Parallel and Distributed Systems: Paradigms and Models

Prof. Massimo Torquati & Prof. Marco Danelutto

Project: Distributed Wavefront Computation: Using Sequential, FastFlow, and MPI

Name: Natesh Kumar
Matricola: 667574

Email: n.kumar3@studenti.unipi.it

Computer Science and Networking

Contents

Page

1. **Introduction** …………………………………………………………………… **3**
2. **Implementation** ………………………………………………………………….. **4**
   2.1 FastFlow Implementation ………………………………………………….. 5
   2.2 MPI Implementation ……………………………………………………….. 6

3. **Performance Evaluation** ……………………………………………………… **7**
   3.1 Experimental Setup and Metrics …………………………………………… 7
   3.2 Sequential Performance ………………………………………………….... 7
   3.3 FastFlow Performance …………………………………………………... 8
      3.3.1 Strong Scaling – Speedup ……………………………………………... 8
      3.3.2 Weak Scaling – Speedup ……………………………………………… 8
      3.3.3 Strong Scaling – Efficiency …………………………………………… 9
      3.3.4 Weak Scaling – Efficiency …………………………………………..... 9

   3.4 MPI Performance ………………………………………………………….. 10
      3.4.1 Strong Scaling – Speedup …………………………………………….. 10
      3.4.2 Weak Scaling – Speedup ……………………………………………... 10
      3.4.3 Strong Scaling – Efficiency …………………………………………... 11
      3.4.4 Weak Scaling – Efficiency ………………………………………….... 11

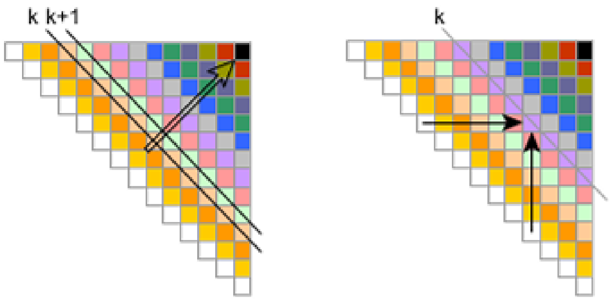
4. **Comparison of FastFlow and MPI** ………………………………………… **12**
   4.1 Speedup (Matrix Size 4000 & 4800) ………………………………….. 12
   4.2 Efficiency (Matrix Size 4000 & 4800) ………………………………... 13
5. **Conclusion** …………………………………………………………………... **14**

1. Introduction

Distributed Wavefront computation

Consider the same problem of Assignment 1 (wavefront computation). Instead of the work function "wasting time" used in the Assignment 1 version, for each diagonal element of the matrix M (NxN) of double precision elements, the new distributed version computes the *n-k* diagonal element $e^k_{m,m+k}$ ($m \in [0, n-k[$) as 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

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



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

Implement two parallel versions:

1. For a single multi-core machine using the FastFlow library
2. For a cluster of multi-core machines using MPI.

In this project, we implement and analyze the wavefront algorithm using three approaches:

1. **Sequential**: a baseline, single-threaded implementation (which is wasting time for more and larger problem sizes).
2. **FastFlow (ParallelFor)**: a shared-memory approach leveraging FastFlow's parallel_for abstraction.
3. **MPI**: a distributed-memory approach executed on a system spanning multiple nodes (up to 6).

For these versions (Sequential, FastFlow, and MPI), these Matrix Sizes (400, 800, 1200, 1600, 2000, 2400, 2800, and 3200) being executed. This report focuses on the parallel implementations FastFlow and MPI using matrix sizes of 800, 1600, 2400, and 3200 with 1, 2, 4, 8, 12, 16, and 20 workers/processes. And for the comparison between FastFlow and MPI, matrix sizes (4000 and 4800) have been taken with same number of workers/processes mentioned here.

The key performance metrics are **Execution time (ms)**, **Speedup**, and **Efficiency**, evaluated under both Strong and Weak Scaling conditions.

2. Implementation

- Choosing FastFlow’s parallel_for abstraction for our shared-memory implementation offers a straightforward and efficient way to harness multi-core processors. By implementing this approach, all worker threads will run within a shared memory space that allows them to take benefit from the processor’s high-speed cache and coherent memory architecture. So this minimizes data access Latency and ensures that the wavefront algorithm executes fast and reliably. By removing the need for complex locking mechanisms or dealing with deadlocks, the FastFlow method simplifies the code while maintaining high performance. Overall, FastFlow is not only to simplifies the parallelization process but also makes optimal use of single-node hardware, resulting in significant speedup and efficiency.

