Performance of Distributed and Shared Memory Parallel Sparse Matrix Vector Multiplication

Kristian Sørdal

Thesis for Master of Science Degree at the University of Bergen, Norway

2025

Year:	2025
Title:	Performance of Distributed and Shared Memory Parallel Sparse Matrix Vector Multiplication
Author:	Kristian Sørdal

Acknowledgements

To err is human; but to really foul things up requires a computer.

Paul R. Ehrlich

Abstract

Sparse Matrix Vector Multiplication is an important kernel used in scientific computing. It is a problem that lends itself well to parlallization. The problem is bounded by, and scales with the memory bandwidth of the system. Therefore in order to efficiently perform *SpMV* on large distributed memory systems, it is important to reduce the communication between nodes, in order to extract as much as possible out of the memory bandwidth of the system.

This thesis aims to investigate the results of *SpMV* when ran using different communication strategies.

Contents

Chapter 1

Introduction

Sparse Matrix Vector Multiplication (SpMV) is a fundamental computational kernel in numerous scientific and engineering applications, including numerical simulation, optimization, and machine learning. Due to its common presence in iterative solvers and graph based computations, improving the performance of SpMV remains a critical goal in the field of high performance computing. However, SpMV is inherently memory bound and characterized by irregular memory access patterns, which makes it challenging to achieve high computational efficiency.

Parallelization of SpMV offers a pathway to performance improvement, but the associated communication overhead becomes a primary bottleneck when operating in distributed memory environments. Unlike shared memory systems, where all processors can access a global memory space, distributed systems require explicit data exchanges between compute nodes. These communications often dominate the runtime cost, especially for large scale problems where data dependencies span across multiple nodes. The situation is exacerbated by the low computational intensity of SpMV operations, which further limits the benefits of increased computational resources unless communication is carefully managed.

This thesis aims to investigate the impact of different communication strategies on the performance of parallel SpMV in hybrid shared-distributed memory settings. By combining a shared memory model within individual compute nodes through the use of OpenMP and a distributed memory model for inter node communication using MPI, the study explores how various strategies, ranging from full vector broadcasts to minimal selective updates, affect both communication volume and execution time.

We evaluate several communication schemes, including:

- Full vector exchange, where the entire input or result vector is communicated across nodes.
- Separator based methods, where only the boundary elements (separators) are exchanged.
- Minimally selective communication, which restricts communication to only those separator values that are actually required for computation on neighbouring nodes.

These strategies are benchmarked on modern multicore architectures to assess their scalability and efficiency. Particular attention is paid to how different partitioning techniques, such as graph and hypergraph partitioning, influence the communication patterns and associated costs.

The central research question guiding this work is:

How do different communication strategies affect the performance of distributed SpMV, and what trade offs exist between communication volume, computational load balancing, and total execution time?

Chapter 2

Theory

2.1 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental operation encountered in many areas of scientific computing. It is especially prominent in solving large systems of linear equations and in large-scale simulations. The matrices involved are typically both very large and very sparse.

A matrix can in theory be considered sparse if it is worthwhile to treat zero values separately. In theory, this translates to a matrix being less than full, i.e. less than $\mathcal{O}\left(n^2\right)$ nonzeros for a $n \times n$ matrix. However, in the context of sparse linear algebra, sparse means that there is a constant number of nonzeros per row, i.e. $\mathcal{O}\left(n\right)$ nonzeros per row. The matrices used in scientific computing, such as matrices based on meshes, or graphs such as social networks all have this property. Optimizing the performance of SpMV, particularly through parallel computing techiniques, is crucial for enhancing the efficiency of many scientific applications.

However, SpMV is notoriously difficult to optimize, both in sequential and parallel implementations. One major reason is its inherently low computational intensity.

2.2 Definitions

Term	**Definition**
Node	A node, or a compute node, is a computeunit within a larger parallel computing system.
Dual/single socket	A dual or single socket node refers to the amount of processors on a node. Dual socketed nodes have two processors, that are connected through an interconnect.

2.3 Latency

The execution time of computational operations can vary significantly, and it is important to have some understanding of the latency times associated with typical operations. Referencing these latency numbers can be valuable when interpreting benchmark results and the performance of different programs.

Operation	**Time [ns]**
L1 cache reference	1
Branch misprediction	3
L2 cache reference	4
Mutex lock/unlock	17
Main memory reference	100
Compress 1K bytes with Zippy	2000
Send 2kB over 10 Gbps network	1600
Send 1K bytes over 1 Gbps network	10 000
Read 4K bytes randomly from SSD*	20 000
Round trip within same datacenter	500 000
Read 1MB sequentially from memory	1 000 000
Disk seek	10 000 000
Read 1MB sequentially form disk	10 000 000
TCP packet round trip between continents	150 000 000

Figure 2.1: Latency numbers for common operation, adapted from [9]

2.4 Parallel Architectures

There are two main architectures used in the parallel computing industry: Shared Memory Architecture and Distributed Memory Architecture. The following sections gives an overview of the key difference between the two.

2.4.1 Shared Memory Architecture

On a system with shared memory architecture, every processing unit (PU) have access to the same memory, treat it as a global address space. On such systems, the biggest challenge is that of *cache coherency*, where in order to prevent race conditions, every read of the cache must reflect the latest write (adapted from [7]).

2.4.2 Distributed Memory Architecture

On systems with distributed memory architecture, every processor have their own local memory, not accesible by other processors. When a process needs to access memory from another process, explicit communication of the data stored at that memory address needs to occur, and happens through whichever network the processors are connected with (adapted from [7]). Figure 2.2 shows the difference between shared and distributed memory architectures.

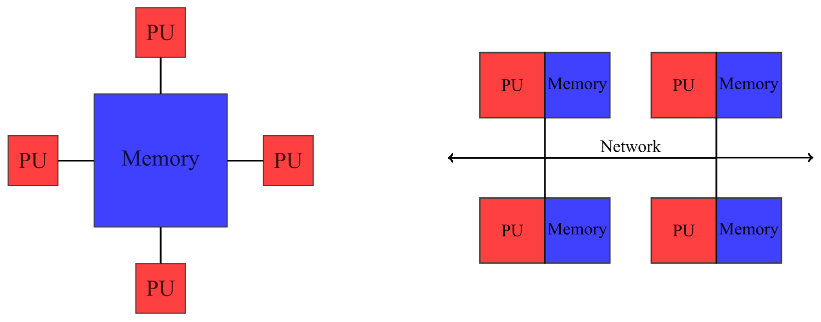

Figure 2.2: Shared and Distributed Memory Architecture (adapted from [6])

2.4.3 Non-Uniform Memory Access

Non-Uniform Memory Access (NUMA) refers to a multiprocessor system architecture in which memory access latencies depend on the location of the memory relative to a given processor. Unlike traditional symmetric multiprocessing (SMP) systems, where all processors share equal access times to a centralized memory pool, NUMA architectures consist of multiple processor sockets or nodes, each directly connected to its own local memory. These nodes are interconnected by a high-speed communication network, typically an interconnect, facilitating access to remote memory residing on other nodes.

2.4.4 IPUs and emerging architectures

In the ever evolving world of computer chip manufacturing, there has in the past years emerged a new trend in the field of architecture design. This comes in the form of packaging thousands of small processor cores into a single device, where each core has its own local memory, and no device-level shared memory [1]. An example of such a processor is the relatively new Cerebras Wafer-Scale Engine (WSE-3), which is the largest chip ever built. With a spec sheet of 4 trillion transistors, 900 000 cores, memory bandwidth of 21PB/s, and 44 GB of on-wafer memory (see [4]). Such architectures necessitates a careful distribution of relevant data across each processor, and is where memory scalable communication strategies are beneficial.

Chapter 3

Background

3.1 CSR Storage Format

CSR (Compressed Sparse Row) is the most widely used storage format for sparse matrices. As its name suggests, it compresses the amount of memory used to store a matrix without loss of information. It does so by utilizing three vectors A_p, A_j, A_x. Figure 3.1 shows an example of a matrix stored in CSR format, adapted from [3].

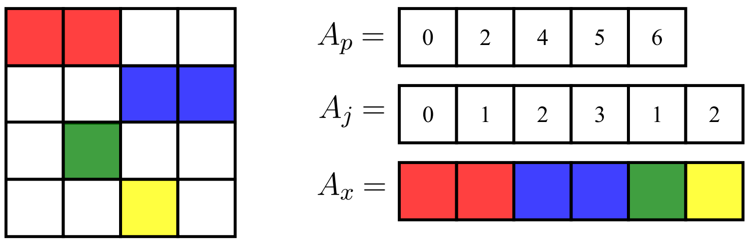

Figure 3.1: Matrix represented in the CSR format.

The first vector, A_p stores the indices of the first non-zero in the vectors A_p and A_x. For a given entry $A_p[i]$, $A_p[i]$ is the index of the first non-zero in the i^{th} row. $A_j[j]$ and $A_x[j]$ denotes the column index and value of the j^{th} non-zero, respectively.

Throughout the remainder of this thesis, we operate under the assumption that all matrices are represented in CSR format, unless explicitly noted otherwise.

3.1.1 Computational Intensity

The *computational intensity* of an operation describes the relation between the number of floating-point operations (FLOPS) and the number of memory accesses required. It is formally defined as:

$$\text{Computational intensity} = \frac{\text{FLOPS}}{\text{Memory accesses}} \tag{3.1}$$

Operations with low computational intensity, such as SpMV, are often *memory bound* rather than *compute bound*. This means that increasing the computational power of a system (e.g., faster processors) does not necessarily lead to proportional speedups in SpMV performance, as memory bandwidth remains the limiting factor.

3.2 Other storage formats

There exists many other lesser used storage formats for matrices

3.2.1 COO Format

The Coordinate List (COO) format stores the matrix as a set of triples on the form i, j, x, where i is the row index, j is the column index, and x is the value stored at (i, j) in the matrix. In this format all 0 entries are ignored.

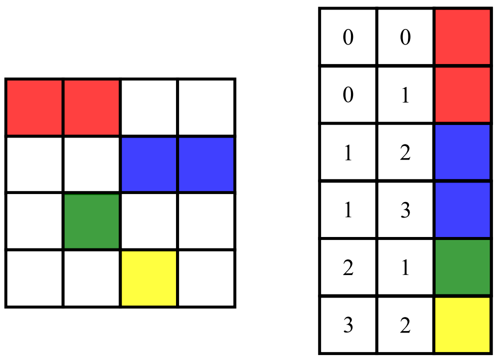

Figure 3.2: Example Matrix represented in the COO format.

3.2.2 CSC Format

The Compressed Sparse Column (CSC) format is similar to the CSR format, but instead of compressing the rows, we compress the columns. In this matrix format, we have irregular memory writes, but the reads are more regular. This can however be a problem

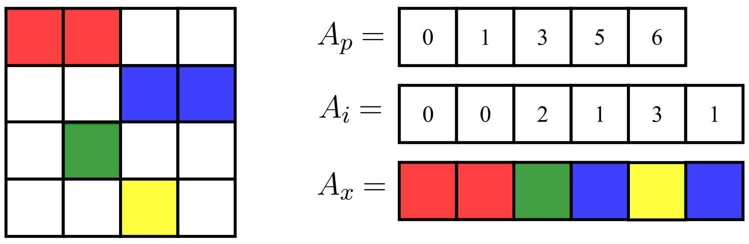

Figure 3.3: Matrix represented in the CSC format.

3.2.3 ELLPack Format

For an $M \times N$ matrix with a maximum of K non-zeros per row, the ELLPack format stores the non-zeros in an $M \times K$ matrix `data`, and an $M \times K$ matrix `indices`. The `data` matrix store the values of the non-zeros, and `indices` store the column index of every element. Rows that have fewer than K non-zeros are padded with zeros. Adapted from [11].

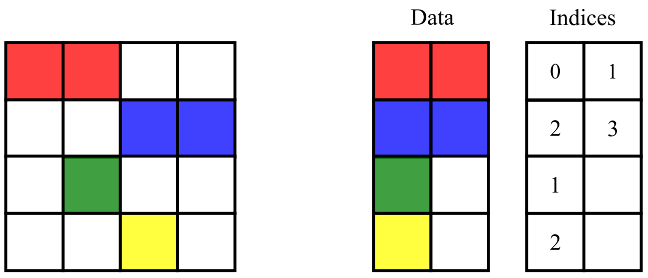

Figure 3.4: Matrix Represented in the ELLPACK format.

3.3 Sequential SpMV

A sequential implementation of SpMV on a matrix stored in the CSR format can be implemented in the following manner:

Algorithm 1: Sequential CSR-based SpMV

```
Input  : $A_p, A_j, A_x, x$
Output : $y$

for $i \leftarrow 0$ to $n$ do
    sum $\leftarrow 0$
    for $j \leftarrow A_p[i]$ to $A_p[i+1]$ do
        sum $=$ sum $+ A_x[j] \cdot x[A_j[j]]$
    $y[i] \leftarrow$ sum
```

For SpMV on well structured matrices such as those similar to the matrix illustrated in Figure 3.5, each non-zero element typically incurs a data movement cost of approximately 12 bytes and results in 2 floating-point operations (FLOPs). This estimation may appear inconsistent with the data access pattern described in algorithm 3.3, where two double-precision values and one integer are accessed per non-zero, corresponding to a total of 20 bytes. The apparent discrepancy arises from the caching behavior of the input vector x: after the initial access, elements of x are frequently reused and thus remain in cache, reducing the effective memory traffic associated with subsequent accesses.

In contrast, for highly unstructured matrices, the absence of data locality can significantly degrade cache reuse. In the worst case scenario, each non-zero may trigger the loading of an entire 64 byte cache line, with only a small fraction of it being used. Consequently, the effective data movement can increase to as much as 76 bytes per 2 FLOPs.

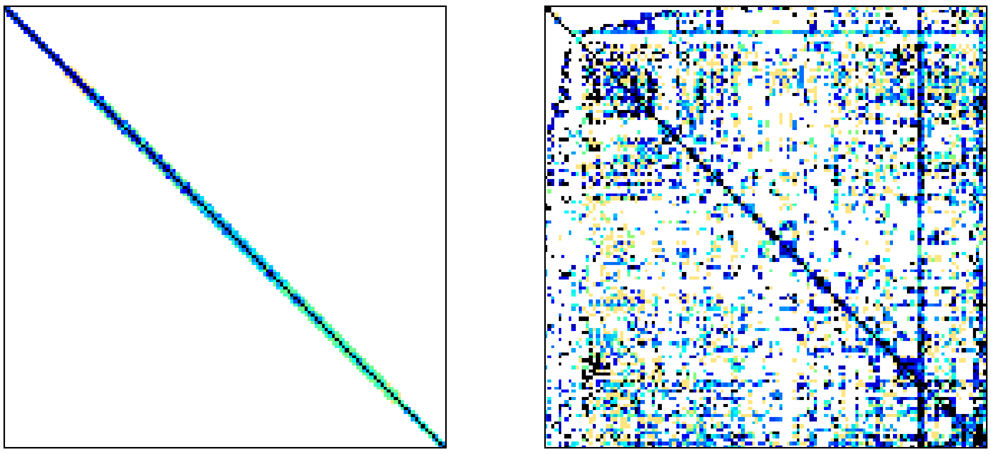
(a) Cube_Coup_dt0 (b) shermanACd

Figure 3.5: Well structured (a) and poorly structured (b) matrices.

