Project 3: LSH-based Similarity Join

Student Irene Pisani - 560104 - i.pisani1@studenti.unipi.it
Course Parallel and Distributed Systems: Paradigms and Models - A.Y. 23/24
Exam date March 2025

Abstract

The aim of this project was to implement several parallel versions of the Locality Sensitive Hashing-based Similarity Join algorithm (LSHSJ), using FastFlow (FF) for a single multi-core node and Message Passing Interface (MPI) for a cluster of nodes. This report describes all project phases and is structured as follows. Sect. 1 provides theoretical introductions to SJ and LSHSJ algorithms, Sect. 2 describe the employed performance evaluation set-up. Sect. 3, outline the sequential implementation and some potential parallelization strategy. Sect 4 and Sect. 5 depict the implementations based on FF and MPI, exploring their performance, limitations, and advantages. Conclusions are drown in Sect. 6.

1 Similarity Join algorithm

Similarity Join (SJ) is the problem of finding similar pairs of data according to a given distance metrics. An ideal SJ algorithm should ensure completeness of the solution, good scalability property, and insensitivity to data skew. In other words, it should be able to efficiently generate all the data-pairs having a strong similarity, even when processing large and skewed datasets.
SJ problem can be tackled using exact or approximate approaches. Exact SJ algorithms ensure completeness but often face scalability challenges on large datasets, as they may require comparing all possible pairs. Approximate SJ algorithms trade a small degree of completeness for better scalability. These methods typically leverage on a random hashing framework called Locality Sensitive Hashing (LSH) that allows to generate candidate pairs by hashing similar data into the same bucket with high probability.
SJ is widely applied in domains such as set or trajectory analysis, where balancing these trade-offs is crucial for handling real-world datasets effectively.

1.1 Introduction to LSH-based Similarity Join

The SJ algorithm considered in this project lies in the context of approximate LSH-based methods in the trajectory application domain. We are going to refer to this algorithm as LSHSJ.

According to literature, a trajectory is a polygonal line where each points belongs to $\mathbb{R}^d$, with d dimension of the trajectory. We consider a dataset obtained by the union of two collections of trajectories U and V. The objective is to find all pairs of trajectories having similarity distance lower than a certain threshold λ. More formally, we want to compute:

$$S = (u, v) \in U \times V \mid Similarity(u, v) \leq \lambda$$

LSHSJ exploits the discrete distance of Frechet as similarity metric and Frechet LSH functions as random hashing function. Each Frechet LSH function uses a random grid of dimension d, defined from a resolution parameter $\sigma = 4 \times d \times \lambda$ and an origin (or seed) randomly chosen in the hypercube $[0, \sigma[^d$, to transform a trajectory into a sequence of grid nodes that is then hashed to obtain the corresponding LSH value.
To ensure a good fraction of similar pairs to be founded, n independent iterations are required; this means the algorithm instantiates a family of n distinct Frechet LSH functions. Choosing $1 \leq n \ll d$ we can describe each trajectory through n LSH values (i.e., each trajectory now belongs to $\mathbb{R}^{n \ll d}$). Each distinct LSH value, obtained by applying the n LSH functions to each trajectory in the dataset, will define a bucket. Each bucket will be populated with all those trajectories having such LSH value for at least one of the applied LSH functions. The interesting property is that the LSH framework will place similar trajectories in the same bucket with high probability. This helps reduce the dimensional space in which to search for similar pairs. In fact, during the final similarity join operation, the Fréchet distance is computed only for pairs of trajectories that belong to the same bucket and come from different collections. Moreover, to reduce the time required to compute the distance, the algorithm uses additional near-linear time heuristics to check whether two trajectories have a distance lower than λ. Once an heuristic is satisfied, the pair is returned as similar.
The whole algorithm described above is summarized in Pseudo-code 1.

Algorithm 1 LSH-based Similarity Join

Require: Union of 2 collection of trajectories D, λ similarity threshold, n LSH function, SimilarityTest()

```
Initialize an empty set of buckets B
Initialize an empty set of similar pairs S
for each trajectory t ∈ D do
    Compute LSH signature values h(t) = [h_1(t), h_2(t), ..., h_n(t)]
    for each values i = 1 to n do
        If not exist, create a new buckets for the new LSH value b = h_i(t)
        Store LSH signature in the given bucket B[b] = h(t)
    end for
end for
for each bucket b in B do
    for each pair (h(t_1), h(t_2)) in b do
        if (t_1, t_2) come from different collections then
            if SimilarityTest(h(t_1), h(t_2)) ≥ λ then
                Add (t_1, t_2) to S
            end if
        end if
    end for
end for
return S
```

1.2 Objective

The objective of this project is to highlight the limitations of the algorithm when implemented in a sequential version and to provide C++ parallel implementations capable of operating on both shared-memory and distributed-memory systems. These parallel approaches aim to enhance the overall performances, in particularly when applied to large datasets, by maximizing with critical approach the usage of the available computational resources. This report will present a comprehensive analysis of the advantages and disadvantages of each adopted parallelization strategy, leveraging libraries such FastFlow, Message Passing Interface (MPI) and OpenMP. The comparative evaluation will focus on performance improvements to determine the most effective and suitable parallelization strategy for different computational environments.

2 Performance evaluation schema

All performance analysis were pursued following the same experimental setup described below.

Datasets and LSH framework parameters Experiments have been carried out on three different-sized datasets: a small dataset of 1 GB, a medium datasets of 5 GB and a large datasets of 10 GB. Two tiny *taxi* datasets were used just for functional tests. A total number of 8 LSH functions was used with seed and resolution, respectively, equal to 234 and 80. Similarity threshold λ values was fixed to 10.

Computational Resource All implementations mentioned in this reports were tested on `spmcluster.unipi.it` server machine, specifically a compute cluster. The testing procedure use the `sbatch` command to submit jobs through the SLURM workload manager. This ensured efficient resource allocation and parallel job execution, allowing for accurate and reliable results. Tests have been always executed on one or more internal node (i.e., ensuring that the login node was not used for computational tasks).

Metrics Performance of final solutions are measured and evaluated according to speedup, scalability and efficiency metrics using both weak and strong approaches for performance analysis. Each combination of input parameters (like number of threads, processes, or computing nodes) was executed three times to ensure a better statistical significance by taking the average completion time.

All instructions for compilation, execution, and experiments reproducibility are reported in Appendix A.1

3 From sequential towards parallel implementation of LSHSJ

The sequential version of the LSHSJ algorithm employed in this project strictly follows the pseudo-code previously outlined.

However, as demonstrated in Fig. 1, the completion time of this implementation is substantially high, growing almost linearly as the size of the dataset increases. It takes about 1 minute to process 1 GB of data and up to about 10 minutes to process 10 GB on a single internal node of the used cluster.

It is evident that the sequential execution model becomes inefficient and suffers from limited scalability when handling large-scale datasets, due to the lack of parallelization or distributed optimizations.

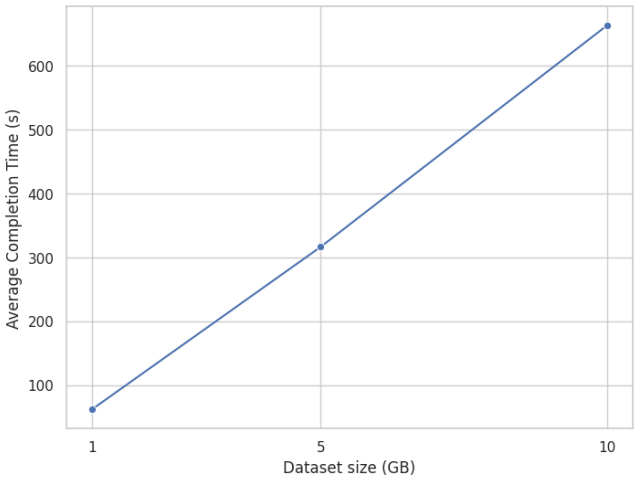


Figure 1: Completion time of sequential LSHSJ for different-sized datasets.

