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

1.1 Research Question

This thesis aims to investigate the impact different communication strategies have on the performance of parallel SpMV. The implementation of SpMV uses a shared memory layout on each socket of a node, and distributed memory for communication across nodes.

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. $\mathcal{O}(n)$ 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

Definition 2.2.1 (Separator)**.** In the context of SpMV, a separator is a node in the graph that has an edge that strides between two partitions.

2.3 Amdahl's Law

Amdahl's Law provides a theoretical framework for understanding the limits of performance improvement when additional computational resources are applied to a given problem. It quantifies the potential speedup achieved by optimizing a specific portion of a system, emphasizing that the overall gain is constrained by the proportion of time the optimized component contributes to execution.

Definition 2.3.1 (Amdahl's Law)**.** The maximum achievable speedup of a computation is limited by the fraction of execution time that remains sequential, even when an arbitrarily large number of parallel resources is employed.

In the context of parallel computing, this principle highlights that while increasing the number of processing units can accelerate the parallelizable portion of a workload, the sequential fraction imposes a fundamental performance ceiling. Formally, if S denotes the total speedup, t_p represents the fraction of execution time that can be parallelized, and s_p is the speedup achieved for that parallelizable portion, Amdahl's Law is expressed as:

$$S = \frac{1}{(1 - t_p) + \frac{t_p}{s_p}} \tag{2.1}$$

This equation reveals that as $s_p \to \infty$, the theoretical maximum speedup approaches $\frac{1}{1-t_p}$, illustrating that the non-parallelizable portion becomes the dominant limiting factor in scalability.

2.4 Latency

It is worthwhile to speak on the typical latency numbers for various operations on a computer.

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 [**?**]

2.4.1 NUMA Architecture

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.

In NUMA systems, accessing local memory, that is, memory located on the same node as the processor, provides significantly lower latency and higher bandwidth compared to accessing remote memory located on other nodes. Thus, maintaining good memory locality is critical for achieving optimal performance. If data accessed by a processor primarily resides in remote memory, performance degradation may occur due to increased latency and reduced bandwidth, particularly when frequent remote memory accesses occur.

Because of this architecture, software must be carefully designed to maximize locality and minimize remote memory access. Strategies such as the first-touch policy, where memory pages are allocated based on the location of the thread first accessing the memory, are commonly employed to enhance memory locality and overall application performance on NUMA systems.

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 [**?**].

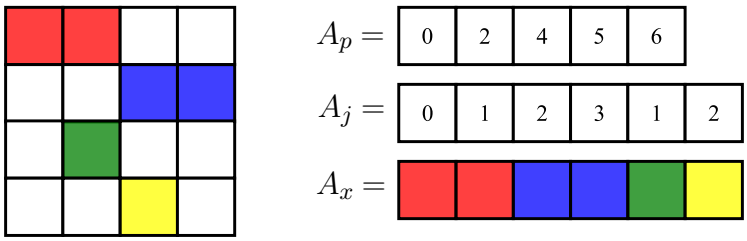


Figure 3.1: Example matrix represented in CSR Format.

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 (Compressed Sparse Row) 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 are of course other ways to store a matrix, each with their own benefits and drawbacks.

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.

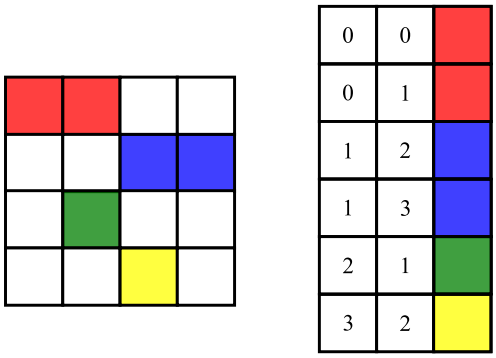


Figure 3.2: cooformat

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

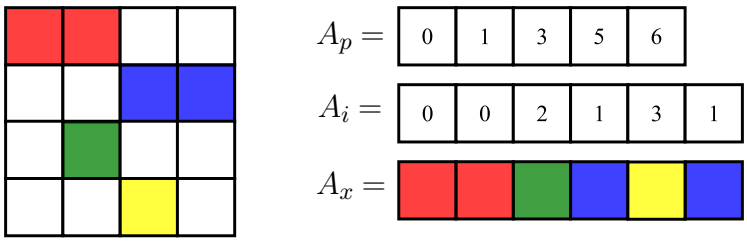


Figure 3.3: cscformat

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 [**?**].

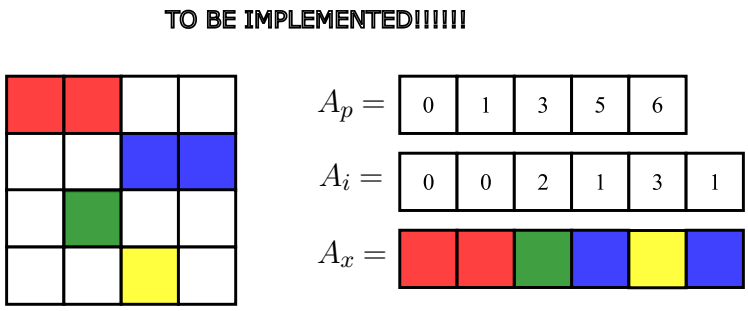


Figure 3.4: ellpackformat

3.3 Sequential SpMV

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

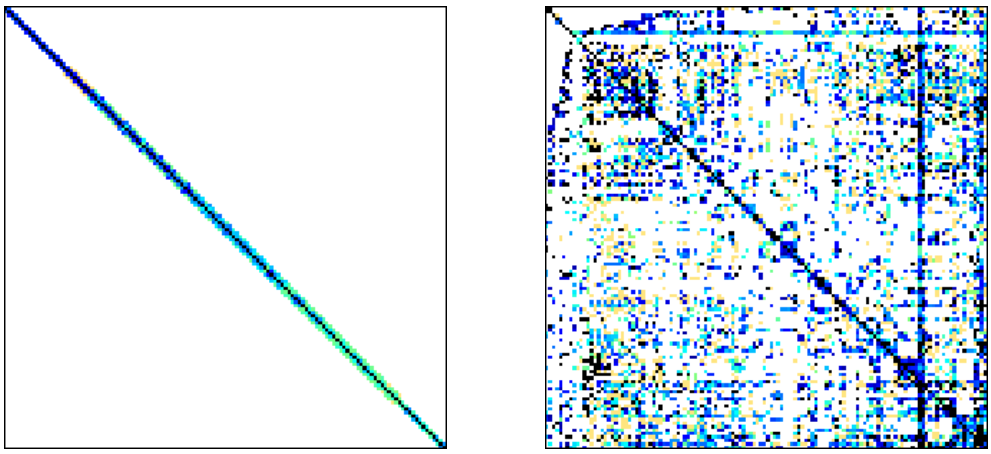
Algorithm 1: Sequential CSR-based SpMV

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, i.e. those similar to the matrix shown in Figure 3.5 we read 12 bytes, and perform 2 FLOPS for each non-zero in the matrix. Keen-eyed readers might notice that this does not coincide with the amount of FLOPS read per non-zero in algorithm 3.3. Here we read two doubles, and one integer, which would be equivalent to 20 bytes. The reason for the discrepancy is due to the fact that after the first time x is accessed, it is loaded into cache, and heavily reused in subsequent iterations, and can for that reason be disregarded.

For heavily unstructured matrices, it is possible that we acutally read up to 76 bytes per 2 FLOPS. This will occur if there is no cache reuse, and a new cache line (64 bytes) is loaded for each non-zero.



(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 ← 0 to n do
    sum ← 0
    for j ← A_p[i] to A_p[i+1] do
        sum = sum + A_x[j] · x[A_j[j]]
    y[i] ← 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.

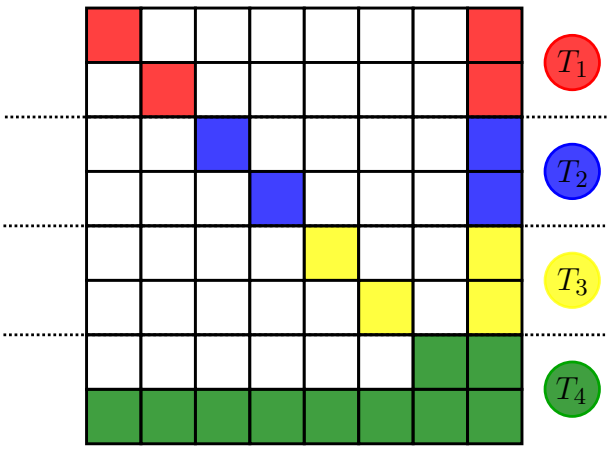


Figure 3.6: staticscheduling

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, the workload is spread across multiple nodes. Physically, each node is its own dual-socketed system, each with its own memory. If data stored on one node is needed by another node, explicit

communication has to occur between the nodes. This is typically done using the Message Passing Interface (MPI).

For distributed memory SpMV, algorithm 2 is still utilized for computing the result vector y, but the matrix is split between nodes, and each node computes a local part of y. The final vector has to be assembled at the end of each iteration of SpMV.

On distributed memory systems, the challenges discussed in 3.4.1 are still relevant, but such systems also pose challenges in regards to the communication volume between nodes. As is evident from Table 2.1, communication between nodes (i.e. sending bytes over the network) is slow. As communication between nodes is slow, reducing the total communication volume per iteration of SpMV is crucial for performance.

