SPM Final Project

Distributed Wavefront computation

Veronica Pistolesi - MAT. 563663 - v.pistolesi6@studenti.unipi.it
M.Sc. Computer Science, Artificial Intelligence curriculum
Parallel and Distributed Systems: Paradigms and Models [305AA]
AY 2023/2024

UNIVERSITÀ DI PISA

Exam Session: September 6, 2024

1 Problem description

The objective of the project is to parallelize an algorithm that performs a wavefront computation (Fig.1) on the upper triangular part of an $N \times N$ matrix. The matrix is initialized with values on the main diagonal, where each element $e_{m,m}$ is assigned $(m+1)/n$, with $m \in [0, N)$.
For each diagonal element of the matrix M, the $n-k$ diagonal elements $e^k_{m,m+k}$ are the result of a dotproduct operation 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:

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

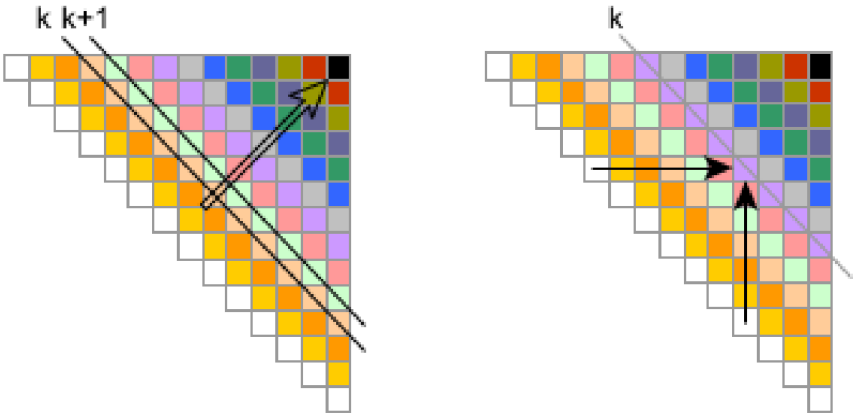

Figure 1: Graphical representation of the wavefront computation.

2 Project outline

The project is available at `https://github.com/VeronicaPistolesi/Wavefront`.
The GitHub repository contains the following implementations:

- **Sequential approach**: `sequential.cpp`, `sequential_t.cpp`.
- **Parallel approach with FastFlow**: `fastflow_pfor.cpp`, `fastflow_pfor_t.cpp`, `fastflow_pforgrain.cpp`, `fastflow_pforgrain_t.cpp`
- **Distributed approach with MPI and OpenMP**: `mpi.cpp`, `mpi_t.cpp`.

A Makefile is provided for easy compilation of all files by running `make`. In addition, a number of **SLURM** and **Python** scripts are provided for batch execution.

3 Sequential approach

Solving the problem using the sequential method involves scanning the diagonals in an ordered sequence, starting with the largest diagonal, and calculating the value of each element of these diagonals. In this way, the dependency constraints between the various elements are respected.

```
void wavefront(std::vector<double> &M, const uint64_t &N) {
    for(uint64_t k=1; k<N; k++){
        for(uint64_t m=0; m<N-k; m++){
            compute_element(M, m, k, N);
        }
    }
}
```

Listing 1: Sequential approach.

There are two strategies for computing the scalar product: naive and cache-optimized.

3.1 Naive strategy

In our problem, the computation of the scalar product involves multiplying a row vector with a column vector. In the simplest strategy, the vectors are kept in their original form and consequently read as such.

```
inline void compute_element(std::vector<double> &M, const uint64_t &m, const uint64_t &k
    , const uint64_t &N) {
    double c = 0;
    for(uint64_t i=0; i<=k; i++){
        c += M[index(m, m+i, N)]*M[index(m+k-i, m+k, N)];
    }
    c = std::cbrt(c);
    M[index(m, m+k, N)] = c;
}
```

Listing 2: Naive strategy.

