Università di Pisa

Parallel and Distributed Systems: Paradigms and Models 2023/2024 Project report

Distributed Wavefront Computation

Salvatore Salerno

May 2024

Introduction

The goal of this project is to optimize an algorithm by parallelization, utilizing the `C++` programming language in conjunction with various APIs and libraries, leading to distinct solutions, each with their advantages and disadvantages.

The algorithm computes a square matrix by following a Wavefront pattern across its upper diagonal. The process is as follows: given an upper-triangular square matrix `mtx` of size $n \times n$, only the elements on the main diagonal are known initially:

$$\forall i \in [0, n-1].mtx[i][i] = (i+1)/n \tag{1}$$

For the each of the upper diagonals, the elements are computed as:

$$\forall i \in [0, n-1].\forall j \in [i+1, n-1].mtx[i][j] = \sqrt[3]{\sum_{k=0}^{j-1} mtx[i][k] * mtx[j][k+1]} \tag{2}$$

The element is obtained as the dot product between the row vector at index i and the column vector at index j, containing the elements prior to the one being computed. This means that to compute an element on the z^{th} diagonal, the elements of the previous diagonal must already be completed. In contrast, elements within the same diagonal can be computed independently of each other.

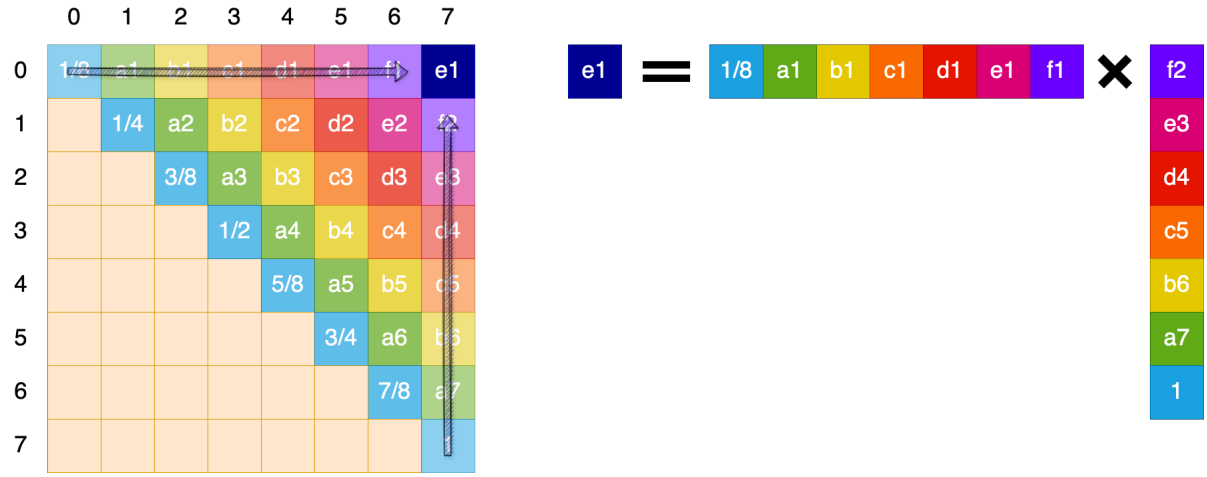

Figure 1: Example of a 8x8 Matrix generated using the Wavefront Computation

Sequential Implementation

We first develop a sequential solution to the Wavefront problem in order to identify initial optimizations and find potential parallelization opportunities. The simplest approach is to compute the diagonals sequentially, starting from the first after the major diagonal and ending with the last. Each diagonal has one less element than the previous, and computation on the next diagonal begins only after completing the current one. A basic pseudocode is shown in Algorithm 1.

The SquareMatrix class

In all implementations, the square matrix is represented by the custom `SquareMatrix` class. The matrix data is stored internally in a `std::vector<double>` of size $n \times n$, where n is the number of rows and columns. By using a single vector instead of a `std::vector<std::vector<double>>`, we optimize both storage and efficiency.

Algorithm 1 Basic Wavefront Sequential Pattern

```
for diag = 1 to matrix_length - 1 do
  for elem = 1 to diag_length do
    Compute(elem)
  end for
end for
```

The class provides methods to store and retrieve elements using row and column indices, hiding the underlying implementation to the user. Given a matrix `mtx` of type `SquareMatrix`, then an element is indexed as follows:

$$\forall i \in [0, n-1].\forall j \in [0, n-1].mtx[i][j] = data[(i * n) + j] \tag{3}$$

By offering methods that avoid the need to explicitly write the above formula we prevent potential bugs caused by typos when applying it multiple times in the project.

Additionally, the class features the method `InitializeMatrix` which sets the values of the major diagonal according to the project's requirements.

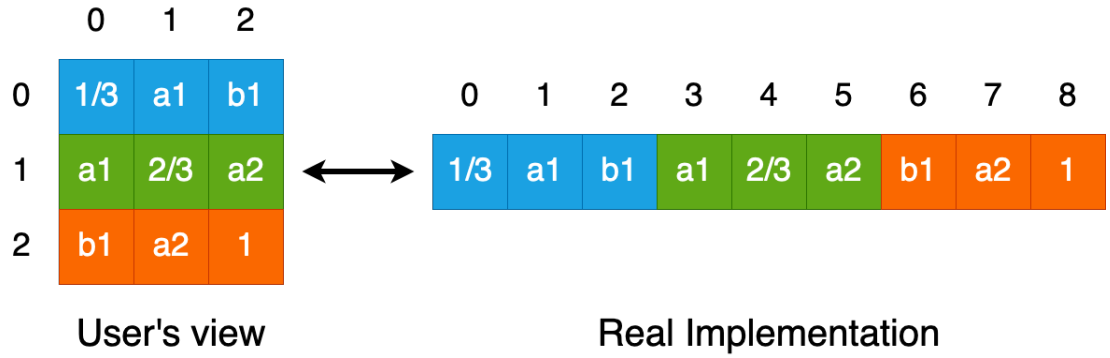

Figure 2: How the user sees a square matrix and its real implementation

Cache Optimization

When computing an element of an upper diagonal, we need to calculate the dot product of its associated row and column vectors. Since the matrix is stored in a single `std::vector<double>`, elements are stored contiguously by rows. As a result, when accessing a row vector, the processor can take advantage of the locality of reference principle, storing adjacent elements in cache and optimizing retrieval during the dot product computation.

