Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics

Daniele Buono[†]
dbuono@us.ibm.com

Fabrizio Petrini[†]
fpetrin@us.ibm.com[*]

Fabio Checconi[†]
fchecco@us.ibm.com

Xing Liu[†]
xliu@us.ibm.com

Xinyu Que[†]
xque@us.ibm.com

Chris Long[§]
chris.longc@gmail.com

Tai-Ching Tuan[§]

[†] IBM Research, T. J. Watson Research Center
Yorktown Heights, NY, USA

[§] US Department of Defense

ABSTRACT

Sparse Matrix-Vector multiplication (SpMV) is a fundamental kernel, used by a large class of numerical algorithms. Emerging big-data and machine learning applications are propelling a renewed interest in SpMV algorithms that can tackle massive amount of unstructured data—rapidly approaching the TeraByte range—with predictable, high performance. In this paper we describe a new methodology to design SpMV algorithms for shared memory multiprocessors (SMPs) that organizes the original SpMV algorithm into two distinct phases. In the first phase we build a scaled matrix, that is reduced in the second phase, providing numerous opportunities to exploit memory locality. Using this methodology, we have designed two algorithms. Our experiments on irregular big-data matrices (an order of magnitude larger than the current state of the art) show a quasi-optimal scaling on a large-scale POWER8 SMP system, with an average performance speedup of 3.8×, when compared to an equally optimized version of the CSR algorithm. In terms of absolute performance, with our implementation, the POWER8 SMP system is comparable to a 256-node cluster. In terms of size, it can process matrices with up to 68 billion edges, an order of magnitude larger than state-of-the-art clusters.

CCS Concepts

•Computing methodologies → Linear algebra algorithms; Shared memory algorithms; *Vector / streaming algorithms;* **•Mathematics of computing → Graph algorithms; •Theory of computation → Graph algorithms analysis; Data structures design and analysis; Shared memory algorithms;** *Vector / streaming algorithms;*

Keywords

Power8, Sparse matrices, Storage Formats, SpMV, Data analytics

1. INTRODUCTION

Unstructured streams of information can be intuitively represented as graphs, whose vertices can be labeled with user names or events, and edges with the relationships between them. Recently, many research efforts have tried to provide powerful ways to extract structural properties from graphs. For example, recent studies show how several graph algorithms can be recast as a sequence of linear algebraic operations, such as generalized sparse matrix-matrix multiplication (SpGEMM) and sparse matrix-vector multiplication (SpMV) [7, 8]. This approach is becoming more and more prominent [7], because the capability of using linear algebra can greatly simplify data analysis.

Sparse matrix-vector multiplication constitutes one of the fundamental operations in linear algebra, and it is typically used in the solution of linear equations and for the application of linear transformations. The product operation is defined between a sparse $m \times n$ matrix A and a column vector $\mathbf{x}$ of size n, and produces a vector $\mathbf{b}$ of size m. For the sake of simplicity, in the rest of the paper we will assume A to be an $n \times n$ matrix. While SpMV has been extensively studied over the past few decades, it is gaining a renewed interest as a powerful tool to perform big data analysis. Emerging data intensive problems, and the shift from forensic to real-time analysis, pose emphasis on very large data sets, scalable performance and predictable response times, demanding a fresh look into sparse linear algebra algorithms.

In the domain of graph analysis, various methods require the use of SpMV on the adjacency matrix (or derivatives such as the Laplacian matrix), with the purpose of extracting various kinds of spectral information [18], such as the eigenvalues and eigenvectors of the Laplacian matrix. These, in turn, can be used for clustering, partitioning, and anomaly detection [5]. Other interesting applications of SpMV are in the Power method [17] to compute PageRank, in HITS [19],

[*]Fabrizio Petrini has since changed his affiliation. His current contact is fabrizio.petrini@intel.com

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926278

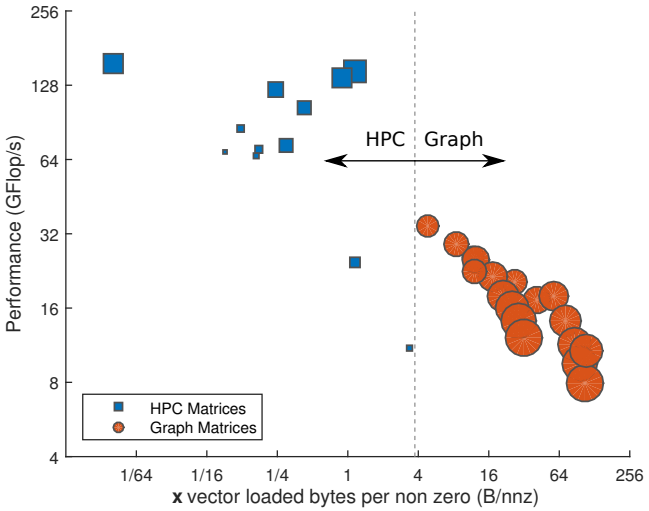


Figure 1: Correlation between Performance and Vector-originated memory transfers in CSR-based SpMV with HPC and Graph matrices. Point size is proportional to the memory footprint of the corresponding matrix.

Reference (Max Size)	Architecture	Improv. over CSR	R-MAT Performance (Graph par.)
Ashari et al. [3] (298M nnz)	GPU	2.34×	N/A
Tang et al. [27] (29M nnz)	Many-core Coprocessor	3×	∼ 7 GFlop/s (S 18, EF 32)
Boman et al. [5] (1.6B nnz)	CPU (Cluster) 256 Nodes	N/A	64.95 GFlop/s (S 26, EF 9)
Our Approach (68B nnz) (S 31, EF 16)	CPU (Shared Mem.)	Small graphs: 2× Large graphs: 3.8×	51.51 GFlop/s (S 26, EF 32) 44.02 GFlop/s (S 29, EF 32)

Table 1: State of the art results in the literature for graph-oriented SpMV algorithm. For synthetic graphs (R-MAT and BTER), S is the graph scale and EF the edge factor.

and in the Random Walk with Restart algorithm [28].

In this paper we use a parallel CSR as a baseline. CSR is a de facto standard for SpMV, available in most Sparse BLAS libraries, and usually compared with in SpMV papers [5, 27, 22]. By measuring the speedup w.r.t. CSR, the reader can easily compare our results to those of many other formats. Moreover, most of the formats introduced in the last decade focus on increasing vectorization opportunities, usually at the cost of padding data structures. This approach is counterproductive with very sparse and unstructured matrices such as graphs. The memory footprint of various formats with graph matrices will be analyzed in Section 5 but, as an example, two commonly implemented formats such as BSR[14] and ELLPACK(ELL)[4] would require, on average, respectively $\sim 7.6\times$ and $\sim 37000\times$ the space of CSR.

The performance of SpMV tends to be limited by the cost of data movement across the memory hierarchy, rather than arithmetic operations. In Figure 1, we characterize—and contrast—existing High Performance Computing (HPC) matrices [12] and social network, graph-structured matrices [9, 20], by analyzing the behavior of a CSR-based parallel SpMV implementation. We report the performance obtained on our target SMP architecture, a 8-socket IBM Power E870 server, on the y axis, and the average number of bytes of the $\mathbf{x}$ vector loaded in the cache hierarchy for each non-zero element of the matrix (B/nnz). Each point represent a single matrix, with the size of the point proportional to the size of the matrix.