3.2 Cache-optimized strategy

The aforementioned strategy can be improved by changing the memory access of all column vectors, reading them as row vectors. In fact, on processors, loading the main memory in cache per row is much more efficient. When the matrix is mirrored along its secondary diagonal, it presents the original column vectors as row vectors. This is why the cache-optimised strategy presents two matrices (the original and the mirrored). Furthermore, the row-by-row product is automatically optimised by the compiler using Single Instruction Multiple Data (SIMD) instructions.

```
inline void compute_element_transpose(std::vector<double> &M1, std::vector<double> &M2,
    const uint64_t &m, const uint64_t &k, const uint64_t &N) {
    double c = 0;
    for(uint64_t i=0; i<=k; i++){
        c += M1[index(m, m+i, N)]*M2[index(N-(m+k)-1, N-(m+k-i)-1, N)];
    }
    c = std::cbrt(c);
    M1[index(m, m+k, N)] = c;
    M2[index(N-(m+k)-1, N-m-1, N)] = c;
}
```

Listing 3: Cache-optimized strategy.

The results show that this second strategy significantly speeds up the computation compared to the naive one.

	$N = 10$	$N = 100$	$N = 1000$	$N = 2000$	$N = 3000$	$N = 5000$	$N = 10000$
`sequential`	11	425	262317	2040216	9198877	61571471	717595554
`sequential_t`	11	427	229347	1760697	6393416	29364946	226871577

Table 1: Exectution times (μs)

4 Parallel approach with FastFlow

FastFlow is a programming library which offers both a set of high-level ready-to-use parallel patterns and a set of mechanisms and composable components, to aid and simplify the development of parallel applications.

4.1 Parallel For with static partitioning

This type of solution parallelizes the computation of individual elements of the same diagonal. Specifically, the work is equally divided into `num_workers` partitions statically assigned to individual workers. Parallelization is possible because the computation of elements belonging to the same diagonal is independent.

```
void wavefront(std::vector<double> &M, uint64_t N, uint64_t num_workers) {
    ff::ParallelFor pf;
    for (uint64_t k = 1; k < N; k++) {
        pf.parallel_for(0, N-k, 1, [&](uint64_t m) {
            compute_element(M, m, k, N);
        }, num_workers);
    }
}
```

Listing 4: Parallel For with static partitioning.

4.2 Parallel For with dynamic partitioning

Similar to the previous solution, the work is divided equally into `num_workers*chunks_per_worker` partitions, this time dynamically assigned to individual workers: i.e., individual workers operate on one chunk of smaller size at a time and once the job is finished they move on to the remaining chunks. This approach reduces the idle time when some workers terminate their job before others do.

```
void wavefront(std::vector<double> &M, uint64_t N, uint64_t num_workers, uint64_t
    chunk_per_worker) {
    ff::ParallelFor pf;
    for (uint64_t k = 1; k < N; k++) {
        pf.parallel_for(0, N-k, 1, (N-k)/(num_workers*chunk_per_worker),
        [&](uint64_t m) {
            compute_element(M, m, k, N);
        }, num_workers);
    }
}
```

Listing 5: Parallel For with dynamic partitioning.

5 Distributed approach with MPI and OpenMP

The Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures. In particular, to distribute

computations over several machines. Open Multi-Processing (OpenMP) is an application programming interface (API) which similarly to FastFlow provides high level patterns to describe parallelized computations.
An application built with the hybrid model of parallel programming can run on a computer cluster using both MPI and OpenMP, such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes.

5.1 Distributed For

The idea behind this approach is similar to the one in Parallel For (Sec. 4.1): each diagonal is divided equally into `num_workers` chuncks computed independently on each node. Since memory is not shared among nodes, MPI allows the computed values to be exchanged so that each node has the full version of the matrix. In this case, this function is performed by `MPI_Allgather`.

```
void wavefront(std::vector<double> &M, const uint64_t &N, int id_worker, int n_worker) {
    for(uint64_t k = 1; k < N; k++) {
        uint64_t chunk_size = std::ceil(double(N-k)/n_worker);
        std::vector<double> diag_sync = std::vector(chunk_size * n_worker, 0.0);
        std::vector<double> diag_worker;
        diag_worker.reserve(chunk_size);
        for(uint64_t m = chunk_size*id_worker; m < std::min(chunk_size*(id_worker+1), N-k); m++) {
            diag_worker.push_back( compute_element_diag(M, m, k, N) );
        }

        MPI_Allgather(diag_worker.data(), chunk_size, MPI_DOUBLE, diag_sync.data(), chunk_size, MPI_DOUBLE, MPI_COMM_WORLD);
        for(uint64_t m=0; m<N-k; m++){
            M[index(m, m+k, N)] = diag_sync[m];
        }
    }
}
```

Listing 6: MPI implementation.