3.5.1 Load Balancing

In order to reduce the total communication volume, it is necessary to partition the graph into several part, usually one part per node, or one part per socket. The question falls on how to most effectively partition the graph in order to reduce the communication volume. Firstly, the graph partitoner needs to ensure that the size of each partition balanced according to some criterion. Usually this criterion is described as a percentage of allowed imbalance. Secondly, it needs to minimize the amount of edges that strides across partitions, or in other words, minimize the *edge cut* between partitions. The endpoints of such edges is called a *separator* element. Formally, a separator S in a graph G is a set of elements such that $G \setminus S$ is a disconnected graph.

There are no fixed criterion of the imbalance value that leads to optimal communication reduction. It is possible to assign the entire graph to one part, leaving the other parts of the partitions empty, which effectively eliminates all communication, but this is clearly not optimal as the entire computational workload is assigned to a single process, leaving all other processes with no work. Usually an imabalence value of around 3% is sufficient for most matrices.

3.5.2 Graph Partitioners

There are many options when it comes to picking which graph partitioner to use, each with their own benefits and drawbacks. For the experiments performed in this thesis, the METIS graph partitioner has been used.

The algorithms used in for graph partitioning in METIS are based on multilevel recursive-bisection. These kinds of algorithms optimize for communication reduction by minimizing the edge-cut, while trying to keep the size difference between partitions within the constraint given by the imbalance value.

It is worth mentioning that the problem graph partitioners are trying to solve, mainly that of a perfect bisection, is NP-Hard. Therefore the best these tools can provide us are approximations. They are however very reliable and produce good partitions that are sufficient for our usage

The following algorithm outlines how a graph (or matrix) is partitioned and reordered. Given a matrix g stored in the CSR Format, the number of partitions n_p, and a partition vector p of size $n_p + 1$, that is to store the size of the partition such that the size of the i^{th} ranks size is given by $p[i+1] - p[i]$.

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 ∈ {0,...r} do
    for i ∈ {0,...,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 ∈ {0,...,g.n_r − 1} do
    d ← g.rowPtr[oldId[i]+1] - g.rowPtr[oldId[i]]
    newRowPtr[i + 1] ← newRowPtr[i] + d
    for j ∈ {0,...,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 ∈ {newRowPtr[i],...,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 Sparse Matrix-Vector Multiplication (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. These approaches leverage knowledge of the matrix structure, partitioning methods, and computational dependencies to minimize communication overhead.

4.1 Exchange entire vector

The most straightforward approach is to have each rank send all of its computed values of y to every other rank. This ensures that all processes possess a complete and updated copy of the output vector before the next iteration. This strategy can be implemented using MPI's collective communication operation `MPI_Allgatherv`, which accommodates variable message sizes from each rank. Figure 4.2 illustrates the state of the y vector before and after communication using this strategy.

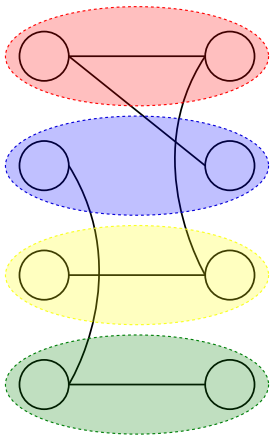


Figure 4.1: examplegraph

Algorithm 4: 1a - Exchange entire vector

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

4.2 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.

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.

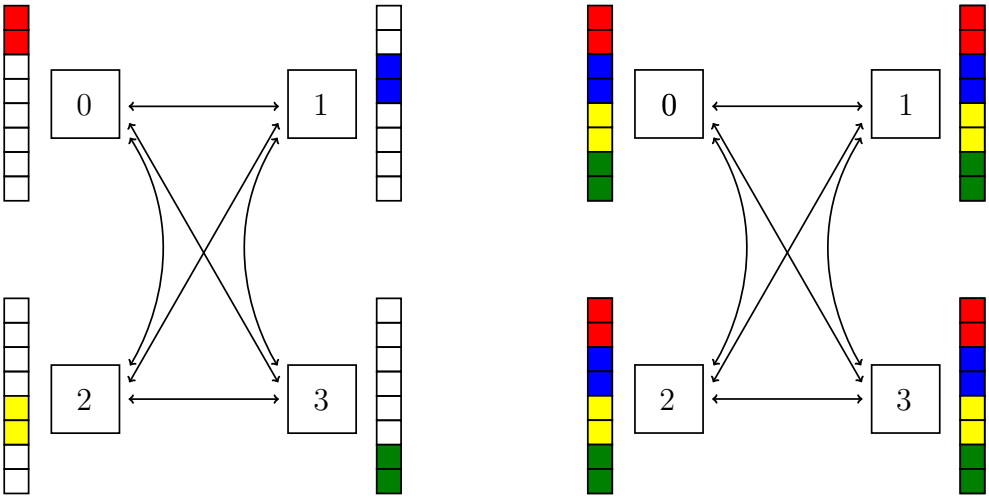


Figure 4.2: 1acomm 2

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.

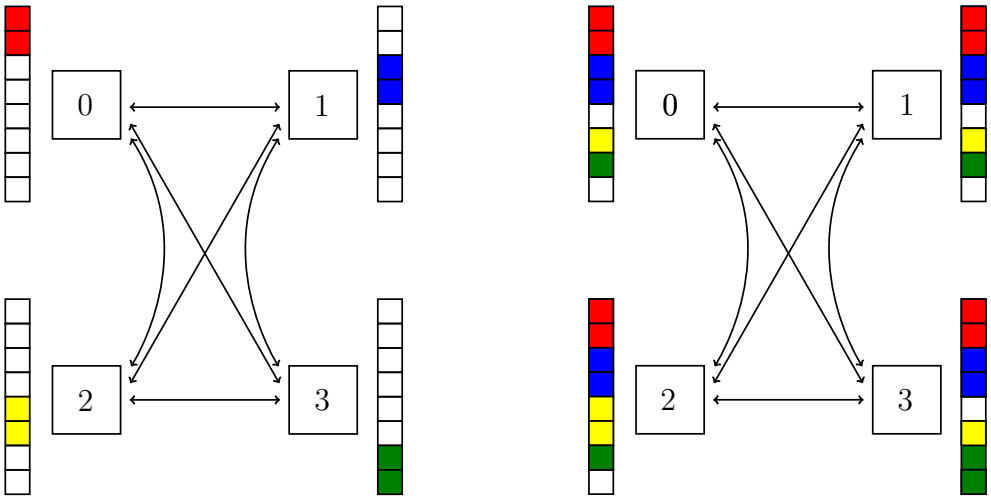


Figure 4.3: 1bcomm

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] · n_r
oldId ← [0] · n_r
id ← 0
p_0 ← 0

for r ∈ {0,...,n_p − 1} do
    for i ∈ {0,...,n_r − 1} do
        if p[i] = r then
            oldId[id] ← i
            newId[i] ← id
            id ← id + 1
    p[r + 1] ← id

newV ← [0] · (n_r + 1)
newE ← [0] · n_c
newA ← [0] · n_c

for i ∈ {0,...,n_r − 1} do
    degree ← A_p[oldId[i] + 1] − A_p[oldId[i]]
    newV[i + 1] ← newV[i] + degree