While we remind the reader to see Section 5 for the complete set of experimental results, for the sake of this discussion we can observe that the peak SpMV CSR performance of this machine configuration is about 190 GFlop/s. The ideal B/nnz value depends on the specific matrix, with a higher number of bytes indicating poor cache locality when accessing $\mathbf{x}$. In complete absence of *temporal* and *spatial* locality on the vector, we can end up reading 128 B (the cache line size on POWER8): for each element of the matrix we read an entire cache line, containing not only the required element, but also others that are not needed for the computation.

Looking at Figure 1, we can observe a sharp difference between HPC and Graph Matrices, with the first appearing on the upper-left quadrant of the plot, while the latter confined to the lower-right quadrant. Most HPC matrices get good performance, averaged at 89 GFlop/s, but with high variance. The number of loaded bytes per non-zero is very low, exceeding 1 B only in rare cases, thus indicating that most matrices have a pronounced degree of locality. This, in turn, makes the classical SpMV CSR implementation effective, even if relatively sensitive to the matrix topology. It's also important to notice that larger matrices tend to show better performance.

Graphs on the other hand, display different characteristics. They can be much larger, approaching a terabyte in size, with much lower performance when compared to the HPC matrices, and a much higher number of vector traffic. The high memory traffic clearly indicates a poor level of locality, with a trend suggesting that bigger graphs show less locality and, consequently, lower performance. On average, graphs are $5\times$ slower than HPC matrices on this SpMV implementation.

1.1 Contributions

The paper provides the following contributions.

- An experimental analysis of the limitations of the classical CSR SpMV algorithm for big-data matrices using a large-scale, shared memory multiprocessor. With an empirical metric that can easily be captured using performance counters, the average number of loaded bytes of the $\mathbf{x}$ vector for non-zero element of the matrix, we are able to characterize intuitively the performance degradation incurred by very-large matrices.
- We propose a new methodology for executing SpMV on very large and irregular matrices, based on two distinct phases. In the first phase we build a scaled matrix, multiplying each nonzero by the corresponding element of $\mathbf{x}$. In the second phase the scaled matrix

is reduced by row to compute the output vector **b**. While this has the disadvantage of reading the input matrix twice—therefore being inherently sub-optimal for regular HPC matrices—it provides numerous opportunities to implement cache-efficient solutions in both phases, with the result of optimizing the amount of memory transferred for the two vectors.

- Using the proposed methodology, we have designed and implemented two algorithms, one using a more classic blocked approach and another one using a novel binning strategy. For both algorithms we provide a compact data representations that allows efficient and cache-friendly execution, offering predictable locality and performance. We presented an early version of the blocked algorithm in [8].
- We also present a large body of experimental results on a high-end shared-memory POWER8 system, with matrices approaching a terabyte in size, and provide insight on the data representation, cache behavior, and overall performance. The binning algorithm is able to achieve nearly optimal scaling on 8 sockets and an average performance speedup of 3.8× with irregular big-data matrices, when compared to an equally optimized version of the CSR algorithm. In terms of absolute performance our binning algorithm on an 8-node NUMA system is comparable to the best cluster algorithm on a 256-node cluster (to the best of our knowledge [5]). In Table 1 we provide a cursory analysis on the largest graph matrices used for SpMV in the literature. Our approach works with graphs with up to 68 billion edges, one to three orders of magnitude larger than the current state of the art, to the best of our knowledge.

The rest of the paper is organized as follows: Section 2 introduces the new class of algorithms and the rationale behind it, while Sections 3 and 4 present two algorithms that belong to this class. Section 5 presents the experimental evaluation of the algorithms, after an introduction on the target architecture and the datasets. Section 6 compares our work with the state of the art, and we conclude the paper in Section 7.

2. RE-DEFINING SPMV: MULTIPLE PHASES TO IMPROVE LOCALITY

We begin our analysis of cache-efficient SpMV algorithms by considering how SpMV is implemented using the CSR format, as shown in the pseudo-code of Algorithm 1.

Algorithm 1: Sequential CSR Algorithm.

Input: $A = (rowstart, colidx, val)$: $n \times n$ CSR matrix;
x: input vector.

Output: b: output vector, initialized to 0.

```
1 for i ← 0 to n − 1 do
2     for j ← rowstart[i] to rowstart[i + 1] − 1 do
3         k ← colidx[j];
4         b[i] ← b[i] + (val[j] ∗ x[k]);
```

Each row of A, indexed by *rowstart*, is stored contiguously in memory, and contributes to a different element of **b**. The column index *colidx* locates the elements of **x** corresponding to each matrix element. Scanning one row of A at a time has the benefit of writing **b** sequentially, ensuring its cacheability, and allowing the accumulation of each element b_i in a register. The downside is that the input vector **x** is accessed according to a sparse pattern defined by the column index. Also, each element of the matrix is accessed only once, limiting the opportunities for cache optimizations.

The techniques commonly employed in HPC matrices are not suitable for graphs, leading us to a different way of tackling the problem. Ideally, to make the best use of caching, we would like to visit the matrix by column when accessing **x**, and by row when accessing **b**. These two conflicting goals can be achieved by decomposing the problem into two separate visits.

Algorithm 2 shows the generic structure of a two-phase algorithm, assuming A is represented in augmented coordinate (COO) format. For each element, along with value and coordinates, we store a temporary value *temp*[*i*], to propagate the partial results from the first to the second phase.

Algorithm 2: Two-phase Algorithm on a COO matrix.

Input: $A = (rowidx, colidx, val, temp)$: COO matrix, with nnz elements;
x: input vector.

Output: b: output vector, initialized to 0.

```
1 for j ← 0 to nnz − 1 do
2     k ← colidx[j];
3     temp[j] ← val[j] ∗ x[k];
4 for j ← 0 to nnz − 1 do
5     i ← rowidx[j];
6     b[i] ← b[i] + temp[j];
```

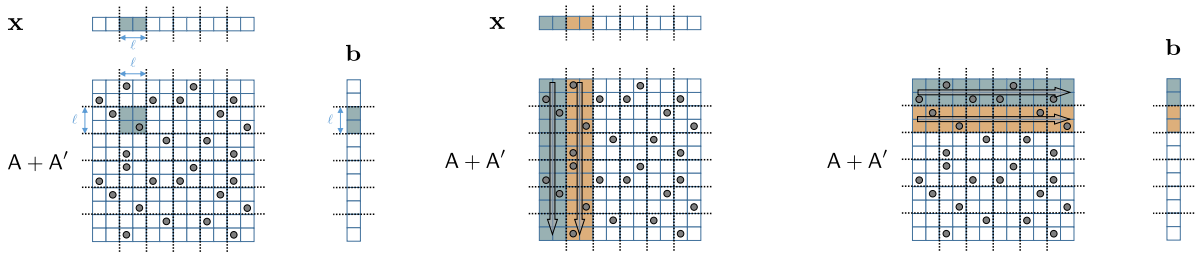
The first phase uses the rows of A and the corresponding elements of **x** to calculate the contributions of each a_{ij} to the output, storing them in a temporary vector. This phase scales the matrix, multiplying each column by the corresponding vector element, and producing A′ such that $a'_{ij} = a_{ij}x_j$. The second phase accumulates the individual contributions on the values of **b**, calculated as $b_i = \sum_{j=0}^{n-1} a'_{ij}$.

We can see how this restructuring of the code achieves the goal of acting only on the matrix and one of the vectors at a time. By using a proper representation of A, we can improve the locality of SpMV for both vectors, ensuring we transfer each data structure only *once* from the main memory, independently of the matrix structure. This removes the biggest source of performance degradation and variability of SpMV.

The price to pay is a larger memory footprint, caused by the need to store the temporary results. Because of the trade-off between cache locality and memory usage, we expect the scheme to work well in graph-like matrices, where CSR offers limited cache reuse on the input vector, and be outperformed for regular matrices, where CSR is already providing excellent performance. We call the class of algorithms that conform to the aforementioned pattern *Two-phase SpMV*.

