Balancing Computation and Communication in Distributed Sparse Matrix-Vector Multiplication

Hongli Mi, Xiangrui Yu, Xiaosong Yu, Shuangyuan Wu and Weifeng Liu
Super Scientific Software Laboratory, China University of Petroleum-Beijing, China
{2020011713, 2019011724}@student.cup.edu.cn, 17326808842@163.com,{wsy, weifeng.liu}@cup.edu.cn

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is a fundamental operation in a number of scientific and engineering problems. When the sparse matrices processed are large enough, distributed memory systems should be used to accelerate SpMV. At present, the optimization techniques for distributed SpMV mainly focus on reordering through graph or hypergraph partitioning. However, although the reordering could reduce the amount of communications in general, there are still load balancing challenges in computations and communications on distributed platforms that are not well addressed.**

In this paper, we propose two strategies to optimize SpMV on distributed clusters: (1) resizing the number of row blocks on the nodes for balancing the amount of computations, and (2) adjusting the column number of the diagonal blocks for balancing tasks and reducing communications among compute nodes. The experimental results show that compared with the classic distributed SpMV implementation and its variant reordered with graph partitioning, our algorithm achieves on average 77.20x and 5.18x (up to 460.52x and 27.50x) speedups, respectively. Also, our method bring on average 19.56x (up to 48.49x) speedup over a recently proposed hybrid distributed SpMV algorithm. In addition, our algorithm achieves obviously better scalability over these existing distributed SpMV methods.

***Index Terms*—Distributed memory system, sparse matrix-vector multiplication, load balancing**

I. Introduction

Sparse matrix-vector multiplication (SpMV) multiplies a sparse matrix A with a dense vector x to get a resulting dense vector y. It is one of the most fundamental routines of many applications in computational science and engineering, and may be the most studied kernel in sparse basic linear algebra subprograms (BLAS) [1].

On shared memory systems like multi-core CPUs and many-core GPUs, accelerating SpMV received much attention, and a number of techniques such as vectorization [2]–[4], blocking [5]–[7], load balancing [8]–[10] and auto-tuning [11]–[13], have been proposed. Despite their effectiveness, when the input matrix A is large enough, a distributed memory system has to be used, and then the techniques for shared memory machines could not be directly migrated to communication-centric distributed computing pattern.

In distributed SpMV, each compute node of a cluster locally stores a part of A and x, and generates a part of y. When the components of x are not in the local storage, communications will occur, the requests of loading x from the other nodes will be sent, and the corresponding components of x will be received. Thereafter, the local SpMV computations will be completed.

As can be analyzed, the performance of distributed SpMV is generally limited by the overheads of communications and local SpMV computations. Furthermore, since the distribution of the nonzeros of sparse matrices can be very irregular, the communication pattern will easily bring a large amount of communications between nodes, and lead to poor scalability. To address the performance challenges, researchers proposed a number of techniques to optimize distributed SpMV, including various matrix partitioning algorithms [14]–[17], tuning schemes for distributed characteristics [18], and computation and communication overlap [19], [20].

However, despite the above efforts, it is worth noting that distributed SpMV still faces challenges on load imbalance, and the problem can not be resolved merely through row/column reordering using graph partitioning [21], [22]. Firstly, when the matrix is divided and stored in each node, the irregularity and diversity of the distribution of nonzeros on sparse matrices generally lead to imbalanced computations. However, graph partitioning will not change the number of nonzeros in each row or column, and thus the pattern of the sparsity is not naturally ready for balanced computations and communications. Secondly, it should be noticed that each node can store a row block, but the amount of local computations (i.e., the number of nonzeros covered by the columns of diagonal block) is not directly given by matrix partitioning, which is another risk leading to load imbalance. In particular, when the number of nodes increases, the performance of distributed SpMV may largely degrade due to the computation and communication imbalance.

To address the above mentioned challenges of distributed SpMV, we in this paper propose an optimization algorithm named `DistSpMV_Balanced` for distributed SpMV. The main target of this algorithm is to balance the computations and communications between compute nodes, and thus to improve the performance and scalability of distributed SpMV. Firstly, a reordering through graph partitioning is still used for providing a basic nonzero distribution to reduce the network traffic. Secondly, the size of diagonal sub-matrix (i.e., its number of rows and columns) stored locally is analyzed according to computations and communications, and then corresponding numbers of rows and columns are decided. Thirdly, a two-level parallelism of MPI+OpenMP is exploited for better fine-grained cache locality within core-groups of nodes.

In our experiments on a 256-core CPU cluster, our `DistSpMV_Balanced` algoritm proposed is compared with three distributed SpMV methods: (1) one called `DistSpMV` on top of naïve nonzero distribution, (2) the second one called `DistSpMV_Reordered` running on the matrix preprocessed through the graph partitioning tool METIS developed by Karypis and Kumar [21], and (3) the third one called `DistSpMV_hybrid` developed by Page and Kogge [23]. By using 20 representative matrices from the SuiteSparse Matrix Collection (previously known as the University of Florida Sparse Matrix Collection) [24] as the benchmark suite, the experimental results show that our `DistSpMV_Balanced` achieves on average 77.20x, 5.18x and 19.56x (up to 460.52x, 27.50x, and 48.49x) speedups over `DistSpMV`, `DistSpMV_Reordered` and `DistSpMV_hybrid`, respectively. The performance data on various number of processes/threads also demonstrate that our method has obvious better scalability.

This work makes the following contributions:

- We identify that matrix reordering techniques are not adequate to achieve good computation and communication balancing, and thus more schemes are required.
- We design an algorithm called `DistSpMV_Balanced` that reorganizes the distribution of sparse matrix on compute nodes for balanced computation and communication.
- We evaluate the new algorithm by using 20 representative sparse matrices on a 256-core cluster, and bring significant speedups over existing work.

II. Background and Motivation

A. SpMV

The SpMV operation multiplies a sparse matrix A with a dense vector x to produce a dense vector y. Figure 1 shows a simple example of SpMV. In this case, y_i is calculated by taking the dot product of A_i (row i of A) with the vector x. It is easy to see that there are no dependencies between rows throughout the execution. Therefore, SpMV can be paralleized through dividing the matrix into many row blocks on modern processors such as CPUs and GPUs.

