Cloud Computing and Data Science
https://ojs.wiserpub.com/index.php/CCDS/

Research Article

A *K*-Means, Ward, and DBSCAN Repeatability Study

Anthony Bertrand, Engelbert Mephu Nguifo, Violaine Antoine, David R. C. Hill*

University Clermont Auvergne, Clermont Auvergne INP, ENSM St Etienne, CNRS, LIMOS, Clermont-Ferrand, 63000, France
E-mail: david.hill@uca.fr

Received: 22 December 2025; **Revised:** 27 February 2026; **Accepted:** 18 March 2026

Abstract: Reproducibility is essential in machine learning because it ensures that a model or experiment yields the same scientific conclusion. For specific algorithms, repeatability with bitwise identical results is also a key for scientific integrity because it allows debugging. We decomposed several very popular clustering algorithms: *K*-Means, Density-Based Spatial Clustering of Applications with Noise (DBSCAN), and Ward into their fundamental steps, and we identify the conditions required to achieve repeatability at each stage. We use an implementation example with the Python library scikit-learn to examine the repeatable aspects of each method. Our results reveal non-repeatable behavior with *K*-Means when the number of OpenMP threads exceeds two. This work aims to raise awareness of this issue among both users and developers, encouraging further investigation and potential fixes.

***Keywords*:** repeatability, reproducibility, clustering methods, *K*-Means, Density-Based Spatial Clustering of Applications with Noise (DBSCAN), Ward method

1. Introduction

Experimental science relies on the reproducibility of experiments. When results are reproducible, they are analyzed and processed to draw scientific conclusions. It is therefore important to ensure that our results are similar from one experiment to the next. Reproducibility issues introduced by the computer science stack of tools (hardware, software, libraries and execution environments) and procedures are now affecting all scientific disciplines [1]. When run under the same conditions, Machine Learning (ML) and Artificial Intelligence (AI) models or experiments should give the same scientific conclusions. This requirement is critical for several reasons: scientific integrity, model validation and comparison but also for debugging and software development. In the latter, we also need bitwise identical traces and results. Indeed, developers need deterministic behavior to identify implementation bugs, optimize performance, and ensure consistent behavior across platforms and hardware configurations.

However, with modern machine learning frameworks which hide technical details, it becomes more and more difficult to control the pseudorandom sources hidden in stochastic machine learning models [2]. Parallelization, aimed at accelerating calculations, introduces additional challenges for reproducibility. To overcome repeatability issues, some researchers perform multiple replications of the same experiment without realizing the bias they introduce when they do not properly manage parallel random sources. However, this flawed approach is common, and many scientists present just biased statistics on their results. Before working on reproducibility, we often need to have repeatable numerical experiments. Bitwise identical results from run to run on the same machine and environment with the same code

Copyright ©2026 David R. C. Hill, et al.
DOI: https://doi.org/10.37256/ccds.7220269455
This is an open-access article distributed under a CC BY license
(Creative Commons Attribution 4.0 International License)
https://creativecommons.org/licenses/by/4.0/

and input data are mandatory for debugging. This corresponds to the Association for Computing Machinery (ACM) definition of repeatability with a stated precision equal to 0 (identical results). With modern machine learning, we often have to fight to keep this precious repeatability.

In this paper, we focus on clustering methods. They are data analysis techniques that group data, based on shared characteristics. They fall under unsupervised learning, where the machine independently identifies patterns without predefined labels. According to [3], we can classify clustering methods in two categories: Hierarchical and Partitional. Hierarchical clustering methods build cluster iteratively with two approaches. The first approach called Agglomerative clustering uses a bottom-up approach, starting with every data as its own cluster, and merging similar clusters iteratively until only one cluster remains. The second approach, called divisive clustering, uses a top-down approach, starting with one cluster and dividing it into multiple ones until each object forms a cluster. These two approaches end up building dendrograms. Partitional clustering optimizes objective functions. Finally, we retain three methods: (1) Distance-based clustering—which often uses the Euclidean distance between data points; (2) Model-based clustering—which tries to optimize a mathematical model; (3) Density-based clustering—which uses density function to find clusters.

One of the most used Python libraries for machine learning is scikit-learn [4]. It is an open-source machine learning library. It offers many useful tools for AI developers, such as generation of synthetic data, supervised and unsupervised learning algorithms and metrics to evaluate created models. Due to its versatility and ease of use, this library is widely used in ML.

In this paper, we investigate repeatability issues with three well-known algorithms from three different clustering methods: K-Means for distance-based clustering, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) for density-based clustering, and the Ward method for agglomerative clustering. These algorithms are very popular and cover different categories of the taxonomy above. We organize our paper as follows: Section 2 gives definitions of reproducibility, repeatability, and their importance in experimental sciences. Section 3 presents pitfalls that ML scientists must avoid to keep their work repeatable. Section 4 describes the three chosen algorithms. Section 5 presents our experiments and the results we obtained. Section 6 discusses these results and Section 7 concludes the article.

2. Reproducibility and repeatability

In our article, we focus on both reproducibility and repeatability of results. To fully understand these issues, it is necessary to address the question of reproducibility in science.

Reproducibility is one of the pillars of science. Philosophers of science consider it a key criterion distinguishing science from pseudoscience. Here is the definition for reproducibility established by the ACM in 2020: "The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts."

The term "reproducibility crisis" emerged and gained popularity in the past decade. This crisis stems from the large number of articles whose experiments and results are not reproducible, and from the frequent inability of reviewers to replicate them. The sources of non-reproducibility are numerous and are not explored in our article. This crisis has led to rapid developments in the field of reproducibility, and several surveys report developments in this area [5, 6]. In the field of machine learning, reproducibility is still vague for researchers, and need clarification [7]. Many studies demonstrate this problem, particularly [8]. They made the hypothesis that AI work is not enough documented to be reproducible. To evaluate this, they set variables to describe how much documented a scientific work is. Then they surveyed 400 research papers published in top AI conferences and used their metrics. None of the papers were fully reproducible. From this, Semmelrock et al. [9] examined barriers and drivers to find solutions. Haibe-Kains et al. [10] provided a paper dealing with AI for Breast cancer screening. It denounces the aspects that must be clarified by article authors to be able to reproduce their work. This article is a good educational example for reproducibility. Other works noticed the confusion in the terminology used in reproducible research, and they tried to introduce frameworks to facilitate validation [11, 12]. The Journal of Artificial Intelligence Research (JAIR) also adopted a "reproducibility checklist" to enhance reproducibility [13]. Authors must now fill this reproducibility checklist and send it with their article.

Repeatability is different from reproducibility, but it is an important aspect, particularly when we deal with computer code. Here is the definition for repeatability proposed by the ACM: "The measurement can be obtained with stated precision by the same team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same location on multiple trials. For computational experiments, this means that researchers can reliably repeat their own computations." When setting up programs used for scientific research, we need to be able to debug them, and this implies that the stated precision is equal to zero. Indeed, we need bitwise identical results from run to run under the same conditions, otherwise how do you debug? The computers we use are deterministic machines, and after the early years of hardware and software development, it became straightforward to obtain consistent results every time with a deterministic stack of tools. This was taken for granted for many decades. However, since the development of sophisticated frameworks to hide complexity, hardware optimization in microprocessors, the massive use of hyperthreading and the trickiness of parallel random number generation, many scientists do not even notice that they produce different results from run to run. The software stack they use drives them into biased and non-repeatable experiments.

Reproducibility is less demanding since you just expect "similar" results leading to the same scientific conclusion, but it also relies on a sound repeatable program. Ideally, all experiments should be reproducible. However, many reproducibility issues in machine learning are largely due to repeatability issues, meaning that the code is developed on quicksand. Many recent studies have tried to come up with a framework to enhance reproducibility in machine learning [14-16]. Software best practices like code documentation or managing dependencies are also tackled [17]. Alahmari et al. [18] tried to obtain a repeatable work in deep learning with a Graphics Processing Unit (GPU). They used Pytorch and Tensorflow/Keras as ML frameworks. They show their inability to obtain repeatable results with Tensorflow 2.1/ Keras 2.3 in an image segmentation task, even when setting properly the seeds of the random number generators and disabling Compute Unified Device Architecture (CUDA) flags responsible for non-repeatable results. Nagarajan et al. [19] worked on repeatability in deep reinforcement learning. They first identified the sources of non-determinism that needed to be controlled to obtain repeatable results. Then, they studied the effect of these sources of non-determinism.

In our paper, we focus on repeatability for three main families of clustering algorithms. When conducting experiments to identify repeatability pitfalls in clustering methods, we encountered an interesting issue discussed on scikit-learn's GitHub repository (https://github.com/scikit-learn/scikit-learn/issues/27518). This issue describes inconsistent results when using parallelism with OpenMP for the *K*-Means algorithm. We found it still current in the 1.7.0 version of scikit-learn. OpenMP is an Application Programming Interface (API) for shared-memory multiprocessing programming. When correctly used, it is known to be deterministic. To our knowledge, *K*-Means is the only scikit-learn algorithm with a repeatability problem. We propose the hypothesis that the scikit-learn implementation of *K*-Means contains parallelism mismanagements with OpenMP.

3. Repeatability pitfalls

In this section, we explain the possible causes of hidden randomization in computation. A person willing to obtain bitwise repeatability must be aware of these issues.

3.1 *Random sources*

When working with stochastic experiments, using randomness in calculations, it is necessary to master random sources for repeatable experiments on computers. This is the precise purpose of pseudorandom number generators, whose behavior is designed to be deterministic models of randomness.

In this domain, we must be careful with the widely used legacy terminology of seeds and seeding. They are still in use in major APIs, but they can be really confusing for modern generators. Indeed, sound and modern generators cannot be reliably initialized by a "poor integer". The initial states (or statuses) of modern generators are much bigger and or structured. There is no bijection between integers and the set of initial states of modern generators [2]. In addition, the API and the state structure might change from one language/library to another, even when considering the same generator, and we may find different traces with the same generator and the same seed. To guarantee a good initialization we must use the complete state of the generator. Machine learning APIs often oversimplify the seeding. If we go back to

the generator implementations, they provide a way to get/set the precise state of a generator, the latter taking the shape of a more complex object. It can be, for instance, a list of integers for the Mersenne Twister generator [20], or a couple of key/counter for the Philox generator [21]. Even if something changes in the library implementation, or if we want to use another library, the state guarantees the same generator.

3.2 *Parallelism*

Modern computers have CPUs with multiple cores. This allows them to run different tasks in parallel to increase the speed of the program. To take advantage of this, Python often uses compiled files written in C or C++ with the OpenMP library. The code written with these two languages is compiled and therefore runs faster than pure Python code. The OpenMP library consists of a set of compiled directives influencing the program behavior. It provides for instances ways to optimize loops, using multiple threads. Programmers must implement these optimizations correctly to avoid thread synchronization problems and race conditions.

One common mistake to avoid for repeatable results is to keep the order of floating-point operations. They are non-associative and lead to inconsistent results, as illustrated by Figure 1.

```
>>> 0.1 + (0.2 − 0.1)
0.2
>>> (0.1 + 0.2) − 0.1
0.20000000000000004
```

Figure 1. Python example demonstrating non-associativity in floating-point operations

One easy solution when we notice inconsistent results with a compiled code is for instance to disable parallelism induced by OpenMP. We can set the maximum number of threads usable by OpenMP in system variables to 1 or use context managers in Python with the threadpoolctl library for example. However, it is possible to achieve repeatable multithreading, as demonstrated in [22] with OpenMP, by defining the behavior using flags during compilation.

4. Clustering methods

In this section, we give a formal definition of the studied algorithms. Then we describe them with simple step to follow and explain how repeatability pitfalls defined in Section 3 can intervene in the algorithm.

4.1 *Distance-based clustering: study of K-Means*

The K-Means algorithm has been proposed by several researchers as summarized by [23], but the name generally refers to the Lloyd algorithm [24] which is the most used algorithm. It has some flaws, like its linearity, creating spherical clusters due to the use of Euclidean metrics as a distance measure. This is a greedy algorithm, meaning that for each step of the algorithm, it will make the locally optimal choice. It then converges to a local minimum and needs multiple runs with different initializations to obtain a better result.

However, despite these flaws, it is widely used in data mining for its simplicity and low polynomial complexity. K-Means solves the partitioning problem by minimizing the sum of squared error of each cluster. Let $X = (x_i) \in \mathbb{R}^{d*K}$ be our n data points with d features, $C = (c_k) \in \mathbb{R}^{d*K}$ our K centers and p_{ik} a Boolean representing the membership of point x_i to cluster represented by the center c_k. The sum of squared error, also known as Minimum Sum-of-Squares Clustering (MSSC) [25] is:

$$\min_{p,c} \sum_{i=1}^{n} \sum_{k=1}^{K} p_{ik} \left\| x_i - c_k \right\|^2$$

$$\text{subject to} \begin{cases} 1. \sum_{k=1}^{K} p_{ik} = 1, \ \forall i \in [1 \ldots n] \\ 2. p_{ik} \in \{0, 1\} \\ 3. \sum_{i=1}^{n} p_{ik} \geq 1, \ \forall k \in [1 \ldots K] \end{cases}$$

The first double sum gives the objective function that we try to minimize, and the next mathematical expressions give the 3 main constraints. The first constraint states that each data point will be associated with a center, the second constraint states that p_{ik} are binary information (i.e., whether the i data point associated with the k center). The main idea of the third constraint is to ensure that each center is associated with at least one point. It ensures a solution with exactly K centers.

This algorithm can be separated into three steps:

1. **Initialization:** We initialize the centers c_k.

2. **Assignment:** For each point x_i, we calculate the Euclidean distances and assign it to the cluster represented by the closest center c_k.

3. **Update:** We replace each center c_k by the average of the points which compose the cluster.

