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 Problem evaluation **3**

2 Sequential implementation **4**
2.1 Implementation 4
2.2 Compile and execute 4

3 Parallel implementation **5**
3.1 Observation 5
3.2 Fastflow implementation 5
3.3 Compile and execute Fastflow implementation 5
3.4 MPI implementation 5
3.5 Compile and execute MPI implementation 6

4 Results **7**
4.1 Strong Scalability Analysis 7
4.2 Weak Scalability Analysis 8

5 Conclusion **13**

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 belog to the previous diagonal. For this reason, a possible approach is to compute all elements belog to the same diagonal, before to pass the next one.

2 Sequential implementation

2.1 Implementation

For the sequential 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. A problem that we can observe is the poor spatial locality when we access a column e^k_j 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. The code is in **wavefront.cpp**.

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 2: wavefront computation with good spacial locality

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

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

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

2.2 Compile and execute

In the base directory of the project, run:

- To compile: **make sequential or make all**
- To execute: **./wavefront ⟨size of rows/columns ⟩**

3 Parallel implementation

3.1 Observation

As we can see from the table 1, the elements $e^k_{i-1,j}$ and $e^k_{i,j+1}$ are required to compute the element $e^k_{i,j}$. This means the elements of the previous diagonal ($e^k_{i-1,j}$ and $e^k_{i,j+1}$) are required for the computation of the current one, but the elements belog to the same diagonal can be computed in parallel because is independent of each other. According to the track and following this observation, we implement the parallel version of the code with two different algorithms using:

- fastflow (FF)
- Message Passing Interface (MPI)

3.2 Fastflow implementation

The wavefront computation proceeds diagonally across the matrix, starting from the diagonal just above the main diagonal and continuing upwards. To achieve parallelism, we define a ParallelFor to distribute the computation of each element along the diagonal across multiple threads. The ParallelFor function manages the division of work among the specified number of worker threads (nworkers) enabling the computation to be carried out efficiently in parallel.

3.3 Compile and execute Fastflow implementation

In the base directory of the project, run:

- To compile: **make ff** or **make all**
- To execute: **./wavefront_ff** ⟨**size of rows/columns**⟩

3.4 MPI implementation

At the beginning of the program, the MPI environment is initialized, and the total number of processes involved in the computation, as well as the rank (or ID) of each process, is determined. These ranks are crucial because they dictate how the workload is divided and managed across the different processes. The wavefront algorithm itself is implemented within a nested loop structure. The outer loop iterates through the diagonals of the matrix, beginning with the second diagonal (since the first diagonal is already initialized). For each diagonal, the computation is distributed across the available processes. The work is divided using a function **divide_job_into_parts** that calculates the number of elements each process should handle and identifies the starting point for each process's computation. Within their assigned portion, each process computes values based on elements of the matrix that have already been calculated in previous iterations. The computed values are then stored in a local vector. To maintain consistency and ensure that all processes have the necessary data for the next iteration, the program uses the **MPI_Allgatherv** function. This function collects the computed values from all processes and distributes the complete set of results to every process. This step is crucial because the wavefront algorithm relies on the availability of previously computed values to calculate the next set of results. Once the values for the current diagonal have been gathered, each process updates its local copy of the matrix with these new values. This update prepares the matrix for the next iteration, where the process will work on the subsequent diagonal. '

3.5 Compile and execute MPI implementation

In the base directory of the project, run:

- To compile: **make mpi** or **make all**
- To execute:
  - **salloc -N ⟨number of nodes⟩**
  - **mpirun -n ⟨number of nodes⟩ ./wavefront_mpi ⟨size of rows/columns⟩** or **srun –mpi=pmix -n ⟨number of nodes⟩ ./wavefront_mpi ⟨size of rows/columns⟩**

4 Results

To evaluate the different implementations, we conducted a scalability analysis to assess the performance of the parallelized code under varying conditions, including the number of processors/cores and the input size. Specifically, we performed two types of scalability analysis:

- **Strong scalability**: analize the programs behavior varing number of processors and fix the data size
- **Weak scalability**: analize the programs behavior varing both number of processors and the data size

For each analysis, we employed four key metrics to compare the different versions of the code: runtime, speedup, efficiency, and cost.