3. TWO-PHASE SPMV WITH BLOCKING

To guarantee optimal cache reuse we need to organize the matrix memory layout carefully, and schedule the two visits



(a) Matrix Representation (b) First Phase (c) Second Phase

Figure 2: Graphical Representation of the Two-phase with Blocking algorithm. A and A' are stored as a single matrix. (a) highlights, for a block, the required parts of the vectors; ℓ is selected so that the vectors can fit in cache. (b) and (c) illustrate the first and second phase, respectively, including the required part of vectors for each column/row of blocks.

in a way that allows reuse on both vectors. At the same time, we want to avoid sparse access to the temporary values. Simply having one phase proceed by row and the other by column would not be enough. The first visit would scan the matrix and generate the scaled values by column. The second visit would then need to scan the scaled values by row. This would require either sparse reading in the second phase, or sparse writing in the first.

The idea behind the algorithm that we will call *Two-phase SpMV with Blocking* (TP-Blk) is to represent A and A' as a single augmented matrix, stored by block, as depicted in Figure 2a. We use square blocks of size $\ell \times \ell$. Each block requires at most ℓ elements from $\mathbf{x}$ and updates at most ℓ elements of $\mathbf{b}$ figure. The selection of ℓ determines which level of the cache hierarchy contains the vectors.

The first phase visits the matrix by columns of blocks, as depicted in Figure 2b. Blocks are scanned linearly, and each element is updated with the computed a'_{ij}. Each block may access several elements of $\mathbf{x}$, however, the blocks belonging to the same block-column share a common contiguous subset of $\mathbf{x}$, and ℓ is selected so that said subset is contained in cache. As a result, scanning a block-column requires fetching a subsection of $\mathbf{x}$ into the cache, and accessing it from there until we are ready to move to the next block-column. This causes a new subset of $\mathbf{x}$ to be fetched, allowing the cache to discard the old one, and so on for the rest of the phase. The vector $\mathbf{x}$ is read from memory only once, in consecutive sections, regardless the structure of the matrix.

The second phase uses the same approach with the output vector. In this case the algorithm scans the matrix by block-row, and sums the elements of each row. Again, because of the size of the blocks and the order in which they are accessed, at any given time only a subset of $\mathbf{b}$ has to be in cache. This second step is shown in Figure 2c.

The coordinated access by block limits the effects of the visit order on the caches, compared to accessing matrix and vectors by individual element. This has two main benefits:

- we can use a single copy of the blocks, while working directly on the elements would require two copies of the matrix (one stored by column and one by row), and
- by using the same visiting order inside the block, the access pattern to the temporary vector of each block is linear in both visits, consequently offering good locality and prefetching opportunities.

It should be noted that a sparse access to the temporary vector would have the same drawbacks of the sparse access we are trying to avoid on the input and output vectors.

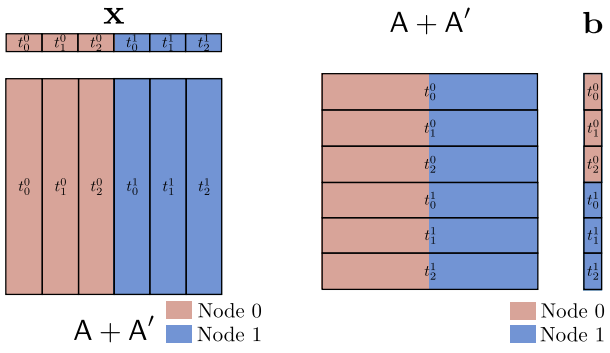
The block size is determined by the cache hierarchy: we want to make the blocks as large as possible, to allow as much sequential access (and, consequently, prefetching) as possible, but we don't want the working set on the vectors to ever exceed the cache size. Given a fixed value of ℓ, the matrix structure determines the sparsity of the blocks, and the efficiency of the access. Increasingly sparse blocks incur more storage overhead for information such as the position of the block in the matrix, the number of nonzeros, and the pointers to the data structures. Also, because the blocks are not stored contiguously, the last cache line of each data structure of the block is always read entirely, possibly causing extra accesses. The inefficiency of extremely sparse blocks is an issue with large scale-free graphs. Graphs in this class usually have a constant average degree[1], given by the modeled problem. This means that as a graph grows, it becomes increasingly sparser, and the expected number of elements in an $\ell \times \ell$ block decreases. For example, an R-MAT 24 has an average block size of $\sim 12,000$ elements, while with an R-MAT 31 the value drops to ~ 63. We expect performance to degrade in the presence of such extremely sparse blocks.

When blocks are hypersparse (i.e. have fewer nonzeros than the number of rows/columns), CSR-like formats become inconvenient to store the blocks. Considering the relatively small size of ℓ, we can use a limited number of bits for the indices, making a COO format very space efficient. Two bytes, corresponding to a maximum $\ell = 64k$ and allowing us to index sub-vectors of 512 KB, are enough for the blocks to fill up the cache. Each nonzero requires $2+2+8+8 = 20$ bytes, with the only source of overhead with respect to non-blocked CSR being the temporary value. A similar storage format was introduced, for different reasons (mainly reducing the matrix size), as CSB (Compressed Sparse Block) in [6].

3.1 Parallel Implementation

The two phases can be parallelized on a shared-memory system by assigning, respectively, a set of contiguous block-columns and block-rows to each thread. Changing the block assignment between phases creates a placement issue in non uniform memory architectures, such as POWER8. Since moving the entire matrix would generate too much traffic,

[1]Most scale-free graphs follow a power-law distribution with $2 < \gamma < 3$[24]



(a) First Phase (b) Second Phase

Figure 3: Graphical Representation of the partitioning for the TP-Blk. t_j^i represents thread j of node i.

a choice has to be made with respect to where to place the blocks, and during one of the phases threads will have to access some blocks stored on a remote socket. We decided to efficiently support the first phase, which is more memory-intensive. All the data used by a thread in the first phase, including the matrix and the vector, is stored in the local memory controller. Figures 3a and 3b show how the data structures are partitioned on a two-node system. As shown in Figure 3b, with this placement the second phase accesses remote blocks.

Load balancing the work assigned to each thread is made quite difficult by the different visit orders in the two phases, which may have contrasting requirements. Creating partitions based on the number of nonzeros per block may not be sufficient. Depending on the structure of the matrix, we may have an uneven distribution of nonzeros per block that, in turn, can create imbalance in the number of blocks per partition. A simple approach to distribute the load evenly is to randomly reorder the nodes of the graph, a procedure commonly known as "scrambling." While this is generally a bad idea, because it completely removes any locality the matrix structure could present, it does not affect the locality of our two-phase algorithm. On the other hand, a uniform distribution of nonzeros inside the matrix allows us to keep, on average, the same number of elements per block, and balance the computation.

4. TWO-PHASE SPMV WITH BINNING

In this section we present a second algorithm, of the Two-phase class, that overcomes most of the shortcomings of the previous one. The problems in TP-Blk stem from the use of blocking, specifically the low density of the blocks in very large graphs. To avoid this, we need an algorithm that, as opposed to scanning each block linearly and then jumping to another one, scans the entire matrix linearly in both phases. We already discussed how a double representation by rows and by columns of elements could easily work for the two visits, but would require to read or write sparsely the scaled values a'_{ij}. The idea of this algorithm is to improve the access pattern on A' by using a *binning* technique, hence the name *Two-phase SpMV with Binning* (TP-Bin).