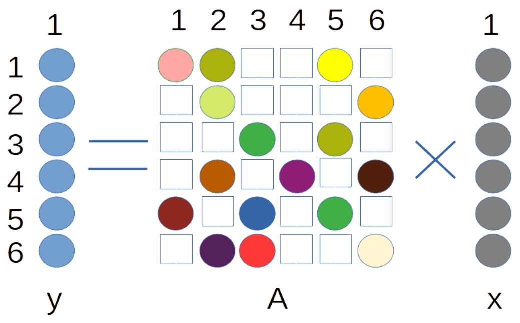


Fig. 1: An example of SpMV that multiplies a 6-by-6 sparse matrix with a dense vector and gets a dense vector.

B. Distributed SpMV

In distributed SpMV, part of the matrix A and two vectors x and y are allocated in the corresponding compute nodes of a cluster. A variety of division methods can generate different sub-matrices and sub-vectors, and communications between the nodes occur when running SpMV in the distributed environment. After receiving the components of x required, each process can calculate SpMV in parallel, and finally the resulting vector y is generated and communicated if needed. The following procedure lists the steps:

1) Send sub-matrices and sub-vectors to each process.
2) Processes communicate with each other, each sending x_j to the process that has the nonzero element A_{ij}. As shown in Figure 2, there are four processes in total. According to the principle of equalization, each process calculates four lines respectively, and each node is assigned a corresponding vector. As can be seen, node3 needs to receive C0 from node 0, C4 and C5 from node1, C8, C9, C10 and C11 from node2, and node3 receives seven pieces of information in total. Similarly, node0 needs to receive C4 and C6 from node1, and C13 and C14 from node3, node1 needs to receive C0, C1, and C2 from node0, C8, and C9 from node2, and C12, C13, and C15 from node3. Finally, node0 receives a total of four messages, node1 receives eight messages, and node2 receives six messages.
3) y_i*+=A_{ij} times x_j in the process, where y_i* is the intermediate result.
4) Send intermediate result y_i* to the final y_i process.
5) Add the y_i* received to get the final result y_i.

C. Motivation

The scalability of Distributed SpMV could be affected by various factors. Now we analyze the example shown in Figure 2, which is a sparse matrix already got reordered through graph partitioning. We thus can see that the diagonal blocks contains the most nonzeros. However, although it is well preprocessed, the problem of imbalanced load still exists.

First of all, as shown in Figure 2, distributed SpMV operation generally requires a lot of communications between various processes, which is one of the limiting factors of distributed SpMV. Secondly, as explained in Section II-B, the four nodes need to receive four, eight, six, and seven pieces of messages (i.e., components of x) respectively. It can be seen that the problem of imbalanced communication load is restricting the performance of distributed SpMV. Thirdly, the diversity of the sparsity patterns of matrices may lead to imbalanced calculation after the matrix is divided into each node. Although Figure 2 has been reorganized after graph partitioning, the computational loads of each node are 16, 17, 18 and 19, respectively, and thus lead to imbalanced computations. Finally, with the expansion of node size, the communication volume and time of distributed SpMV will increase significantly as well.

This has led to further optimization requirement of distributed SpMV for computation and communication balancing. In the next section, our `DistSpMV_Balanced` algorithm and how we address these challenges will be explained in detail.

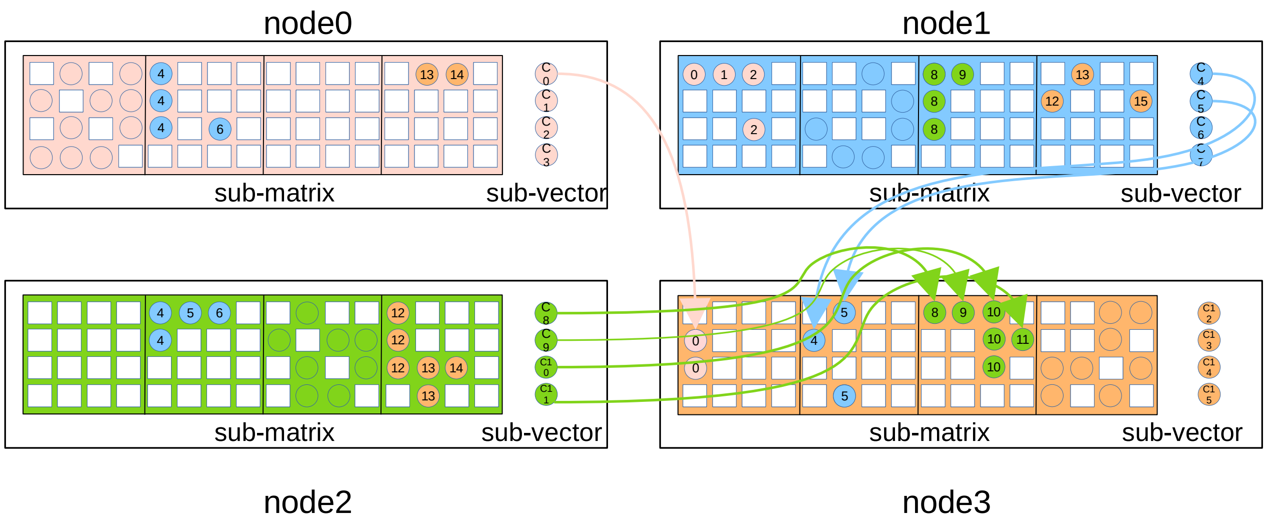


Fig. 2: A 16×16 matrix and a vector of length 16 are evenly distributed on a four-node cluster for distributed computation of SpMV, where each sub-matrix is labeled with the column number of the non-zero element, and the non-zero element without the label is the local non-zero element, which can be computed with the sub-vector on that node, while the non-zero element with the label needs to communicate with the node containing the corresponding vector The non-zero element with the label needs to communicate with the node containing the corresponding vector to obtain the vector needed for the computation. Here we draw the communication situation with node 3 as an example, node 3 needs to get the 0th element of the vector from node 0, the 4th and 5th elements of the vector from node 1, and the 8th, 9th, 10th and 11th elements of the vector from node 2, which requires 7 times of communication.

III. Methodology

A. Overview

In order to describe our `DistSpMV_Balanced` algorithm for optimizing distributed SpMV with more details, the data structure and the meanings of some variables used in our algorithm are introduced in Section III-B. In Section III-C, a 16×16 matrix as an example is taken to describe in detail our `DistSpMV_Balanced` optimization strategy for distributed SpMV. Firstly, in the preprocessing stage, graph partitioning is used to divide the graph corresponding to the sparse matrix, and the partition results are used to reorder the matrix. At this time, the nonzero elements are concentrated on the diagonal block as much as possible to reduce the communication volume. Secondly, we adjust the number of columns of the diagonal block, and expand the number of nonzero elements in the diagonal block again to reduce the communication volume between the computing nodes. Then, we split the matrix into a local matrix and a remote matrix. For the remote matrix, the number of row blocks are adjusted to balance the computation. Finally, the MPI+OpenMP hybrid programming pattern is used to achieve two levels of parallelism and to optimize the distributed SpMV with computational load balancing between processes and threads. Detailed steps in `DistSpMV_Balanced` algorithm above are described in Section III-C.