Steps 2 and 3 are repeated until convergence or stopping of the algorithm. Note that K-Means is a heuristic to the MSSC objective function that omits the third constraint. This omission allows empty clusters, leading to degenerated results [26]. Let us see the repeatability aspect of each step.

Initialization: The algorithm is very sensitive to initialization. Finding the minimum of the objective function is NP-hard (Non-deterministic Polynomial-time). K-Means is a heuristic approach. This is why researchers generally execute the algorithm several times with different initializations to keep the most efficient execution. During this step, we initialize cluster centers randomly. This requires correct management of the random source. It must be set with a known initial state to be repeated.

Assignment: When calculating distances, it is important to define in advance the behavior of the algorithm in case of equidistance of a point with several centroids. We must ensure a deterministic selection (example: assign the point to the centroid that occupies the smallest index in the list of centroids).

Update: We need to compute the average of all the points belonging to the cluster. The centroid of each cluster will then update its position accordingly. During this step, there could be an empty cluster due to a centroid being too far away from the data. The position of the centroid of this cluster will not be able to be updated. The solution is then considered degenerated. To have the correct number of clusters, rules must be set for this specific case scenario.

4.2 *Density-based clustering: study of DBSCAN*

The DBSCAN algorithm proposed by [27] is an algorithm that uses density to group or separate data. This allows discovering clusters with various shapes, offering non-linear clustering possibilities, unlike K-Means, for example. There is no need to specify the number of clusters, and it has the capacity to handle outliers. However, the default algorithm struggles with data exhibiting highly variable neighborhood densities, and data with overlapping clusters.

Let $P \subset \text{powerset}(\chi)$ be the space of solutions such that $C = \{C_1, ..., C_l\}$ partitions the dataset. Let $d_{db}^{u}(p, q)$ be the DBSCAN-distance, which gives the smallest ε, such that two points are in the same DBSCAN cluster. Then we define the ε-density-based-clustering (εDBC) objective as [28]:

$$\min_{\substack{C \subset P \\ d_{db}^{\mu}(p,q) \leq \varepsilon \ \forall p, q \in C_i \ \forall C_i \in C}} |C|$$

• The algorithm requires two input parameters: the distance ε defining the neighborhood of a point, and a number

of points *MinPts* defining the number of points that must be part of this neighborhood to define the point as a core point. The algorithm classifies the data into three categories:

• Core points, whose neighborhood is dense (i.e., the number of points present within a radius ε is greater than or equal to *MinPts*).

• Border points, which belong to the neighborhood of a core point without being one itself (i.e., the number of points present within a radius ε is less than *MinPts*).

• Noise points, which are neither core nor border points.

The algorithm can be separated into 3 steps [29]:

1. **Identification:** For each point, calculate the neighborhood and identify the core points.
2. **Creation:** Among the core points, group those which are neighbors into clusters.
3. **Assignment:** Among the non-core points, if a point is close to a core point, it becomes a border point and is assigned to the cluster. Otherwise, it becomes a noise point.

Identification: We need to check for each point if it is a core point. To perform this, we compute how many neighbors each point has. A point has a neighbor if its distance from another point is less than ε. If a point has *MinPts* neighbors, we consider it as a core point. We should not have repeatability issues except if we have troubles with floating point operations (linked to dynamic execution inside microprocessors for optimization purposes). We can think of a rare case scenario where a point has a distance of exactly ε with another point. If we consider what we said in the parallelism section (section 3.2), we could have the result of the distance equal to $\varepsilon + error$. The point would sometimes be considered as a neighbor, and sometimes not.

Creation: If a core point belongs to the neighborhood of another core point, we put them in the same cluster. Same as before, there are no repeatability issues except if we have indeterminism during floating point operations.

Assignment: the neighbor exploration order is very important. If a border point p has two neighbors, a and b, which are core points belonging to different clusters, border point p can be added to the cluster of either point a or point b. This order must be set in advance to obtain the same result each time the algorithm is run.

4.3 *Agglomerative clustering: study of Ward*

Agglomerative clustering methods create nested clusters in a tree-like structure with a bottom-up approach. This allows clusters to be formed without specifying their number, by grouping the closest clusters together using a distance function. The Ward method [30] is a famous hierarchical clustering method. Just like K-Means, it seeks to minimize the variance when merging two clusters. Let q_i, $i = [1...K]$ be our cluster at step n-K, with q_i^* the center of the cluster. Let q_i, $i = [1...K]$, $j = [1...K]$, $i \neq j$ be the fusion of cluster i and j, with q_{ij}^* the center of this cluster. At each step, we try to minimize:

$$\min_{\substack{i,j \\ i \neq j}} \frac{1}{|q_{ij}|} \sum_{x \in q_{ij}} \left(x - q_{ij}^*\right)^2$$

The greedy nature of the algorithm makes the choice at each next step the optimal solution but may result in non-optimal clustering. Here are the main steps:

1. **Initialization:** Each point is its own cluster.
2. **Update:** Compute the predicted variance for each pair of clusters equal to the variance we would have if we merged the clusters.
3. **Merge:** Merge the two most similar clusters (i.e. clusters that give the minimum variance).

Steps 2 and 3 continue until only one cluster remains. The grouping can then be represented using a dendrogram.

Here, the objective function is the mean squared error. At each step, we choose the two clusters that give the smallest Mean Squared Error (MSE).

Initialization: This step is repeatable. Compared to the other algorithms, there is no choice to make here. Every point becomes a cluster at the initialization step.

Update: We compute the predicted variance for each pair of clusters. Only the non-associativity of floating-point operation can cause troubles.

Merge: During the merging step, it is necessary to define the behavior of the algorithm when several pairs of clusters have the same distance. For example, choose the clusters with the smallest indices.

4.4 *Summing up*

Table 1 is an array summarizing the important steps and the repeatable aspects that need attention.

We can split repeatable aspects into two categories:

• Random Number Generator (RNG) problems, which include the use of random numbers during the execution. This problem occurs when shuffling data or using a random initialization.

• Operation order problems designate floating-point operations problems that lead to the loss of bitwise repeatability.

Table 1. Recap of repeatable issues that may appear in each algorithm step

Algorithms	Aspects	
	RNG	Operation order
K-Means	Initialization	Assignment, update
DBSCAN		Identification, creation, assignment
WARD		Update, merge

Here, *K*-Means is the only algorithm using random numbers during the initialization process. Most of the problems come from the operation order because of the IEEE754 standard for floating-point operations. These problems exist because of the limitation of our machine. We must think about them when developing our programs.

5. Experiments

5.1 *Hardware and software*

For hardware, we use two CPU Xeon Platinum 8470 (104 workers, 208 logical threads).

For software, we use Python 3.11.2, scikit-learn 1.7.0, NumPy 2.3.1, OpenMP 4.5 and OpenBLAS 0.3.29. For more information about libraries version, head on to our Gitlab repository: https://gitlab.limos.fr/anbertrand1/repeatability_quest_clustering.

5.2 *Method*

5.2.1 *Energy measurement*

To measure the energy consumption, we designed a minimalist Python class to retrieve Running Average Power Limit (RAPL) values [31] during each replication loop. Python libraries like pyRAPL or pyjoules have a similar behavior. However, they do not consider the reset of RAPL counters. Because of this, we can obtain negative results with these libraries. That is what motivated us to develop a simple solution adapted to our configuration. Our method reads RAPL counters when we call the start and stop methods. It then computes energy consumption by subtracting the end value and the start value. If the result is negative, we add the maximum value a counter is supposed to handle. This method allows us to handle negative values when a reset happens during the experiment. However, it cannot handle multiple counter resets. According to Intel documentation, RAPL's package counter "has a wraparound time of around 60 secs when power consumption is high […]". From our tests, we know that on our platform, the counter reset happens approximately every 960 s (16 min) with intense workload. We ensure the mean duration of one experiment is less than the time it takes for a counter to reset. Energy counters are updated approximately every 1 ms. Therefore, for fast

algorithms, we measure 30 replications together to avoid recording zero energy consumption. For slow algorithms, we measure each replication independently. We consider an algorithm slow if one execution takes more than 10 seconds on our platform. We consider our method to be good enough, but we plan to improve it in the future.

5.2.2 *Datasets*

We chose seven datasets from the University of California Irvine (UCI) Machine Learning Repository (https://archive.ics.uci.edu). Clustering accuracy does not matter, our purpose being to show traces of non-repeatability. We think it is still a good idea to have labeled data people often use to work with. We completed the seven datasets with a generated one thanks to the scikit-learn library. This last dataset allows us to have custom data with a defined number of instances, features and clusters. Table 2 shows the characteristics of the chosen datasets.

Table 2. Sum-up of the chosen datasets, including number of instances, features and classes

Datasets	Dataset's real names	UCI Id	# Instances	# Features	# Classes
Iris	Iris	53	150	4	3
Toxicity	Toxicity	728	171	1,203	2
Wine	Wine	109	178	13	2
Breast cancer	Breast Cancer Wisconsin (Diagnostic)	17	569	30	2
Taiwan	Taiwanese Bankruptcy Prediction	572	6,819	95	2
Letter	Letter Recognition	59	20,000	16	26
Credit card	Default of Credit Card Clients	350	30,000	23	2
Generated	-	-	60,000	2	10

For the remainder of the article, we will refer to the names listed in the 'Datasets' column of Table 2. For the Generated dataset, we used the make_blobs function of scikit-learn with these parameters:

- n_sample = 60,000,
- n_features = 2,
- centers = 10,
- cluster_std = 0.7,
- random_state = 42.

For each dataset, we applied a Min-Max Scaler to scale features between 0 and 1.

5.2.3 *Algorithms*

We have tested *K*-Means, DBSCAN, and Ward implementation of scikit-learn. Because we noticed non-repeatable behaviors from the *K*-Means implementation of scikit-learn, we added SciPy, as well as a custom implementation of *K*-Means using OpenMP.

Our custom implementation has been largely generated by a Large Language Model (GPT 5.2) to fasten the development time. To avoid any issues related to thread concurrency, we choose to simply parallelize the code across the number of initializations. Each parallel part can run a *K*-Means clustering with a different initialization. We keep the best score. There is no parallelization during calculations inside a single *K*-Means, therefore no floating-point errors.

This implementation is just a minimal example to show that we can safely implement a repeatable parallel *K*-Means. Our parallelization choice is similar to the implementation retained when *K*-Means was implemented back in version 0.22 of scikit-learn, and this approach is still bitwise repeatable.

5.2.4 *Execution*

For each dataset, we run the three algorithms described in this article. We focus on multiple things:

• Repeatability of the results.

• Duration of the execution depending on the number of OpenMP threads used.

• Energy consumption of the execution depending on the number of OpenMP threads used.

To do so, we run 30 times each combination of dataset, algorithm and number of OpenMP threads. As the first execution of the algorithm is always slower compared to the following executions (due to memory allocation), we do not measure it and use it to determine if this is a fast or slow algorithm. Then, we measure the time and energy cost of the 30 executions according to our method described in 5.2.1, and because we have then enough experiments according to the Central Limit theorem if we need precise statistics.

Here is a sum-up of the execution.

• Algorithms: sklearn_dbscan, sklearn_kmeans, sklearn_ward, scipy_kmeans, custom_kmeans;

• Datasets: Iris, Toxicity, Wine, Breast cancer, Taiwan, Letter, Credit card, Generated;

• OpenMP threads: [1, 2, 3, 4, 16, 64, 128, 192];

• Replications: 30.

For Hyperparameters (HP) selection, we chose only parameters that work without trying to obtain better results. For *K*-Means, we opted for the '*k*-means++' initialization method, as it does not affect overall execution and because SciPy's *K*-Means encountered issues with its 'random' method on certain datasets. Here is a sum-up of HP selection:

• For *K*-Means

∘ n_clusters = n_classes

∘ rand_init = 42

∘ n_init = 5

∘ init = kmeans++

• For Ward

∘ n_clusters = n_classes

• For DBSCAN, HP selection is crucial to obtain a valid clustering. Bad HP selection can lead to a result where every point is considered as an outlier. To avoid this, we used a method described in [32]. We chose min_samples = 2 * n_features and found eps empirically by following [32]. Table 3 shows HP considered for each dataset.

Table 3. Hyperparameters for DBSCAN and for each dataset (e.g., eps, min_samples)

Name of the datasets	Epsilon	Min_samples
Iris	0.2	8
Toxicity	4.67	5
Wine	0.6	26
Breast cancer	0.5	60
Taiwan	0.915	190
Letter	0.22	32
Credit card	0.34	46
Generated	0.02	4

During the execution, we save every clustering result in JSON files for later analysis. We also save performance and energy results as complementary results in Comma-Separated Values (CSV) files.

We deactivated the use of Basic Linear Algebra Subprograms (BLAS) backend by setting the OPENBLAS_NUM_THREADS system variable to 1. This library is supposed to enhance scalar calculation like dot products. In our case, it raises errors when using large datasets due to incorrect configuration.

5.3 *Results*

5.3.1 *Repeatability results*

We ran each algorithm on all datasets 30 times and saved every result on a JSON file. For sklearn_dbscan, sklearn_ward, custom_kmeans, and scipy_kmeans, everything is bitwise repeatable. However, we noticed some non-repeatable results in sklearn_kmeans. You can see more details in Table 4, each non-repeatable result is marked with a cross. Tests are performed with up to 192 threads.

