Università di Pisa

Department of Computer Science

Parallel and Distributed Systems: Paradigms and Models

Project 1:
Distributed Wavefront Computation

Student
Michele Mattiello

ID
683950

A.Y. 2023-2024

Contents

1 Introduction

1.1 Problem evaluation

The task consists, given a Matrix M (NxN), for each diagonal k compute a element $e^k_{m,m+k}$ ($m \in [0, n-k[$) as the result of dot product between two vectors v^k_m and v^k_{m+k} of size k composed by the elements on the same row m and on the same column $m+k$. Specifically:

$$e^k_{i,j} = \sqrt[3]{dotprod(v^k_m, v^k_{m+k})}$$

The values of the element belong to the major diagonal $e^0_{m,m}$ are initialized with values $(m+1)/n$. For instance:

0.10	0.28	0.52	0.80	1.11	1.43	X	X	X	X
	0.20	0.40	0.65	0.94	1.25	1.57	X	X	X
		0.31	0.50	0.76	1.06	1.36	1.68	X	X
			0.40	0.59	0.86	1.16	1.47	1.79	X
				0.50	0.67	0.95	1.25	1.56	1.88
					0.61	0.75	1.04	1.34	1.65
						0.70	0.83	1.12	1.42
							0.80	0.90	1.19
								0.90	0.97
									1.00

Table 1: wavefront computation

The symbol X represent the values to compute, the element highlighted in red is the actual computed value $e^5_{1,6}$ and the yellow cells are the elements of the two vectors v^5_1 and v^5_6.

$$v^5_1 = [\, M[1][1], M[1][2], M[1][3], M[1][4], M[1][5] \,] = [0.20, 0.40, 0.65, 0.94, 1.25]$$

$$v^5_6 = [\, M[6][6], M[5][6], M[4][6], M[3][6], M[2][6] \,] = [0.70, 0.75, 0.95, 1.16, 1.36]$$

$$e^5_{1,6} = \sqrt[3]{0.20 * 0.70 + 0.40 * 0.75 + 0.65 * 0.95 + 0.94 * 1.16 + 1.25 * 1.36} = 1.57$$

To compute $e^5_{1,6}$ element, the values $e^4_{1,5}$ and $e^4_{2,6}$ are required that belong to the previous diagonal. For this reason, a possible approach is to compute all elements belong to the same diagonal, before to pass the next one.

1.2 Report structure

In this report, two different approaches are proposed to solve this problem, each of which is implemented in three versions: sequential, parallelized with shared memory (FastFlow) and distributed memory (MPI). After describing the implementations, they will be compared by analyzing weak and strong scalability to understand the advantages and disadvantages of using one or the other. The metrics for the strong and weak analysis are performed using the best runtime obtained in sequential implementation.

2 Sequential implementations

2.1 Diagonal implementation

For the diagonal implementation, the main diagonal of the matrix is initialized with values $M[m][m] = (m+1)/n$ where m is the element index of the diagonal and n is the row(column) size. For each diagonal k (excluding the main diagonal), it computes values for each element $e^k_{i,j}$ in the diagonal using the cubic root of the dot product between two vectors e^k_i and e^k_j. The figure show the execution flow. The execu-

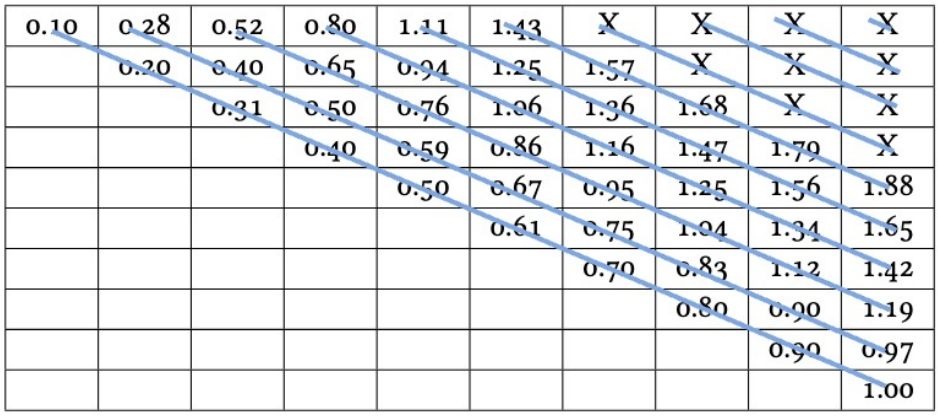


Figure 1: diagonal division

tion starts from the upper left element and it continues on the diagonal until the end. The same computation is applied for the rest of diagonals until the upper right element. The code is in **wavefront_diagonal.cpp**.

2.2 Triangular implementation

The purpose of the problem is to compute the upper triangular matrix, based on this objective, we can divide this triangle into small triangles to improve space locations, take advantage of vector optimization and reduce communication overhead in parallel implementation.

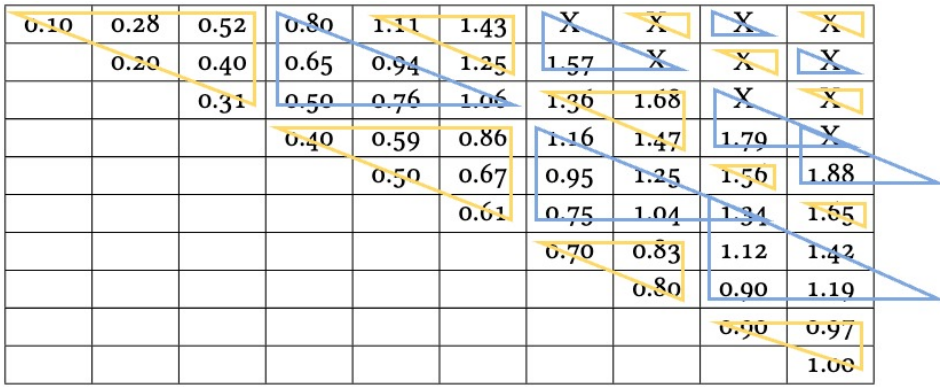


Figure 2: triangles division example

We can identify two different type of triangles in this example, the first type has the angle of ninety degrees in the upper right and the second one has the same angle in the lower left. Each triangle has the same

number of elements in all sides. From this description, we can represent each triangle by coordinates of one angle, size of the side and if it is reversed or not.