3.1 LSHSJ problems as Map-Reduce paradigm

In literature, major attempts to parallelize SJ and LSH-based SJ algorithms have been successfully carried out by approaching the problem through the MapReduce paradigm. Several well-established and open-source MapReduce implementations are available, which have been effectively applied to parallelize SJ problems across distributed systems. However, this project does not rely on any preexisting MapReduce framework. Instead, the goal is to conceptualize and formalize the LSHSJ problem within the MapReduce paradigm as follow.

The fundamental idea behind representing LSHSJ problems as a MapReduce computation is to divide the problem into primary stages: map, shuffle and reduce. Each phases can be formally redefined as follows:

1. **Map Phase.** For each trajectory t, emit a key-value pair $\langle (i, h_i(t)), t \rangle$ for every $i \in \{1, ..., n\}$, where n is the number of LSH functions, and $h_i(t)$ represents the LSH function result for trajectory t under the i-th function.
2. **Shuffle Phase.** The emitted key/value pairs $\langle (i, h_i(t)), t \rangle$ are shuffled and grouped by their key values, ensuring that all pairs with the same key are grouped together.
3. **Reduce Phase.** For each couple of pair $\langle (i, h_i(t^U)), t^U \in U \rangle$, $\langle (i, h_i(t^V)), t^V \in V \rangle$ within the same key group, output the pair as similar if $\text{dist}(t^U, t^V) \leq \lambda$.

This problem reformulation provides a preliminary understanding of the potential parallelization strategies that can be adopted.
Map and reduce phases are the main computational stages. These 2 tasks are dependent on one another and both can be easily parallelized by enforcing data parallelism. In the parallel map phase, computing entities (a thread, a process, or a node) can independently handle a chunk of the dataset, processing the trajectories and emitting key/value pairs. Similarly, reduce phase can be parallelized by assigning to each computing entity a subset of the grouped key-values pairs to perform the similarity join operation on.
Shuffle phase can be regarded as a communication phase between computing entities; it is used to ensure that all key-value pairs with the same key are processed together during reduce task.

The following sections detail several attempts and solutions for parallelizing the LSHSJ problem using the aforementioned parallel programming libraries. These approaches aim to optimize the algorithm execution by appropriately leveraging data parallelism. FF and MPI provide ideal built-in functionalities for achieving efficient parallelization of the LSHSJ algorithm in a MapReduce perspective.

4 Parallel implementation for Shared-Memory Systems with FastFlow

4.1 Implementation details

A preliminary phase focused on identifying the most suitable FF network topology for the MapReduce paradigm, evaluating how different FF building blocks could be used, or possibly composed/nested, to ensure an appropriate problem parallelization.

4.1.1 Network architecture: All-to-All Building Block

In FF, the Map-Reduce paradigm can be implemented by exploiting a single all-to-all (A2) building block without feedback connections, i.e., `ff::a2a`. This building block instantiates two sets of nodes/workers and uses shuffle communication between them. This topology fits the LSHSJ problem well. On one hands, we can directly incorporate the tasks dependency between map and reduce phase into the structure of the network. On the other hand, by assigning to the first set of workers the business logic of the map phase and to the second set of workers the business logic of the reduce phase, it is possible to introduce data-parallelism in both phases of the Map-Reduce paradigm. Final FF network is showed in Fig. 2

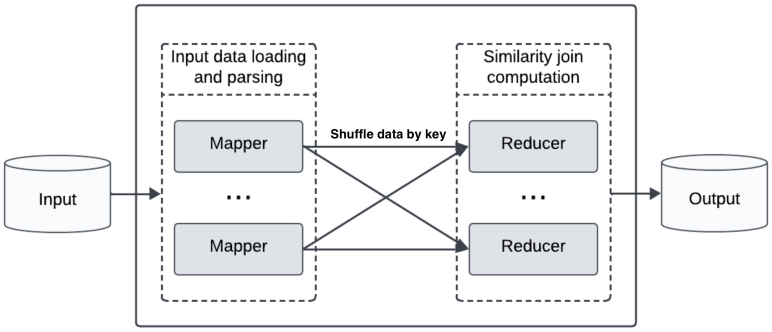


Figure 2: FF network for the final solution.

First set of workers: Mappers. The first set of workers (also called mappers) consists of *m* multi-output sequential node. A mapper receives as input the information needed to identify its own chunk of data to process independently. After seeking its portion of the dataset, a mapper begins processing it line by line, calling the provided parsing function to interpret each line as an item. For each item, the mapper invokes all the given LSH functions to compute the *key-values* pairs. Each pair is forwarded to the corresponding reducer.

Output selectivity and shuffle communication. The output selectivity property of the sequential multi-output node was exploited to emit the mapper's results (i.e., the *key-value* pairs) to the appropriate reducers. The mappers use the modulo operation on the *key* to identify the ID of the reducers to forward each pair to. This procedure is implemented by exploiting the `ff::ff_send_out_to()` method. It ensures that pairs with the same *key* are correctly routed to the same reducers and that each reducer has a finite and complete set of *key-value* pairs on which to perform SJ computations. So, mappers sends directly to reducers the computed *key-values* pairs in streaming fashion exploiting the output selectivity. Then, when the chunks have been fully processed mapper forwards to all reducers a special EOS message (exploiting defaults shuffle communication of the all-to-all BB) and terminates.

Second set of workers: Reducers. The second set of workers (reducers) consist of *r* multi-input sequential node. Thanks to input non-determinism property a reducer can receive *key, value* pairs from all mappers. A reducers collects the received pairs, aggregating and sorting them by *key* directly while receiving. As soon as all EOS messages have been received, the similarity join operation is triggered and similarity heuristics are tested over all the possible pairs. In the end, each reducers will have its own set of similar pairs.

Policy The A2A building block offers by defaults two scheduling/gathering policies: *"loose" Round-Robin* (RR) policy is used for unbounded capacity of the communication channels and On-Demand (OD) policy is used when capacity of the communication channels is bounded to 1. Both of them were taken into account during experimental phase to study how network behavior could change by changing the employed policy and the channel's capacity.

4.2 Faced challanges

Dataset reading and chunk size The first challenge encountered was identifying the most appropriate way to distribute a chunk of the dataset to each mapper. In the final solution, which proved to be the most performative, each mapper thread processes a chunk of the dataset consisting of k lines, where $k = lines/m$. Thus, static block distribution was applied to assign work to the map-workers. Some alternatives, such as using a source node to distribute data to the mappers via composition and/or nesting of `ff::pipeline`, `ff::comb` or `ff::farm`, were considered but preliminarily discarded.

Balancing work load of reducers. Since the mappers use the modulo operation on the key to select the reducers to which to send a given pair, it could happen that one reducer ends up working on more key-values pairs than another, thus leading to an unbalanced workload among the various reducers. This behavior was observed on `taxi1` and `taxi2` datasets of size $\ll 1$ GB. Some more sophisticated hashing techniques could be introduced into the algorithm to ensure a more balanced distribution of keys among reducers. However, it was observed that in practice the work load assigned to each reducer is rather balanced when measuring the cardinality of the key-value set of each reducer on the 1, 5 and 10 GB datasets. For this reason, the final implementation does not introduce any additional hashing techniques.

