Report SPM Final Project

Francesco Benocci 602495

agosto 2024

Contents

1 Introduction

The full project is available on github: github.com/Benocci/SPM/ExamProject.

The goal of the project is to parallelize an algorithm that compute an upper-triangular wavefront computation pattern on a NxN matrix. The matrix is initialized with the leading diagonal with the values $(m+1)/n$ with $m \in [0, n[$. Each element of an upper diagonal is the result of a dotproduct operation between the two vectors of size k composed by the elements on the same row m and the same column $m+k$. Specifically

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

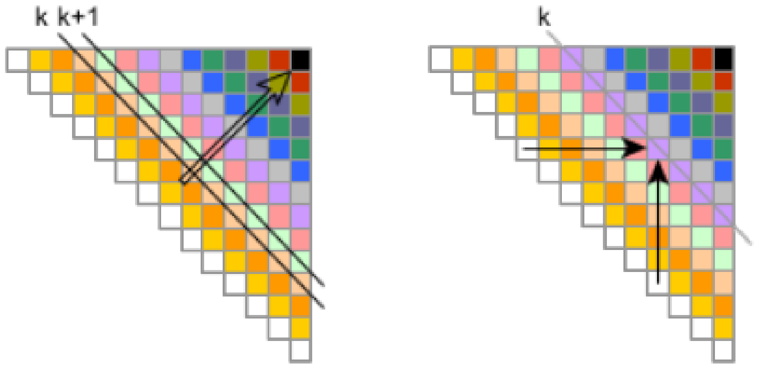

Figure 1: Graphical example of the computation.

The project is composed of three version of the code:

- **Sequential version**: inside the files SequentialUTW.cpp and SequentialUTWv2.cpp;
- **Parallel version using FastFlow**: inside the files FFUTW.cpp and FFUTWv2.cpp;
- **Distributed version using MPI**: inside the file MPIUTW_Bcast.cpp and MPIUTW_Allgather.

In the folder there is a Makefile that provide the way to compile and execute the code easily. To compile the sequential and the FF version it's possible to use the `make all` command, for the MPI version it's necessary to use `make mpi`. It is possible to make two type of execution: a single execution with `make run_single FILE=<executable_name.o> ARGS="<N> <num_thread(optional)>`

`<print>"` or a set of 10 execution that print the average of the time with `make run_average FILE=<executable_name.o> ARGS="<N> <num_thread(optional)> <print>"`. The "*print*" value enables printing the entire matrix if set to 1, only the last value if set to 2 and with any other value it prints nothing.

It is also possible to use the *.sh* files to compile and execute the code on the nodes with `sbatch <name_file.sh> <name_file_object> <N> <num_thread(optional)> <print>`.

2 Sequential version

The sequential code consists of three for loops, the first one representing the ascending diagonal, the second one making a loop on the elements of the diagonal and the inner one calculating the dot product with the vectors of the same row and column of the element.

```
    for (uint64_t k = 1; k < N; ++k) {
        for (uint64_t i = 0; i < N - k; ++i) {
            double dotProduct = 0.0;
            for (uint64_t j = 1; j < k + 1; ++j) {
                dotProduct += M[i][i + k - j] * M[i + j][i + k];
            }
            M[i][i + k] = cbrt(dotProduct);
        }
    }
```

Listing 1: Sequential Row-by-Column Implementation

With the previous code, it is clear that to perform the dot product, a row-by-column calculation will be necessary, which leads to a cache miss for each column access. So the first decision to speed up the code was therefore to eliminate this cache miss by introducing a dot product calculation in a row-by-row computation, taking advantage of the unused lower half of the matrix.

```
    for (uint64_t k = 1; k < N; ++k) {
        for (uint64_t i = 0; i < N - k; ++i) {
            double dotProduct = 0.0;
            for (uint64_t j = 1; j < k + 1; ++j) {
                dotProduct += M[i][i+k - j] * M[i+k][i+j];
            }
            M[i][i+k] = cbrt(dotProduct);
            M[i+k][i] = M[i][i+k];
        }
    }
```

Listing 2: Sequential Row-by-Row Implementation

As we can see, it is simply necessary to insert the calculated values into the lower part of the matrix so that we can use the rows to compute the dot product instead of the columns.

