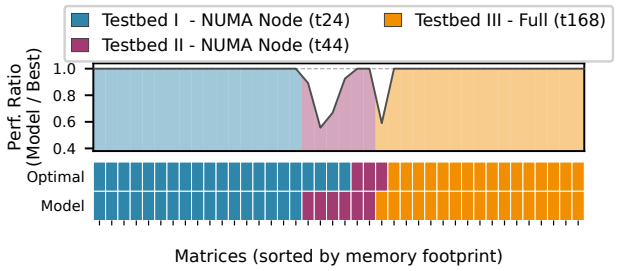


Figure 7: Comparison of model-achieved vs optimal performance for CG across all three testbeds for close policy (T_I and T_{II}: single NUMA, T_{III}: full node). The top subplot shows the performance of model-selected systems, normalized to the optimal performance (1.0) for each matrix. The bottom shows the optimal vs model-selected system for each matrix (colour mismatch = misprediction). The model accurately identifies the distinct performance regions each system excels in, with only a few boundary mispredictions (5/42).

T_{II} for a few middle cases. A few mispredictions (5 out of 42) happen for matrices that lie in the intermediate region, where the latency–bandwidth trade-off is balanced and small modeling errors influence the selected system. Overall, the model can effectively guide high-level hardware selection for problems with clearly differentiated system characteristics, with selection becoming more challenging when performance differences are marginal.

5.4.2 Algorithm comparison. Next, we consider algorithm selection: estimating the relative costs of two algorithms for a given problem and system to select the most efficient. To validate this scenario, we compare Conjugate Gradient (CG) and BiCGSTAB using an algorithm rating metric R_{algo} defined as

$$R_{algo} = \rho_{conv} \times \rho_{cost}$$

where $\rho_{conv} = conv\_iter_{BiCGSTAB} \ / \ conv\_iter_{CG}$ denotes their relative convergence ratio and $\rho_{cost} = time_{BiCGSTAB} \ / \ time_{CG}$ their *per-iteration* time ratio. Using this metric, BiCGSTAB has a lower

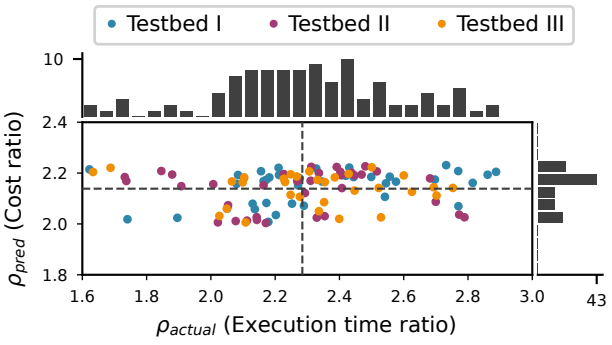


Figure 8: Predicted *per-iteration* cost ratio (ρ_{pred} = $cost_{BiCGSTAB}$ / $cost_{CG}$) vs measured time (ρ_{actual} = $time_{BiCGSTAB}$ / $time_{CG}$) for the best-performing configuration for each matrix, grouped by testbed. The bars show the value distribution for ρ_{pred} and ρ_{actual}. There is only a slight over-prediction of $cost_{CG}$ relative to $cost_{BiCGSTAB}$ (6% on average), showing the model captures the relative cost of the two algorithms well.

total execution time for a problem if $R_{algo} < 1$, while CG is better if $R_{algo} > 1$. We note that this process requires domain-expert input, since the model can predict per-iteration costs but not the convergence rates of solvers, so it emulates a high-level co-design scenario where ρ_{conv} is known a priori.