The first phase scans the matrix by column; instead of saving the scaled values in the same order they appear in the original matrix, however, we group them in bins, as shown in Algorithm 3 and Figure 4a.

Algorithm 3: First phase with Binning on a COO matrix.

Input: $A = (rowidx, colidx, val)$: COO matrix, stored by columns;
x: input vector.

Output: *bins*: array of bins; a bin is an array of tuples;
size: array containing the size of each bin; Initialized to 0.

```
1 for j ← 0 to nnz − 1 do
2     k ← colidx[j];
3     tmp ← val[j] ∗ x[k];
4     bin ←FLOOR(rowidx[j]/m);
5     bins[bin][size[bin]] ←< tmp, rowidx[j] >;
6     size[bin] ← size[bin] + 1;
```

We want to bin the data in a way that can be stored efficiently in the first phase, and read efficiently during the second phase. We adopt the same concept of locality on **b** used before: elements belonging to a small subset of rows can be processed together as we can accumulate the results in cache. Similarly to the previous algorithm, in which we defined blocks of $\ell \times \ell$, we define bins that contain m contiguous rows of the matrix. We use a different letter because, in general, $m \neq \ell$, as we may want to exploit the cache hierarchy in a different way.

Binning is an efficient technique to reorder the elements from a column-wise storage to a partial row-wise storage. Inside each bin we write linearly, but subsequent elements of A will likely belong to different bins. Therefore, to mask the sparse access to the bins, we must keep the working set of each bin in cache. Because of the linear access inside each bin, this consists in a single cache line per bin, allowing us to work with a relatively large number of bins.

Once we binned the results (producing A') we can easily scan each bin and accumulate elements belonging to the same row. Because each bin contains data from multiple rows, we store each element as a tuple $< a'_{ij}, i >$ in the bin. The pseudocode in Algorithm 4 and Figure 4a show the second phase. By having a single bin that spans across all the columns of the matrix, we remove the problem we had on the blocked algorithm, making the number of elements per bin sufficiently large on scale-free graphs.

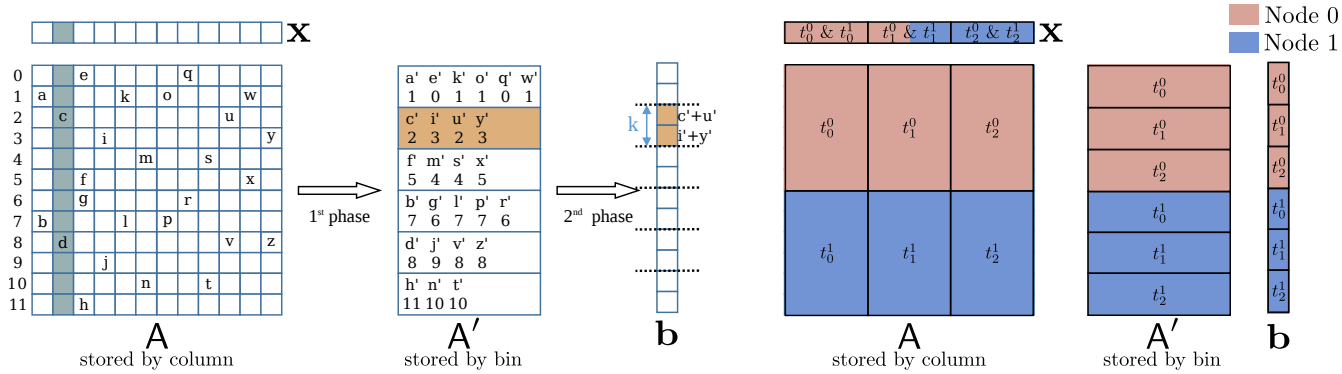
Algorithm 4: Second phase with binning on a binned A'.

Input: *bins*: array of bins; each bin is an array of tuples;
nbins: the number of bins;
size: array containing the size of each bin.

Output: b: output vector.

```
1 for h ← 0 to nbins do
2     for j ← 0 to size[h] do
3         < tmp, i >← bins[h][j];
4         b[i] ← b[i] + tmp;
```

4.1 Optimizations



(a) Graphical representation of the algorithm

(b) Partitioning on the parallel implementation

Figure 4: Two-phase SpMV with Binning. (a) represents the two phases of the algorithm. Inside A and A′ bins elements are visited following the alphabetical order. In the first phase elements of A are multiplied by the corresponding element of **x** and stored in A′ coupled with the row. In the second, elements of each A′ bin are accumulated on the corresponding element of **b**. (b) represents the partitioning, with t_j^i indicating thread j of node i.

Compared to the Two-phase SpMV with Blocking, with this algorithm we have to pay more attention to the optimization of the sequential code and to the effects on the cache hierarchy. Binning as described in Algorithm 3 is complex both in terms of computation and memory accesses. Computing the destination bin requires an integer division; however, by selecting power-of-two bin sizes we can compute the bin with a simpler bit-wise operation. Writing to the bins, on the other hand, heavily pounds on the caches, as we need to identify the correct offset inside the bin and increment the offset for the next write. In addition to creating more read/write traffic, the *size* array of Algorithm 3 should be kept in L1, to perform the increment efficiently, thus subtracting space to the bins.

In this specific case we can improve the binning because the destination bin depends only on the row index of the element. Since this is a property of the matrix, we can easily precompute the destination bin while converting the matrix, prior to the actual SpMV computation. More importantly, we can also assign a unique position inside the bin and save it in the matrix representation. This makes the first phase much simpler, both in terms of computation and cache traffic, as we only have to read a pointer corresponding the exact destination (i.e., the bin and the offset inside the bin) and write only the computed value (we can pre-store the row of each element when allocating the bins). Of course, the pointer will be read from memory, thus increasing the read traffic, but the advantages in terms of cache traffic and computation make it worthwhile. On the other hand, the cost of computing the pointer during the matrix conversion can be considered negligible, as it can be made on the fly with no additional scans of the matrix. Furthermore, since SpMV is generally used in iterative algorithms, we amortize the conversion cost.

With respect to TP-Blk, in this case we need two different representations of the matrix: one, stored by columns, for A, and one, stored by bins, for A′. The amount of space, however, can be limited because most of the data required in the first phase is not needed on the second and vice versa. In particular, the representation for A will need, for each element: a) the value (a_{ij}), b) the column (j), and c) the write pointer for a'_{ij}. On the other hand, for A′ we will just need: a) the value (a'_{ij}), and b) the row (i). In comparison with the TP-Blk, we are only adding the write pointer to the matrix representation.

4.2 Tuning the Bin Size

Binning is very sensitive to the presence of caches. If we are not able to keep the bins in cache, the performance will decrease sensibly. Bin characteristics have also strong implications on the maximum size of the matrix: to optimize the first phase we need to keep the number of bins under control, while to optimize the second we need to keep the size (i.e. number of rows) of each bin under a certain threshold. The two requirements together limit the maximum amount of rows we can efficiently support. For this reason we developed two micro-benchmarks to analyze the limits of this algorithm. The micro-benchmarks are made to mimic the behavior of the two phases, using random data.

The first benchmark tests the maximum number of bins we can support in the first phase. To this end we create an environment to mimic the phase: we have an array of pairs $< element, pointer >$. The pointer represents the precomputed destination in a randomly selected bin. We scan this array of tuples and write the element in the corresponding destination. In this microbenchmark we do not simulate the multiplication of the elements by the vector, because it should not affect the threshold; in other words, the results in Table 2 do not show the real performance of the first phase, but are sufficient to determine the maximum amount of bins we can efficiently support.

