Parallel Apriori Algorithm with MPI and OpenMP

Giulio Mattolin
Università degli Studi di Trento
Master in Data Science
giulio.mattolin@studenti.unitn.it

1 INTRODUCTION

Over the last years the progress of technology and the reduction of the cost regarding the collection of data have led to an increasing interest of companies towards the world of Big Data. Indeed, nowadays almost every firm with moderate costs has the possibility to collect a big volume of data regarding their clients, their behaviour and their purchases in order to achieve a competitive advantage. One of the ways through which an organization can benefit from these data is by employing the data mining technique called Market Basket Analysis. Indeed, this analysis can be very useful to study and analyse the transactions of clients in order to understand customers' behaviour and carry out ad hoc marketing strategies or reorganisations of the shop layout. In particular, this technique aims to extract from a large dataset, which is typically composed of transactions made by customers, sets of items that appear together with high frequency, thus enabling the discovery of recurring patterns in clients' purchases.

The Market Basket Analysis is divided in two stages: the first concerning the identification of the itemsets that can be considered frequent in the dataset and the second concerning the use of these frequent itemsets to obtain the association rules, which indicate relationships among them. One of the most popular algorithms to perform the first, and more complex stage, of this analysis is the Apriori algorithm, which enables to extract frequent itemsets by incrementally searching for larger itemsets. However, even if this technique is popular, with very large datasets composed of many items it raises computational problems due to the high memory required and the necessity to scan the complete dataset multiple times.

Therefore, due to these problems, the following project presents a parallel implementation of the Apriori algorithm through the use of both MPI and OpenMP in order to overcome the current obstacles and provide a more efficient version of this algorithm that can run on a computer cluster. Moreover, to compare the performance of the resulting work, the algorithm has been implemented in 4 versions: serial implementation, parallelized using MPI, parallelized using OpenMP and finally parallelized using both MPI and OpenMP. The code is available at: https://github.com/GiulioMat/HPC-Apriori.

The remaining of the paper is organized as follows. Section 2 illustrates the Apriori algorithm and the addressed problem. Section 3 presents the design of the proposed solution. Section 4 describes the implementation details of the solution. Section 5 presents the dataset on which the analysis was performed and the data preprocessing stage. Section 6 illustrates the experimental evaluation carried out and presents a discussion regarding the obtained results. Lastly, section 7 draws conclusions about the overall project.

2 RELATED WORK

The work discussed in this paper is related to the problem of frequent itemset mining. The objective of this technique is to extract sets of items that appear together with high frequency from a dataset composed of transactions carried out by customers. In order to assess the frequency of an itemset it is important to introduce a key concept which is support. The support $Sup(X)$ of an itemset X is defined as the fraction of transactions that contains X and, given a minimum support s, an itemset becomes frequent if $Sup(X) \geq s$. The process of frequent itemset mining can be highly resource-intensive since the number of frequent itemsets may correspond to the number of all possible itemsets, which is $2^I - 1$ with I equal to the number of items in the dataset, and the evaluation of their support may require multiple scans of the entire dataset. To perform this process many algorithms have been implemented, but in particular the one examined during this work is the Apriori algorithm.

The Apriori algorithm is the most popular algorithm to extract frequent itemsets and helps to reduce the number of candidate itemsets by exploiting the anti-monotone property of support. Consequently, if an itemset is not frequent, then all of its supersets are also not frequent and thus can be discarded. Thanks to this principle, the algorithm is able to reduce a priori the candidate itemsets without the need to measure their support.