3.4 Shared Memory SpMV

SpMV can be parallelized using the OpenMP directive `#pragma omp parallel for`. By default, this tells OpenMP to use `static` scheduling when parallelizing the outer iteration loop. When static scheduling is used, the span of the iteration that each thread will execute is precomputed, and stays static, as the naming suggests. There are other scheduling options, such as `dynamic` and `guided`, which will be discussed in later sections.

An implementation of shared memory SpMV is outlined below.

Algorithm 2: Shared Memory CSR-based SpMV

```
Input  : $A_p, A_j, A_x, x$
Output : $y$

#pragma omp parallel for
for $i \leftarrow 0$ to $n$ do
    sum $\leftarrow 0$
    for $j \leftarrow A_p[i]$ to $A_p[i+1]$ do
        sum $=$ sum $+ A_x[j] \cdot x[A_j[j]]$
    $y[i] \leftarrow$ sum
```

3.4.1 Scheduling options

As seen in algorithm 2, the outer loop is parallelized, which translates to dividing the rows of the matrix evenly among the threads. This work fine for well structured matrices, but for matrices with dense rows, such as the matrix shown in Figure 3.6, we obtain large imbalances in the computational load for each thread, which impacts performance.

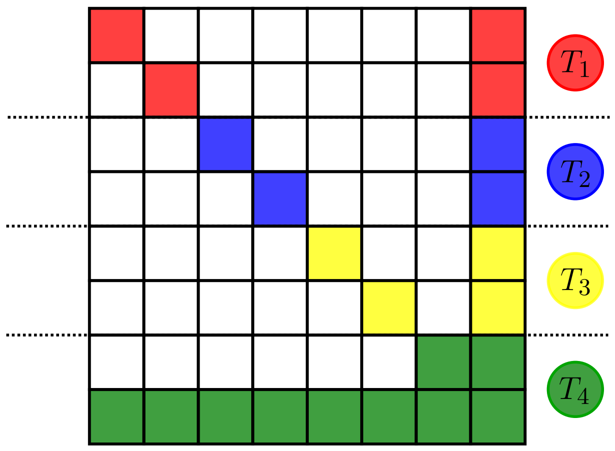

Figure 3.6: Row distribution among threads under static scheduling.

3.4.2 Dynamic Scheduling

Dynamic scheduling in OpenMP involves precomputing the range of iterations without assigning specific iteration subsets to individual threads in advance. Under static scheduling, if thread A receives sparse rows and thread B receives dense rows, thread A will become idle prematurely while thread B remains computationally engaged. In contrast, dynamic scheduling allocates iterations at runtime to threads as they become available, thus potentially balancing the computational load more effectively, particularly when iteration workloads vary significantly.

At first glance, dynamic scheduling appears to resolve the workload imbalance inherent in static scheduling, even though it comes with more overhead. While this assumption holds true on smaller systems, such as personal laptops equipped with fewer physical cores and a single processor socket, it does not readily apply to larger, dual-socket systems utilized in this thesis. Here, the first-touch memory policy becomes particularly significant.

3.4.3 First-Touch Policy

The first-touch policy stipulates that the initial thread accessing a memory page allocates this page to its local memory domain. Consequently, if thread first accesses a memory page, it is stored locally to thread . Subsequent accesses by thread to this page result in non-local memory access, compelling thread to retrieve the data from either remote or main memory. Such accesses incur performance penalties due to significantly higher latency compared to local memory access.

3.4.4 Performance Implications of the First-Touch Policy

Table 2.1 illustrates that accesses to main memory exhibit substantially higher latency compared to local cache (e.g., L1 cache) accesses. In dual-socket configurations, accessing memory residing on the remote socket involves inter-socket communication through an interlink, further exacerbating latency. Consequently, dynamic (or guided) scheduling alone does not mitigate the performance degradation arising from poorly structured matrices, as it inadvertently exacerbates memory locality issues inherent to the first-touch policy on NUMA architectures.

3.5 Distributed Memory SpMV

In distributed memory systems, the computational workload is distributed across multiple nodes, each typically comprising a dual-socket architecture with local memory. Unlike shared memory systems, data access across nodes is non-trivial and requires explicit communication, most commonly implemented using the Message Passing Interface (MPI).

For distributed sparse matrix-vector multiplication (SpMV), the computation proceeds similarly to the shared memory case (algorithm 2). The sparse matrix is divided across nodes, and each node computes its local segment of the output vector y. At the end of each iteration, partial results are assembled to produce the global vector, necessitating inter-node communication.

While the thread scheduling and memory locality issues discussed in 3.4.1 remain relevant, distributed memory systems introduce the additional challenge of communication overhead. As shown in Table 2.1, inter-node communication is significantly more expensive than memory accesses. Consequently, minimizing the total communication volume per SpMV iteration is critical for performance.

3.5.1 Load Balancing and Graph Partitioning

Even with a well-balanced distribution, excessive communication between nodes can become a performance bottleneck if data dependencies span many partitions. An effective partitioning strategy must therefore strike a careful balance between distributing the computational load evenly and reducing the amount of communication required between ranks. This trade-off is naturally captured by the graph partitioning problem.

The graph partitioning problem involes dividing an undirected graph $G(V, E)$, where V is the set of vertices and E is the set of edges into K disjoint subsets (or partitions) such that each subset contains a comparable amount of vertices, and the number of edges connecting different subsets is minimized. Each vertex $v_i \in V$ may be assigned a weight w_i, and each edge $e_{ij} \in E$ may carry a cost c_{ij}. The sum of weights W_k of each partition P_k must not exceed a specified imbalance threshold ϵ relative to the average weight of each partition. An edge can be classified as either an internal edge (those that connect nodes in the same partition), or an external edge (those that connect vertices in different partitions), and the cutsize is defined as the sum of the costs of all external edges. The goal is to minize the cutsize, while preserving the balance between the size of each partition. This problem is known to be NP-Hard, even for obtaining bipartitions on unweighted graphs (adapted from [2]).

Algorithms that aim to solve the graph partitioning problem rely on heuristics that provide good approximations in practice. Optimizing for one objective at the expense of the other is trivial but undesirable: assigning all vertices to a single partition minimizes communication to zero but results in extreme load imbalance, whereas naïvely assigning vertices to maintain balance without regard to connectivity may lead to high communication costs.

3.5.2 METIS

The graph partitioner used for the experiments in this thesis is called METIS. METIS is a software package for partitioning large irregular graphs, and has other uses such as partitioning meshes and computing fill-reducing orderings of sparse matrices. METIS provides algorithms that are based on multilevel graph partitoning. These algorithms partition the graph through a coarsening and refinement phase. In the coarsening phase, the size of the graph is reduced by way of collapsing the vertices and edges, and then partition the smaller graph. In the refinement phase, the graph is gradually expanded to its original size, and the portion of the graph that is close to the boundary is the primary focus, ensuring the quality of the partition is kept (adapted from [5]). algorithm 3 shows how to partition and reorder the vertices of a graph according to the generated partition, using METIS' K-way partition algorithm.

The way METIS partitions graph does not necessarily optimize for minimizing the communication volume, but rather obtaining partitions that are roughly equal in size. It is possible that there are partitions that have a larger imbalance between the size of partitions, but much smaller communication volumes. In order to obtain partitions that optimize for minimizing communication, it is necessary to look to hypergraph partitioners.

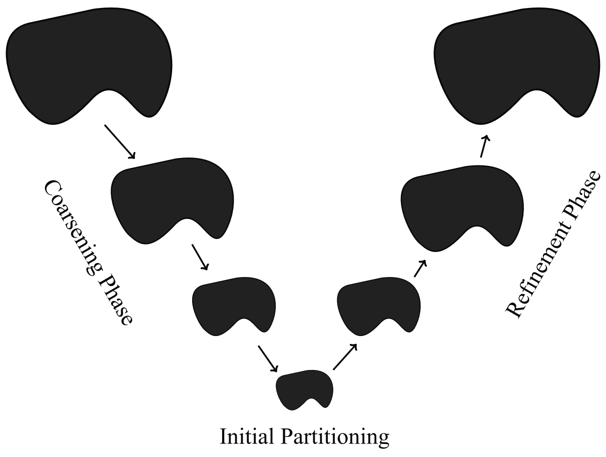

Figure 3.7: Illustration of multilevel graph partitioning (adapted from [5]).

3.5.3 Hypergraph Partitioning

> A *hypergraph* $H = (V, N)$ is defined as a set of vertices V and a set of nets (hyperedges) N among those vertices. Every net $n_j \in N$ is a subset of vertcies, i.e., $n_j \subseteq V$. The vertices in a net n_j are called its *pins* and denoted as $pins[n_j]$. The size of a net is equal to the number of its pins, i.e., $s_j = |pins[n_j]|$. The set of nets connected to a vertx v_i is denoted as $nets[v_i]$. The degree of a vertex is equal to the number of nets it is connected to, i.e., $d_i = |nets[v_i]|$.

[2]

Hypergraph partitioners try to solve the same problems, and employ similar algorithms as those graph partitioners use, however they use these algorithms on hypergraphs, and not regular graphs. Çatalyürek and Ayakanat showed in [2] that using the hypergraph partitioner they developed, they could get up to 63% less communication volume, with an average reduction in communication volume of 30% - 38%, when compared to METIS.

As the improvement of the communication strategies are the main focus of the contents of this thesis, the specific partitioner tool that is used is not the biggest concern, and the programs implemented have opted for the usage of METIS, which as mentioned is a regular graph partitioner. It would be possible to refactor the code in such a way that a hypergraph partitioner such as PaToH can be used.

Algorithm 3: Partitioning and reordering a matrix.

```
Input   : $g$, $n_p$, $p$
Output  : Partitioned and reordered $g$

if $n_p = 1$ then
    p[0] ← 0
    p[1] ← $g.n_r$
    return $g$
partitionVector ← METIS_PartGraphKway(arguments
 specifying constraints for partition)
newId ← [0] · $g.n_r$
oldId ← [0] · $g.n_r$
id ← 0
p[0] ← 0
for $r \in \{0, \ldots r\}$ do
    for $i \in \{0, \ldots, g.n_r\}$ do
        if $partitionVector[i] = r$ then
            oldId[$id$] ← $i$
            newId[$i$] ← id
            id ← id + 1
    partitionVector[$r + 1$] ← id
newRowPtr ← [0] · $g.n_r + 1$
newColIdx ← [0] · $g.n_c$
newValues ← [0] · $g.n_c$
for $i \in \{0, \ldots, g.n_r - 1\}$ do
    d ← g.rowPtr[oldId[i]+1] - g.rowPtr[oldId[i]]
    newRowPtr[i + 1] ← newRowPtr[i] + d
    for $j \in \{0, \ldots, d - 1\}$ do
        newColIdx[newRowPtr[i] + j] ← g.colIdx[g.rowPtr[oldId[i]]
         + j]
        newValues[newRowPtr[i] + j] ← g.values[g.rowPtr[oldId[i]]
         + j]
    for $j \in \{newRowPtr[i], \ldots, newRowPtr[i + 1] - 1\}$ do
        newColIdx[j] ← newId[newColIdx[j]]
g.rowPtr ← newRowPtr
g.colIdx ← newColIdx
g.values ← newValues
return g
```

Chapter 4

Communication Strategies

In parallel implementations of SpMV, effective communication management is critical due to its significant influence on overall performance. Communication often emerges as a bottleneck in distributed-memory systems because the speed at which data moves between nodes is significantly lower than within-node memory access speeds. Consequently, reducing communication volume and optimizing communication patterns can yield substantial performance improvements.

This chapter evaluates a series of progressively optimized communication strategies employed in distributed-memory parallel SpMV. Starting from the simplest method, exchanging the entire result vector between all nodes, the strategies become increasingly selective and efficient, focusing specifically on exchanging only the essential data elements required by each node. Finally, a communication strategy that is fully memory scalable is described. These approaches leverage knowledge of the matrix structure, partitioning methods, and computational dependencies to minimize communication overhead.

In the subsequent sections, each figure illustrating a communication strategy is based on the graph shown in Figure 4.1 as an example of a partitioned graph. In this example, each colour denotes the rank assigned to the vertices contained in the corresponding partition.

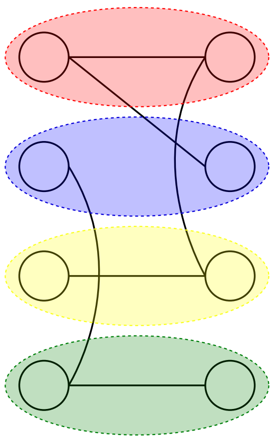

Figure 4.1: Simple graph partitioned amongst rank - each color represents a rank.

4.1 Strategy A - Exchange entire vector

In this strategy, at each iteration every rank sends its locally computed values of y to all other ranks so that each process obtains a complete copy of the output vector before proceeding. This approach can be implemented with the MPI collective operation `MPI_Allgatherv`, which supports different send and receive counts on each rank. 4 shows how this communication pattern can be implemented.

Algorithm 4: Strategy A - Communication Pattern

```
for each iteration do
    spmv(g,x,y)
    MPI_Allgatherv(local_y, sendcount, MPI_DOUBLE, y,
      recvcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD)
    swap pointers of $x$ and $y$
```

Let n_p denote the total number of ranks, and let $|x_i|$ be the number of vector entries assigned to rank i (given by the average; $\frac{|x|}{n_p}$). Since each rank must send its local block size of $|x_i|$ to the other $n_p - 1$ ranks, the total volume of communication per iteration is given by 4.1.

$$\text{Total Communication} = n_p \cdot (n_p - 1) \cdot |x_i| \tag{4.1}$$

Figure 4.2 illustrates the state of the vector before and after communication using this strategy.