To achieve similar efficiency for column vectors, we can utilize the unused lower-triangular part of the matrix. Since all cells below the major diagonal are empty and unused in the Wavefront pattern, we can copy the values of the upper-triangular columns into the transposed rows of the lower-triangular part, such that:

$$\forall i \in [0, n-1].\forall j \in [0, n-1].mtx[i][j] = mtx[j][i] \tag{4}$$

By retrieving elements of the column vector in their corresponding lower-triangular row, we both utilize wasted space in the matrix and let the processor benefit from the locality of reference for both vectors, making the dot product more efficient.

To facilitate this process, the `SquareMatrix` class provides the method `SetValue(row, col, val)`, which stores the value in both the original and transposed positions within the matrix.

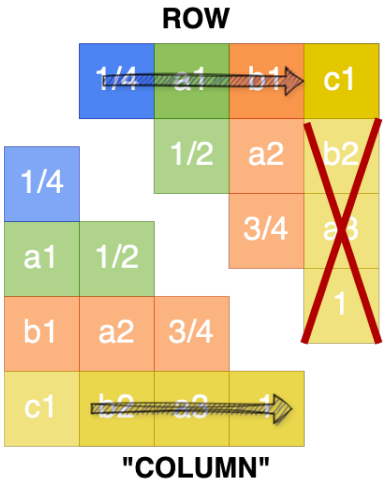

Figure 3: Example of a dot product computation for the element `c1` in a 4x4 Matrix

Asymptotic Complexity

For an $n \times n$ matrix, the asymptotic complexity of a Wavefront computation is determined by the processing of all upper diagonals. The i^{th} diagonal has a length of $n - i$, which is equal to the number of elements to compute. Each element of a diagonal requires a dot product between vectors of length $n - i - 1$ and the application of the cube root to the result. Assuming both the cost of multiplying two elements and applying the cube root is $\mathcal{O}(1)$, then:

- **Complexity of computing an element of a diagonal =** $k \times \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(k)$ where k is the length of the vectors.
- **Complexity of computing all elements of one diagonal =** $z \times \mathcal{O}(z-1) = \mathcal{O}(z^2)$ where z is the length of the diagonal.
- **Overall complexity =** $n-1 \times \mathcal{O}((n-1)^2) = \mathcal{O}(n^3)$ where n is the length of the matrix.

Measurements

A C++ implementation of the sequential approach stored in *"src/sequential.cpp"* has been developed to study its time complexity compared to the more efficient parallel algorithms introduced in the following sections. All project code was tested on the *"spmcluster.unipi.it"* compute cluster at the University of Pisa, using various matrix sizes. Each test was run five times per matrix size, and the tables present the average results for each size.

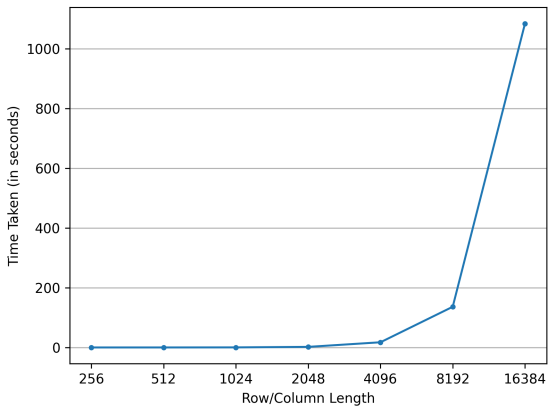

Figure 4: Time measurements in seconds of the sequential C++ version of the algorithm.

FastFlow Implementation

The first parallel implementation uses the `FastFlow C++` library, developed by the *Universities of Pisa* and *Turin*. The library is designed for multi/many-core CPUs and distributed systems. Each application is interpreted as a directed graph, where each node represents a concurrent processing unit in a shared-memory environment. The library provides basic building blocks and high-level patterns for common parallel computation schemes. For this project, we used the `Farm` parallel building block.

This version of the algorithm improves on the sequential approach by parallelizing the computation of each diagonal. Since the elements within a diagonal rely on different row and column vectors, they can be computed independently. This enables efficient use of shared memory, as threads can safely access the same matrix without race conditions. Each thread is assigned an equal (or nearly equal) portion of elements to compute. After all elements of a diagonal are processed, the algorithm advances to the next one. Threads must wait for the completion of the current diagonal before proceeding to the next. A basic pseudocode of the computation of each thread is shown in Algorithm 2.

Algorithm 2 Wavefront Pattern for a single thread of the Farm

```
for diag = 1 to matrix_length - 1 do
    chunk_size = ⌈diag.length/num_threads⌉
    Compute(chunk_size elements)
    Waits for other threads to finish their part of the diagonal
end for
```

The Farm Building Block

The key challenge in implementing Algorithm 2 is ensuring that threads know when to start processing their part of elements of the next diagonal. To ensure all threads have completed their current task before moving forward, this project employs the Farm high-level pattern. The Farm pattern consists of a pool of `Workers` coordinated by an `Emitter`. The `Emitter` signals the `Workers` to begin computation on the next diagonal by sending a token, which identifies the specific chunk of elements they have to process. Each `Worker` uses the token to determine its assigned elements, processes them, and then sends the token back to the `Emitter` via a feedback channel. Once the `Emitter` receives all the tokens back, it redistributes them to the `Workers` to initiate the next diagonal's computation. Notably, the *Emitter* also acts as a `Worker`, processing its chunk of elements after dispatching the tasks and before receiving them back.