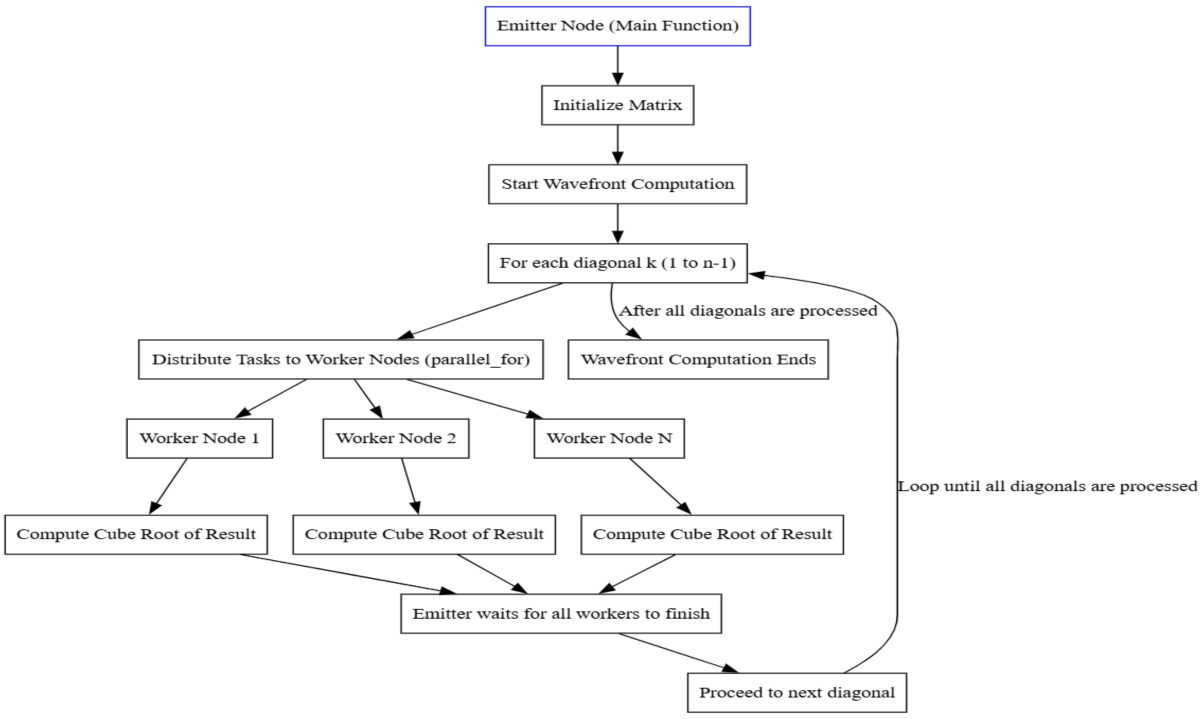
- On the other hand, the MPI implementation is created for distributed-memory systems, where the workload is distributed across multiple nodes. Each process works on its own local copy of the matrix and computes a selected portion of the wavefront independently. Once a process completes its assigned computation, collective communication (using MPI_Allgatherv) is employed to exchange results among processes to ensure that every process has the most current data before moving on to the next diagonal. Although this approach initiate communication overhead due to network latency and the costs associated with data exchange between nodes but it enables the system to handle much larger problem sizes that exceed the capacity of a single machine. MPI’s ability to support the combined resources of multiple nodes makes it a highly scalable solution for large-scale computations, even if some overhead is incurred during inter-node synchronization.

In the next section, we outline the workflow for both the FastFlow and MPI implementations, providing a clear picture of how each approach processes the wavefront computation. Following this description, we present the experimental results, focusing on execution times and scaling performance under various configurations. These results will reveal which approach is more resource-efficient and better suited to handling the problem.

2.1 FastFlow Implementation

The FastFlow implementation (**ff_parallelfor.cpp**) leverages FastFlow's `parallel_for` abstraction:

1. **Initialization**:
   - The main function reads **n** and initializes the matrix.
2. **Parallel Wavefront Computation**:
   - For each diagonal **k**, it calculates the number of tasks (**n-k**).
   - It then calls "parallel_for(0, total_tasks, 1, [&](long m){ ... })", where each worker thread computes the cube root for its assigned element.
3. **Synchronization**:
   - The `parallel_for` call blocks until all worker threads complete, ensuring that the entire diagonal is processed before moving on.
4. **Completion**:
   - After all diagonals have been processed, the FastFlow implementation completes.