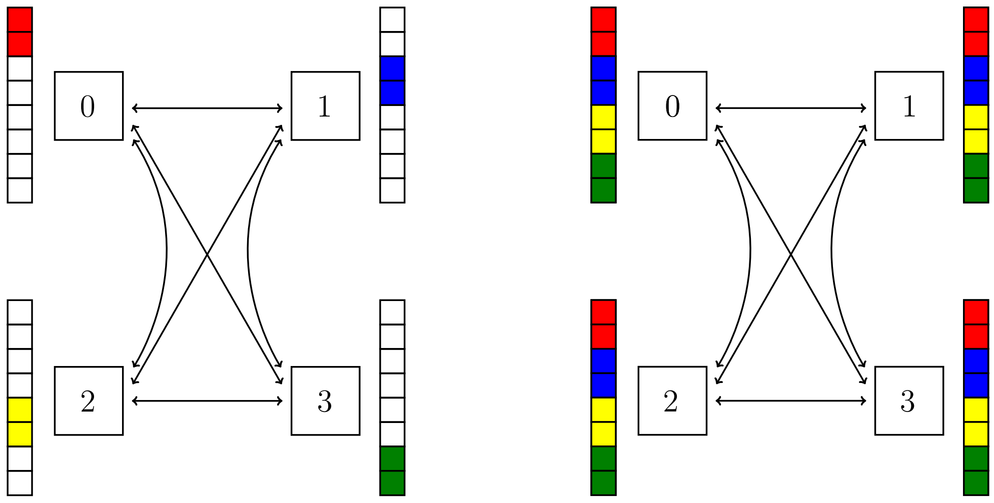

Figure 4.2: Contents of each ranks vector before and after communication under the *Exchange Entire Vector* communication pattern.

4.2 Strategy B - Exchange only separators

An improvement to the previous strategy can be achieved by recognizing that only separator values, those required by multiple processes, must be communicated. Non-separator values are used exclusively by the process that computed them and therefore do not need to be communicated.

Let n_p denote the total number of ranks, and let s_i be the size of the separator of rank i. Since each rank sends its separator to each other rank (i.e. $n_p - 1 otherranks$), then the total communication volume is given by 4.2.

$$\sum_{i=1}^{n_p} (n_p - 1) \cdot s_i \tag{4.2}$$

To facilitate this strategy, separator values are reordered such that they appear at the beginning of each process's local segment of y. Once this structure is established, communication is performed using `MPI_Allgatherv`, transmitting only the subset of y that contains separator values. The number of separators on each process must be known beforehand, which can be computed by counting the number of elements that have neighbours belonging to a different partition.

4.2.1 Reordering

After partitioning the matrix into different parts, we obtain a partition vector p, where the $p[i]$ stores the index of the partition the i^{th} entry in A_p. It is necessary to reorder the entries in A_p in accordance with the partition vector, such that all entries belonging to the same partition are stored in sequence. The algorithm below gives an outline of how this can be achieved. Here n_p is the number of partitions, n_r is the size of A_p, and n_c is the size of A_j and A_x.

Algorithm 5: Reordering of Separators

```
Input    : $n_p, n_r, n_c, p, A_p, A_j, A_x$
Output : Reordered $A_p, A_j, A_x$
newId ← $[0] \cdot n_r$
oldId ← $[0] \cdot n_r$
id ← 0
$p_0 \leftarrow 0$

for $r \in \{0, \dots, n_p - 1\}$ do
    for $i \in \{0, \dots, n_r - 1\}$ do
        if $p[i] = r$ then
            oldId[id] ← $i$
            newId[i] ← id
            id ← id + 1
    $p[r+1]$ ← id

newV ← $[0] \cdot (n_r + 1)$
newE ← $[0] \cdot n_c$
newA ← $[0] \cdot n_c$

for $i \in \{0, \dots, n_r - 1\}$ do
    degree ← $A_p[\text{oldId}[i] + 1] - A_p[\text{oldId}[i]]$
    newV$[i+1]$ ← newV[i] + degree

for $i \in \{0, \dots, n_r - 1\}$ do
    degree ← $A_p[\text{oldId}[i] + 1] - A_p[\text{oldId}[i]]$
    for $j \in \{0, \dots, degree - 1\}$ do
        newE[newV[i]+j] ← $A_j[A_p[\text{oldId}[i]] + j]$
        newA[newV[i]+j] ← $A_x[A_p[\text{oldId}[i]] + j]$
    for $j \in \{newV[i], \dots, newV[i+1] - 1\}$ do
        newE[j] ← newId[newE[j]]

Overwrite $A_p$ ← newV, $A_j$ ← newE, $A_x$ ← newA
```

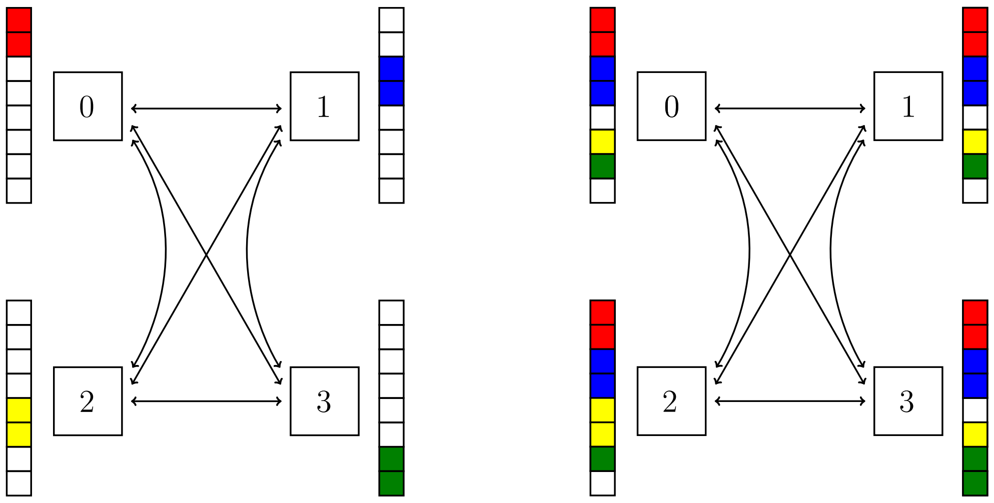

Figure 4.3: Contents of each ranks x vector before and after communication under the *Exchange Separators* communication pattern.

4.3 Strategy C - Exchange only required separators

Further reduciton to the communication volume can be achieved by observing that not all separator values are required by every process. As the number of partitions increases, the set of dependencies between partitions tends towards sparcity. As a consequence of this, certain sets of separators may only need to be communicated to a given subset of processes. Using this strategy, each process only communicates its set of separator values to the processes that require them. The communication pattern in illustrated in Figure 4.4, and Algorithm 6 shows how this can be implemented.

Let n_p be the total number of ranks, n_i be the number of neighbours of rank i, and s_i the size of the separator of rank i. With $n_i \leq n_p$, the total communication volume is given by 4.3.

$$\sum_{i=1}^{n_p} s_i \cdot n_i \tag{4.3}$$

Algorithm 6: Exchange only required separators

```
Input  : y, rank,size, displacements, sendItems, sendCount
Output :

MPI_Requests ← 0
for $r$ = 0; $r$ < size; $r$++ do
    if rank = r or sendItems[r][rank] = 0 then
        continue
    Post non-blocking MPI receive for $sendCount[r]$ elements at
      address $y + displacements[r]$
    MPI_Requests++
for $r$ = 0; $r$ < size; $r$++ do
    if rank = r or sendItems[r][rank] = 0 then
        continue
    Post non-blocking MPI send of $sendCount[rank]$ elements to
      address $y + displacements[rank]$
    MPI_Requests++
Wait for all non-blocking requests to complete
```

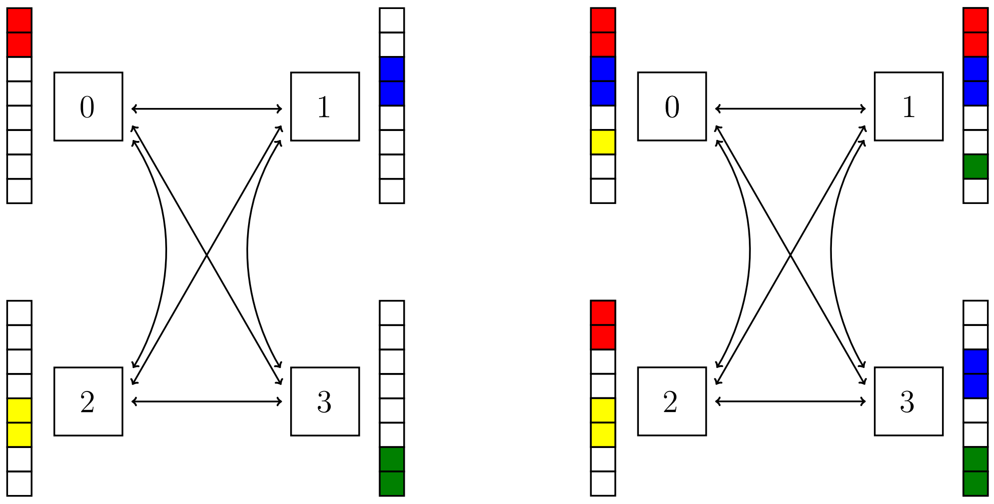

Figure 4.4: Contents of each ranks x vector before and after communication under the *Exchange Reuired Separators* communication pattern.

4.4 Exchange only required separator values

The final strategy aims to minimize communication overhead by transmitting only the exact subset of separator values that are both computed by and required for inter-process computation. If a specific separator value computed by one process is needed by exactly one other process, then only that single recipient receives the value.

This approach eliminates all unnecessary data transfers but introduces additional complexity in managing communication schedules. Dependencies must be mapped at a fine-grained level, and communication patterns must be explicitly tailored to the structure of the matrix and its partitioning.

Let n_p be the total number of ranks, and let d_i be the number of elements in rank is separator that are required by other ranks. Since this communication strategy only communicated these elements, the total communication volume is given by 4.4.

$$\sum_{i=1}^{n_p} d_i \tag{4.4}$$

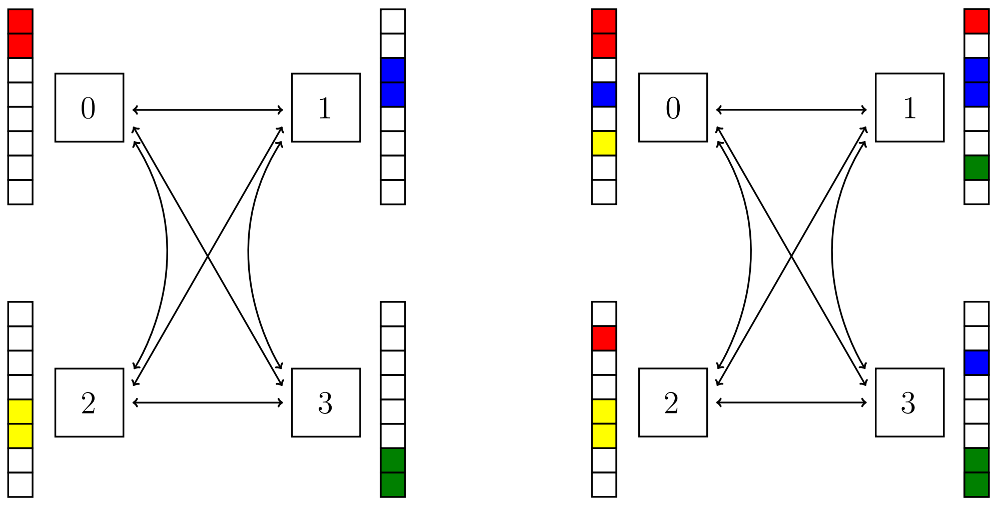

Figure 4.5: Contents of each ranks x vector before and after communication under the *Exchange Reuired Elements* communication pattern.

4.5 Strategy E - Memory Scalable

The communication strategies discussed so far all have a common problem that prevents them from scaling to large matrices. These strategies all store the entire vector x, and will run into performance issues when x is so large that it doesn't fit into memory. Usually, this is not a problem when SpMV is ran on CPUs, as they have large amounts of memory. Even on GPUs this problem might not be encountered, as modern GPUs have sufficient memory for large matrices.

Instead of storing the entire vector, each rank only stores its local part of the vector. In addition, it is necessary to allocate enough space for the separators elements that are needed from the other ranks. In order to achieve this, x is renumbered such that every ranks part of the vector is 0-indexed. For the local part of the x vector this is done by simply subtracting the position of the first local entry assigned to that rank from each entry in the local vector. It is slightly more complicated to renumber the separator elements, and the technicalities of how to do this will not be discussed here.

Algorithm 7: Exchange Required Separators

```
Input  : c, $x$, rank, size
Output : Updated vector $x$
totalSend, totalRecv ← 0
for $i$ = 0; $i < size$; $i$++ do
    totalSend ← totalSend + c.sendCount[i]
    totalRecv ← totalRecv + c.receiveCount[i]
Allocate sendBuffer[totalSend], recvBuffer[totalRecv]
sendOffset ← 0
for $i$ = 0; $i < size$; $i$++ do
    for $j$ = 0; $j < c.sendCount[i]$; $j$++ do
        sendBuffer[sendOffset++] ← $x$[c.sendItems[i][j]]
Compute sendDispls, recvDispls as prefix sums of c.sendCount,
  c.receiveCount
MPI_Ialltoallv(sendBuffer, c.sendCount, sendDispls, recvBuffer,
  c.receiveCount, recvDispls)
MPI_Waitall()
recvOffset ← 0
for $i$ = 0; $i < size$; $i$++ do
    for $j$ = 0; $j < c.receiveCount[i]$; $j$++ do
        $x$[c.receiveItems[i][j]] ← recvBuffer[recvOffset++]
Free sendBuffer, recvBuffer, sendDispls, recvDispls
```

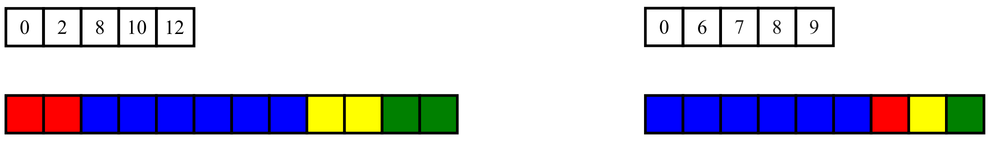

Figure 4.6: Global indexing vs. Local indexing of x.

Chapter 5

Results

5.1 Theoretical Maximum

As has been established, the maximum performance scales with the available memory bandwidth of the system, and not with the amount of processes used. For SpMV, a total of 6 bytes are read per FLOP, which means that the theoretical maximum is given by 5.1. This is under the assumption that all resources on the system is dedicated to the SpMV operation, which is not the case in reality. At most, somewhere in the neighbourhood of 80% of the systems resources can be expected to be utilized.

$$\text{Maximum Theoretical Performance} = \frac{\text{Memory bandwidth}}{6} \tag{5.1}$$

5.2 Matrices

For the results presented in the following sections, the matrices shown in Figure 5.1 were selected as they are in general well-structure and symmetrical.

