Parallel Expectation-Maximization algorithm

Olha Khomyn[1], Davide Moletta[2]

Abstract

The goal of the High-Performance Computing for Data Science course project was to develop an MPI-based parallel implementation of the Expectation-Maximization algorithm for Gaussian points clustering. To achieve this goal we propose three implementations, the first utilizes a sequential approach, while the other two versions use a parallel one. Firstly, we present the serial version development and the algorithm structure. Secondly, we explain how the two parallel versions were implemented, what are their differences, pros and cons. Lastly, we compare the performances of the presented implementations and we discuss the findings as well as providing possible improvements. The code, results and executable are available in the GitHub repository [1].

[1] *olha.khomyn@studenti.unitn.it*
[2] *davide.moletta@studenti.unitn.it*

Introduction

Expectation-Maximization is an iterative algorithm, the goal of which is to estimate the hypothesis maximizing the log-likelihood of the data. Formally, it can described as follows: we are given a dataset made of an observed part $\mathbb{X}$ and an unobserved part $\mathbb{Z}$. We wish to estimate the hypothesis maximizing the expected log-likelihood for the data, with expectation taken over unobserved data [2]:

$$h^* = argmax_h \, E_z(\ln p(\mathbb{X}, \mathbb{Z}|h))$$

EM algorithm finds its usage for Gaussian Mixture Model to cluster the data into k Gaussians distributions. The goal is to estimate the mean and variance of each k Gaussians [2] following the steps described in Algorithm 1.

The complexity of the algorithm depends on the number of training examples that need to be clustered, the dimensions of each example and the number of Gaussians. In this project, we focus on the parallel implementation that can handle large amounts of training data. Furthermore, we present an implementation designed specifically to speed up the training process in case of high-dimensional data.

1. Sequential algorithm

In this section we explain how the sequential version works.

1.1 Initialization

Let $\mathbb{X}^{n \times d}$ be our matrix with the training examples that need to be clustered, where n is the number of training examples and d their feature dimension. The number of clusters is assumed to be unknown and is chosen empirically.

Before initializing the hypothesis and running the algorithm, we perform Z-score normalization, which transforms the dataset in such a way that the mean of each training example is 0 and the standard deviation is 1:

$$x' = \frac{x - \mu}{\sigma}$$

The mean, covariance and weights and then randomly initialized for each cluster. In order to preserve the non-singularity of the covariance matrix, we assign the values in range $(0,1)$ in the main diagonal, and $1e^{-6}$ everywhere else. This is an important step to make sure that the covariance matrix is invertible. For the mean vectors we simply randomly assign the values between 0 and 1. At the initialization stage, we assign the same weight to each cluster equal to $\frac{1}{k}$.

After the initialization is done, the algorithm starts with the steps described below.

1.2 Expectation step

The goal of the expectation step is to estimate the soft cluster assignment of each training example. The soft assignment means that we don't assign 1 if the example i belongs to the cluster j, and zero to all other clusters. The assignment is represented in terms of probability, thus for each training example, the sum of probabilities must sum up to 1.

In the expectation step, we iterate over the whole training set. At each step, we iterate over k clusters and, for each one, we estimate the Gaussian p.d.f. (probability density function) for the current training example given the current mean and covariance matrix for the examined cluster.

Since the Gaussian p.d.f. requires calculating the inverse of the covariance matrix, at some point in the algorithm we might encounter the singularity problem, making the matrix non-invertible. When this happens, we reset the mean and covariance matrix in the same way as during the initialization stage and continue with the algorithm. This approach was suggested by [3]: "We can hope to avoid the singularities by using suitable heuristics, for instance by detecting when a Gaussian component is collapsing and resetting its mean to a

Algorithm 1: Expectation-Maximization (EM)

```
Data: K - Number of Gaussians
Initialize the current hypothesis $h$ - mean, variance, weights

while $new\_Likelihood - old\_Likelihood > threshold$ do
    E-step: The expected probability that $x_i$ is generated by Gaussian $j$ assuming current hypothesis holds:
        $$p_{ij} = \frac{w_j p(x_i|\mu_j,\Sigma_j)}{\sum_{n=1}^{k} w_n p(x_i|\mu_n,\Sigma_n)}$$
    M-step: Estimate a new ML hypothesis given the estimation done in the E-step. Update current hypothesis:
        $$\mu_j = \frac{\sum_{i=1}^{n} p_{ij}x_i}{\sum_{i=1}^{n} p_{ij}}$$
        $$\sigma_{j,r,c} = \frac{\sum_{i=1}^{n} p_{ij}(x_{ir}-\mu_{i,r})(x_{ic}-\mu_{i,c})}{\sum_{i=1}^{n} p_{ij}}$$
        $$w_j = \frac{\sum_{i=1}^{n} p_{ij}}{\sum_{l=1}^{k}\sum_{i=1}^{n} p_{li}}$$
end
```

randomly chosen value while also resetting the covariance to some large value, and then continue with the optimization".

After the Gaussian p.d.f. has been estimated, we multiply it by its weight and do the same for every cluster. At the end, the probability of example i belonging to Gaussian j is given by:

$$p_{ij} = \frac{N(i,j)w_j}{\sum_{l=1}^{k} N(i,l)w_l}$$

$$N(i,j) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} exp(-\frac{1}{2}(x_i-\mu_j)^T\Sigma_j^{-1}(x_i-\mu_j))$$

1.3 Maximization step

The maximization step consists of estimating the new values for mean, covariance and weights. Since sum of probabilities is used to update each hypothesis, we estimate it separately for each cluster j, to avoid computing it three times.

$$p_j = \sum_{i=1}^{n} p_{ij}$$

After that we estimate the numerator parts of mean, covariance and divide them by the sum of probabilities estimated before.

$$\mu_j = \frac{\sum_{i=1}^{n} p_{ij}x_i}{p_j}$$

$$\sigma_{j,r,c} = \frac{\sum_{i=1}^{n} p_{ij}(x_{ir}-\mu_{i,r})(x_{ic}-\mu_{i,c})}{p_j}$$

$$w_j = \frac{p_j}{\sum_{j=1}^{k} p_j}$$

Using this approach we estimate the new hypothesis maximizing the expected log-likelihood of the data.