The performance we get is steady up to a certain size, and using multiple threads does not improve the performance. The resulting threshold to achieve maximum performance correspond to $16k$ bins. In this case the working set for each bin is 128 bytes (a cache line), meaning that we can use up to $16k * 128 = 2\ MB$ of cache and still obtain peak performance. Using a cache area so large is possible because in this case we are only writing to the selected location. Since we are not limited by the write latency, but only by the bandwidth, any level of the cache hierarchy is fine, so long as we do not have to go back to the memory.

The second benchmark tests the maximum size allowed for a single bin, modelling the second phase. We create a

Bins \ Threads	Performance (GFlop/s) 1	2	4	8
512	**0.47**	**0.42**	**0.42**	**0.41**
1024	**0.47**	**0.42**	**0.44**	**0.42**
2048	**0.47**	**0.44**	**0.44**	**0.42**
4096	**0.47**	**0.46**	**0.44**	0.35
8192	**0.48**	**0.45**	0.37	0.14
16384	**0.46**	0.39	0.14	0.07
32768	0.40	0.14	0.07	0.05

Table 2: First phase *µbench* results for different bin counts.

Rows \ Threads	Performance (GFlop/s) 1	2	4	8
512	**0.94**	**0.93**	**0.92**	0.70
1024	**0.96**	**0.94**	**0.91**	0.68
2048	**0.96**	**0.91**	0.70	0.55
4096	**0.93**	0.72	0.49	0.51
8192	0.68	0.41	0.44	0.47
16384	0.42	0.33	0.42	0.41
32768	0.33	0.30	0.39	0.38

Table 3: Second phase *µbench* results for different bin sizes.

long array of floating point values and pair each element with a random number selected in a small interval. The random number (similarly to i in Algorithm 4) identifies the location where the element will be added. By increasing the interval of the random numbers we simulate larger bins, to find the optimum value of m for the second phase. The results in Table 3 show that the performance of the second phase should be steady when each bin has up to 4096 rows. After this threshold, the performance constantly decrease. The number is not a coincidence: accumulating over 4k rows produces random access over $4096 * 8 = 32\ KB$, half of the L1 cache of a POWER8 core. Up to this size we are able to keep the entire area in L1, resulting in a very low latency access, while larger bins do not fit in L1, increasing the latency of each access. In this phase we are accumulating (i.e., performing a sequence of load, add, store), therefore keeping the latency low is of paramount importance. The result is confirmed by using multiple threads per core, that do not increase the performance and, in addition, lower the maximum number of rows because the cache is shared by multiple threads.

To summarize we can support, at full speed, up to 16384 bins each with 4096 rows. In other words, each POWER8 core can compute a sequential TP-Bin on a matrix with 67 million rows, corresponding to an R-MAT 26.

4.3 Parallel Implementation

Since each bin must be stored contiguously in memory, we cannot afford a solution like the one used in the TP-Blk, in which we would have to write remotely. Ideally, we would like to be able to let all the data accessed in both phases to be local. Unfortunately, no partitioning is able to guarantee that. We can, however, partition the data so that the matrices A and A' are kept local, and let the vectors be accessed remotely.

We perform a horizontal 1d partitioning of the matrices between nodes. Inside each node we partition A by columns for first phase. Each thread scans its partition and a subset of the input vector, and writes the data in bins. Because bins contain groups of rows, we have multiple threads on the same socket writing concurrently to the same bins, *but on different parts*. In the second phase, each thread scans its own partition, a set of contiguous bins, and accumulate the results. The partitioning is depicted in Figure 4b. With this partitioning each node needs to access to the entire input vector $\mathbf{x}$ (as opposed to a single visit for the entire server for the previous algorithm), thus lowering the reuse on $\mathbf{x}$. On the other hand, all the matrix data is kept local.

As with TP-Blk, we perform a "scrambling" of the matrix to uniformly distribute elements among bins.

Regarding the matrix size limit, we have that each thread bins data in the entire node partition. Since all the threads on a node "share" the same partition (even if they work on different areas), the previously introduced per-core bound becomes a per-node bound: each POWER8 chip is able to compute up 67 Million rows. On the other hand, different nodes have completely independent partitions. Therefore, we have that a server with ns sockets will support up to $ns*67M$ rows, making the algorithm scalable on the number of nodes (sockets) of the system.

If the matrix is larger than the supported number of rows, we can adjust the number of bins by assigning multiple "partitions" to a single node, and process one at a time. This, however, increases the traffic on the vector, because each partition requires the entire input vector, resulting in multiple reads of the vector for each node.

5. PERFORMANCE RESULTS

5.1 Overview of POWER8

Before analyzing the experimental results, we give a brief overview of the IBM POWER8, the latest RISC microprocessor from IBM, and the architecture we used in our study. Of particular interest to this paper are its cache and memory subsystems. The cache subsystem of POWER8 consists of four levels. Each POWER8 core owns a store-through L1 data cache, a store-in L2 cache, and an eDRAM based L3 cache with a capacity of 64 KB, 512 KB, and 8 MB, respectively. The L3 caches of POWER8 have a NUCA (Non-Uniform Cache Architecture) design, with each L3 also serving requests for other cores, and working as a victim cache for other L3s [26]. Additionally, the POWER8 processor is connected to a maximum of eight external memory buffer chips called Centaur. Each Centaur chip contains 16 MB of eDRAM and serves as the fourth cache level.

Centaur chips also function as the memory controllers of POWER8. By moving the memory controllers to Centaur, the memory bandwidth and capacity of POWER8 are significantly improved compared to its predecessor, without affecting the memory access latency. A different design point of POWER8 is that Centaur chips have independent channels for memory reads and writes, with a per-chip bandwidth of 19.2 GB/s and 9.6 GB/s, respectively. Each POWER8 processor can be connected to up to eight Centaur chips, offering a maximum 230 GB/s of raw memory bandwidth (154 GB/s read and 76 GB/s write).

5.2 Testing Environment

We study the performance of SpMV using both HPC and graph matrices. The HPC matrices we use are selected

from the University of Florida Sparse Matrix Collection [11], and were used in previous studies on SpMV [22, 30]. With various sizes, numerical properties and sparsity structures, these HPC matrices represent a wide range of applications. The graph matrices we use include two real-world graphs from the Stanford Large Network Dataset Collection [21] and synthetic matrices of various sizes. R-MAT is generated with the following parameters: $a = 0.57, b = 0.19, c = 0.19, d = 0.05, ef = 16$. For BTER we compute α and δ using the reference code[20]; the other parameters are $cmax = 0.5, gcc = 0.15, ef = 16$. Both generators produce undirected edges. Table 4 summarizes our test matrices and their properties.

The parallel CSR used as a baseline exploits a static 1D partitioning to assign a group of contiguous rows to the same thread and balance the number of nonzero per partition. Each thread keeps its own partition (consisting in a set of rows of A and elements of $\mathbf{b}$) on the nearest memory controller, to minimize the amount of chip-to-chip communications. Each thread also requires access to the entire input vector. Distributing the vector across the machine would significantly lower the read bandwidth; to improve the performance, we keep a replica of $\mathbf{x}$ on each memory controller. In comparison, the Two-Step algorithms do not require this replication, making them a better option for iterative algorithms.

Performance results were measured on a 8-socket SMP system, the IBM Power E870 System (E870), with each socket containing an 8-core POWER8 processor running at 4.35 GHz. To compare the memory transfers of different SpMV implementations, we used a 2-socket server, the IBM Power S824L system (S824L). Each processor of S824L contains 12 cores running at 3.3 GHz. Table 5 lists some important characteristics of E870 and S824L. The memory bandwidth was measured using the STREAM benchmark.