Data ordering The A2A building block does not guarantee any data ordering, but since the goal was to collect all similar pairs without special restrictions on their ordering, this feature does not raise any issues.

Write similar pairs to output file. The implementation required writing to a single output file all the similar pairs found by each reducer. However, considering that each reducer node is responsible for a subset of all similar pairs, a safe way to populate the output file was needed to avoid problems related to data-races and corrupted output due to concurrent attempt to write into a shared file. Some tests have been carried out to try to collect the A2A results in a sink node that safely writes the received ID pairs to the output file, but this solution seems to slow down the completion time. Final solution aggregates, at the end of the network execution, the results of each reducer into a single buffer that is then written sequentially to output file (so, no parallelization was applied in the final output writing operation).

4.3 Preliminary experiments

4.3.1 CPU threads affinity and blocking/non blocking mode

The FF implementation of LSHSJ was tested with configurations of FF compilation flags to enable or disable default thread mapping/affinity and non-blocking mode. The flag `-DBLOCKING_MODE` was used to set blocking execution mode, while the `-DNO_DEFAULT_MAPPING` flag allowed the operating system kernel to control thread mapping (i.e., disabling the default thread mapping specified by the `mapping_string.sh` command). This setup enabled us to control concurrency and identify the most effective configuration among the possible ones.
The 4 concurrency setup have been evaluated using different number of total threads: 8 (fewer than the total number of cores), 16 (equal to the number of physical cores), and 32 (equal to the number of logical cores). Both Round-Robin (RR) and On-Demand (OD) policies were considered. A 10 GB dataset was excluded from this experiment due to time constraints.
Based on the results presented in Fig. 6 in the appendix, the following observations can be made:

- **On-Demand Policy:** Completion time consistently improves when using non-blocking mode, particularly when the number of spawned threads is fewer than the number of logical cores. Thread mapping appears to have a lesser impact. However, the best completion times are achieved when default thread mapping is combined with non-blocking mode.
- **Round-Robin Policy:** Performance consistently improves when using default thread mapping. In this case, enabling or disabling blocking mode does not show any significant advantages or disadvantages.

These results support the decision to use default thread mapping and non-blocking mode for FF execution in the subsequent performance analysis.

4.3.2 Trade-off between number of mappers and reducers threads

Another preliminary experiment was conducted on the 1 GB and 5 GB datasets to investigate whether and how the number of threads spawned as mappers and reducers could impact completion times. The goal was to determine whether the map phase or the reduce phase requires a greater number of workers.
For this experiment, the total number of threads was fixed at 16 (equal to the number of physical cores), and

various configurations of mappers and reducers were tested.
As shown in Fig. 7 in A.2, the following observations can be made:

- Starting with fewer mappers and more reducers, completion times progressively decrease as the number of reducers is decreased and the number of mappers is increased up until the point where the number of mappers equals the number of reducers.
- The best completion times are achieved when the number of mappers and reducers is equal. This result holds true for both the Round-Robin (RR) and On-Demand (OD) policies.
- When the number of mappers continues to increase while the number of reducers decreases, completion times progressively grows under the RR policy. In contrast, with the OD policy, there is only a slight increase in completion time, which remains almost stable.

It is worth-noting that same behavior can be observed as well when fixing the total number of threads to a values lower than the number of cores. In light of these observations, the number of mapper and reducer threads was consistently set to be equal in all subsequent experiments.

4.4 Performance analysis

Strong performance analysis aims to analyze the behavior of the FF network's behavior as the number of threads increases up to the number of logical cores of the used node, considering both RR and OD policies.

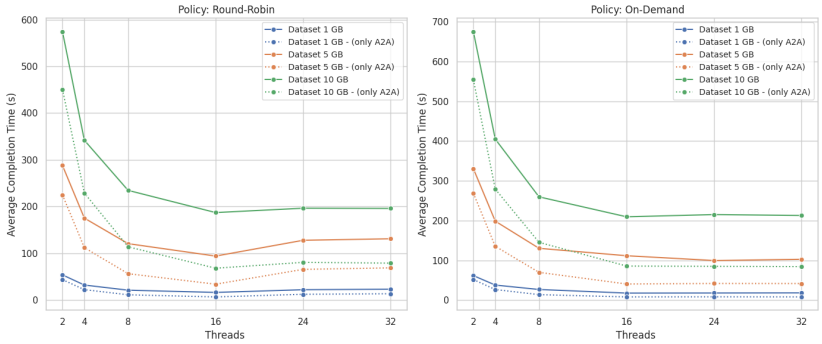
Regarding the **average completion time** (Fig. 3a), it takes into account both time for A2A network (with its creation and execution) and time spent in writing operations to output file. As the number of threads increases, A2A time progressively decrease and time spent in sequential output writing operations ends up accounting for almost 50% of the overall completion time (becoming the expensive part of the program). Looking at completion time in a weak perspective, we note that A2A completion time remains relatively stable transitioning from 1 GB to 5 GB dataset, but show a more noticeable increase for the 10 GB dataset.

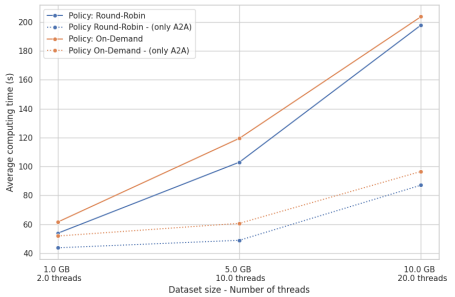
Regarding **strong speed-up** (Fig. 3b), we observe a sublinear growth until the number of threads reaches the number of physical cores. However, when transitioning to a hyperthreading context, the trend reverses, and speed-up starts to decline, leading to performance degradation. This degradation is more pronounced with the RR policy, whereas the OD policy experiences a slower decline, allowing performance to remain relatively stable at least until the number of threads matches the number of logical cores. Across all datasets, RR outperforms OD in terms of speed-up when using a number of threads equal to or fewer than the number of cores, whereas OD surpasses RR once hyperthreading is required.

Regarding **strong efficiency** (Fig. 3d), it fall down as the number of threads increases due to the growing overhead of parallelization. The cost of creating and managing a large number of threads becomes increasingly significant. The trends is similar for both OD and OR.

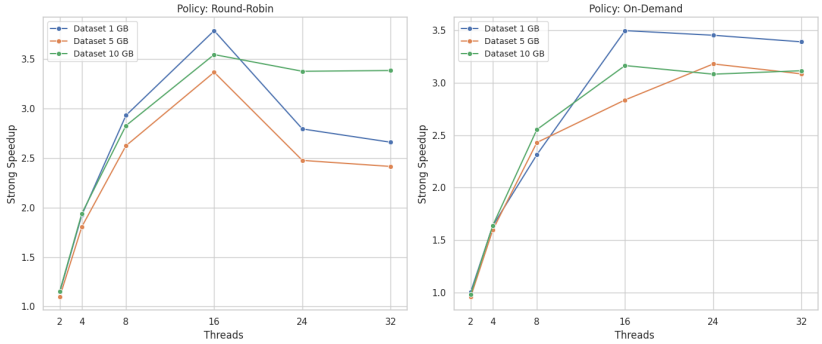
Regarding **strong scalability** (Fig. 3f), similar conclusion to strong speed-up still hold. However, when evaluating scalability while excluding output writing operations, new interesting results arise. Since scalability considers $T_{par(1)}$ instead of T_{seq}, we can restrict the scalability analysis to the A2A time alone. A2A scalability exhibits sublinear growth with a faster rate of increase, which is a positive outcome. However, when exceeding the number of physical cores, A2A scalability stops improving. Huge pitfalls is observed with the RR policy, whereas OD maintains a more stable trend.