for i ∈ {0,...,n_r − 1} do
    degree ← A_p[oldId[i] + 1] − A_p[oldId[i]]
    for j ∈ {0,..., degree − 1} do
        newE[newV[i]+j] ← A_j[A_p[oldId[i]] + j]
        newA[newV[i]+j] ← A_x[A_p[oldId[i]] + j]
    for j ∈ {newV[i],...,newV[i + 1] − 1} do
        newE[j] ← newId[newE[j]]
```

4.3 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.

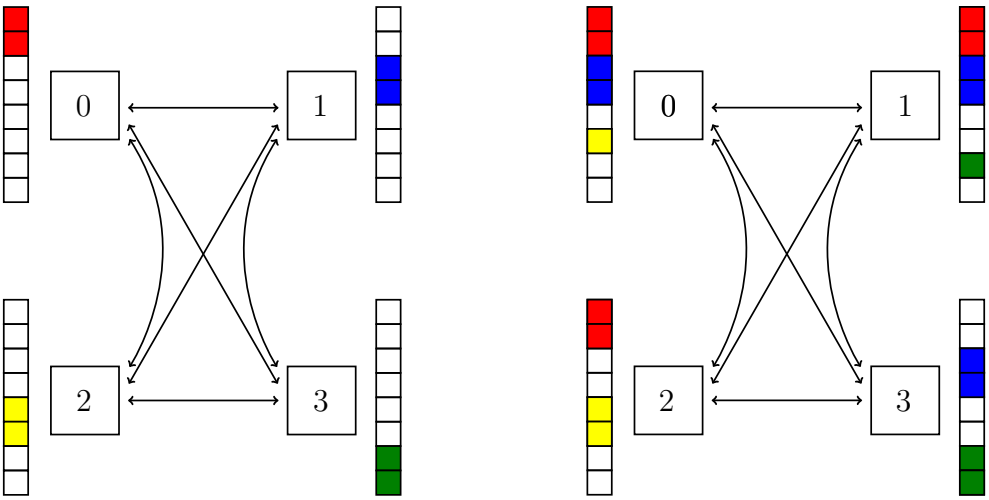


Figure 4.4: 1ccomm

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.

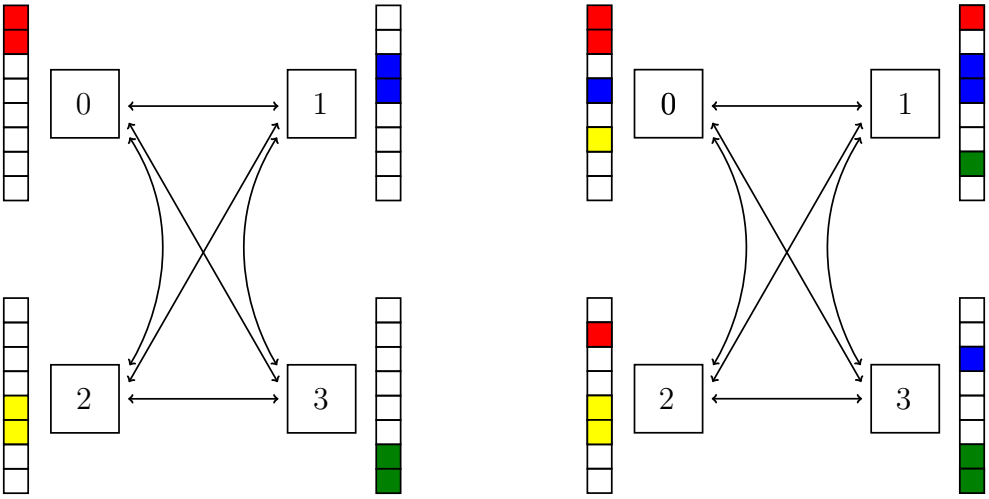


Figure 4.5: 1dcomm