During each iteration the algorithm combines the F_k frequent itemsets that share the same prefix of length $k-1$ to generate all the C_{k+1} candidate itemsets, whose support will be measured and compared with the minimum support to assess if they are frequent. At each step, in order to calculate the support of candidates, a scan of the entire dataset is performed, thus to extract all frequent itemsets up to k elements there is the need to read the dataset k times. This phase is the one that requires the most computational resources and time since the algorithm has to check for the presence of each candidate in each transaction in the dataset and also, if there are many candidates, there may be memory problems caused by the storage of their support together with the full dataset. This problem especially arises during the first generation of candidates starting from the frequent itemsets of size $k = 1$ where the candidates C_2 will be $\binom{n}{2}$, with n = number of F_1. Therefore, even by taking advantage of the anti-monotone property of support, the Apriori algorithm is expensive to run on large datasets, because it produces a large number of candidates whose support has to be stored and performs several scans of the dataset.

3 DESIGN

The main idea behind the parallelization of the Apriori algorithm is the use of a master-slave architecture, in which the work is divided among multiple slave processes and there is a master process that coordinates and keeps track of progress.

In particular, the parallel solution on a distributed memory architecture works as follows. The dataset is equally divided among all the processes, so that if we have a dataset composed of 100 transactions and we use 10 processes, each one will work on 10 transactions. Then, each process will read its own part of dataset, thus referring to the previous example 10 transactions, and store the C_1 candidate items found in their local partition together with their frequency. Subsequently the measured frequency will be divided by the full length of the dataset, thus 100, in order to obtain the support. After that, each slave process will communicate to the master the extracted candidate items and their support in order to achieve a complete view of the dataset and proceed to the pruning. The master will thus receive the found items and their supports from each local partition of the dataset and sum them in order to obtain a global support count that will be exploited to prune items X with $Sup(X) < s$. Afterwards, once obtained the frequent items F_1, the master will proceed to broadcast them to all slave processes. In this way each process will be able to combine them to generate the candidates C_2 and, if there are any, search for them in their local partition. The same procedure will be applied to obtain the frequent itemsets F_2 with the corresponding candidates C_3 and so on. After the first step, the iteration of this process will continue until there will not be any frequent itemset F_k found, meaning that all frequent itemsets have been found. This is due to the Apriori principle which indicates that if there are no frequent itemset of size k, then also their supersets of bigger size will not be frequent.

This solution helps to minimize the problems of the Apriori algorithm described previously, improving both the execution time and the memory usage. Indeed, thanks to the division of the dataset among processes it is not anymore necessary to store the entire dataset and scan it at each iteration, but the work will be divided among processes which will parallelly inspect just a partition of it, reducing the complexity. Moreover the processes will just store the supports of the candidate itemsets they find in their local partition of dataset, avoiding the need to keep a counter for all the ones present in the full dataset.

For this to work there is a communication cost that has to be taken into account. Indeed, at each iteration that results in the extraction of frequent itemsets F_K and computation of candidates C_{k+1}, the solution requires to carry out 2 sends from each slave process to the master, in order to communicate both the found itemsets and their support, and a broadcast of the pruned frequent itemsets and their number from the master to the slaves.

In addition, in order to add another layer of parallelization and take advantage of a shared memory architecture, some operations are executed simultaneously by multiple threads and adapted to achieve faster results and avoid race conditions. In particular, such procedures are: the division of each itemset frequency by the number of transactions to obtain support, the extraction and counting of the frequency of candidate itemsets of size $k > 1$ from the dataset and the combination of frequent itemsets to get the next candidates. During these operations, the work is divided among threads that parallelly handle a section of the structure they have to process in order to achieve better performances.

4 IMPLEMENTATION

The solution was implemented using C++ in order to take advantage of STL data structures such as vectors and maps. The description of how the algorithm has been implemented will start from the serial version, then move to the MPI version, then to the OMP one and lastly to the MPI + OMP version.