Table 4. *K*-Means results repeatability for each dataset depending on the number of OpenMP threads used (up to 192 threads) over 30 replications. C = final centers, L = final labels, I = inertia score, M = best iteration. A cross indicates that different results were found in at least two runs out of the 30 replications

	Bitwise-repeatability check for Scikit-Learn's *K*-Means																																
Datasets	nb threads	1				2				3				4				16				64				128				192			
	Results	C	L	I	M	C	L	I	M	C	L	I	M	C	L	I	M	C	L	I	M	C	L	I	M	C	L	I	M	C	L	I	M
Iris												×				×				×				×				×				×	
Toxicity																				×				×				×				×	
Wine																×				×				×				×				×	
Breast cancer										×		×		×		×		×		×		×		×		×		×		×		×	
Taiwan										×		×		×		×		×		×		×		×		×		×		×		×	
Letter										×		×		×		×		×		×		×		×		×		×		×		×	
Credit card																×		×		×		×		×		×		×		×		×	
Generated										×		×		×		×		×		×		×		×		×		×		×		×	

Table 5. Adjusted Rand Index (ARI) scores of the tested algorithms for each dataset. As label results are repeatable, we took the first execution of the algorithm applied to the dataset to compute the score

Datasets	Algorithms				
	scipy_kmeans	sklearn_kmeans	custom_kmeans	sklearn_dbscan	sklearn_ward
Iris	0.7163	0.7163	0.7163	0.5464	**0.7196**
Toxicity	0.0013	-0.0164	-0.0164	**0.0355**	-0.0164
Wine	0.8685	0.8537	0.8992	0.4259	**0.9310**
Breast cancer	**0.7422**	0.7302	0.7302	0.4877	0.5383
Taiwan	0.0103	0.0103	0.0103	**0.0916**	0.0015
Letter	0.1128	0.1261	0.1230	0.0317	**0.1472**
Credit card	0.0087	0.0087	0.0087	**0.0140**	0.0095
Generated	0.7767	**0.8828**	0.7813	0.7364	0.8475

When only one or two threads are used, every result is bitwise repeatable. From three threads, we start to notice some non-repeatable behaviors. The inertia score is almost never repeatable across our 30 replications. The inertia score is the sum of squared distances of samples to their closest cluster center. A more paradoxical fact is that sometimes, the final positions of cluster centers are repeatable (for example with Iris, Toxicity and Wine datasets) but the inertia score changes. Finally, we notice some combination of datasets and number of threads that are repeatable. For example, the resulting centers for the Credit card dataset with 4 threads are also repeatable.

If we look at the ARI score in Table 5, all *K*-Means implementations have close results, except for our Generated dataset where sklearn_kmeans stands out. We will give an explanation about this in section 6.

5.3.2 *Performance results*

Performance and energy consumption results are given in Appendix for each algorithm over 30 replications.

For the duration, we notice an increase in sklearn_dbscan when we increase the number of threads for Iris, Toxicity, Wine, and Breast cancer datasets. In the worst case, we have for the Breast cancer dataset an increase in time ranging from less than 0.1 s with one thread to 1 s with more than 128 threads. For Taiwan, Letter, Credit card, and Generated datasets, the duration decreases when we increase the number of threads up to 16, then it starts to increase. The Letter dataset presents a good performance gain, from 16 s with one thread, to less than 2 s with 16 threads. The kmeans_scipy algorithm does not seem to have a special behavior with parallelization. Duration results are constant. Same remark for sklearn_ward, results are constant except for Iris dataset with 192 threads, and Wine dataset with 128 threads. For sklearn_kmeans, increasing the number of threads generally degrades performance. The only exception is Taiwan dataset for 1 and 4 threads, which gives a ×2 speed-up. The performance degradation is quite high, with for example a ×13 "speed-down" for Letter dataset from 1 to 128 threads meaning too much overhead to handle many threads for this dataset.

For the energy consumption, the overall tendency follows the duration tendency.

6. Discussion

We checked the source code of sklearn_kmeans and found an internal check for parallelization. If the number of data points is not high enough, the number of threads used is downgraded compared to what the user asked for. For example, the Iris, Toxicity, and Wine dataset does not contain enough data points to be parallelized. That explains the repeatable results of the final positions of the centers for these three datasets. However, it seems that the number of threads given by the user is used correctly to compute the inertia, leading to non-repeatable results even for small datasets. Differences in results are at the order of 10^{-14}. As stated at the beginning of this paper, we think this small difference comes from uncontrolled operation order, leading to rounding errors during reduction operations. It can also come from a conversion between C data type and Python data type, as it happened to us during the development of our custom *K*-Means. We do not have an explanation about repeatable combinations we found. We tried the combination "Credit card with 3 threads" a thousand times and obtained repeatable results every time. It is unlikely due to pure luck but may be due to some properties related to this specific dataset.

When we look at *K*-Means performance, scipy_kmeans is always faster than sklearn_kmeans, even if the first does not use parallelization. However, this method seems embarrassingly parallel with the use of different initializations. Even worse, using a larger number of threads for sklearn_kmeans degrades performances in our configuration, with the exception of Taiwan dataset. It should definitely take advantage of parallelization, as demonstrated by our custom implementation, which was not meant to be fast, only repeatable. Works on parallelization with OpenMP exist for *K*-Means [33, 34]. They mainly focus on accelerating the assignment or update step by using multiple threads. Datasets used in our experiments had no gain from parallelism with sklearn_kmeans. They may not be suitable to the type of parallelism used by scikit-learn. Regarding the ARI scores, tested implementations give similar scores (Table 5) except for the Generated dataset. You can see in Figure 2 the original data, and clustering results, alongside the final position of the centers. As *K*-Means is very sensitive to initialization, we can see that scipy_kmeans and custom_kmeans have fallen into a local minimum compared to sklearn_kmeans, which has found a result close to the best solution. Five initializations for SciPy and our custom implementation were not enough to achieve the same accuracy as scikit-learn.

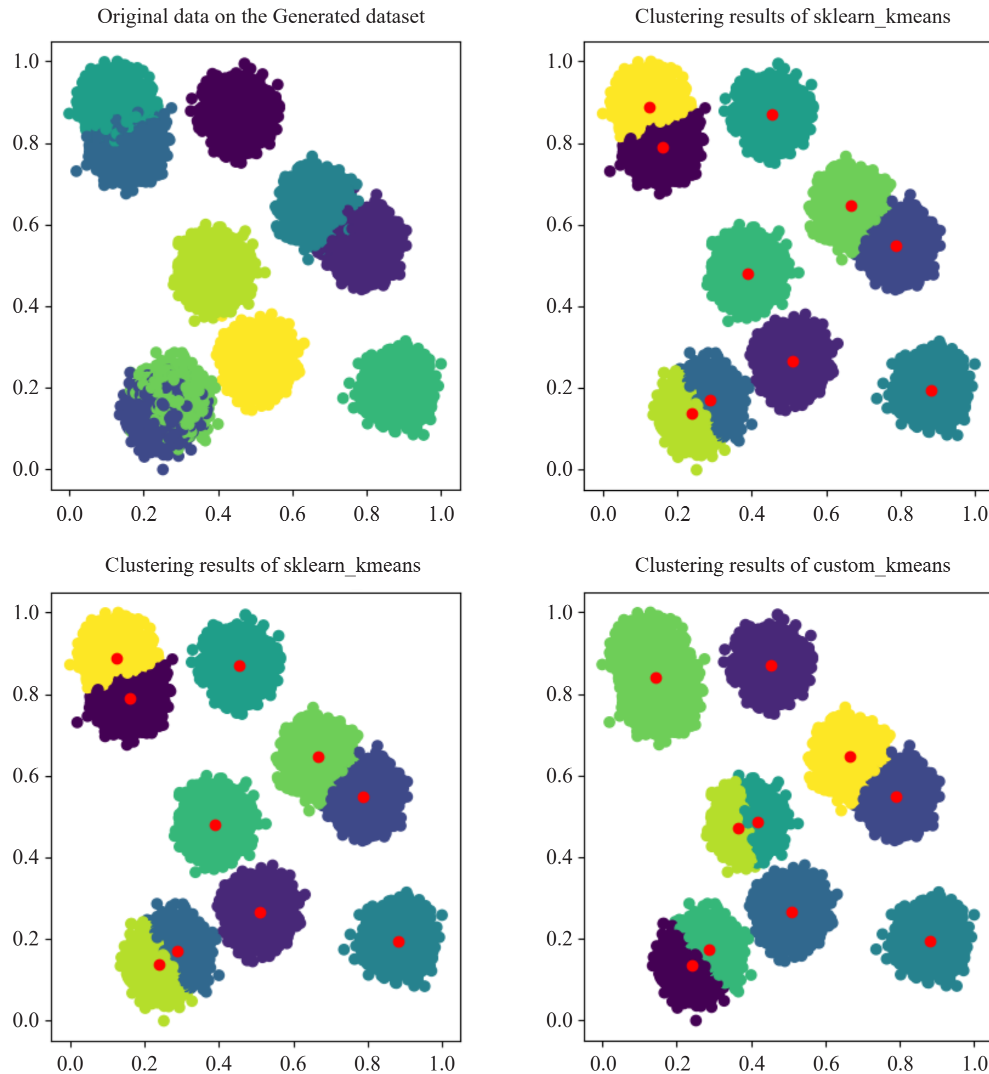

Figure 2. Clustering results of scipy_kmeans, sklearn_kmeans, and custom_kmeans on the original data. Centers are represented with red dots

7. Conclusion

In this paper, we have explored repeatability issues observed in popular clustering methods such as *K*-means, Ward, and DBSCAN. We identified two repeatability pitfalls that developers should address to ensure repeatable behavior at each step of the three aforementioned algorithms. Bitwise repeatability is required primarily for debugging purposes. We found that the scikit-learn implementation of *K*-means produces non-repeatable results when parallelized using OpenMP and most of all, we do not observe any performance gain when parallelization is activated. Dealing with the different numerical results observed in the same conditions, problems are most likely due to the mismanagement of the OpenMP library, which leads to a random ordering of floating-point operations during computation (and these are non-associative). Our advice is to use a maximum of two threads for OpenMP with scikit-learn when bitwise-repeatable results are needed. We then compared the scikit-learn implementation with SciPy's *K*-means and our own implementation of *K*-Means in terms of accuracy, execution time, and energy consumption. Both SciPy's and our *K*-Means are fully repeatable. SciPy outperforms scikit-learn in terms of execution time on all tested datasets, while maintaining a comparable ARI score, except for our dataset named "Generated", where it converges to a local minimum compared to scikit-learn using the same hyperparameters. While there is a constant pursuit of higher performance in computing, compromising repeatability and reproducibility moves us beyond the scientific method; furthermore, the

loss of repeatability fundamentally undermines our ability to debug complex systems.

We believe that many reproducibility problems encountered when using clustering methods are, to a large extent, caused by repeatability issues. These issues could result from a lack of in-depth understanding of the tools, which are often used as black boxes by AI scientists. Additional attention is needed in ML when applied to critical fields like medicine, where interpretability is also needed [35]. While practitioners generally have a strong theoretical background in knowledge representation and modeling, and often sufficient programming skills to implement solutions, they do not necessarily possess detailed knowledge of how the underlying machine implementation works, nor of how to write code in a repeatable manner. Such a lack of repeatability directly impacts the ability to properly debug the application and as said previously, it can ultimately diminish scientific contributions. The study presented in [8] reported reproducibility problems mainly due to insufficient documentation and the absence of source code availability (only 6% of the studies provided access to their source code). For studies that did provide source code, it remains unclear what proportion suffers from repeatability issues, as opposed to genuine reproducibility problems such as portability. We aim to increase the visibility of these issues. As future work, we plan to provide a Docker-based version of this experiment to enhance reproducibility. We also intend to improve our measurement tool to address the limitations discussed in this paper.

Funding support

This work is financed by the CPER IDEAL and Clermont Auvergne Métropole (CAM).

Conflict of interest

The authors declare no competing financial interest.

References

[1] Baker M. 1,500 scientists lift the lid on reproducibility. *Nature*. 2016; 533(7604): 452-454. Available from: https://doi.org/10.1038/533452a.

[2] Hill DRC, Antunes BA, Bertrand A, Mephu Nguifo E, Yon L, Nautré-Domanski J, et al. Machine learning and reproducibility impact of random numbers. In: *38th European Simulation and Modelling Conference (ESM)*. San Sebastian, Spain: The European Multidisciplinary Society for Modelling and Simulation Technology; 2024. p.65-70.

[3] Saxena A, Prasad M, Gupta A, Bharill N, Patel OP, Tiwari A, et al. A review of clustering techniques and developments. *Neurocomputing*. 2017; 267: 664-681. Available from: https://doi.org/10.1016/j.neucom.2017.06.053.

[4] Kramer O. Scikit-learn. In: *Machine Learning for Evolution Strategies*. Switzerland: Springer; 2016. p.45-53.

[5] Antunes B, Hill DRC. Reproducibility, replicability and repeatability: A survey of reproducible research with a focus on high performance computing. *Computer Science Review*. 2024; 53: 100655. Available from: https://doi.org/10.1016/j.cosrev.2024.100655.

[6] Hernández JA, Colom M. Repeatability, reproducibility, replicability, reusability (4R) in journals' policies and software/data management in scientific publications: A survey, discussion, and perspectives. *arXiv:2312.11028*. 2023. Available from: https://doi.org/10.48550/arXiv.2312.11028.

[7] Raff E, Benaroch M, Samtani S, Farris AL. What do machine learning researchers mean by "reproducible"? *Proceedings of the AAAI Conference on Artificial Intelligence*. 2025; 39(27): 28671-28683. Available from: https://doi.org/10.1609/aaai.v39i27.35093.

[8] Gundersen OE, Kjensmo S. State of the art: Reproducibility in artificial intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2018; 32(1): 1644-1651. Available from: https://doi.org/10.1609/aaai.v32i1.11503.

[9] Semmelrock H, Ross-Hellauer T, Kopeinik S, Theiler D, Haberl A, Thalmann S, et al. Reproducibility in machine-learning-based research: Overview, barriers, and drivers. *AI Magazine*. 2025; 46(2): e70002. Available from: https://doi.org/10.1002/aaai.70002.