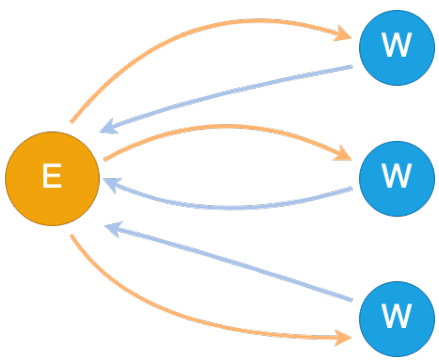

Figure 5: Graphical representation of the Farm employed

Static Chunk Size vs Dynamic Chunk Size

For each diagonal, the length is not always a multiple of the number of nodes available. In such cases, we round up the division to the nearest integer using the standard C++ function `std::ceil`. However, this can result in the last Worker receiving fewer elements than the others, or in the worst case, some Workers might receive no elements at all and not contribute to the computation.

A possible workaround is to determine the chunk size dynamically, recalculating it each time a task is sent. The dynamic chunk size is computed as:

$$dynamic\_chunk\_size = \lceil remaining\_elements\_to\_send / remaining\_workers\_with\_no\_task \rceil$$

Although this approach requires slightly more computation compared to static case, it ensures a more even distribution of elements. However, if the number of elements in a diagonal is smaller than the number of nodes, some nodes will still remain inactive.

Theoretically the dynamic approach seems to be better than the static method, however when comparing the two in our measurements, as Figure 18 shows, using a dynamic chunk size proves slower. This slowdown occurs probably because the overhead of managing communication between all `Workers` outweighed the marginal decrease in elements per `Worker`. Therefore, the algorithm defaults to using static chunk sizing, with the dynamic approach available as an optional alternative through the use of the guard `DYNAMIC_CHUNK`.

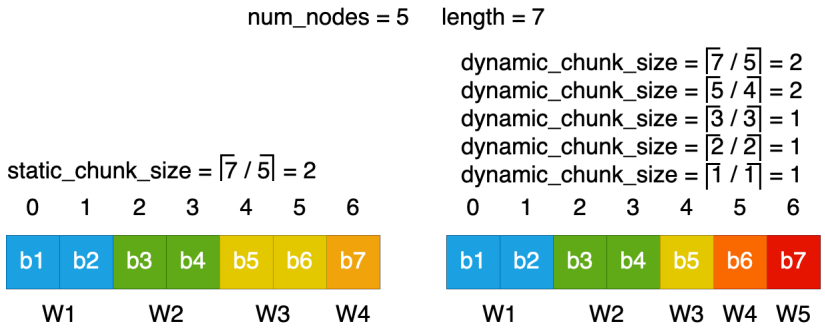

Figure 6: Example of how using a dynamic chunk size compared to a static one leads to a more equal distribution among Workers

The DiagInfo class

Each node needs to track the current diagonal it is processing and its length to determine which elements to compute. Instead of requiring each node to update a local copy of these values with each new task, the information is shared in the object of custom type `DiagInfo`. Only the `Emitter` has the responsibility to update before sending the task the object through the `PrepareNextDiagonal()` method and by filling the dynamic chunk size array. The `Workers` simply read from it during their computations.

Farm Limitation

Another potential parallel optimization lies in the dot product computation. Instead of calculating it element by element, the row and column vectors could be divided into segments, with each segment handled by a different thread. The partial results would then be summed to produce the final dot product. While this idea is appealing, it is impractical in our current approach. As the computation progresses, the diagonals become smaller, eventually having fewer elements than available `Workers`. For example, the last diagonal has only one element, leaving

most Workers idle. Unfortunately, in FastFlow's Farm building block, Workers cannot directly communicate, so coordinating this task would require extensive communication managed by the Emitter, leading to inefficiencies. Another option could involve switching to a sequential approach when we encounter the first diagonal that has fewer elements than the number of workers, using a Parallel-For construct to distribute the workload for the dot products. However, the overhead of closing the Farm and initializing a new parallel construct would likely outweigh any potential benefits, making this solution inefficient.

Asymptotic Complexity

Compared to the sequential approach, the asymptotic complexity of a Wavefront computation using FastFlow is more efficient for computing a single diagonal, as all elements can be processed simultaneously in a single step. Assume the cost of communication between the Emitter and one Worker is a constant $\mathcal{O}(c)$ with $c > 0$, the complexity simplifies to:

- **Complexity of sending tasks to each Worker** $= num\_workers \times \mathcal{O}(num\_workers \times c)$
- **Complexity of computing all elements of one diagonal** $= \mathcal{O}(num\_workers \times c) + \mathcal{O}(\max chunk\_size) = \mathcal{O}(num\_workers \times c) + \mathcal{O}(\lceil n - 1/num\_threads \rceil)$ where n is the length of the matrix.
- **Overall complexity** $= \; n - 1 \times (\mathcal{O}(num\_workers \times c) + \mathcal{O}(\lceil n - 1/num\_threads \rceil)) = \mathcal{O}((n-1) \times num\_workers \times c) + \mathcal{O}((n-1)^2/num\_threads)$ where n is the length of the matrix.

