UNIVERSITÀ DI PISA

PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

PROJECT 1 - DISTRIBUTED WAVEFRONT COMPUTATION

De Castelli Fabrizio *f.decastelli@studenti.unipi.it*

Academic Year 2023–2024

Contents

Introduction

The upper-triangular wavefront computation pattern is a parallel computing strategy used to efficiently perform operations on the diagonal elements of a matrix A of size $n \times n$. Specifically, this pattern focuses on computing diagonal elements starting from the main diagonal and moving upwards towards the first row.

In this context, for each diagonal element of the matrix A, the computation is distributed by calculating the $n-k$ diagonal element $e^k_{m,m+k}$ ($m \in [0, n-k[$) as the result of a dot product operation between two vectors $\mathbf{v}^k_m$ and $\mathbf{v}^k_{m+k}$, both of size k, composed by the elements on the same row m and on the same column $m+k$. Specifically,

$$e^k_{i,j} = \sqrt[3]{\langle \mathbf{v}^k_m, \mathbf{v}^k_{m+k} \rangle}$$

The values of the element on the major diagonal $e^0_{m,m}$ are initialized with the values $\frac{m+1}{n}$.

This report shows the implementation choices to address this problem in both the sequential and parallel versions as well as their performance analysis.

Algorithms

2.1 Sequential

The sequential version of the algorithm simply consists of two nested for loops:

- The outer has $n-1$ iterations and identifies the diagonal of which we would like to compute elements;
- The inner loop identifies the element inside the diagonal and computes the cubic root of the dot product of the associated row and column vectors.

Initially, a version that stored the matrix in a `std::vector<std::vector<double>>` was considered but was not efficient since the compiler couldn't allocate the 2D vector in a contiguous block. Instead, the matrix is stored as a contiguous $1d$ array. To access the element at position (i, j) we can simply multiply the row offset i by its dimension n and sum j, leading to an expression of the form $A[i * n + j]$. This allows to have a more efficient memory management during the execution of the program.

Algorithm 1: Sequential-Wavefront

Input : **n** dimension

```
1  double**A
2  for m ∈ 0...n−1 do
3  |  A[m∗n+m] = (m+1)/n
4  end
5  for k ∈ 1...n−1 do
6  |  for m ∈ 0...n−k do
7  |  |  double res ← 0.0;
8  |  |  for i ∈ 0...k do
9  |  |  |  res ← res + A[m∗n+m+i] ∗ A[(m+i+1)∗n+m+k]
10 |  |  end
11 |  |  A[m∗n+m+k] ← std::cbrt(res)
12 |  end
13 end
```

2.2 Fastflow

The *fastflow* version of the solution takes advantage of the farm pattern with feedback loops. In particular, we have an *emitter* thread that has the goal of emitting tasks that have to be executed by *worker* threads. The task has been serialized with a proper `struct Task` and contains all the relevant information that a worker has to know in

order to complete it. The pattern does not include a collector since it was not very useful as the workers themselves could directly write computed elements to the right matrix memory location (passed in **Task** as a pointer).

The emitter is an object with state variables to keep track of the matrix, the number of spawned threads, the problem size n and the equivalent of the counter of the outer-most for loop in the sequential version, namely k. The latter is incremented every time all necessary workers have been spawned with `ff_send_out`, until all $n-1$ major diagonals have been computed. The local size of a worker workload is computed by indexing workers with integers ranging from 0 to the number of necessary workers defined as follows:

$$nec_w = \left\lfloor \frac{n-k}{n_w} \right\rfloor == 0 \; ? \; [(n-k) \mod n_w : n_w] \tag{2.1}$$

where n_w is the number of workers passed by command line.
The condition to spawn threads is that the state variable *spawned* is equal to zero: all workers have returned a valid task pointer and the emitter has decremented the counter in the proper conditional statement at line 4-6 of algorithm 3. The termination is reached when $k == n$. The pseudocodes of the emitter and worker are shown in algorithm 3 and algorithm 2 respectively.