Our SpMV implementations were written in standard C99 and compiled with gcc v4.8.4 using –O3 optimization. SpMV kernels were optimized using inline assembly.

All the performance results presented are computed by averaging the results of five executions, each performing ten SpMV iterations. GFlop/s are computed considering the two floating point operations per element (nnz) of the original SpMV kernel. The number of threads used depends on the algorithm. For CSR, we present the best result among 1, 2 or 4 threads/core. For TP-Blk, we use sparse blocks of $64k \times 64k$ elements, and present the best result among 2 and 4 threads/core. TP-Bin uses bins of 4096 rows, a maximum of $16k$ bins, and 1 thread/core. The Two-Step algorithms always use the scrambled version to improve load balancing. For CSR, on the other end, where it is more important to maintain, as much as possible, the locality properties of the matrix, we use the original (unscrambled) version of the matrix.

5.3 Memory Analysis

We start the analysis with a study on the matrix memory consumption for the Two-phase formats, compared to CSR and other well-know formats. The results are reported in Table 6. We report the matrix size for CSR in GB, while for the other formats we express the matrix size as $\frac{Size}{CSR\ Size}$, to show the overhead we incur w.r.t. our baseline. We also project, based on the matrix characteristics, the space needed for two common formats: BSR[14] and ELL[4]. These are implemented, together with non-blocked algorithms such as CSR, in several open-source and vendor-optimized libraries. In particular, the first is implemented in Intel MKL[15], Nvidia cuSPARSE[23] and OSKI/pOSKI[30, 16], while the second is used in Intel MKL[15], cuSPARSE[23] and SPARSKIT[25].

Name	Rows - Cols	nnz	nnz / row μ	nnz / row Max
Dense	17K	293M	17K	17K
HPCG	10M	275M	26.91	27
Wind Tunnel	218K	5.9M	27.1	180
Rail4284	1M	11.2M	10.28	56K
Mip1	66K	5.2M	78.4	66K
Si41Ge41H72	185K	7.6M	41	662
Ldoor	952K	46M	48.85	77
Bone010	986K	71M	72.63	77
12month1	872K	22M	25.92	75K
Spal_004	321K	46M	143.51	6K
Crankseg2	63K	7M	111.32	3423
Nlpkkt240	27M	774M	27.66	28
Friendster	124M	3.6B	28.9	5.2K
Orkut	3M	234M	76.3	33K
R-MAT 22-31	4M-2B	13M-68B	32	N/A
BTER 22-31	4M-2B	13M-68B	32	N/A

Table 4: Test matrices used in performance evaluation.

	E870	S824L
Frequency	4.35 GHz	3.30 GHz
Cores	64	24
Memory	4 TB	512 GB
Total L3 Cache	512 MB	192 MB
Local BW (Read)	1,141 GB/s	219 GB/s
Local BW (Write)	589 GB/s	117 GB/s

Table 5: Characteristics of E870 and S824L.

BSR works by creating small dense $k \times k$ blocks, to improve reuse on the input vector and vectorize the SpMV code. For our projection we selected 4×4 blocks. While this tend to work well on HPC matrices, where points are *clustered*, the overhead for graphs is too big, ranging from $\sim 11\times$ to $\sim 4\times$.

In ELL the matrix is represented in a dense format, with as many rows as the original matrix and as many columns as the largest row. Rows with fewer elements are padded with zeros. Unfortunately, this approach does not work with graphs, whose degree distribution follows a power law: the space required will be given by the single very large row, and all the others will be padded with zeros.

Our Two-phase algorithms, on the other hand, show a much better behavior, with both formats requiring less than two times the memory of CSR. We can also appreciate the "stability" of our results: given the structure-independence of our formats, the overhead is the same for any matrix.

To make sure the Two-phase algorithms behave as expected, we also measured the amount of memory transferred during the computation, and compared it to the theoretical bounds. Since the poor performance of CSR SpMV on graph matrices is mainly due to excessive memory reads from the vector $\mathbf{x}$, our analysis will focus on the read traffic. Memory transfers are computed measuring the number of cache miss with the PAPI library; the theoretical bounds assume the vector $\mathbf{x}$ is accessed only once (i.e. maximum locality).

	CSR (GB)	BSR	ELL	TP-Blk	TP-Bin
			(Size / CSR Size)		
Dense	3.28	0.71	1	1.67	2
Nlpkkt240	8.86	1.89	0.99	1.63	1.95
HPCG	3.15	2.16	0.98	1.63	1.95
Bone010	0.81	1.2	1.11	1.65	1.98
Ldoor	0.53	1.18	1.55	1.64	1.97
Crankseg2	0.16	1.35	15.4	1.66	1.99
Spal_004	0.52	2.7	41.82	1.66	1.99
Si41Ge41H72	0.17	2.32	8.12	1.65	1.98
Mip1	0.12	0.86	424.43	1.66	1.99
Wind Tunnel	0.13	1	3.33	1.65	1.98
12month1	0.26	7.04	2833.53	1.46	1.75
Rail4284	0.13	3.33	5128.95	1.57	1.88
R-MAT 22	1.47	10.01	5204.62	1.63	1.96
R-MAT 31	778.23	10.87	286824.02	1.74	1.96
BTER 22	1.53	4.07	865.55	1.63	1.96
BTER 31	784.03	4.14	4288.08	1.73	1.96
Friendster	41.3	11.08	176.14	1.64	1.95
Orkut	2.64	10.26	432.95	1.65	1.98

Table 6: Matrix memory usage for different formats. For CSR is expressed in GB, for the other formats as a ratio. < 1 means less space than CSR, > 1 means more.

The computation on each nonzero requires reading its row and column indexes (2 bytes each), one value from the original matrix (8 bytes) and one value from the temporary matrix (8 bytes). In addition, TP-Bin requires reading one write pointer (4 bytes). The contribution of the vector to each element is computed as[2] $8\ n/nnz$. Thus, the lower bound of the memory reads per nonzero for TP-Blk can be calculated as $20+8\ n/nnz$, and for TP-Bin as $24+8\ n/nnz$.

Figure 5 presents the results for the proposed algorithms, showing that, for most of the test matrices, the measured memory transfers of the Two-phase algorithms match the theoretical bounds. For very large graph matrices, the measured amount of memory transferred by TP-Blk is higher than the theoretical bounds. As discussed in Sect. 3, this is because the number of nonzeros per block decreases as the graph scale increases. On the other hand, the amount of memory transferred by TP-Bin is more stable, though it is larger than in the TP-Blk case.

For the sake of comparison we also report the total transfers of CSR in Table 7. As expected, these are higher for graphs, with an average of $55B/nnz$, making our algorithms better suited for this class of matrices. On the other hand, as expected, HPC matrices show a good reuse on the vector with CSR, producing smaller transfers compared to our Two-phase algorithms (average of $12.86B/nnz$).

HPC Matrices		Graph Matrices	
Name	B/nnz	Name	B/nnz
Dense	12.00	Friendster	122.38
HPCG	13.19	Orkut	24.20
Wind Tunnel	12.32	R-MAT 22	23.97
Rail4284	16.15	R-MAT 23	38.69
Mip1	12.14	R-MAT 24	53.08
Si41Ge41H72	12.27	R-MAT 25	69.08
Ldoor	12.59	R-MAT 26	83.79
Bone010	12.35	R-MAT 27	98.39
12month1	13.47	R-MAT 28	108.72
Spal_004	12.35	R-MAT 29	117.32
Crankseg_2	12.16	BTER 22	17.03
Nlpkkt240	13.44	BTER 23	20.62
		BTER 24	24.80
		BTER 25	29.36
		BTER 26	33.69
		BTER 27	37.66
		BTER 28	41.20
		BTER 29	44.04

Table 7: Memory transferred by SpMV with CSR.