Measurements

A C++ implementation of the FastFlow approach is implemented in *src_parallel_fastflow.cpp*. The program takes the matrix size and the number of threads as arguments. By passing m and n, the process creates one Emitter and $n - 1$ Workers to handle a matrix of size m. When compiling the project using the CMake configuration files, it will generate two executables: one using static chunk sizes and the other using dynamic chunk sizes. Both versions have been tested on the *"spmcluster.unipi.it"* compute cluster with various thread counts and matrix sizes.

The speedup graph in Figure 9 demonstrates that using more threads results in greater performance gains. However, the efficiency graph in Figure 10 reveals that as the number of threads increases, efficiency diminishes, indicating that the overhead associated with managing additional threads becomes more and more large. This suggests to take into account the computational cost of coordinating a larger number of threads when computing the Wavefront pattern in a shared memory setting.

MPI Implementation

The second parallel implementation utilizes the OpenMPI and OpenMP C++ libraries. OpenMPI is an open-source implementation of the MPI standard, which provides a message-passing interface to enable communication between multiple processes, often distributed across different nodes in a cluster or supercomputer. OpenMP, similar to FastFlow, simplifies parallel programming within shared-memory systems by enabling efficient use of multiple processors or cores within a single node.

In this version of the algorithm, OpenMPI handles communication between nodes, while OpenMP is employed inside each node to parallelize local computation. In contrast to the previous implementation, this solution operates in a distributed memory environment, where each node functions as an independent process with its own local memory. As a result, explicit communication between nodes is necessary to share data and coordinate tasks.