Starting from the serial version, the algorithm begins by iterating through each line of the input dataset and creating a matrix structure, composed of sorted vectors, in which each row corresponds to a transaction. In this process, while reading each line, the frequency of each item is also tracked and stored in a map structure in which the key is the item and the value is

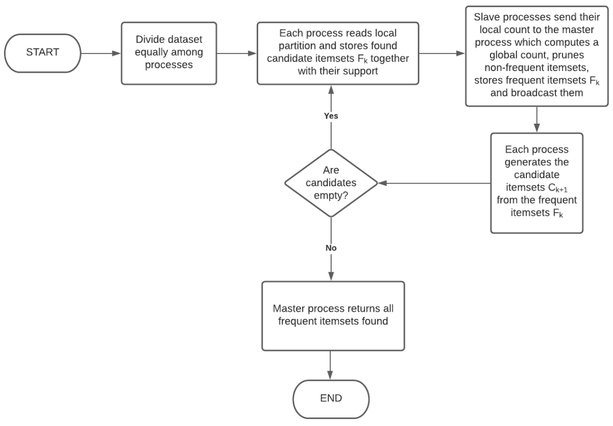

START
Divide dataset equally among processes
Each process reads local partition and stores found candidate itemsets F_k together with their support
Slave processes send their local count to the master process which computes a global count, prunes non-frequent itemsets, stores frequent itemsets F_k and broadcast them
Each process generates the candidate itemsets C_{k+1} from the frequent itemsets F_k
Are candidates empty?
Yes
No
Master process returns all frequent itemsets found
END

Figure 1: Flowchart of the proposed Apriori algorithm.

its frequency. Therefore, at the end of this we will obtain a matrix representing the dataset, that will be used in the following scans, together with the 1-candidates and their frequency. To note that rows in the matrix were sorted in order to avoid that, in the extraction of itemsets of size $k > 1$, itemset with the same elements but in different order will be considered unequal. After that, the frequency of each item is divided by the number of transactions in order to obtain the support. Then the pruning stage is executed, so each 1-candidate support is compared with the minimum support: if it is lower it will be removed from the map, while if it is greater or equal it will be kept in the map and added to a vector. The latter vector will then be used to create the 2-candidates vector by combining extracted frequent itemsets that have $k - 1$ items in common, so in this case 0. After that the first iteration is finished and the candidate vector will be checked to ensure that it is not empty and, if not, the second iteration will start. A temporary map, in which 2-candidates will be stored, is created and the matrix representing the dataset is scanned in a recursive way to find 2-candidates and their frequency. To improve the performance, a vector containing the single items that compose the candidates is used in this process to skip items that should not be considered. At the end of this, the temporary map will be filled with 2-candidates and their frequency, which will be divided by the number of transactions to obtain the support. Once again the process of pruning is applied with the generation of the 3-candidates and the resulting 2-frequent itemsets will be appended to the main map, which contains also the previous 1-frequent itemsets. The 3-candidates vector will be checked if it is empty and, if not, the process will iterate until all frequent itemsets are extracted. Finally all the obtained frequent itemsets and their support are printed.

The MPI version takes advantage of different processes to achieve a master-slave architecture. At the beginning, all processes calculate a local start and end of the dataset based on their rank, in order to divide the full dataset among all processes. After that, each process executes the read of the file and the store of 1-candidates together with their frequency considering only the assigned portion of transactions. Then each process divides the obtained frequencies by the number of transactions of the full dataset to obtain the support. At this point, the pruning process is carried out taking advantage of the master-slave architecture in order to have a master node, which is designated as the process with rank 0, containing all the extracted 1-candidates and their total support. Since the extracted candidates and their support

Order	Products
0	33120 28985 9327 45918 30035 17794
1	33754 24838 17704
2	46842 26434 39758 27761

Table 1: Illustration of the dataset after preprocessing.

are stored in a map, to send them it was adopted the strategy of sending the candidates as a single string and their support as a vector. In particular, the itemsets are concatenated in a single string in which the character | divides one itemset from another. In this way only 2 sends, one for the string of candidates and another one for their support, are required from each slave process to the master. On the other hand, the master receives these data from the other processes in a dynamic way, taking advantage of MPI_Status and MPI_Probe. The master then processes the strings in order to get the single itemsets and sums their supports to achieve a global count for each candidate. After having received everything the master starts the pruning process and, at the end, it broadcasts the resulting frequent itemsets. The frequent itemsets are sent as a single string with the same method used for the candidates but, differently from before, there is the need for two broadcasts in order to first send the size of the string and then the string itself. After every process has received the frequent itemsets, they combine them in order to generate the next candidates. If the candidates vector is empty the process ends and the master prints all the extracted frequent itemsets together with their support, while if there are candidates the process continues with the same technique by looking at the next size frequent itemsets.