Workers, instead, only have to use the task parameter to compute the cubic root of the dot product and return the task to notify the worker to decrement the *spawned* state variable. The pseudocode for the worker is shown in algorithm 2.

Finally, the main code initializes the matrix, starts the emitter and waits for the end of the execution.

Algorithm 2: Worker-svc()

Input : Task* *task*

```
1 n ← task.n
2 for m ∈ task.chunk_start ... task.chunk_end do
3 |   double res ← 0.0
4 |   for i ∈ 0 ... k − 1 do
5 |   |   res ← res + task.A[m ∗ n + m + i] ∗ task.A[(m + 1 + i) ∗ n + m + k]
6 |   end
7 |   task.A[m ∗ n + m + k] ← std::cbrt(res)
8 end
9 return task
```

Algorithm 3: Emitter-svc()

Input : Task* *task*

```
1  if this.k == this.n then
2  |  return EOS
3  end
4  if task ≠ nullptr then
5  |  this.spawned ← this.spawned − 1
6  end
7  if this.spawned == 0 then
8  |  nec_w ← number of necessary workers based on k (Equation 2.1)
9  |  Compute workers offsets with balancing
10 |  begin ← 0
11 |  for worker_id ∈ 0 . . . nec_w do
12 |  |  size ← local size of a worker based on worker_id
13 |  |  ff_send_out(newTask(this.A, begin, begin + size, n, k))
14 |  |  begin ← begin + size
15 |  |  this.spawned ← this.spawned + 1
16 |  end
17 |  this.k ← this.k + 1
18 end
19 return GO_ON
```

2.3 MPI

An initial version was implemented with P2P communication using the utilities `MPI_[I]send` and `MPI_[I]recv`. Similarly to the Fastflow version we have an emitter and multiple workers: at iteration k the emitter sends the row and column vector stacked together in a single vector of dimension $2k$ to the next available worker. The emitter then writes to the correct location the result collected by the worker. However, this approach was not optimal due to its communication overhead between processes so the analysis is omitted.

The final version took advantage of collective communication to impose a well-balanced workload between processes. The utility `MPI_Allgatherv` was used to collect all partial results of processes together at each iteration. Each process computes the number of elements it has compute the cubic root of by combining the iteration index k with its rank in the MPI environment. If a process as rank r, with $r \in \{0, \dots n_p\}$, and n_p is the number of available processes, it has to compute

$$\left\lfloor \frac{n-k}{n_p} \right\rfloor + [r < (n-k) \mod n_p] \tag{2.2}$$

elements of the $n-k$ elements of the k^{th} major diagonal. In this way all processes have equal workload except the first $(n-k) \mod n_p$ which have one more. Since each process has its own local copy of the initialized matrix, it is enough to compute the offset and use the number of assigned elements as parameters of a single `MPI_Allgatherv` in order to synchronize all processes with the updated diagonal.

Algorithm 4: MPI-Collective-Wavefront

Input : n dimension

1 `MPI_Init` the environment
2 initialize(A)
3 `MPI_Barrier()`
4 **for** $k \in 1 \dots n-2$ **do**
5 Compute processes offsets as in Equation 2.2
6 `std::vector<double>` partial
7 **for** $m \in chunk\_start \dots chunk\_end$ **do**
8 **for** $i \in 0 \dots k-1$ **do**
9 $partial[m - start] += A[m*n+m+i] * A[(m+i+1)*n+m+k]$
10 **end**
11 $partial[m*n+m+k] \leftarrow$ `std::cbrt`(res)
12 **end**
13 `MPI_Allgatherv( ... )`
14 **end**
15 ▷ $A[n-1]$ computation cannot be parallelized
16 **for** $i \in 0 \dots n-2$ **do**
17 $A[n-1] += A[i] * A[(i+1)*n+n-1]$
18 **end**

Performance Analysis

After a small consideration on the running time of all versions, in this chapter we evaluate its parallel implementations with two metrics: speedup and efficiency. The evaluation is done considering both cases of strong analysis and weak analysis. In all upcoming results we assume that the matrix is provided as input with the filled diagonal and every metric has been averaged by collecting data from 5 runs.