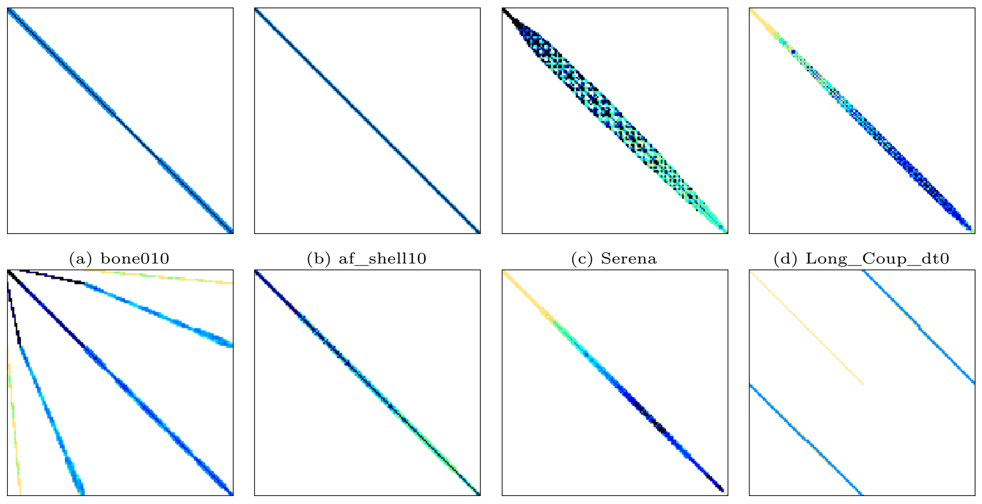
(a) bone010 (b) af_shell10 (c) Serena (d) Long_Coup_dt0
(e) dielFilterV3real (f) Cube_Coup_dt0 (g) Bump_2911 (h) nlpkkt200

Figure 5.1: Matrices used to generate results.

Name	**Purpose**
bone010	Trabecular Bone Micro-Finite Element Model
af_shell1	Sheet metal forming matrix
Serena	Gas reservoir simulation for CO_2 sequestration
Long_Coup_dt0	3D coupled consolidation problem (geological formation)
dielFilterV3real	High-order vector finite element method in EM
Cube_Coup_dt0	3D coupled consolidation problem (3D cube)
Bump_2911	3D geomechanical reservoir simulation
nlpkkt200	Symmetric indefinite KKT matrix

5.3 Overview of Experiments

The experiments that have been ran in order to obtain the results in the following thesis have made use of systems on the Experimental Infrastructure for Exploration of Exascale Computing (eX^3).

For each experiment, every matrix has been put through 100 iterations of SpMV, and every communication strategy has been tested on different configurations. For the single node experiments, each communication strategy has been ran with the number of MPI ranks being doubled until every available physical core on the chip is assigned one MPI rank. For multi node experiments on single socketed nodes, one MPI rank is assigned to each rank, and shared memory parallelization has been used within each node. On dual socketed nodes, experiments placing one MPI rank per node and placing one MPI rank per socket have been ran. All programs have been compiled with `-march=native` and `-O3` compilation flags.

In the following presentation of the results, **??** gives an overview of what the different strategies will be referred to, both when analyzing the results, and in the legends of the figures.

Strategy name	**Strategy Description**
Strategy A	Exchanges entire local vector.
Strategy B	Exchanges entire separator.
Strategy C	Exchanges required separator.
Strategy D	Exchanges required separator elements
Strategy E	Exchanges required separator elements, memory scalable

5.4 AMD EPYC 7601- Single Node Performance

The results in this section illustrates the performance of the communication strategies when performed on a single node containing two AMD EPYC 7601 chips.

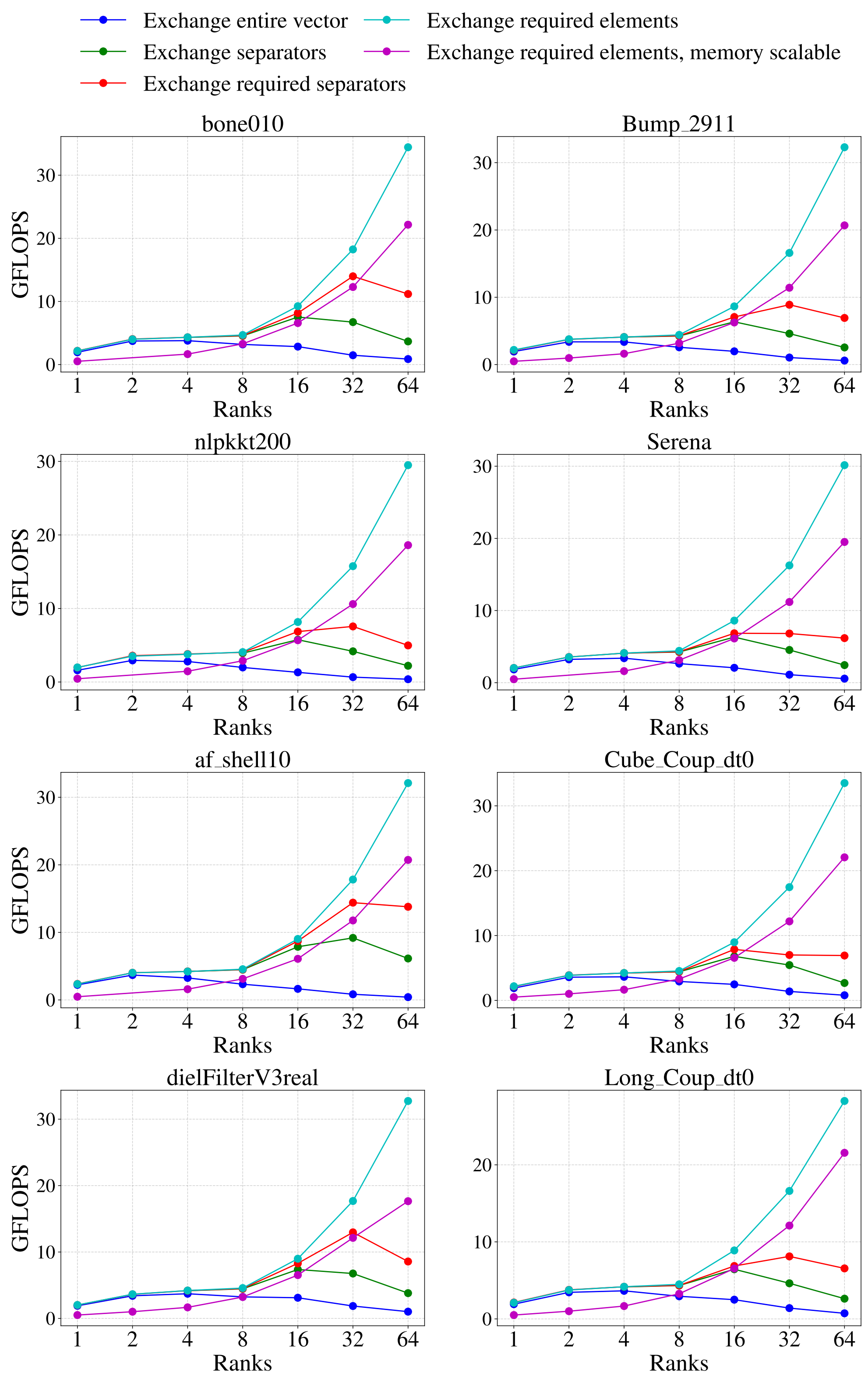

Figure 5.2: Single Node - AMD EPYC 7601

Figure 5.2 presents the achieved GFLOPS for each communication strategy. The performance trends largely align with expectations: each successive strategy generally exhibits improved performance over its predecessor. However, an exception arises with strategy E, which consistently underperforms relative to strategy D. As detailed in Algorithm 7, strategy E incurs a significant overhead due to its requirement to pack and unpack every individual element exchanged across separator boundaries. This repeated packing and unpacking operation introduces substantial computational and memory overhead, which offsets the potential benefits of the strategy and explains its inferior performance. It is however important to keep in mind that the goal of strategy E is not necessarily to achieve the absolute highest possible GFLOPS performance, but rather to ensure that it is scalable to large matrices, and is able to run on architectures that have lower local memory.

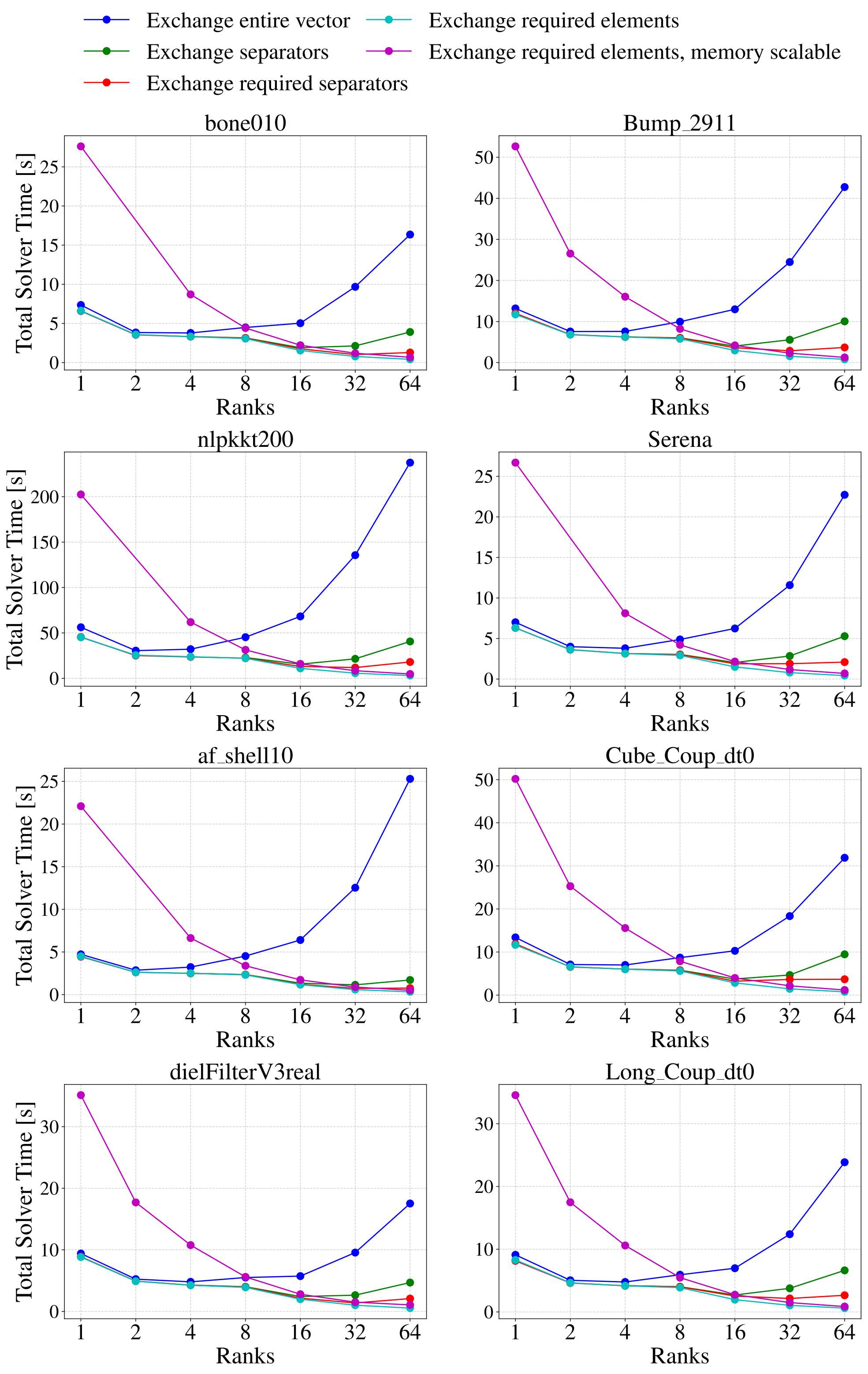

Figure 5.3: Total execution time of each communication strategy on a single node using the AMD AMD EPYC 7601 chip.

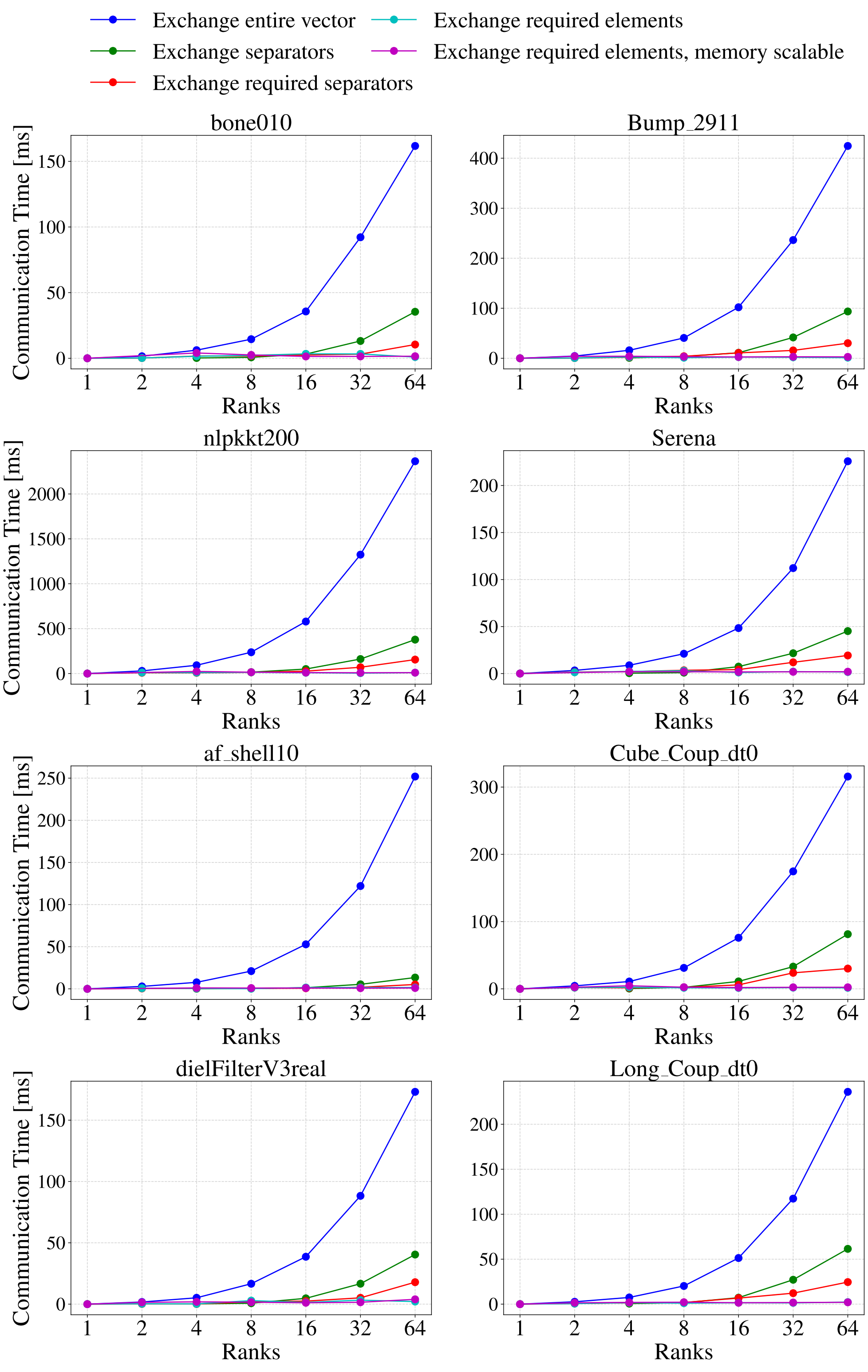

Figure 5.4: Communication time component of each strategy on a single node using the AMD EPYC 7601 chip.