B. Data Structure

We first define p as the number of processes. The large matrix need to be divided into p local and p remote matrices. All matrices are stored in the `CSR` format. The `CSR` format consists of three arrays: (1) array `rowptr` stores the row offset of the matrix, (2) array `colidx` stores the column index of each nonzero element of the matrix, and (3) array `val` stores the value of each nonzero element of the matrix. There are also three integers and an array: (1) $rownum$ is row number, (2) $colnum$ is column number, (3) $nnznum$ is number of nonzero elements, and (4) array `nodeid` stores the process id that this nonzero element need to receive.

In the preprocessing process, six arrays are needed when the original matrix is being divided the local matrix and the remote matrix: (1) array `subrowadd` of size $p+1$ records the row number at the beginning of the diagonal matrix, (2) array `left_bound` of size p records the column number at the beginning of the diagonal matrix, (3) array `right_bound` of size p records the column number at the end of the diagonal matrix, (4) the size of array `rowptr_start` is the number of rows in the matrix and records the left-most nonzero element included by the local matrix in each row; (5) the size of array `rowptr_end` is the number of rows in the matrix and records the right-most nonzero element included by the local matrix in each row. (6) The array `nnznum` is of size p, and records the number of nonzero entries contained in each local matrix.

Each process calculates the communication information needed in the SpMV procedure, including one integer and four arrays: (1) *infocount* is the number of messages, (2) array `sendid` records the id of the process that sent the message, (3) array `recvid` records the id of the process that received the message, (4) array `index` records the position of the

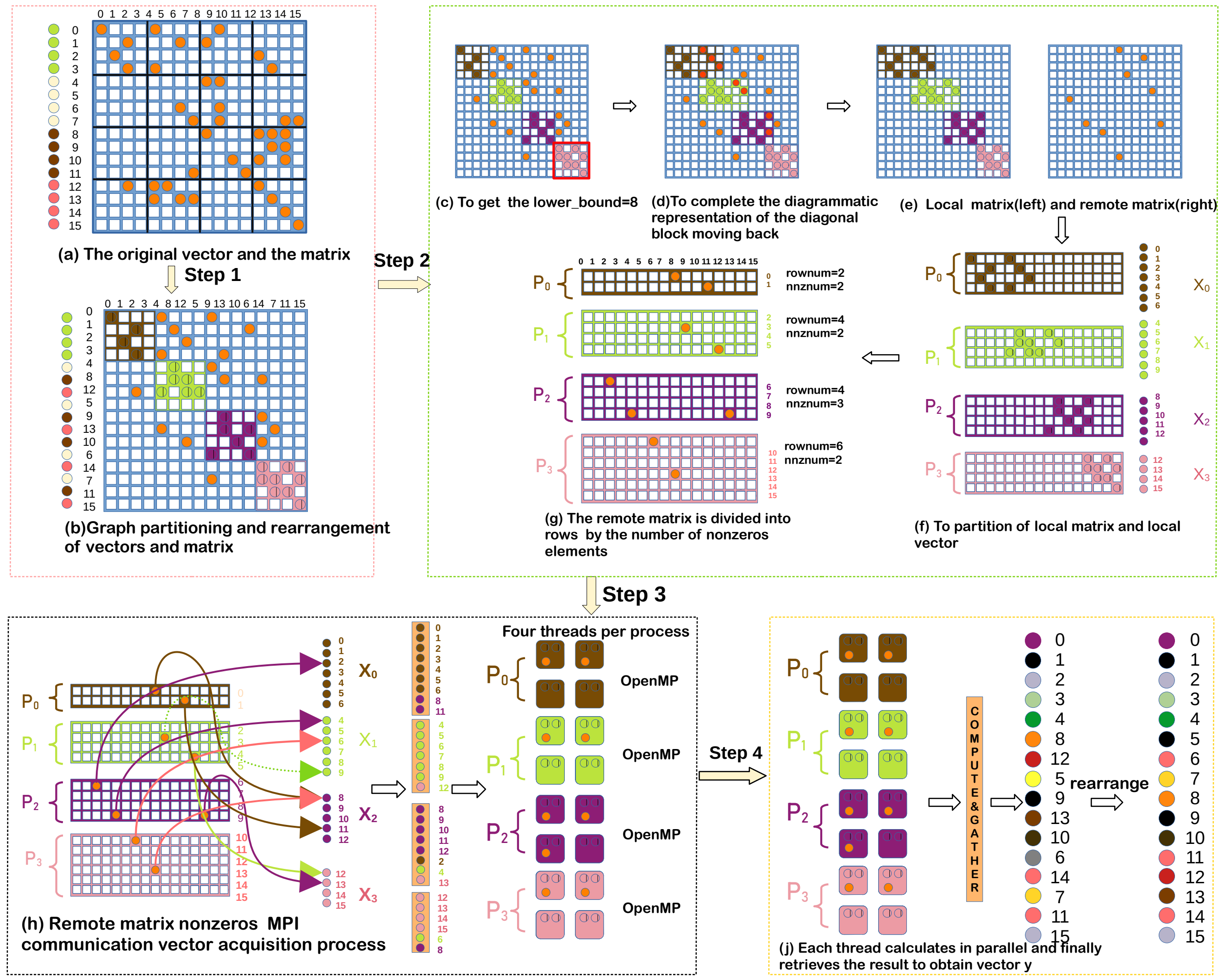

(a) The original vector and the matrix
Step 1
(b)Graph partitioning and rearrangement of vectors and matrix
Step 2
(c) To get the lower_bound=8
(d)To complete the diagrammatic representation of the diagonal block moving back
(e) Local matrix(left) and remote matrix(right)
(f) To partition of local matrix and local vector
(g) The remote matrix is divided into rows by the number of nonzeros elements
rownum=2 nnznum=2
rownum=4 nnznum=2
rownum=4 nnznum=3
rownum=6 nnznum=2
Step 3
(h) Remote matrix nonzeros MPI communication vector acquisition process
Four threads per process
OpenMP
Step 4
COMPUTE&GATHER
rearrange
(j) Each thread calculates in parallel and finally retrieves the result to obtain vector y