5.4 Strong Scaling Results

Figure 6 shows the strong scaling results of the Two-phase algorithms compared to CSR on three graph matrices. The experimental results were measured on E870 with various numbers of sockets. In the figure, "Ideal" cases (dotted lines) represent the optimal performance (in GFlop/s) assuming each SpMV kernel scales perfectly among nodes.

The results show that the Two-phase algorithms have a significant advantage over CSR in terms of both performance and scalability. TP-Bin has the best performance among all the three algorithms. The scaling of TP-Blk is poorer than TP-Bin when more than 4 sockets are used. This is because TP-Blk requires more remote memory accesses than TP-Bin, and the cost of accessing remote memory increases as the number of sockets increases. This result suggests that TP-Bin is a better choice for large-scale NUMA architectures.

5.5 Performance Comparison

We now compare the performance of the Two-phase algorithms with CSR on E870 using 8 sockets. Figure 7 and 8 show the results on HPC and graph matrices, respectively.

The results show that CSR outperforms the Two-phase algorithms on most HPC matrices, except for *12month1* and *Rail4284*. This is expected since HPC matrices have relatively low vector I/Os and the Two-phase algorithms require more memory transfers on the matrix than CSR. The two matrices on which the Two-phase algorithms performs better, on the other hand, are the ones with the highest vector

[2]Total size of the vector divided by the number of elements in the matrix. n is the number of rows, nnz the number of non zero elements of the matrix

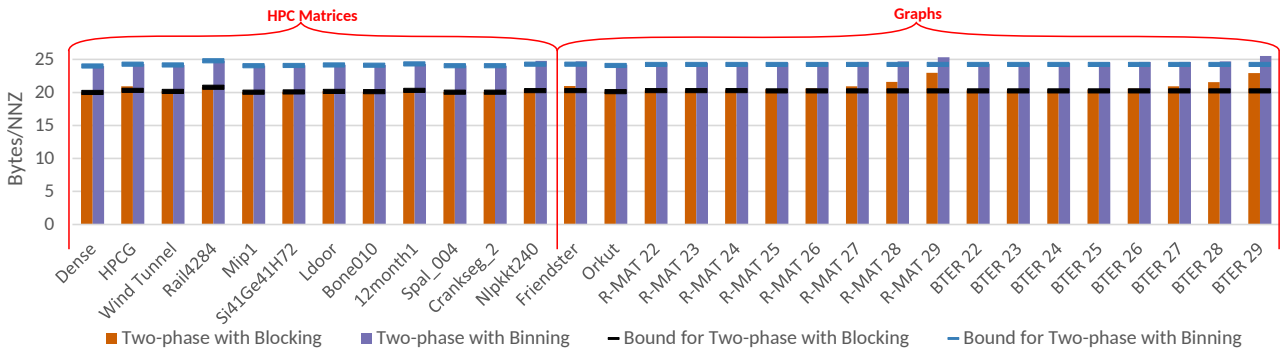
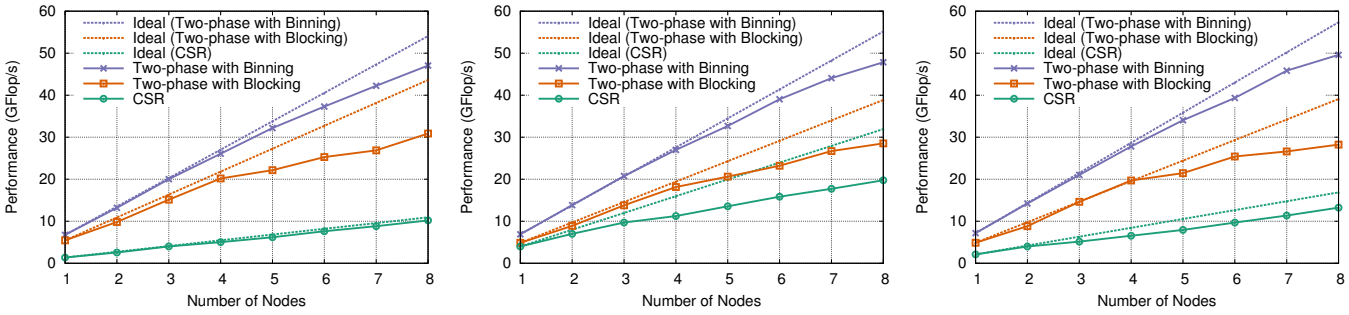


Figure 5: Memory transferred by SpMV with the Two-phase algorithms



(a) R-MAT 24 (b) BTER 24 (c) Orkut

Figure 6: Performance of SpMV with different algorithms when varying the number of NUMA nodes (sockets). Dotted lines show the theoretical performance assuming perfect scaling for each algorithm.

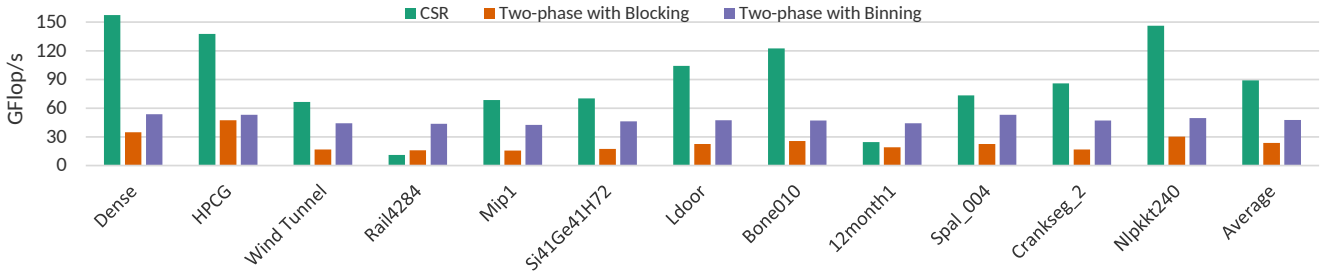


Figure 7: Performance comparison of the Two-phase algorithms and CSR on HPC matrices.

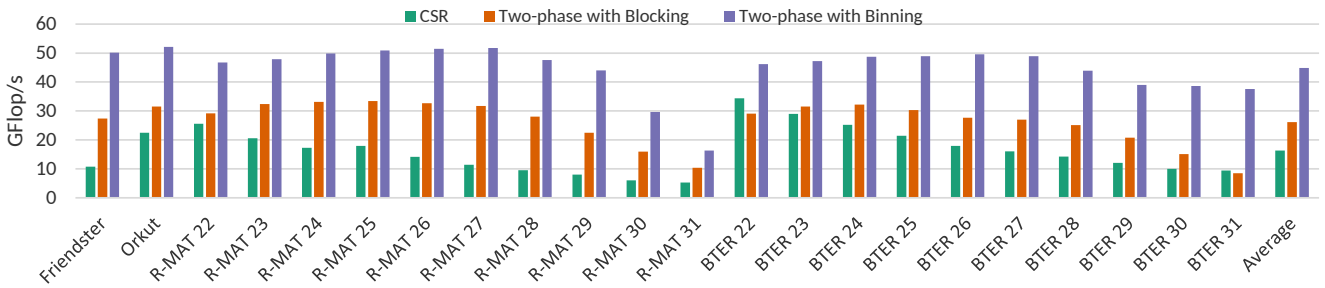


Figure 8: Performance comparison of the Two-phase algorithms and CSR on Graph matrices.

I/Os.