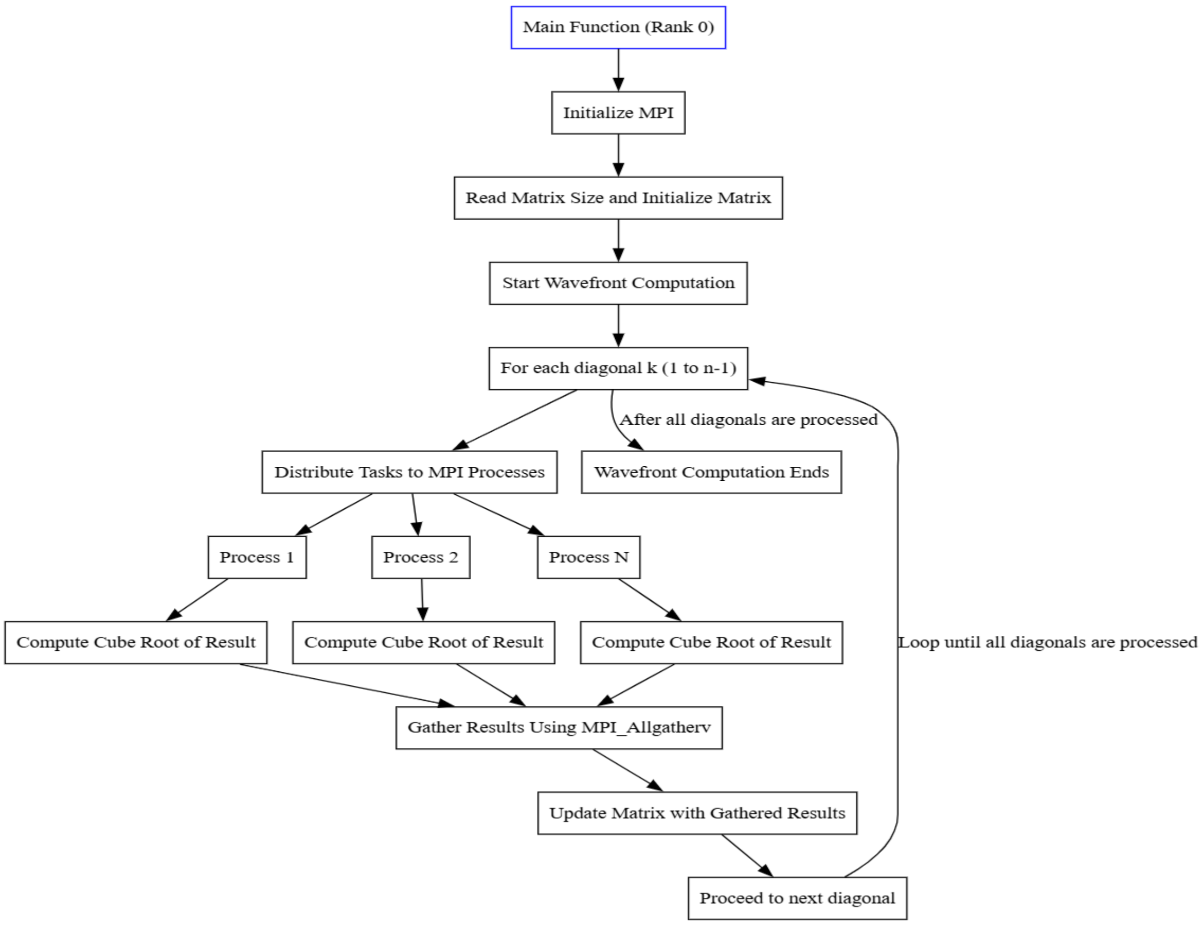

Emitter Node (Main Function)
Initialize Matrix
Start Wavefront Computation
For each diagonal k (1 to n-1)
After all diagonals are processed
Distribute Tasks to Worker Nodes (parallel_for)
Wavefront Computation Ends
Worker Node 1
Worker Node 2
Worker Node N
Loop until all diagonals are processed
Compute Cube Root of Result
Compute Cube Root of Result
Compute Cube Root of Result
Emitter waits for all workers to finish
Proceed to next diagonal

(Figure 2.1: Flow Diagram for FastFlow Implementation.)

2.2 MPI Implementation

The MPI implementation (**mpi.cpp**) distributes the workload across multiple processes:

1. **Initialization**:
    - MPI is initialized (using **`MPI_Init`**), and each process reads the matrix size **n.** And then each process initializes its own copy of the matrix and initializes the main diagonal.
2. **Distributed Wavefront Computation**:
    - For each diagonal **k**, the available tasks are partitioned among the processes (p) using static block distribution and each process computes its assigned portion independently.
3. **Collective Communication**:
    - Processes use **`MPI_Allgatherv`** to share their computed values, ensuring that each process's matrix is updated consistently before moving to compute the next diagonal.
4. **Completion**:
    - Once all diagonals have been processed, then MPI computation completes.


Main Function (Rank 0)
Initialize MPI
Read Matrix Size and Initialize Matrix
Start Wavefront Computation
For each diagonal k (1 to n-1)
After all diagonals are processed
Distribute Tasks to MPI Processes
Wavefront Computation Ends
Process 1
Process 2
Process N
Compute Cube Root of Result
Compute Cube Root of Result
Compute Cube Root of Result
Loop until all diagonals are processed
Gather Results Using MPI_Allgatherv
Update Matrix with Gathered Results
Proceed to next diagonal

(Figure 2.2: Flow Diagram for MPI Implementation.)