[10] Haibe-Kains B, Adam GA, Hosny A, Khodakarami F, Waldron L, Wang B, et al. Transparency and reproducibility

in artificial intelligence. *Nature*. 2020; 586: E14-E16. Available from: https://doi.org/10.1038/s41586-020-2766-y.
[11] Desai A, Abdelhamid M, Padalkar NR. What is reproducibility in artificial intelligence and machine learning research? *AI Magazine*. 2025; 46(2): e70004. Available from: https://doi.org/10.1002/aaai.70004.
[12] Gundersen OE. The fundamental principles of reproducibility. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*. 2021; 379(2197): 20200210. Available from: https://doi.org/10.1098/rsta.2020.0210.
[13] Gundersen OE, Helmert M, Hoos H. Improving reproducibility in AI research: Four mechanisms adopted by JAIR. *Journal of Artificial Intelligence Research*. 2024; 81: 1019-1041. Available from: https://doi.org/10.1613/jair.1.16905.
[14] Koenigstorfer F, Haberl A, Kowald D, Ross-Hellauer T, Thalmann S. Black box or open science? Assessing reproducibility-related documentation in AI research. In: *Hawaii International Conference on System Sciences 2024 (HICSS-57)*. Hawaii: AIS eLibrary; 2024. p.682-691.
[15] White M, Haddad I, Osborne C, Liu XYY, Abdelmonsef A, Varghese S, et al. The model openness framework: Promoting completeness and openness for reproducibility, transparency, and usability in artificial intelligence. *arXiv:2403.13784*. 2024. Available from: https://doi.org/10.48550/arXiv.2403.13784.
[16] Chen B, Wen M, Shi Y, Lin D, Rajbahadur GK, Jiang ZM. Towards training reproducible deep learning models. In: *Proceedings of the 44th International Conference on Software Engineering*. Pennsylvania, USA: Association for Computing Machinery; 2022. p.2202-2214.
[17] Wolter M, Veeramacheneni L, Hoyt CT. More rigorous software engineering would improve reproducibility in machine learning research. *arXiv:2502.00902*. 2025. Available from: https://doi.org/10.48550/arXiv.2502.00902.
[18] Alahmari SS, Goldgof DB, Mouton PR, Hall LO. Challenges for the repeatability of deep learning models. *IEEE Access*. 2020; 8: 211860-211868. Available from: https://doi.org/10.1109/ACCESS.2020.3039833.
[19] Nagarajan P, Warnell G, Stone P. The impact of nondeterminism on reproducibility in deep reinforcement learning. In: *Machine Learning Workshop at ICML 2018*. Stockholm, Sweden: International Machine Learning Society; 2018. p.1-10.
[20] Matsumoto M, Nishimura T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*. 1998; 8(1): 3-30. Available from: https://doi.org/10.1145/272991.272995.
[21] Salmon JK, Moraes MA, Dror RO, Shaw DE. Parallel random numbers: As easy as 1, 2, 3. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, Washington, USA: Association for Computing Machinery; 2011. p.1-12. Available from: https://doi.org/10.1145/2063384.2063405.
[22] Antunes B, Claude M, Hill DRC. Performance and reproducibility assessment of quantum dissipative dynamics framework: A comparative study of fortran compilers, MKL, and FFTW. *Cloud Computing and Data Science*. 2025; 6(2): 244-262. Available from: https://doi.org/10.37256/ccds.6220256280.
[23] Ikotun AM, Ezugwu AE, Abualigah L, Abuhaija B, Heming J. *K*-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*. 2023; 622: 178-210. Available from: https://doi.org/10.1016/j.ins.2022.11.139.
[24] Lloyd S. Least squares quantization in PCM. *IEEE Transactions on Information Theory*. 1982; 28(2): 129-137. Available from: https://doi.org/10.1109/TIT.1982.1056489.
[25] Aloise D, Hansen P, Liberti L. An improved column generation algorithm for minimum sum-of-squares clustering. *Mathematical Programming*. 2012; 131(1): 195-220. Available from: https://doi.org/10.1007/s10107-010-0349-7.
[26] Alguwaizani A. Degeneracy on *K*-means clustering. *Electronic Notes in Discrete Mathematics*. 2012; 39: 13-20. Available from: https://doi.org/10.1016/j.endm.2012.10.003.
[27] Ester M, Kriegel HP, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. Portland, Oregon, USA: AAAI Press; 1996. p.226-231.
[28] Beer A, Draganov A, Hohma E, Jahn P, Frey CMM, Assent I. Connecting the dots—density-connectivity distance unifies DBSCAN, k-center and spectral clustering. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Long Beach, CA, USA: Association for Computing Machinery; 2023. p.80-92.
[29] Schubert E, Sander J, Ester M, Kriegel HP, Xu X. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems*. 2017; 42(3): 19. Available from: https://doi.org/10.1145/3068335.
[30] Ward Jr JH. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical*

Association. 1963; 58(301): 236-244. Available from: https://doi.org/10.1080/01621459.1963.10500845.

[31] Desrochers S, Paradis C, Weaver VM. A validation of DRAM RAPL power measurements. In: *Proceedings of the Second International Symposium on Memory Systems*. Alexandria, VA, USA: Association for Computing Machinery; 2016. p.455-470.

[32] Sander J, Ester M, Kriegel HP, Xu X. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery*. 1998; 2(2): 169-194. Available from: https://doi.org/10.1023/A:1009745219419.

[33] Abdullah A, Mateenuddin QH, Ansari Z. An OpenMP based approach for parallelization and performance evaluation of *k*-means algorithm. *Turkish Journal of Computer and Mathematics Education*. 2021; 12(10): 1524-1537.

[34] Naik DSB, Kumar SD, Ramakrishna SV. Parallel processing of enhanced *K*-means using OpenMP. In: *2013 IEEE International Conference on Computational Intelligence and Computing Research*. Madurai, India: IEEE; 2013. p.1-4.

[35] Ciobanu-Caraus O, Aicher A, Kernbach JM, Regli L, Serra C, Staartjes VE. A critical moment in machine learning in medicine: on reproducible and interpretable learning. *Acta Neurochirurgica*. 2024; 166(1): 14. Available from: https://doi.org/10.1007/s00701-024-05892-8.

Appendix A. Duration of algorithms

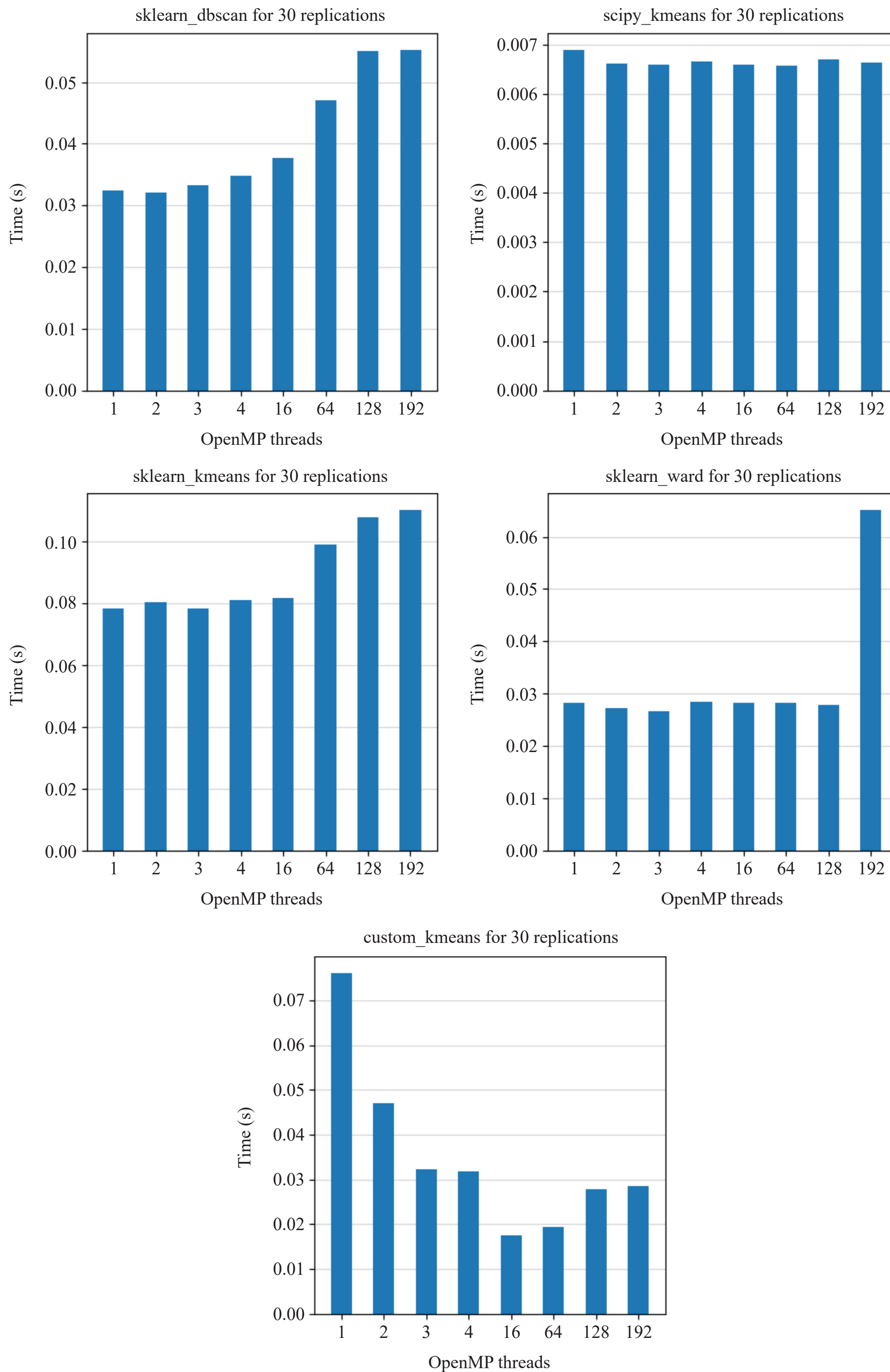

Figure A1. Duration of each algorithm on Iris data depending on the number of OpenMP threads used

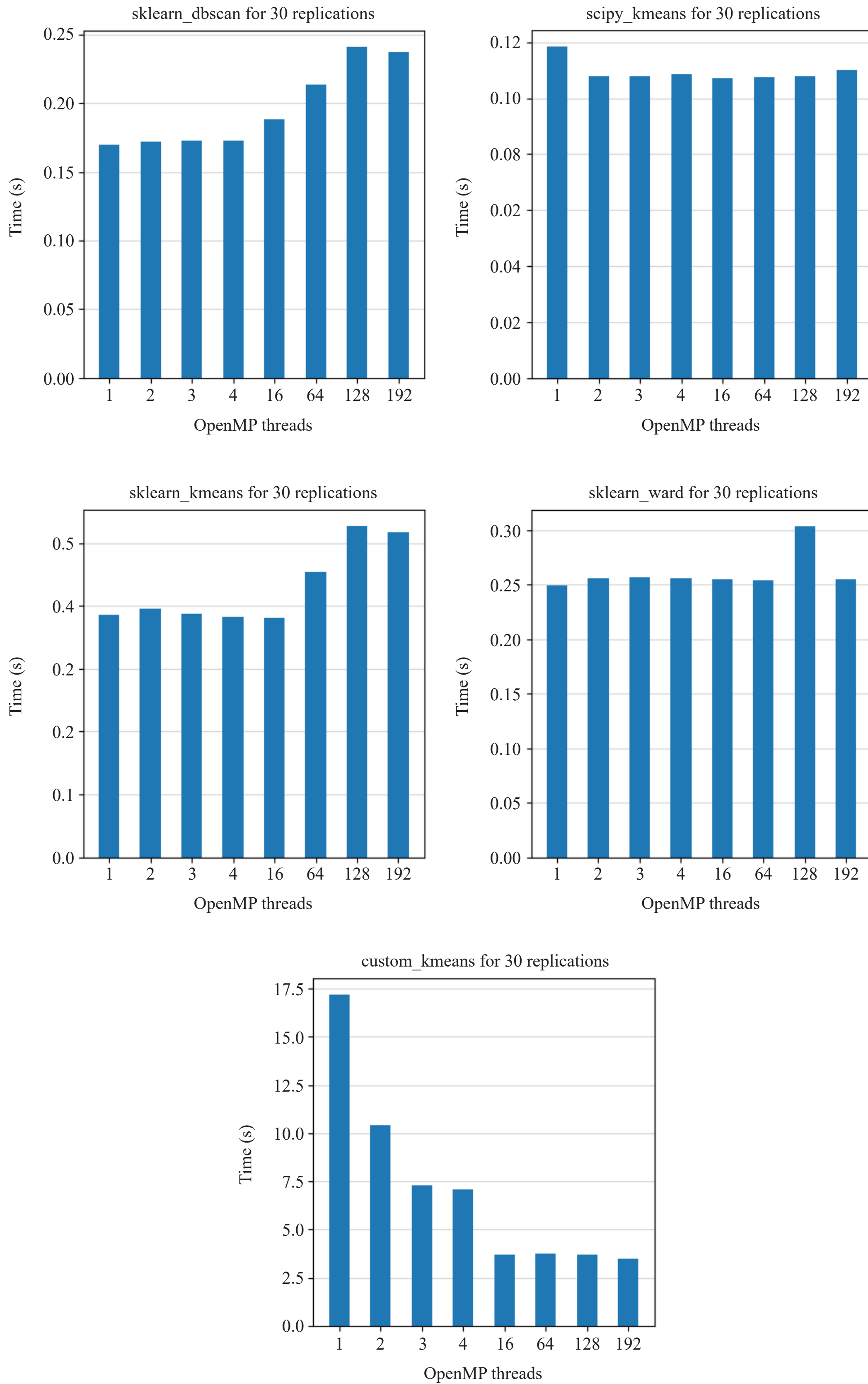

Figure A2. Duration of each algorithm on Toxicity data depending on the number of OpenMP threads used