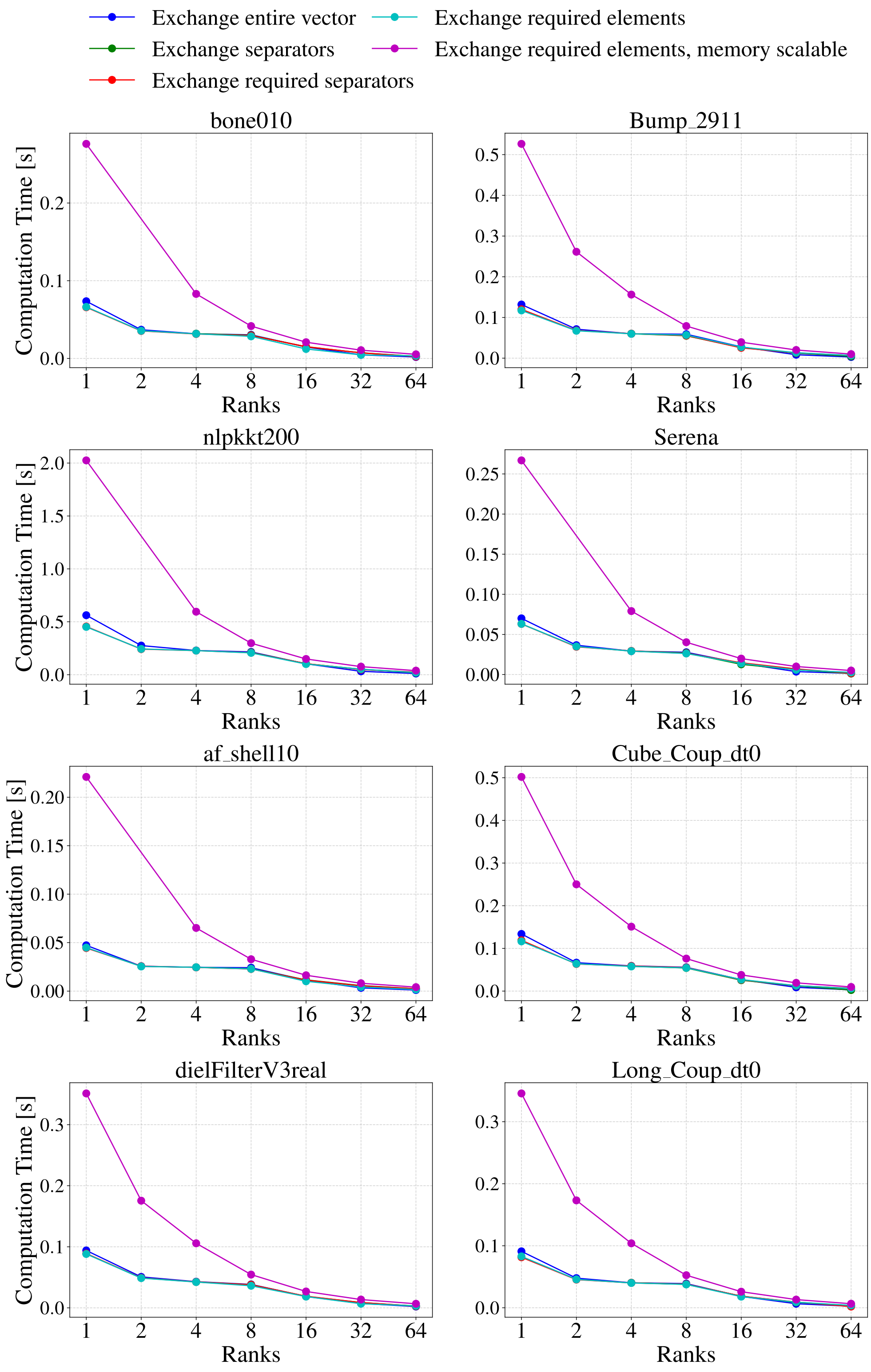

Figure 5.5: Computation time component of each strategy on a single node using the AMD EPYC 7601 chip.

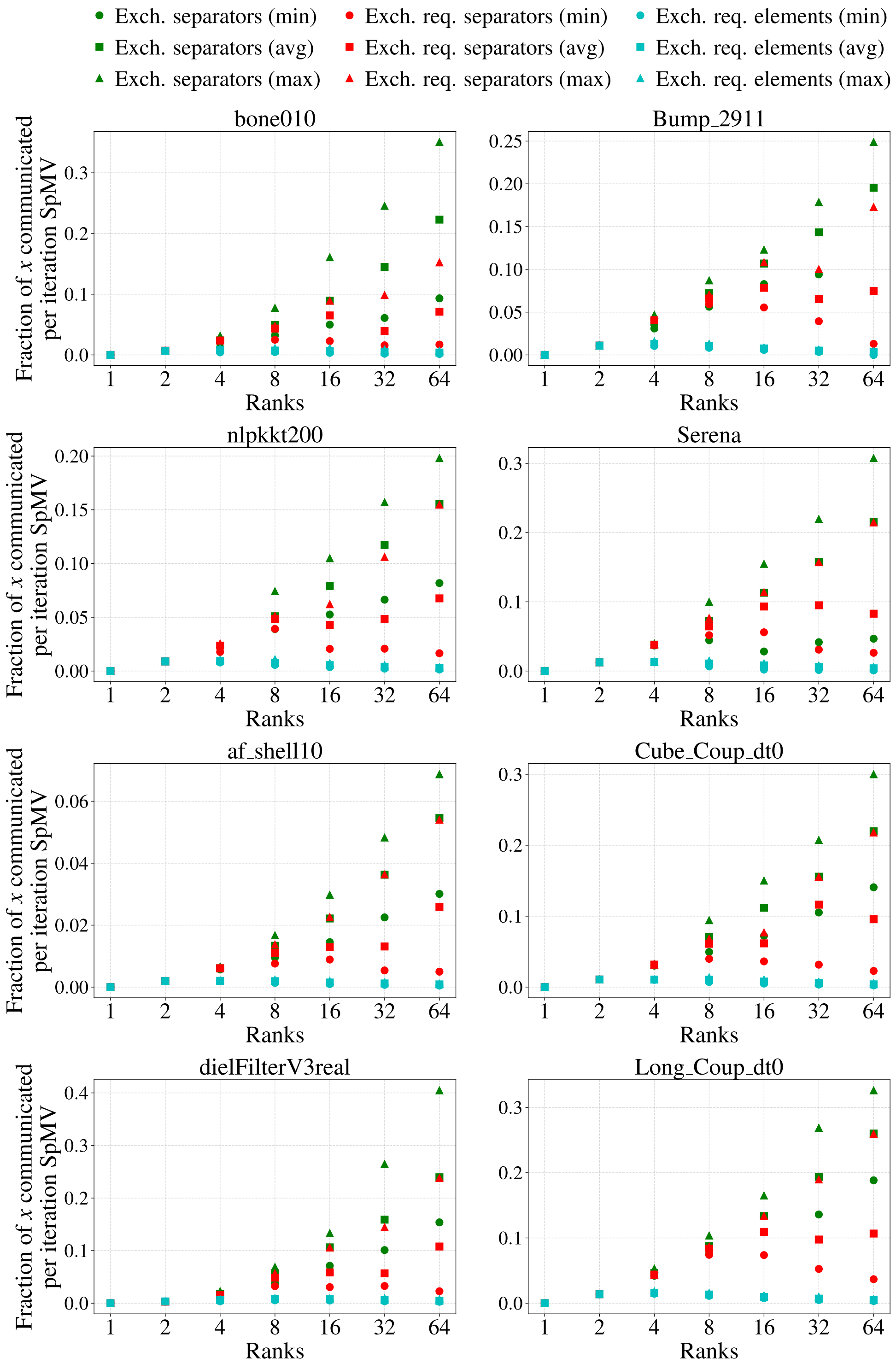

Figure 5.6: Fraction of x communicated per SpMV iteration each strategy on a single node using the AMD EPYC 7601 chip.