The tests conducted show a substantial reduction in time, even starting with small values of N. From now on, this type of calculation will also be used for parallel and distributed versions to perform the dot product.

	N=1024	N=2000	N=4000	N=10000
Sequential row-by-column	0,930957	9,79952	84,8628	1053,79
Sequential row-by-row	0,16829	1,21103	14,1695	215,12

Table 1: Time with and without transposed matrix

3 Parallel version for a single multi-core machine using FastFlow

3.1 Implementation

The idea for the parallel version was to leave the external for loop without parallelization, ensuring that the previous diagonal was fully computed before proceeding to the next one. The inner loop instead was parallelized with the high level data parallel pattern **ParallelFor**. In this way the elements of the diagonal are computed in parallel by the threads. The chosen configuration involves the use of spin-locks to minimize latency, along with a dynamic iteration assignment to balance the load.

```
    ParallelFor pf(numThreads, true, true);

    for (uint64_t k = 1; k < N; ++k) {
        pf.parallel_for(0, N-k, 1, [&, N, k](const int i) {
            double dotProduct = 0.0;
            for (uint64_t j = 1; j < k + 1; ++j) {
                dotProduct += M[i][i+k - j] * M[i+k][i+j];
            }
            M[i][i+k] = cbrt(dotProduct);
            M[i+k][i] = M[i][i+k];
        }, numThreads);
    }
```

Listing 3: Parallel implementation with FastFlow

3.2 Performance and consideration

For the time analysis, matrices of sizes 1024, 2000, 4000, and 10000 were considered. Specifically, the obtained speedups are shown in the following chart 2:

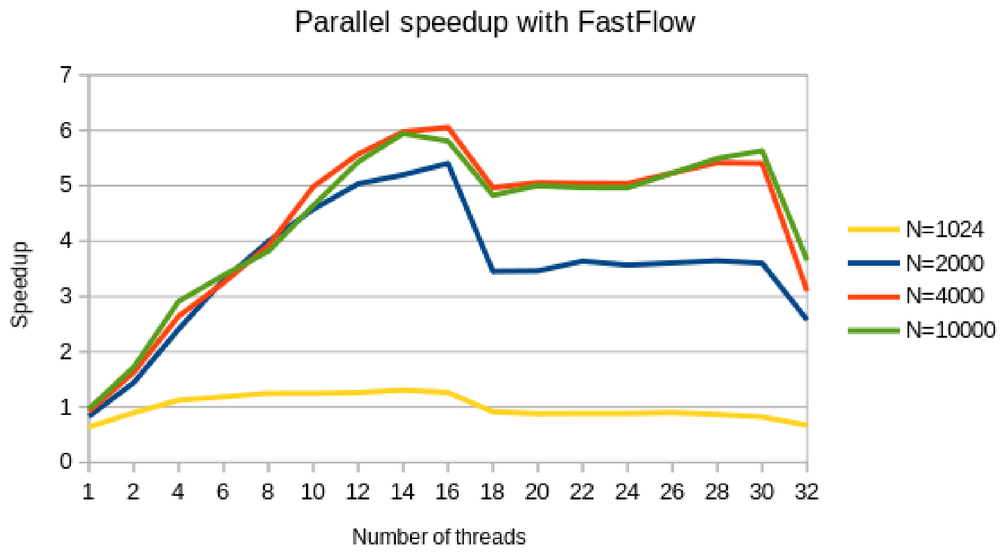

Figure 2: Parallel version strong scalability

As can be seen, with smaller matrices, the speedup tends to be lower due to the overhead of thread creation, which has a greater impact on shorter computations. As the problem size increases, this overhead becomes more amortized, and the speedup consequently increases. It is also important to note that after 16 threads, there is a decline in performance, likely due to the limitations of the nodes on which the code was tested.