Type	Field Name	Description
int	start_index	The index of a triangle angle.
int	size_side	The number of elements on the side of the triangle.
bool	is_diag	A boolean flag indicating if the start_index lies on the hypotenuse of the triangle.

Table 2: Description of the fields in the `triangle` structure.

The **start_index** choosen for the yellow triangles is the index of the upper left element while for the blue ones it is the lower left. This choise is motivated by each triangle has to iterate one diagonal at a time, starting from the leftmost one, to mantain the constraint of the problem. If **is_diag** is true, the triangle structure represent yellow triangles, otherwise the blue ones. In some cases the triangles are overlapped between them, it is not a problem for sequential implementation, but for mpi and fastflow we need to pay a lot of attention to resolve this case.
After the triangle definition, we need to define algorithm to divide the upper triangular matrix into triangles (**divide_upper_matrix_into_triangles** in the code). To do this, we need to define the number of triangles (**ntriangles**) that we want on the first diagonal, this is a variable that we will vary to analyze the execution time. The number of remaining triangles are dependent by this choice because if we have n triangles on the first diagonal, we have to define n-1 reversed triangles to go on. After this, we can divide again the third triangles row for the number that we defined and go on until the end of the elements. Execution steps:

1. Divide the main diangonal for the number of triangles that we want and create the first triangles row. If our number is less than elements number of the diagonal, we use the number of elements.

2. Build the second row of triangles (reversed ones) that fit on the first row in a way that we have another diagonal where we can create triangles with the same logic of the previous point.

3. Go to point 1. and repeat until the end of elements.

The third step is to define a function in which we can iterate on triangles based on which triangle we have (**iterate_on_matrix_by_triangle** and **iterate_on_matrix_by_reversed_triangle** functions). The execution flow is the same but we have to handle the fact that one triangle is the opposite of the other and some reversed triangles near the left margin of the matrix (yellow) are cut. The code is in **wavefront_triangles.cpp** file.

2.3 Triangular collapsed implementation

This implementation is based on the triangular implementation, the only difference is in **iterate_on_matrix_by_triangle** and **iterate_on_matrix_by_reversed_triangle** functions in which the two for used to iterate on the matrix given a triangle are collapsed in only one trying to take advantage of vectorization. The execution times of this implementation are better than the execution times of the base triangle implementation by a few hundred milliseconds, at the expense of code readability. For this reason, this implementation is not carried forward in subsequent analyses but is only mentioned.

2.4 Transposition of the upper triangular matrix

A problem that we can observe is the poor spatial locality when we access a column e_j^k to compute the dotProduct. To solve this question we can use a transposed column for improving the access at the memory, obtained by replicate the upper triangular matrix to the lower one. For instance, starting from the previous

computation shown in table 1, we can create a symmetric matrix by copying the upper triangular portion to the lower triangular portion in a specular way and we use the transposed column so the result is unchanged. We can easily obtain the symmetric matrix without additional work; when we compute the value $e^k_{i,j}$ we assign the same value at element $e^k_{j,i}$ so we can use row instead column.

0.10	0.28	0.52	0.80	1.11	1.43	X	X	X	X
0.28	0.20	0.40	0.65	0.94	1.25	1.57	X	X	X
0.52	0.40	0.31	0.50	0.76	1.06	1.36	1.68	X	X
0.80	0.65	0.50	0.40	0.59	0.86	1.16	1.47	1.79	X
1.11	0.94	0.76	0.59	0.50	0.67	0.95	1.25	1.56	1.88
1.43	1.25	1.06	0.86	0.67	0.61	0.75	1.04	1.34	1.65
X	1.57	1.36	1.16	0.95	0.75	0.70	0.83	1.12	1.42
X	X	1.68	1.47	1.25	1.04	0.83	0.80	0.90	1.19
X	X	X	1.79	1.56	1.34	1.12	0.90	0.90	0.97
X	X	X	X	1.88	1.65	1.42	1.19	0.97	1.00

Table 3: wavefront computation with good spacial locality

$$v^5_1 = [\, M[1][1], M[1][2], M[1][3], M[1][4], M[1][5] \,] = [0.20, 0.40, 0.65, 0.94, 1.25]$$
$$v^5_6 = [\, M[6][6], M[6][5], M[6][4], M[6][3], M[6][2] \,] = [0.70, 0.75, 0.95, 1.16, 1.36]$$
$$e^5_{1,6} = M[1][6] = M[6][1] = \sqrt[3]{0.20 * 0.70 + 0.40 * 0.75 + 0.65 * 0.95 + 0.94 * 1.16 + 1.25 * 1.36} = 1.57$$

This improvement is applied to all sequential implementations.

2.5 Test structure

To test the correctness of the implementations and to evaluate the execution results, we define 3 shell script, one for each code. Each script creates a csv file with this relative path:

test_sequential/{code_name}/timestamp}/wavefront_results.csv.

In the csv file, the fields name that we will save are:

Field name	Description
Name	Name of the tested executable.
Matrix size	Elements number on the side of the matrix.
Triangles number	Triangles number (only for triangles implementations).
Time	Execution time of the code.
Value	The top right element in the matrix. It is useful to check the correctness of the results.

Table 4: csv file structure for testing and analysis

The same test structure is used for the following implementations showed in the chapters below.