4.5 Exchange Required Elements - 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.

4.5.1 Intelligence processing units

paragraph on IPUs goes here

4.5.2 Memory Scalable

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.

Figure 4.6: 2dcomm

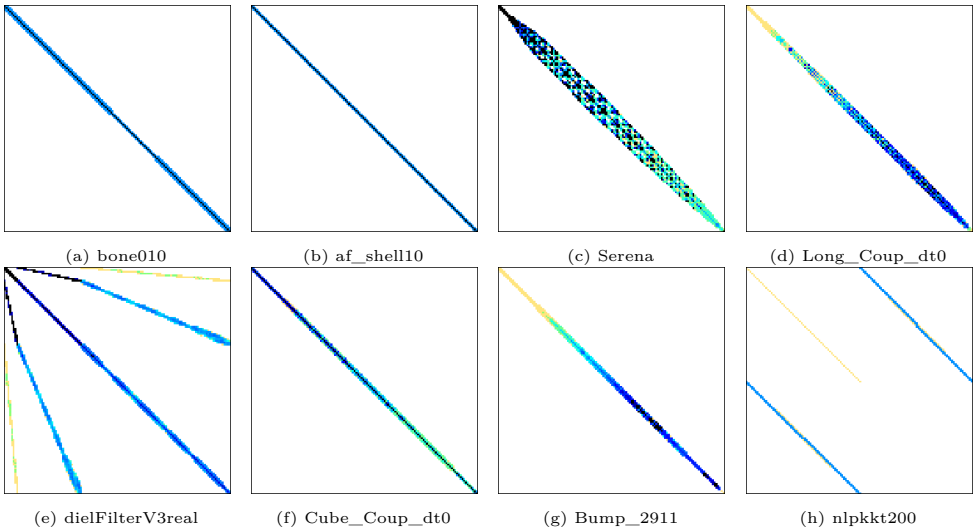
Chapter 5

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.

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



(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: Overview of all eight images.

5.2 Matrices

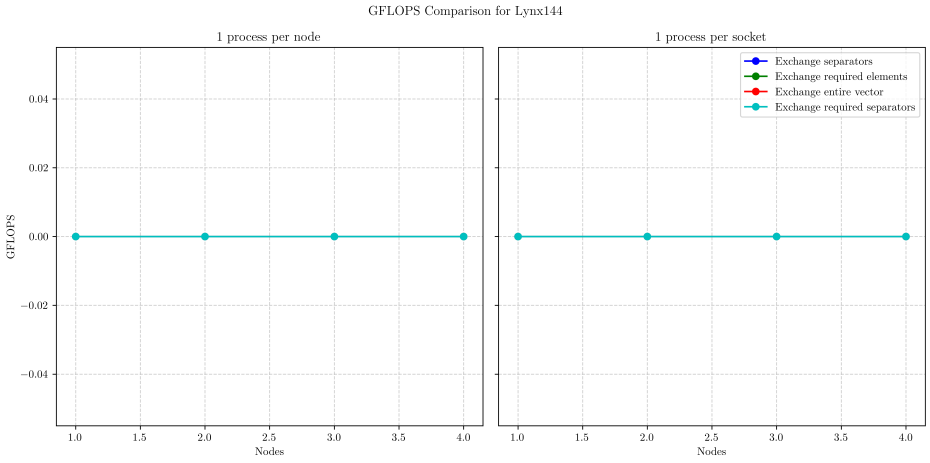