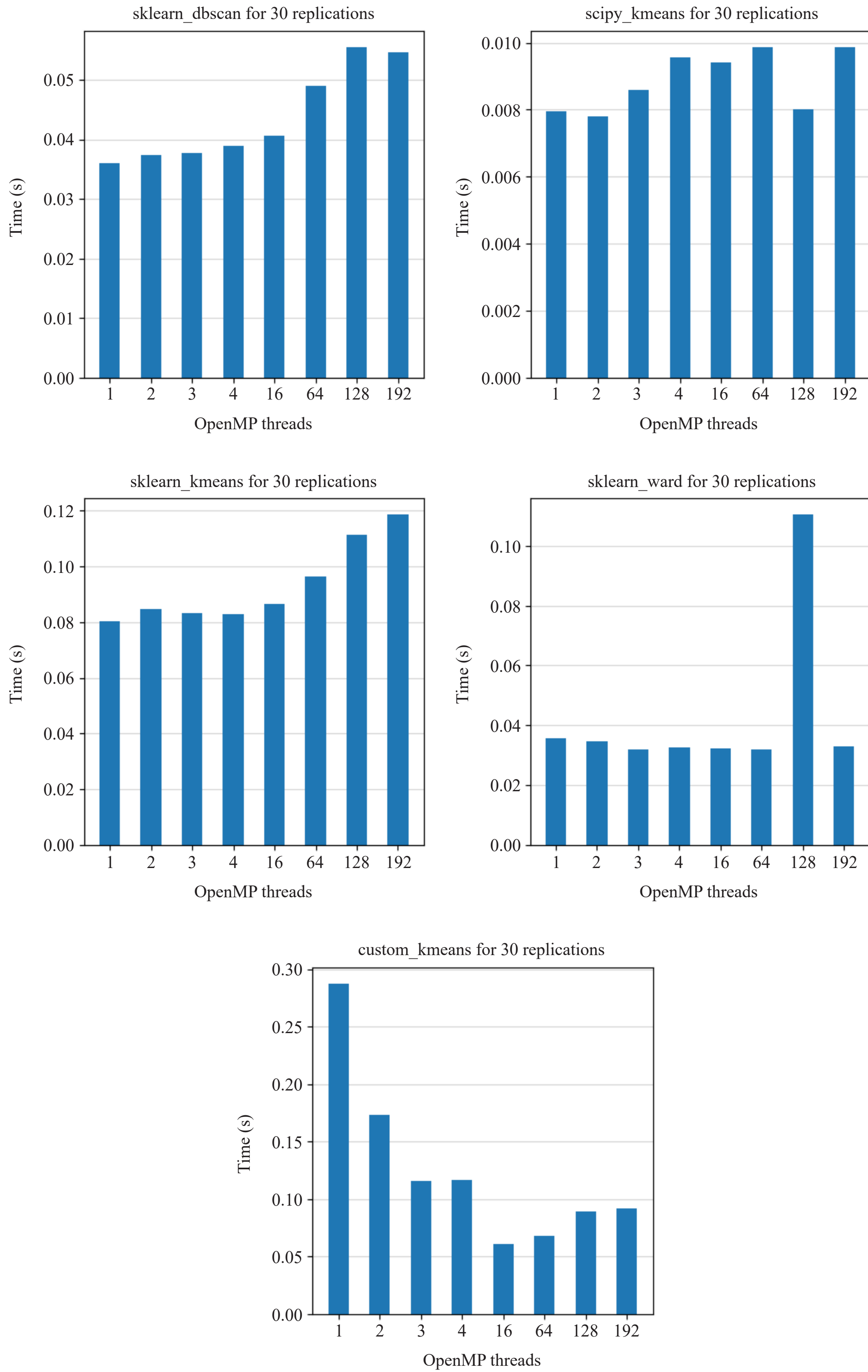

Figure A3. Duration of each algorithm on Wine data depending on the number of OpenMP threads used

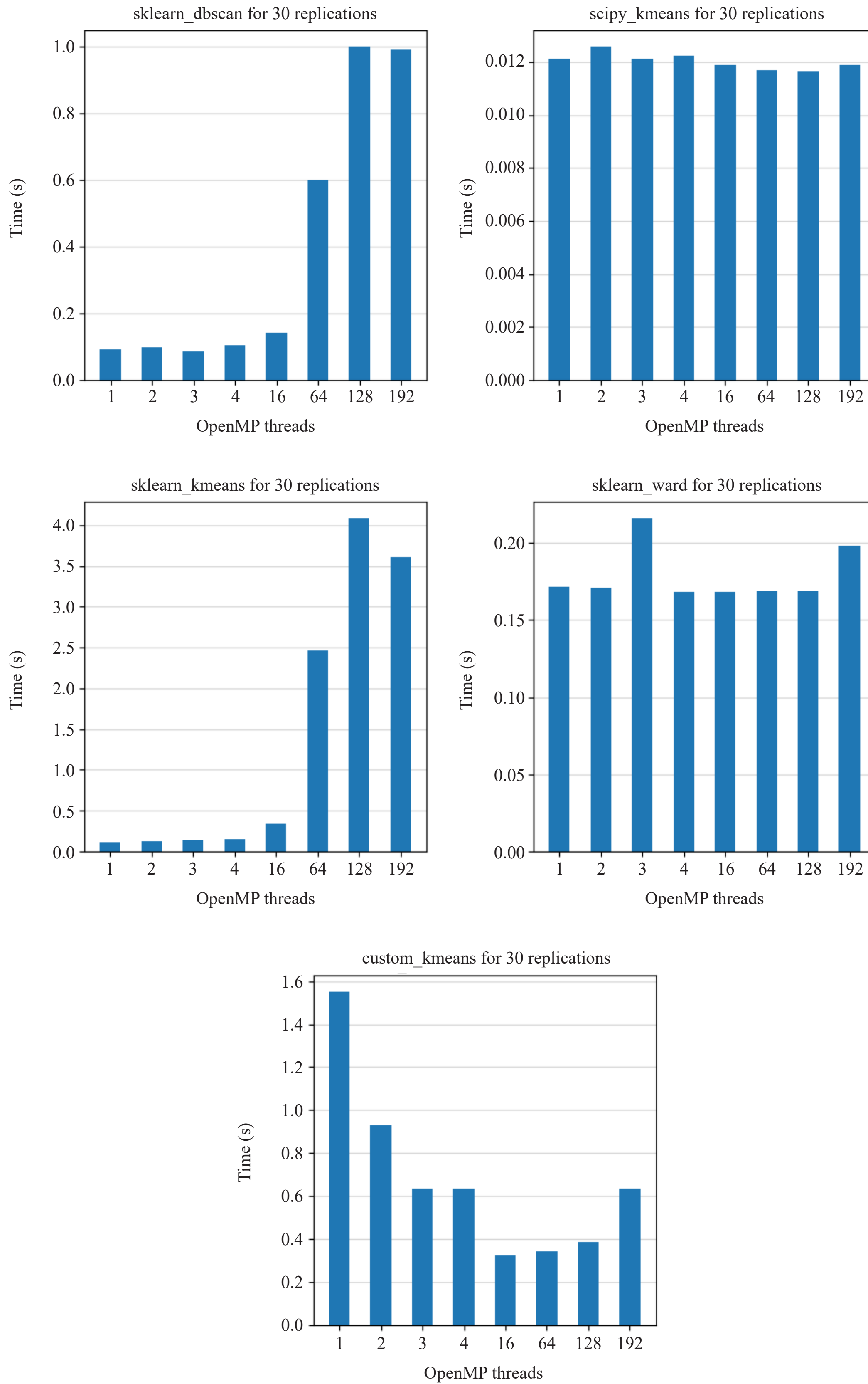

Figure A4. Duration of each algorithm on Breast cancer data depending on the number of OpenMP threads used

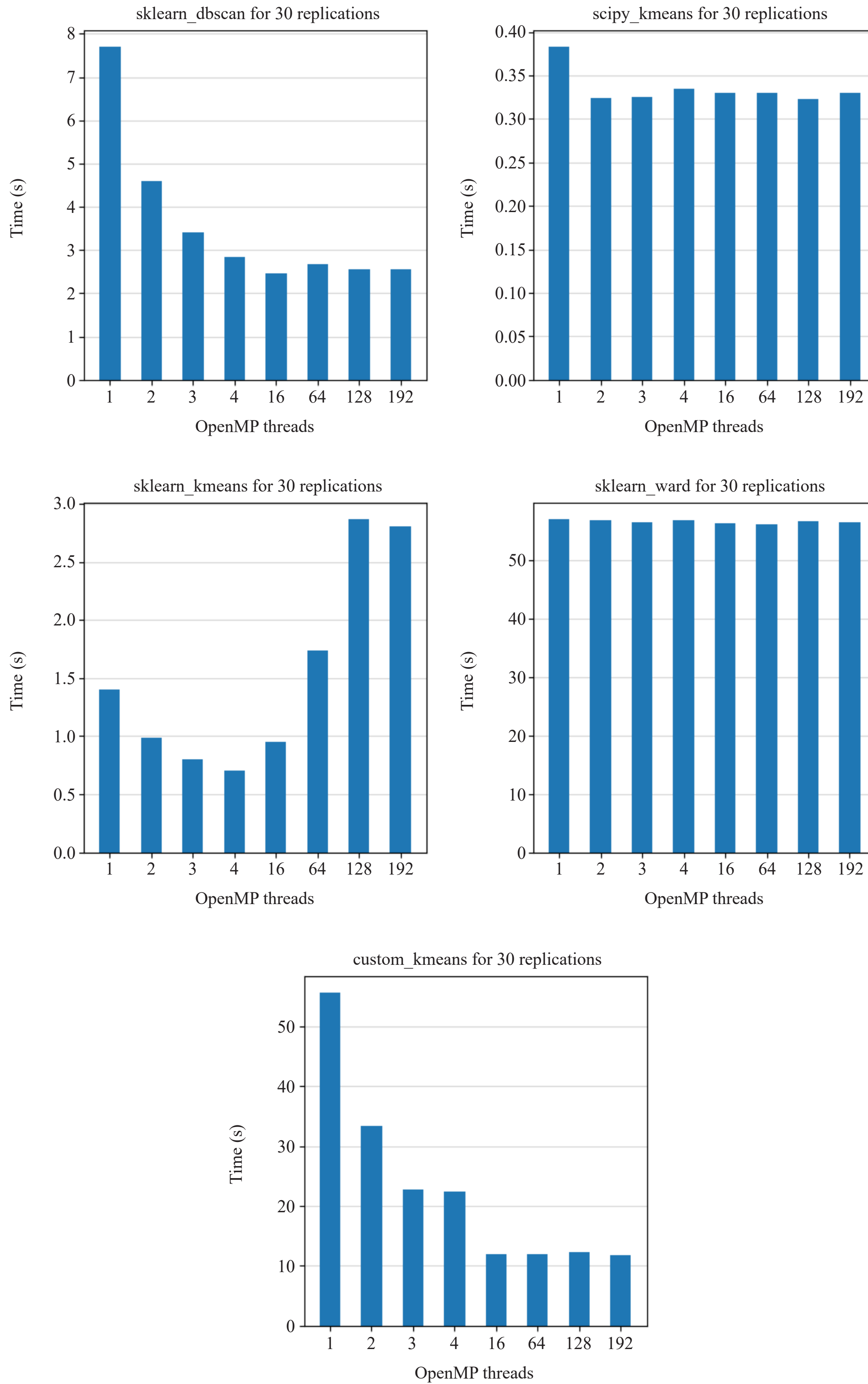

Figure A5. Duration of each algorithm on Taiwan data depending on the number of OpenMP threads used