2.6 Execution time comparisons

In the tables shown below, we can observe the execution significant difference between diagonal and triangular implementation. In case of execution with triangles, the execution time decreases with increasing number of triangles until a certain point, after which the performance degrades. Tests show that the implementation of

the triangle is better than the diagonal, in some cases the execution time is halved. The **wavefront_diagonal** run time and the best **wavefront_triangle** run time are in bold.

Implementation	Matrix size	Triangles number	Execution time (ms)
diagonal	2048	-	**1030**
triangle	2048	1	1232
triangle	2048	2	995
triangle	2048	4	935
triangle	2048	8	**935**
triangle	2048	16	996
triangle	2048	32	1072

Table 5: Execution time on a matrix 2048x2048

Implementation	Matrix size	Triangles number	Execution time (ms)
diagonal	5793	-	**39765**
triangle	5793	1	40835
triangle	5793	2	34035
triangle	5793	4	25045
triangle	5793	8	19183
triangle	5793	16	**19098**
triangle	5793	32	19784

Table 6: Execution time on a matrix 5793x5793

Implementation	Matrix size	Triangles number	Execution time (ms)
diagonal	8192	-	**108768**
triangle	8192	1	108114
triangle	8192	2	108033
triangle	8192	4	105877
triangle	8192	8	89688
triangle	8192	16	68689
triangle	8192	32	**54449**

Table 7: Execution time on a matrix 8192x8192

2.7 Compile and execute

All codes showed in this chapter are located in **wavefront_sequential** directory and their names are **wavefront_diagonal.cpp**, **wavefront_triangles.cpp** and **wavefront_triangles_collapsed.cpp**.

In the base directory of the project, run:

- To compile: **make sequential or make all**
- To execute:
  - **srun wavefront_diagonal ⟨size of rows/columns ⟩**
  - **srun wavefront_triangles ⟨size of rows/columns⟩ ⟨triangles number on the diagonal⟩**

3 Parallel implementation of Fastflow

3.1 Introduction

In this section, two different fastflow approaches are shown. All codes are in **wavefront_fastflow** directory.

3.2 Diagonal implementation

For parallelization of diagonal implementation with Fastflow, we define a farm structure in which we have an Emitter and workers node with feedback channel.

For each diagonal in the matrix, the Emitter node:

- Divide the diagonal for the number of workers plus the emitter
- Send the obtained tasks to the workers
- Compute his task
- Wait that all workers have finished their job
- Go to the next diagonal

Worker node:

- Wait a task from the emitter
- Compute his job
- Send a feedback to the emitter to say that his job is done

3.3 Triangular implementation

For parallelization of the triangular implementation with Fastflow, the approach is based on the same structure of the diagonal implementation.

For each vector of triangles given by divide_upper_matrix_into_triangles function, the Emitter node:

- Send a triangle to each worker
- Compute his task
- Wait that all workers have finished their job
- Go to the next vector of triangles

Worker node:

- Wait a task from the emitter
- Compute his job
- Send a feedback to the emitter to say that his job is done

The number of triangles that we pass at divide_upper_matrix_into_triangles function is the same of the number of threads that we have. The only thing that we observe is that the number of reversed triangles is one less than the number of workers. For this reason, the emitter will not always have a task. Another important thing is that, two workers rarely could compute the same element because the triangles can have one element in common and, seen that there is not an execution order between threads, it can be wrong. Although this event is very rare because they are at most n-2 where n is the triangles number, we prevent implementing a function in the emitter that finds this shared elements and computes again those values.

3.4 Scalability analysis

Observing the strong scalability analysis of the two implementations on a 2896 matrix, **figure 3**, the diagonal implementation performs better than the triangular one, showing better speedup, efficiency, and costs, with these performance metrics holding steady even as the number of workers increases. In the case of the triangular implementation, performance deteriorates beyond 8 processes. Analyzing the remaining two strong scalability tests, **figures 4 and 5**, with larger matrices (5793 and 8192), the two implementations exhibit the same trend as in the previous analysis, with one notable difference: the triangular implementation with 8 threads surpasses the best performance achieved with the diagonal one (with 32 workers). In the weak scalability analysis, **figure 6**, which examines the behavior of an implementation under steady load per worker, not just an increase in the number of processes, the triangular implementation demonstrates better scalability up to 16 processes, after which execution times increase significantly, while the diagonal implementation shows a slower and more controlled improvement but also declines after 16. Given these observations, this analysis shows that, in the case of large matrices, the triangular implementation parallelized with FastFlow achieves better performance with fewer processes compared to the diagonal implementation, even when using a greater number of processes.

3.5 Compile and execute

All codes showed in this chapter are located in **wavefront_fastflow** directory and their names are **wavefront_diagonal_ff.cpp** and **wavefront_triangles_ff.cpp**. In the base directory of the project, run:

- To compile: **make fastflow** or **make all**
- To execute:
  - **srun wavefront_diagonal_ff ⟨size of rows/columns⟩ ⟨number of workers⟩**
  - **srun wavefront_triangles_ff ⟨size of rows/columns⟩ ⟨number of workers⟩**

4 Parallel implementation of MPI

4.1 Introduction

In this section three different implementation are proposed for the parallelization using mpi:

- **Diagonal impl with Scatterv and Gatherv**: to have only one copy of the matrix at the expense of speed.
- **Diagonal implementation with AllGatherv**: all processes have a copy of the matrix to reduce comunication overhead at the expense of the memory.
- **Triangular implementation with AllGatherv**: the same of the previous point but with triangle implementation.

4.2 Diagonal implementation with Scatterv and Gatherv

This implementation starts defining the processes number and the id for each process. The computation is different between the worker with rank 0 and the others because only the root process has the matrix. All processes iterate for the number of diagonals in the matrix but only the rank 0 can access to it.