3.1 Running Times

In Table 3.1, Table 3.2 and Table 3.3 are reported the running times of the three algorithms considering different configurations.

n	200	400	800	1600	3200
time (ms)	2.00	19.00	143.60	1582.20	19635.40

Table 3.1: *Sequential running times*

n	200	400	800	1600	3200
w			**time (ms)**		
1	3.00	20.60	157.00	1712.00	21090.40
2	2.00	10.60	78.60	859.00	10831.80
4	1.20	6.00	40.00	444.40	5917.80
8	2.20	5.00	22.80	232.00	3134.60
12	3.00	5.00	17.00	157.40	1604.60
16	4.00	6.80	20.40	144.60	1291.20
20	4.00	7.00	25.00	185.80	1903.40

Table 3.2: *Fastflow running times*

In the case of MPI the running time is the average of all processes running times. The best configuration is the one with 2 nodes as overall.

n / **p**	200	400	800	1600	3200
			time (ms)		
1 node					
2	1.86	11.03	79.41	812.17	7454.74
4	1.87	10.12	48.63	470.33	5096.73
8	2.13	9.18	46.85	269.51	2526.73
12	2.59	9.10	44.69	231.13	2021.15
16	2.19	5.75	22.22	156.53	1483.58
20	3.13	11.04	46.68	248.01	2241.20
2 nodes					
2	9.84	29.09	116.78	964.35	7835.07
4	15.35	32.69	87.05	588.98	3989.27
8	20.99	37.66	126.59	435.30	2481.91
12	25.62	46.51	118.26	424.87	2016.37
16	28.00	53.18	118.23	334.59	1587.39
20	29.88	55.23	140.17	392.41	1605.38
4 nodes					
4	18.15	41.51	115.68	631.91	4047.50
8	23.24	48.39	143.33	471.49	2434.65
12	25.04	49.81	135.72	436.91	2013.95
16	33.19	67.46	132.81	384.42	1651.78
20	34.47	80.64	163.01	425.10	1660.36
8 nodes					
8	24.74	56.80	150.99	505.65	2506.60
12	29.78	59.77	148.45	453.44	2007.48
16	41.26	77.04	147.60	413.47	1703.52
20	40.25	78.89	176.25	445.49	1698.86

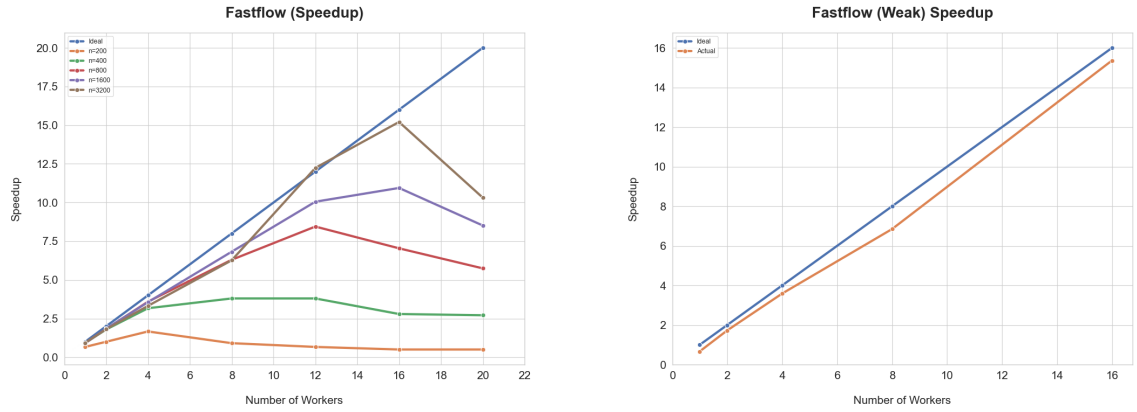
Table 3.3: MPI running times for different numbers of nodes. Each sub-table corresponds to a different number of nodes (1, 2, 4, 8).