3. Performance Evaluation

3.1 Experimental Setup and Metrics

- **Matrix Sizes**:
    - **Sequential**: 400, 800, 1200, 1600, 2000, 2400, 2800, 3200
    - **FastFlow and MPI**: 800, 1600, 2400, 3200
    - **Comparison**: 3200, 4000, 4800
- **Worker/Process Counts**:
    - **FastFlow**: 1, 2, 4, 8, 12, 16, 20
    - **MPI**: 1, 2, 4, 8, 12, 16, 20 (using single node and multiple nodes: 1, 2, 4, 6)
- **Metrics**:
    - **Execution Time (ms)**
    - **Speedup**: S(p) = Tseq / T(p)
    - **Efficiency**: E(p) = S(p) / p
    - **Scalability**: Evaluated under strong and weak scaling.

Scripts that were used to collect csv data and generate graphs. (**run_sequential.sh, run_fastflow.sh, run_mpi.sh**)

3.2 Sequential Performance

For the Sequential method, the matrix size (400, 800, 1200, 1600, 2000, 2400, 2800, and 3200) were encountered. And by graph it can be seen that, the execution time for the Sequential wavefront algorithm proportionally increases as the problem size increases, the matrix size 3200, reaching nearly 20000 milliseconds or 20 seconds. So this rapid rise indicates the computational overhead of a purely sequential approach. So, because of this reason we moved to parallel implementations to handle larger problem sizes more efficiently and achieve significantly reduced execution times while executing in parallel.

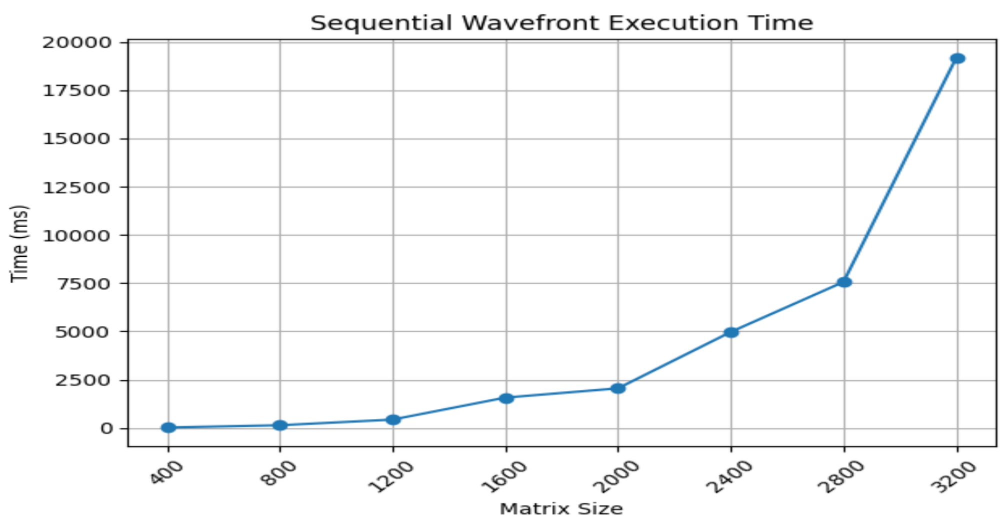


Figure 3.2

3.3 FastFlow Performance

3.3.1 Strong Scaling – Speedup

By strong scaling graph it can be seen that performance improves almost linearly as the number of workers increases from 1 to 8, and continues to scale well up to around 16 workers for matrix sizes of 1600, 2400, and 3200 but at 20 workers, speedup degrades due to factors like resource saturation, increased synchronization overhead, and higher cache miss rates. So by this observation it can say that the FastFlow approach provides effective strong scaling up to around 16 workers, after which adding more threads yields diminishing returns.

3.3.2 Weak Scaling – Speedup

The analysis of weak scaling shows that as the matrix size increases, each worker handles a larger workload, leading to improved speedup, at 8 workers the speedup is modest/medium with smaller matrices but significantly scales with larger matrices i.e. 3200. Thus this improvement becomes even more pronounced at 16 workers, while at 20 workers, a slight degradation occurs due to overhead, reinforcing that maintaining an optimal per-worker workload is key and that exceeding around 16 workers results in diminishing returns.