1.4 Convergence check

Since the aim of the EM algorithm is to estimate the hypothesis that maximizes the likelihood of the data, we compute the log-likelihood after each iteration of the algorithm, using the following formula:

$$l(\Theta|X) = \sum_{i=1}^{n} log(\sum_{l=1}^{k} w_l p(x_i|\mu_l,\Sigma_l)$$

If the log-likelihood is improving, the algorithm is working and the data is being clustered accordingly. In case the log-likelihood does not change for 5 consecutive iterations, we assume that the algorithm has converged and we stop the training. We added a maximum number of iterations as an additional parameter to ensure that the program is able to terminate in case the algorithm fails to converge.

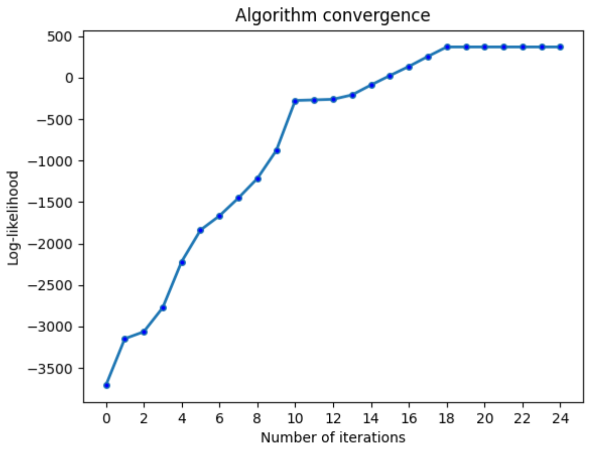

Figure 1. Log-likelihood results for N=800, K=4, D=3

2. State of the art of the parallel design

Before developing our parallel versions we have done some research on the state of the art of this algorithm and its parallel implementations.

With our research, we discovered that most of the proposed implementations work by distributing the computations of the E-step and M-step across multiple processes, specifically:

- **E-step**: the dataset is divided among the processes and, since the expectation for each component can be computed independently, each process can compute its part of the E-step. In the end, the results are combined and distributed;

- **M-step**: similarly to the E-step the update of parameters can be distributed as it does not rely on other components.

In essence, parallelizing the EM algorithm leverages the fact that it's possible to break down the computations into independent tasks. This type of parallelization was proposed by [4].

The most interesting implementation is proposed by [5] and it's based on MPI parallel computations. The implementation sees a master process and many slave processes that interact in the following way:

- **Master process**: the master process is in charge of reading the arguments, standardising data and populating the matrices. After this step, the master process sends jobs to all the available slaves and waits for results. This goes on until the algorithm converges, in such case the master process sends a kill message to all the slaves;
- **Slave processes**: slave processes simply wait for the job by the master and, once they get it, they execute the required task and send back the result to the master.

This solution is slightly similar to ours. We do differentiate between master and normal processes but in our case the master process is just responsible for operation that are performed by a single process such as initializing the matrices. During the parallel execution instead, the master process act as a normal process with the difference that it is used to gather data. However, the implementation proposed by [5] would allow us to differentiate the task that the processes will perform.

Other implementations are provided, for example, [6] and [7] provide a solution for the parallel EM algorithm specifically tailored for GPUs that utilize NVIDIA's CUDA architecture. This version is restructured to exploit data parallelism, a key feature of CUDA, enabling faster computations on GPUs.

3. Parallel implementation 1

We developed a parallel implementation of the EM algorithm that is able to efficiently cluster large datasets. As mentioned before, let's suppose we have an $\mathbb{X}^{n\times d}$ matrix, where n is the number of training examples and d is the feature vector dimensionality. Let p be the number of available processors. In case n is evenly divisible by p, then each process receives $\frac{p}{n}$ rows. Otherwise, the remaining rows are distributed among the processes.

We followed the parallelization approach similar to [8], however, they assume that at the beginning of the execution, each process already holds its own sub-matrix, for example by reading its portion from a file. In our case, the master process (the process with id 0) is in charge of reading the data from an external file. Then it divides the rows and scatters sub-matrices to each process. The master process is also in charge of initializing the mean, covariance and weights matrices.

Note that when we say matrices, we refer to their flattened 1D array versions.

The parallel implementation of the EM algorithm works as follows.

Parallel EM steps:

- **Initialization step**: the master process reads the dataset, divides and distributes the rows, and initializes mean, covariance and weights. The distribution of data is performed using *MPI_Scatterv* for the input matrix and with *MPI_Bcast* for mean, covariance and weights.

- **Log-likelihood step**: each process computes the log-likelihood on its sub-matrix, then we use *MPI_Allreduce* to estimate the total log-likelihood and distribute the result to each process.

- **E-step**: each process computes the soft cluster assignment of its sub-matrix as depicted in the sequential implementation.

- **M-step**:

  - Each process computes the sum of probabilities of assignments for each cluster on its sub-matrix. *MPI_Reduce* is used to calculate the total sum and the result is sent to the master process.

  - Each process computes the numerator of the mean on its sub-matrix. *MPI_Reduce* is called to calculate the total sum gathered from all the processes and the result is sent to the master process. The master process updates the mean values using the result gathered from the processes and the sum of probabilities estimated before.

  - Each process computes the numerator of the covariance on its sub-matrix. *MPI_Reduce* is called to calculate the total sum gathered from all the processes and the result is sent to the master process. The master process updates the covariance values using the result gathered from the processes and the sum of probabilities estimated before.

  - Master process updates the values of the weights using the sum of probabilities estimated before.

  - Master process broadcasts updated values of mean, covariance and weights to all the processes with *MPI_Bcast*.

- **Log-likelihood step**: each process estimates the new log-likelihood of the data on its sub-matrix after each M-step. *MPI_Allreduce* is used to estimate the total log-likelihood. If the log-likelihood does not change for more than 5 consecutive iterations or the maximum number of iterations is reached the algorithm stops, otherwise, it keeps iterating going back to the E-step.

4. Parallel implementation 2

After the development and testing of the first implementation, we developed the second one with a different approach. Also, we kept in mind some weak points of the first implementation and tried to fix them.

The general structure of the second implementation is similar to the first one with some differences listed below:

- **Data reading**: with this version, we implemented a parallel file reading procedure. Each process computes its number of samples as before and then computes the following:

  $$start\_index = process\_rank \times samples$$
  $$end\_index = start\_index + samples - 1$$

  In this way, each process will have its starting line and its ending line. After this, each process starts reading from the input file in a parallel fashion taking into consideration only the lines for which it is responsible;

- **Standardization**: Since each process reads its own part of the input none of them is able to compute the global mean and standard deviation. To solve this problem we changed the behaviour in the following way:

  - Each process computes the mean of its samples (local mean);
  - *MPI_Allreduce* is called to distribute the global mean to each process;
  - Each process computes the standard deviation of its samples (local standard deviation);
  - *MPI_Allreduce* is called to distribute the global standard deviation to each process;
  - Each process computes the standardization process.