5.5 (Rome16q) AMD EPYC 7302P

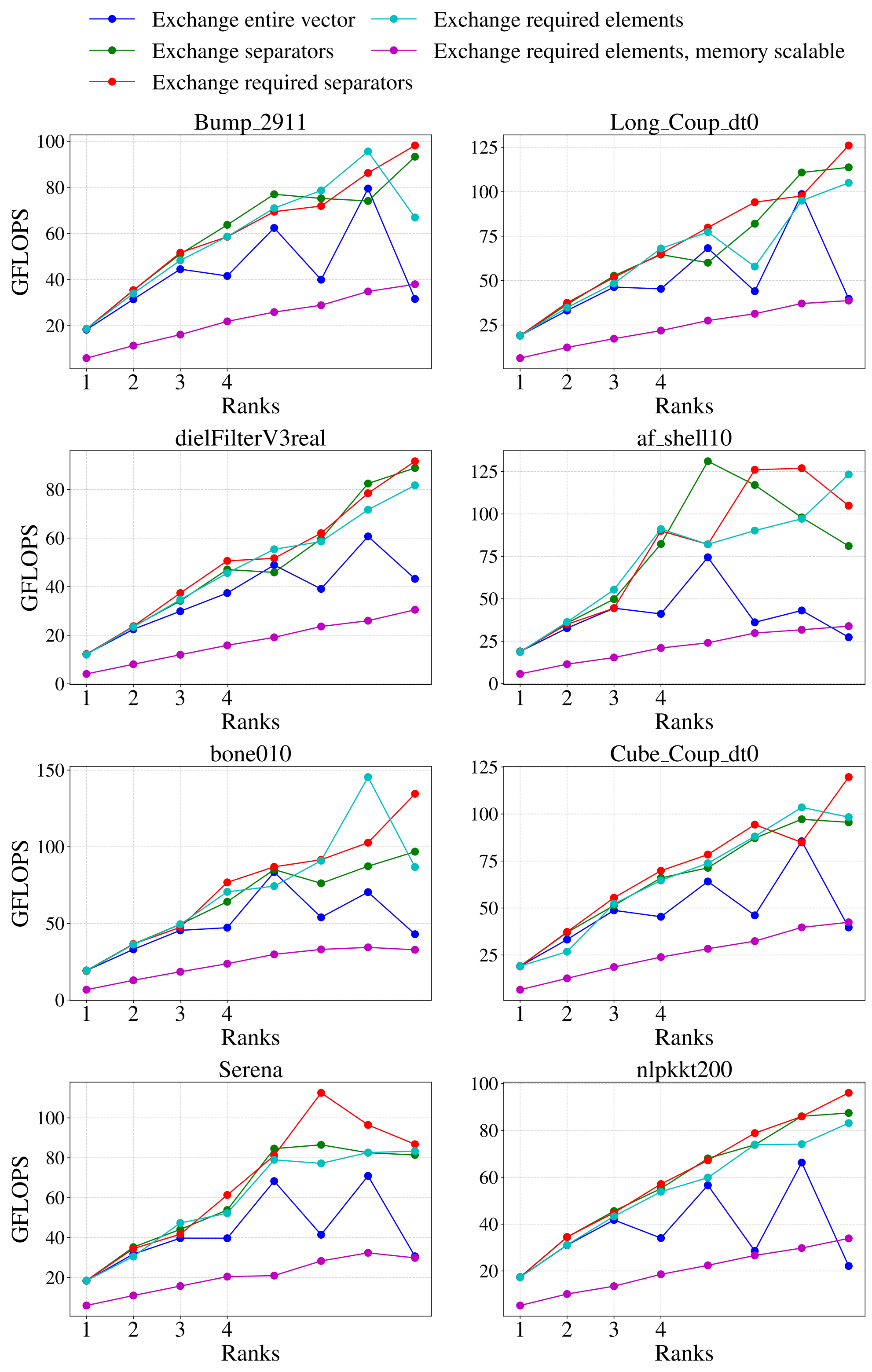

Figure 5.7

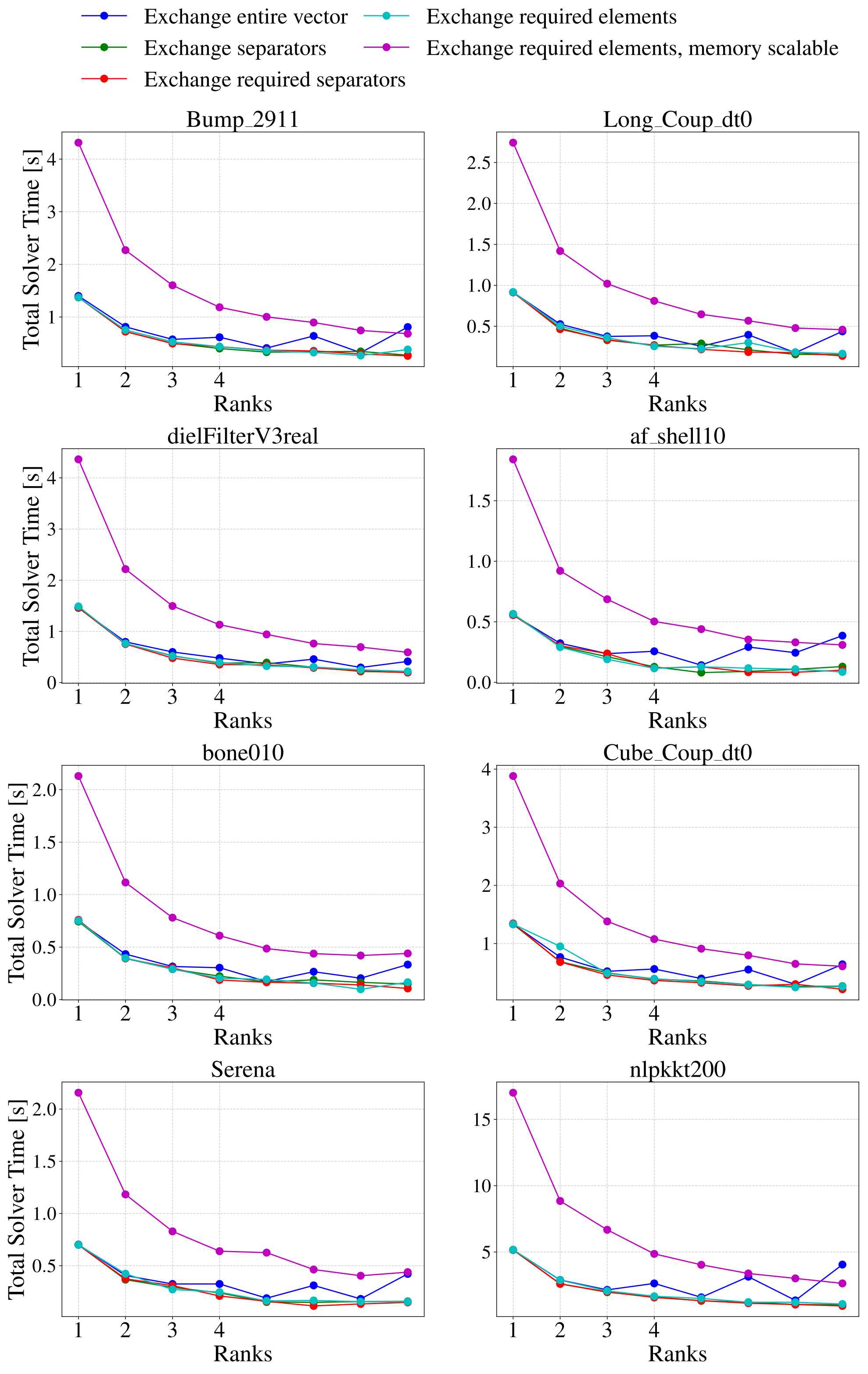
Exchange entire vector
Exchange separators
Exchange required separators
Exchange required elements
Exchange required elements, memory scalable
Bump_2911
Long_Coup_dt0
dielFilterV3real
af_shell10
bone010
Cube_Coup_dt0
Serena
nlpkkt200
Total Solver Time [s]
Ranks

Figure 5.8

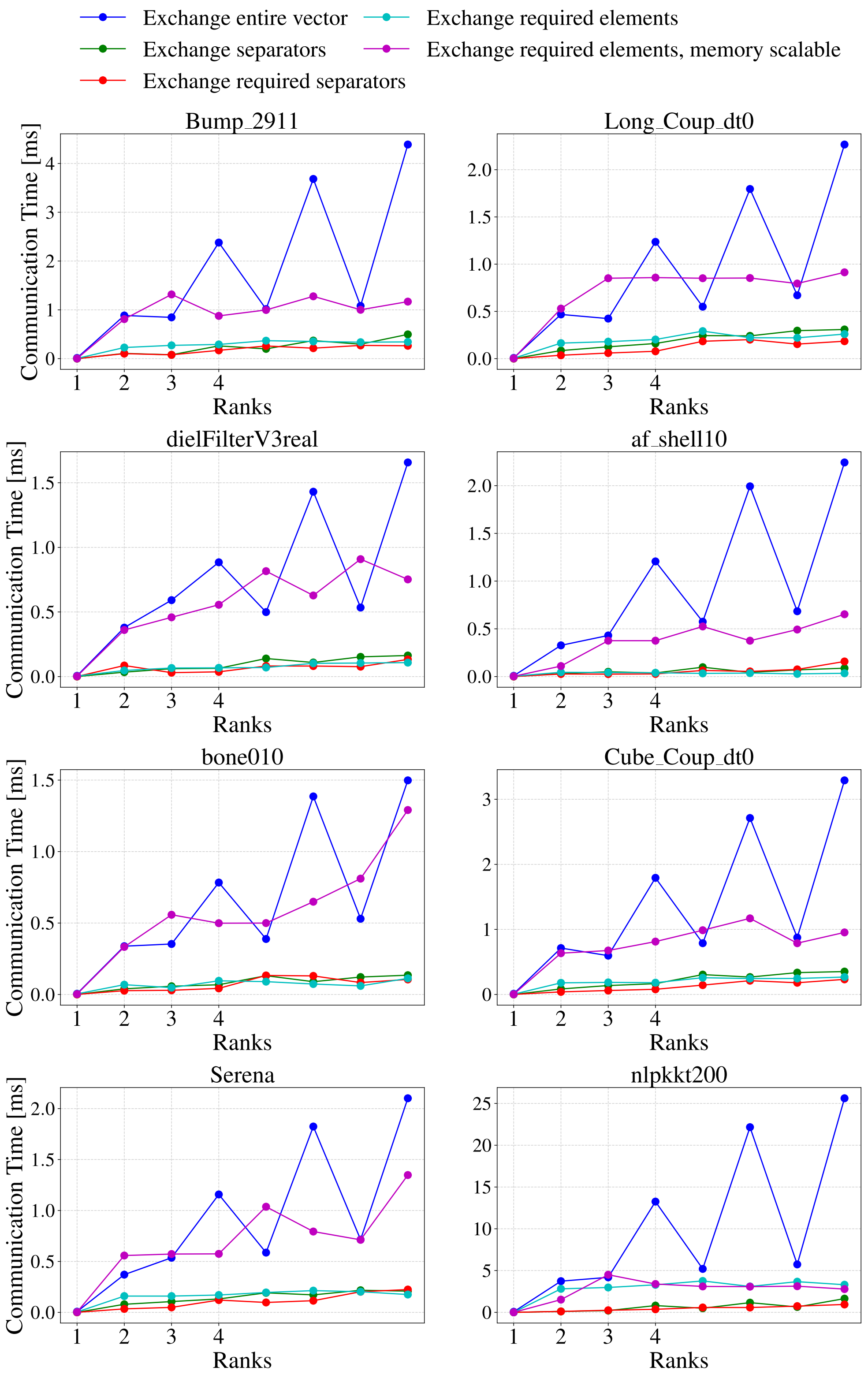
Exchange entire vector
Exchange separators
Exchange required separators
Exchange required elements
Exchange required elements, memory scalable
Bump_2911
Long_Coup_dt0
dielFilterV3real
af_shell10
bone010
Cube_Coup_dt0
Serena
nlpkkt200
Communication Time [ms]
Ranks
1
2
3
4

Figure 5.9

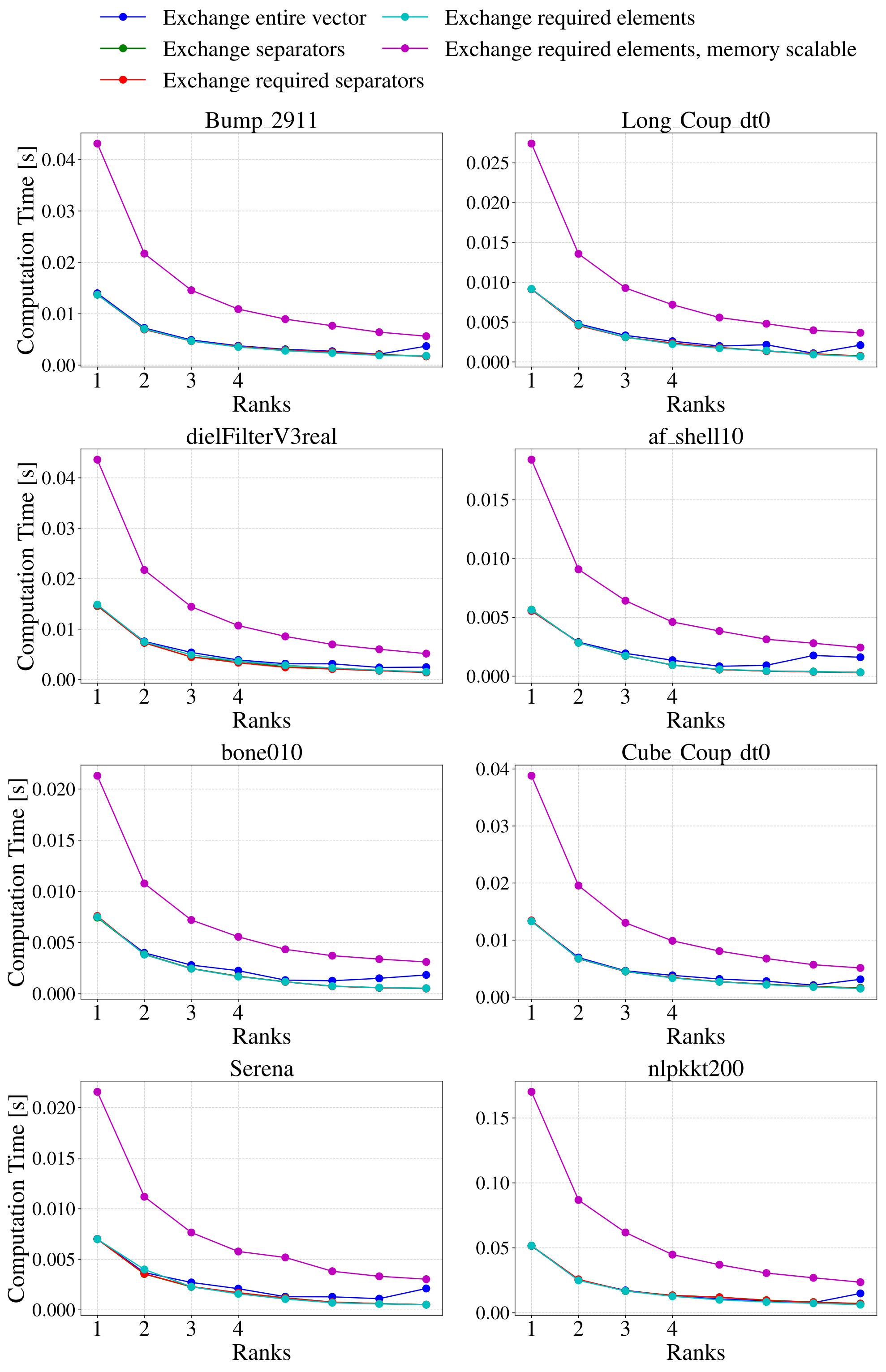

Figure 5.10: Tcomp 2X4 Multi Rome16Q

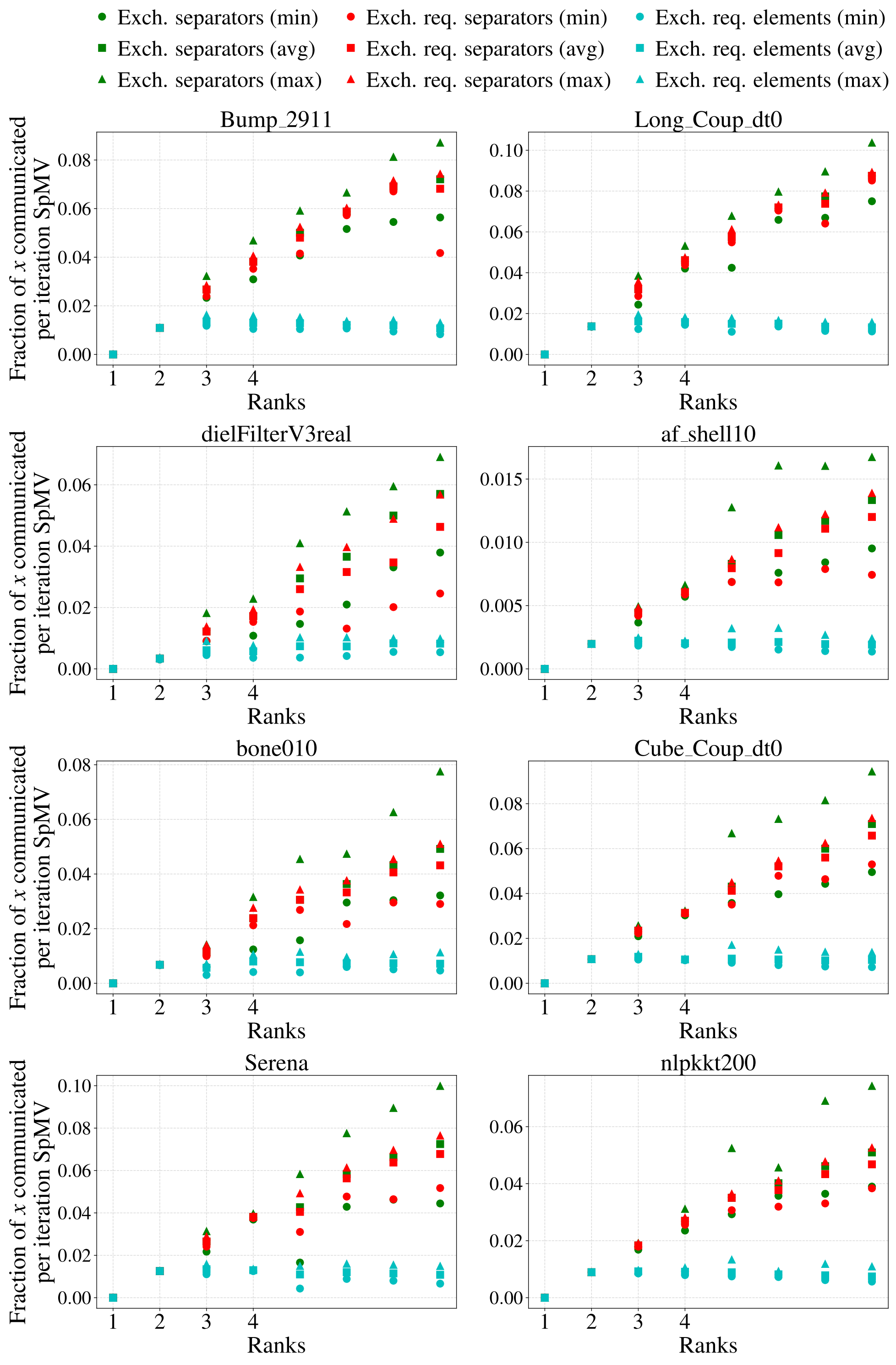
● Exch. separators (min)
■ Exch. separators (avg)
▲ Exch. separators (max)
● Exch. req. separators (min)
■ Exch. req. separators (avg)
▲ Exch. req. separators (max)
● Exch. req. elements (min)
■ Exch. req. elements (avg)
▲ Exch. req. elements (max)
Bump_2911
Long_Coup_dt0
dielFilterV3real
af_shell10
bone010
Cube_Coup_dt0
Serena
nlpkkt200
Fraction of x communicated per iteration SpMV
Ranks
1
2
3
4
0.00
0.02
0.04
0.06
0.08
0.10
0.000
0.005
0.010
0.015