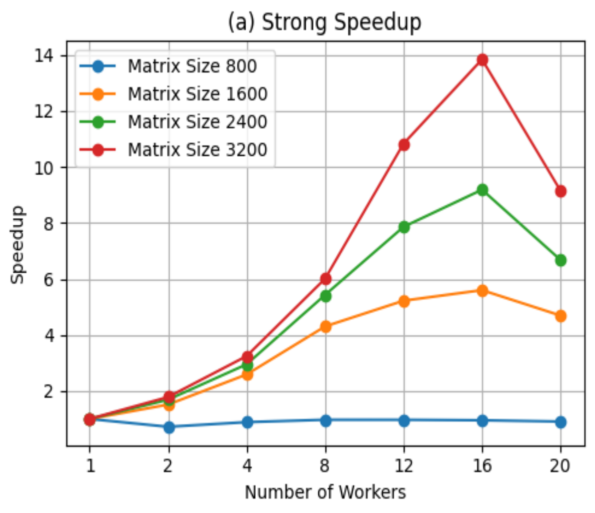


Figure 3.3.1

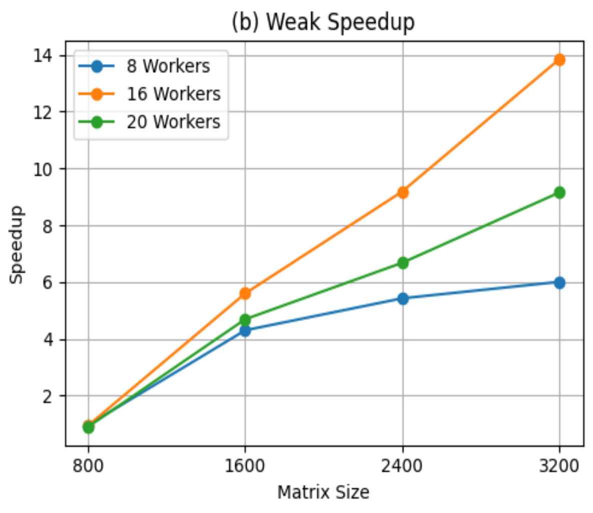


Figure 3.3.2

3.3.3 Strong Scaling – Efficiency

The strong scaling efficiency graph indicates nearly 100% efficiency at low worker counts. However, as the number of workers increases from 8 to 16, efficiency gradually declines and it drops significantly at 20 workers. This decline is because of overhead from thread management and resource contention, particularly when the worker count exceeds the number of physical cores.

3.3.4 Weak Scaling – Efficiency

The weak scaling efficiency graph shows that larger matrix sizes, like 3200, achieve higher efficiency compared to smaller ones when using 8 or 16 workers but also it can be say that using 20 workers causes efficiency to drop across all matrix sizes, emphasizing that while larger workloads help amortize overhead, exceeding the optimal worker count leads to reduced performance.

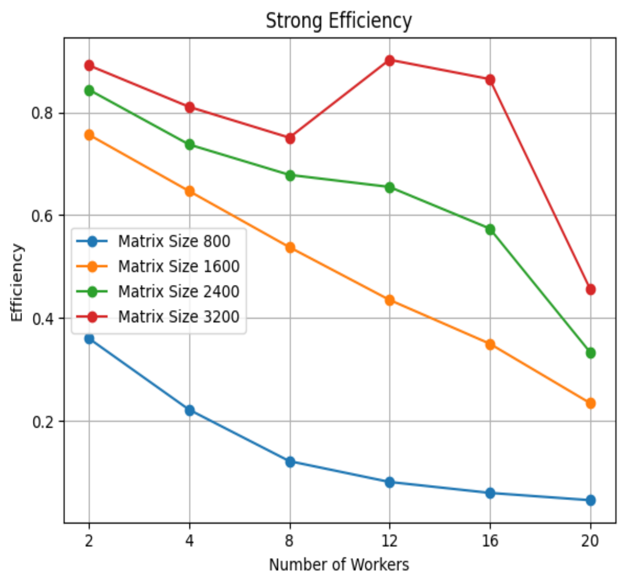


Figure 3.3.3

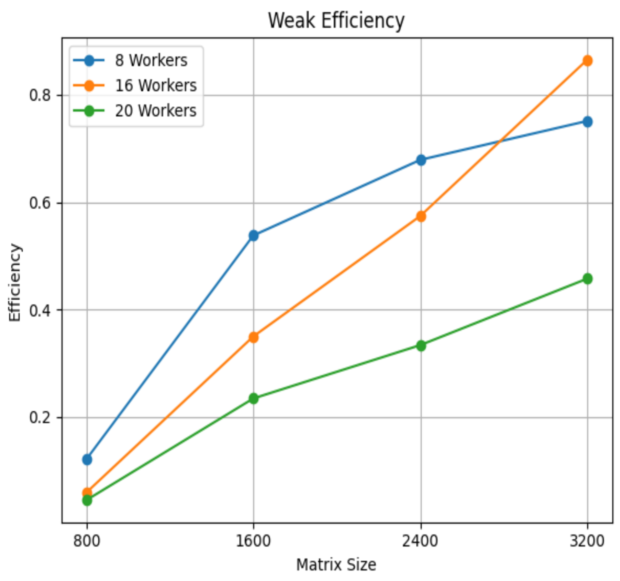


Figure 3.3.4

3.4 MPI Performance

Here in this section, we analysis the MPI across multiple nodes (1, 2, 4, and 6) to see how inter-node communication and distributed resource use affect strong/weak scaling, speedup, and efficiency.