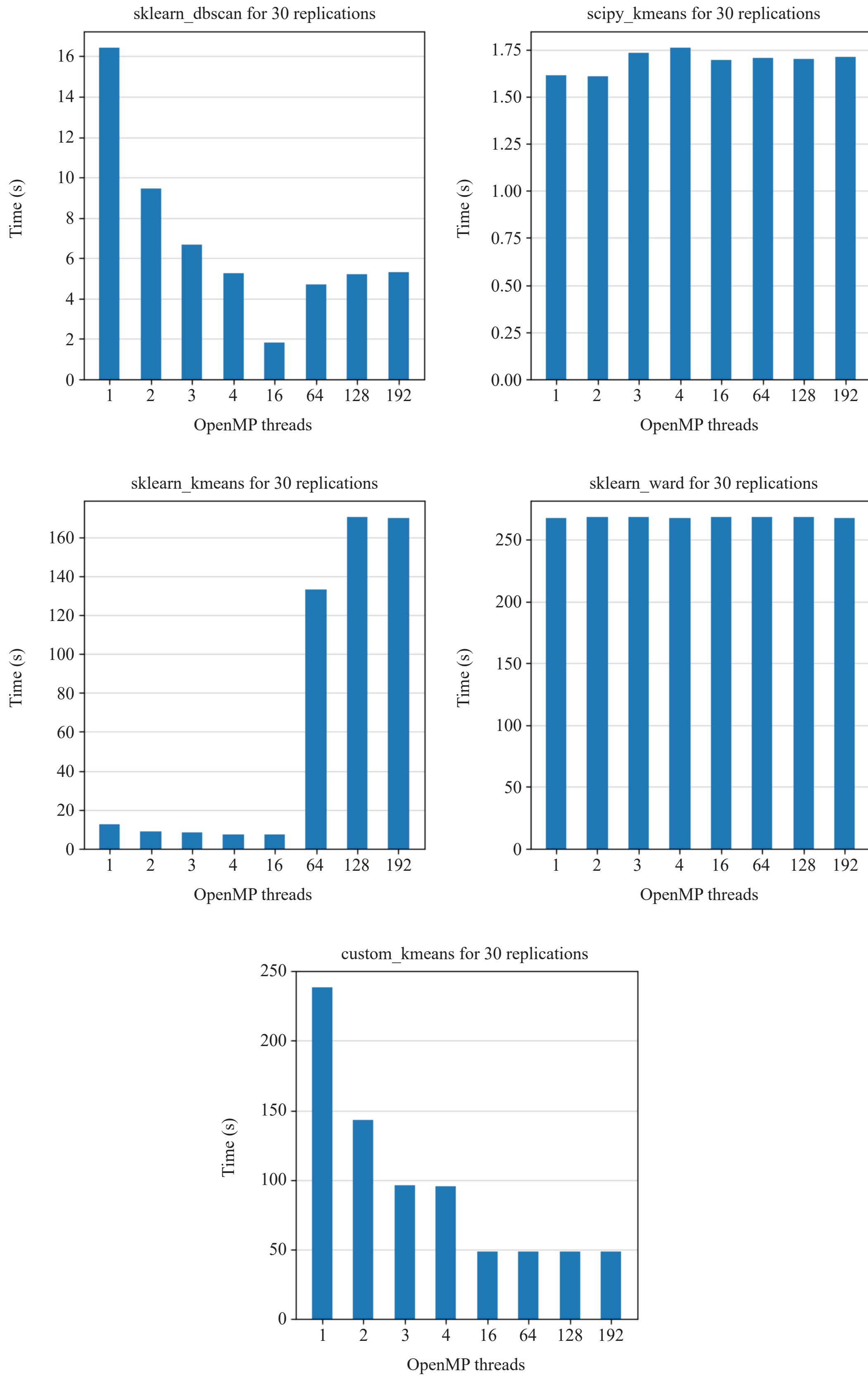

Figure A6. Duration of each algorithm on Letter data depending on the number of OpenMP threads used

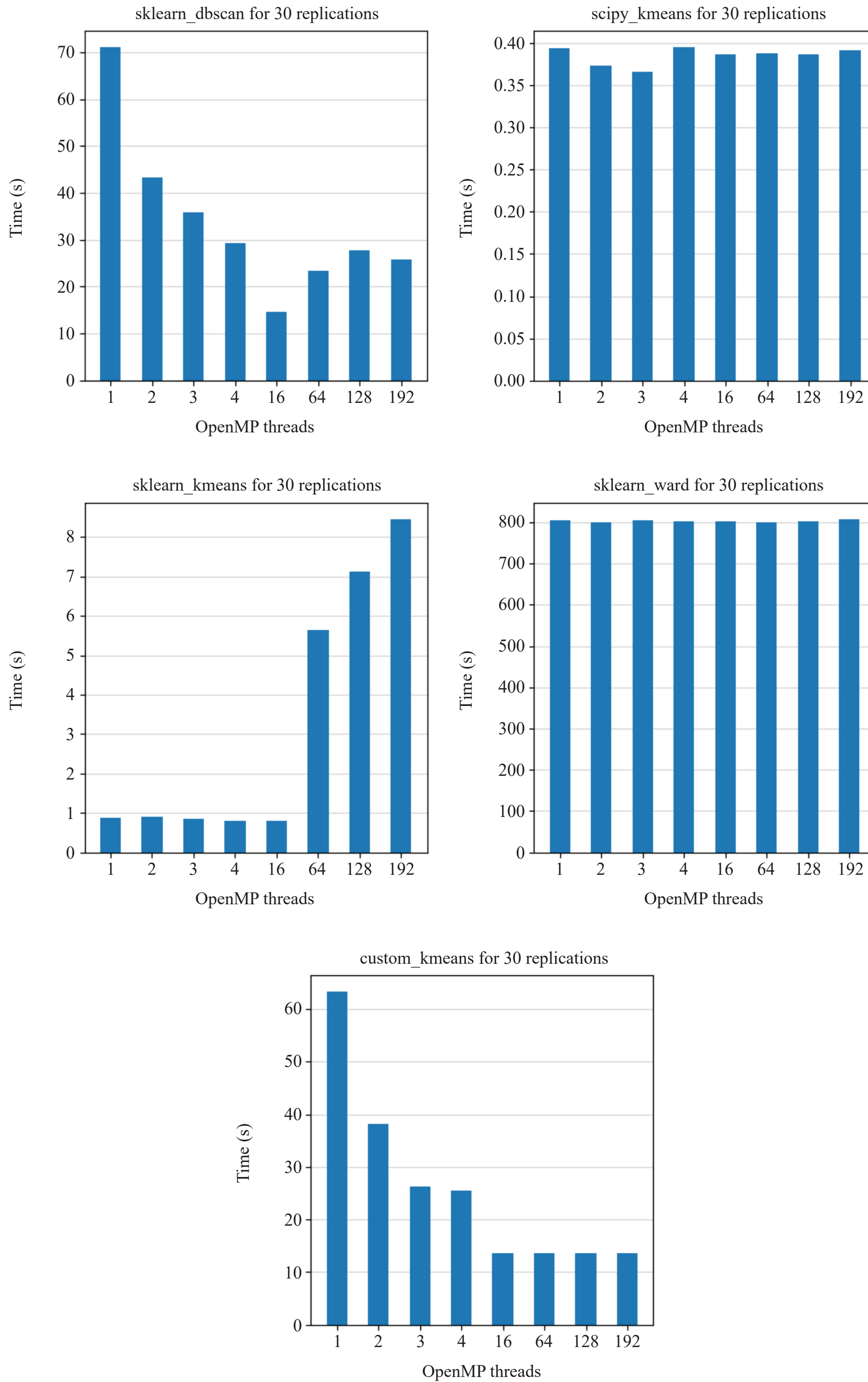

Figure A7. Duration of each algorithm on Credit card data depending on the number of OpenMP threads used

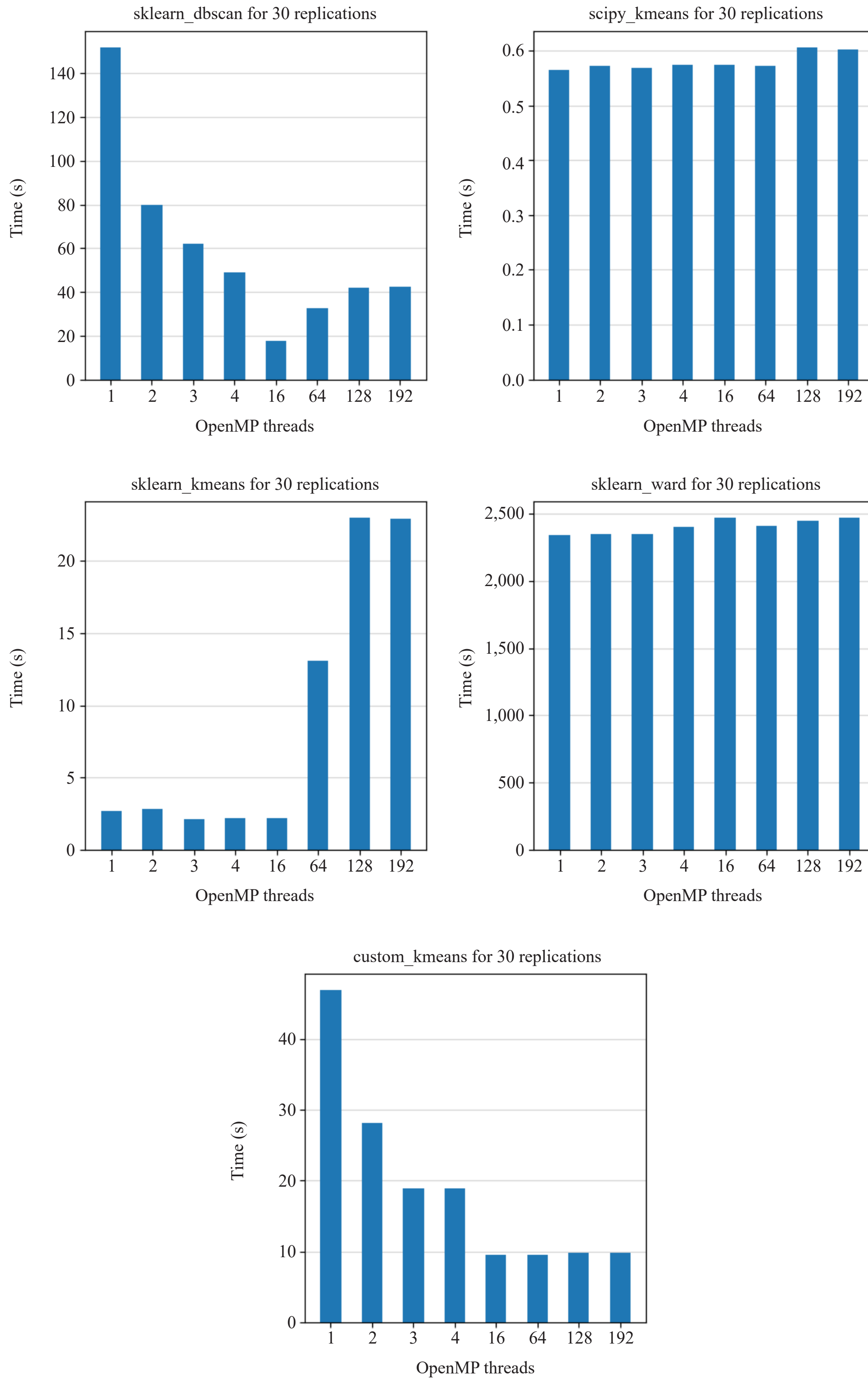

Figure A8. Duration of each algorithm on Generated data depending on the number of OpenMP threads used

Appendix B. Energy consumption of algorithms

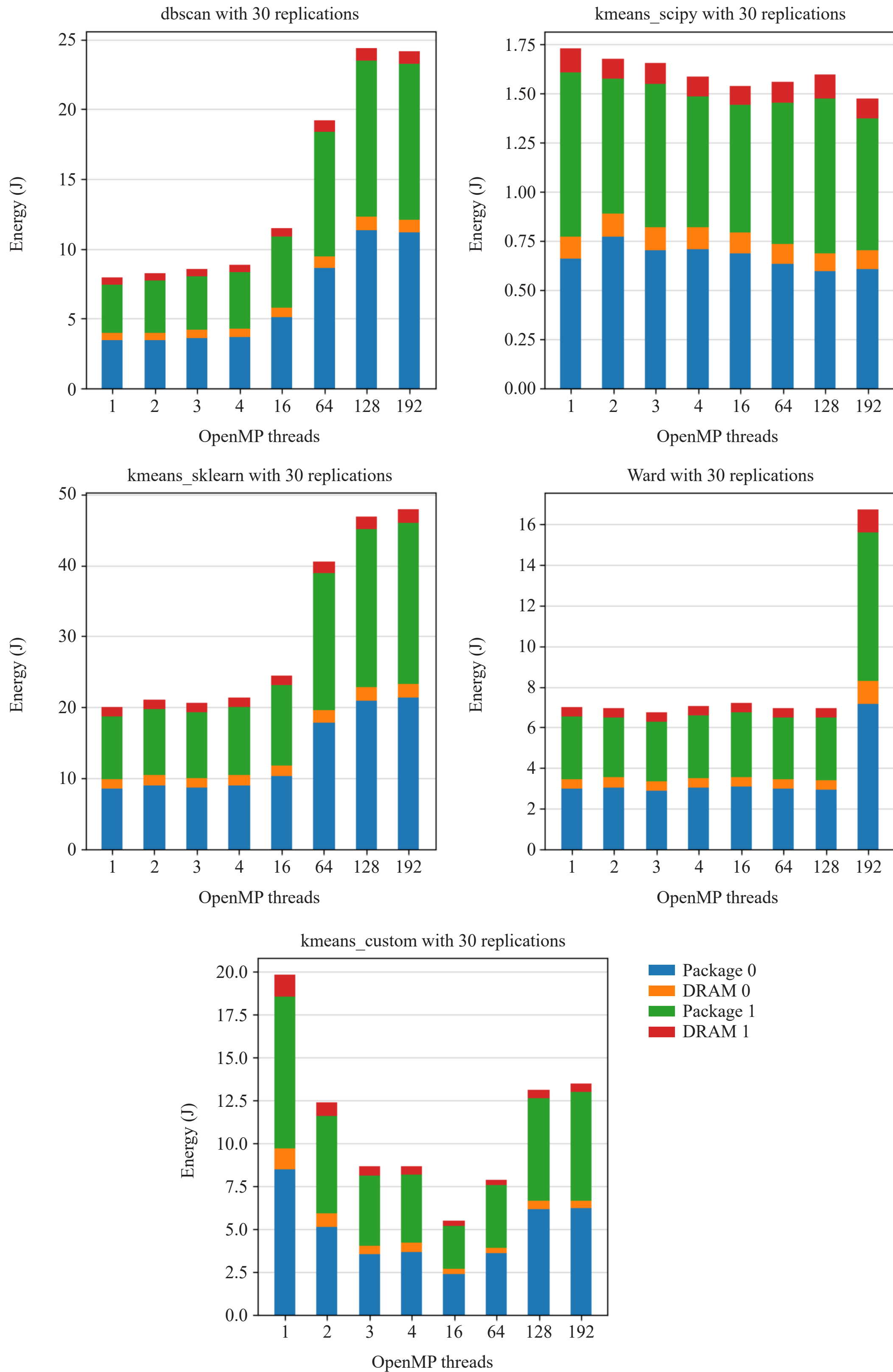

Figure B1. Energy consumption of each algorithm on Iris data depending on the number of OpenMP threads used