5.2 Parallel Dot Product

To fully utilize the resources of each node, the distributed MPI implementation is further experimented with by parallelizing the computation of the scalar product of each individual element using OpenMP's parallel for. Therefore, the distributed computation can be further scaled using the hybrid approach.

6 Experimental results

Figures 2 and 3 show the computation times of parallelized approaches with FastFlow, using the naive and cache-optimized strategies, respectively, compared with the ideal perfect parallelization time. It is possible to see that computation times decrease as the number of threads increases, reaching a plateau on the 16 units of parallel computation. The difficulty in achieving optimal parallelization is inherent in the problem being addressed, which does not allow parallelization of the computation of two successive diagonals.

For time and speedup analysis, tests were performed by varying the the input size of the matrix. Figure 4 shows the results. The observed speedup is less than ideal due to the underlying nature of the computation and the dependencies between elements. Notably,

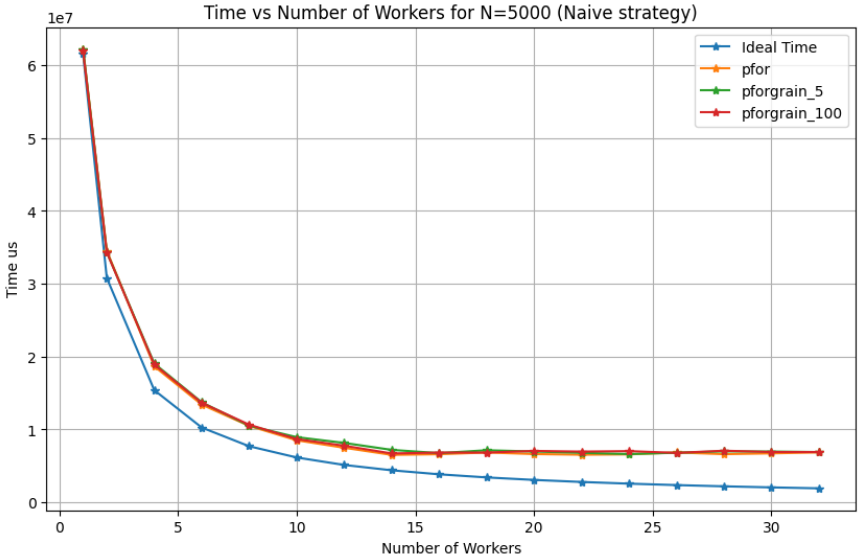

Figure 2: Time with respect to number of workers (Naive strategy).

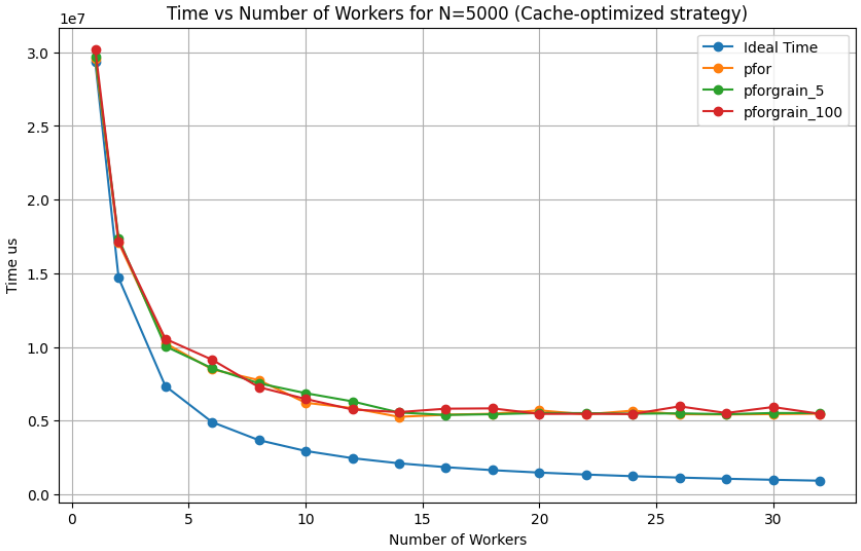

Figure 3: Time with respect to number of workers (Cache-optimized strategy).

the speedup is increasing up to 16 threads, while it remains constant beyond that. The results fit with the hardware on which the tests were run, which has no more than 16 units of parallel computation on a single node.