For all diagonals in the matrix:

- All processes compute the size of the piece of the diagonal that each one have to compute and number of the elements needed to compute it.
- The process 0 finds the dependencies for each piece of diagonal and it builds a dependency matrix.
- The process 0 sends a piece of the dependency matrix to each process based on the part of diagonal that it have to compute.
- All processes compute the diagonal part using the piece of dependency matrix received.
- All processes send their piece of the diagonal to the process 0.
- The process 0 updates its matrix.

4.3 Diagonal implementation with AllGatherv

This implementation starts defining the processes number and the id for each process. The difference between the previous section is that all processes have a copy of the matrix to update, so we can avoid the overhead given by the dependency matrix to improve the execution time at the expense of memory.

For all diagonals in the matrix:

- All processes compute the size of the piece of the diagonal that each one have to compute.
- Each process computes its piece of the diagonal using the matrix.
- Each process sends its calculated piece to all the others.
- Each process updates its matrix.

4.4 Triangular implementation with AllGatherv

This is a mpi implementation of the triangular approach. This implementation starts defining the processes number and the id for each process.

- All processes divide the matrix into vector of vectors of triangles using **divide_upper_matrix_into_triangles**
- For each vector of triangles in the vector of vectors of triangles:
  - All processes compute the size of the triangles that each one have to compute.
  - Each process search for overlapping indices between the triangles in the vector.
  - Each process computes its triangle using the matrix.
  - Each process sends its calculated triangle to all the others.
  - Each process updates its copy of the matrix.
  - Each process computes again the overlapping indices.

4.5 Group optimization

For all implementations, we make an improvement defining a group of processes when the number of elements is less than the workers. In this way, we can avoid that processes, that have no work to do, receive results they will not use.

For each implementation, a Group of processes has been defined in order to avoid unused processes during the execution. In this way, we reduced communication overhead since that these precesses will not receive useless data.

4.6 Scalability analysis of Allgatherv implementations

Observing the strong scalability analysis plots, (**figure 7, 8 and 9**), MPI_AllG diagonals demonstrates better strong scalability across all problem sizes, achieving higher speedups and maintaining relatively better efficiency at higher processor counts. This indicates that MPI_AllG diagonals is generally more efficient in handling additional processors without excessive overhead. For both implementations, speedup and efficiency gains taper off as processors increase, particularly beyond 16 processors. This diminishing return is more pronounced for MPI_AllG triangles, suggesting higher parallel overhead or communication costs. MPI_AllG triangles incurs a significantly higher computational cost as the processor count increases, which implies that it might be more communication-heavy or have more redundant computations that become costly with more processors. Both implementations exhibit a sharp efficiency drop as the processor count rises. This indicates that for high processor counts, neither implementation maintains strong efficiency, though MPI_AllG diagonals is relatively better. The weak scalability analysis (**figure 10**) confirms the observations made in the strong scalability analysis.

4.7 Scalability analysis of diagonal implementation with Scatterv and Gatherv

Observing the plots, this approach is worse than sequential one because the overhead given by the distribution of dependencies to the processes is very expensive in terms of runtime. Strong scalability plots (**figure 11, 12 and 13**) shows initial benefits with added processors, but diminishing returns and increased overhead dominate beyond 8–16 processors. There is limited strong scalability at higher processor counts. Weak scalability plots (**figure 14**) displays poor scalability as both problem size and processor count grow. Increasing runtime and cost with more processors highlight inefficiencies and increased overhead.

4.8 Compile and execute

All codes showed in this chapter are located in **wavefront_mpi** directory and their names are **wavefront_diagonal_sg_mpi.cpp**, **wavefront_diagonal_allg_mpi.cpp** and **wavefront_triangles_allg_mpi.cpp**. In the base directory of the project, run:

- To compile: **make mpi** or **make all**
- To execute:
  - **mpirun -np ⟨number of workers⟩ wavefront_diagonal_sg_mpi ⟨size of rows/columns⟩**
  - **mpirun -np ⟨number of workers⟩ wavefront_diagonal_allg_mpi ⟨size of rows/columns⟩**
  - **mpirun -np ⟨number of workers⟩ wavefront_triangles_allg_mpi ⟨size of rows/columns⟩**

5 Conclusion

This report explores sequential, shared-memory (FastFlow), and distributed-memory (MPI) implementations of wavefront computation. In Chapter 2, the sequential analysis shows that the triangular implementation outperforms the diagonal approach by improving memory locality, though its efficiency peaks at an optimal number of subdivisions.

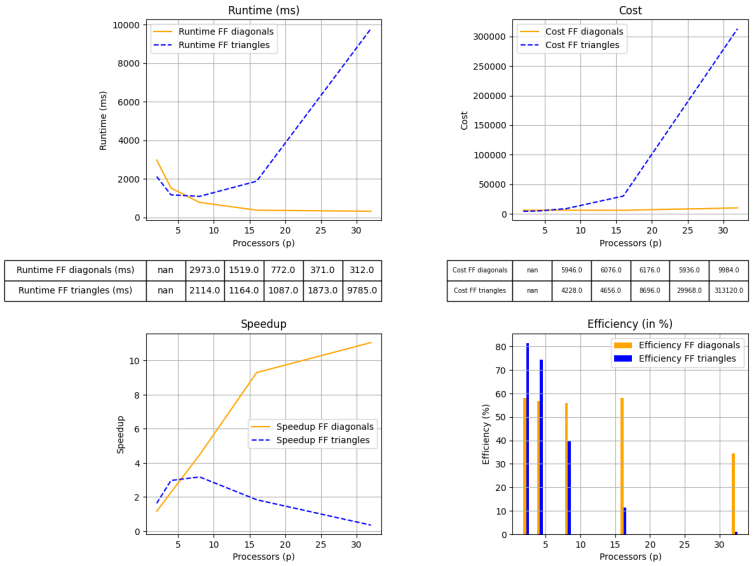
Chapter 3 extends this work to FastFlow parallelization, where the triangular approach achieves better performance on larger matrices but scales optimally with fewer threads, while the diagonal implementation scales more predictably with increasing workers.