In the weak analysis (Fig. 3c,3e and 3g,), we try to maintain a constant problem size per thread and this results in the same positive trends for **weak speed-up** and **weak scalability** and negative trend for **weak efficiency**. The following observations emerge: RR policy enables greater weak speed-up and weak efficiency for 1 GB and 5 GB datasets, with no significant differences when scaling to the larger 10 GB dataset. However, OD policy outperforms RR in terms of weak scalability, achieving higher values when scaling to larger 10 GB datasets.

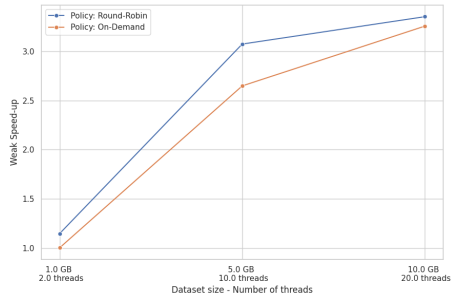




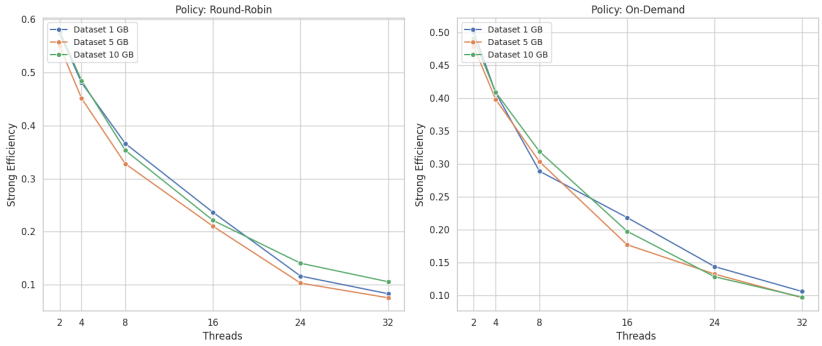
(a) Average Completion Time (s)



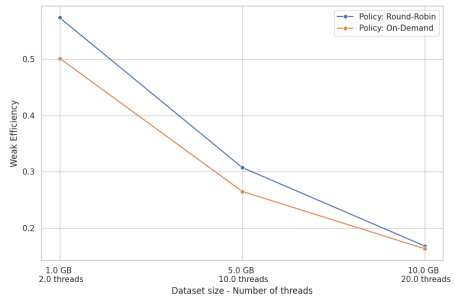
(b) Strong Speed-Up



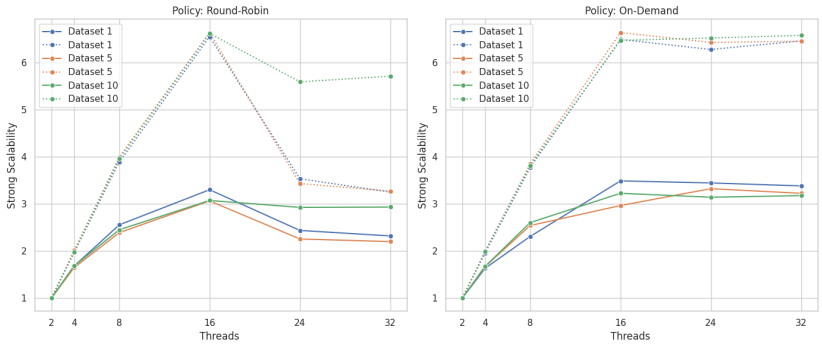
(c) Weak Speed-Up



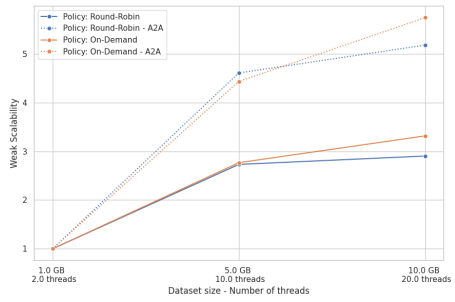
(d) Strong Efficiency



(e) Weak Efficiency



(f) Strong Scalability



(g) Weak Scalability

Figure 3: Performance analysis for FastFlow parallel implementation of LSHSJ algorithm.

4.5 Conclusion: Pro and cons of this FF implementation

The FF network composed of A2A building block with the Round Robin policy executed with threads mapping, balanced (or better equal) amount of mapper-reducer and non blocking mode, allows to achieve best parallel performance among all tested network topologies and configurations. For this reason, it has been chosen as the final version. Performances are maximized across all datasets when the total number of threads matches the number of physical cores, at the expense of a drop in efficiency.

However, this implementation exhibits critical behavior when dealing with largest dataset (10 GB) in a hyper-threading context, leading to execution failures in some cases. These failures have been thoroughly analyzed, resulting in key observations and proposed solutions.

For definition of Round-Robin gathering policy, each reducers start gathering one data element from communication channel of mappers with ID 0; than gather data from channel ID 1, channel ID 2 and so on. Due to "loose" nature of the RR policy, reducers continuously cycle through a large number of other incoming channels while mappers (es. Mapper ID 0), could keep inserting computed elements into their respective channels due to their unbounded capacity. If mappers operate at a higher rate than reducers, this imbalance causes buffers to not being emptied from the reducers, as they keep being busy extracting and processing elements from others channels. Consequently, buffer sizes grow uncontrollably, stressing the whole systems until execution failures in the worst case.

When using hyper threading, performance degradation occurs for the same reasons even with smaller datasets; with bigger dataset the behavior is just accentuated and execution can collapse for memory exhaustion reason. Other reasons like suboptimal threads mapping or non-blocking busy-waiting loop of threads can concur to failure.

In practice, this critical issue can be effectively mitigated through one of the following strategies: switch to OD policy, use a lower number of mapper threads compared to the number of used reducer threads, or compile the program with flags `-DFF_BOUNDED_BUFFER -DDEFAULT_BUFFER_CAPACITY=C` to enforce bounded communication channel with a given capacity C. During performance analysis we decide to prevent runaway buffer growth while keeping the same concurrency configuration, policy and buffer capacity; thus, we run experiments for 10 GB with a number of threads greater than 16 ensuring a bigger set of reducers with respect to the size of mappers-set.

5 Parallel implementation of LSHSJ for distributed memory systems with Message Passing Interface

5.1 Implementation details

According to the preliminary idea described in Sect. 3.1 this distributed implementation also aims to follow the MapReduce paradigm. It is based on blocking MPI collective communication operations and each instantiated process takes part in all map and reduce steps; therefore, no grouped-architecture was used. The main steps are the following:

- **Dataset reading and chunks distribution.** Since the used cluster is a diskless cluster, reading the dataset in parallel using standard MPI I/O operations could be inefficient. As solution, a single root process reads the dataset and distributes its chunks to other processes using `MPI_Scatter` operation.
- **Map phase.** Once a process has its own chunk of the dataset, the map phase could start over it. Lines of a chunk are parsed as item and n LSH function are applied over each item to obtain several key-values pairs.
- **Shuffle phase.** Computed pairs were re-distributed towards their destination process using `MPI_AlltoAll` operations during which each process send and receive a distinct different-sized amount of data to and from other processes. Also in this case, the ID of the destination process for a given key-values pairs is choosing according to modulo operation over keys.
- **Reduce Phase.** Received pairs are grouped by key, and similarity test are performed for distinct couple of pairs inside the same group.
- **Results gathering.** As final communication step, `MPI_Gather` and `MPI_Reduce` operations are called to aggregate all the founded similar pairs and their total number in the root process.