- **The speedup** (S) is defined as the quotient of the time taken using a single processor (T(1)) over the time measured using p processors (T(p)):

$$S = \frac{T_{seq}}{T_{par}(p)}$$

- **Efficiency** (E) relates the speedup to the number of utilized processors or cores. It is defined as the quotient of S over p:

$$E = \frac{S}{p} = \frac{T(1)}{T(p) * p}$$

- **Cost** T(p) × p is also called cost of parallelization (or simply cost).

4.1 Strong Scalability Analysis

Powerful scalability analysis evaluates how performance scales when the number of processors increases while maintaining the size of the problem. Three different situations with matrix sizes of 8192, 5793 and 2896 respectively are analysed in terms of runtime, cost, speedup and efficiency. The results are broken down by case below.

Analizing the 8192 matrix size analysis, the execution time of both FastFlow and MPI implementations decreases with increasing number of processors. MPI shows slightly better runtime for the larger number of processors (16 processors) than Fastflow. The cost (number of running processors) shows a linear increase for FastFlow and MPI up to 16 processors. The cost of MPI increases less rapidly after 8 processors, suggesting that it becomes more efficient at higher processor counts. The speed improvement for both implementations improves as the number of processors increases. FastFlow shows better acceleration performance up to 8 processors, while MPI recovers after this. Productivity decreases with the number of processing units. FastFlow maintains higher efficiency up to 8 processors, but beyond that, its efficiency drops rapidly, dropping to 29.06% at 16 processors. MPI shows better efficiency beyond the 8 processors but still records significant decreases, reaching 33.62% with 16 processors.

Looking at the 5793 matrix size analysis, the runtime decreases as processors are added. The cost of implementing FastFlow increases linearly compared to MPI, which stabilizes after 8 processors. Speedup for FastFlow is slightly better than MPI below 8 processors, a situation that flips

over reaching 4.82x on 16 processors, while MPI reaches 5.32x. Efficiency decreases for both implementations as the number of processors increases. FastFlow efficiency drops to 30.12% at 16 processors, while MPI reaches 33.25%, showing a lower degradation of efficiency.

The smaller size of the problem shows a faster decrease in runtime as the number of processors increases, with FastFlow maintaining a slight runtime advantage over MPI. The cost is constantly increasing as the number of computers increases to 8, which decreases, indicating better management of workloads. For this smaller issue, FastFlow reaches a 10.14x Speed on 16 processors, significantly better than MPI 8.94x. This strong speed increase to 16 processors demonstrates better scalability for smaller issues with FastFlow. Both implementations show improved efficiency over larger issues, with FastFlow reaching 63.38% efficiency at 16 processors, while MPI reaches 55.88%. Implementation in FastFlow shows greater efficiency for smaller problems, demonstrating better adaptability for small-scale tasks. This trend may be due to the fact that run times (in milliseconds) are very low and can significantly change performance even with small variations.

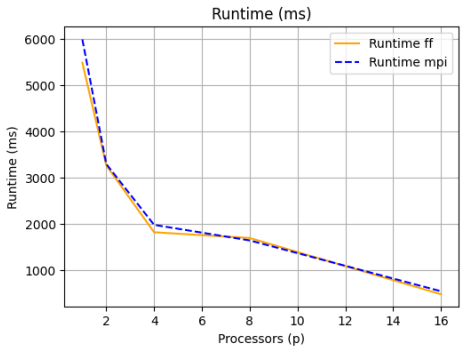
FastFlow generally performs better in terms of runtime and efficiency for smaller to medium problem sizes, but MPI exhibits better scalability at higher processor counts, particularly for larger workloads. The diminishing returns in efficiency for both implementations indicate that neither is infinitely scalable without performance degradation, but FastFlow is more suited for smaller problem sizes, while MPI might excel in scenarios where larger tasks and higher processor counts are needed.

4.2 Weak Scalability Analysis

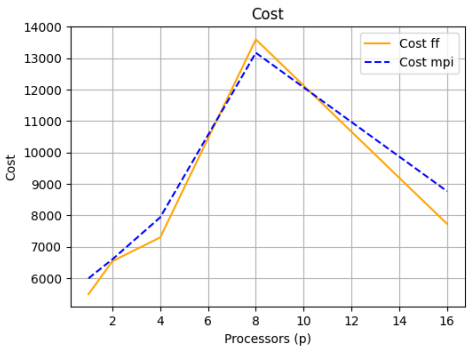
Weak scalability shows the behavior of our parallel code for varying both the number of processors and the input data size; when doubling the number of processors we also double the input of matrix data size. Ideally, our goal is to keep the problem size per processor constant.