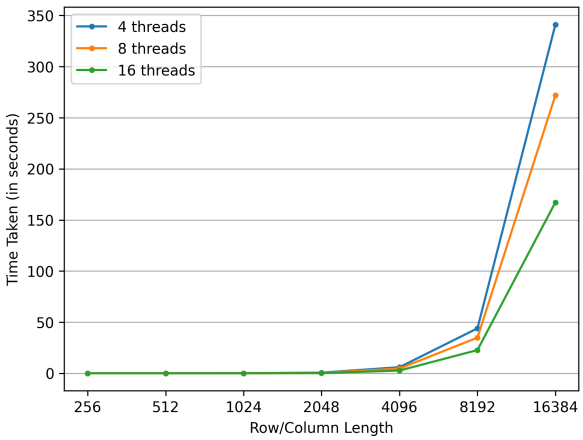

Figure 7: Strong scaling comparison

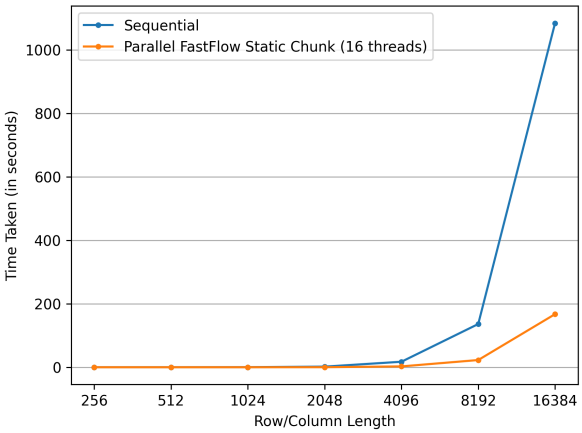

Figure 8: Comparison between sequential and FastFlow

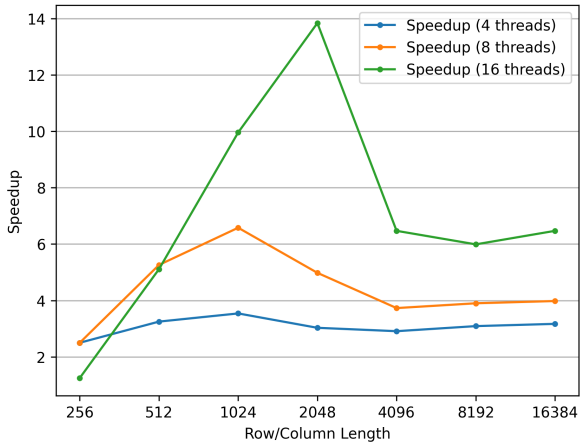

Figure 9: Speedup comparison in FastFlow

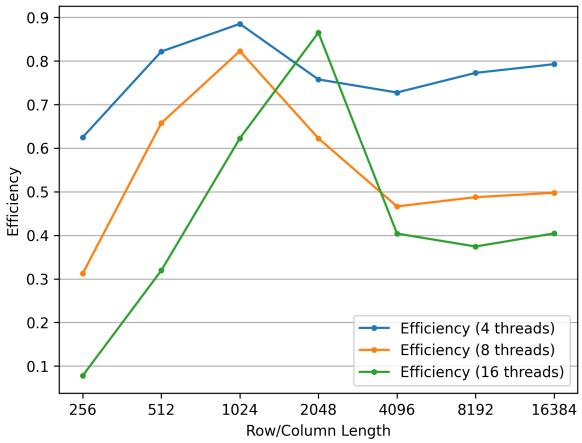

Figure 10: Efficiency comparison in FastFlow

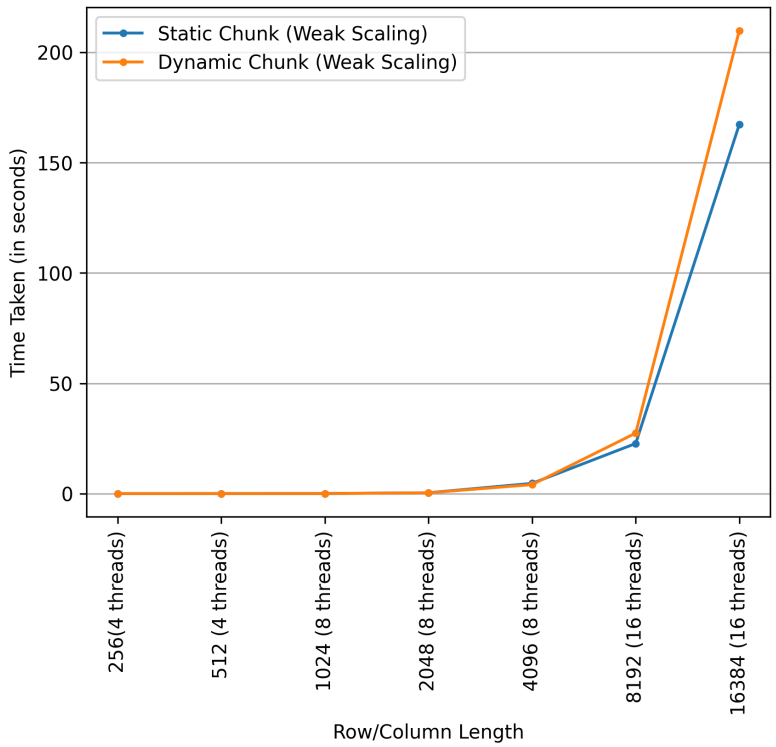

Figure 11: Weak scaling of chunck approaches

The Divide et Impera approach

To optimize the balance between local computation and inter-process communication, a *Divide et Impera* strategy has been employed. At the start of execution, the system consists of n nodes, each of which can assume the role of either `Master` or `Supporter`. A `Master` can have up to two `Supporters`, while each `Supporter` is associated with only one `Master`. `Masters` do not share `Supporters` with one another.

After determining their roles and associations independently, each node computes a submatrix of the final matrix. The size of each submatrix is determined by dividing the entire matrix by the number of active processes, while the portion of elements is determined by an id associated to the node. Once a `Supporter` has completed its assigned submatrix, it sends the result to its `Master`. The `Master` then merges the received results with its own computation.

After merging, the `Supporters` terminate their execution, decreasing the number of active processes. The remaining `Masters` redetermine their roles, potentially becoming `Supporters`, and the process iterates. The algorithm concludes when only one node remains, referred to as the `Last` node, which computes the final portion of the matrix and completes the execution. A pseudocode describing a node behaviour of this approach is shown in Algorithm 3.

Algorithm 3 Wavefront Pattern for a node in the MPI algorithm

```
while TRUE do
  role = DetermineMyRole();
  ComputeMySubMatrix();
  if I am a Master or a Supporter then
    MergeMatrices();
  end if
  if I am a Supporter or the Last active node then
    EndExecution();
  end if
end while
```

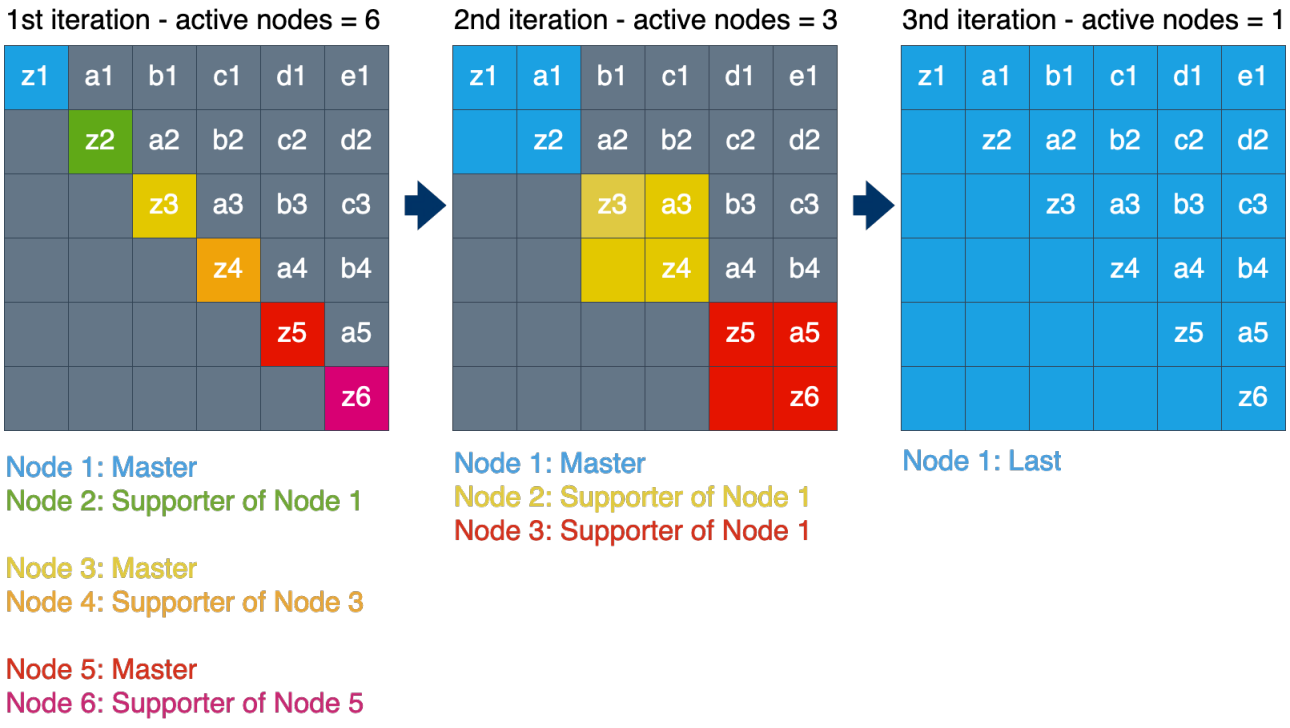

Figure 12: Example of an execution of the MPI parallel algorithm

OpenMP usage

`OpenMP` is utilized within a node to compute its submatrix. Previous implementations have shown that both the computation of elements along the same diagonal and the dot product computation for individual elements can be parallelized. Both parallelization are achieved using separate `Parallel For` constructs. In the case of the dot product computation, since the local multiplications needs to be summed to obtain the final result, the `reduction` paradigm is employed to combine these local copies into a final value. Additionally, `OpenMP` is used by both roles when merging their matrices. Because the rows to be copied are distinct, these communications can be executed concurrently.