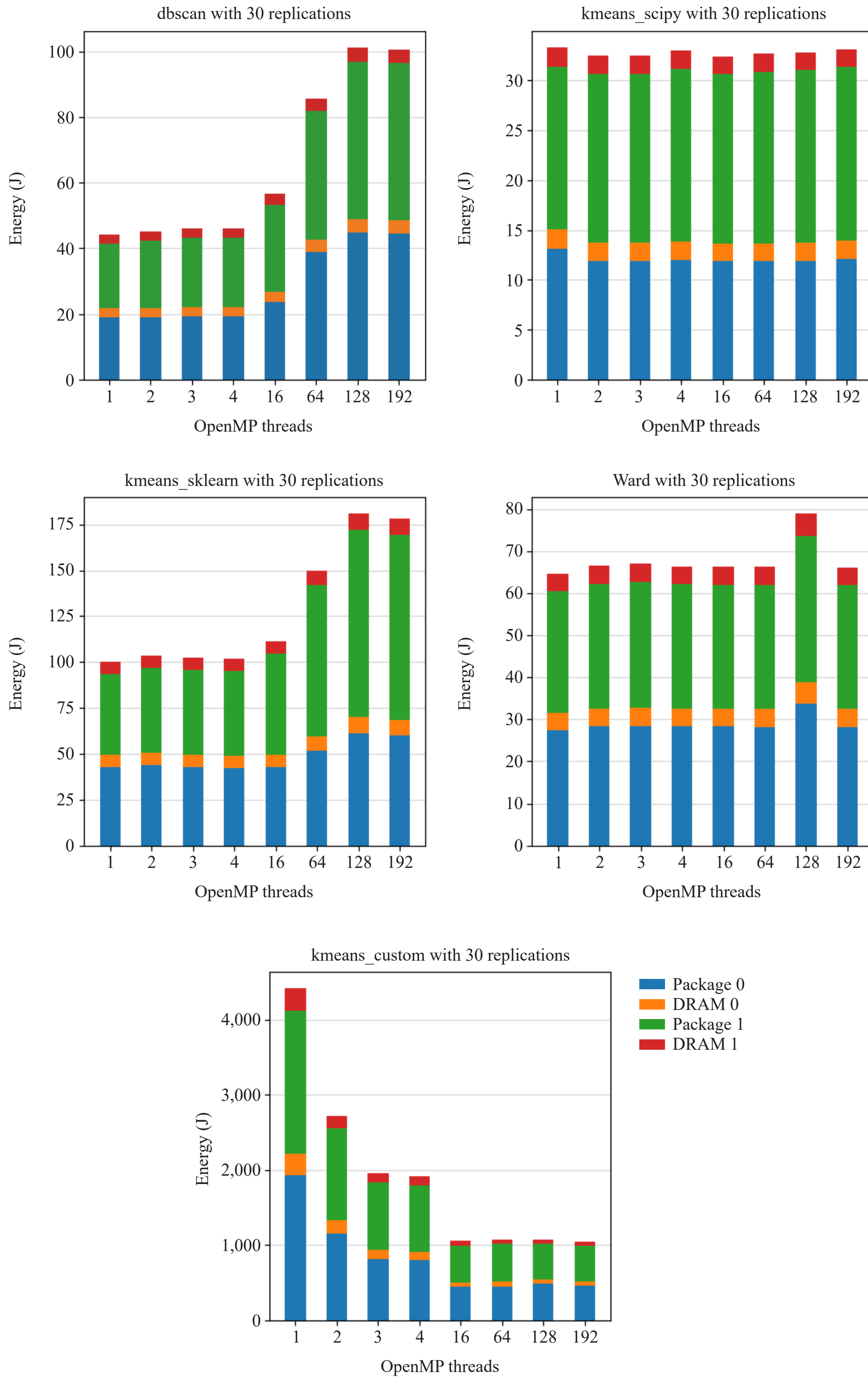

Figure B2. Energy consumption of each algorithm on Toxicity data depending on the number of OpenMP threads used

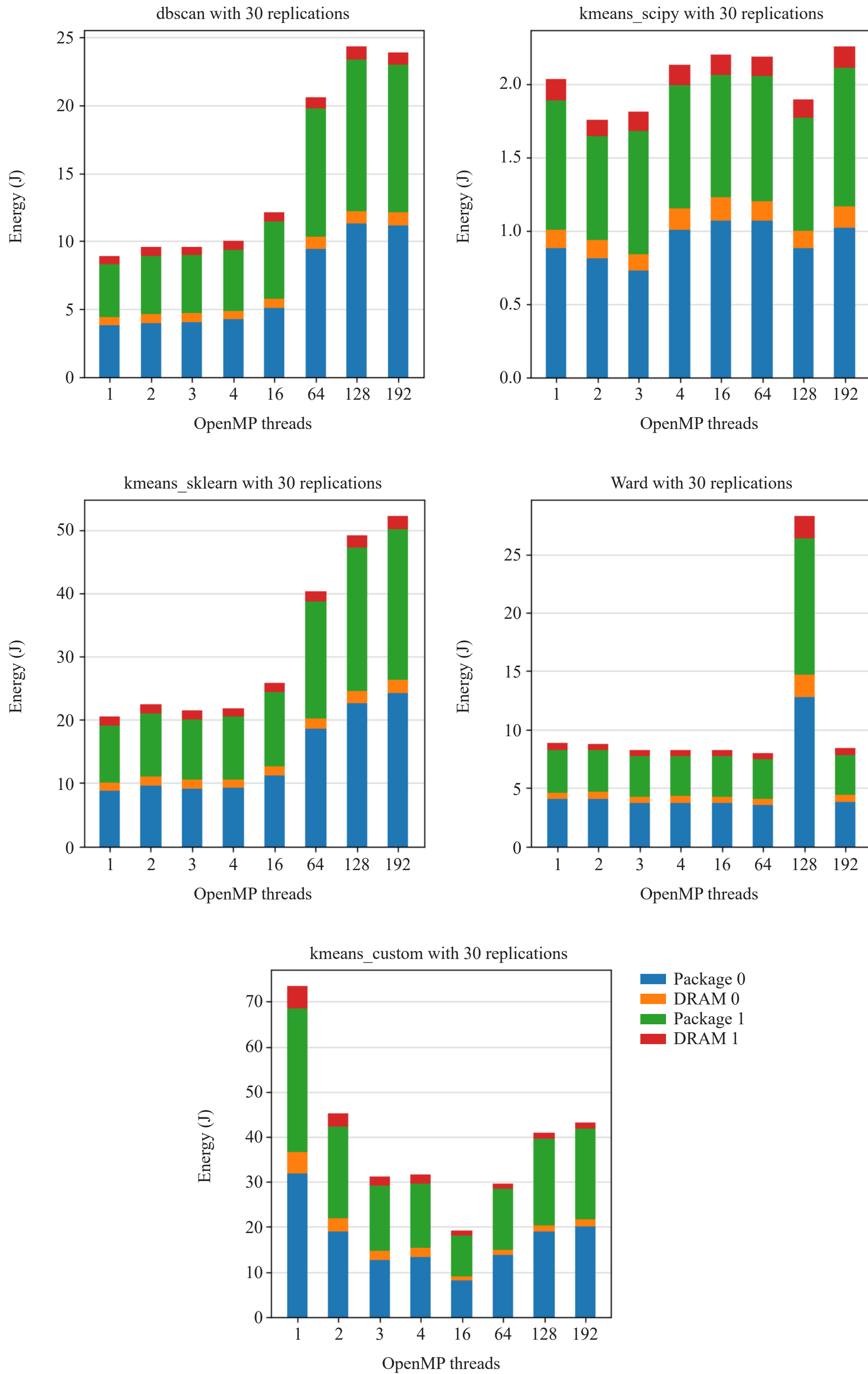

Figure B3. Energy consumption of each algorithm on Wine data depending on the number of OpenMP threads used

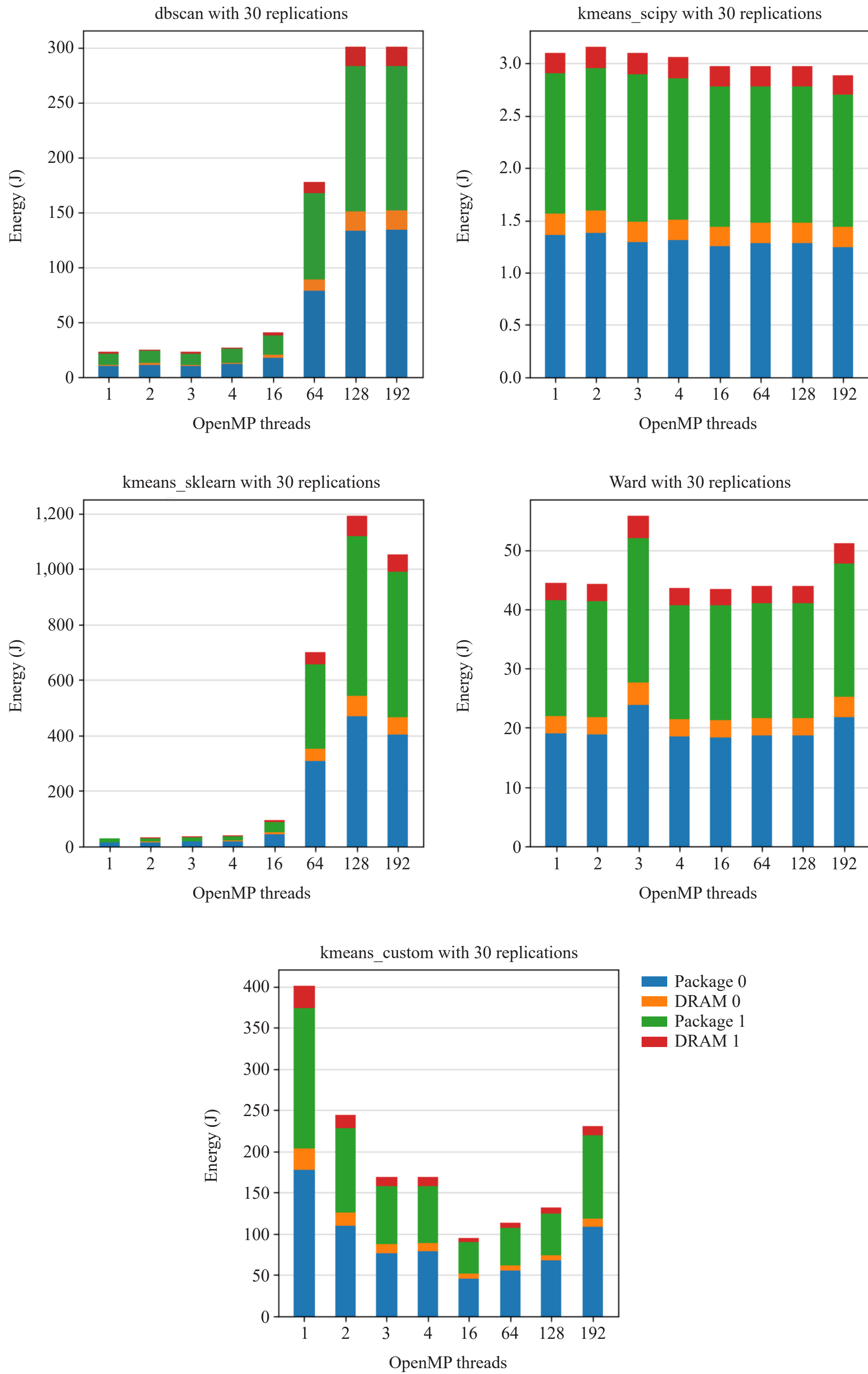

Figure B4. Energy consumption of each algorithm on Breast cancer data depending on the number of OpenMP threads used