5.2 Faced challenges

Handling large amount of data during MPI collective communication During the chunk distribution or shuffle phase, the amount of data each process must send/receive via collective operations such as `MPI_Scatter` or `MPI_Alltoall` scales with the dataset size. However, MPI collective operations rely on count and displacement parameters, which are defined as integer data types. This means the the maximum amount of transferable data per process for each MPI Collective operation is limited by the maximum value representable by an integer. On most systems, this limit corresponds to approximately 2 GB. This constraint becomes problematic when dealing with large datasets and a small number of processes, where each process have to deal

with data chunks that may exceed the integer limit. To address this issue, MPI collective communications are divided into multiple, iterative communications.

- Chunk Distribution: Instead of scattering the entire dataset at once, dataset is split into smaller partitions, each within the 2 GB limit. The root process iteratively distributes these partition using multiple `MPI_Scatter` calls. Each process progressively receives its assigned chunks until the full dataset has been transmitted.
- `MPI_Alltoall` of the shuffle phase has been executed in multiple iterations as well. Each process sends and receives manageable data segments within the allowable size, ensuring that no single communication exceeds the integer limit.

This iterative approach ensures that data exchange between processes remains feasible even for extremely large datasets, effectively mitigating MPI communication limit. Although this method introduces additional overhead due to repeated communication calls, it it enables efficient large-scale data processing.

Data Serialization for MPI Communication During shuffle phase is required to exchange complex data structures but standard MPI data-types are insufficient for representing such data structures. Initial attempts were made to define new custom MPI data-types directly; however, these efforts were unsuccessful due to the complexity of the involved objects, makes it challenging to define an MPI datatype that correctly maps the memory layout of such objects. In alternatives, the usage of Cereal serialization library for converting complex objects into a format suitable for transmission was taken into account as well. Despite its flexibility and ease of integration, preliminary test have revealed an additional overhead; thus, this approach was discarded. Final solution, exploits `memcpy()` function to copy raw memory contents. This method enables the serialization of complex objects into a byte stream that can be easily transmitted between processes, simplifying data transfer and minimizing overhead.

Memory Management and Batch Processing An issue arise when running the program on the largest dataset (10 GB) using only one process; execution fails for bad memory managements and memory overload. This was problematic, as such a configuration was necessary for computing the scalability metric, which by definition requires $T_{Par(1)}$. The only effective solution proved to be processing the entire dataset in batches. This means that map and shuffle phases are executed immediately after each scatter repetition on the received chunk. Without batch processing, processes accumulate chunks received in each scatter repetitions before start next phase; with batch processing, chunks received in each scatter repetitions are no more collected in single "bigger" chunk but directly used in map and shuffle phase.

Blocking vs. Non-blocking MPI communication Significative effort was addressed to explore the potential of non-blocking MPI calls for overlapping the communication and computation phases. Since this implementation relies on iterative and interleaved computational and communication operations, it seems to be a suitable context for effectively leveraging non-blocking communication. Despite being functionally correct, all non-blocking attempts resulted in longer completion times. As this outcome was unexpected and likely indicative of suboptimal use of non-blocking communication, the performance analysis for the non-blocking version has been omitted from this report, as they always and consistently fell down compared to the blocking version. Nevertheless, the source code for the non-blocking implementation remains available in the project submissions and may serve as a basis for discussion and further investigation in future work.

Output writing operations in a disk-less cluster Once processes terminate their reduce phase, their results are gathered in the root process. This step is crucial since all founded similar pairs must be written into an output file. Given that the cluster in use is a disk-less cluster, MPI parallel I/O operations cannot be employed for each reducer to write to the output file simultaneously, as this would be inefficient. Therefore, root process sequentially write to output file the collected result; this approach ensures that no parallelization is applied to the final output operation, thereby optimizing the process in this specific environment.

5.3 Preliminary experiments

5.3.1 Trade-off between number of nodes and process per nodes

A preliminary experiment was conducted to analyze how the distribution of processes across nodes influences the overall performance of the program when executed with a total of p processes. Specifically, the study aimed to determine whether it is more efficient to allocate a larger number of processes per node while using fewer nodes or to distribute the processes across a greater number of nodes with fewer processes per node. The key objective was to identify which configuration yields better performance improvements.

To inspect this, we carried out experiments over 1 GB datasets with a total process count ranging from 1 to 16, systematically varying both the number of nodes and the number of processes per node. The results, presented in Figure 8 in the Appendix A.3, demonstrate that performances are always optimized when processes are distributed across a higher number of nodes, with fewer processes assigned to each node. This configuration led to improvements in speedup, scalability, efficiency, and overall completion time, suggesting that reducing resource-contention within individual nodes enhances performance.

5.4 Performance analysis

The strong scaling analysis aimed to evaluate the behavior of the MPI implementation as the number of nodes increased from 1 to 8, with a corresponding increase in the number of processes from 1 to 16 [1].

The weak scaling analysis was carried out under two configurations: (i) increasing the number of nodes while increasing the dataset size, keeping a constant number of processes, and (ii) increasing the number of processes while increasing the dataset size, keeping a constant number of computing nodes. Weak scaling results for the first configuration might be slightly unfair due to the limited number of computing nodes available in our cluster, which prevents assigning the same amount of work per node when scaling to larger datasets.

Regarding the **average completion time** (Figure 4a), it accounts for both completion time of the LSHSJ algorithm (including the map, shuffle, and reduce phases) and the time related to associated with I/O operations required in a diskless cluster (including dataset loading, chunk distribution, result aggregation, and output writing). As the number of processes increases, the time required for algorithmic computations progressively decreases, whereas the time spent on I/O operations remains relatively stable or increases due to communication overhead, eventually becoming the heavier part of the execution. From a weak scaling perspective, if I/O operations are excluded, completion time remains relatively stable when transitioning from a 1 GB to a 10 GB dataset.

Regarding **strong speedup** and **strong scalability** (Fig. 4b, 4d), both exhibit sub-linear growth as the number of processes increases. Speed and scalability are bounded by the amount of sequential operation involved in the programs (e.g., sequential writing operations impose a limit on achievable metrics values). If the time required for I/O operations is excluded, strong scalability exhibits a growth at higher rate, approaching linearity; the bigger the dataset the more this behavior is accentuated indicating good scalability property of the implementation, at least for the core LSHSJ algorithmic steps.

Regarding **strong efficiency** (Fig. 4c), it generally decreases as the number of processes increases due to rising communication overhead; the cost of managing and synchronizing a large number of processes becomes increasingly significant. However, a notable increase in efficiency is observed when transitioning from 1 to 2 processes for the 10 GB datasets; this improvement can be attributed to the combined effects of batch processing and iterative MPI collective operations in cases with large datasets and a small number of processes.