3.4.1 Strong Scaling – Speedup

When extending MPI execution from a single node to multiple nodes (1, 2, 4, 6), the strong scaling curves generally show higher overall speedups from our experiment for larger matrix sizes 2400, 3200. Distributing the workload across more nodes allows each process to work on a smaller portion or chunks of the problem, which often yields near-linear improvements up to a certain process count. Although, as the number of processes increases 8, 16 or 16, 20, inter-node communication overhead and synchronization costs become more noticeable. Smaller matrices (e.g., 800) see limited benefits past a modest process count because the communication overhead can quickly outweigh the gains from parallelism. So compared to the single-node scenario, multiple nodes do deliver better raw speedup for the large matrices, only until the overheads of inter-node communication start to dominate.

3.4.2 Weak Scaling – Speedup

In weak scaling (where the problem size grows in proportion to the number of processes), using multiple nodes can provide a more balanced workload distribution, helping maintain or improve speedup as the matrix size increases. For instance, moving from 8 to 16 processes across multiple nodes is yield a noticeable jump in speedup for larger matrices, because each node can manage local resources to handle its share of the problem. In fact, at higher process counts the results show that 16 and 20 processes achieve nearly the same speedup for the tested matrix sizes, suggesting the additional overhead of going beyond 16 processes does not yield a significant benefit. Consequently, these findings highlight the importance of choosing a process count that balances workload distribution and overhead to avoid decline in output.

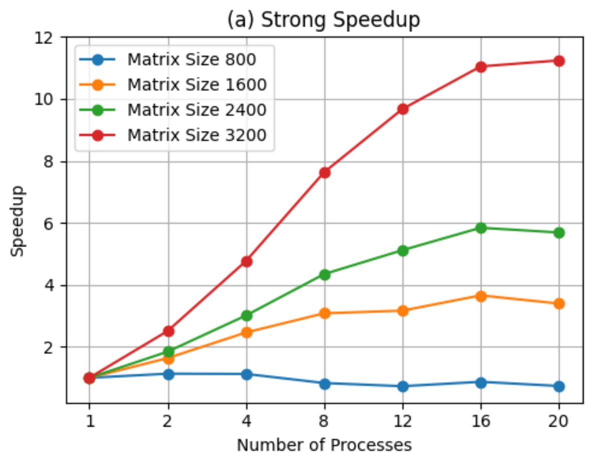


Figure 3.4.1

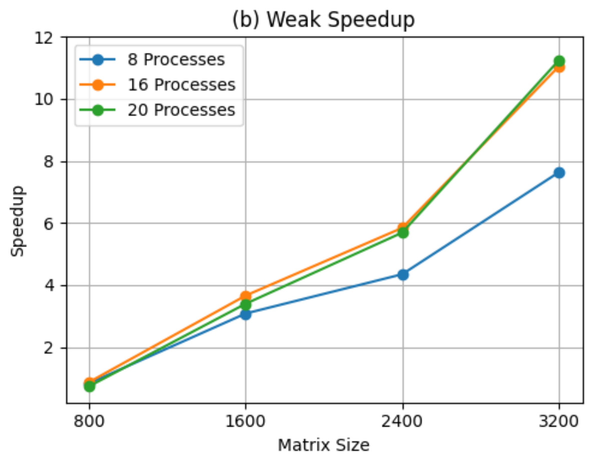


Figure 3.4.2

3.4.3 Strong Scaling – Efficiency

When moving from a single node to multiple nodes (1, 2, 4, and 6), the strong scaling efficiency trends remain similar: The strong efficiency is highest at lower process counts and steadily drops as the number of processes grows. The larger matrices 2400 or 3200 maintain better efficiency at moderate process counts because the workload is substantial enough to pay off communication overhead but, once it move beyond an optimal range often around (16-20 processes), the additional inter-node communication introduces higher latency and synchronization costs, leading to a huge drop in efficiency. Compared to the single-node scenario, multi-node configurations can initially deliver stronger performance gains, but the eventual drop in efficiency still occurs once the communication overhead outweighs the benefits of parallelism.

3.4.4 Weak Scaling – Efficiency

For weak scaling, where the problem size proportionally increases with the number of processes, the efficiency benefits more from larger matrix sizes because each node can handle a bigger part of the work (diagonal) without immediately saturating interconnect bandwidth. As it can been seen that at 8 and 16 processes, efficiency often remains relatively high, especially for matrices of size 3200. Still, when scaling to 20 processes, the increased overhead from inter-node communication causes efficiency to decline. Thus, while multi-node setups can achieve higher overall performance for large problems, they still exhibit a clear efficiency threshold, similar to the single-node case, beyond which added processes no longer translate into proportional gains.