Chapter 4 assesses MPI-based implementations, with the AllGatherv diagonal approach showing the best scalability and efficiency due to reduced communication overhead. The triangular MPI implementation also performs well in weak scalability but becomes less efficient at higher processor counts due to communication demands.

Overall, the triangular approach proves superior in shared-memory setups, while MPI AllGatherv with diagonal implementation is optimal for distributed environments, highlighting the importance of implementation choice based on memory architecture.

6 Appendix

6.1 FastFlow scalability plots



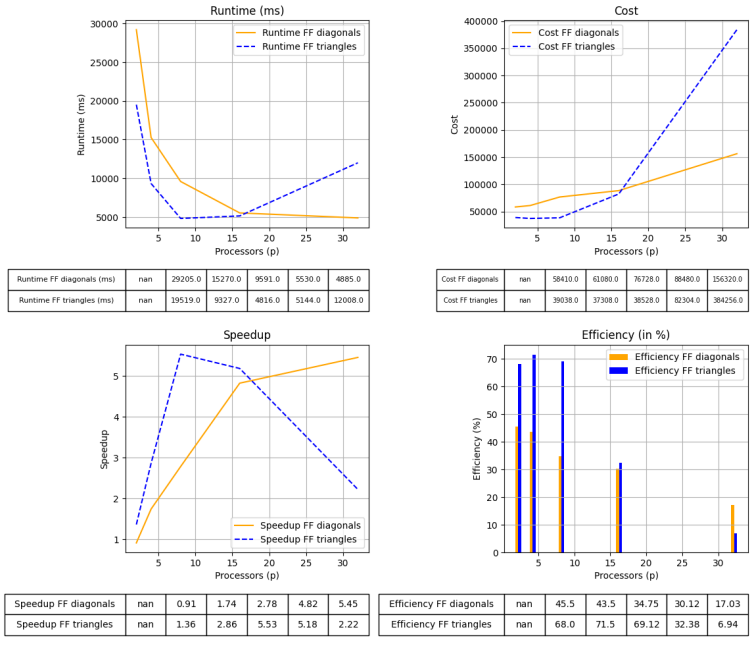
Runtime FF diagonals (ms)	nan	2973.0	1519.0	772.0	371.0	312.0
Runtime FF triangles (ms)	nan	2114.0	1164.0	1087.0	1873.0	9785.0

Cost FF diagonals	nan	5946.0	6076.0	6176.0	5936.0	9984.0
Cost FF triangles	nan	4228.0	4656.0	8696.0	29968.0	313120.0

Speedup FF diagonals	nan	1.16	2.27	4.47	9.3	11.06
Speedup FF triangles	nan	1.63	2.97	3.18	1.84	0.35

Efficiency FF diagonals	nan	58.0	56.75	55.88	58.13	34.56
Efficiency FF triangles	nan	81.5	74.25	39.75	11.5	1.09

Figure 3: Fastflow strong scalability with 2896x2896 matrix



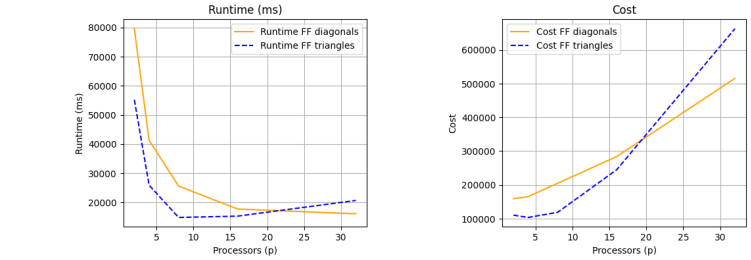
Runtime FF diagonals (ms)	nan	29205.0	15270.0	9591.0	5530.0	4885.0
Runtime FF triangles (ms)	nan	19519.0	9327.0	4816.0	5144.0	12008.0

Cost FF diagonals	nan	58410.0	61080.0	76728.0	88480.0	156320.0
Cost FF triangles	nan	39038.0	37308.0	38528.0	82304.0	384256.0

Speedup FF diagonals	nan	0.91	1.74	2.78	4.82	5.45
Speedup FF triangles	nan	1.36	2.86	5.53	5.18	2.22

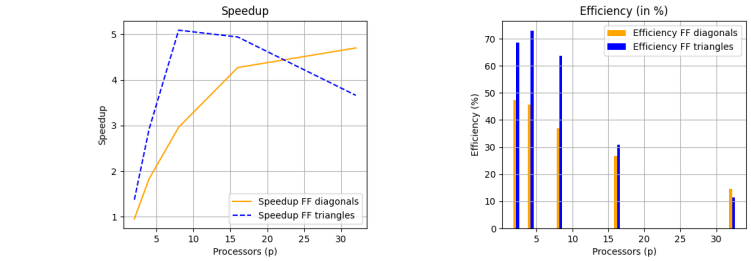
Efficiency FF diagonals	nan	45.5	43.5	34.75	30.12	17.03
Efficiency FF triangles	nan	68.0	71.5	69.12	32.38	6.94

Figure 4: Fastflow strong scalability with 5793x5793 matrix



Runtime FF diagonals (ms)	nan	79613.0	41316.0	25616.0	17758.0	16111.0
Runtime FF triangles (ms)	nan	55239.0	25919.0	14871.0	15327.0	20709.0

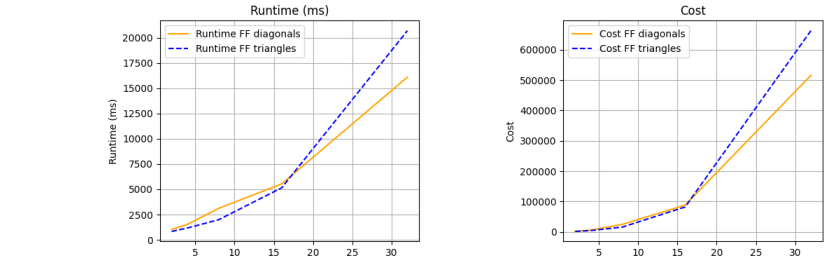
Cost FF diagonals	nan	159226.0	165264.0	204928.0	284128.0	515552.0
Cost FF triangles	nan	110478.0	103676.0	118968.0	245232.0	662688.0