Fig. 3: An example of the `DistSpMV_Balanced` algorithm. The sample 16×16 sparse matrix is divided into four processes and every process has four threads. First, the graph partitioning tool METIS is used to divide the corresponding graph of the matrix, and then the original matrix and vector are reordered according to the partition results. Second, we count the nonzero elements of the four diagonal blocks, take the maximum value as the threshold *lower_bound* to adjust the number of columns of diagonal blocks, and move the right boundary of each diagonal block backward until the nonzero element in the block is greater than or equal to *lower_bound* or move to the right boundary of the original matrix. The local matrix is evenly divided according to rows and the local vector is divided according to the position of nonzero element. At the same time, the number of rows of the remote matrix is adjusted to realize communication and calculate load balancing. Third, MPI parallelism between nodes and OpenMP parallelism within nodes are used for communication and computation. Finally, the calculated value of each process is recovered and sorted to get the final result.

message in the vector, (5) The size of the array `flag` is the number of columns of the matrix, which records whether the elements at each position of the vector have been passed, and if they have been passed, they do not need to be passed again

That is all about the data structure. In the next subsection, `DistSpMV_Balanced` Algorithm will be described in depth.

C. Algorithm Description

Figure 3 shows a sparse matrix of size 16×16 divided into four nodes. Distributed SpMV of four threads in each node is used as an example to show the specific process of the `DistSpMV_Balanced` Algorithm.

In the first step, it should be noticed that the communication size in the distributed SpMV may restrict the scalability and performance of the algorithm. As the number of processes increases, the communication volume in the algorithm will keep increasing, and the communication time will become the performance bottleneck of the algorithm after reaching a certain level. In the algorithm, at first, the graph partitioning tool **METIS** is used to divide the corresponding graphs of the matrices, and then reorder the original matrices according

Algorithm 1 Calculating local_matrix boundary

```
Input: subrowadd, rowptr_start, rowptr_end
Output: left_bound, right_bound
for each $i \in [0, p-1]$ do
  while right_bound[i] + 1 < $matrix.colnum$ and nnznum[i] < $lower\_bound$ do
    $right \leftarrow matrix.colnum$
    for each $j \in$ [subrowadd[i], subrowadd[i+1]] do
      $begidx \leftarrow$ sourcematrix.rowptr[j+1]
      $next \leftarrow$ rowptr_end[j]+1
      if $next < begidx$ then
        $next\_idx$ = sourcematrix.colidx[next]
        if $next\_idx < right$ then
          $right$ = $next\_id$
        end if
      end if
    end for
    right_bound[i] = $right$
    for each $j \in$ [subrowadd[i], subrowadd[i+1]] do
      $begidx \leftarrow$ sourcematrix.rowptr[j+1]
      $next \leftarrow$ rowptr_end[j]+1
      if $next < begidx$ then
        $next\_idx$ = sourcematrix.colidx[next]
        if $next\_idx$ = $right$ then
          rowptr_end[j] = $next\_idx$
          nnznum[i] ++
        end if
      end if
    end for
  end while
end for
```

to the result of the partitioning to achieve load balancing and reduce the communication. Compared with the matrix before reordering, the number of non-zero elements within the four diagonal blocks of the reordered matrix increases and the number of non-zero elements in the non-diagonal blocks decreases. This means that the number of vectors that need to be communicated by each node becomes reduced, and the communication time as well as the program running time will be reduced also. At the same time, the vector is reordered according to the division results to ensure the correctness of the calculation results.

(a) to (b) of Figure 3 show an example of dividing the graph corresponding to the sparse matrix and using the result to reorder the matrix. Figure 3(a) is the original matrix, and Figure 3(b) is the matrix after the graph partitioning and matrix rearrangement. Some nonzero elements of the matrix move toward the diagonal block. In Figure 3(a), the number of non-diagonal block nonzero elements is 28, but in Figure 3(b), the number of non-diagonal block nonzero elements is reduced to 17, which reduces the amount of communication volume between nodes.

Algorithm 2 Getting communication information

```
Input: Remote_matrix, Left_bound, Right_bound
Output: Comm
$Flag \leftarrow 0$
for each $nz \in Remote\_matrix$ do
  $r \leftarrow nz.rowidx$
  if $nz.colidx \notin$ [ Left_bound[r], Right_bound[r]] and Flag[nz.colidx] = 0 then
    Flag[nz.colidx] = 1
    $Length$ = $\lfloor Colnum/p \rfloor$
    $need$ = $nz.colidx/Length$
    if $need$ = p then
      $need$ = $p - 1$
    end if
    $Cur \leftarrow$ Comm[need]
    Cur.recvid[Cur.infocount] = $i$
    Cur.sendid[Cur.infocount] = $need$
    Cur.index[Cur.infocount] = $nz.colidx$
    $Cur.infocount$ ++
  end if
end for
```

In the second step, as shown in step 2 of the description process of the Figure 3, the matrix is divided into local matrix and remote matrix. The local matrix does not need to communicate with each other. Therefore, we will try our best to divide non-zero elements into the local matrix to further reduce the amount of communication volume. Meanwhile, in order to achieve communication load balancing, we will make the following operations:

1. Count the number of nonzero elements of each diagonal block, and use the maximum value as the threshold *lower_bound*. As shown in (e) in Figure 3, the fourth diagonal block in (d) has a maximum of eight nonzero elements, so the value of *lower_bound* is selected as eight in the example.

2. Adjust the number of columns for each diagonal block based on *lower_bound*. Move the right boundary *right_bound* of each diagonal block to the right until the number of nonzero elements is greater than or equal to *lower_bound* or until the right boundary of the large matrix is reached. In the Figure 3, the red nonzero elements are added to the local matrix. To achieve this, we employ a scanline algorithm, as shown in Algorithm 1, which maintains the left-most nonzero and right-most nonzero column numbers *rowptr_start* and *rowptr_end* for each row in the diagonal block, and the left-most and right-most column numbers *left_bound* and *right_bound* for the diagonal block. For each diagonal block, we follow these steps to move *right_bound*:

a) If the matrix is not swept to the last column or the number of non-zero entries is less than *lower_bound*, perform **b** and **c**; otherwise, exit the loop.

b) Find the non-zero element next to *rowptr_end* for each row and set the minimum column id to the new *right_bound*.

c) Update the *rowptr_end* and number of non-zero bits in each row.

3. Divide the large matrix into local matrix and remote matrix according to whether the nonzero element is included in the extended diagonal block, as shown in (e) in Figure 3.