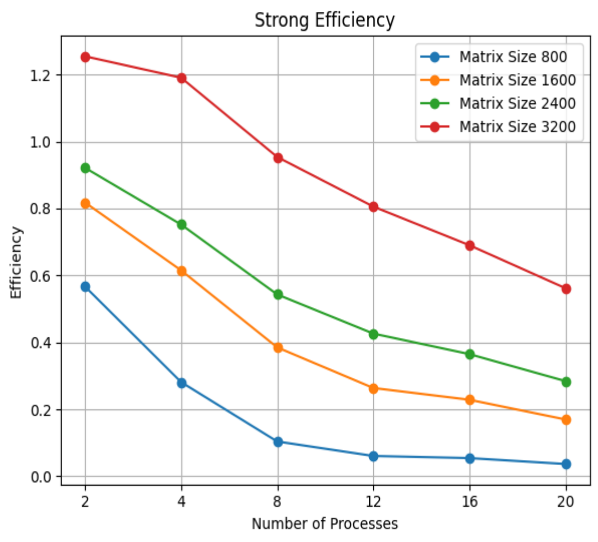


Figure 3.4.3

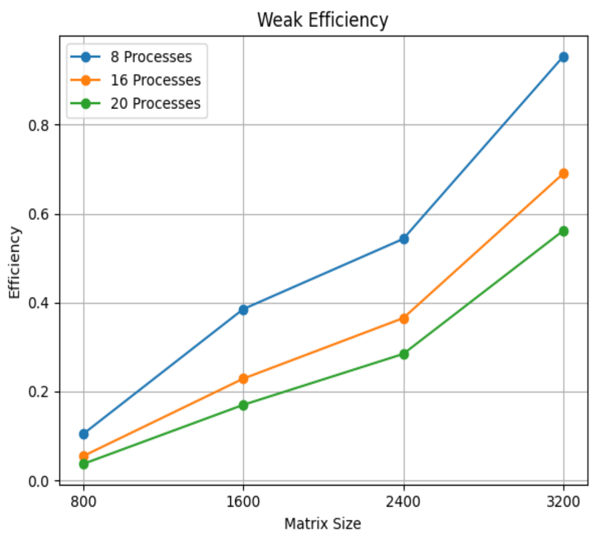


Figure 3.4.4

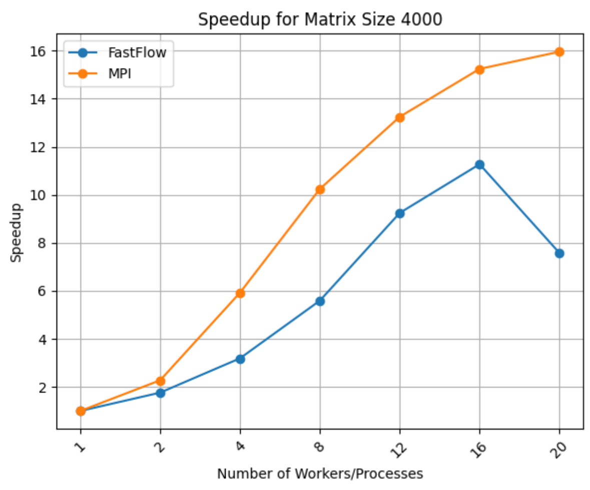
4. Comparison of FastFlow and MPI

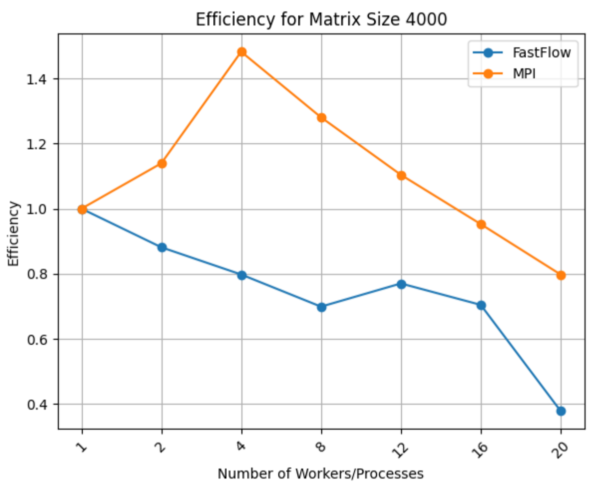
4.1 Speedup (Matrix Size 4000 & 4800):