Figure 5.11

5.6 (defq) AMD EPYC 7601

5.7 (defq multi) AMD EPYC 7601

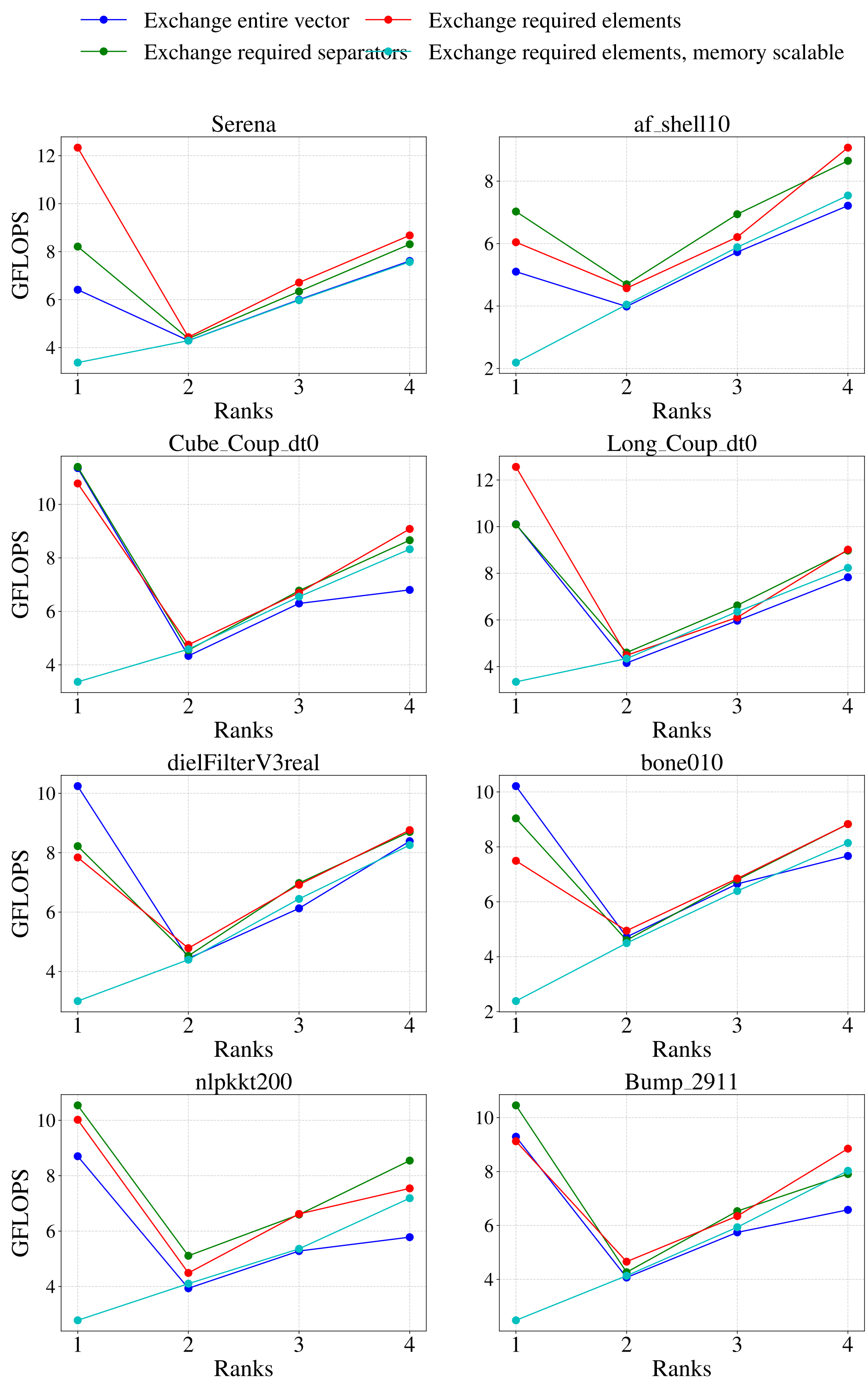

Figure 5.12: multi Node - AMD EPYC 7601

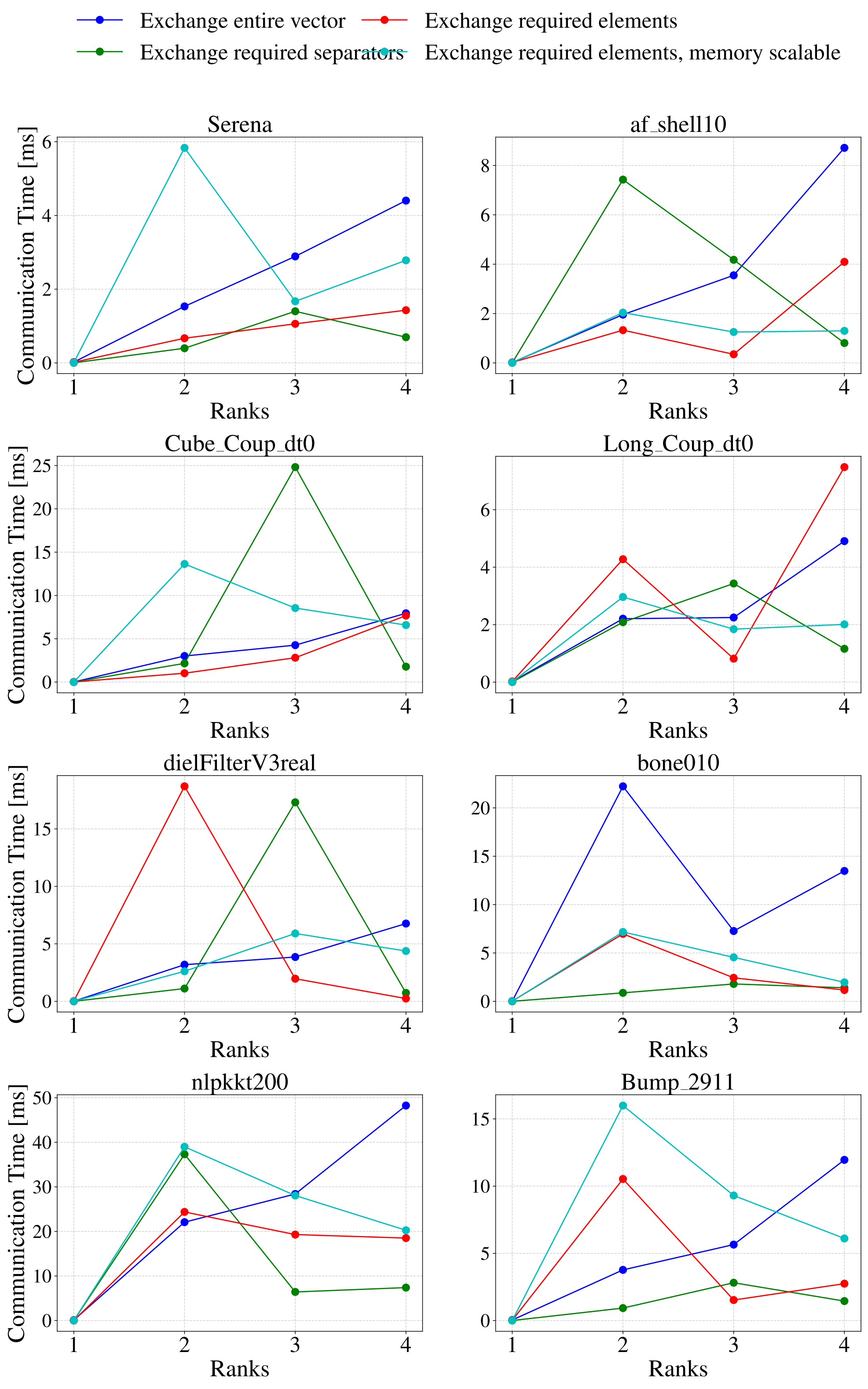
Exchange entire vector
Exchange required elements
Exchange required separators
Exchange required elements, memory scalable
Serena
af_shell10
Cube_Coup_dt0
Long_Coup_dt0
dielFilterV3real
bone010
nlpkkt200
Bump_2911
Communication Time [ms]
Ranks

Figure 5.13

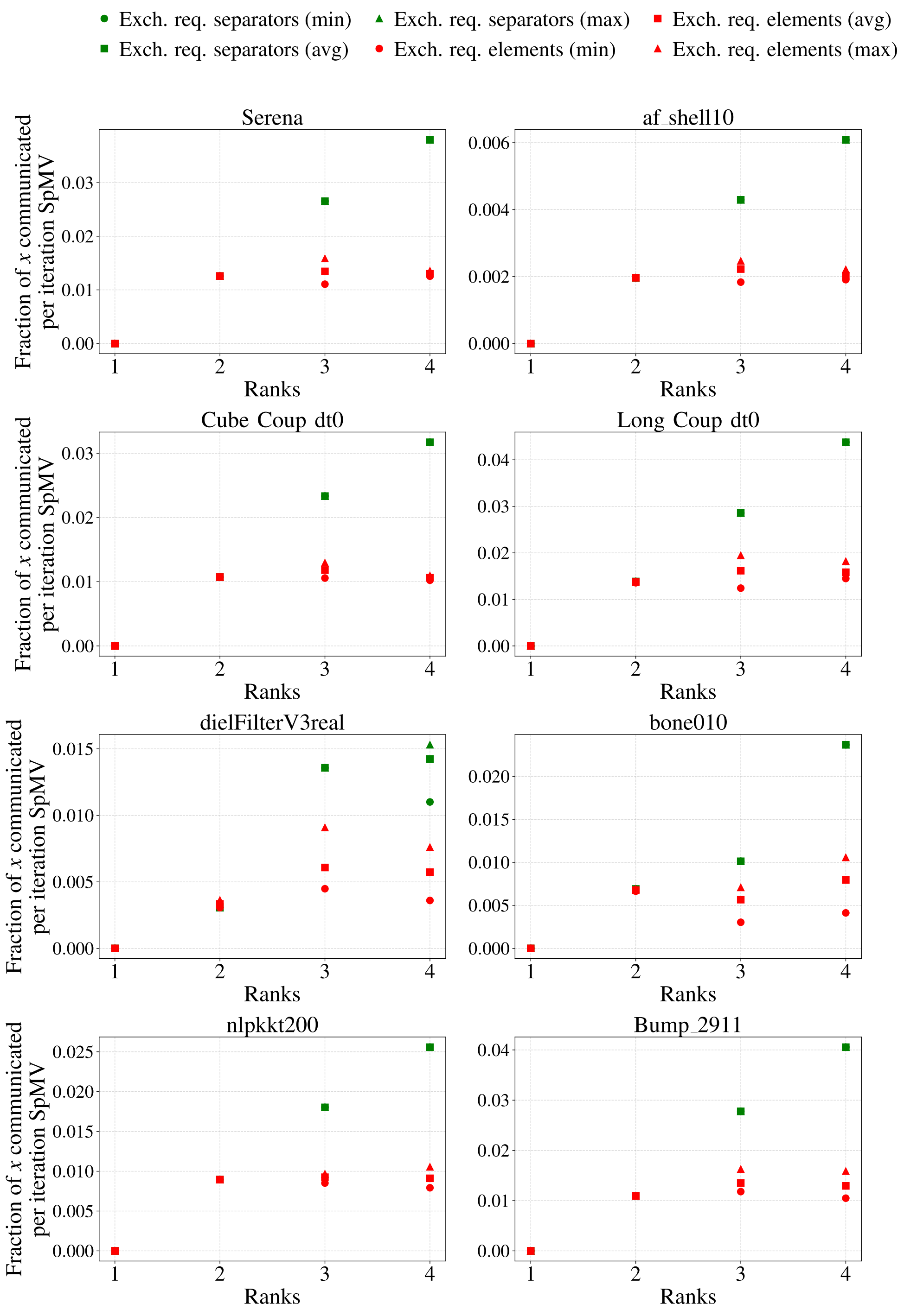
● Exch. req. separators (min)
▲ Exch. req. separators (max)
■ Exch. req. elements (avg)
■ Exch. req. separators (avg)
● Exch. req. elements (min)
▲ Exch. req. elements (max)
Serena
af_shell10
Cube_Coup_dt0
Long_Coup_dt0
dielFilterV3real
bone010
nlpkkt200
Bump_2911
Fraction of x communicated per iteration SpMV
Ranks