Speedup FF diagonals	nan	0.95	1.83	2.96	4.27	4.7
Speedup FF triangles	nan	1.37	2.92	5.09	4.94	3.66

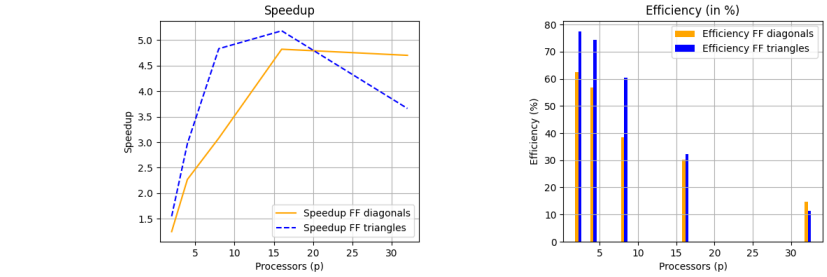
Efficiency FF diagonals	nan	47.5	45.75	37.0	26.69	14.69
Efficiency FF triangles	nan	68.5	73.0	63.62	30.88	11.44

Figure 5: Fastflow strong scalability with 8192x8192 matrix



Runtime FF diagonals (ms)	1020.0	1519.0	3116.0	5530.0	16111.0
Runtime FF triangles (ms)	823.0	1164.0	1990.0	5144.0	20709.0

Cost FF diagonals	2040.0	6076.0	24928.0	88480.0	515552.0
Cost FF triangles	1646.0	4656.0	15920.0	82304.0	662688.0

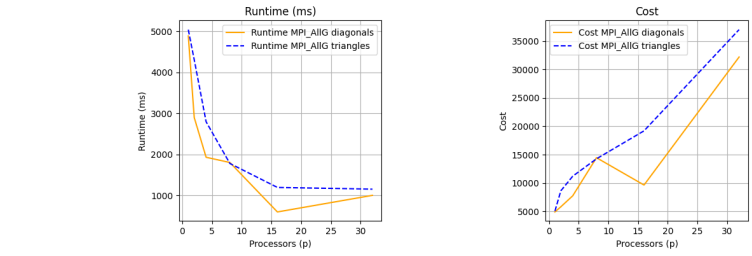


Speedup FF diagonals	1.25	2.27	3.08	4.82	4.7
Speedup FF triangles	1.55	2.97	4.83	5.18	3.66

Efficiency FF diagonals	62.5	56.75	38.5	30.12	14.69
Efficiency FF triangles	77.5	74.25	60.38	32.38	11.44

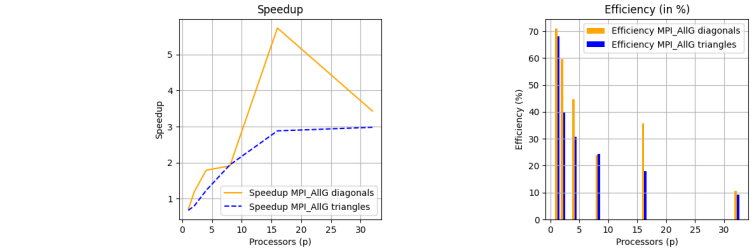
Figure 6: Fastflow weak scalability with processes 2,4,8,16,32 and size 2048, 2896, 4096, 5793, 8192

6.2 MPI AllGatherv scalability plots



Runtime MPI_AllG diagonals (ms)	4887	2900	1933	1806	602	1006
Runtime MPI_AllG triangles (ms)	5040	4307	2797	1782	1199	1157

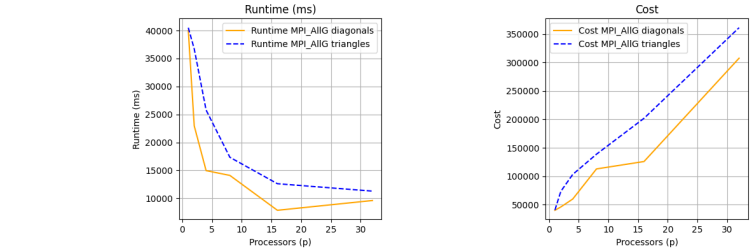
Cost MPI_AllG diagonals	4887	5800	7732	14448	9632	32192
Cost MPI_AllG triangles	5040	8614	11188	14256	19184	37024



Speedup MPI_AllG diagonals	0.71	1.19	1.79	1.91	5.73	3.43
Speedup MPI_AllG triangles	0.68	0.8	1.23	1.94	2.88	2.98

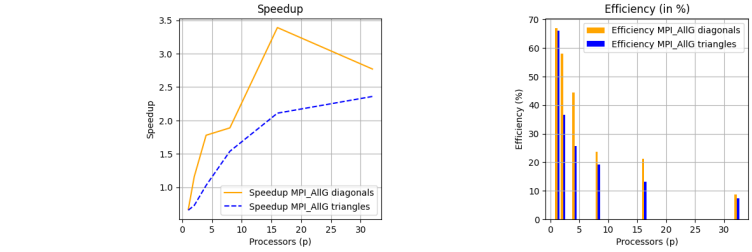
Efficiency MPI_AllG diagonals	71.0	59.5	44.75	23.88	35.81	10.72
Efficiency MPI_AllG triangles	68.0	40.0	30.75	24.25	18.0	9.31

Figure 7: MPI strong scalability with 2896x2896 matrix



Runtime MPI_AllG diagonals (ms)	40011	22949	14952	14096	7862	9610
Runtime MPI_AllG triangles (ms)	40501	36703	25776	17335	12613	11287