Figure 5.2: SpMV performance with 1 process vs 2 processes per node on Lynx144

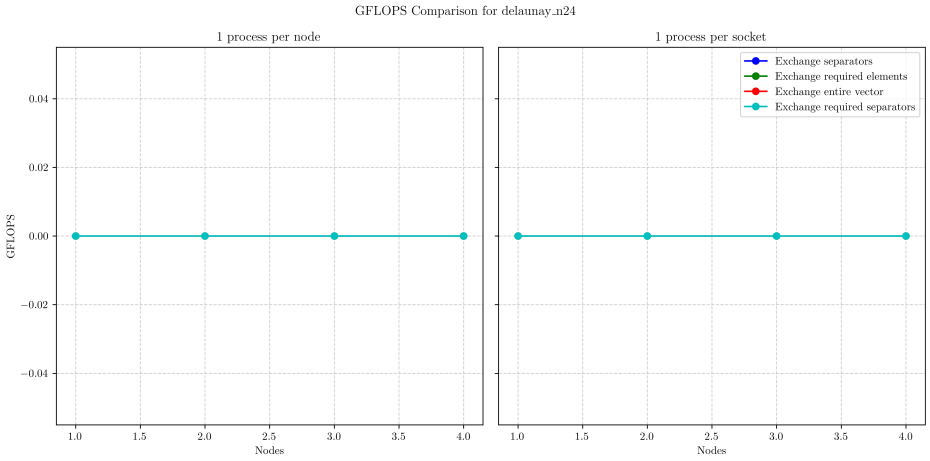


Figure 5.3: SpMV performance with 1 process vs 2 processes per node on delaynay_n24

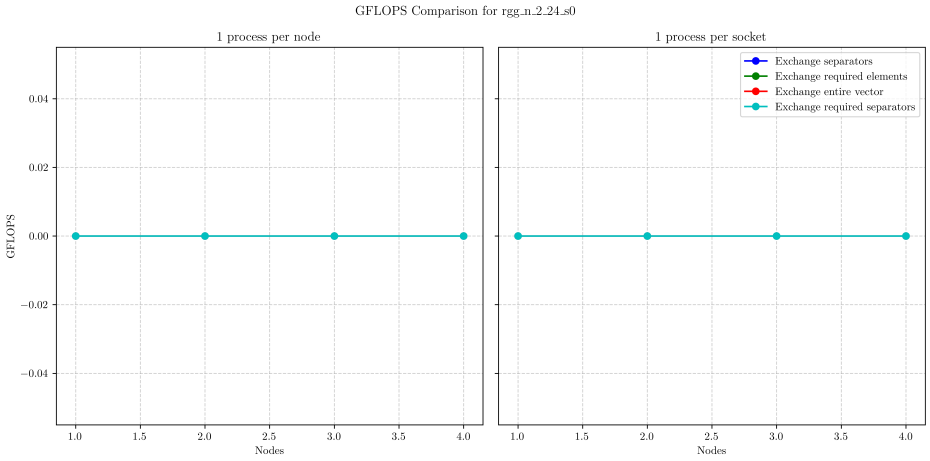


Figure 5.4: SpMV performance with 1 process vs 2 processes per node on rgg_n_2_24_s0

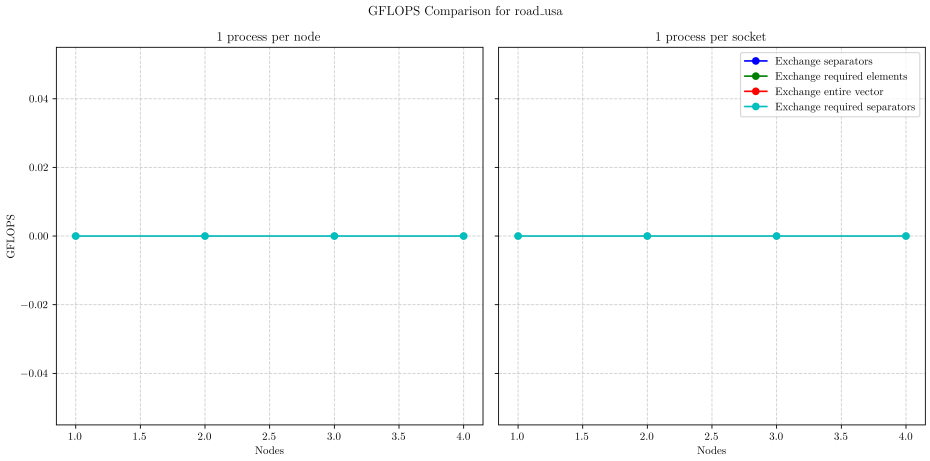


Figure 5.5: SpMV performance with 1 process vs 2 processes per node on road_usa

5.3 GFLOPS AMD EPYC 7302P

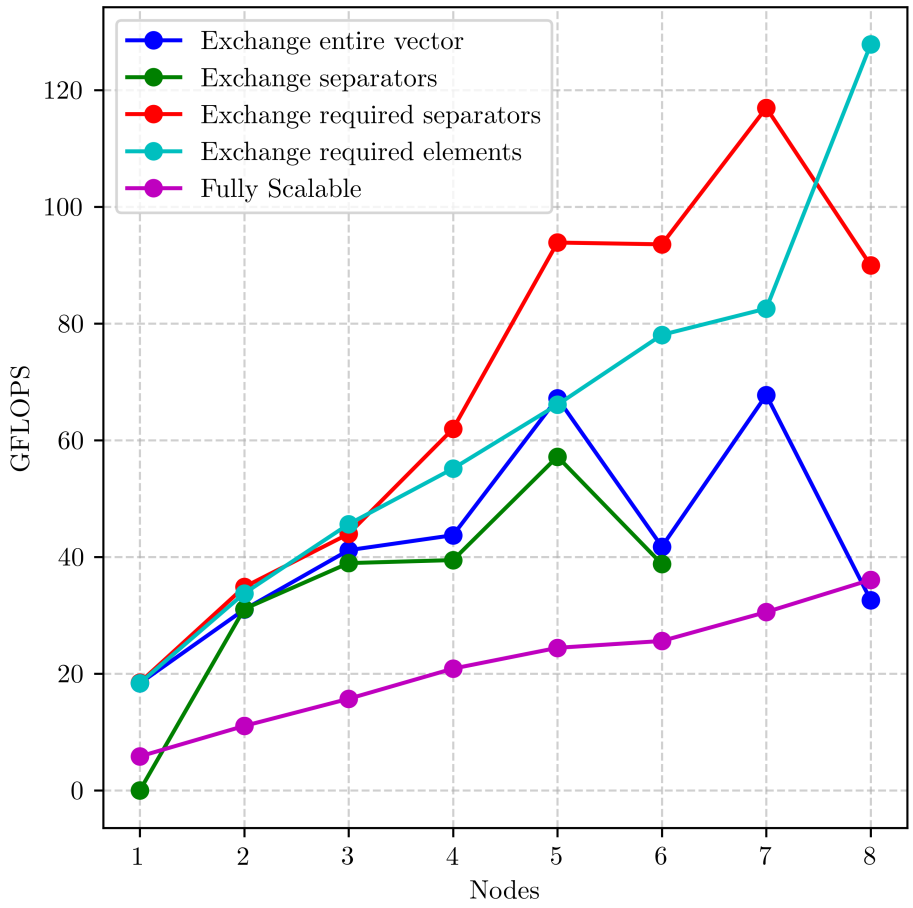


