Università di Pisa

Department of Computer Science

Parallel and Distributed Systems:
Paradigms and Models

Project 1:
Distributed Wavefront Computation

Student — ID

Michele Mattiello — ***603900***

A.Y. 2023-2024

Contents

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, we start to initialize the main diagonal of the matrix 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 matrix⟩**

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 matrix⟩**

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 matrix⟩** or **srun –mpi=pmix -n ⟨number of nodes⟩ ./wavefront_mpi ⟨size of matrix⟩**

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.

4.1 Speedup

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)):

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

The ideal speedup is linear, meaning that the maximum achievable speedup with p processors is p. In some cases, super-linear speedup can occur, where the speedup exceeds the number of processors.

4.2 Efficiency

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

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

An efficiency of 1 indicates perfect scalability.

4.3 Cost

T(p) $\times$ p is also called cost of parallelization (or simply cost).

4.4 Strong Scalability Analysis

With strong scalability analysis, we measure efficiencies for a varying number of processors and keep the input data size fixed. In our tests, we fix the size of matrix at 4096x4096. The obtained runtimes are:

Processors	Matrix size	seq runtime (ms)	FF runtime (ms)	MPI runtime (ms)
1	4096	13203	13534	16395
2	4096	_	7214	8517
4	4096	_	4214	4717
8	4096	_	3173	3584
16	4096	_	3540	2260

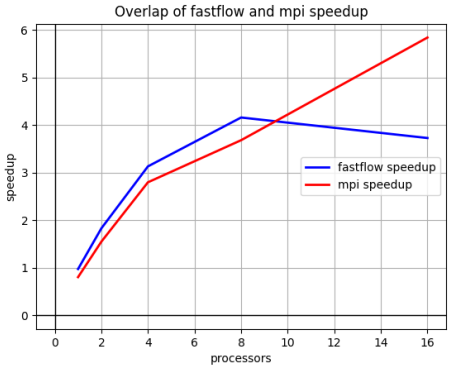

Figure 1: Speedup plot

processors	Fastflow	MPI
1	0.98	0.81
2	1.83	1.55
4	3.13	2.8
8	4.16	3.68
16	3.73	5.84

Table 3: Speedup results

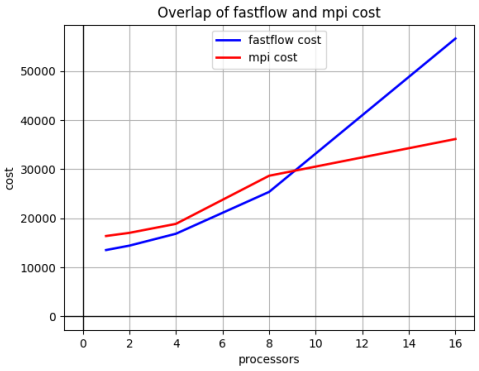

Figure 2: Cost plot

processors	Fastflow	MPI
1	13534	16395
2	14428	17034
4	16856	18868
8	25384	28672
16	56640	36160

Table 4: Cost results

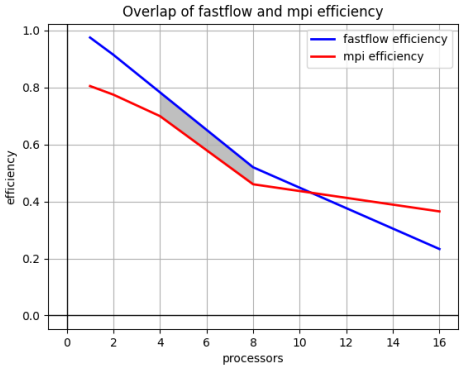

Figure 3: Efficiency plot

processors	Fastflow	MPI
1	0.98	0.81
2	0.92	0.78
4	0.78	0.70
8	0.52	0.46
16	0.23	0.37

Table 5: Efficiency results

As processor count increases, the FastFlow implementation initially shows lower cost and higher efficiency compared to MPI. However, as the scale grows, FastFlow's costs rise sharply, and its efficiency declines faster than MPI's. While both implementations exhibit similar speedup up to 4 processors, MPI continues to scale more effectively, maintaining a linear speedup as FastFlow plateaus and declines beyond 8 processors. Consequently, for smaller scales, FastFlow may be more efficient and cost-effective. However, at larger scales, MPI outperforms FastFlow with lower costs, better efficiency, and superior speedup, making it the preferable choice for high parallelism.