Determining the role of a Node

As previously mentioned, a node can take on one of three roles: `Master`, `Supporter`, or `Last`. The role of a node is determined at the beginning of each iteration based on its `id`, the number of active nodes, and the size of the final matrix. If the matrix size is smaller than the number of nodes, the excess nodes are removed at the start of the computation as they are not needed. The node's `id` is recalculated at each iteration: initially, it matches the `rank` assigned by `MPI`, but in subsequent iterations, it depends on the number of active nodes. `id`s begin at zero. The relation between the `rank` of a node and its `id` is the following:

$$current\_id = \lceil rank / 2^{current\_iteration - 1} \rceil \tag{5}$$

A `Master` node is always assigned an even `id` and must have an adjacent node. `Supporter` nodes either have an odd `id` or are even-numbered without any following nodes. Once a `Supporter` node completes its computation, it exits the Wavefront computation and waits for the program to terminate. Thus the number of active nodes at each iteration is equal to the previous one divided by two. The `Last` node is, as the name implies, the only node left in the final iteration. It has a unique role since it is the only node that does not need to merge its matrix with any other node; it computes the remaining elements and then exits the computation.

Communication between nodes

After completing the computation of its sub-matrix, a node must exchange data with its partners, either sending its own results or receiving results from others. A node can communicate only with its associated `Master` or `Supporter`s. During the role determination of a node, the algorithm also identify the `rank`s of the nodes it needs to communicate with. For a `Supporter`, the node stores the rank of the `Master` in the variable `my_master`, while for a `Master`, it stores the `rank`s of its `Supporter`s in the vector `my_supporters`. The `MergeMatrices` method manages this communication.

In this method, both roles determine which rows to send or receive based on the final matrix configuration. The rows are calculated using the `Supporter` node's `id` as follows:

$$\begin{aligned} first\_row &= supporter\_id \times sub\_mtx\_length \\ last\_row &= first\_row + (sub\_mtx\_length - 1) \\ num\_rows &= (last\_row - first\_row) + 1 \end{aligned}$$

Next, the process identifies for each row, regarding its role, the index of the first element computed by the `Supporter`. Communication is then handled synchronously using either `MPI_Send` or `MPI_Recv`, depending on the role. The `std::vector<double>` containing the data is passed as a buffer, offset to the calculated index, and $lenght\_sub\_matrix$ elements are retrieved. Since

each send/receive is done on a different row, the process is sped up through parallelization using the `Parallel For` construct.

This implementation still exploits the locality of reference principle storing a copy of the columns of the upper-triangular as rows of the lower-triangular part, thus these values are considered during the copy. Although it might seem beneficial to send only the necessary elements of a row, rather than copying repeating elements, this avoids the `Master` to copy manually the received elements in their transposed position on the lower-triangular matrix.

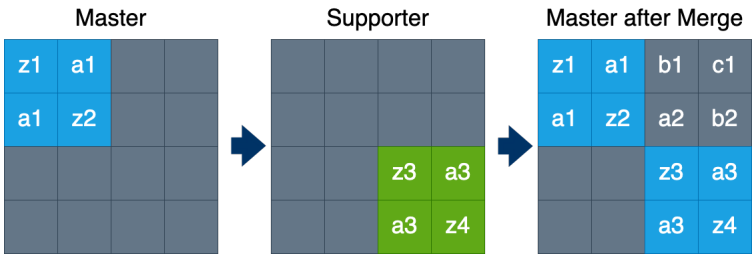

Figure 13: Example of merging the matrices of a Master and its associated Supporter.

The WavefrontNode Class

Each process spawned by `MPI` creates an instance of the `WavefrontNode` class. This class encapsulates all the methods and variables required for a node to perform the necessary computations. Upon creation, the constructor initializes essential information such as the node's `id` and `rank`, and then calls the `WaveFrontComputation()` method, which implements the pseudocode outlined in Algorithm 3. By the end of the program, the node assigned the role of `Last` holds the final matrix, which is stored as a `SquareMatrix` object within itself.

Asymptotic Complexity

Compared to the `FastFlow` algorithm, the asymptotic complexity of the current solution is theoretically more efficient. This is because each node computes a smaller portion of the overall matrix, and during the computation it can leverage both diagonal and dot product parallelism. Assuming the communication cost for sending/receiving a single element of the matrix is $\mathcal{O}(1)$, and the cost for determining the node's current role is also $\mathcal{O}(1)$, then:

- **Complexity of computing a single sub matrix** $= \mathcal{O}(\max sub\_matrix\_length) = \mathcal{O}((n-1)/num\_nodes)$ where n is the length of the matrix and num_nodes is the number of nodes employed in the pattern.
- **Complexity of sending/receiving a single row** $= row\_length \times \mathcal{O}(1) = \mathcal{O}(row\_length) = \mathcal{O}(n-1)$ where n is the length of the matrix
- **Complexity of merging matrices** $= 2 \times \mathcal{O}(row\_length) = \mathcal{O}(n-1)$
- **Complexity of an iteration for one node** $= \mathcal{O}(1)+\mathcal{O}(n-1)+\mathcal{O}((n-1)/num\_nodes) = \mathcal{O}(n-1)$
- **Overall Complexity** $= \sum_{iteration=0}^{(n-1)/num\_nodes} \mathcal{O}(n-1) = \mathcal{O}(n-1)^2$

However, as the measurement section will demonstrate, approximating the communication is overly optimistic. In reality, for smaller matrices, the communication overhead significantly impacts the overall complexity, making it less efficient than the previous shared memory solution and in some cases even the sequential case.