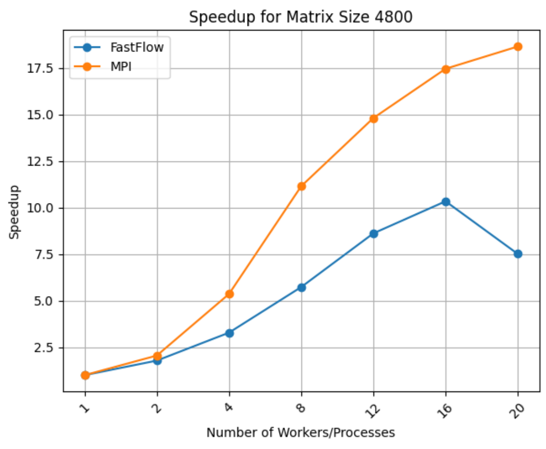
For both FastFlow and MPI, the speedup trends between matrix sizes 4000 and 4800 generally increase with additional parallelism. In the FastFlow implementation, the 4000 matrix reaches a speedup of about 9.24 with 12 workers and 11.27 with 16 workers, while the 4800 matrix achieves slightly lower peak speedups—approximately 8.61 with 12 workers and 10.34 with 16 workers. In contrast, MPI shows a more pronounced improvement with the larger problem size: for the 4000 matrix, MPI delivers a speedup of roughly 5.93 at 4 processes and 15.24 at 16 processes, whereas for the 4800 matrix the speedup scales to about 11.14 at 8 processes, 17.45 at 16 processes, and reaches 18.65 with 20 processes. This indicates that MPI’s distributed architecture extracts additional benefits from increased problem sizes, yielding higher speedup at larger scales.

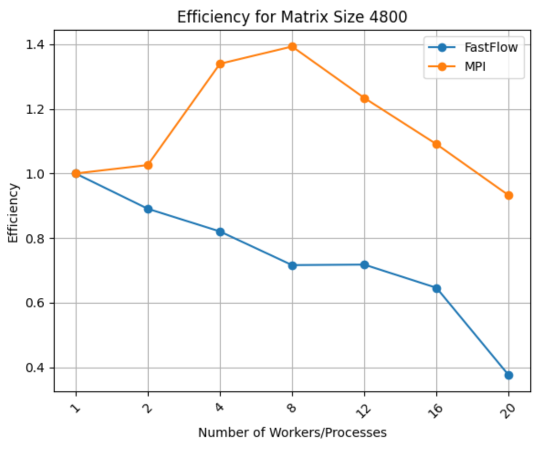
4.2 Efficiency (Matrix Size 4000 & 4800):

Examining efficiency, FastFlow maintains a similar profile for both matrix sizes starting near 1.00 for a single worker and gradually declining as the worker count increases (dropping from around 0.88–0.82 at lower counts to roughly 0.70 at 8 workers and further down to approximately 0.38 at 20 workers). In contrast, MPI exhibits superlinear efficiency at lower process counts. For the 4000 matrix, efficiencies are about 1.14 and 1.48 for 2 and 4 processes, respectively, then decline to around 0.95 at 16 processes and 0.80 at 20 processes. For the 4800 matrix, although efficiency starts slightly lower for 2 and 4 processes, it peaks at about 1.39 with 8 processes and then drops to 1.09 at 16 processes and 0.93 at 20 processes. Thus, while both matrix sizes experience a decline in efficiency at higher process counts, the 4800 matrix maintains relatively higher efficiency at moderate to high process counts compared to the 4000 matrix.









CSV Table Matrix Size 4000 and 4800

FastFlow

Matrix Size	Number of Workers	Execution Time (ms)	Speedup	Efficiency
4000	1	41824.96	1	1
4000	2	23725.54	1.76	0.88
4000	4	13105.36	3.19	0.80
4000	8	7478.488	5.59	0.70
4000	12	4524.64	9.24	0.77
4000	16	3710.356	11.27	0.70
4000	20	5520.188	7.58	0.38
4800	1	77108.34	1	1
4800	2	43280.86	1.78	0.89
4800	4	23494.44	3.28	0.82
4800	8	13457.06	5.73	0.72
4800	12	8951.494	8.61	0.72
4800	16	7460.394	10.34	0.65
4800	20	10266.42	7.51	0.38

MPI

Matrix Size	Nodes	Number of Processes	Execution Time (ms)	Speedup	Efficiency
4000	1	1	42534.7	1	1
4000	2	2	18666.66	2.28	1.14
4000	4	4	7175.38	5.93	1.48
4000	6	8	4152.128	10.24	1.28
4000	6	12	3210.75	13.25	1.10
4000	6	16	2790.754	15.24	0.95
4000	6	20	2665.746	15.96	0.80
4800	1	1	77534.84	1	1
4800	2	2	37782.2	2.05	1.03
4800	4	4	14469.2	5.36	1.34
4800	6	8	6956.974	11.14	1.39
4800	6	12	5236.514	14.81	1.23
4800	6	16	4442.906	17.45	1.09
4800	6	20	4157.892	18.65	0.93

5. Conclusion

After all the implementations and observations, it is clear that the sequential approach is too slow for large wavefront computations. The FastFlow shared-memory approach significantly improves performance for moderate problem sizes, while the MPI distributed strategy outperforms FastFlow on very large tasks by leveraging multiple nodes. Thus, FastFlow is ideal for medium-sized jobs, on the other hand MPI is the best choice for large, complex tasks requiring scalable distributed computing.