  In this way the standardization works as expected even if the input is distributed at the beginning;

- **Parallel determinant**: Since the EM algorithm requires computing the determinant of the covariance matrix a lot of times during the execution and the determinant has a complexity of $\mathcal{O}(n^3)$ (given that covariance is $D \times D$ we have $\mathcal{O}(D^3)$ for each cluster) we focused on speeding up this process. To do so, we utilized OpenMP to parallelize part of the computation of the determinant, thus saving time with high dimensional samples. The functioning of the parallel determinant is depicted in Algorithm 2.

The remaining part of the implementation works as depicted in Section 3. It is possible to see this implementation as the same version but optimized for high-dimensional datasets. In fact, for small datasets, we do not gain time neither from the parallel read nor from the parallel determinant which for low-dimensional points is slower than the serial determinant due to the threads overhead. However, if we start feeding to our algorithm larger and larger datasets this version will eventually outperform the first implementation.

5. Data generation

To test our implementations we developed a data generator with Python. The script generates a random μ and σ for each Gaussian and creates random points for each of them. The points are then saved in a csv file.

With this program, we were able to create a sample dataset for testing purposes and the final datasets on which we have made the evaluation. The datasets used to gather the time consumption of each implementation are the following:

- Small dataset: $N = 250000$, $K = 5$, $D = 4$;
- Medium dataset: $N = 625000$, $K = 5$, $D = 4$;
- Large dataset: $N = 1250000$, $K = 5$, $D = 4$;
- Six-dimensional dataset: $N = 20000$, $K = 4$, $D = 6$;
- Eight-dimensional dataset: $N = 4000$, $K = 4$, $D = 8$.

Where N is the number of total samples and K and D are the number of Gaussian and the dimensions of each sample respectively.

6. Evaluation

All the tests were conducted on a range of cores between 1 and 64 distributed in the following configurations: *Packed*, *Packed exclusive*, *Scattered* and *Scattered exclusive*.

6.1 Parallel implementation 1

It is possible to see in Figure 3 that the first implementation showed a similar trend in all the proposed configurations and input sizes. Overall, the algorithm's execution time kept decreasing as the number of cores increased with some variations in the *Scattered* configuration. However, upon reaching a high core count, the curve reached a plateau indicating a reduction in performances.

Following this execution time we can see in Figure 4 that the speedup increases as the number of cores grows. Also, we can see that the speedup mostly shows a sub-linear trend especially in the in the *Scattered* and *Scattered exclusive* configurations.

Instead, the efficiency shows an inverted trend as it decreases when the number of cores grows (Figure 5). In the *Packed* and *Packed exclusive* configurations the algorithm showed promising values for efficiency staying above the community-agreed threshold of 70% most of the time. On the other hand, in the *Scattered* and *Scattered exclusive* configurations the efficiency dropped quicker reaching low values.

In Figure 9, it is possible to see that this implementation suffers from high-dimensional points. In fact, if we feed such points to the algorithm, the execution time becomes very high even for datasets that consist of a small number of samples, such as the eight-dimensional dataset. As we have said before, this is due to the determinant computation as well

Algorithm 2: Parallel determinant with OpenMP

```
Function Determinant(matrix, size):
    if $size > 3$ then
        $new\_matrix = matrix$
        $det = factorize(new\_matrix, size)$
        if $det \neq 0$ then
            parallel for with $*$ reduction
            for $i = 0$ to $size - 1$ do
                $det* = new\_matrix[i * size + i]$
            end
        end
    else
        compute determinant normally if $size \leq 3$
    end
    return $det$
Function Factorize(mat, s):
    $parity = 1$
    for $k = 0$ to $s - 1$ do
        $max = abs(mat[k * s + k])$
        $index = k$
        for $i = k$ to $s - 1$ do
            if $abs(mat[i * s + k]) > max$ then
                update max and index
            end
        end
        if $max = 0$ then
            return 0
        end
        if $index \neq k$ then
            $parity = -parity$
            parallel for
            for $i = 0$ to $s - 1$ do
                swap $[k * s + i]$ and $[index * s + i]$
            end
        end
        parallel for
        for $i = k + 1$ to $s - 1$ do
            $mat[i * s + k] / = mat[k * s + k]$
        end
        parallel for
        for $i = k + 1$ to $s - 1$ do
            $mat_{ik} = mat[i * s + k]$
            for $j = k$ to $s - 1$ do
                $mat[i * s + j] - = mat_{ik} * mat[k * s + j]$
            end
        end
    end
    return $parity$
```

as the inverse of matrix, which have a high time complexity for high-dimensional matrices.

All the tests result are collected in Table 1, Table 2, Table 3 and Table 7.

These data clearly show that the algorithm is not fully parallelizable and that communication overhead is holding back the performance especially in scattered configurations.

6.2 Parallel implementation 2

We can see in Figure 6 that this version shows a similar trend to the first counterpart but with more stable results in the *Scattered* and *Scattered exclusive* configurations.

Overall, this version has higher execution times, reaches a plateau sooner and, at around 32 cores, the speedup starts decreasing (Figure 7).

Moreover, if we look at Figure 8, we can see that in each configuration the efficiency drops very quickly and always reaches around 0.2 when the algorithm is executed with 64 cores. As we have said before, this may be due to the OpenMP overhead. Since points have four dimensions, the process of spawning threads during the parallel determinant computation leads to a loss in performance by the algorithm.

Instead, in the six and eight-dimensional datasets this implementation outperforms the first one in a significant way. In fact, by looking at Figure 9, we can see that the second version is constantly faster than the first one in each core configuration. These numbers show that the second implementation strategy addressed the high-dimensionality problem of the first one. However, since, we scaled down the problem to meet the limits of the first approach, this implementation has no room for speedup and reaches a plateau instantly resulting in poor performances in the evaluation datasets. This comparison has been made just to show the execution time gain for high-dimensional datasets but, if we were to increase the dataset size, the second implementation would eventually increase its efficiency and reach the plateau later.