The OMP version is similar to the serial one except for the use of different threads to: divide the frequencies by the number of transactions to obtain the support, find candidate itemsets of size $k > 2$ from the dataset and combine the frequent itemsets to get the next candidates. The first operation is handled by using a parallel for along all the $key : value$ pairs of the map. In particular, at each iteration the threads allocate a private pointer towards the pair they need to work on and apply the division. Differently from before, there is not the use of a single pointer since each thread would have tried to use it simultaneously causing race conditions. The second operation is executed by using a parallel for along all transactions of the dataset, in order to split the matrix among threads and find the itemsets in parallel. To note that a critical directive has been applied in the insertion of candidates and their frequency in the map in order to avoid that multiple threads try to insert a new element in the same memory address. Lastly, the third operation was parallelized once again taking advantage of the parallel for along the vector of frequent itemsets. The combination process is executed through two nested loops and only the outer one was parallelized. In order to not generate race conditions some variables were defined as private and a critical directive was applied when the generated combinations are added in the vector containing the candidates, once again to avoid the insertion of elements in the same memory address.

Finally, the MPI + OMP version was implemented by combining the changes applied in the OMP version to the MPI version, thus taking advantage of the two different levels of parallelization and enabling each process to use several threads to carry out the previously illustrated operations.

5 DATASET

The dataset used in this work is taken from Kaggle and contains data of over 3 million orders from more than 200,000 Instacart users. [1] In particular, for the experimental evaluation presented in this paper the dataset taken into consideration is in the *order_products__prior.csv* file, which contains 3214874 transaction composed of 49677 unique items and has a size of 550MB. Moreover, in order to measure the performance of the solution on different dataset sizes, the dataset was divided into different random samples containing a fraction of it. In the following list are illustrated the considered fractions of the full dataset and the number of transactions they contain:

- 0.01 → 32149 transactions
- 0.05 → 160744 transactions
- 0.15 → 482231 transactions
- 0.30 → 964462 transactions
- 0.50 → 1607437 transactions

Initially the dataset was in a .cvs file containing on each row one order_id and one product_id. To obtain a more usable structure it was preprocessed using Python. Indeed, the products have been grouped by order_id and each row was arranged to have products of the same order separated by an empty space. Then the dataset was split. as previously described, and it was saved in .txt files in order to be easily read using C++. The preprocessing code can be found in the *data_preprocessing.ipynb* file.

	Dataset size					
Processes	0.01	0.05	0.15	0.30	0.50	1
Serial	50.8	247.2	758.8	1568.5	3281.6	6567.2
10	6.3	26.7	82.5	178.7	284	715.8
30	2	10	27	62.5	99.6	237
60	1.5	5.8	15	32.9	52.5	131.6
100	1.3	4.1	11.3	20.9	32.5	64.4

Table 2: Execution times with MPI.

	Dataset size					
Threads	0.01	0.05	0.15	0.30	0.50	1
Serial	50.8	247.2	758.8	1568.5	3281.6	6567.2
10	34.2	175.5	319.4	814	1135.5	2850.5
30	16.6	66.6	242.2	468.4	618.2	1044.2
60	10.9	50.7	141.2	273	451.3	909.3
100	6.8	42.2	85.2	173.5	356.2	822.2

Table 3: Execution times with OMP.

	Dataset size					
Processes/ threads	0.01	0.05	0.15	0.30	0.50	1
Serial	50.8	247.2	758.8	1568.5	3281.6	6567.2
10/10	4.5	14.7	40	108.8	150.7	265
30/10	1.5	6.4	17.4	39.9	58.9	79.4
60/10	1	3.6	10	18.4	25	40.7
100/10	0.8	2.5	7.6	14.5	19.6	33.8

Table 4: Execution times with MPI + OMP.