Figure 5 shows the speedup executing the distributed approach with MPI by varying the number of nodes over which the computation is distributed while keeping the matrix size fixed ($N = 5000$ in this case). It is evident that as the number of nodes rises, the speedup keeps getting higher. Distributed computation is unfortunately quite slower than sequential computation. This result is due to the necessary synchronization, after the computation of each diagonal, imposed by the distribution of work over multiple nodes, which cannot benefit from having the main memory shared as in the case of the parallelized approach.

Efficiency is defined as the ratio of speedup to the number of parallel computing units used, describing the effectiveness of using more resources for the same computation.

With 32 threads, Figure 6 illustrates the effectiveness of parallel computing as the size of the task changes. We can observe that the method performs better on average-sized

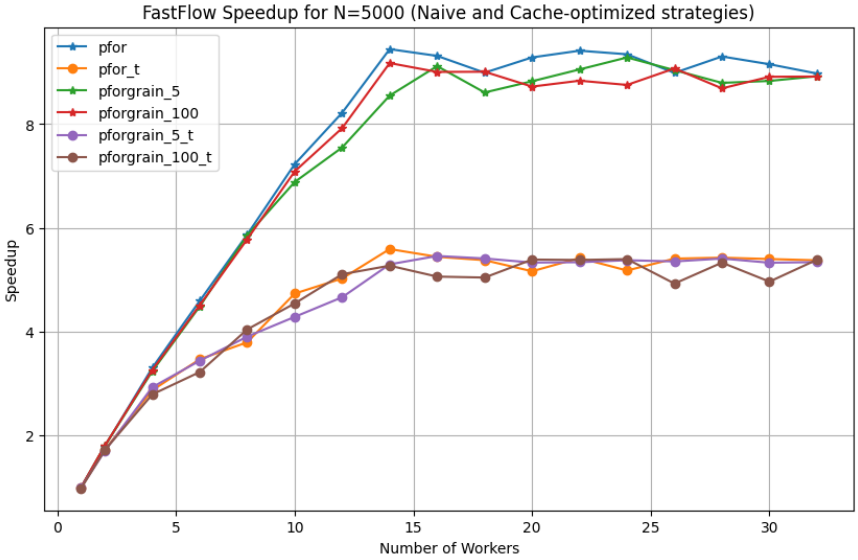

Figure 4: FastFlow Speedup.

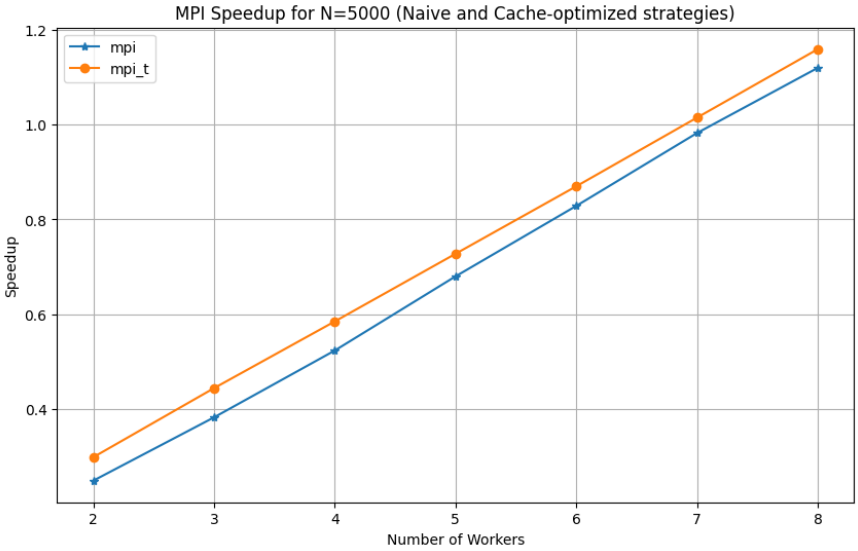

Figure 5: MPI Speedup.

matrices, becoming less efficient when the matrix grows too big (most likely due to the fact that the number of components contributing to each scalar product tends to grow rapidly).

Figure 7 shows the efficiency of distributed computation using MPI. In contrast, the distributed approach, as opposed to the parallel approach, improves almost linearly as the matrix size increases.