Figure 5.6: Serena

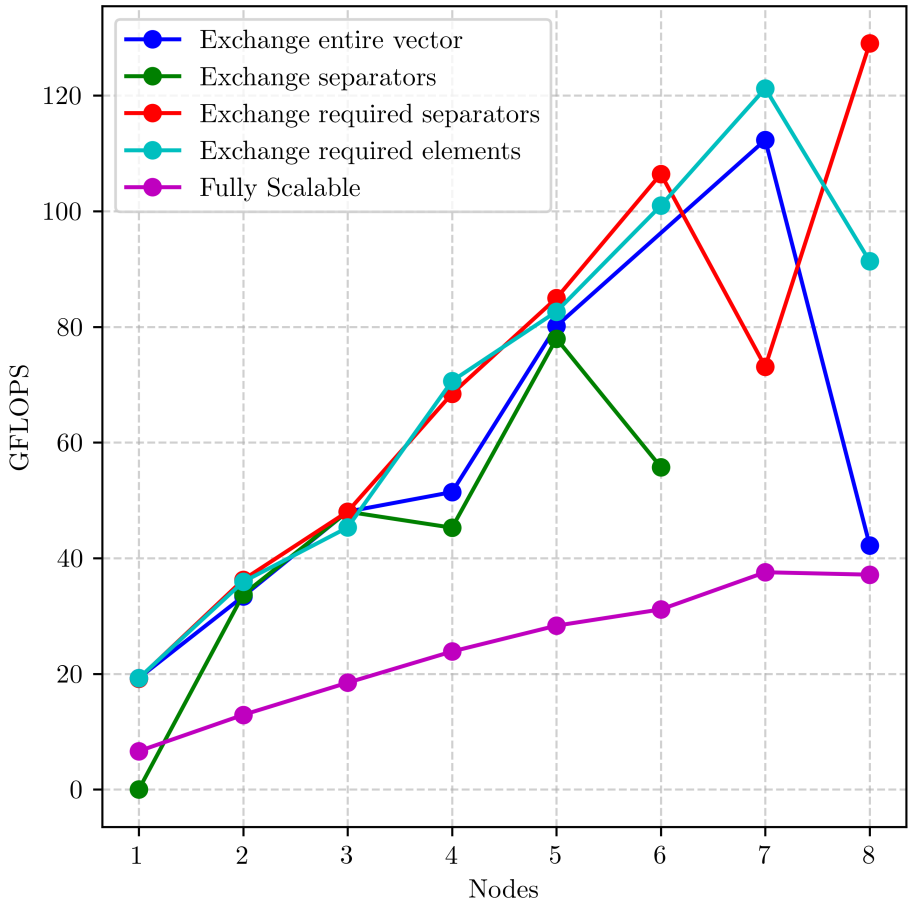


Figure 5.7: bone010

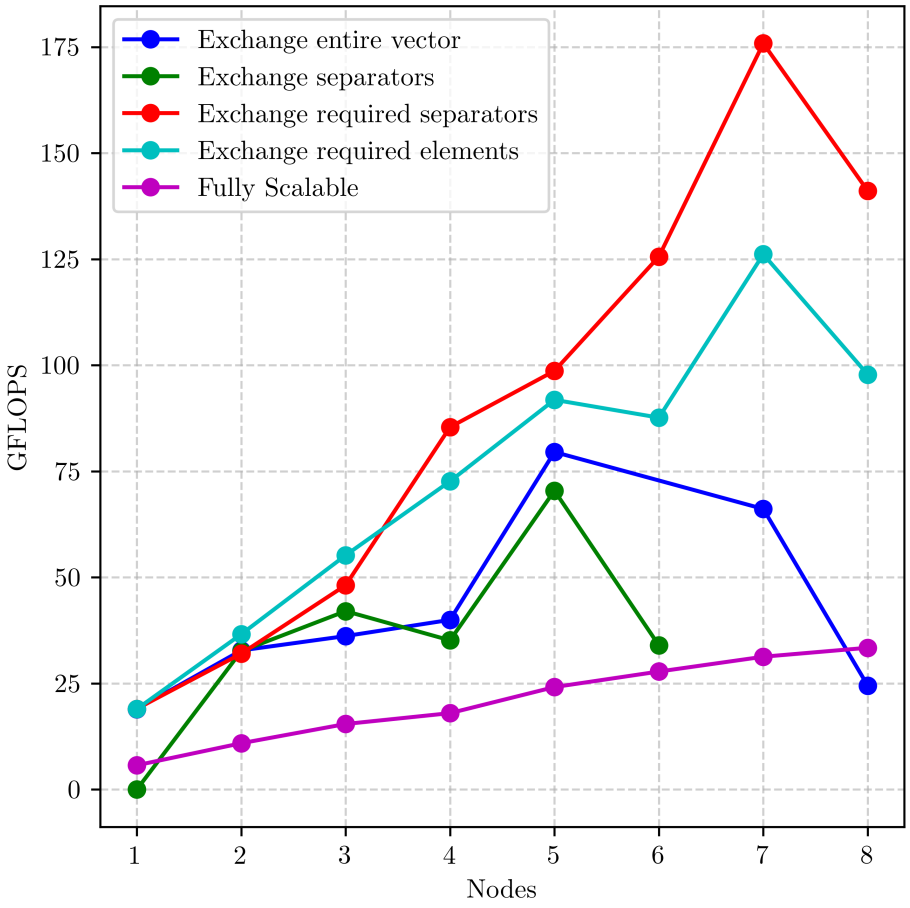


Figure 5.8: af_shell10

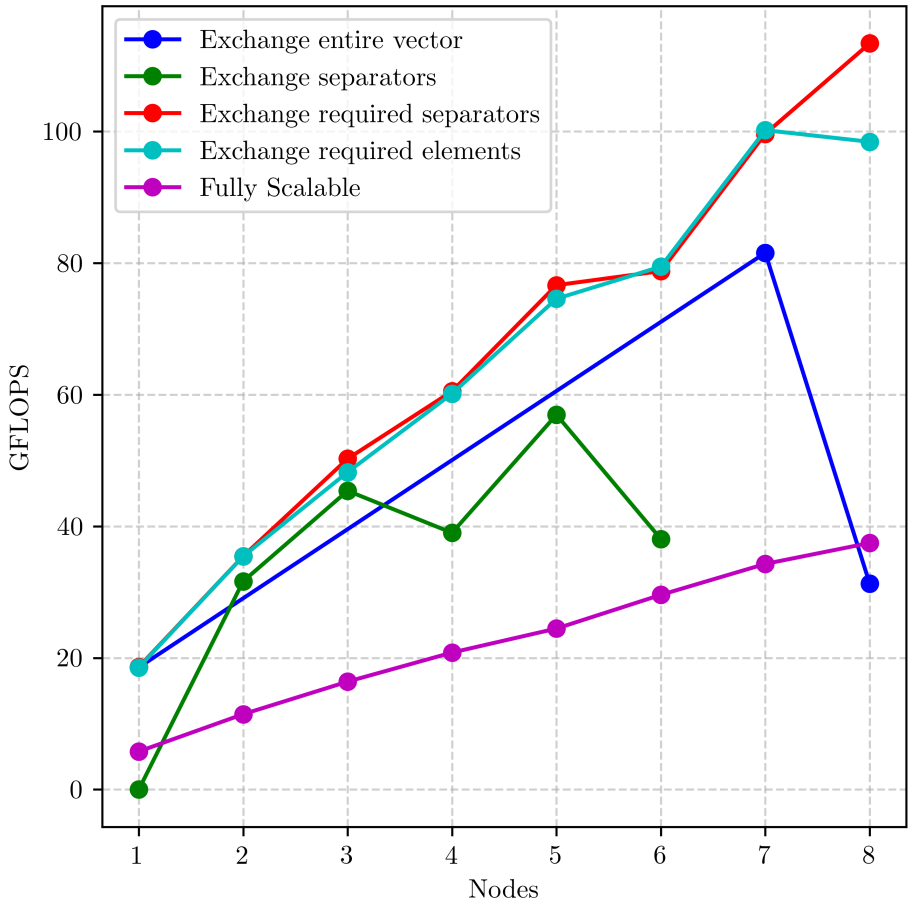


Figure 5.9: Bump_2911

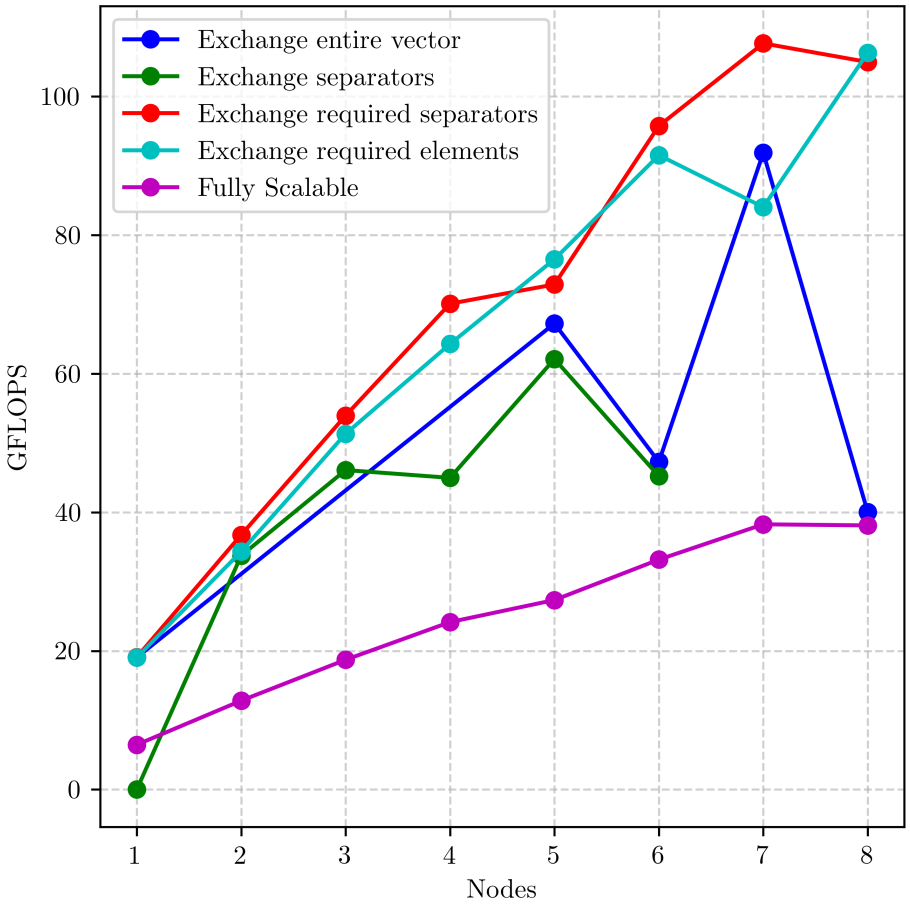


Figure 5.10: Cube_Coup_dt0

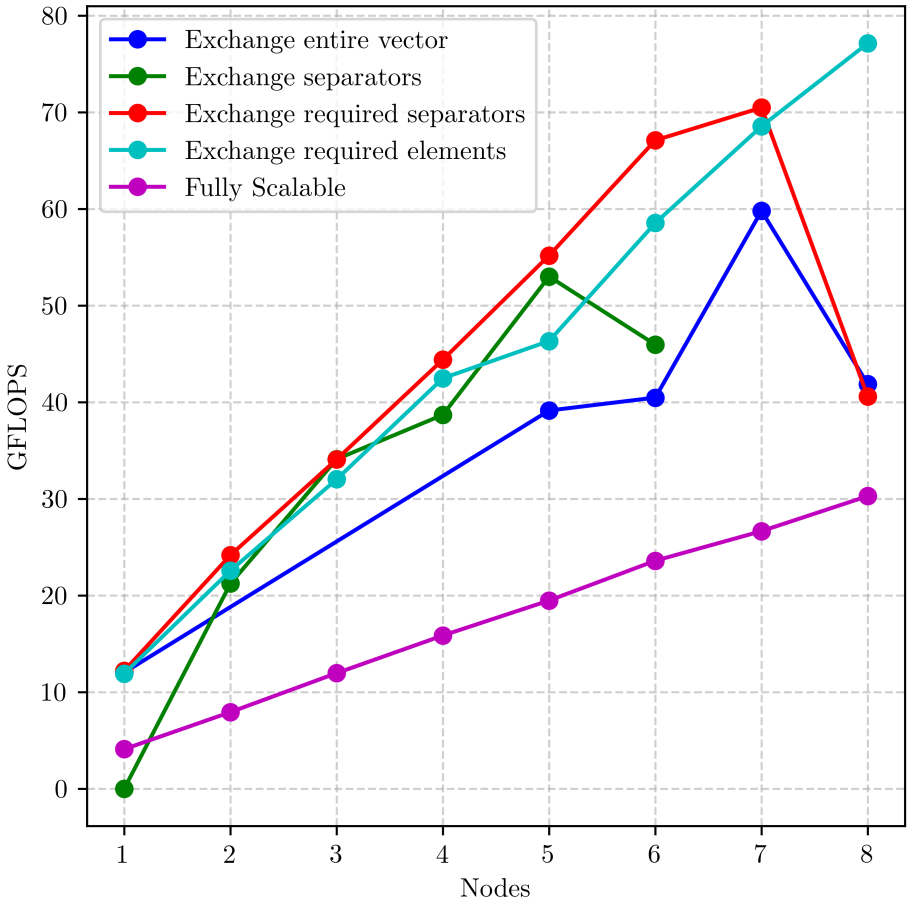


Figure 5.11: dielFilterV3real

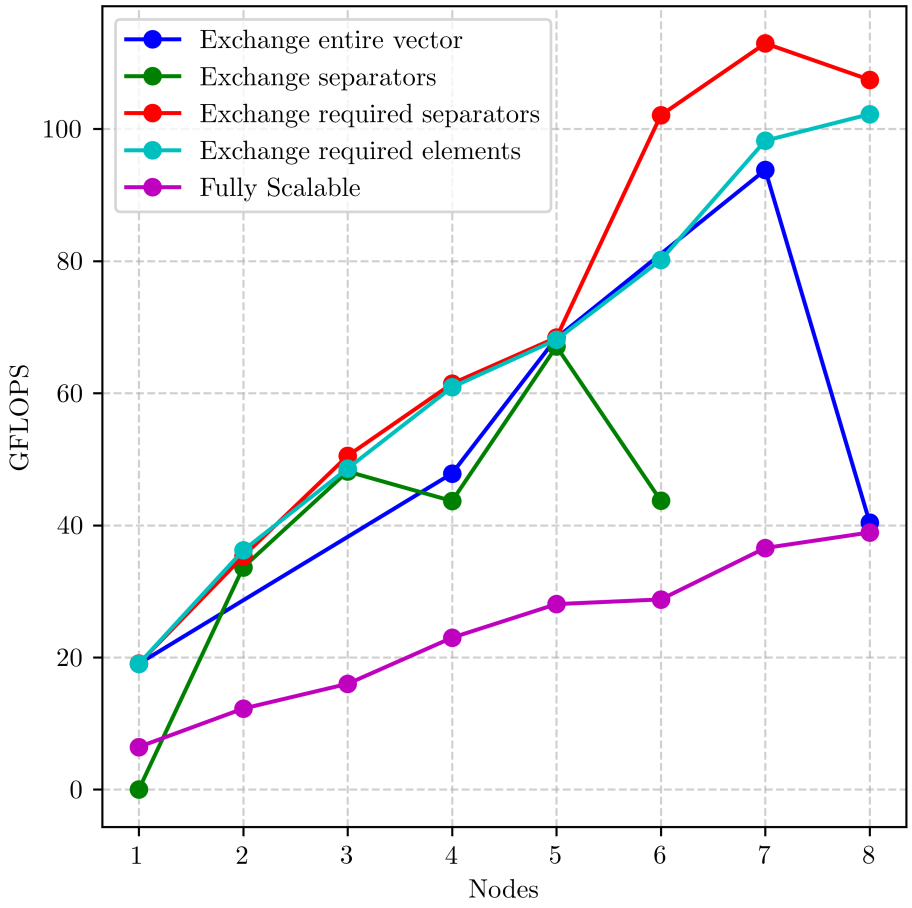


Figure 5.12: Long_Coup_dt0

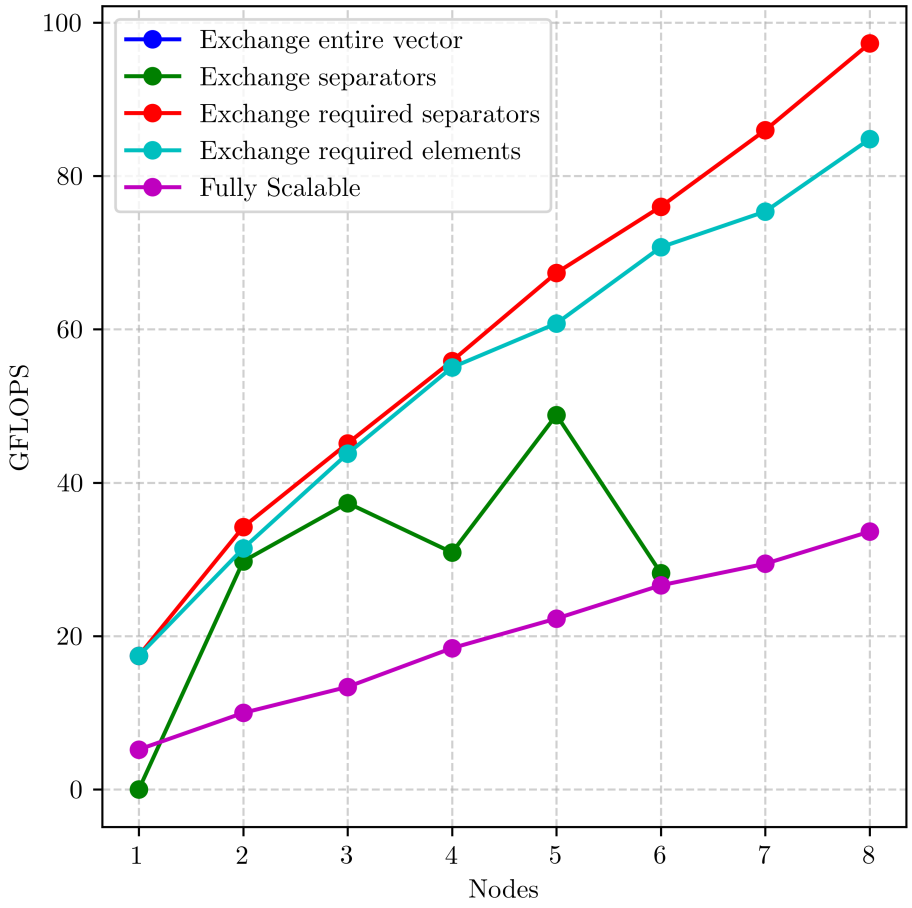


Figure 5.13: nlpkkt200

5.4 GFLOPS AMD EPYC 7413