Figure 8 compares a predicted ρ_{cost} (= ρ_{pred}) with the actual measured (= ρ_{actual}) between CG and BiCGSTAB, for each matrix and testbed. We assume that the aforementioned autotuning mechanism is applied internally to select the best-performing thread configuration for each algorithm, system, and matrix, and restrict the comparison to these configurations. The measured ratios ρ_{actual} lie in the range 2.1–2.5, with no noticeable differences across testbeds, while the predicted ratios ρ_{pred} are concentrated in the range 2.0–2.15 (considerably lower variance). This translates to an average overprediction of CG cost relative to BiCGSTAB by 6%, which is carried to the predicted algorithm rating $R_{algo-pred}$. As a result, the model might incorrectly choose BiCGSTAB over CG only if ρ_{conv} is in the $\{0.43, 0.45\}$ range. In practice, this range represents a very small subset of cases and corresponds to a region where CG and BiCGSTAB exhibit nearly identical performance, resulting in negligible performance impact from mispredictions. For all other values of ρ_{conv}, the model selects the correct algorithm for a given problem and system.

6 Discussion and Future Work

In this section, we briefly mention the three current open paths under development.

6.1 Library integration

The current modeling integration in GraphBLAS is implemented through function wrapping and compile-time flags that enable *model runs* in place of actual execution, and is intended as a proof of concept. A full end-to-end integration of modeling and autotuning within a library is considerably more complex: it requires partial offline processing during compilation, combined with a lightweight runtime mechanism for online cost evaluation based on problem characteristics used for autotuning decisions. To this end, we are developing a symbolic layer that enables automatic generation and evaluation of symbolic cost expressions for algorithms implemented in GraphBLAS. More specifically, we will use automated analysis passes to prune non-deterministic paths and derive fully symbolic models for any solver implemented in ALP GraphBLAS, which we subsequently can use to target any subset of parameters for hardware-software co-optimization.

6.2 Accelerator modeling

The model proposed in this work is intentionally structured to align with the hierarchical execution model of modern accelerators. Its hardware abstraction levels naturally map to the multiple levels of parallelism exposed by GPUs (warps, thread blocks, and grids in CUDA), and algorithms expressed in accelerator programming frameworks map directly to our algorithmic parameters, since they are already expressed hierarchically using different memory, data movement, and synchronization scopes. Additionally, the microbenchmarks we designed for coefficient initialization use very simple core kernels that are easily implementable on accelerators, making model deployment on such platforms relatively simple. Validation on accelerators was left outside the scope of this work, as our reference framework (ALP GraphBLAS) currently does not support accelerator offloading, but extending the evaluation to accelerator platforms is our current focus.

6.3 Distributed modeling

Similarly, the model is already compatible with distributed-memory execution: adding a single inter-node communication level at the top of the hierarchy is straightforward and mirrors previous parallel models [45, 46]. We plan to validate this using the distributed backend of ALP-GraphBLAS, but omit it in this work because the current MPI-based implementation suffers from software-layer inefficiencies that overshadow the hardware-driven bottlenecks (L2/L3/NUMA) the model aims to expose. The OpenMP backend of ALP-GraphBLAS used in this work provides intra-node performance that more closely reflects the actual hardware behavior we want to validate against.

7 Conclusion

In this work, we introduce PaHiS, a hierarchical cost model that bridges the gap between high-level synchronous parallel models and fine-grained sparse performance modeling. The model consists of two parts: a hardware abstraction layer that captures system topology as a set of hierarchical parameters and coefficients, and an algorithmic layer that expresses algorithms in terms of hierarchical communication, computation, and synchronization patterns. Exploiting this hierarchical design, large iterative solvers are represented as sequences of simpler computational primitives whose costs can be modeled individually. For the hardware layer, we introduce an automated pipeline to initialize coefficients on new systems. For the algorithmic layer, we integrate the model with GraphBLAS

by defining expressions for its primitives, enabling cost estimation for any solver expressed on it.

We validate the proposed model on three distinct CPU systems, showing a 0.967 Spearman correlation between predicted costs and measured performance versus 0.785 for our hierarchical roofline baseline. We then evaluate its practical applicability in two representative scenarios: (i) thread autotuning and (ii) high-level HW–SW co-design. For thread autotuning, we demonstrate an end-to-end use of the model within ALP GraphBLAS, achieving substantial performance improvements across all testbeds. For HW–SW co-design, we consider two targets: selecting the most suitable hardware for a given algorithm and problem, and comparing alternative algorithms for a problem on a given system. In both cases, the model accurately identifies the distinct performance regions, with only minor mispredictions confined to their intermediate boundary areas. Future work will focus on extending the framework to accelerator-based and distributed environments and integrating symbolic runtime support for adaptive optimizations.