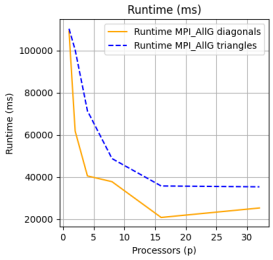
Cost MPI_AllG diagonals	40011	45898	59808	112768	125792	307520
Cost MPI_AllG triangles	40501	73406	103104	138680	201808	361184



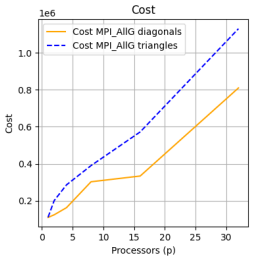
Speedup MPI_AllG diagonals	0.67	1.16	1.78	1.89	3.39	2.77
Speedup MPI_AllG triangles	0.66	0.73	1.03	1.54	2.11	2.36

Efficiency MPI_AllG diagonals	67.0	58.0	44.5	23.62	21.19	8.66
Efficiency MPI_AllG triangles	66.0	36.5	25.75	19.25	13.19	7.38

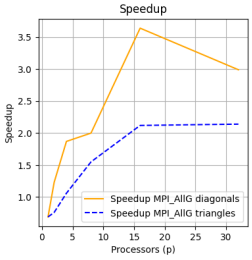
Figure 8: MPI strong scalability with 5793x5793 matrix



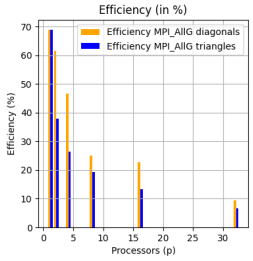
Runtime MPI_AllG diagonals (ms)	109064	61772	40506	37783	20834	25318
Runtime MPI_AllG triangles (ms)	110275	100235	71357	48730	35756	35340



Cost MPI_AllG diagonals	109064	123544	162024	302264	333344	810176
Cost MPI_AllG triangles	110275	200470	285428	389840	572096	1130880

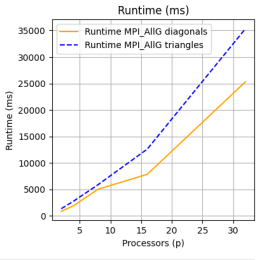


Speedup MPI_AllG diagonals	0.69	1.23	1.87	2.0	3.64	2.99
Speedup MPI_AllG triangles	0.69	0.76	1.06	1.55	2.12	2.14

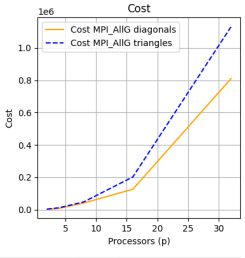


Efficiency MPI_AllG diagonals	69.0	61.5	46.75	25.0	22.75	9.34
Efficiency MPI_AllG triangles	69.0	38.0	26.5	19.38	13.25	6.69

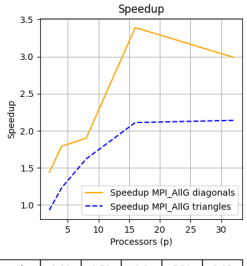
Figure 9: MPI strong scalability with 8192x8192 matrix



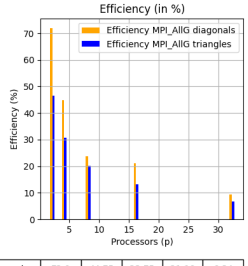
Runtime MPI_AllG diagonals (ms)	886	1933	5053	7862	25318
Runtime MPI_AllG triangles (ms)	1376	2797	5914	12613	35340



Cost MPI_AllG diagonals	1772	7732	40424	125792	810176
Cost MPI_AllG triangles	2752	11188	47312	201808	1130880



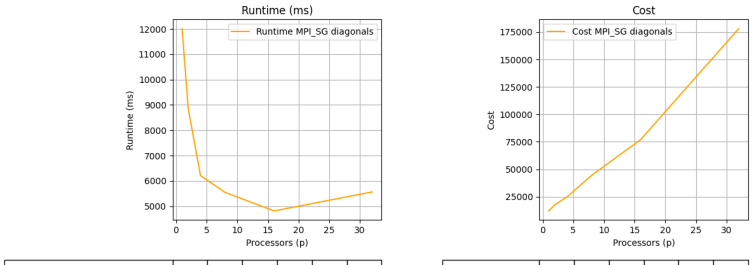
Speedup MPI_AllG diagonals	1.44	1.79	1.9	3.39	2.99
Speedup MPI_AllG triangles	0.93	1.23	1.62	2.11	2.14



Efficiency MPI_AllG diagonals	72.0	44.75	23.75	21.19	9.34
Efficiency MPI_AllG triangles	46.5	30.75	20.25	13.19	6.69

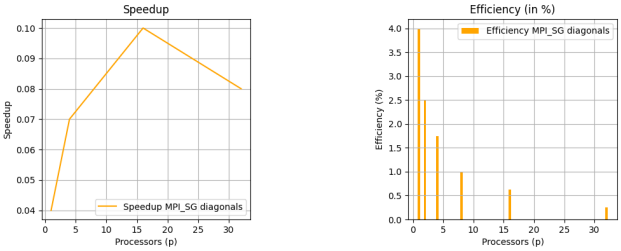
Figure 10: MPI weak scalability with processes 2,4,8,16,32 and size 2048, 2896, 4096, 5793, 8192