FastFlow has a generally increasing runtime with the number of processors. The increase is non-linear, with a significant jump after 8 processors. MPI, on the other hand, starts with a higher runtime with 2 processors but has a smoother and less steep curve as the number of processors increases. The cost of FastFlow increases significantly with the number of processors. MPI follows a similar pattern, but with a lower overall cost than higher processor counts. FastFlow speed starts at 0.93 with 2 processors and increases to 4.65 with 16 processors. This indicates a moderate increase in acceleration, but it indicates that the implementation does not achieve linear acceleration. MPI shows a slightly better speed curve, starting from 0.69 and growing to 5.38 with 16 processors. This suggests that MPI resizes better than FastFlow, especially with more processors.

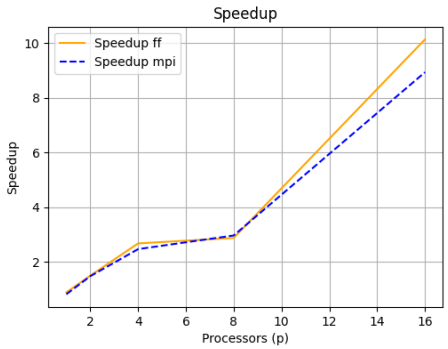
The weak scalability confirms the observations made in the strong scalability suggesting a better performance of MPI for large problems with high number of processors, inverse for fastflow



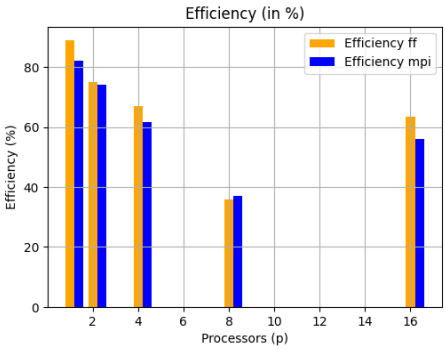
Runtime ff (ms)	5489	3271	1824	1700	483
Runtime mpi (ms)	5995	3300	1984	1647	548



Cost ff	5489	6542	7296	13600	7728
Cost mpi	5995	6600	7936	13176	8768

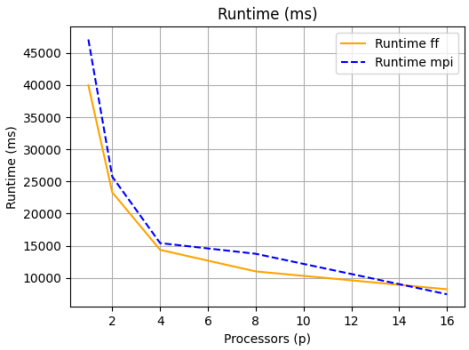


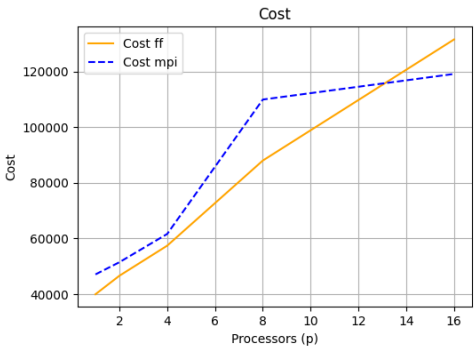
Speedup ff	0.89	1.5	2.68	2.88	10.14
Speedup mpi	0.82	1.48	2.47	2.97	8.94



Efficiency ff	89.0	75.0	67.0	36.0	63.38
Efficiency mpi	82.0	74.0	61.75	37.12	55.88

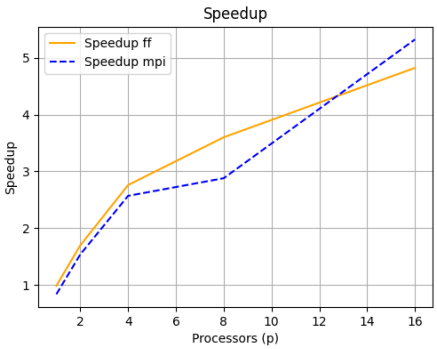
Figure 1: Strong Scalability analysis with matrix size 2896x2896