References

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1997. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *J. Parallel and Distrib. Comput.* 44, 1 (1997), 71–79. doi:10.1006/jpdc.1997.1346

[2] Bowen Alpern, Lemuria Carter, Ephraim Feig, and T. Selker. 1995. The uniform memory hierarchy model of computation. *Algorithmica* 12 (01 1995), 72–109. doi:10.1007/BF01185206

[3] Nicholas H Bacon, Patrick Bridges, Scott Levy, Kurt Ferreira, and Amanda Bienz. 2023. Evaluating the Viability of LogGP for Modeling MPI Performance with Non-contiguous Datatypes on Modern Architectures. In *Proceedings of the 30th European MPI Users' Group Meeting* (Bristol, United Kingdom) *(EuroMPI '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. doi:10.1145/3615318.3615326

[4] Gianfranco Bilardi, Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. 2001. On the Effectiveness of D-BSP as a Bridging Model of Parallel Computation. In *Proceedings of the International Conference on Computational Science-Part II (ICCS '01)*. Springer-Verlag, Berlin, Heidelberg, 579–588.

[5] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. 1996. BSP vs LogP. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy) *(SPAA '96)*. Association for Computing Machinery, New York, NY, USA, 25–32. doi:10.1145/237502.237504

[6] 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. doi:10.1145/567806.567807

[7] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 355–366. doi:10.1145/1989493.1989553

[8] Benjamin Brock, Scott McMillan, Aydın Buluç, Timothy Mattson, and José Moreira. 2023. GraphBLAS C++ Specification. https://github.com/GraphBLAS/graphblas-api-cpp.

[9] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 180–186. doi:10.1109/PDP.2010.67

[10] Jan-Willem Buurlage, Tom Bannink, and Abe Wits. 2017. Bulk-synchronous pseudo-streaming algorithms for many-core accelerators. arXiv:1608.07200 [cs.DC] https://arxiv.org/abs/1608.07200

[11] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. 2009. LogGPO: An accurate communication model for performance prediction of MPI programs. *Science in China Series F: Information Sciences* 52, 10 (01 Oct 2009), 1785–1791. doi:10.1007/s11432-009-0161-2

[12] Rezaul Alam Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. 2010. Oblivious algorithms for multicores and network of processors. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 1–12. doi:10.1109/IPDPS.2010.5470354

[13] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. 1996. LogP: a practical model of parallel computation. *Commun. ACM* 39, 11 (Nov. 1996), 78–85. doi:10.1145/240455.240477

[14] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. doi:10.1109/99.660313

[15] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. doi:10.1145/3322125

[16] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. 1982. Yale sparse matrix package I: The symmetric codes. *Internat. J. Numer. Methods Engrg.* 18, 8 (1982), 1145–1151. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1620180804 doi:10.1002/nme.1620180804

[17] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 292–301. doi:10.1109/icpp.2017.38

[18] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. BASMAT: bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 423–424. doi:10.1145/3293883.3301490

[19] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '78)*. Association for Computing Machinery, New York, NY, USA, 114–118. doi:10.1145/800133.804339

[20] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* 8, 1, Article 4 (Jan. 2012), 22 pages. doi:10.1145/2071379.2071383

[21] Alexandros V. Gerbessiotis. 2015. Extending the BSP model for multi-core and out-of-core computing: MBSP. *Parallel Comput.* 41 (2015), 90–102. doi:10.1016/j.parco.2014.12.002

[22] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2008. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. 283–292. doi:10.1109/PDP.2008.41

[23] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *J. Supercomput.* 50, 1 (Oct. 2009), 36–77. doi:10.1007/s11227-008-0251-8

[24] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. 1998. BSPlib: The BSP programming library. *Parallel Comput.* 24, 14 (Dec. 1998), 1947–1980. doi:10.1016/S0167-8191(98)00093-3

[25] Roger W. Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (March 1994), 389–398. doi:10.1016/S0167-8191(06)80021-9

[26] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC '07)*. Association for Computing Machinery, New York, NY, USA, Article 52, 10 pages. doi:10.1145/1362622.1362692

[27] T. Hoefler, T. Schneider, and A. Lumsdaine. 2009. LogGP in theory and practice – An in-depth analysis of modern interconnection networks and benchmarking methods for collective operations. *Simulation Modelling Practice and Theory* 17, 9 (2009), 1511–1521. doi:10.1016/j.simpat.2009.06.007 Advances in System Performance Modelling, Analysis and Enhancement.