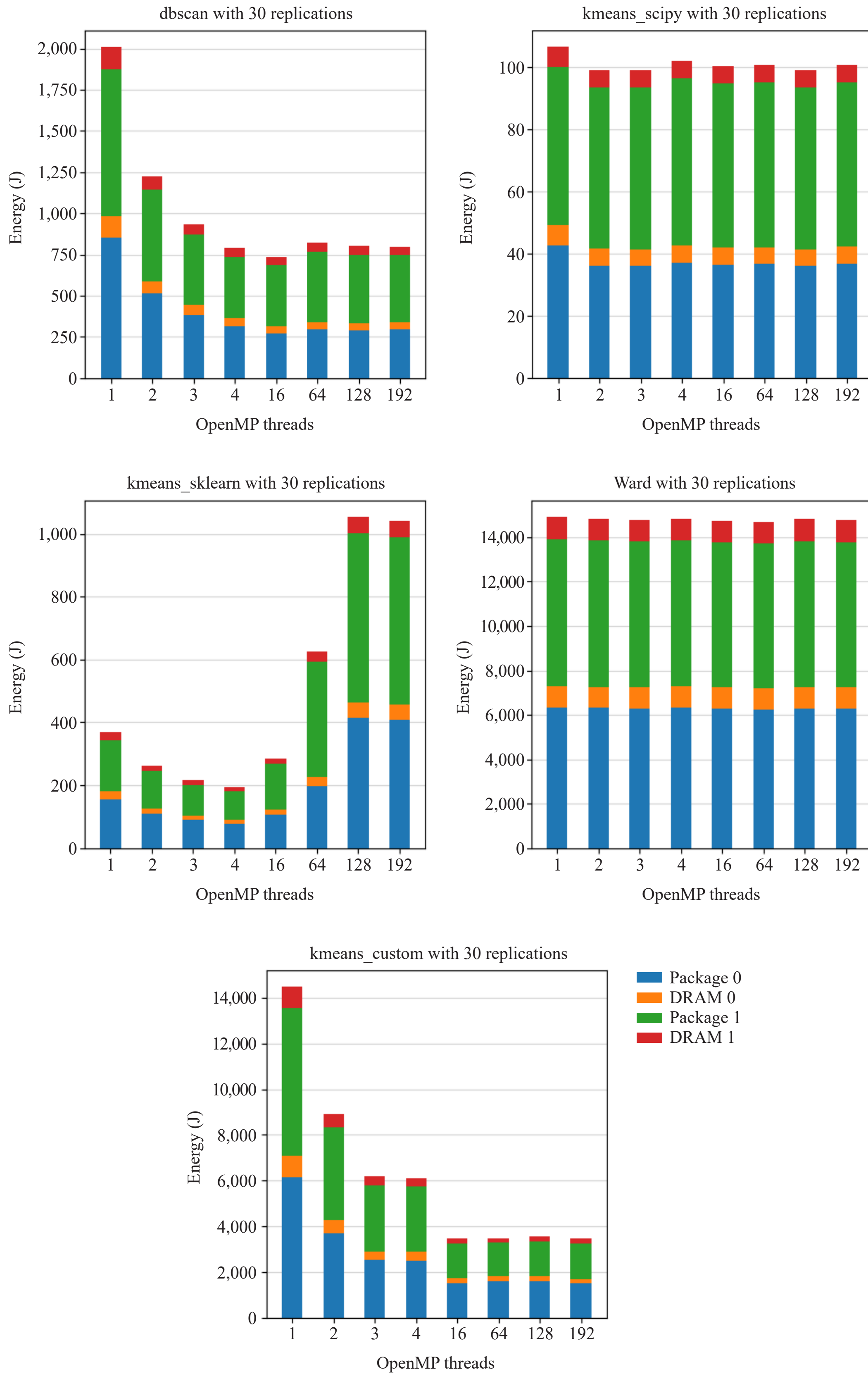

Figure B5. Energy consumption of each algorithm on Taiwan data depending on the number of OpenMP threads used

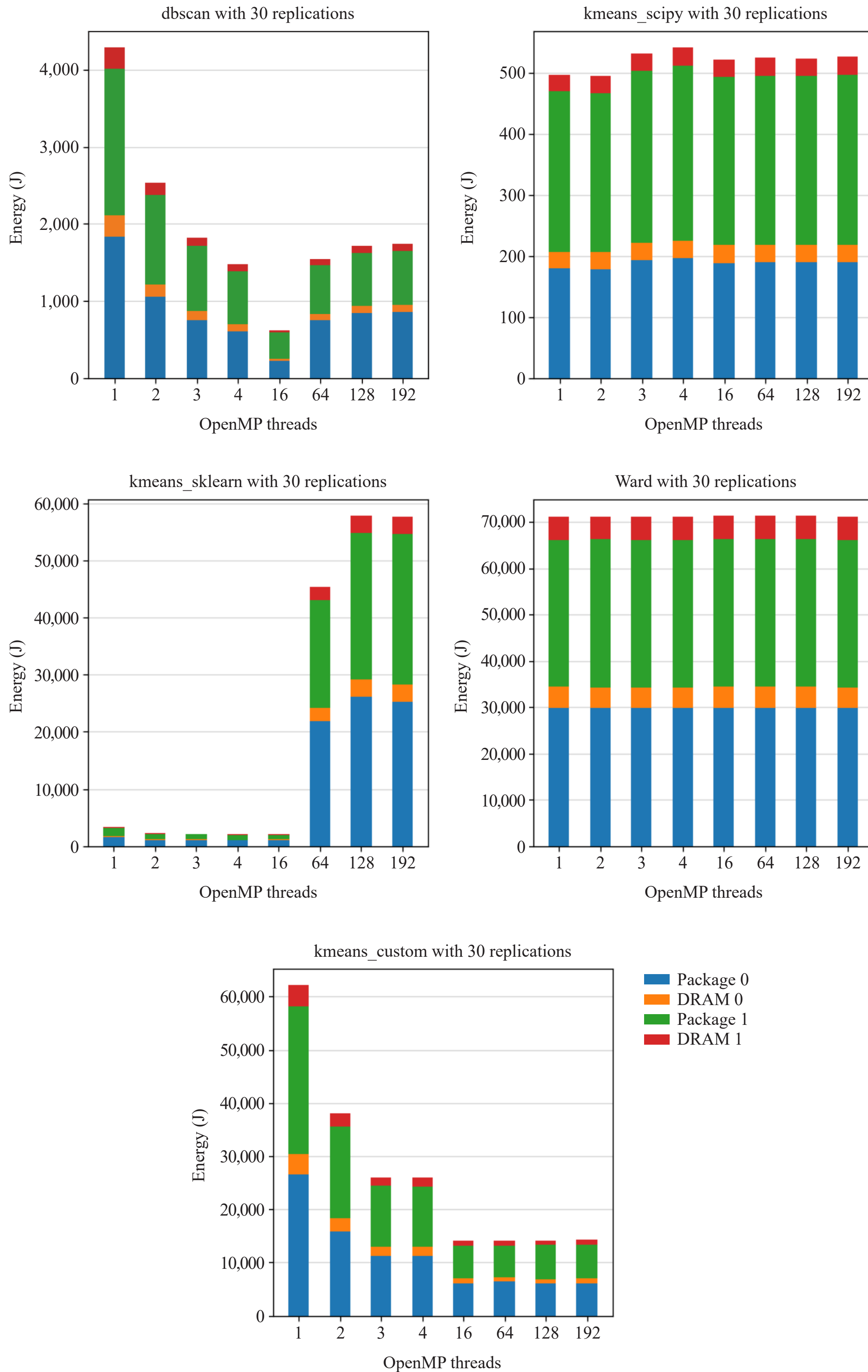

Figure B6. Energy consumption of each algorithm on Letter data depending on the number of OpenMP threads used

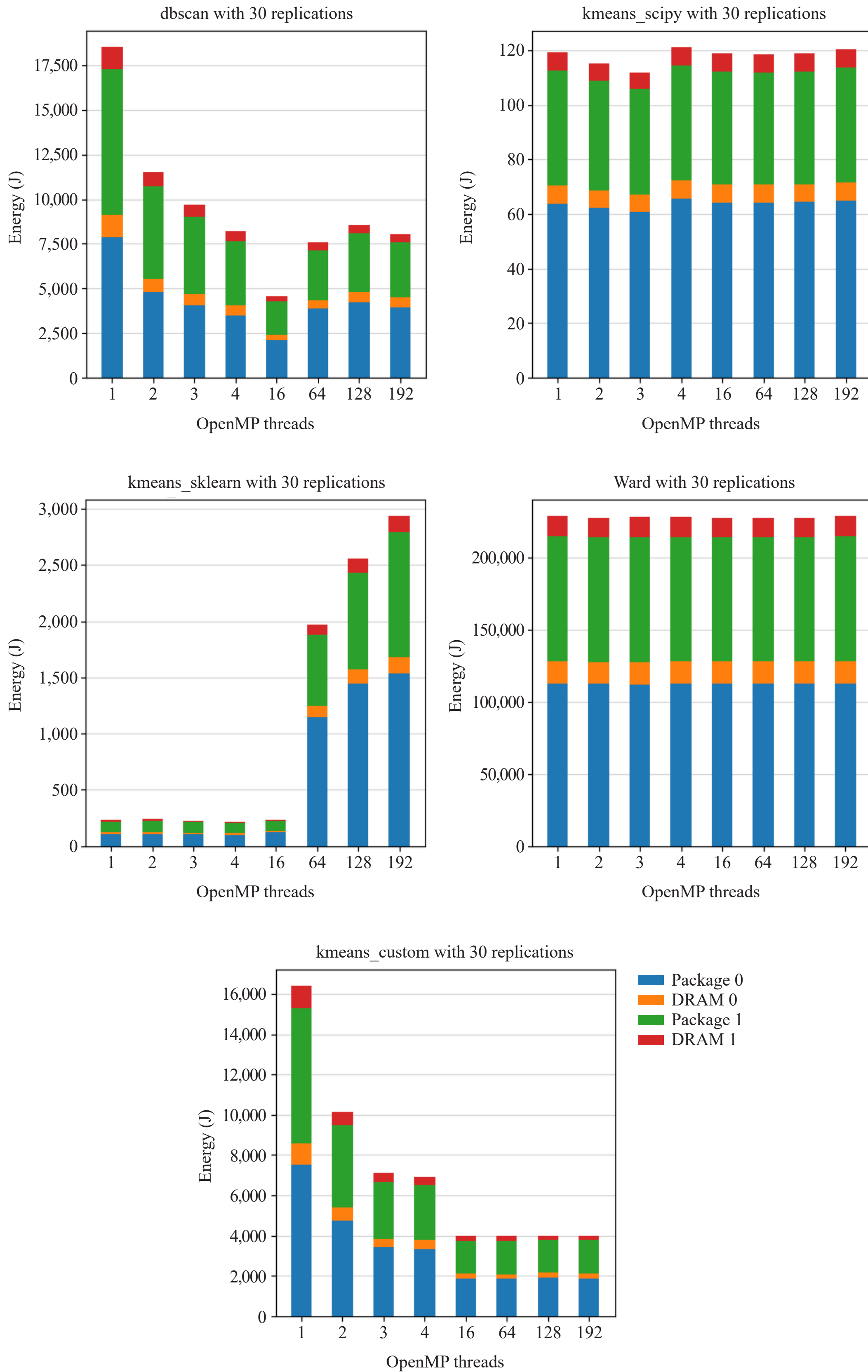

Figure B7. Energy consumption of each algorithm on Credit card data depending on the number of OpenMP threads used

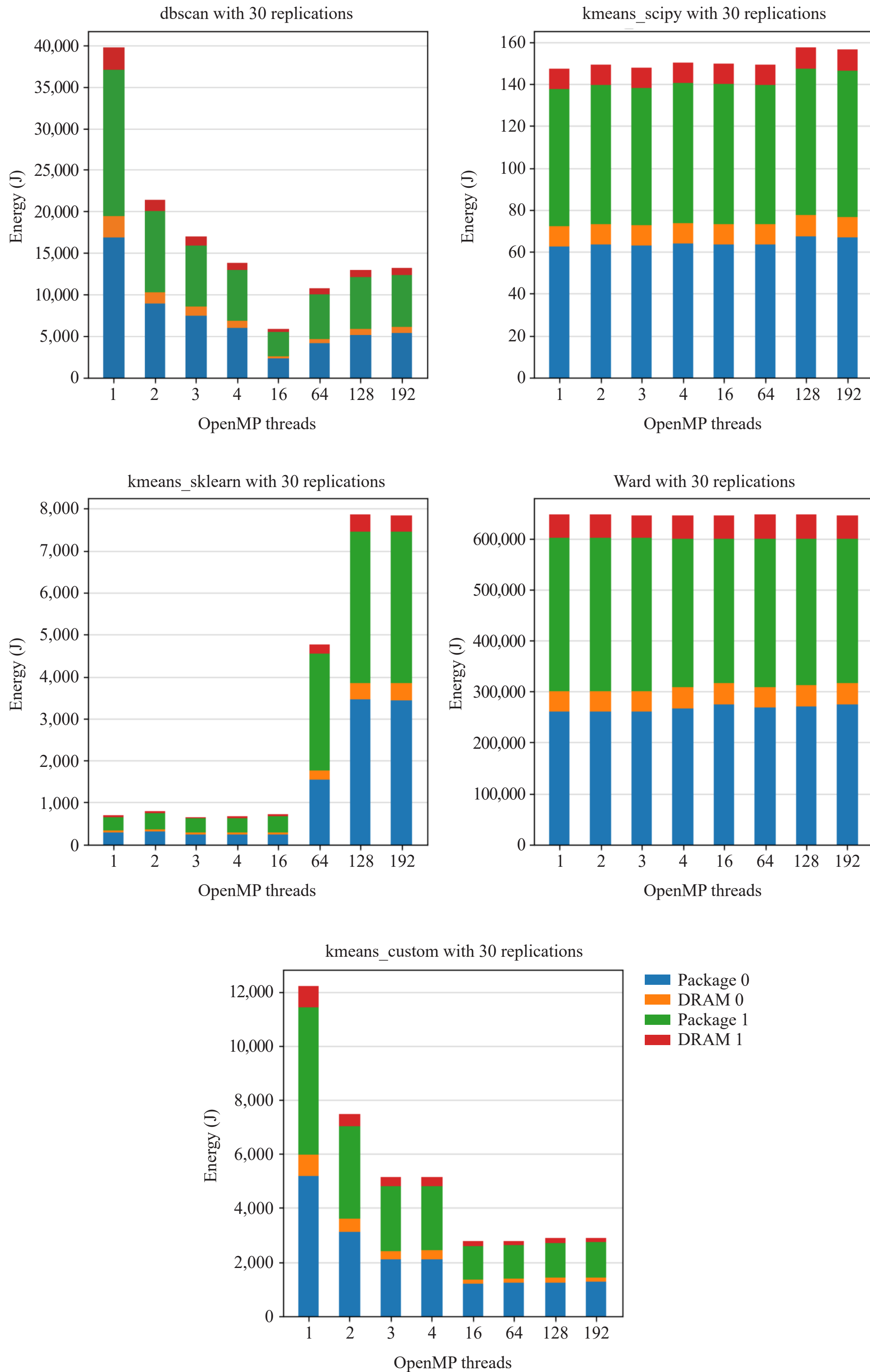

Figure B8. Energy consumption of each algorithm on Generated data depending on the number of OpenMP threads used