4.5 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 data size. Ideally, our goal is to keep the problem size per processor constant. The obtained runtimes are:

Processors	Matrix size	seq runtime (ms)	FF runtime (ms)	MPI runtime (ms)
1	2048	1006	1114	2022
2	2896	3896	1411	3307
4	4096	13168	4328	5584
8	5793	39088	11743	13738
16	8192	107787	33389	20188

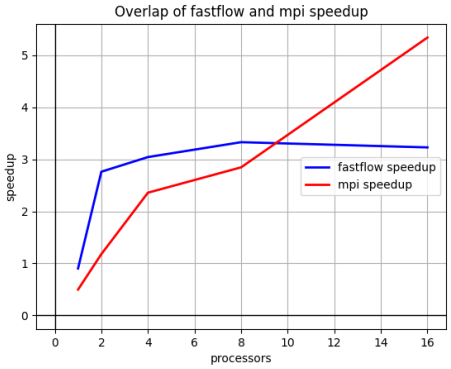

Figure 4: Speedup plot

processors	Fastflow	MPI
1	0.90	0.50
2	2.76	1.18
4	3.04	2.36
8	3.33	2.85
16	3.23	5.34

Table 6: Speedup results

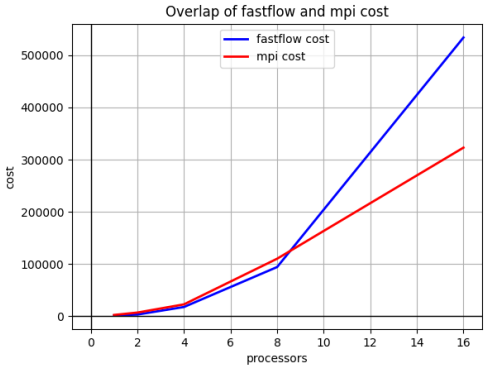

Figure 5: Cost plot

processors	Fastflow	MPI
1	1114	2022
2	2822	6614
4	17312	22336
8	93944	109904
16	534224	323008

Table 7: Cost results

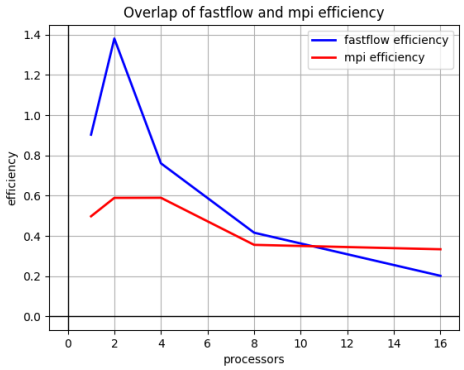

Figure 6: Efficiency plot

processors	Fastflow	MPI
1	0.90	0.50
2	1.38	0.59
4	0.76	0.59
8	0.42	0.36
16	0.20	0.33

Table 8: Efficiency results

The weak scalability analysis confirms the observations made in the strong scalability analysis. The weak scalability analysis of FastFlow and MPI reveals distinct differences. FastFlow initially shows strong performance, with higher efficiency and speedup for small processor counts. However, its performance deteriorates rapidly as the number of processors increases. By 8 processors, its cost rises sharply, and both efficiency and speedup stagnate, indicating poor scalability.

In contrast, MPI exhibits more controlled cost growth, with efficiency remaining stable over a larger range of processors. Its speedup consistently increases, surpassing FastFlow at higher processor counts. MPI handles additional processors more effectively, demonstrating better resource utilization and sustained performance gains.

Overall, while FastFlow performs well in smaller systems, MPI's more stable cost, higher scalability in speedup, and better efficiency retention make it a superior choice for large-scale parallelism and weak scalability scenarios.

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: efficiency drops, costs increase dramatically and acceleration plateaus decrease to more than 8 processors.

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 implementation in FastFlow may be advantageous for small-scale parallelism, the superior scalability of implementation in MPI, Improved storage efficiency and consistent performance at large scales make it the preferred choice for high-performance computing tasks that require extensive parallel processing.