All the tests result are collected in Table 4, Table 5, Table 6 and Table 7.

6.3 Scalability

Based on our results, we are confident in saying that neither the first nor the second implementation are strongly scalable. As we have said before, not every part of the algorithm is parallelizable and this leads to an eventual loss in performance if we keep increasing the number of cores while keeping the problem size unchanged.

6.4 File reading

Since the second implementation provides a solution for parallel file reading, we gathered data about this part of the execution (Figure 2). Overall, the parallel reading is faster almost every time compared to the serial reading. However, even with the biggest dataset (consisting of over one million samples), we noticed variations in the order of deciseconds, which are not significant results. This part of the second parallel implementation may come in hand with datasets that consist of many millions, if not billions, of samples. In any case, it will provide a limited speedup compared to the algorithm execution time.

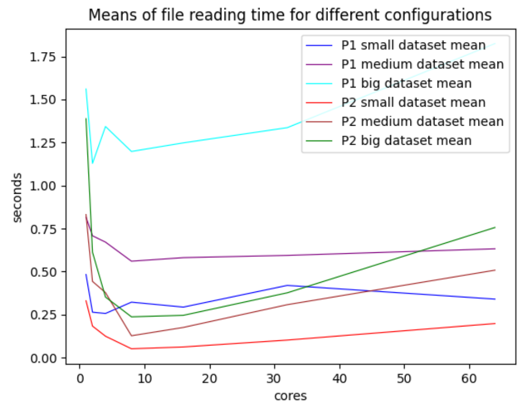

Figure 2. File reading time

7. Conclusions

During this report, we explained how we provided two parallel solutions to the Expectation-Maximization algorithm. In the first part, we described how the algorithm works and what are the current parallel implementations. Then, we depicted how our implementations work, how we created them, and which are their weak points. After that, we saw the difference between the two implementations and described the parallelization strategies utilised.

As we have seen in Section 6, the algorithms effectively provide better time performances. While the first version provided better speedup and showed good efficiency, we saw that it suffers from high-dimensional datasets. Instead, the second one addresses this problem but suffers a higher overhead due to the OpenMP threads.

Overall, the provided versions are capable of reducing the execution time and are effective at parallelizing the EM algorithm in each core configuration with some variations in the *Scattered* and *Scattered exclusive* ones.

Finally, with this project, we discovered the potentiality of parallelization strategies to address time-demanding tasks.

7.1 Credits

The code for the inverse of matrix is derived from [9] while the code for the parallel determinant with OpenMP is derived from [10].

8. Future works

Some areas of improvements with respect to the clustering accuracy include improving the initialization of the hypothesis, which is currently initialized with random values. Some studies suggest performing the initial clustering using k-means clustering, initialize the hypothesis using the produced results and then run the expectation-maximization algorithm as usual.

Secondly, more attention should be given to make sure that the covariance matrix does not collapse, instead of re-initializing it when it happens.

With regards to the performance of the algorithm, we may need to optimize the communication between the processes in order to minimize the communication overhead.

Lastly, we may parallelize the computation of the inverse of matrix which requires significant time, similarly to the determinant.

References

[1] “GitHub repository,” https://github.com/High-performance-computing-unitn/expectation-maximization.

[2] “Unsupervised Learning,” https://disi.unitn.it/~passerini/teaching/2023-2024/MachineLearning/slides/21_clustering/talk.pdf.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] S. X. Lee, K. L. Leemaqz, and G. J. McLachlan, “A Simple Parallel EM Algorithm for Statistical Learning via Mixture Models,” in *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2016, pp. 1–8.

[5] P. McNicholas, T. Murphy, A. McDaid, and D. Frost, “Serial and parallel implementations of model-based clustering via parsimonious gaussian mixture models,” *Computational Statistics & Data Analysis*, vol. 54, no. 3, pp. 711–723, 2010, second Special Issue on Statistical Algorithms and Software. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167947309000632

[6] N. S. L. P. Kumar, S. Satoor, and I. Buck, “Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA,” in *2009 11th IEEE International Conference on High Performance Computing and Communications*, 2009, pp. 103–109.

[7] “Parallel implementation of Expectation-Maximisation algorithm for the training of Gaussian Mixture Models,” https://ri.ufs.br/jspui/handle/riufs/1764.

[8] W.-C. Chen, G. Ostrouchov, D. Pugmire, P. Wehner, and M. Wehner, “A parallel em algorithm for model-based clustering applied to the exploration of large spatio-temporal data,” *Technometrics*, vol. 55, no. 4, pp. 513–523, 2013. [Online]. Available: http://www.jstor.org/stable/24587004

[9] “Inverse of matrix,” https://www.geeksforgeeks.org/adjoint-inverse-matrix/.

[10] “Parallel determinant with OpenMP,” https://matematicamente.it/forum/viewtopic.php?f=15&t=145130.

Table 1. Small dataset Parallel 1 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	248	1,0	1,0	199	1,0	1,0	600	1,0	1,0	274	1,0	1,0
2	124	2	1,0	108	1,8	0,92	464	1,3	0,64	127	2,15	1,0
4	62	4	1,0	50	3,9	0,99	311	1,9	0,48	61	4,5	1,1
8	34	7,3	0,91	35	5,7	0,71	272	2,2	0,28	29	9,5	1,1
16	18	13,8	0,86	16	12,4	0,78	174	3,4	0,22	15	18,3	1,1
32	8	31	0,97	10	19,9	0,62	39	15,4	0,48	11	24,9	0,77
64	8	31	0,48	5	39,8	0,62	28	21,4	0,33	5	54,8	0,86

Table 2. Medium dataset Parallel 1 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	541	1,0	1,0	540	1,0	1,0	513	1,0	1,0	627	1,0	1,0
2	251	2,2	1,0	249	2,2	1,0	365	1,4	0,7	290	2,2	1,0
4	126	4,3	1,0	126	4,3	1,0	306	1,7	0,42	206	3,0	0,76
8	64	8,5	1,0	75	7,2	0,9	178	2,9	0,36	139	4,5	0,56
16	31	17,5	1,0	37	14,6	0,91	81	6,3	0,4	83	7,6	0,47
32	19	28,5	0,89	24	22,5	0,7	61	8,4	0,26	49	12,8	0,4
64	10	54,1	0,85	11	49,1	0,77	27	19	0,29	37	16,9	0,26