6.3 MPI Scatterv-Gatherv scalability plots



Runtime MPI_SG diagonals (ms)	12009	8844	6210	5547	4813	5564

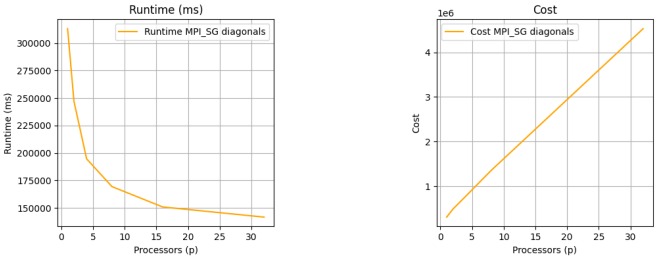
Cost MPI_SG diagonals	12009	17688	24840	44376	77008	178048



Speedup MPI_SG diagonals	0.04	0.05	0.07	0.08	0.1	0.08

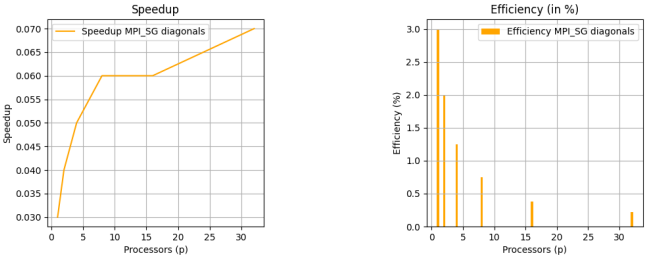
Efficiency MPI_SG diagonals	4.0	2.5	1.75	1.0	0.62	0.25

Figure 11: MPI_SG strong scalability with 1448x1448 matrix



Runtime MPI_SG diagonals (ms)	313268	247066	194622	169309	150771	141521

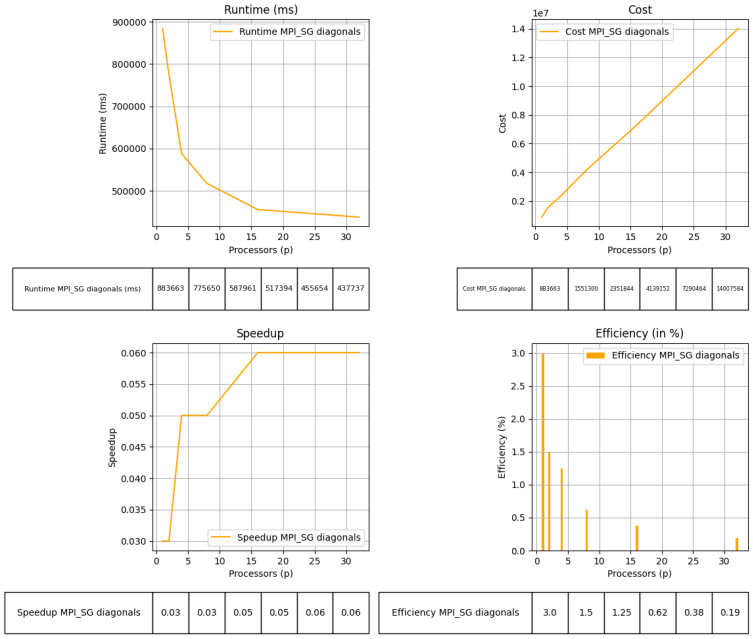
Cost MPI_SG diagonals	313268	494132	778488	1354472	2412336	4528672



Speedup MPI_SG diagonals	0.03	0.04	0.05	0.06	0.06	0.07

Efficiency MPI_SG diagonals	3.0	2.0	1.25	0.75	0.38	0.22

Figure 12: MPI_SG strong scalability with 4096x4096 matrix



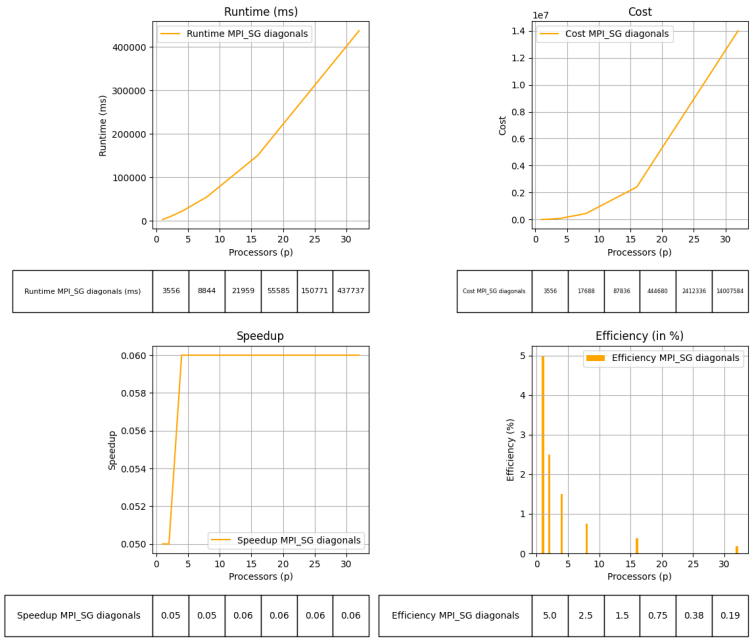
Runtime MPI_SG diagonals (ms)	883663	775650	587961	517394	455654	437737

Cost MPI_SG diagonals	883663	1551300	2351844	4139152	7290464	14007584

Speedup MPI_SG diagonals	0.03	0.03	0.05	0.05	0.06	0.06

Efficiency MPI_SG diagonals	3.0	1.5	1.25	0.62	0.38	0.19

Figure 13: MPI_SG strong scalability with 5793x5793 matrix



Runtime MPI_SG diagonals (ms)	3556	8844	21959	55585	150771	437737

Cost MPI_SG diagonals	3556	17688	87836	444680	2412336	14007584

Speedup MPI_SG diagonals	0.05	0.05	0.06	0.06	0.06	0.06

Efficiency MPI_SG diagonals	5.0	2.5	1.5	0.75	0.38	0.19

Figure 14: MPI_SG weak scalability with processes 1,2,4,8,16,32 and size 1024, 1448, 2048, 2896, 4096, 5793