4. Divide the local matrix into four local matrices according to the row equalization strategy, and at the same time, the local vectors required for partitioning are corresponding, as shown in (f) of Figure 3. For example, there are nonzero elements in columns 0 to 6 of P_0, so the corresponding local vector is rows 0 to 6 of the original vector.

Algorithm 3 Communication between nodes

```
Input: sub_Comm, x
Output: Comm
 1: for each $message \in$ sub_Comm do
 2:     $Id \leftarrow message.recvid$
 3:     $Index \leftarrow message.index$
 4:     $Count \leftarrow$ send_Cnt[Id]
 5:     vec_Val[Id][Count] = x[Index]
 6:     send_Cnt[Id] ++
 7: end for
 8: MPI_Alltoall(send_Cnt,recv_Cnt)
 9: for each $i \in [0, p-1]$ do
10:     if $i \neq ProcessId$ then
11:         MPI_Isend(vec_Val[i], send_Cnt[i])
12:     end if
13: end for
14: for each $i \in [0, p-1]$ do
15:     if $i \neq ProcessId$ then
16:         MPI_Recv(recv_Val[i], recv_Cnt[i])
17:     end if
18: end for
19: $pos \leftarrow 0$
20: for each $colidx, nodeid \in sub\_Remote\_matrix$ do
21:     $val \leftarrow$ recv_Val[nodeid][pos[nodeid]]
22:     if x[colidx] = 0 then
23:         x[colidx] = $val$
24:         pos[nodeid] ++
25:     end if
26: end for
```

5. The number of rows in the remote matrix is divided according to the nonzero element equalization strategy to achieve communication load balancing and reduce communication volume, as shown in Figure 3 (g).

After that, we count the communication information between each process. As shown in the Algorithm 2. Finally, we send each local matrix, remote matrix, vector and communication information to the four processes.

In the third step, each process of the distributed SpMV needs to communicate with other processes to exchange vector values with each other. Some processes communicate within the same node and some communicate across nodes, but the communication mode using MPI alone cannot take advantage of shared memory well for processes within the same node. Therefore, we use OpenMP within a node to fully use the shared memory. In summary, a two-level parallel mode using MPI parallelism between nodes and OpenMP parallelism within nodes is used.

In each process, local matrices can calculate SpMV using local vectors, while remote matrices need to get the corresponding vectors from other processes. So each process needs to send and receive vector information to and from each other. In step 2, the information that needs to be sent has been passed to each process. As shown in the Algorithm 3, We use *MPI_Alltoall* to get the number of messages that each process needs to receive based on the number of messages that need to be sent, while sending and receiving the values of the required vectors. After that, we iterate through all the non-zero elements of the remote matrix and expand the local vector.

Then, we divide the number of nonzero elements equally and allocate them to each thread so that each thread computes the same number of nonzero elements as possible to achieve computation equalization, as shown in Algorithm 4.

Algorithm 4 Computation balancing

```
Input: nnznum, nthreads
Output: y
 1: $stridennz$ = $\lceil nnznum/nthreads \rceil$
 2: for each $tid \in [0, nthreads]$ in paraller do
 3:     $boundary = tid * stridennz$
 4:     if $boundary > nnznum$ then
 5:         $boundary = nnznum$
 6:     end if
 7:     iter[tid] = $binary\_search\_right(boundary)$
 8: end for
 9: for each $tid \in [0, nthreads]$ in paraller do
10:     for each $u \in$ [iter[tid], iter[tid+1] $- 1$] do
11:         y[u] = 0
12:         for each $colidx, val \in sub\_matrix$ do
13:             y[u] += $val *$ x[colidx]
14:         end for
15:     end for
16: end for
```

In the last step, we combine the vectors computed by each process, and use the *MPI_Bcast* function to send the number of rows *subm* and the number of rows prefix and *submadd* of the sub-vectors to the individual processes, then gather the sub-vectors using *MPI_Gatherv*. Finally, restore the result vector y in the order of rearrangement, as shown in Figure 3 (j).

IV. Experiment Results

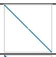
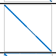
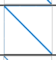
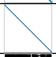
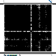
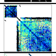
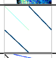
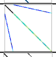
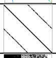
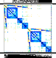
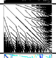
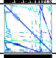
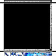
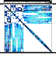
A. Experimental Setup and Dataset

`DistSpMV_Balanced` algorithm is tested on a cluster equipped with eight nodes, and each one has one AMD 32-core EPYC 7551 CPU and 128GB DRAM, the nodes are connected with 200Gb InfiniBand network. We use the

SuiteSparse Matrix Collection (previously known as the University of Florida Sparse Matrix Collection) [24] as our dataset. 20 representative matrices are selected to illustrate the performance of `DistSpMV_Balanced` algorithm. These matrix structures are shown in Table I. In order to prove the generality of algorithm optimization, the 20 sparse matrices include the first four regular matrices with non-zero elements mainly concentrated in the diagonal and the last 16 irregular matrices with random distribution of non-zero elements.

Three algorithms, `DistSpMV`, `DistSpMV_Reordered`, and `DistSpMV_Balanced`, are tested on the platform. We set the number of processes to 1, 2, 4, 8, 16, 32, and 64, with four threads for each process. We run each SpMV 50 times and report the average execution time.

TABLE I: The 20 representative matrices tested.

Matrix	Plot	Size	*nnz*
cant		62.4K×62.4K	4M
bone010		986.7K×986.7K	47.8M
rajat31		4.6M×4.6M	20.3M
ecology1		1M×1M	4.9M
asia_osm		11.9M×11.9M	25.4M
ldoor		852.2K×952.2K	42.4M
nlpkkt80		1M×1M	28.1M
dielFilterV2real		1.1M×1.1M	48.5M
rgg_n_2_21_s0		2M×2M	28.9M
road_central		14M×14M	33.8M
inline_1		503.7K×503.7K	36.8M
hugebubbles00000		18.3×18.3M	54.9M
germany_osm		11.5M×11.5M	24.7M
italy_osm		6.6M×6.6M	14M
adaptive		6.8M×6.8M	27.2M
vas_stokes_1M		1M×1M	34.7M
AS365		3.7M×3.7M	22.7M
M6		3.5M×3.5M	21M
NLR		4.1M×4.1M	24.9M
audikw_1		943.6K×943.6K	77.6M

B. *Performance Comparison of Three Algorithms*

Figure 4 shows the performance comparison of the 20 representative matrices `DistSpMV`, `DistSpMV_Reordered` and our optimization algorithm `DistSpMV_Balanced`, respectively. The `DistSpMV_Balanced` optimization results in an average speedup of 77.20x compared with `DistSpMV`. Compared with `DistSpMV_Balanced`, an average speedup ratio of 5.18x is obtained, with the best observed speedup of 8.94x (found in matrix 'cant'). Also, both the regular and the irregular matrices gain significant acceleration. With the expansion of the number of processes, the performance of most matrices has been improved. Although the performance of the other four matrices ('nlpkkt80', 'vas_stokes_1M', 'Ecology1', 'cant') has decreased, the overall performance compared with the other two algorithms still greatly improved. Among them, matrix 'cant' obtains the maximum performance acceleration ratio.