Figure 5.14

5.7.1 2 proc per node

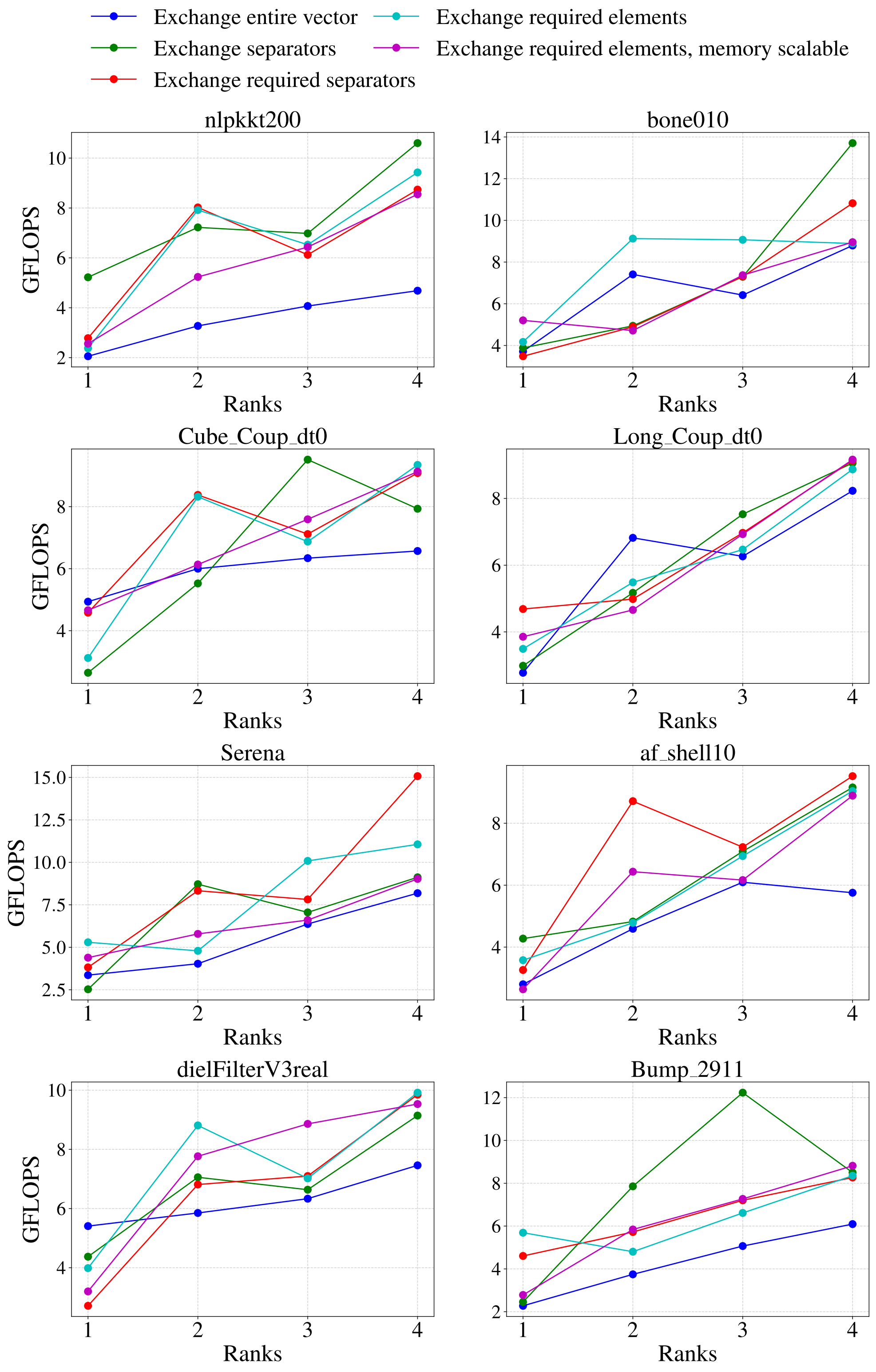

Figure 5.15: Multi Node - AMD EPYC 7601

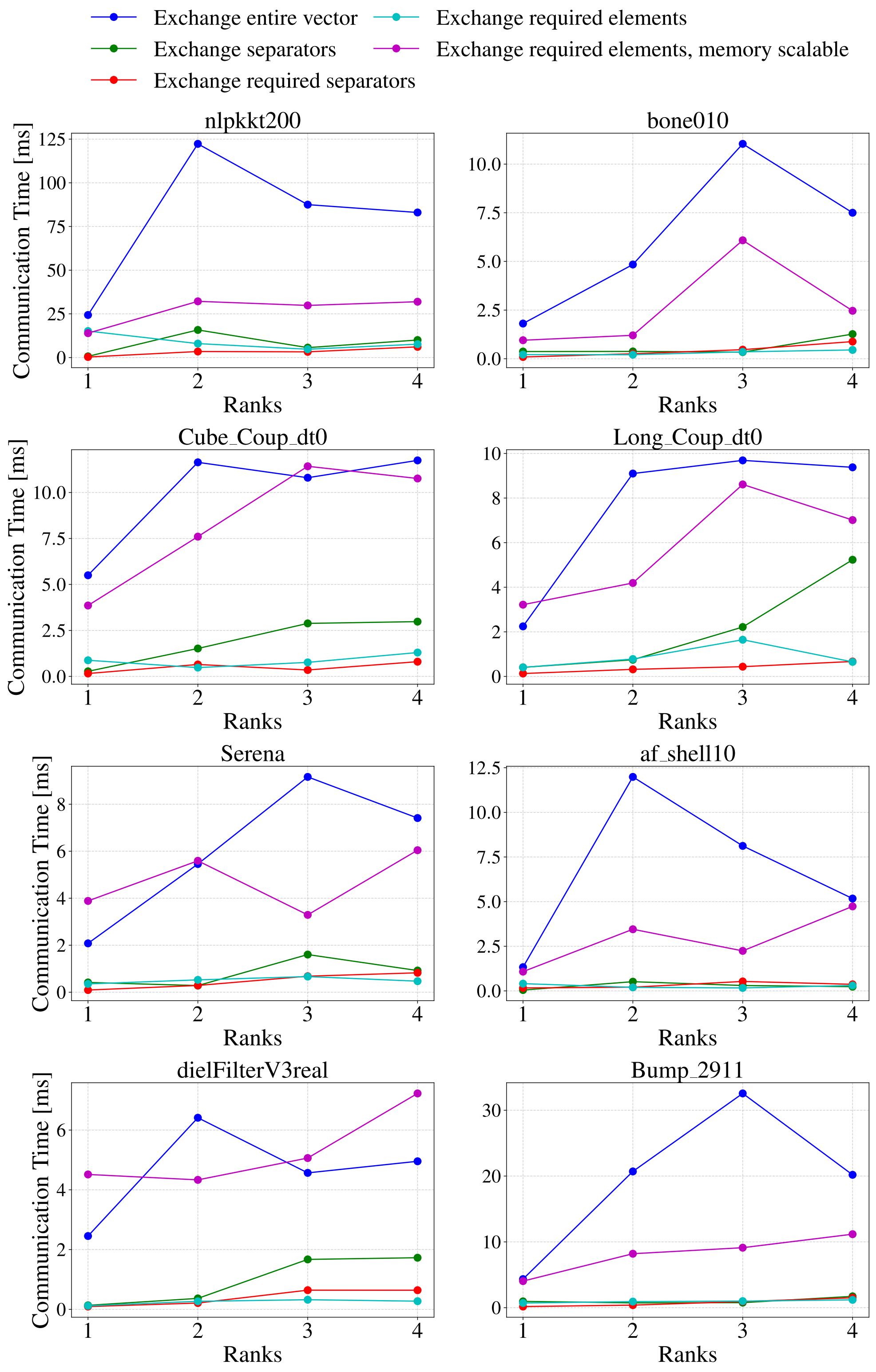
Exchange entire vector
Exchange separators
Exchange required separators
Exchange required elements
Exchange required elements, memory scalable
nlpkkt200
bone010
Cube_Coup_dt0
Long_Coup_dt0
Serena
af_shell10
dielFilterV3real
Bump_2911
Communication Time [ms]
Ranks

Figure 5.16

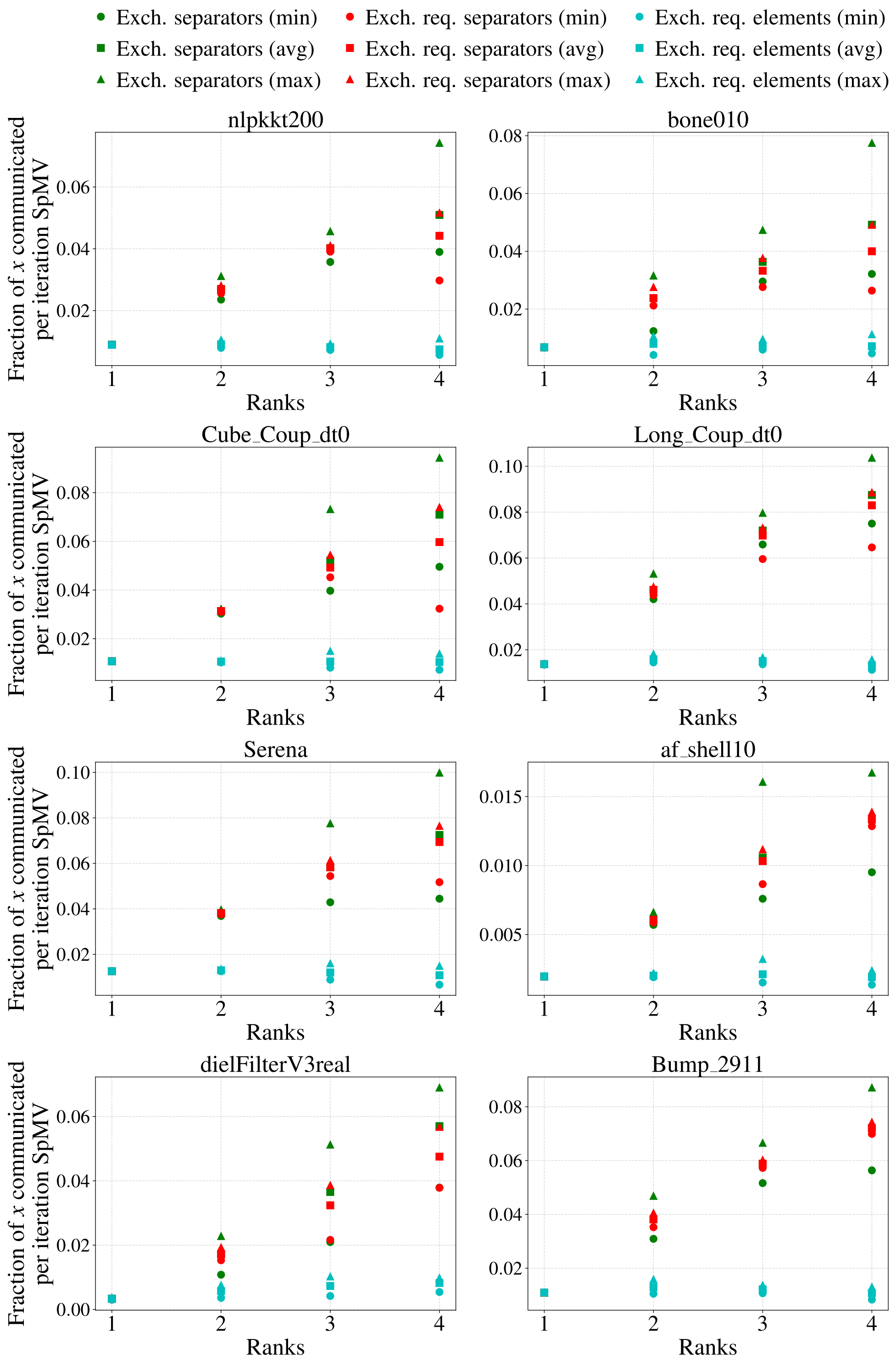
Exch. separators (min)
Exch. separators (avg)
Exch. separators (max)
Exch. req. separators (min)
Exch. req. separators (avg)
Exch. req. separators (max)
Exch. req. elements (min)
Exch. req. elements (avg)
Exch. req. elements (max)
nlpkkt200
bone010
Cube_Coup_dt0
Long_Coup_dt0
Serena
af_shell10
dielFilterV3real
Bump_2911
Fraction of x communicated per iteration SpMV
Ranks

Figure 5.17

Table 5.1: Hardware used in our experiments. STREAM Triad was run with the `-march=native` and `-O3` compilation flags.

	Xeon-A	FPGAQ	Naples	Rome	Milan	TX2	Hi1620
CPUs	Intel Xeon Gold 6130	AMD Epyc 7413	AMD Epyc 7601	AMD Epyc 7302P	AMD Epyc 7763	Cavium TX2 CN9980	HiSilicon Kunpeng 920-6426
Instr. set	x86-64	x86-64	x86-64	x86-64	x86-64	ARMv8.1	ARMv8.2
Microarch.	Skylake	Zen 3	Zen	Zen 2	Zen 3	Vulcan	TaiShan v110
Sockets	2	2	2	1	2	2	2
Cores	2×16	2×24	2×32	1×16	2×64	2×32	2×64
Freq. [GHz]	1.9–3.6	2.6–3.6	2.7–3.2	1.5–3.3	2.5–3.5	2.0–2.5	2.6
L1I/core [KiB]	32	64	64	32	32	32	64
L1D/core [KiB]	32	32	32	32	32	32	64
L2/core [KiB]	1024	512	512	512	512	256	512
L3/socket [MiB]	22	128	64	16	256	32	64
Mem. channels	2×6	2×8	2×8	1×8	2×8	2×8	2×8
Bandwidth [GB/s]	256	256(to be added)	342	204.8	409.6	342	342
Triad [GB/s]	147.1	137.4(to be added)	169.7	90.9	256.5	236.4	260.4

Chapter 6

Related Work

Merrill and Garland [8] propose a merge-based parallel SpMV algorithm that operates directly on the Compressed Sparse Row (CSR) format without requiring reformatting or auxiliary data structures. Their method employs a 2D merge-path decomposition that ensures strictly balanced workloads across processing elements, independent of row length variability. This design addresses the primary bottleneck of irregular row distributions in parallel SpMV, which often lead to performance inconsistencies in traditional row- and nonzero-splitting strategies. Evaluated across over 4,000 matrices, their approach demonstrated superior performance consistency and scalability, particularly on architectures with constrained local memories such as NUMA and GPUs. This work offers a compelling alternative to format-specific optimizations by achieving portability and high throughput directly on CSR representations.

Trotter et al. [10] present a comprehensive evaluation of matrix reordering strategies for enhancing sparse matrix-vector multiplication (SpMV) performance on multicore CPUs. Evaluating six reordering algorithms over 490 matrices on eight architectures, they find that reorderings based on graph and hypergraph partitioning (e.g., METIS, PaToH) offer the most consistent SpMV performance benefits. Their results indicate that performance improvements arise primarily from better data locality and reduced off-diagonal nonzero density, while bandwidth and profile reductions are less influential. They also underscore the architectural sensitivity of reordering effectiveness, suggesting that careful algorithm-architecture matching is critical. This study provides empirical validation for the effectiveness of reordering as a pre-processing optimization for SpMV, especially in shared-memory systems.

Bibliography

[1] Luk Burchard, Kristian Gregorius Hustad, Johannes Langguth, and Xing Cai. Enabling unstructured-mesh computation on massively tiled ai processors: An example of accelerating in silico cardiac simulation. *Frontiers in Physics*, Volume 11 - 2023, 2023.

[2] U.V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[3] Gautam Gupta, Sivasankaran Rajamanickam, and Erik G. Boman. GAMGI: Communication-reducing algebraic multigrid for gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 61–75. ACM, 2024.

[4] Cerebras Systems Inc. Wse-3 datasheet. https://cerebras.net/product/system/, 2024. DS03 v3 821.

[5] George Karypis and Vipin Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Technical report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1997. Version 3.0.3.

[6] Lawrence Livermore National Laboratory. Introduction to parallel computing tutorial. http://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial, 2024. Accessed: 2025-05-08.

[7] Nakul Manchanda and Karan Anand. Non-uniform memory access (numa). *New York University*, 4, 2010.

[8] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689, 2016.

[9] Peter Norvig. Latency numbers every programmer should know. https://norvig.com/21-days.html#Latency, 2021. Accessed: 2025-04-29.

[10] James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. Bringing order to sparsity: A sparse matrix reordering study on multicore cpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Cong Zheng, Shuo Gu, Tong-Xiang Gu, Bing Yang, and Xing-Ping Liu. Biell: A bisection ellpack-based storage format for optimizing spmv on gpus. *Journal of Parallel and Distributed Computing*, 74(7):2639–2647, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.