[28] Kaixi Hou, Wu-chun Feng, and Shuai Che. 2017. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 713–722. doi:10.1109/IPDPSW.2017.155

[29] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. 2001. LogGPS: a parallel computational model for synchronization analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (Snowbird, Utah, USA) *(PPoPP '01)*. Association for Computing Machinery, New York, NY, USA, 133–142. doi:10.1145/379539.379592

[30] Hong Jia-Wei and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81)*. Association for Computing Machinery, New York, NY, USA, 326–333. doi:10.1145/800076.802486

[31] Ben H. H. Juurlink and Harry A. G. Wijshoff. 1996. The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II (Euro-Par '96)*. Springer-Verlag, Berlin, Heidelberg, 339–347.

[32] Thilo Kielmann, Henri E. Bal, Sergei Gorlatch, Kees Verstoep, and Rutger F.H. Hofman. 2001. Network performance-aware collective communication for clustered

wide-area systems. *Parallel Comput.* 27, 11 (2001), 1431–1456. doi:10.1016/S0167-8191(01)00098-9 Clusters and computational grids for scientific computing.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323. doi:10.1145/355841.355847

[34] S. Lennart Johnsson. 1991. Performance modeling of distributed memory architectures. *J. Parallel and Distrib. Comput.* 12, 4 (1991), 300–312. doi:10.1016/0743-7315(91)90002-Q

[35] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 117–126. doi:10.1145/2491956.2462181

[36] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman. 2022. Nonblocking execution in GraphBLAS. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 230–233. doi:10.1109/IPDPSW55747.2022.00051

[37] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman. 2023. Design and Implementation for Nonblocking Execution in GraphBLAS: Tradeoffs and Performance. *ACM Trans. Archit. Code Optim.* 20, 1, Article 6 (March 2023), 23 pages. doi:10.1145/3561652

[38] Panagiotis Mpakos, Dimitrios Galanopoulos, Petros Anastasiadis, Nikela Papadopoulou, Nectarios Koziris, and Georgios Goumas. 2023. Feature-based SpMV Performance Analysis on Contemporary Devices. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 668–679. doi:10.1109/IPDPS54959.2023.00072

[39] Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Albert-Jan N. Yzelman. 2024. Efficient Multi-Processor Scheduling in Increasingly Realistic Models. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*. ACM, 463–474. doi:10.1145/3626183.3659972

[40] Abdorreza Savadi, Morteza Moradi, and Hossein Deldari. 2012. Multi-DaC programming model: a variant of multi-BSP model for divide-and-conquer algorithms. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming* (Philadelphia, Pennsylvania, USA) *(DAMP '12)*. Association for Computing Machinery, New York, NY, USA, 41–46. doi:10.1145/2103736.2103743

[41] Martin Schmollinger and Michael Kaufmann. 2001. kappa NUMA: A model for clusters of SMP-machines. 42–50.

[42] Martin Schmollinger and Michael Kaufmann. 2002. Algorithms for SMP-Clusters Dense Matrix-Vector Multiplication. doi:10.1109/IPDPS.2002.1016540

[43] Alexandre Tiskin. 1998. The bulk-synchronous parallel random access machine. *Theoretical Computer Science* 196, 1 (1998), 109–130. doi:10.1016/S0304-3975(97)00197-7

[44] Pilar de la Torre and Clyde P. Kruskal. 1996. Submachine Locality in the Bulk Synchronous Setting (Extended Abstract). In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II (Euro-Par '96)*. Springer-Verlag, Berlin, Heidelberg, 352–358.