6 EXPERIMENTAL EVALUATION

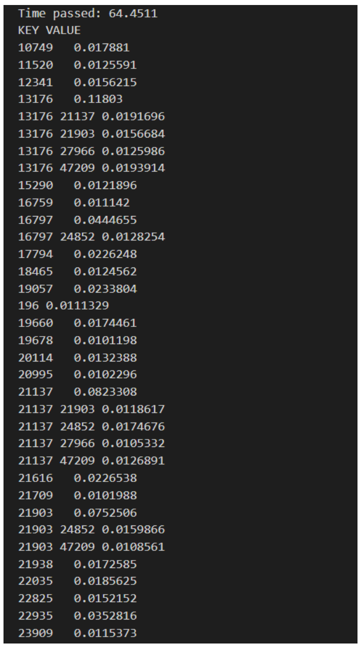
The performance of the implemented solutions were measured in order to assess the speedup and the efficiency of each version. Execution times, starting from the read of the file up to the print of all frequent itemsets, were collected varying the number of processes/threads used and the size of the dataset. To note that for solutions with MPI the considered time was the one measured by the master process. The versions were run on the different datasets illustrated in the previous section with a minimum support of 0.01, since it produced at least 2-frequent itemsets in all dataset sizes and thus enabled to analyse the algorithm more deeply.

In order to also determine the correctness of the obtained frequent itemsets and their support, the results were compared with the ones obtained by the library *mlxtend* in Python. The Apriori implementation available in the library however did not succeed to extract the frequent itemsets due to memory problems, thus it was used the FP Growth algorithm. By comparing the results it was confirmed that the implemented algorithm was working properly and returning the correct frequent itemsets and supports.

It is possible to inspect the execution times in seconds of the different parallel versions implemented and compare them with the serial version in Table 2, 3, and 4. To note that in the third table "10/10" means 10 nodes with 10 CPUs each were used, resulting in 10 MPI processes with 10 OMP threads each. All solutions are faster than the serial version. In particular, by looking at the first 2 tables, the MPI solution works better than the OMP one and can achieve good execution times with much less processes. This is motivated by the fact that the design is centered around a master-slave architecture that takes advantage of distributed memory and OMP handles just a set of operations. Indeed, most of them required the insertion of elements into shared vectors of unknown size and, since this had to be done in sequence in order to not generate race conditions, they could only be partially parallelized with OMP. Also the extraction of 1-candidates could not be optimized with OMP since it occurs during the reading of the input file, whose number of lines is unknown a priori. Therefore, the addition of OMP in the solution seems not to be very efficient, as it can also be seen in the last table in which to achieve good execution times it is required to have more CPUs than the ones required for the just MPI solution.

Moreover, in order to understand better what are the performances with respect to the serial version and how much are efficient the adopted parallelizations, speedup and efficiency were calculated using Python. It is possible to visualize the related plots in Figure 3, 4, 5, 6, 7, 8 and inspect the used code in the *speedup_efficiency_plot.ipynb* file.