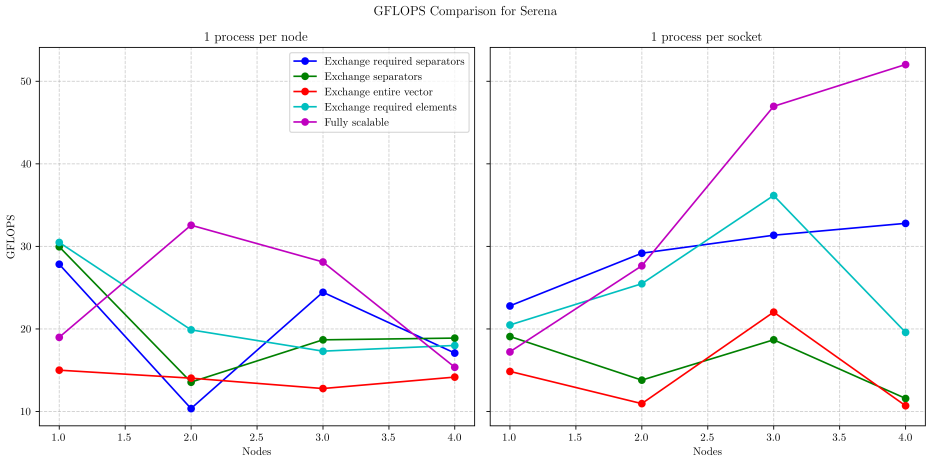


Figure 5.14: Serena

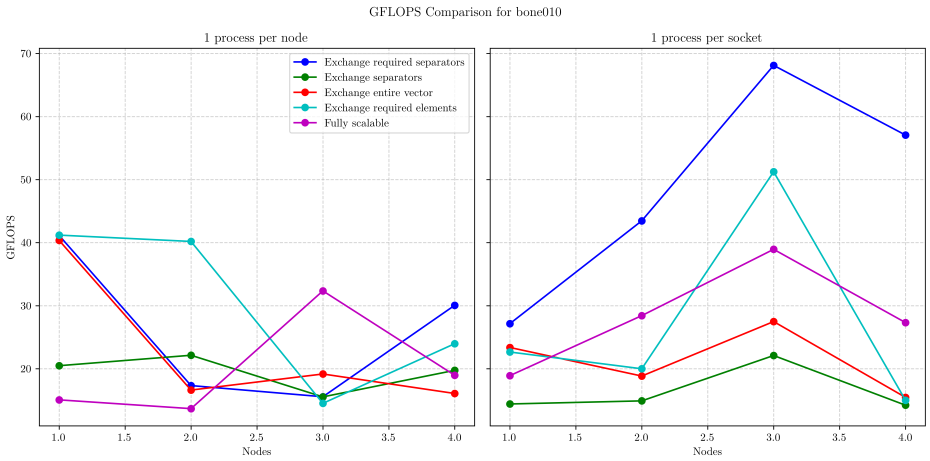


Figure 5.15: bone010

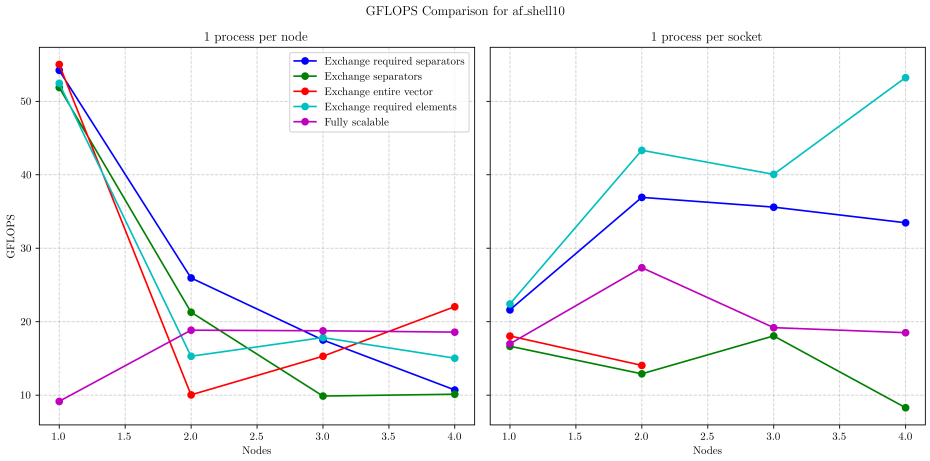


Figure 5.16: af_shell10

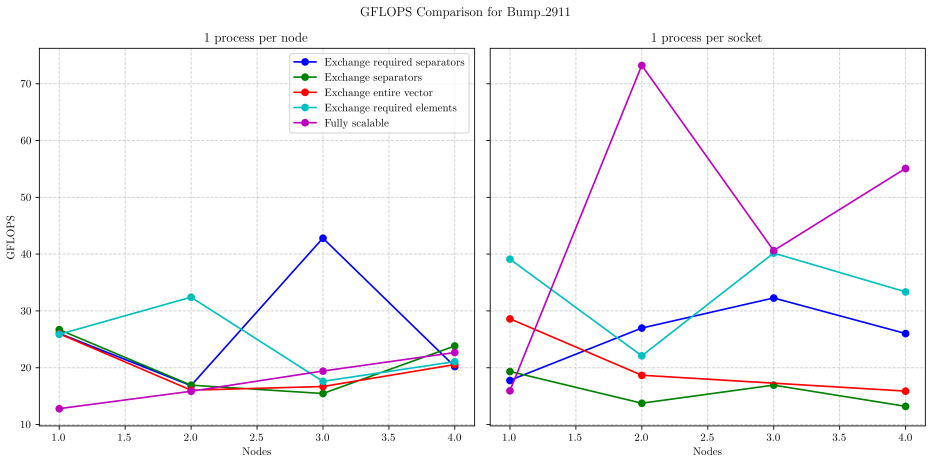


Figure 5.17: Bump_2911

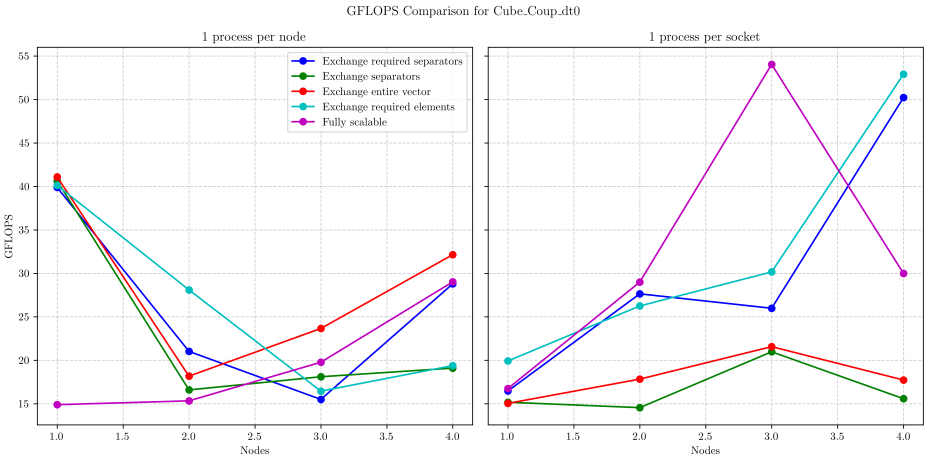


Figure 5.18: Cube_Coup_dt0

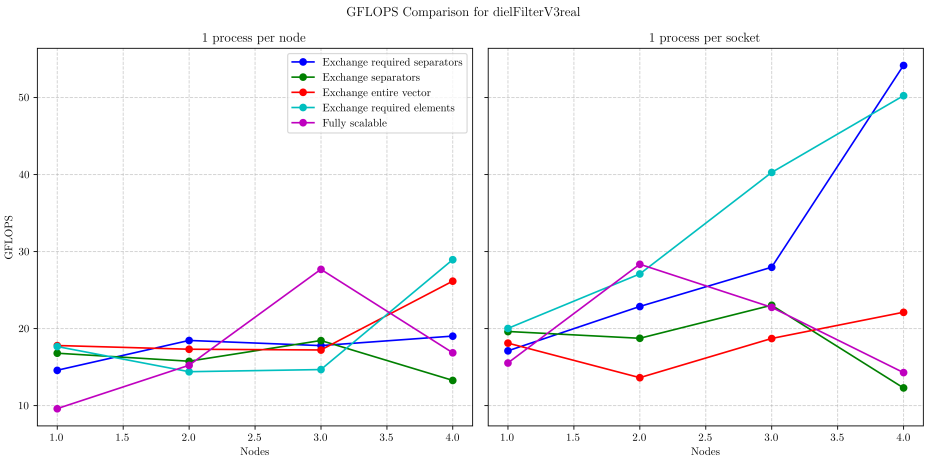


Figure 5.19: dielFilterV3real

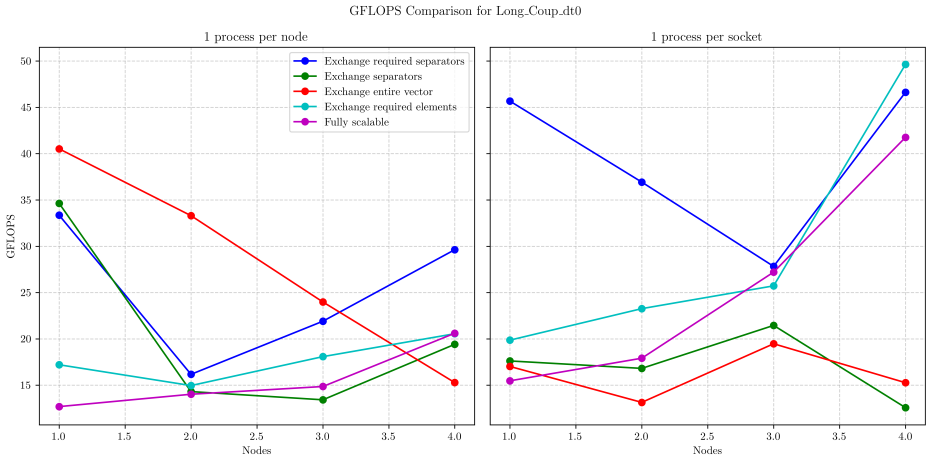


Figure 5.20: Long_Coup_dt0

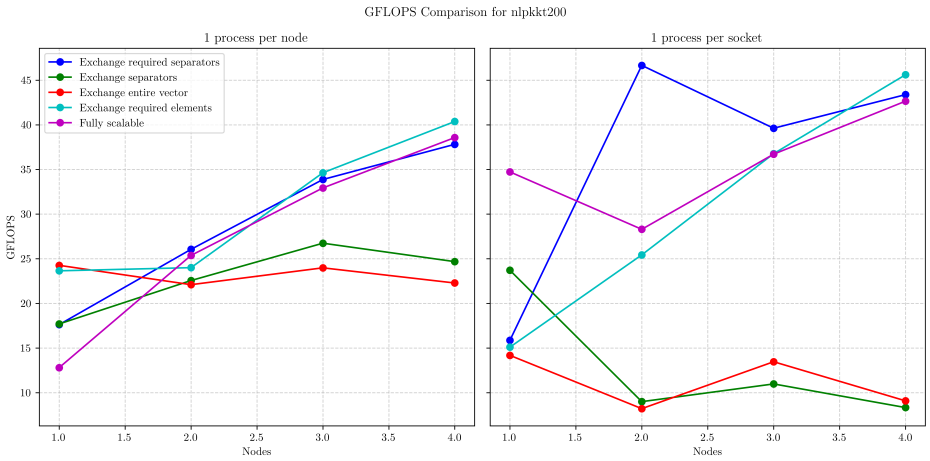


Figure 5.21: nlpkkt200