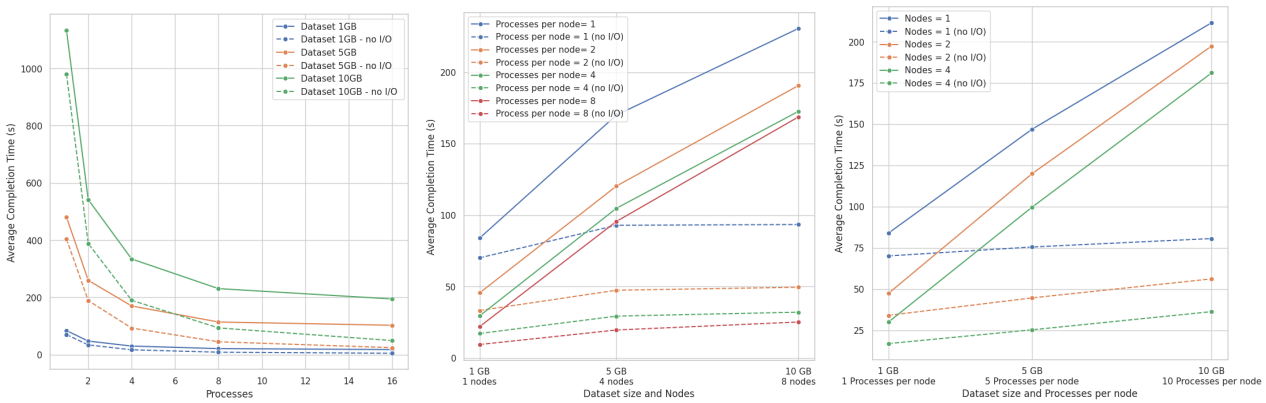
In the weak scaling analysis (Fig. 4b, 4c, and 4d), both configurations exhibit positive trends in **weak speedup** and **weak scalability**, whereas **weak efficiency** consistently follows a negative trend. Performance improvements are observed when scaling to larger datasets while maintaining a lower number of processes per node, reinforcing the outcomes described in Section 5.3.1. However, the overall scalability of the implementation is constrained by the volume of output I/O operations, which are not effectively parallelized or distributed among processes. However, focusing only on the LSHSJ algorithmic steps (map, shuffle, and reduce), the implementation demonstrates good scalability and performance enhancements in a distributed environment.

5.5 Conclusions: pro and cons of MPI parallel implementation

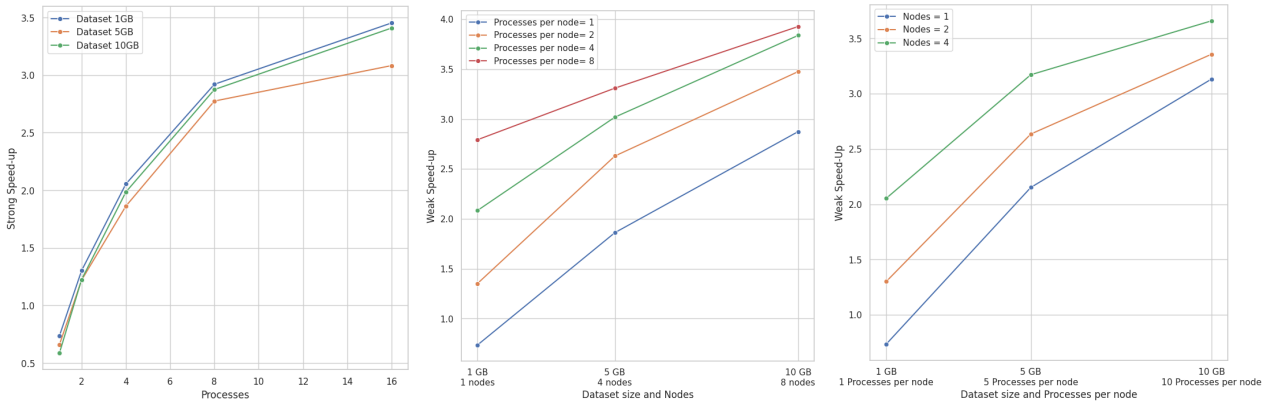
The main advantages of this distributed implementation lies in its capability to efficiently scale the core phases of the LSHSJ algorithm (map, shuffle, and reduce) when applied to larger datasets. However, the main limitations are related to the time-consuming phase sequential writing of founded similar pairs.

In a disk-based cluster these limitations could be mitigated by leveraging MPI I/O built-in functionalities to enable simultaneous file reading and writing across all available processes. It is worth noting that an alternative approach to avoid the final gathering and sequential writing of all pairs would have been to allow each process/node to write independently to separate output files. However, to maintain consistency with other provided implementations, we opted for a single output file.

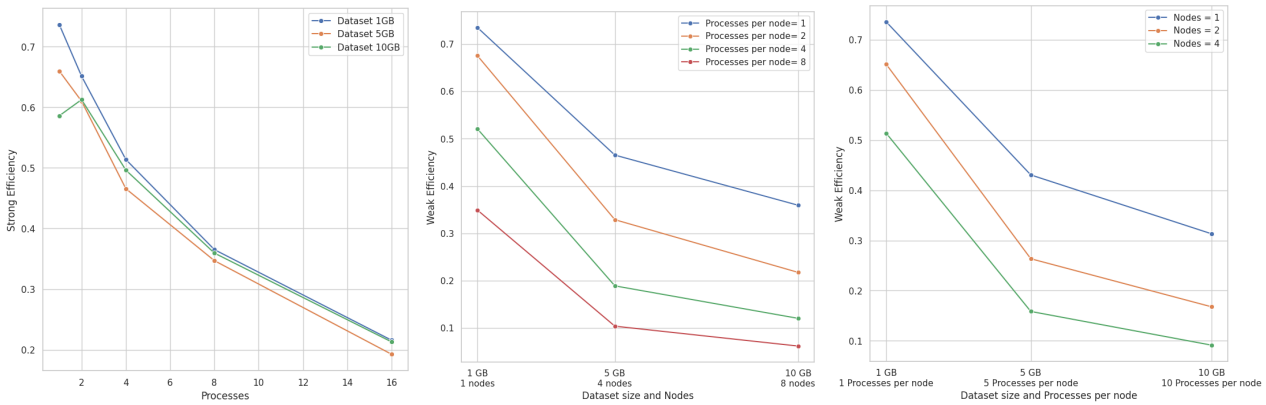
[1] The configuration for 16 processes is the only case in which multiple processes are spawned per computational node



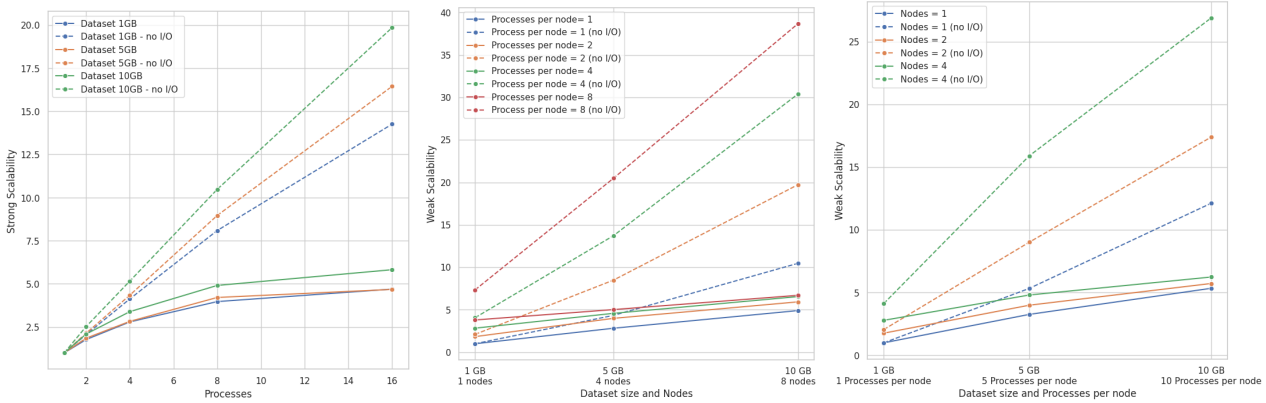
(a) Average Completion Time (s)



(b) Speed-Up



(c) Efficiency



(d) Scalability

Figure 4: Performance analysis for MPI distributed implementation of LSHSJ algorithm.