For graph matrices, on which the performance of CSR suffers from excessive vector I/Os, our Two-phase algorithms show a large advantage. On average, TP-Blk and TP-Bin are respectively 2× and 3.8× faster than CSR on large graphs (more than 2 billion edges). TP-Bin always outperforms TP-Blk, with a larger advantage on very large graphs. This is mainly because of two reasons. First, TP-Bin has better scalability than TP-Blk. Second, larger graphs lead to sparser blocks in TP-Blk, and TP-Blk performs poorly on very sparse blocks (see Section 3).

The results also show that the performance of the Two-phase algorithms is more stable than that of CSR. For example, the performance of TP-Bin varies from 42 to 53 GFlop/s, depending on the scale of the graphs, while that of CSR varies from 11 to 157 GFlop/s.

Finally, we observe that the performance of the Two-phase algorithms starts to drop when the scale is larger than 28. For TP-Blk, this is because larger graphs have sparser blocks, leading to lower performance. For TP-Bin, this is because the largest scale that a single binning partition can handle on POWER8 is 29 (see Section 4). When the scale is larger than 29, multiple partitions are required for each socket. As a result, the vector I/O increases (the computation on each partition requires reading $\mathbf{x}$ once), leading to performance degradation with TP-Bin.

6. RELATED WORK

SpMV has been studied extensively on various architectures over the past few decades. For a comprehensive review of CPU-oriented optimizations, we refer to survey papers such as [13], [14], [29] and [30]. One of the recurring themes in previous work is blocking. However, in most cases it has been applied in techniques targeting regular matrices, which prove inefficient for scale-free graphs, especially in terms of storage. We already discussed that BSR[14] and ELLPACK[4] are not suitable. Some research papers, such as BELLPACK[10], present formats derived from ELLPACK in which the rows of the matrix are sorted by row size. These can efficiently remove the padding on graphs and may improve locality, at the cost of sorting the rows. Considering the graph sizes we tackle, we prefer to avoid this costly step.

More advanced forms of blocking, to explicitly improve cache locality, are presented in [1] and [32]. Both papers exploit hypergraph partitioners to split the matrix in a set of blocks, determined to optimize the locality on both the input and the output vectors. While both papers show promising results (i.e. improved cache miss ratio), the performance improvement is generally low, and further reduced by the high cost of the partitioning step.

In a recent work, Anh et al. [2] identify the problem of the sparse vector access in SpMV. However, the authors target GPU architectures, and the approach they propose is not directly applicable to CPU systems. Ashari et al. [3] study an SpMV kernel optimized for graphs on GPUs, focusing mostly on load-balancing aspects, with no explicit effort in reducing the vector traffic. Tang et al. [27] specifically optimize SpMV for graphs (scale-free matrices) on a many-core coprocessor. The authors also show the large difference in performance between regular HPC matrices and graphs. However, their work is focused on improving vector unit utilization and load balancing. Finally, SpMV on graphs is also addressed in [31] and [5] for distributed environments. The first presents a 2d partitioning and motivate its efficiency w.r.t. 1d partitioning of graphs, that produce a much higher memory traffic. The second exploit graph partitioning algorithms to reorder and reorganize the graph in order to limit the communication between nodes. The 2d partitioning requires multiple phases of the algorithm, in a way resembling our approach. The motivations are also similar, as the communication on the distributed approach is required to exchange the input vector. Nevertheless, our approach focuses on optimizing the reuse of cache lines, and thus sensibly differ in the way data is exchanged. In [31], each node broadcast its partition of $\mathbf{x}$ in a specific phase of the algorithm, while with our Two-phase algorithms there are no explicity communications. Compared to [5], our Two-phase algorithms do not require a graph partitioning algorithm do distribute the data among nodes.

7. CONCLUSIONS

In this paper we address the performance problems of SpMV on graphs. To this end, we analyzed multiple algorithms on graphs with billions of nodes and edges, an order of magnitude larger than in any other work of which we are aware. We show that classic approaches, such as the CSR algorithm, are unable to provide good performance. We motivate the performance degradation as a memory transfer problem on the input vector of the multiplication, and introduce a new class of algorithms, able to deal with such shortcoming. We present two different algorithms, each using different techniques to improve the memory transfers at the cost of reading the matrix twice. Both algorithms optimize memory traffic and reduce the memory transfers up to 9× on very large graphs. Among the two, one is showing excellent scalability on NUMA systems and provides an average performance improvement of 3.8× on large graphs.

8. REFERENCES

[1] K. Akbudak, E. Kayaaslan, and C. Aykanat. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM Journal on Scientific Computing*, 35(3):C237–C262, 2013.

[2] P. N. Q. Anh, R. Fan, and Y. Wen. Reducing vector i/o for faster gpu sparse matrix-vector multiplication. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1043–1052, May 2015.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, Piscataway, NJ, USA, 2014. IEEE Press.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. ACM/IEEE Conf. Supercomputing*, SC '09, page 18. ACM, 2009.

[5] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis, SC '13, pages 50:1–50:12, New York, NY, USA, 2013. ACM.

[6] A. Buluç, J. Fineman, M. Frigo, J. Gilbert, and C. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proc. 21th Annual Symposium Parallelism in Algorithms and Architectures*, pages 233–244. ACM, 2009.

[7] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011.

[8] D. Buono, J. Gunnels, X. Que, F. Checconi, F. Petrini, T.-C. Tuan, and C. Long. Optimizing sparse linear algebra for large-scale graph analytics. *Computer*, 48(8):26–34, Aug 2015.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM Data Mining*, volume 4, pages 442–446. SIAM, 2004.

[10] J. Choi, A. Singh, and R. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Notices*, volume 45, pages 115–126. ACM, 2010.

[11] T. Davis. The University of Florida sparse matrix collection. In *NA digest*. Citeseer, 1994.

[12] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[13] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *J. Supercomput.*, 50:36–77, 2009.

[14] E. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *Intl J. High Perf. Comput. Appl.*, 18:135–158, 2004.

[15] Intel Corporation. Math kernel library: http://software.intel.com/en-us/articles/intel-mkl.

[16] A. Jain. *pOSKI: An extensible autotuning framework to perform optimized SpMVs on multicore architectures*. 2009.

[17] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating pagerank computations. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 261–270, New York, NY, USA, 2003. ACM.

[18] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, Sept. 1999.

[20] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, September 2014.

[21] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[23] NVIDIA Corporation. cusparse library (included in cuda toolkit): https://developer.nvidia.com/cusparse.

[24] J.-P. Onnela, J. SaramÃd'ki, J. HyvÃűnen, G. SzabÃş, D. Lazer, K. Kaski, J. KertÃl'sz, and A.-L. BarabÃąsi. Structure and tie strengths in mobile communication networks. *Proceedings of the National Academy of Sciences*, 104(18):7332–7336, 2007.

[25] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[26] W. Starke, J. Stuecheli, D. Daly, J. Dodson, F. Auernhammer, P. Sagmeister, G. Guthrie, C. Marino, M. Siegel, and B. Blaner. The cache and memory subsystems of the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):3:1–3:13, Jan 2015.

[27] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 136–145, Washington, DC, USA, 2015. IEEE Computer Society.

[28] H. Tong, C. Faloutsos, and J.-Y. Pan. Random walk with restart: Fast solutions and applications. *Knowl. Inf. Syst.*, 14(3):327–346, Mar. 2008.

[29] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, Univ. of California, Berkeley, 2003.

[30] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conf. Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[31] A. Yoo, A. H. Baker, R. Pearce, and V. E. Henson. A scalable eigensolver for large scale-free graphs using 2d graph partitioning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 63:1–63:11, New York, NY, USA, 2011. ACM.

[32] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrixâĂŞvector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.