[45] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. doi:10.1145/79173.79181

[46] Leslie G. Valiant. 2011. A bridging model for multi-core computing. *J. Comput. System Sci.* 77, 1 (2011), 154–166. doi:10.1016/j.jcss.2010.06.012 Celebrating Karp's Kyoto Prize.

[47] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[48] T. Williams and R. Parsons. 2001. Exploiting hierarchy in heterogeneous environments. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001.* 1424–1431. doi:10.1109/IPDPS.2001.925125

[49] A.N. Yzelman and Rob H. Bisseling. 2012. An object-oriented bulk synchronous parallel library for multicore programming. *Concurr. Comput.: Pract. Exper.* 24, 5 (April 2012), 533–553. doi:10.1002/cpe.1843

[50] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. 2020. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation.

[51] Xing Zhao, Manos Papagelis, Aijun An, Bao Xin Chen, Junfeng Liu, and Yonggang Hu. 2019. Elastic Bulk Synchronous Parallel Model for Distributed Deep Learning. In *2019 IEEE International Conference on Data Mining (ICDM)*. 1504–1509. doi:10.1109/ICDM.2019.00198

[52] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 94–108. doi:10.1145/3178487.3178495

Table 7: Empirically measured hardware parameter coefficients for T_I (*Huawei Kunpeng ARM-920*). The synthetic *global_sync* level is omitted for g ($g = 0$ by construction) and denoted as *sync* for ls. Compute rates reported for the `double` dtype.

T	policy	Compute rate ($\times 10^9$ ops/s)		Bandwidth g ($\times 10^{10}$ bytes/s)				Latency ls ($\times 10^{-9}$ s)				
		r_{scalar}	r_{vec}	L2	L3	NUMA	Mem	L2	L3	NUMA	Mem	Sync
1	close	2.155	4.022	3.634	1.621	1.049	0.598	1.275	3.086	4.193	7.205	3612
1	spread	2.152	3.981	3.617	1.622	1.046	0.600	1.202	3.115	4.389	7.324	3612
2	close	4.296	7.841	7.448	2.400	1.781	1.150	0.734	1.946	3.621	3.996	3853
2	spread	4.307	7.960	7.247	2.296	1.805	1.193	0.626	2.035	2.558	3.662	3957
4	close	8.580	15.42	12.60	2.498	2.298	1.882	0.523	1.812	3.407	3.670	4507
4	spread	8.585	16.08	14.59	6.458	4.125	2.656	0.317	0.782	1.051	1.645	4812
8	close	17.19	30.84	23.42	5.293	3.682	3.559	0.249	1.523	1.809	1.949	6504
8	spread	17.27	31.79	29.16	9.319	7.113	5.020	0.197	0.494	0.896	0.976	7186
16	close	34.39	61.44	43.55	11.69	7.721	3.335	0.147	0.710	0.969	1.956	13258
16	spread	34.52	63.25	50.10	9.779	9.140	7.635	0.151	0.462	0.844	0.894	14587
24	close	51.43	87.80	55.34	18.92	9.401	3.622	0.098	0.451	0.798	1.823	23691
24	spread	51.72	94.38	77.99	17.59	13.24	10.24	0.085	0.427	0.487	0.638	25524
32	close	68.29	117.8	77.84	32.32	7.996	6.661	0.074	0.314	0.835	1.050	37803
32	spread	68.96	125.0	96.92	19.87	14.38	10.91	0.074	0.386	0.453	0.641	39999
48	close	102.7	174.5	110.5	37.88	10.86	6.377	0.049	0.225	0.608	1.111	77063
48	spread	103.3	186.2	140.2	30.93	17.53	11.09	0.049	0.246	0.351	0.650	79560
64	close	136.6	235.6	150.8	63.65	15.45	10.54	0.037	0.157	0.429	0.670	131038
64	spread	137.1	241.7	172.9	43.53	19.58	10.84	0.036	0.178	0.335	0.696	133270
96	close	204.8	349.5	222.0	75.57	21.09	10.60	0.026	0.114	0.312	0.711	283134
96	spread	202.5	350.9	212.1	74.57	21.07	10.58	0.028	0.114	0.313	0.714	283134

Table 8: Empirically measured hardware parameter coefficients for T_{II} (*Intel Xeon Gold 6238*).