Looking at the speedup it is possible to confirm what was previously affirmed, that the OMP version is performing worse than the MPI one on all dataset sizes, achieving at maximum a speedup near to 8. In addition, it can be seen that the version with MPI + OMP achieves, with the same number of CPUs, greater speedup than OMP but worse than MPI. Finally, it is possible to have a clear opinion on the implemented versions and infer better the performances by looking at the efficiencies. Indeed, the solutions containing OMP have efficiencies all lower than 0.7, meaning that the parallelization with respect to the CPUs used is not much efficient. This is caused by what previously stated, namely that OMP in the implemented solution couldn't handle many operations and thus many of them were kept serial, limiting the use of the available CPUs. On the other hand, the MPI solution works well and almost with all dataset sizes and number of processes it achieves an efficiency greater than 0.7. Furthermore, with the dataset size of 0.5 it is possible to notice that it even achieves an efficiency greater than 1, demonstrating the effectiveness of the implemented master-slave architecture.



```
Time passed: 64.4511
KEY VALUE
10749   0.017881
11520   0.0125591
12341   0.0156215
13176   0.11803
13176 21137 0.0191696
13176 21903 0.0156684
13176 27966 0.0125986
13176 47209 0.0193914
15290   0.0121896
16759   0.011142
16797   0.0444655
16797 24852 0.0128254
17794   0.0226248
18465   0.0124562
19057   0.0233804
196 0.0111329
19660   0.0174461
19678   0.0101198
20114   0.0132388
20995   0.0102296
21137   0.0823308
21137 21903 0.0118617
21137 24852 0.0174676
21137 27966 0.0105332
21137 47209 0.0126891
21616   0.0226538
21709   0.0101988
21903   0.0752506
21903 24852 0.0159866
21903 47209 0.0108561
21938   0.0172585
22035   0.0185625
22825   0.0152152
22935   0.0352816
23909   0.0115373
```

Figure 2: Example of output given by the algorithm.

7 CONCLUSIONS

In this work the problem concerning the low performance of the Apriori algorithm on large datasets was addressed by introducing parallelization and considering both distributed and shared memory systems. In particular, the work focused on implementing a master-slave architecture through which the work could be divided and executed in parallel among different processes. Furthermore, to add another layer of parallelization, some operations that could have been parallelized without race conditions were implemented in order to be simultaneously performed by multiple threads.

Looking at the results of the experimental evaluation carried out, the MPI solution significantly outperforms the serial version of the algorithm by applying the master-slave architecture. On the other hand, the introduction of OMP in the solution didn't bring the desired results since many operations could not have been parallelized and thus the allocated CPUs were not fully exploited.

To conclude, the optimal proposed algorithm solves the problems derived from scanning the entire dataset at each iteration and keeping in memory the support of all candidates in the dataset, and does this by achieving good performances on every dataset size with great efficiency.

REFERENCES

[1] Instacart. 2018. Instacart Market Basket Analysis. https://www.kaggle.com/c/instacart-market-basket-analysis/overview.

[2] Sujith Mohan Velliyattikuzhi. 2012. Parallel implementation of Apriori algorithm and association of mining rules using MPI. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Velliyattikuzhi-Fall-2012-CSE633.pdf.

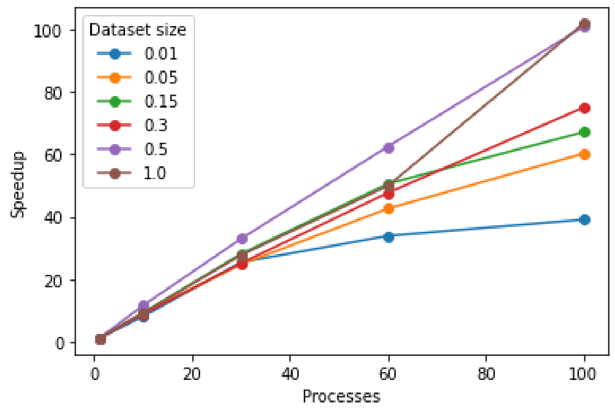


Figure 3: Speedup with MPI.

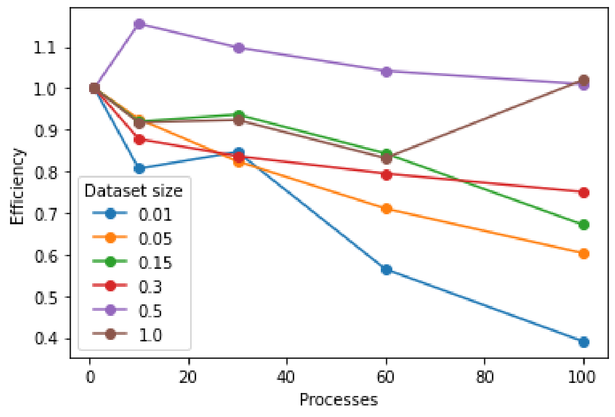


Figure 6: Efficiency with MPI.