3.2 Fastflow

The speedup is measured by taking into account the number of workers and excluding the emitter, since the latter is not enough alone for the execution of the program. We can interpret it as an overhead in the computation of the elements of the matrix. The linear speedup in both strong an weak analysis is the identity function, what we would like to achieve. The actual speedup in the strong analysis shows how for different matrix sizes we have a peak for a certain number of spawned workers, $n_w \in \{1, 2, 4, 8, 16\}$. Of course, smaller sizes imply a smaller number of threads and the speedup decreases afterwards (for instance in Figure 3.1a for $n = 3200$ the optimal number of threads is close to 16, but might be 17 or 18).

In the case of the weak analysis we set the matrix dimension as a linear function of the number of workers $n = 200n_w$ to get the plot in Figure 3.1b. It is very close to

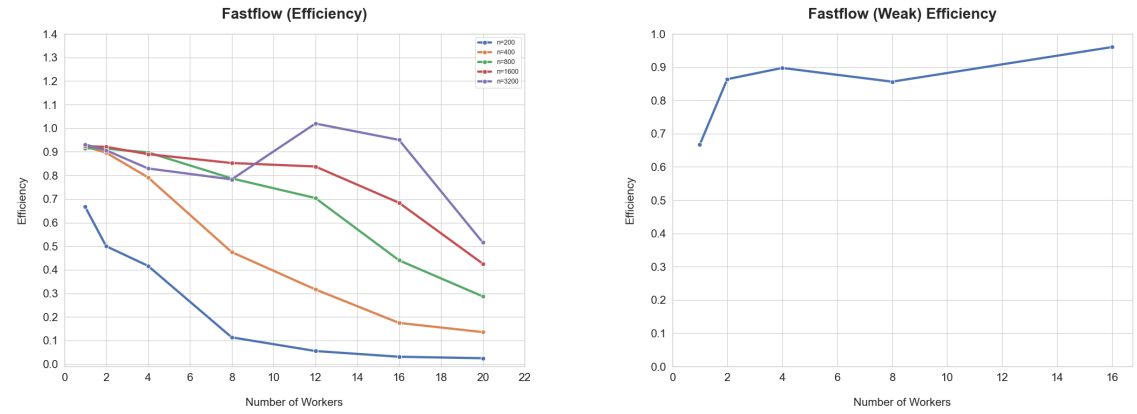
linear but still sublinear as expected.



(a) *Fastflow Strong Speedup* (b) *Fastflow Weak Speedup*

Figure 3.1: *Fastflow Speedup Analysis*

The strong efficiency is coherent with the speedup and trends down increasing the number of worker. However, for $n = 3200$ we have a different trend that rises up when having more than 8 workers. The efficiency overcomes 100%: this might be caused by ah higher number of cache hits when using more threads that improves the running time of the program. In the case of weak efficiency we have that higher number of threads we are taking advantage of the available resources more. The plot insights that 16 is very likely close to the optimal thread count and that the algorithm remains scalable if the matrix size is within that range, otherwise we might need more resources to handle more workers/threads.