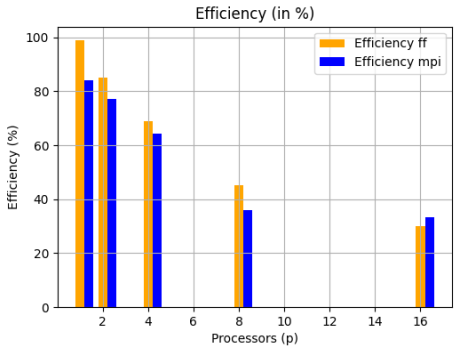




Runtime ff (ms)	39959	23301	14355	11003	8223
Runtime mpi (ms)	47097	25735	15409	13745	7448

Cost ff	39959	46602	57420	88024	131568
Cost mpi	47097	51470	61636	109960	119168

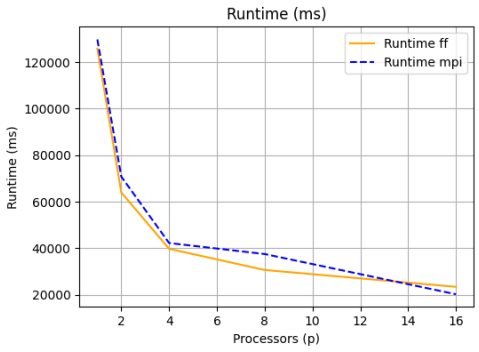




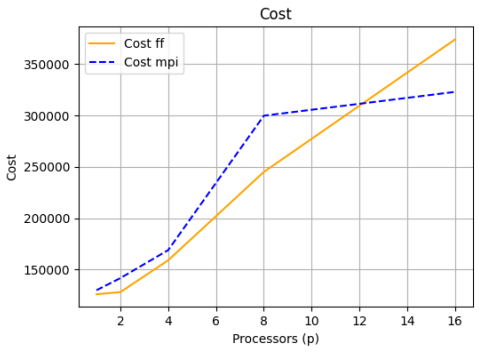
Speedup ff	0.99	1.7	2.76	3.6	4.82
Speedup mpi	0.84	1.54	2.57	2.88	5.32

Efficiency ff	99.0	85.0	69.0	45.0	30.12
Efficiency mpi	84.0	77.0	64.25	36.0	33.25

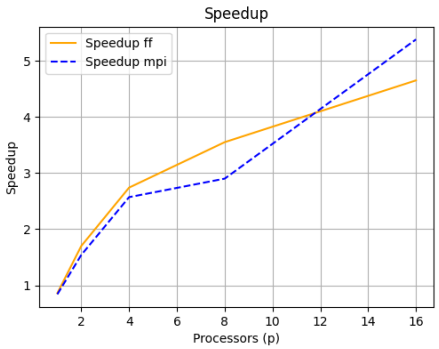
Figure 2: Strong Scalability analysis with matrix size 5793x5793



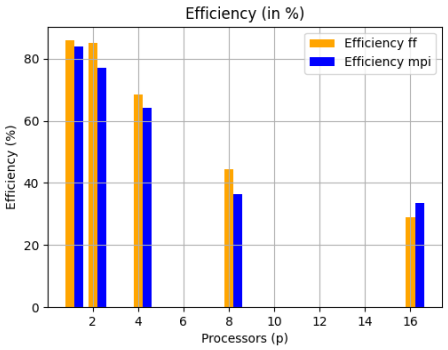
Runtime ff (ms)	125840	63955	39735	30625	23385
Runtime mpi (ms)	129764	70780	42218	37479	20190



Cost ff	125840	127910	158940	245000	374160
Cost mpi	129764	141560	168872	299832	323040

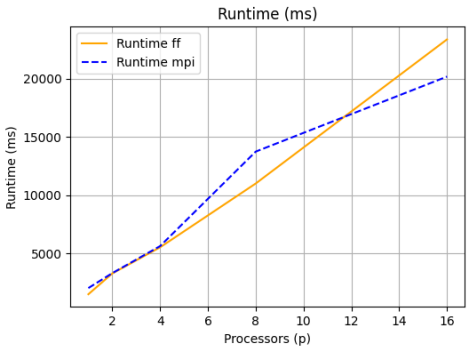


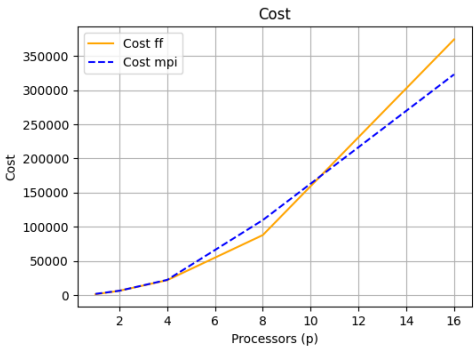
Speedup ff	0.86	1.7	2.74	3.55	4.65
Speedup mpi	0.84	1.54	2.57	2.9	5.38