Specifically, two representative matrices 'road_central' and 'inline_1' are selected to analyze our implementation for reducing communication volume and realizing computational load balancing.

Firstly, the communication volume between the two matrices in eight processes under three algorithms are counted, and the communication heap maps are drawn as shown in (a) and (b) of Figure 5. The horizontal and vertical coordinates of the graph are the process numbers respectively. The `DistSpMV` algorithm of the two matrices has a very imbalanced communication load and a large amount of communication, which causes a large communication volume and leads to poor performance of distributed SpMV. The intermediate thermal map of (a) and (b) is the communication heap map of each process of the `DistSpMV_Reordered` algorithm after being divided by the graph. Compared with the `DistSpMV` algorithm, the traffic of all processes has been greatly reduced. Finally, after adjusting the number of columns by `DistSpMV_Balanced` algorithm and placing nonzero elements on the local matrix as much as possible, the traffic between the eight processes is shown in the last heat map of (a) and (b). The further lightening of the colors compared with `DistSpMV_Reordered` thermal maps indicates a further decrease in traffic. In (c), we further compared `DistSpMV_Reordered` with our `DistSpMV_Balanced` algorithm. The two line charts on the left of (c) compare the traffic of each process, which are more intuitive than the heat map. For the most part, our algorithm has less inter-process communication volumes than the `DistSpMV_Reordered` algorithm. The comparison of the middle two figures in Figure (c) shows that `DistSpMV_Balanced` requires more local computations, which again indicates that the `DistSpMV_Balanced` algorithm reduces the number of nonzero elements in the remote matrix, which reduces the communication volume.

Secondly, for communication load balancing and calculation balancing, the calculation amount of nonzero element of the remote matrix in each process is counted. The two right-most line drawings in Figure 5 (c) compare the calculation amount of the remote matrix of the two algorithms. Our optimization strategy includes adjusting the number of rows of the remote matrix to achieve the nonzero number load balancing of the remote matrix for each node and the calculation balance of the remote matrix, so as to achieve the communication load balancing. Therefore, the `DistSpMV_Balanced` remote matrix of the balanced algorithm is close to a straight line on the line

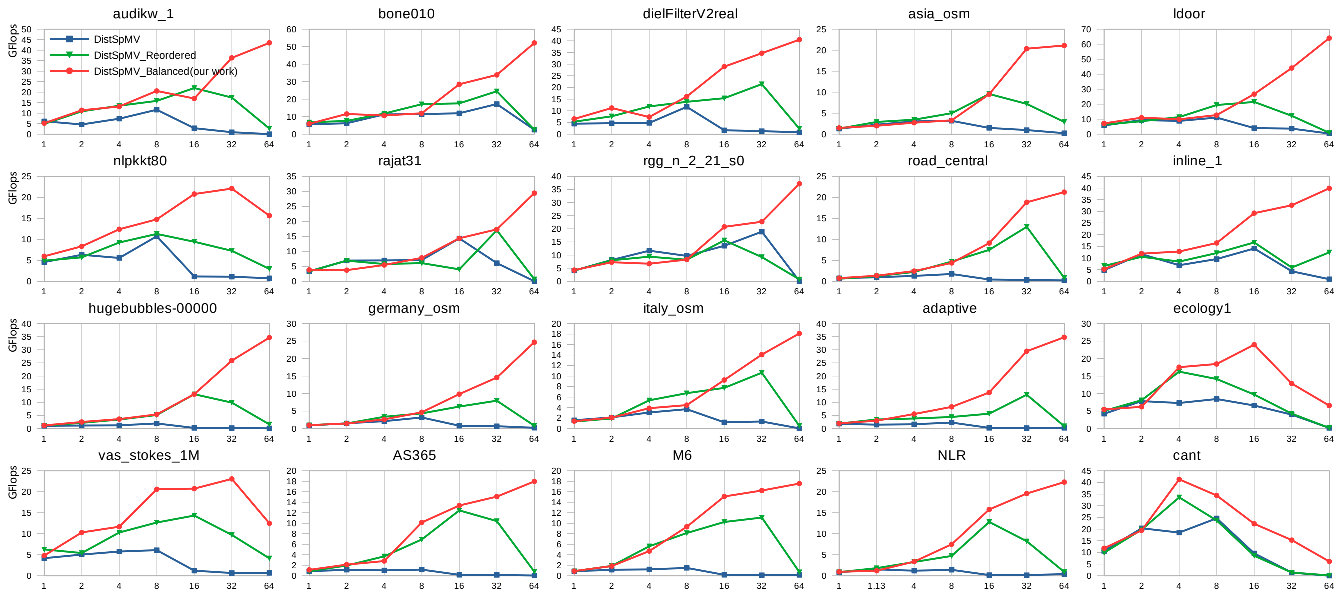
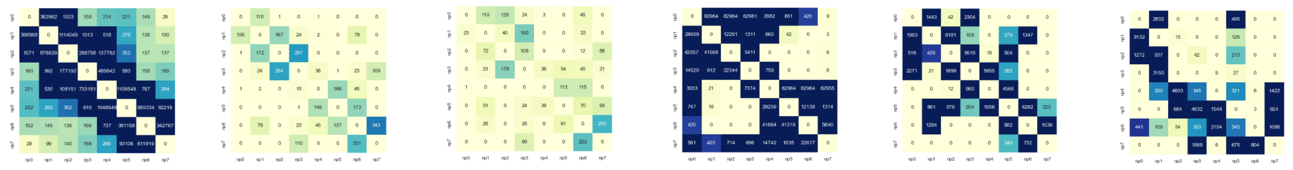
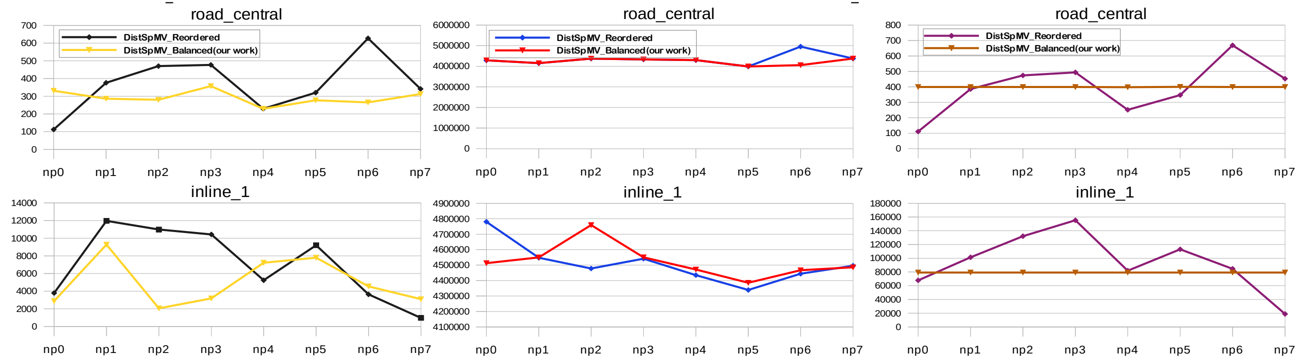