Measurements

The C++ implementation can be found in `src_parallel_mpi.cpp`. It's important to note that the program must be executed using the command `mpirun -N [number of nodes]`; otherwise, it will run sequentially. The implementation has been thoroughly tested with various matrix sizes and different numbers of nodes. Additionally, the behavior of the program has been evaluated both when dedicating an entire cluster node to a single program node and when assigning multiple program nodes to each machine in the cluster. Interestingly, assigning more than one process per node results in worse performance compared to the standard case as Figure 15 shows. This likely occurs because, although each process handles smaller matrices and the initial `Master-Supporter` pairs are typically assigned to processes within the same machine, resulting in lower communication overhead compared to contacting processes on different nodes, the increased overhead of managing more processes still outweighs these benefits.

For relatively small matrices (up to 1024 in length), the execution is slower than previous approaches due to communication overhead. Our tests showed minimal optimization when increasing the number of nodes from four to eight. As illustrated in Figure 16, even though each node handles smaller matrices, the communication overhead of managing additional nodes negates any potential gains. Overall, in the matrices we tested, the `FastFlow` implementation outperforms `MPI`, obviously, as accessing the matrix within a shared memory environment is significantly faster than passing submatrices.

Project Structure

The project is available on GitHub at the repository: https://github.com/Sallo97/SPM_Project-Wavefront_Pattern. The root directory includes a CMake file for compiling the project (details provided in the next section), a document describing the project, this report, a README with a brief overview and compilation instructions, and the following folders:

- **include:** Includes support libraries, specifically the `FastFlow` library for this project. Before compiling, make sure to run the *"mapping_string.sh"* script to ensure that `FastFlow` compiles and optimizes version of the code for the machine used.
- **results:** Stores `CSV` files containing the average results of each implementation. Each result is organized into subdirectories based on the implementation.
- **scripts:** Contains `SLURM` scripts used to measure the performance of the implementations in the compute cluster.
- **src:** Contains the source files for the various implementations. Within this directory, the *"utils"* subfolder holds support header files used throughout the project.

Compiling and executing the Project

To compile the entire project, open the terminal at the project's root directory and run the `cmake` command. It is recommended to keep the build directory separated from the source directory. To do this, use the command `cmake -B <build_folder>`, where `<build_folder>` is the name of the directory where you want to store the build files. Then, navigate to this build directory and execute `make` to compile the project. Following these recommendations, the binaries will be stored in the `src` subfolder. The binaries are the following:

- `sequential [length the matrix]`: execute a sequential implementation of the Wavefront pattern.