5.6 Parallel implementation for distributed-memory systems with Threaded MPI

The last parallel version of LSHSJ was developed to gain confidence with the hybrid programming paradigm of MPI + OpenMP, trying to combine the usage of more than one thread for each MPI process.
Even though MPI processes, in threaded MPI, can use threads in both communication and computation, this implementation explore the usage of more than one threads only during the computation phase of map and reduce functions.

This implementation strictly follows the one described in previous Sec. 5.1, the only main difference lies in the map and reduce functions for which we enables an additional threads-level parallelization via OpenMP. While the map function consist of a main loop over the lines of a chunk, the reduce function consist of a main loop over trajectories pairs. Both these loop can be parallelized adding the pragma directive `omp parallel for`. This allow to simultaneously process more chunk's lines at times during map phase and more than one couples at times during reduce phase. Only static scheduling has been tested in this implementation.

Here, the goal was to study how additional multi-threading could impact the overall performance. Anyway, since an extensive analysis of all the possible configuration of nodes, process, and threads would have been too time-consuming, we have restricted the analysis to a configuration that already provided good performance in the not-threaded MPI version, trying to understand if the combined usage of threads and processes could yields to additional benefits. The number of used nodes and processes per node was fixed, respectively, to 8 and 2.

Looking at Fig. 5, we can observe that threaded-MPI version allows to achieve even better completion time, speedup and scalability values, but we pay these improvements in poor efficiency as the parallelization cost grows with increasing number of spawned threads.

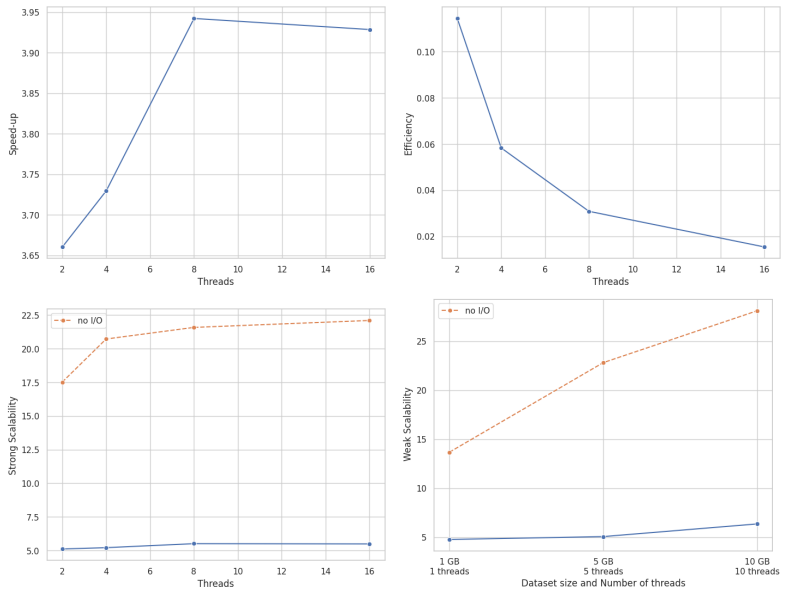


Figure 5: Performance analysis for MPI-OMP parallel implementation of LSHSJ algorithm.

6 Final considerations and further developments

In general, the size of the datasets involved in this LSHSJ problem has significative limited the number of experiments carried out. Regarding FastFlow, future efforts should aim to compare, in a more extensive manner, the performance of other building-blocks/concurrency-graphs, especially in the hyper-threading context (that was not fully explored in this project). Regarding MPI and Threaded MPI, further developments should be addressed to provide better parallelization of the I/O operations or to study advantages and disadvantages of using more than one threads per process even during the communication phases of the algorithm.

In conclusion, this work has highlighted some key challenges that can be encountered during the development of a parallel application. It has been useful to put into practice methods and tools that allow exploiting all available computational resources, thus gaining performance improvements. Moreover, it has emphasized how the field of HPC requires strong programming skills; greater knowledge and proficiency with the C++ programming language would have contributed to achieving even more optimal results.

References

[1] Rares Vernica, Michael J. Carey, and Chen Li. "Efficient parallel set-similarity joins using MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010). url: https://api.semanticscholar.org/CorpusID:10771749.

[2] Sébastien Rivault et al. "A Scalable Similarity Join Algorithm Based on MapReduce and LSH". In: *International Journal of Parallel Programming* 50 (Aug. 2022), pp. 1–21. doi: 10.1007/s10766-022-00733-6.

[3] Nicolò Tonci et al. "LSH SimilarityJoin Pattern in FastFlow". In: *International Journal of Parallel Programming* 52 (May 2024), pp. 1–24. doi: 10.1007/s10766-024-00772-1.

A Appendix

A.1 Compilation, execution and experiments reproducibility

Appropriately set up the project folder by running:

```
$ bash setup.sh
```

This commands create working directory, clone FF repository, and copy datasets and dependencies folder, then generate all executables through `Makefile`. Alternatively, executables can be generated by compiling source code using commands:

```
$ mkdir -p ./build

$ g++ -std=c++20  -I ./dependencies -I ./fastflow -O3 -DNDEBUG -ffast-math
    -o build/LSHSJ_ff src/LSHSJ_ff.cpp
    dependencies/frechet_distance.o dependencies/frechet_distance2.o
    dependencies/geometry_basics.o  -pthread

$ g++ -std=c++20  -I ./dependencies -I ./fastflow -O3 -DNDEBUG -ffast-math
    -o build/LSHSJ_seq src/LSHSJ_seq.cpp
    dependencies/frechet_distance.o dependencies/frechet_distance2.o
    dependencies/geometry_basics.o

$ mpicxx -std=c++20 -I ./dependencies -O3 -DNDEBUG -ffast-math
    -o build/LSHSJ_mpi src/LSHSJ_mpi.cpp
    dependencies/frechet_distance.o dependencies/frechet_distance2.o
    dependencies/geometry_basics.o -fopenmp

$ mpicxx -std=c++20  -I ./dependencies -O3 -DNDEBUG -ffast-math
    -o build/LSHSJ_mpi_omp src/LSHSJ_mpi_omp.cpp
    dependencies/frechet_distance.o dependencies/frechet_distance2.o
    dependencies/geometry_basics.o  -fopenmp
```

or directly with MakeFile using command:

```
$ make
```

Allocate the desired number of nodes with SLURM (i.e., one for sequential and FF implementation, and one or more for MPI implementations) and run the executables as follow:

```
$ salloc -N 1
$ build/LSHSJ_seq datasets\inFile [outputs\outFile]
$ build/LSHSJ_ff datasets\inFile m r p [outputs\outFile]

$ salloc -N 8
$ srun --mpi=pmix -n 8 build/LSHSJ_mpi datasets\inFile [outputs\outFile]
$ mpirun -x OMP_NUM_THREADS=16 --bynode --bind-to none -n 8 build/LSHSJ_mpi_omp
    datasets/inFile [outputs/outFile]
```

All experiments can be reproduced by executing the bash script:

```
$ sbatch test_seq.sh        // to reproduce sequential experiments
$ sbatch test_ff.sh         // to reproduce FF experiments
$ sbatch test_ff_flags.sh   // to reproduce FF experiments for concurrency control
$ sbatch test_ff_MR.sh      // to reproduce FF experiments for mappers-reducers trade-off
$ sbatch test_mpi.sh        // to reproduce MPI experiments
$ sbatch test_mpi_omp.sh    // to reproduce MPI+OpenMP experiments
```