Fig. 4: The performance and scalability (each sub-figure represents running a sparse matrix on different number of processes) of the three methods `DistSpMV`, `DistSpMV_Reordered` and `DistSpMV_Balanced`. The x axis is the number of processes (each process has four threads), the y axis is the performance in GFlops.



(a) From left to right is three heat maps of the traffic between various processes in DistSpMV, DistSpMV_Reordered, and DistSpMV_Balanced for matrix road_central

(b) From left to right is three heat maps of the traffic between various processes in DistSpMV, DistSpMV_Reordered, and DistSpMV_Balanced for matrix inline_1



(c)From left to right, the communication volume of each process, the computation volume of local matrix and the computation volume of remote matrix in METIS and our work are compared

Fig. 5: Heatmaps of communication of each process of 'road_central' and 'inline_1', two representative matrices executed by running eight processes under the three algorithms. And we compare the traffic, local matrix computation, and remote matrix computation of the two matrices under `DistSpMV_Reordered` and `DistSpMV_Balanced` algorithms.

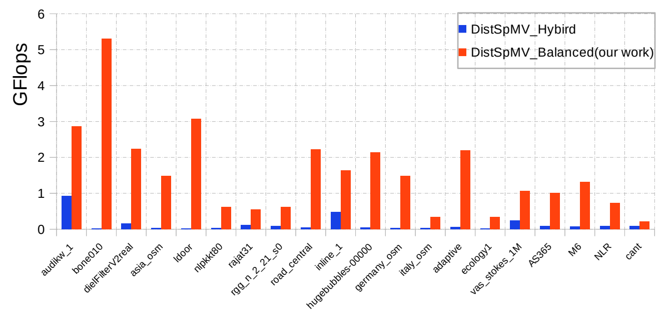


Fig. 6: Performance comparison between `DistSpMV_Balanced` and `DistSpMV _Hybrid` with 256 cores (64 processes $\times 4 threads$).

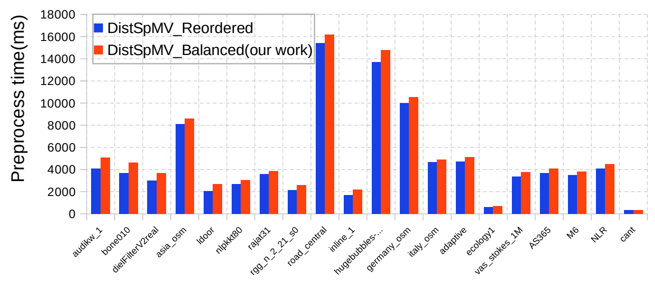


Fig. 7: The preprocessing cost of 20 matrices under `DistSpMV_Reordered` and `DistSpMV_Balanced` algorithms. It can be seen that our method only brings negligible extra cost over reordering.

chart. It shows the effectiveness of our strategy in achieving communication and computation balancing.

C. Comparison with Existing Work `DistSpMV_Hybrid`

The performance data of 20 matrices under `DistSpMV_Balanced` and `Hybrid_SpMV` algorithms are shown in Figure 6. We compare performance under the same conditions of 64 processes with four threads per process. As can be seen from Figure 6, `DistSpMV_Balanced` has achieved significant performance improvement compared with `DistSpMV_Hybrid`. Statistically, we achieved an average acceleration ratio of 19.56x (up to 48.49x). The performance of all matrices has been greatly improved. For 'bone010', 'ldoor' and other matrices, `DistSpMV_Hybrid` degradation is very obvious, while `DistSpMV_Balanced` still has a good performance, which shows the effectiveness of `DistSpMV_Balanced` algorithm.

D. Preprocessing Overhead

The preprocessing overhead is an important metric of distributed SpMV. We test the 20 matrices and record the preprocessing overhead of `DistSpMV_Reordered` and `DistSpMV_Balanced` algorithms in 64 processes (under 256 threads), as shown in Figure 7. the nonzero element distribution diversity of different matrices leads to different preprocessing time cost, and the cost changes of the two algorithms are roughly the same. At the same time, overall, our optimization on top of the graph partitioning does not cost much additional overhead. Among these 20 matrices, the maximum preprocessing cost is 1.31x that of `DistSpMV_Reordered` algorithm (at matrix 'cant'), and the minimum preprocessing cost is only 1.05x (at matrix 'italy_osm').

V. Related Work

Much research has been conducted for **SpMV on shared memory systems**. A number of studies were focusing on designing formats for faster SpMV. Kreutzer et al. [3] and Gómez et al. [4] developed ELL-like sparse formats for utilizing longer vector units in modern processors. Kourtis et al. [2], Greathouse and Daga [8], Liu and Vinter [9], and Merrill and Garland [10] developed variants of the CSR format. Im et al. [5], Vuduc et al. [6] and Niu et al. [7] studied blocking formats for better data locality. Several studies optimized and evaluated SpMV on heterogeneous processors [25], [26].As for auto-tuning methods, Li et al. [11] proposed a machine learning framework called SMAT for auto-tuning SpMV. Zhao et al. [12] used computer vision and deep learning for selecting formats, and Du et al. [13] recently designed AlphaSparse for automatically generating sparse formats for SpMV. However, unfortunately, none of those highly optimized shared memory methods could be directly migrated to distributed platforms. In contrast, we in this work focus on scalability of distributed SpMV. Furthermore, our findings and techniques may bring new performance tuning opportunities for SpMV in a single node, since emerging larger chiplet-based processors [27] are likely encountering more NUMA effects [28], which needs some distributed thinking to optimize.