Efficiency ff	86.0	85.0	68.5	44.38	29.06
Efficiency mpi	84.0	77.0	64.25	36.25	33.62

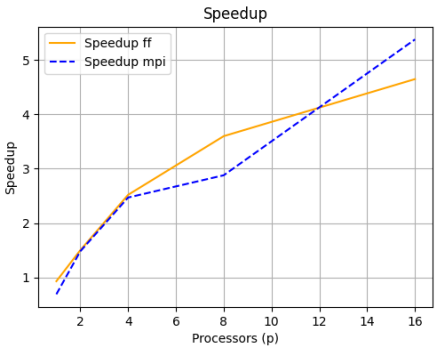
Figure 3: Strong Scalability analysis with matrix size 8192x8192

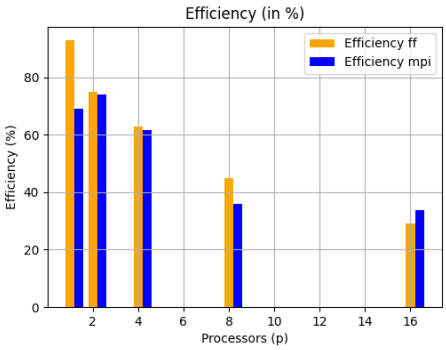




Runtime ff (ms)	1486	3271	5520	11003	23385
Runtime mpi (ms)	2018	3300	5624	13745	20190

Cost ff	1486	6542	22080	88024	374160
Cost mpi	2018	6600	22496	109960	323040





Speedup ff	0.93	1.5	2.52	3.6	4.65
Speedup mpi	0.69	1.48	2.47	2.88	5.38

Efficiency ff	93.0	75.0	63.0	45.0	29.06
Efficiency mpi	69.0	74.0	61.75	36.0	33.62

Figure 4: Weak Scalability analysis with matrix size 2048x2048

5 Conclusion

Comparing implementations in FastFlow and MPI based on strong and weak scalability tests highlights the main trade-offs between the two structures. The implementation in FastFlow initially offers lower costs and greater efficiency, making it effective for small parallel systems, especially with up to 4 processors. However, as the number of processors increases, FastFlow's performance decreases rapidly.

In contrast, the MPI implementation presents a more robust scalability. Despite slightly higher initial costs, MPI maintains stable efficiency and achieves an almost linear speed as the number of processors increases. This makes it more suitable for large-scale parallelism, where controlled cost growth and sustained performance are crucial.

Overall, while FastFlow implementation can be beneficial for small-scale parallelism, the superior scalability of MPI deployment and consistent performance at large scales make it a preferred choice.

That said, it is possible to make some improvements to reduce the overhead of communication between processes/threads. The implementation carried out divides the computation of the current diagonal between the workers who, in the case of fastflow they access the shared memory to save the results while in mpi they communicate to update the matrix. Since the element i,j depends on i-1,j and i,j+1 an alternative division would be to divide the computation of the triangular matrix into smaller independent triangles so as to increase the computation for individual workers and reduce communication/memory access.

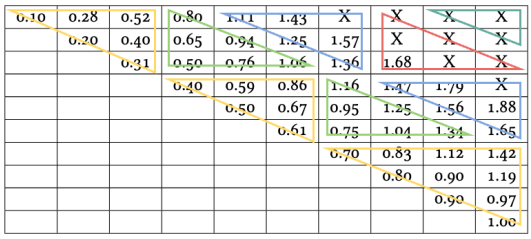


Figure 5: alternative data division

In this Alternative, the communication is reduced to the number of triangles. Triangles of different colors depend on the adjacent triangles to the left.

Another possible improvement would be to exploit the intrinsics in c++ to manage the computation of the dotproduct directly in internal registers instead of in RAM. These two improvements could be useful for future work.