4 Distributed version for a cluster of multi-core machines using MPI

4.1 Implementation

The first distributed version with MPI consists in calculating a portion of the current diagonal for each process based on its rank on its local matrix and then making an `MPI_Bcast` call to propagate the calculated values to the other processes. After some test it's clear that are not necessary to add any barrier for the conformation of the computation.

```
    for (uint64_t k = 1; k < N; ++k) {
        for (uint64_t i = rank; i < N - k; i += size) {
            double dotProduct = 0.0;
            for (uint64_t j = 1; j < k + 1; ++j) {
                dotProduct += M[i][i+k - j] * M[i+k][i+j];
            }
            M[i][i+k] = cbrt(dotProduct);
            M[i+k][i] = M[i][i+k];
        }

        for (uint64_t i = 0; i < N - k; ++i) {
            MPI_Bcast(&M[i][i+k], 1, MPI_DOUBLE,
                        i % size, MPI_COMM_WORLD);
            M[i+k][i] = M[i][i+k];
        }
    }
```

Listing 4: MPI version with MPI_Bcast

This solution was immediately deemed not good enough to be considered due to the high overhead introduced by each process. Its performance is discussed in more detail in the following subsection.
The second proposed solution involves the use of the `MPI_Allgather` and `MPI_Allgatherv` calls. The idea is similar to the previous one, where the diagonal is divided among the various processes based on their rank, but with the aim of reducing communication overhead. To achieve this, each process save the computed values in a vector (`values_to_send`) and the corresponding indices of those elements in another vector (`indices`).

```
    for (uint64_t k = 1; k < N; ++k) {
        vector<double> values_to_send;
        vector<int> indices;

        for (uint64_t i = rank; i < N - k; i += size) {
            double dotProduct = 0.0;
            for (uint64_t j = 1; j < k + 1; ++j) {
                dotProduct += M[i][i+k - j] * M[i+k][i+j];
            }
            double value = cbrt(dotProduct);
            M[i][i+k] = value;
            M[i+k][i] = value;

            // Store the calculated value and its index
            values_to_send.push_back(value);
            indices.push_back(i);
        }
```

Listing 5: MPI version with MPI_Allgather first loop

After that each process send the number of calculated elements and the two vectors in three separate calls.

```
        // Gather all the calculated values from all processes
        int total_values = values_to_send.size();
        vector<int> recv_counts(size);
        MPI_Allgather(&total_values, 1, MPI_INT, recv_counts.data()
            ,
                        1, MPI_INT, MPI_COMM_WORLD);

        vector<int> displs(size, 0);
        int total_elements = 0;
        for (int i = 0; i < size; ++i) {
            displs[i] = total_elements;
            total_elements += recv_counts[i];
        }

        vector<double> gathered_values(total_elements);
        vector<int> gathered_indices(total_elements);

        MPI_Allgatherv(values_to_send.data(), total_values,
            MPI_DOUBLE,
                       gathered_values.data(), recv_counts.data(),
                       displs.data(), MPI_DOUBLE, MPI_COMM_WORLD);

        MPI_Allgatherv(indices.data(), total_values, MPI_INT,
                       gathered_indices.data(), recv_counts.data(),
                       displs.data(), MPI_INT, MPI_COMM_WORLD);
```

Listing 6: MPI version with MPI_Allgather comunication part

In this way, all other processes can receive the values and place them in the correct cells of the matrix.

```
        for (int i = 0; i < total_elements; ++i) {
            M[gathered_indices[i]][gathered_indices[i] + k] =
                gathered_values[i];
            M[gathered_indices[i] + k][gathered_indices[i]] =
                gathered_values[i];
        }
    }
```

Listing 7: MPI version with MPI_Allgather update of the matrix

Thus, instead of an `MPI_Bcast` call for each generated element, there will be three `MPI_Allgather` calls per process. This approach, especially with large matrices, results in significantly reduced times and therefore higher speedups.