Table 3. Big dataset Parallel 1 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	1086	1,0	1,0	1024	1,0	1,0	1064	1,0	1,0	1025	1,0	1,0
2	455	2,4	1,19	649	1,6	0,79	521	2,0	1,0	691	1,5	0,74
4	250	4,3	1,0	324	3,2	0,79	302	3,5	0,88	436	2,4	0,59
8	140	7,8	0,97	176	5,8	0,72	179	5,9	0,74	225	4,6	0,57
16	68	16	1,0	72	14,2	0,89	75	14,2	0,89	164	6,3	0,39
32	33	32,9	1,0	40	25,6	0,8	64	16,6	0,52	61	16,8	0,53
64	21	51,7	0,80	26	39,4	0,62	68	15,6	0,24	50	20,5	0,32

Table 4. Small dataset Parallel 2 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	527	1,0	1,0	583	1,0	1,0	415	1,0	1,0	491	1,0	1,0
2	373	1,4	0,7	364	1,6	0,8	173	2,4	1,19	298	1,7	0,82
4	124	4,3	1,0	132	4,4	1,1	137	3,0	0,75	184	2,7	0,66
8	55	9,6	1,2	74	7,9	0,98	86	4,8	0,6	55	8,9	1,11
16	51	10,3	0,65	46	12,7	0,79	46	9,0	0,56	36	13,6	0,85
32	29	18,2	0,57	26	22,4	0,7	29	14,3	0,48	28	17,5	0,54
64	44	12	0,19	49	11,9	0,18	17	24,4	0,38	42	11,7	0,18

Table 5. Medium dataset Parallel 2 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	975	1,0	1,0	922	1,0	1,0	906	1,0	1,0	898	1,0	1,0
2	542	1,8	0,89	531	1,7	0,87	527	1,7	0,86	520	1,7	0,86
4	297	3,3	0,82	227	4,0	1,0	255	3,6	0,89	234	3,8	0,96
8	163	6,0	0,74	101	9,1	1,1	211	4,3	0,54	190	4,7	0,59
16	111	8,8	0,55	74	12,5	0,78	117	7,7	0,48	112	8,0	0,5
32	67	14,6	0,46	69	13,4	0,41	53	17,0	0,53	57	15,8	0,49
64	86	11,3	0,18	84	11,0	0,17	65	13,9	0,22	62	14,5	0,23

Table 6. Big dataset Parallel 2 evaluation values

Cores	Packed			Packed exclusive			Scattered			Scattered exclusive		
	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency	Time (s)	Speedup	Efficiency
1	2203	1,0	1,0	2002	1,0	1,0	1912	1,0	1,0	1890	1,0	1,0
2	1127	1,9	0,97	1088	1,8	0,92	906	2,0	1,0	979	1,9	0,97
4	541	4,0	1,0	562	3,6	0,89	497	3,8	0,96	540	3,5	0,88
8	373	5,9	0,74	310	6,5	0,81	291	6,6	0,82	312	6,1	0,76
16	186	11,8	0,74	202	9,9	0,62	229	8,4	0,52	355	5,3	0,33
32	106	20,8	0,65	113	17,7	0,55	116	16,5	0,52	221	8,6	0,28
64	195	11,3	0,18	206	9,7	0,15	83	23,0	0,36	147	12,9	0,2

Table 7. High-dimensional datasets evaluation values

Cores	Six-dimensional dataset		Eight-dimensional dataset	
	Parallel 1 time (s)	Parallel 2 time (s)	Parallel 1 time (s)	Parallel 2 time (s)
2	304	17	4347	4,6
4	166	15	2032	4,4
8	83	12	1476	2,3
16	42	7	705	2,4
32	20	4	415	0,7
64	19	8	114	1

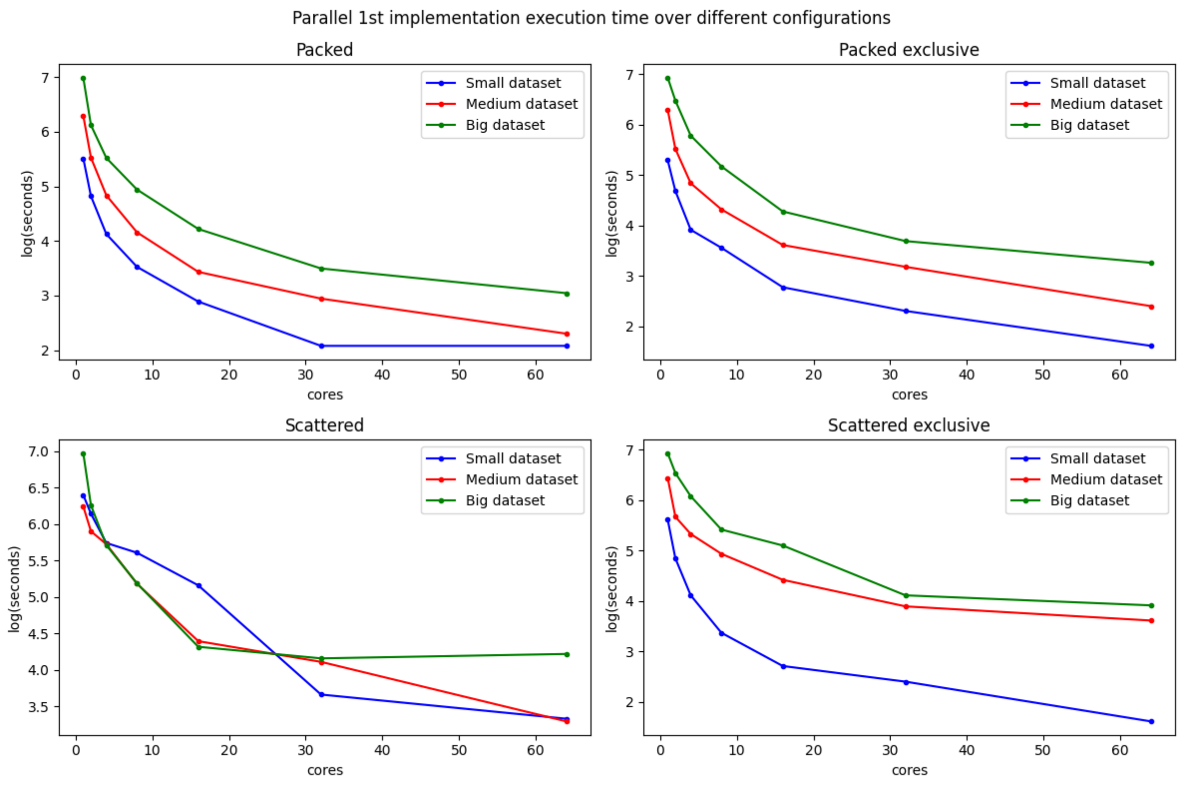

Figure 3. Parallel first: Logarithm of Execution time over different configurations