(a) *Fastflow Strong Efficiency* (b) *Fastflow Weak Efficiency*

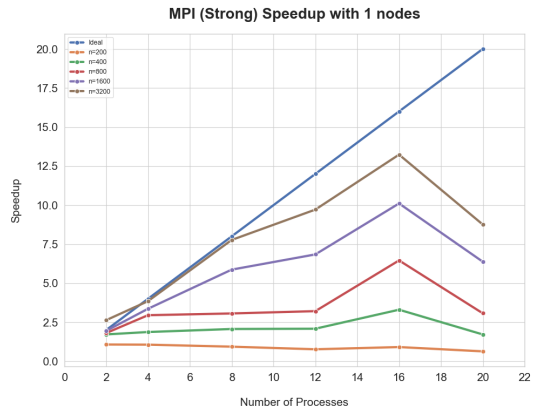
Figure 3.2: *Fastflow Efficiency Analysis*

The (strong and weak) scalability is not measured as not consistent with the design pattern. Since it is defined as $Sc = \frac{T_{par}(1)}{T_{par}(p)}$ we have that $T_{par}(1)$ would be actually measured with two threads (one worker): the emitter asynchronously sends tasks to the worker which would alter the execution and hence the running time.

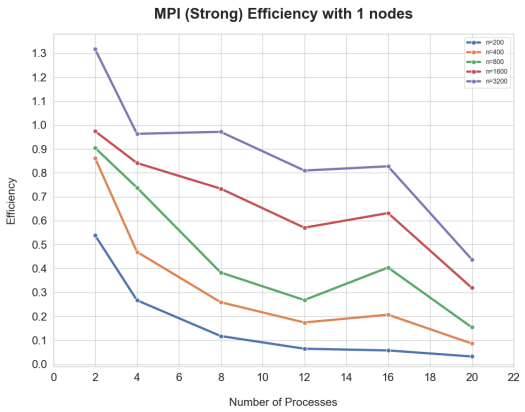
3.3 MPI

For design and usability choices, the minimum number of processes is set to two to allow message passing. Otherwise, it would be a sequential execution with some overhead caused by MPI utilities (sending messages to some other process is ambiguous if there is no other one). Hence, all analyses consider $T_{par}(2)$ as the least parallel version of the algorithm. The setting of the analysis is the same as in fastflow except for this detail and all tests have been executed with $\{1, 2, 4, 8\}$ nodes. For simplicity and report size constraints we report only the plots when the number of nodes is 1 and 4.

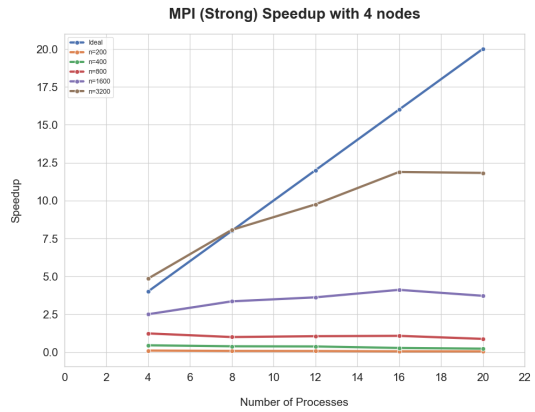
In Figure 3.3a and Figure 3.3c it is noticeable that the speedup is sublinear. When $p = 2$ the speedup is higher than the ideal for almost all n meaning that the parallel version is faster enough than the sequential version scaled by p. Both plots are similar and present as in fastflow a decreasing trend for $p \geq 16$. The efficiency is surprisingly high when the number of processes is low, meaning that very likely there are some cache hits. As overall the results for the strong analysis are consistent with fastflow even the latter is slightly better.