T	policy	Compute rate ($\times 10^9$ ops/s)		Bandwidth g ($\times 10^{10}$ bytes/s)				Latency ls ($\times 10^{-9}$ s)				
		r_{scalar}	r_{vec}	L2	L3	NUMA	Mem	L2	L3	NUMA	Mem	Sync
1	close	1.743	13.10	4.265	1.803	1.057	0.925	1.433	3.480	6.018	6.799	2843
1	spread	1.743	13.10	3.878	1.822	1.078	0.922	1.415	3.438	5.875	6.839	2843
2	close	3.486	26.21	10.66	3.697	2.101	1.831	0.575	1.717	2.995	3.433	3245
2	spread	3.486	26.21	2.348	1.558	1.068	1.038	2.722	4.107	6.092	6.229	3309
4	close	6.972	52.42	21.20	7.179	3.944	3.471	0.292	0.879	1.598	1.806	4064
4	spread	6.972	52.42	4.726	3.047	2.098	2.055	1.357	2.076	3.074	3.130	4253
8	close	13.94	104.8	42.86	14.13	6.227	5.717	0.143	0.445	1.023	1.049	5764
8	spread	13.94	104.8	9.466	5.798	4.067	3.965	0.683	1.093	1.592	1.618	6184
12	close	20.92	157.2	64.03	21.25	7.023	6.813	0.095	0.296	0.906	0.930	7548
12	spread	20.92	157.2	14.02	8.075	5.694	5.516	0.456	0.784	1.133	1.164	8174
22	close	38.35	288.3	117.7	31.69	7.078	6.767	0.052	0.199	0.903	0.945	12371
22	spread	38.35	288.3	25.01	11.51	7.781	7.101	0.256	0.557	0.833	0.902	13410
32	close	55.78	419.3	171.0	64.74	12.45	9.551	0.036	0.095	0.513	0.667	17714
32	spread	55.78	419.3	32.07	10.93	7.885	7.115	0.200	0.586	0.829	0.900	19015
44	close	76.69	576.6	235.5	106.9	14.50	9.371	0.026	0.057	0.441	0.683	24812
44	spread	76.69	576.6	33.26	9.729	7.577	6.909	0.192	0.660	0.864	0.927	26230
64	close	111.6	838.7	187.0	72.63	13.30	9.768	0.034	0.087	0.482	0.655	38306
64	spread	111.6	838.7	83.83	22.93	10.34	9.622	0.076	0.279	0.629	0.666	39439
88	close	153.4	1153	192.2	44.88	14.16	9.213	0.034	0.142	0.453	0.695	57244
88	spread	153.4	1153	184.7	36.02	9.964	9.219	0.034	0.176	0.653	0.694	57244

Table 9: Empirically measured hardware parameter coefficients for T_{III} (*AMD EPYC 9634*).