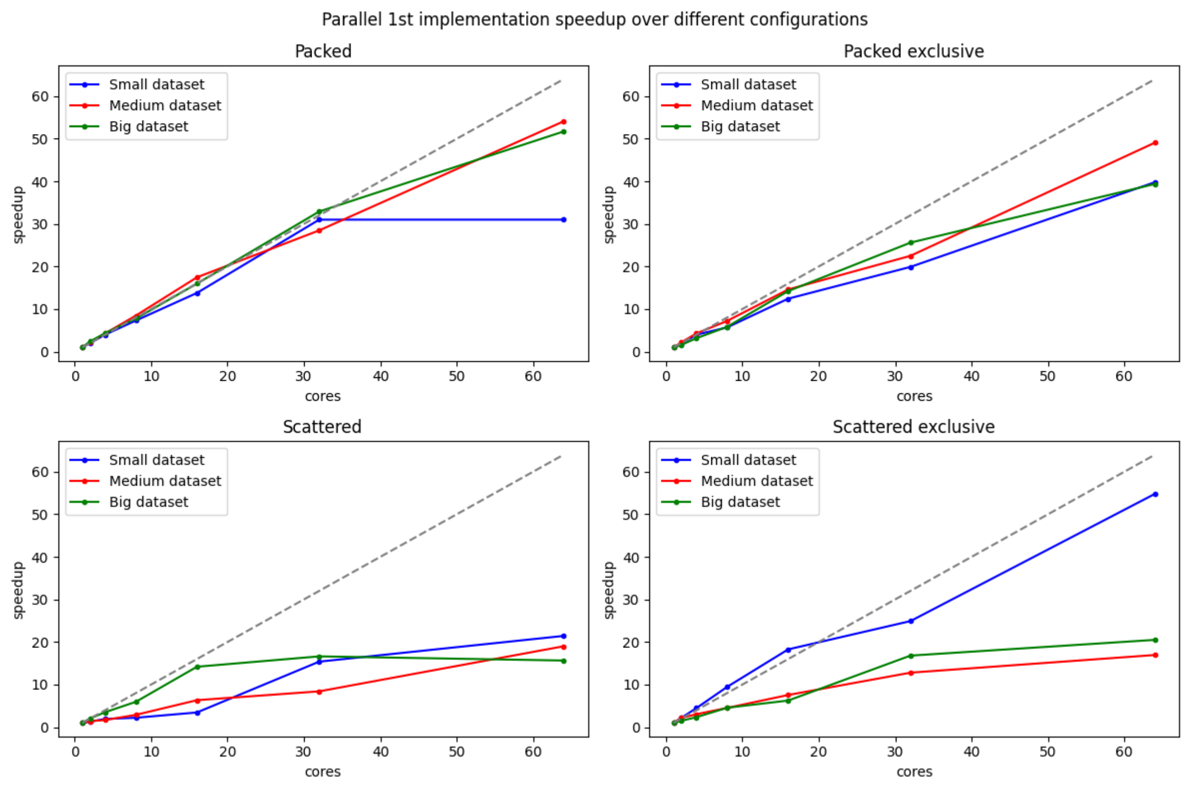

Figure 4. Parallel first: Speedup over different configurations

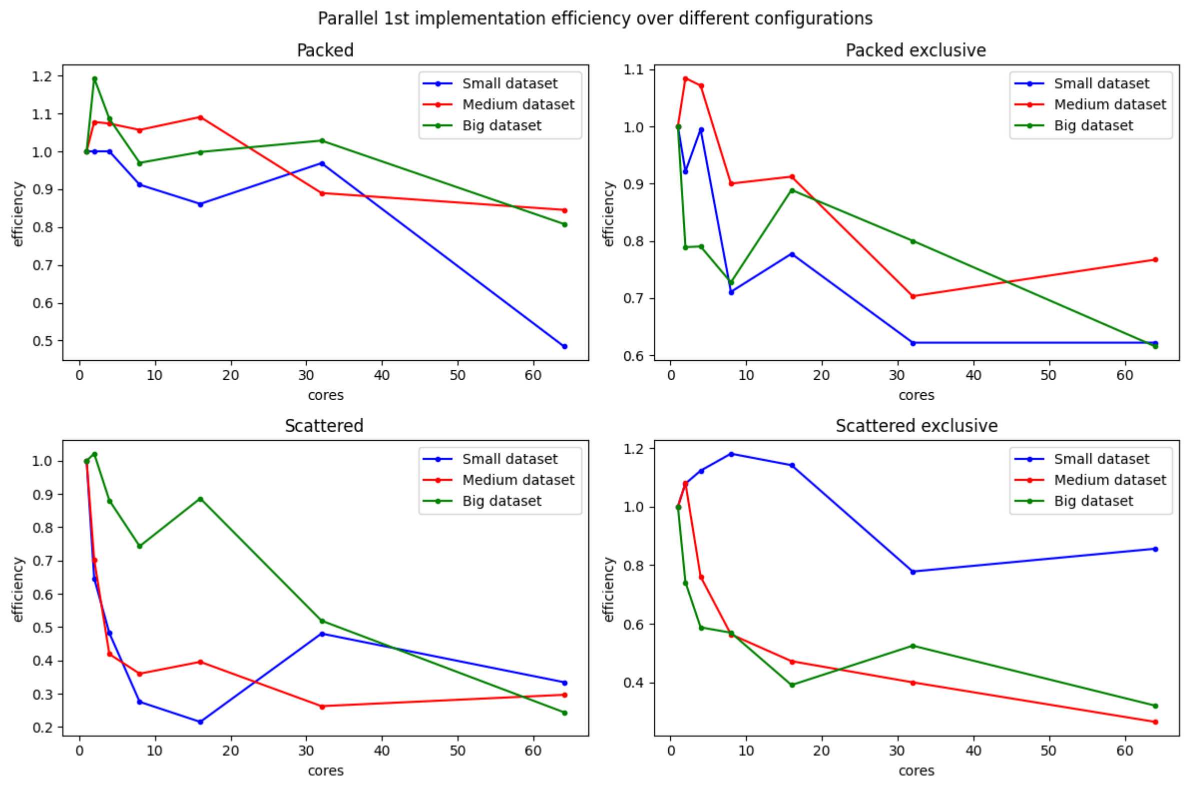