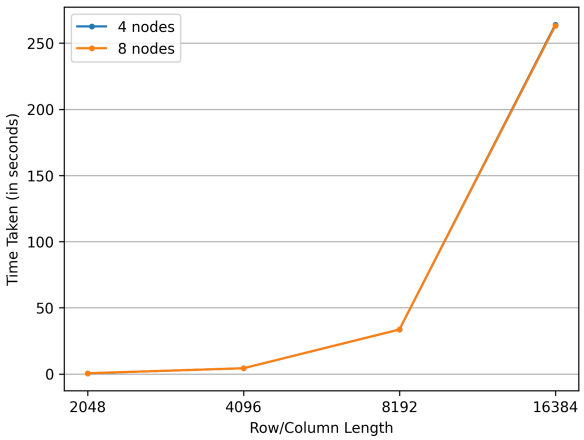

Figure 14: Comparison between MPI using 4 or 8 nodes

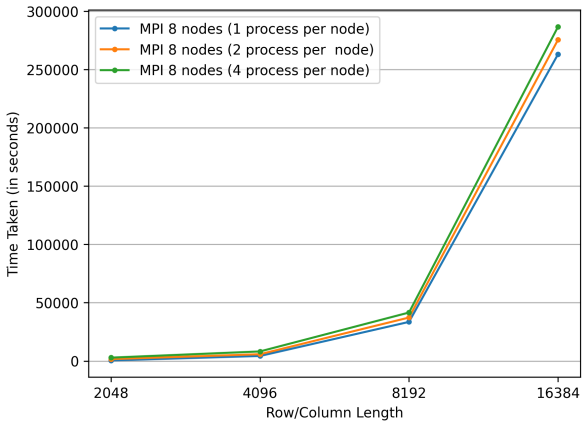

Figure 15: Comparison giving more threads per node

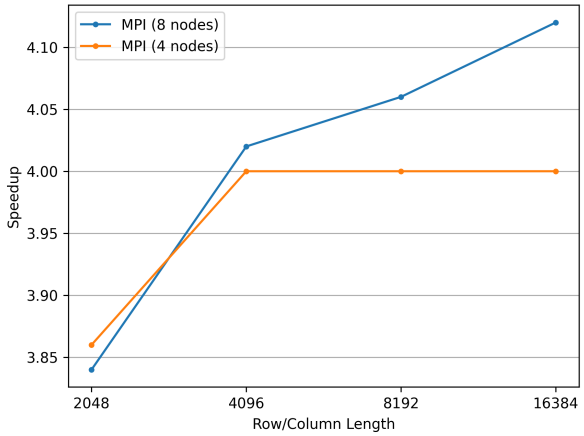

Figure 16: Speedup comparison in MPI

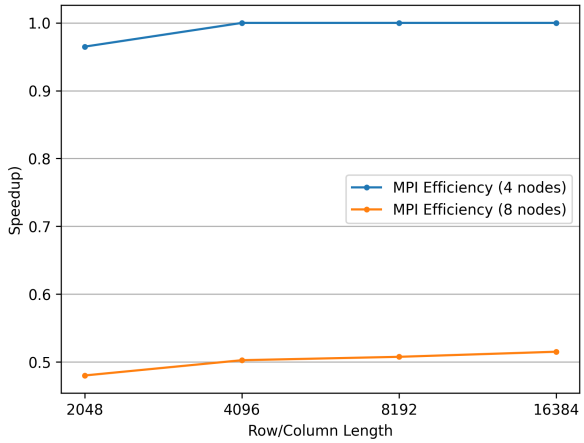

Figure 17: Efficiency comparison in MPI

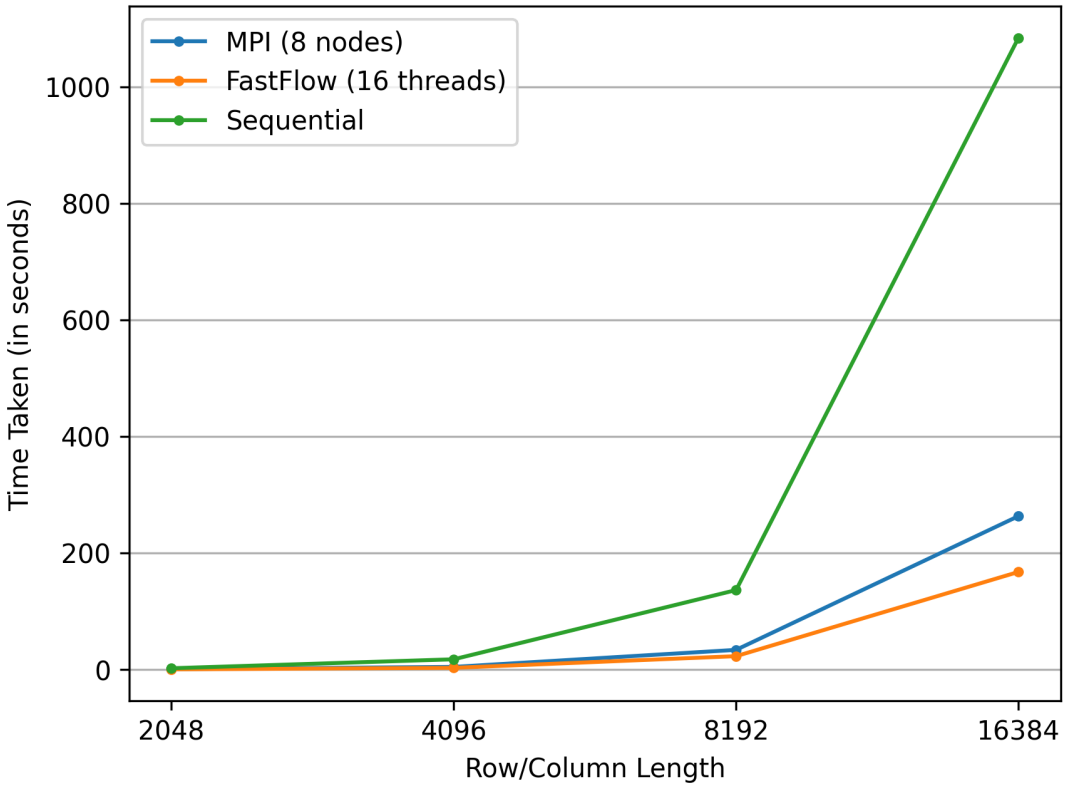

Figure 18: A general comparison between all implementation

- `parallel_fastflow_dynamic_chunk [length of the matrix] [threads to spawn]`: execute the `FastFlow` implementation of the Wavefront pattern with the given number of threads as node of the Farm. It computes dynamically the chunk size.
- `parallel_fastflow_static_chunk [length of the matrix] [num of threads to spawn]`: execute the `FastFlow` implementation of the Wavefront pattern with the given number of threads as node of the Farm. It computes dynamically the chunk size.
- `parallel_mpi [length of the matrix]`: execute the `MPI` implementation of the Wavefront pattern. Recall that to run the following binary it is needed to call `mpirun` in the following way: `mpirun -N [number of nodes to use] ./parallel_mpi [length of a row of the matrix]`.

The `scripts` folder contains scripts for each implementation, executing them with various matrix lengths (typically from 256 to 16,384) and, where applicable, different numbers of threads or nodes. These scripts were run on the *"spmcluster.unipi.it"* compute cluster to gather the measurements discussed in this report. Each script generates a `.out` file in the `results/<implementation>` directory, recording the time for each execution. The average results of the tests done are saved in `CSV` files in the results folder.

Conclusions

In this report, I explored two distinct approaches to parallelizing the Wavefront pattern: a shared memory solution using the `FastFlow` library and a distributed memory approach with `MPI`. Each implementation comes with its own set of advantages and disadvantages, highlighting the fundamental differences in addressing the problem across different architectures. The analysis revealed that the optimal approach depends on the problem's scale. For small matrices (up to 256), a sequential approach is most efficient. As matrix sizes increase, a shared memory solution like `FastFlow` becomes more effective. However, the matrices we tested are still relatively small and can be comfortably stored within the memory limits of a single process. For much larger matrices, such as those encountered in data centers, a distributed memory approach like `MPI` becomes essential. The key takeaway is to always record and compare performance measurements to account for overheads, enabling more effective and efficient parallelization strategies.