4.2 Performance and consideration

Below, we present the speedups obtained from both the `MPI_Bcast` version and the `MPI_Allgather` version with N equal to 4000 and 10000.

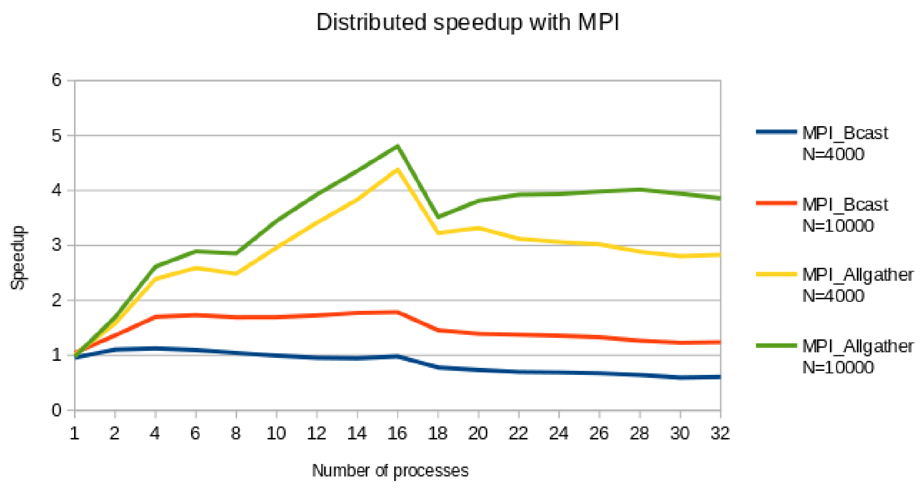

Figure 3: Distributed version strong scalability

It's possible to notice the significant difference in speedup between the two versions, and it can also be observed that increasing the matrix size leads to a greater speedup. In this version as well, it is possible to observe a drop in performance after 16 processes, followed by a gradual recovery. This occurs regardless of the number of nodes used.

In the following there is an analysis of the different number of nodes used for the computation:

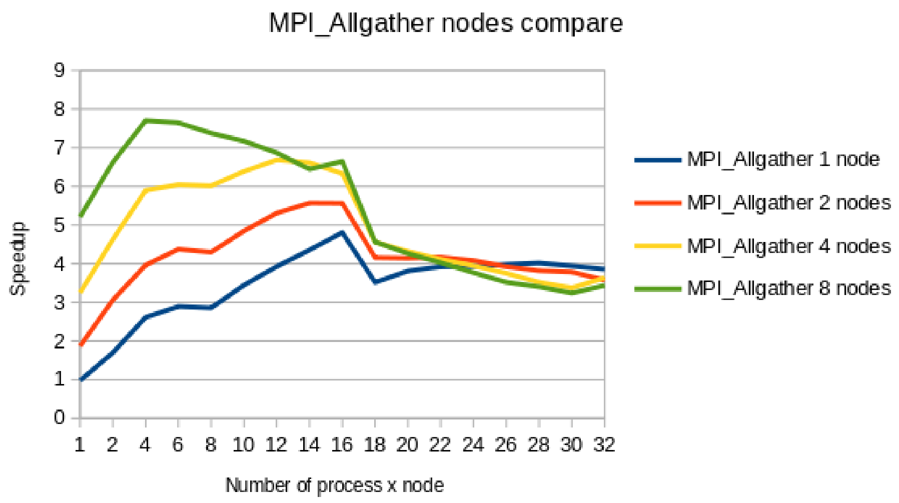

Figure 4: Compare MPI version with different number of nodes

By increasing the number of nodes while keeping the same number of processors per node (chart 4), an initial increase in speedup can be observed, followed by a decline as the number of processes increases. This demonstrates that, with a large number of processes communicating with each other, performance tends to deteriorate.