Figure 5. Parallel first: Efficiency over different configurations

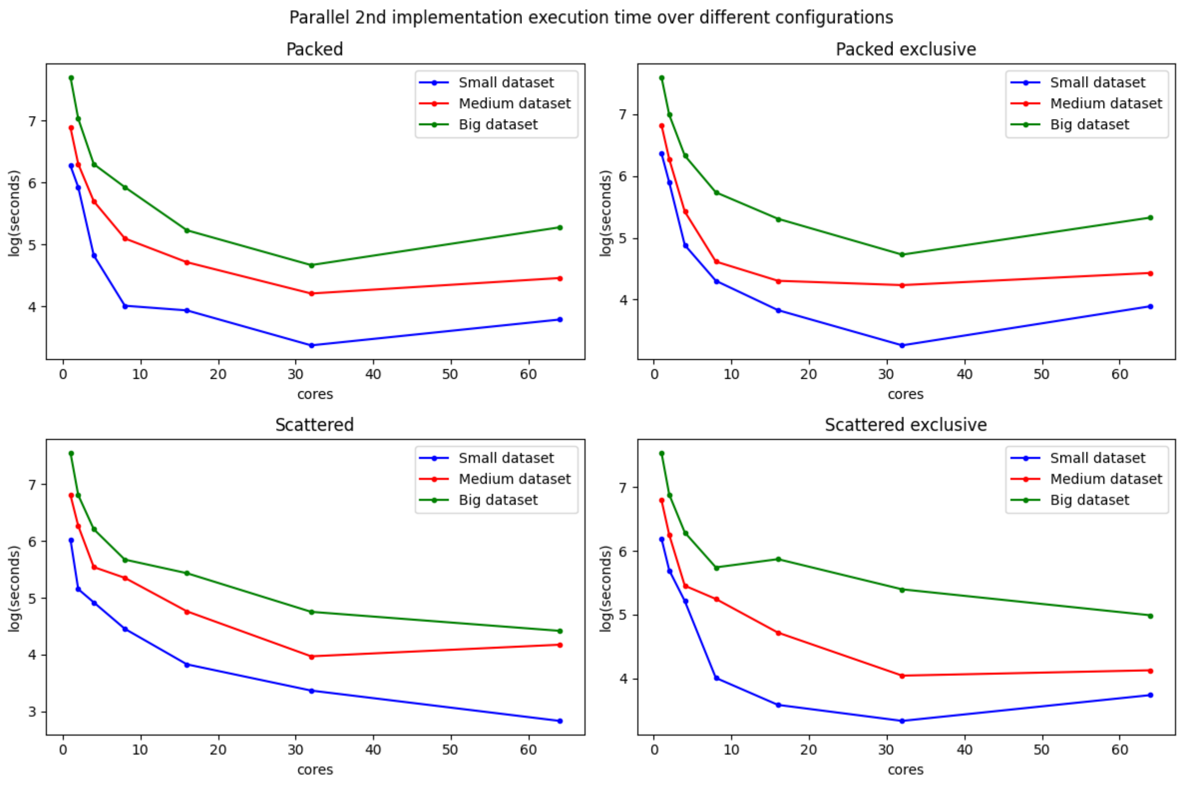

Figure 6. Parallel second: Logarithm of Execution time over different configurations