7 Conclusions

The project demonstrated the effectiveness of parallel and distributed computing techniques in optimizing wavefront computations. The sequential implementation served as a baseline for performance comparisons, revealing the significant computational cost of the problem when handled without parallelization.

The parallel approaches, both using FastFlow with static and dynamic partitioning, successfully reduced execution time by distributing the workload across multiple threads.

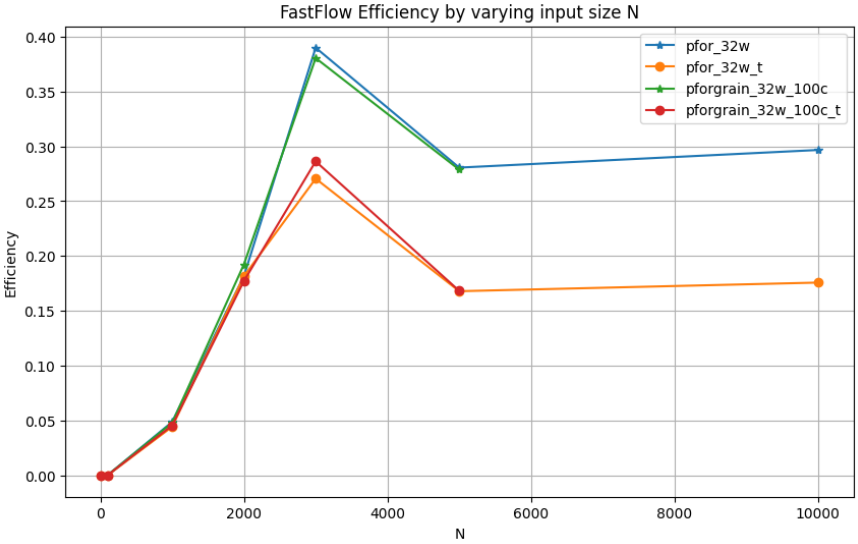

Figure 6: FastFlow Efficiency.

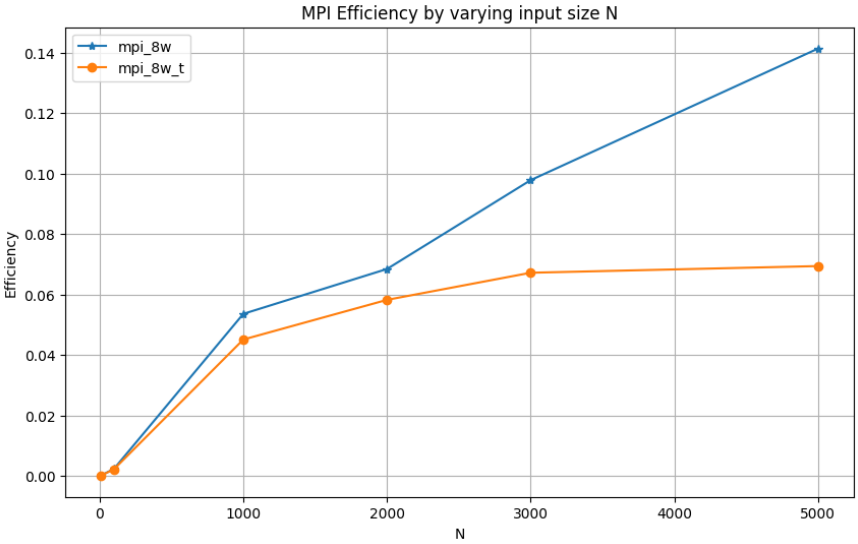

Figure 7: MPI Efficiency.

However, due to the inherent dependencies between matrix elements along successive diagonals, achieving ideal parallelization was not feasible. This limitation caused the speedup to plateau at 16 threads, which matched the number of hardware parallel computation units available on the test platform. Additionally, the cache-optimized strategy showed significant improvements over the naive approach, particularly for larger matrices.

On the distributed side, the MPI implementation showed that distributing the computation across multiple nodes could effectively further reduce execution time, although at a higher communication cost. The hybrid approach with MPI and OpenMP further improved performance by leveraging both node-level and core-level parallelism. Despite the higher communication overhead, especially with frequent synchronizations, the distributed solution proved scalable as the number of nodes increased.

Overall, while parallelization provided significant speedup, the performance gains are constrained by the problem's inherent computational dependencies. In contrast, the distributed approach offers more scalability but suffers from communication overhead.