A.2 FastFlow: additional content and experiments

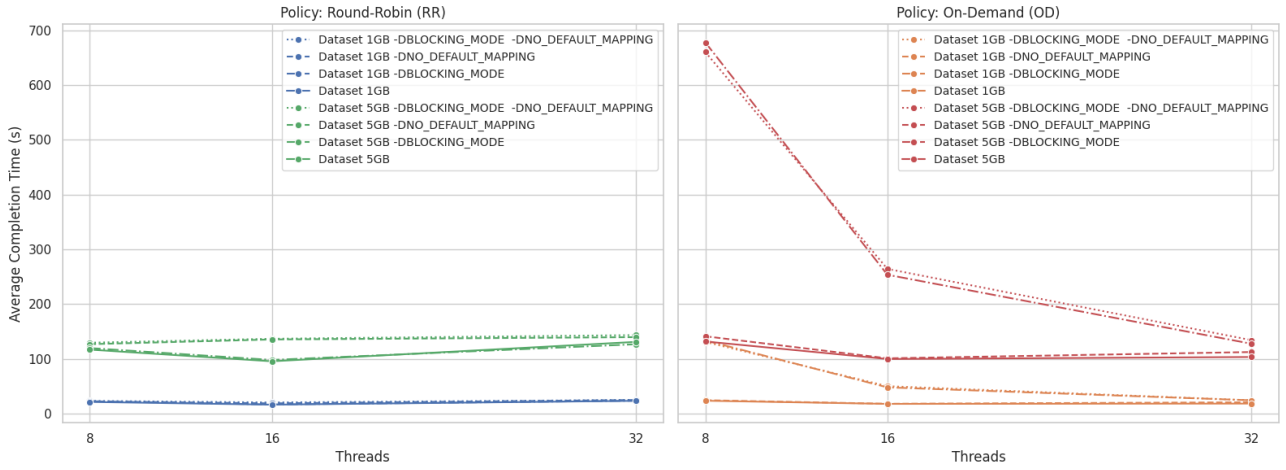


Figure 6: Impact of FF compilation flags over average completion time for 1 GB and 5 GB datasets. Figure on the left show sthe behavior with RR policy, while figure on the right shows the behavior with OD policy.

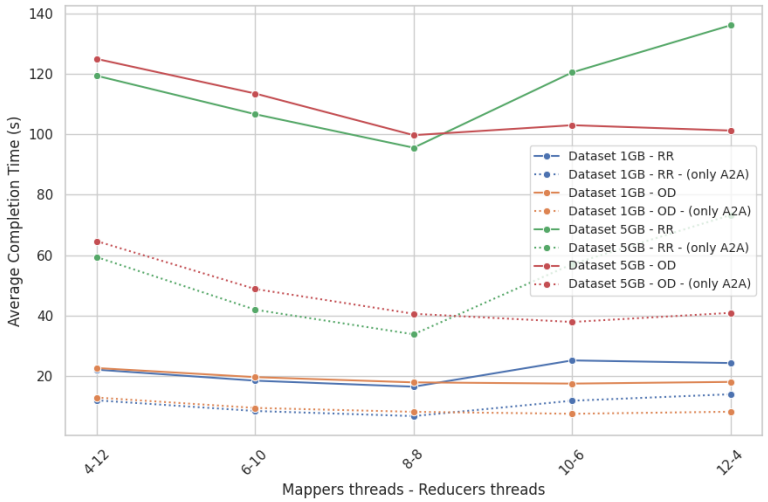


Figure 7: Analysis of how the trade-off between the number of threads in mappers and reducers sets of the A2A building block could impact the completion time for two different-sized dataset of 1GB and 5 GB.

A.3 MPI: additional content and experiments

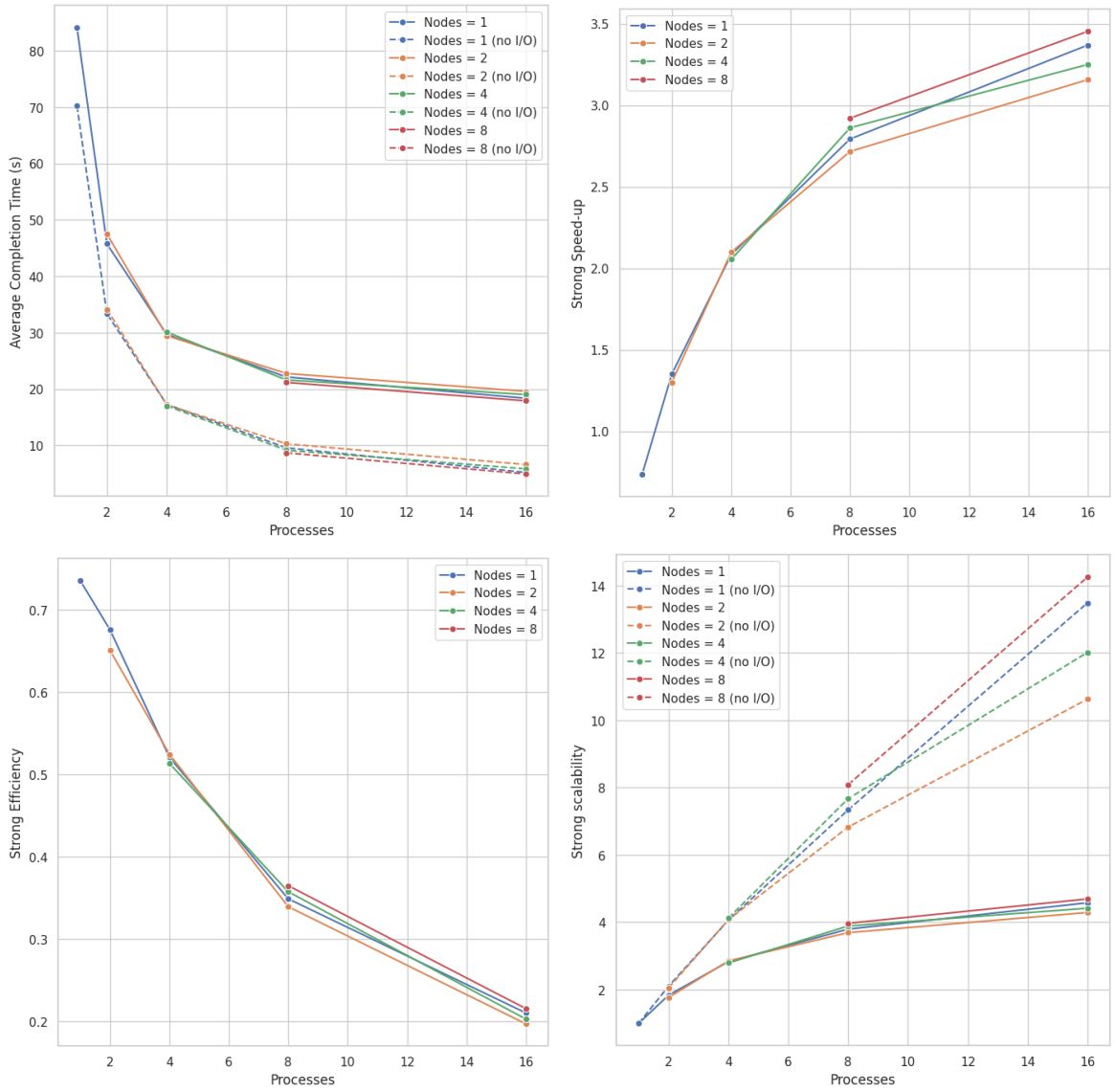


Figure 8: Analysis of how the used number of process per nodes could impact the performance of MPI-based implementation.