5 Version comparison and consideration

5.1 Comparison

The tests were run on the nodes of spmcluster.unipi.it using SLURM scripts. Each result is based on the average of ten executions.

With the following graph (chart 5), we compare the times of the three main versions selected: the sequential version that utilizes caching, the parallel version with FastFlow, also with caching and 16 threads, and the MPI version using `MPI_Allgather` with 16 processes in each of the 8 node (128 processes overall).

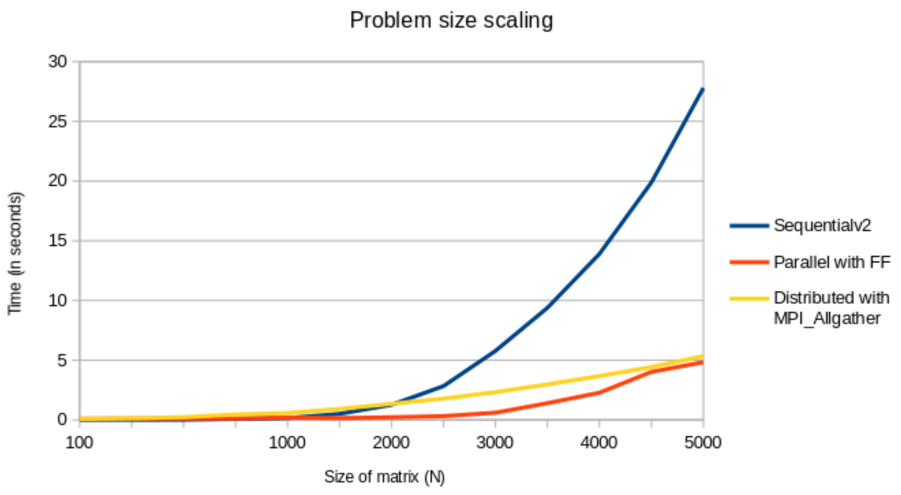

Figure 5: Times with increasing size of N

As can be seen, the parallel and distributed versions are both much faster as the matrix size increases. Specifically, for values up to N=5000, the FastFlow version still has slightly better performance. However, as the matrix size continues to increase, the distributed version's performance surpasses that of the parallel version.

5.2 Weak scalability

Below, we analyze the weak scalability, which involves measuring the times while doubling the matrix size (N) and the number of threads/processes each time.