5.4.1 Communication time

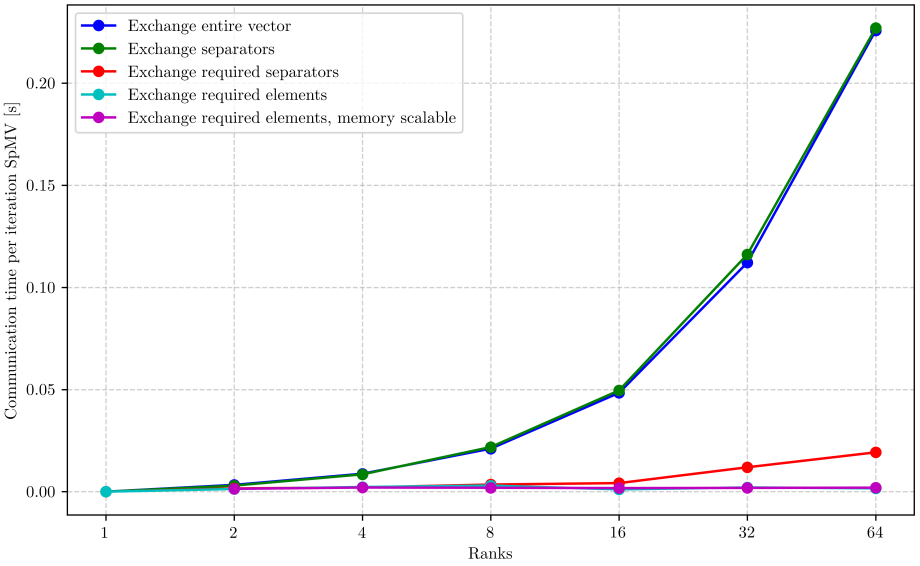


Figure 5.22

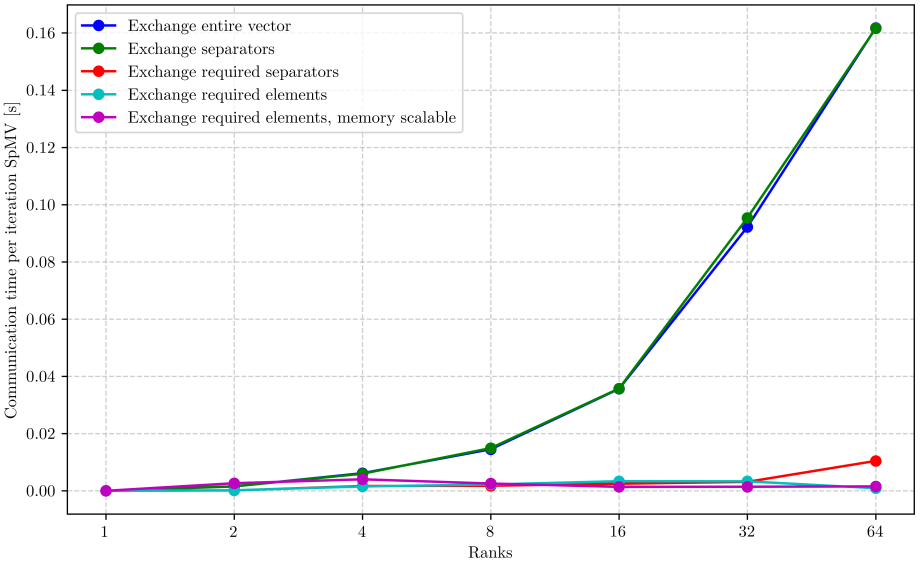


Figure 5.23

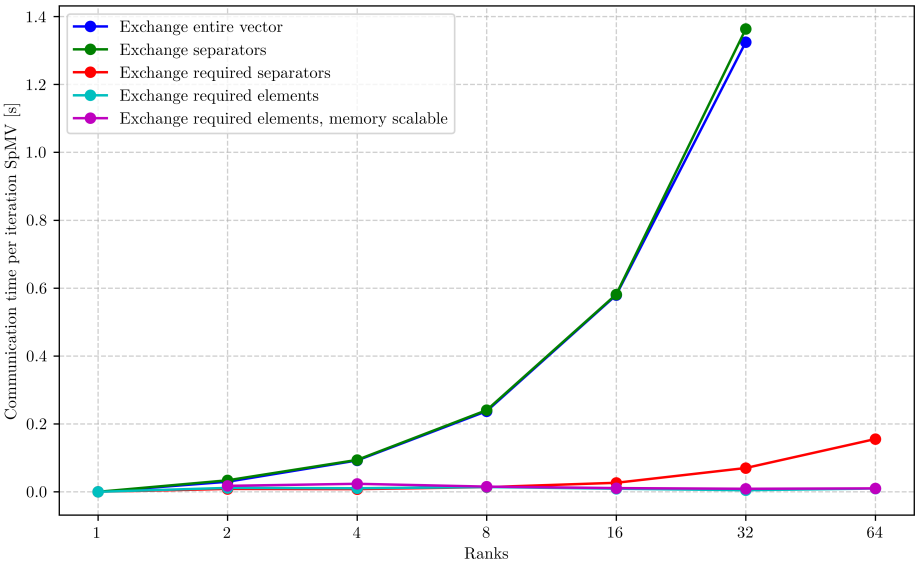


Figure 5.24

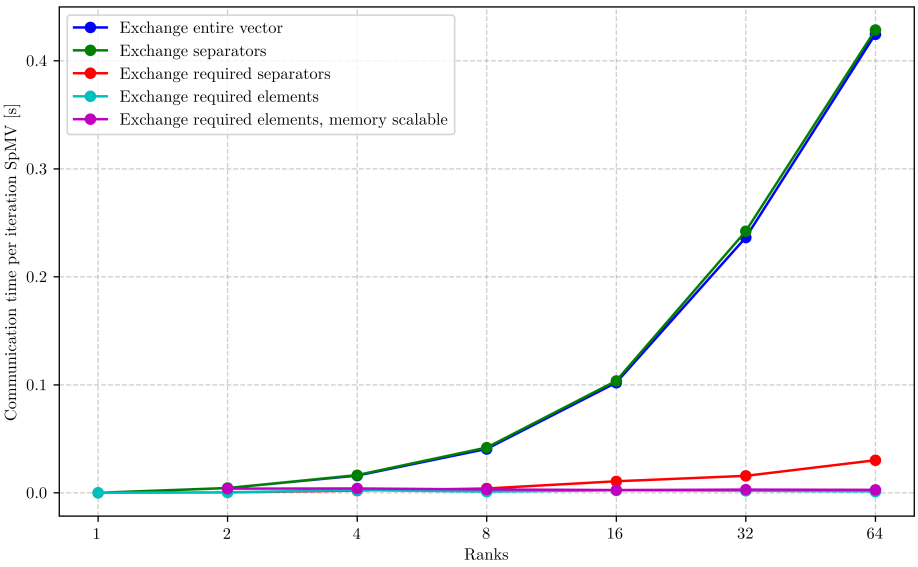


Figure 5.25

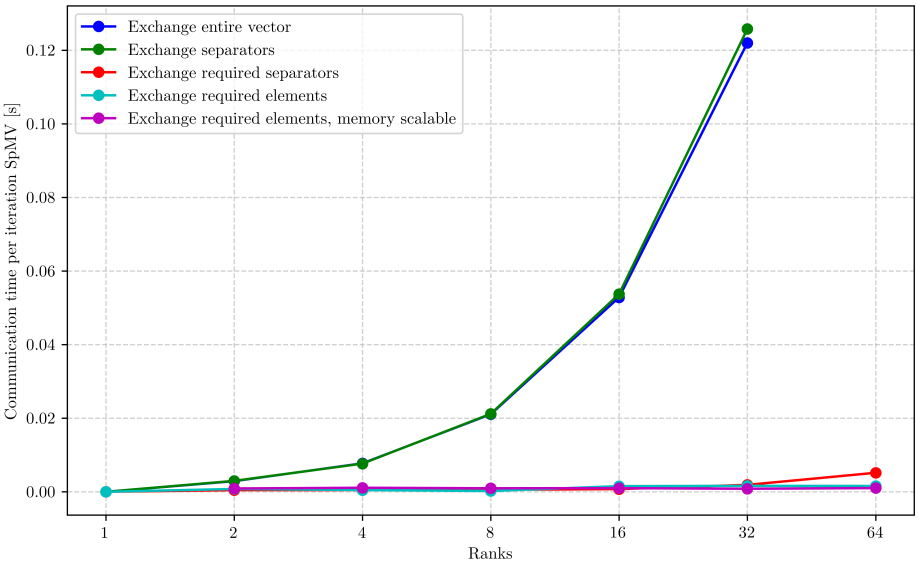


Figure 5.26

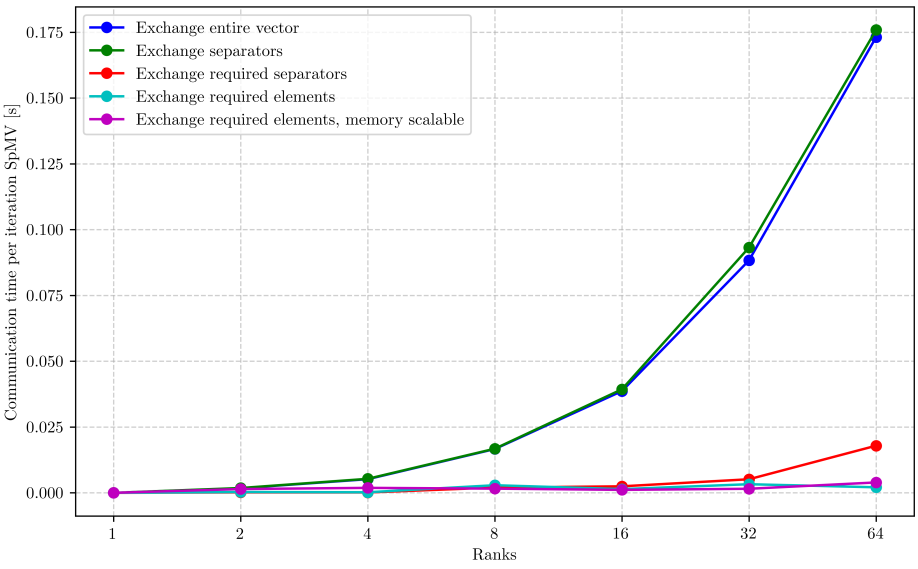


Figure 5.27

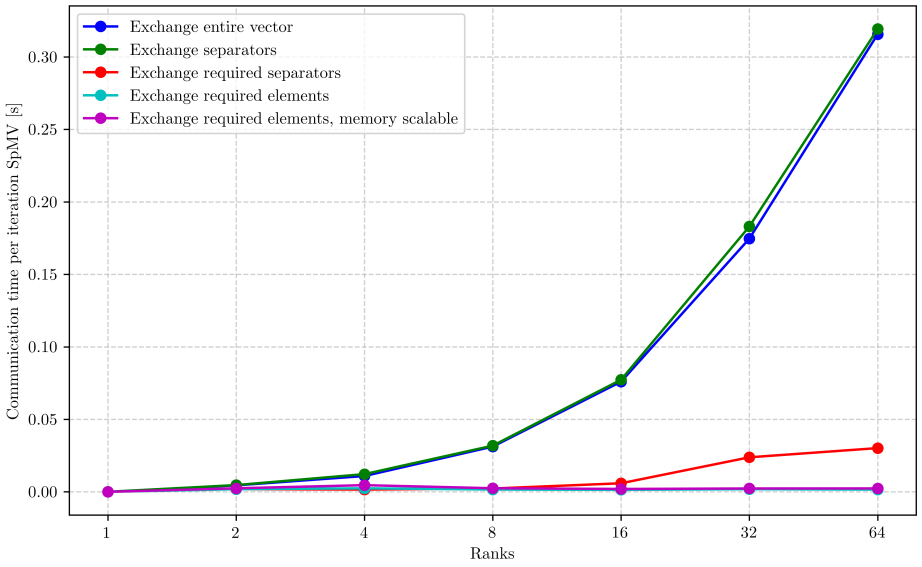


Figure 5.28

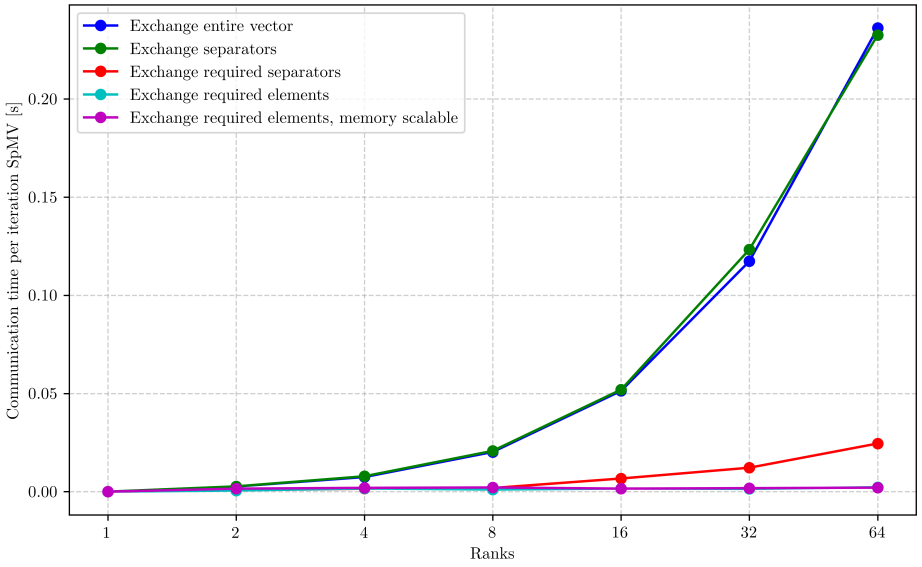


Figure 5.29

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

Previous Work