T	policy	Compute rate ($\times 10^9$ ops/s)		Bandwidth g ($\times 10^{10}$ bytes/s)				Latency ls ($\times 10^{-9}$ s)				
		r_{scalar}	r_{vec}	L2	L3	NUMA	Mem	L2	L3	NUMA	Mem	Sync
1	close	1.867	14.04	9.822	7.076	2.482	2.022	0.623	0.848	2.874	3.508	2897
1	spread	1.867	14.04	8.330	7.147	2.551	2.069	0.771	0.843	2.774	3.384	2897
2	close	3.735	28.08	19.02	13.73	3.273	2.663	0.352	0.444	1.906	2.327	3038
2	spread	3.735	28.08	18.43	13.93	5.674	4.089	0.332	0.437	1.285	1.706	3339
4	close	7.470	56.16	35.71	24.06	4.029	3.154	0.181	0.258	1.676	2.119	3332
4	spread	7.470	56.16	38.53	26.65	13.06	10.53	0.162	0.233	0.608	0.738	4224
7	close	13.07	98.28	61.96	29.35	3.688	3.059	0.106	0.214	1.749	2.110	3800
7	spread	13.07	98.28	70.73	44.16	25.59	20.87	0.091	0.146	0.321	0.404	5552
8	close	14.94	112.3	74.17	51.71	4.461	3.931	0.088	0.121	1.439	1.674	3963
8	spread	14.94	112.3	81.74	50.24	28.43	23.49	0.078	0.128	0.275	0.348	5995
14	close	26.14	196.6	134.3	46.07	10.90	3.867	0.048	0.134	0.595	1.679	5019
14	spread	26.14	196.6	141.7	32.18	27.38	13.75	0.046	0.209	0.239	0.503	8650
21	close	39.22	294.8	207.8	52.90	14.88	4.231	0.031	0.122	0.427	1.516	6415
21	spread	39.22	294.8	210.7	32.33	29.32	16.82	0.031	0.201	0.221	0.383	11748
24	close	44.82	337.0	238.7	117.4	19.68	5.303	0.027	0.054	0.321	1.213	7068
24	spread	44.82	337.0	123.6	52.39	33.24	27.77	0.053	0.270	0.311	0.541	13075
32	close	59.76	449.3	324.4	156.5	26.84	7.938	0.020	0.040	0.235	0.808	8969
32	spread	59.76	449.3	168.0	24.28	21.06	12.14	0.039	0.263	0.301	0.519	16615
42	close	78.44	589.7	416.2	104.3	30.98	8.864	0.015	0.060	0.205	0.725	11671
42	spread	78.44	589.7	162.1	21.58	15.02	14.53	0.041	0.282	0.412	0.429	21041
84	close	156.9	1179	687.5	194.2	61.93	19.58	0.009	0.032	0.102	0.327	26982
84	spread	156.9	1179	292.5	51.11	45.22	26.60	0.023	0.124	0.151	0.152	39627
168	close	313.7	2359	1290	403.5	84.94	76.00	0.005	0.015	0.075	0.081	76801
168	spread	313.7	2359	1295	403.0	85.01	76.03	0.005	0.016	0.076	0.081	76801

Table 10: SuiteSparse matrices used in the evaluation, split by $footprint < 5MB$ (left) and $footprint > 5MB$ (right).

Matrix	CSR footprint (KB)	dim (n=m)	NNZ
west0067	3.71	67	294
fs_183_1	12.41	183	998
steam1	27.29	240	2248
bcsstk05	28.99	153	2423
poisson2D	29.76	367	2417
dwt_758	42.53	758	3,376
bcsstk06	93.75	420	7860
bcsstk07	93.75	420	7860
bcsstk08	156.07	1074	12960
bcsstk09	220.29	1083	18437
delaunay_n12	303.44	4096	24528
bcsstk14	750.66	1806	63454
delaunay_n14	1215.30	16384	98244
wiki-Vote	1247.52	8297	103689
p2p-Gnutella31	1977.59	62586	147892
rgg_n_2_15_s0	3883.63	32768	320480
gyro_m	4057.25	17361	340431
poisson3Da	4186.72	13514	352762
email-Enron	4451.87	36692	367662
delaunay_n16	4863.23	65536	393150
cit-HepPh	5075.32	34546	421578

Matrix	CSR footprint (KB)	dim (n=m)	NNZ
thermal1	7054.80	82654	574458
G2_circuit	9102.05	150102	726674
delaunay_n18	19455.16	262144	1572792
ASIC_320k	23895.73	321821	1931828
vanbody	27477.50	47072	2329056
apache2	59253.07	715176	4817870
ecology2	62453.02	999999	4995991
web-Google	63404.48	916428	5105039
delaunay_n20	77823.02	1048576	6291372
G3_circuit	95968.58	1585478	7660826
thermal2	105347.60	1228045	8580313
bundle_adj	238816.69	513351	20207907
road_central	451883.96	14081816	33866826
Emilia_923	476733.40	923136	40373538
wb-edu	708263.04	9845725	57156537
circuit5M	719262.50	5558326	59524291
Serena	756981.50	1391349	64131971
road_usa	769817.27	23947347	57708624
Long_Coup_dt0	995074.46	1470152	84422970
europe_osm	1465781.17	50912018	108109320
Queen_4147	3725757.80	4147110	316548962