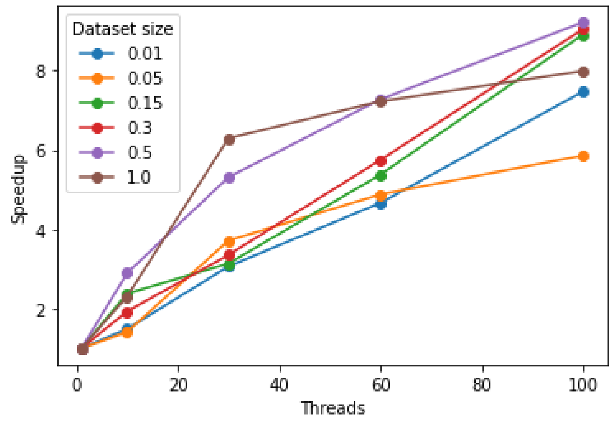


Figure 4: Speedup with OMP.

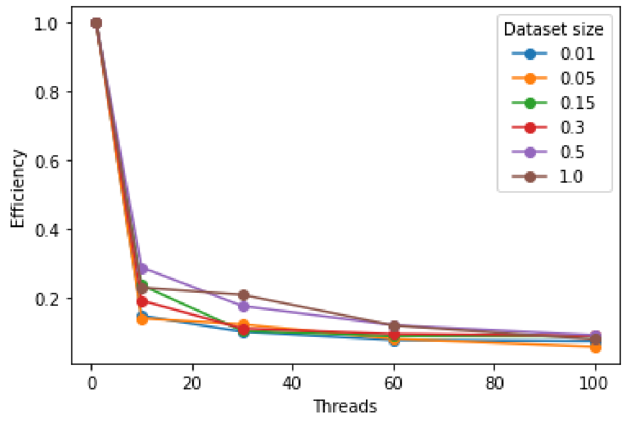


Figure 7: Efficiency with OMP.

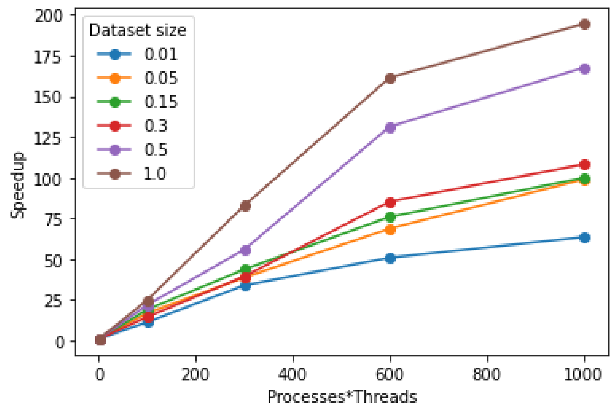


Figure 5: Speedup with MPI + OMP.

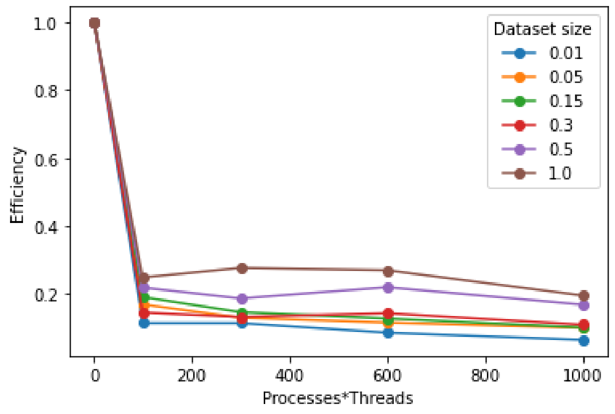


Figure 8: Efficiency with MPI + OMP.