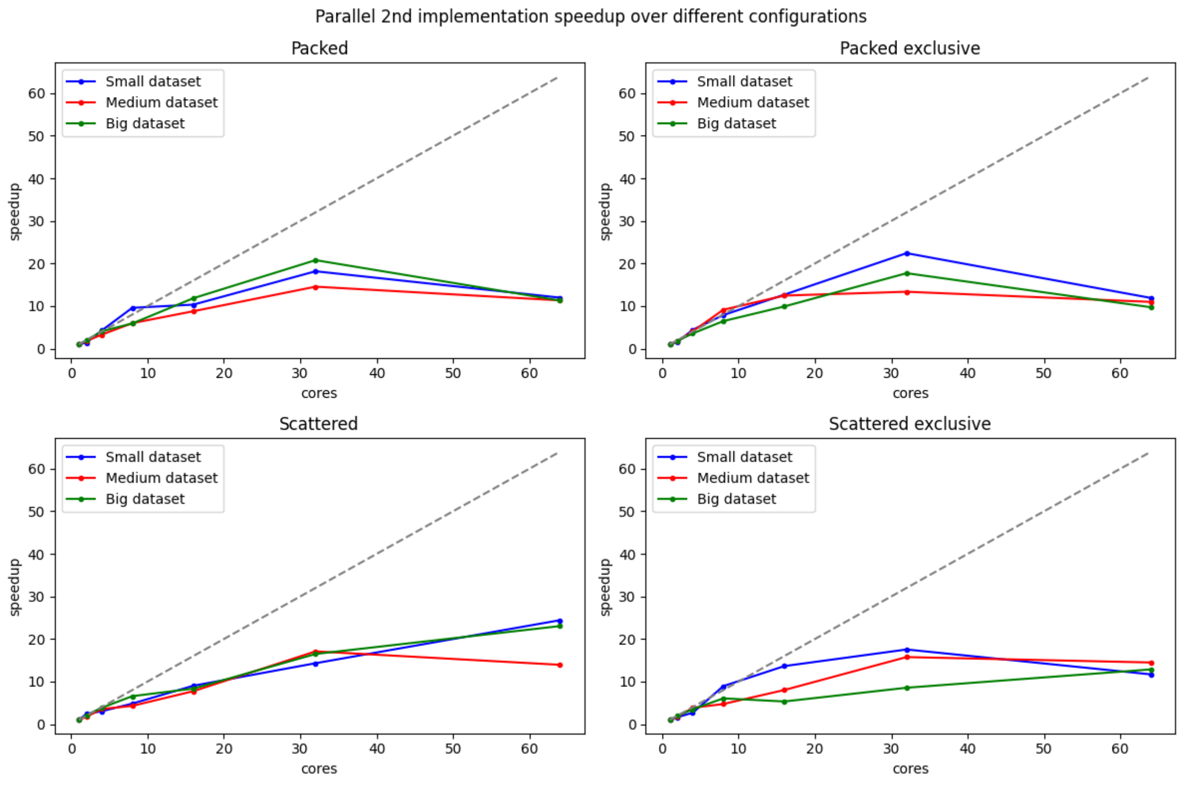

Figure 7. Parallel second: Speedup over different configurations

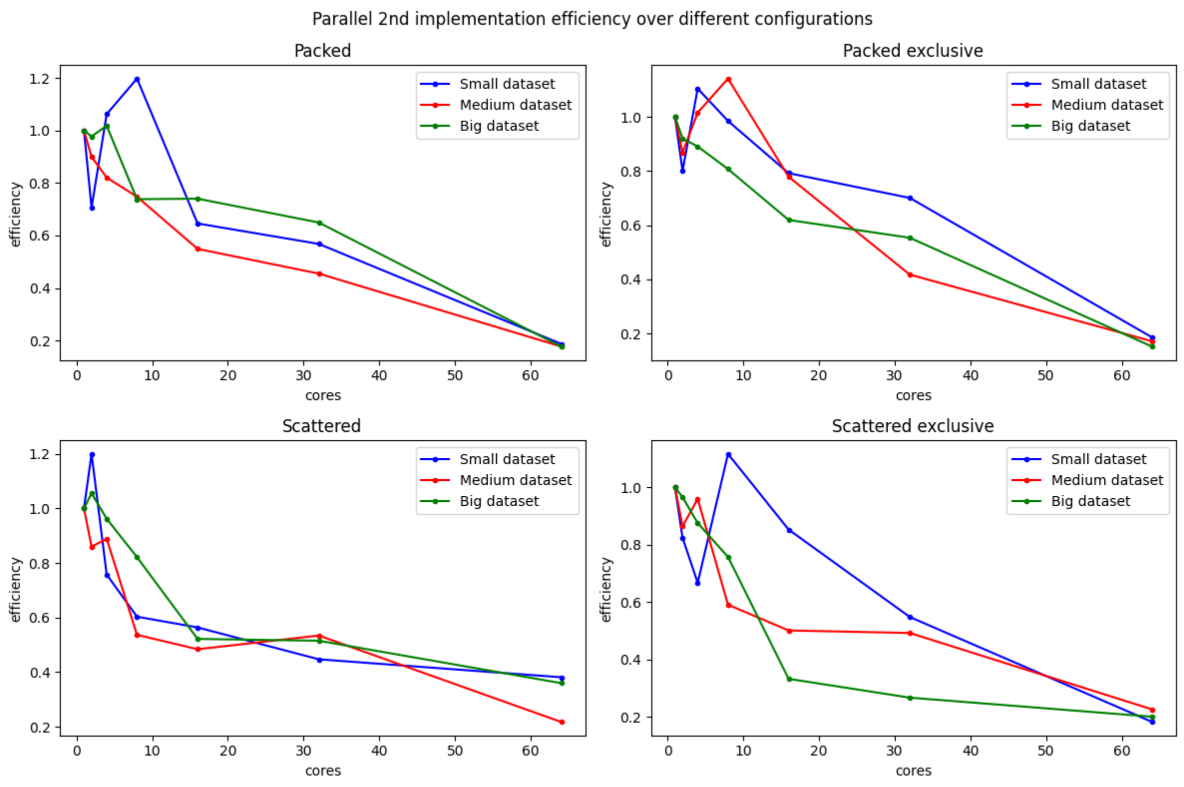

Figure 8. Parallel second: Efficiency over different configurations

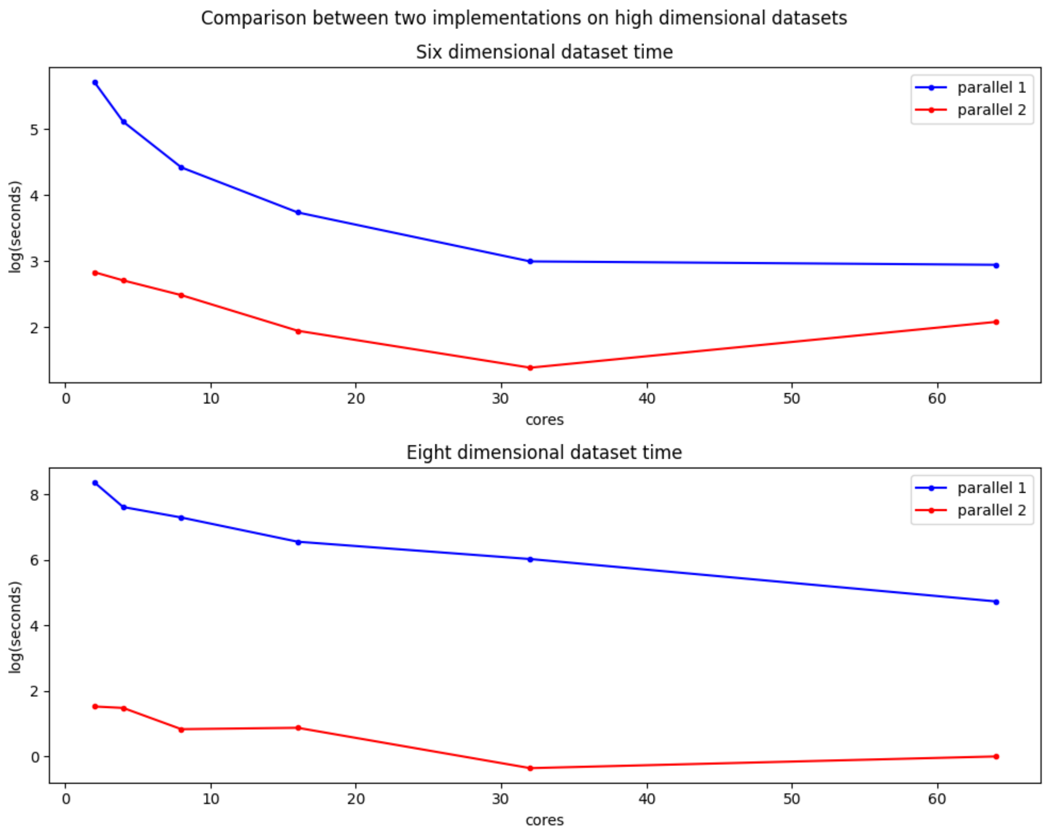

Figure 9. Execution time for high-dimensional datasets