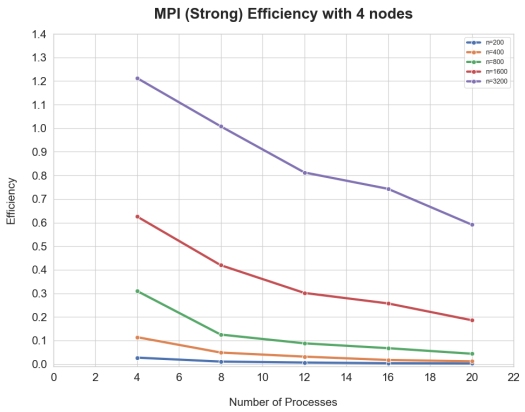
(a) *MPI Strong Speedup*



(b) *MPI Strong Efficiency*



(c) *MPI Strong Speedup*

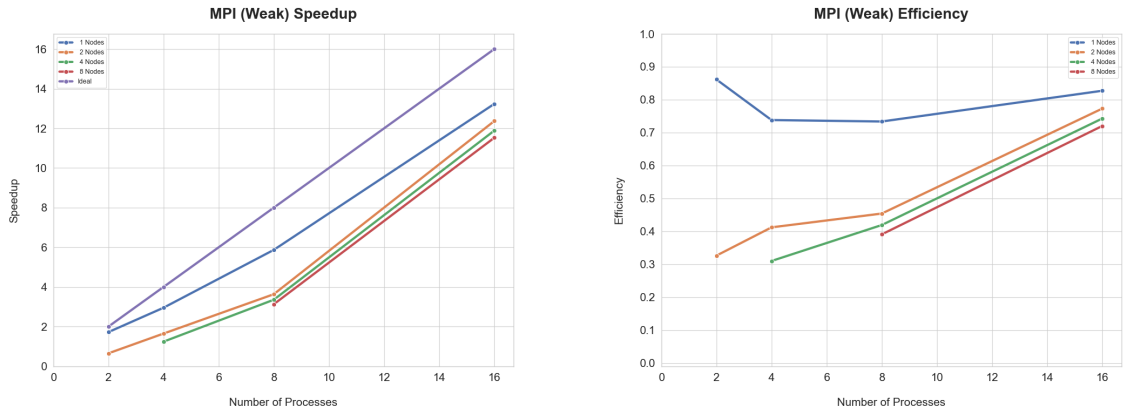


(d) *MPI Strong Efficiency*

Figure 3.3: *MPI Strong Scaling with 1 and 4 nodes*

As far as the weak analysis is concerned, we plotted curves for every number of nodes. The speedup is sublinear but shows an an almost linear trend in all curves. The

efficiency is highest when 1 node is utilized and can be caused by the null amount of communication time between machines for small matrices. Indeed, for bigger n we have that distributing the work across multiple nodes helps taking advantage of the resource properly knowing we have a balanced workload.



(a) *MPI Weak Speedup* (b) *MPI Weak Efficiency*

Figure 3.4: *MPI Weak Scaling Analysis*

The scalability tests are not shown because the plots are very similar to the speedup plots in both strong and weak analysis.

3.4 Final Considerations

The choice between MPI and fastflow is strictly related to the design pattern. In this project, considering the design choices and the problem itself, we can conclude that as comes out by the experiments fastflow is more adapt. We could see it by the metrics shown in the plots and from the tables in which the running time is clearly higher when choosing the ideal number of workers/processes. If one went beyond this ideal number, the communication overhead and resource management would worsen the performances of the programs. In a cluster and for very large tasks MPI would be the best choice, considering that it could be used exploiting threads as well.

Running Instructions

To run the code it is enough navigate to the project directory and run the following script:

Listing 4.1: compilation instructions

```
cmake -B build -S .
cd build
make
```

Finally, decide which executable to run following the syntax and put integers in arguments wrapped by square brackets:

- srun ./Sequential [n]
- srun –mpi=pmix –nodes=[nodes] -n [np] ./MPI_Collective [n]
- srun –mpi=pmix –nodes=[nodes] -n [np] ./MPI_P2P [n]
- srun ./Fastflow [n] [w]
- ./Correctness

Every C++ file has been compiled with the flag `-O3` to achieve the maximum optimization possible by the compiler. The `Correctness` executable serves as test to verify that the matrices in output (from previous runs) are correct and consistent with each other.

To run test and collect data it is enough to go in the `./scripts` directory and run a `.sh` file based on preferred algorithm.