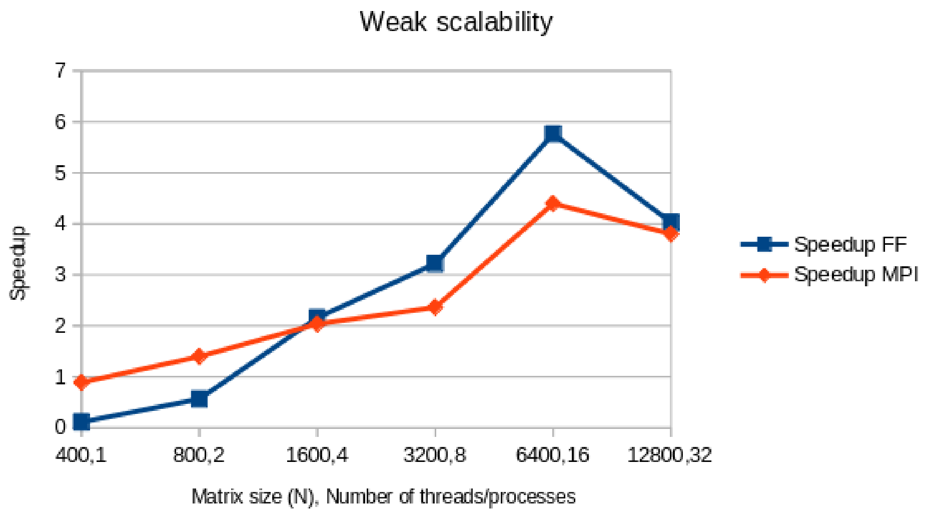

Figure 6: Weak scalability

As mentioned earlier, the highest speedup is achieved at 16 threads/processes with the MPI version that remains slower due to the few processes used (on a single node) but proves to be faster with large matrices. It is important to note that it is also very fast with very small matrices despite the limited number of processes involved. This is likely due to its ability to fully utilize parallel computation, as the vectors containing the calculated elements are short, making the communications quick.

5.3 Efficiency

Now we analyze how effectively multiple processors are utilized compared to a single processor, the efficiency it's defined as the ratio of speedup to the number of processors used.

Efficiency in parallel computing represents the system's ability to effectively utilize available resources during program execution. It measures how closely the system approaches ideal performance, highlighting how well it scales as the number of processors increases. Below, we present the efficiency data obtained from our executions.

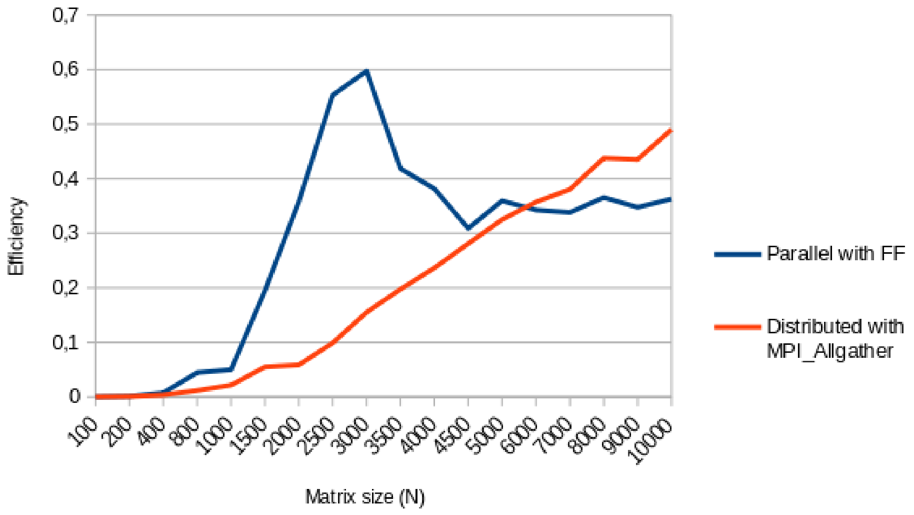

Figure 7: Efficiency

As can be seen, the two versions exhibit very different behaviors: the parallel version with FastFlow shows a peak in efficiency with medium-sized matrices and then stabilizes. In contrast, the distributed version with MPI displays increasing efficiency as the matrix size grows. For both versions, the same configurations as in Figure 5 were considered.

6 Final conclusion

In this project, we explored different strategies to parallelize an upper-triangular wavefront computation on an NN matrix. By implementing and comparing sequential, parallel, and distributed versions of the algorithm, we observed significant performance improvements with the parallelization techniques.

The parallel version using FastFlow demonstrated considerable speedup, especially with larger matrices, although performance plateaued as the number of threads increased due to system limitations. The distributed version using MPI further pushed the boundaries of scalability, particularly with the optimized use of `MPI_Allgather`, which reduced communication overhead and improved efficiency on large-scale matrices.

Our analysis showed that when the matrix size grows, the distributed version outperforms the parallel version. Efficiency analysis further highlighted that while the FastFlow version stabilized at medium-sized matrices, the MPI version continued to scale effectively with larger matrices.

In conclusion, this project demonstrated the importance of selecting the right parallelization strategy based on the problem size and the available computational resources.