As for **SpMV on distributed memory systems**, developing graph and hypergraph partitioning approaches is the major direction [22]. Specifically, Hendrickson and Kolda [29] proved that the performance of parallel SpMV depends on the partitioning, and proposed several effective approaches. Aykanat et al. [30] transformed the matrix decomposition problem into the idea of dividing diagonal blocks for data locality. Nicol [14] proposed that the sparse matrix partitioning can be divided into one- and two-dimensional patterns. In order to better adapt SpMV algorithm to large-scale systems, Hendrickson et al. [31] proposed a 2D algorithm, and Kayaaslan et al. [32] developed a 1.5D method, and Çatalyürek et al. [33], [34] proposed several effective improvements in distributed SpMV. Uçar and Aykanat [35] and Li et al. [19] discussed how computations and communications can overlap. Acer et al. [36], [37] designed methods for lowering latency. For higher scalability, Boman et al. [38] and Bienz et al. [39] proposed a node-aware parallel SpMV algorithms. Devine et al. [40] discussed the use of hypregraph partitioning in scientific computing. Recently, Buluç et al. [41] and Çatalyürek et al. [22] surveyed the progress in this area. Compared to only using graph or hypergraph partitioning, we in this paper further reorganized the distribution of nonzeros for balancing computations and communications, and show obvious speedups over SpMV only using matrix partitioning.

VI. Conclusion

In this paper, we have proposed an algorithm called `DistSpMV_Balanced` that uses optimization strategies to balancing computations and communications of SpMV on distributed clusters. By testing 20 representative sparse matrices, our method is on average 77.20x and 5.18x (up to 460.52x and 27.50x) faster than classic distributed SpMV implementation and its variant reordered with graph partitioning. Compared with a recent hybrid distributed SpMV algorithm, our method bring on average 19.56x (up to 48.49x) speedup.

Acknowledgment

We deeply appreciate the invaluable comments from all the reviewers. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No. 61972415.

References

[1] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum," *ACM Trans. Math. Softw.*, vol. 28, no. 2, p. 239–267, 2002.

[2] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," in *PPoPP '11*, 2011, pp. 247–256.

[3] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[4] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running spmv on long vector architectures," in *PPoPP '21*, 2021, p. 292–303.

[5] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[6] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *SC '02*, 2002, pp. 26–26.

[7] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *IPDPS '21*, 2021, pp. 68–78.

[8] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14*, 2014, pp. 769–780.

[9] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *ICS '15*, 2015, pp. 339–350.

[10] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC '16*, 2016, pp. 678–689.

[11] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *PLDI '13*, 2013, pp. 117–126.

[12] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *PPoPP '18*, 2018, pp. 94–108.

[13] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "Alphasparse: Generating high performance spmv codes directly from sparse matrices," in *SC '22*, 2022, pp. 952–966.

[14] D. M. Nicol, "Rectilinear partitioning of irregular data parallel computations," *Journal of Parallel and Distributed Computing*, vol. 23, no. 2, pp. 119–134, 1994.

[15] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. 36, no. 05, pp. 570–580, 1987.

[16] M. Ujaldon, S. D. Sharma, E. L. Zapata, and J. Saltz, "Experimental evaluation of efficient sparse matrix distributions," in *ICS '96*, 1996, pp. 78–85.

[17] E. Saule, E. Ö. Baş, and Ü. V. Çatalyürek, "Load-balancing spatially located computations using rectangular partitions," *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1201–1214, 2012.

[18] S. Lee and R. Eigenmann, "Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems," in *ICS '08*, 2008, pp. 195–204.

[19] S. Li, C. Hu, J. Zhang, and Y. Zhang, "Automatic tuning of sparse matrix-vector multiplication on multicore clusters," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–14, 2015.

[20] M. Belgin, G. Back, and C. J. Ribbens, "A library for pattern-based sparse matrix vector multiply," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 62–87, 2011.

[21] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *SC '95*, 1995, p. 29–es.

[22] U. V. Çatalyürek, K. D. Devine, M. F. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner, "More recent advances in (hyper)graph partitioning," *ACM Comput. Surv.*, 2022.

[23] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (spmv) multiplication," in *2018 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, 2018, pp. 406–414.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[25] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Computing*, vol. 49, no. C, pp. 179–193, 2015.

[26] F. Zhang, W. Liu, N. Feng, J. Zhai, and X. Du, "Performance evaluation and analysis of sparse matrix and graph kernels on heterogeneous processors," *CCF Transactions on High Performance Computing*, 2019.

[27] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families : Industrial product," in *ISCA '21*, 2021, pp. 57–70.

[28] X. Yu, H. Ma, Z. Qu, J. Fang, and W. Liu, "Numa-aware optimization of sparse matrix-vector multiplication on armv8-based many-core architectures," in *NPC '21*, 2021, pp. 231–242.

[29] B. Hendrickson and T. G. Kolda, "Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing," *SIAM Journal on Scientific Computing*, vol. 21, no. 6, pp. 2048–2072, 2000.

[30] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, "Permuting sparse rectangular matrices into block-diagonal form," *SIAM Journal on scientific computing*, vol. 25, no. 6, pp. 1860–1879, 2004.

[31] B. Hendrickson, R. Leland, and S. Plimpton, "An efficient parallel algorithm for matrix-vector multiplication," *International Journal of High Speed Computing*, vol. 7, no. 01, pp. 73–88, 1995.

[32] E. Kayaaslan, C. Aykanat, and B. Uçar, "1.5d parallel sparse matrix-vector multiply," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C25–C46, 2018.

[33] U. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[34] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[35] B. Uçar and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM Review*, vol. 49, no. 4, pp. 595–603, 2007.

[36] S. Acer, O. Selvitopi, and C. Aykanat, "Optimizing nonzero-based sparse matrix partitioning models via reducing latency," *Journal of Parallel and Distributed Computing*, vol. 122, pp. 145–158, 2018.

[37] ——, "Addressing volume and latency overheads in 1d-parallel sparse matrix-vector multiplication," in *Euro-Par 2017: Parallel Processing*, 2017, pp. 625–637.

[38] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *SC '13*, 2013.

[39] A. Bienz, W. D. Gropp, and L. N. Olson, "Node aware sparse matrix–vector multiplication," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 166–178, 2019.

[40] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *IPDPS '06*, 2006.

[41] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*, 2016, pp. 117–158.