algorithms

MDPI

Article

Parallelization of the Bison Algorithm Applied to Data Classification

Simone A. Ludwig [1,*], Jamil Al-Sawwa [2] and Aaron Mackenzie Misquith [1]

[1] Department of Computer Science, North Dakota State University, Fargo, ND 58105, USA; aaron.misquith@ndsu.edu
[2] Department of Computer Science, Tafila Technical University, P.O. Box 179, Tafila 66110, Jordan; jalsawwa@ttu.edu.jo
* Correspondence: simone.ludwig@ndsu.edu

Abstract: In data science and machine learning, efficient and scalable algorithms are paramount for handling large datasets and complex tasks. Classification algorithms, in particular, play a crucial role in a wide range of applications, from image recognition and natural language processing to fraud detection and medical diagnosis. Traditional classification methods, while effective, often struggle with scalability and efficiency when applied to massive datasets. This challenge has driven the development of innovative approaches that leverage modern computational frameworks and parallel processing capabilities. This paper presents the Bison Algorithm, applied to classification problems. The algorithm, inspired by the social behavior of bison, aims to enhance the accuracy of classification tasks. The Bison Algorithm is implemented using PySpark, leveraging the distributed computing power to handle large datasets efficiently. This study compares the performance of the Bison Algorithm on several dataset sizes using speedup and scaleup as the performance measure.

Keywords: parallelization; spark; classification

check for updates

Citation: Ludwig, S.A.; Al-Sawwa, J.; Misquith, A.M. Parallelization of the Bison Algorithm Applied to Data Classification. *Algorithms* **2024**, *17*, 501. https://doi.org/10.3390/a17110501

Academic Editor: Seyed Jalaleddin Mousavirad

Received: 31 July 2024
Revised: 25 October 2024
Accepted: 29 October 2024
Published: 4 November 2024

1. Introduction

Optimization is a fundamental mathematical discipline that deals with the process of making something as effective or functional as possible. The roots of optimization can be traced back to ancient times when scholars and engineers used basic principles to solve practical problems. In modern times, optimization plays a crucial role in various fields, including engineering, economics, and operations research, where the goal is often to maximize or minimize a certain objective function.

Optimization problems can be broadly classified into different categories based on the nature of the objective function and the constraints. Linear programming, for example, deals with linear objective functions and linear constraints and has been extensively studied since the pioneering work of George Dantzig in the 1940s [1]. Nonlinear programming, on the other hand, involves nonlinear objective functions or constraints and presents additional challenges due to the complexity of the solution landscape.

In recent years, the advent of advanced computational techniques and the availability of large datasets have further expanded the scope and application of optimization. Machine learning, for instance, relies heavily on optimization algorithms to train models and improve predictive accuracy. Metaheuristic algorithms, such as genetic algorithms and particle swarm optimization, have also gained popularity for solving complex, real-world optimization problems where traditional methods may fall short [2,3].

Nature-inspired algorithms have garnered significant attention in the field of optimization due to their robustness and ability to find near-optimal solutions for complex problems. These algorithms are inspired by natural phenomena and processes, such as the behavior of biological species, the laws of physics, and the principles of natural selection. They offer an innovative approach to solving optimization problems, particularly those that are non-linear, non-differentiable, or multi-modal, where traditional methods often struggle [4].

One of the most well-known nature-inspired algorithms is the Genetic Algorithm (GA), introduced by John Holland in the 1970s. GAs mimic the process of natural selection by using operations such as selection, crossover, and mutation to evolve a population of candidate solutions [2]. These algorithms have been successfully applied to a wide range of optimization problems, from engineering design to machine learning and bioinformatics [2].

Ant Colony Optimization (ACO), proposed by Dorigo in the early 1990s, is another influential nature-inspired algorithm. ACO is based on the foraging behavior of ants and their ability to find the shortest path between their colony and a food source by depositing pheromones. This algorithm has been particularly effective in solving combinatorial optimization problems, such as the traveling salesman problem and network routing [5].

Optimization algorithms are fundamental in machine learning, particularly in classification tasks where the objective is to accurately assign labels to instances. Particle Swarm Optimization (PSO) is a well-known method that has been widely used due to its simplicity and effectiveness. However, PSO and similar algorithms often encounter issues such as local minima and convergence inefficiencies.

In many classification problems, regression-based approaches are employed to predict continuous values which are then thresholded to obtain discrete class labels. Traditional optimization algorithms like PSO are often used to optimize the weights in regression models. Despite their success, these algorithms can be improved by incorporating more sophisticated strategies for balancing exploration and exploitation.

This paper uses the Bison Algorithm [6]. The algorithm is inspired by the collective behavior of bison herds, which exhibit efficient exploration and exploitation strategies during migration. By mimicking these strategies, we aim to develop an optimization algorithm that can better navigate the search space, avoid local minima, and achieve higher accuracy in classification tasks. A distinctive feature of the Bison Algorithm is its division into two groups: swarmers and runners. The swarmers focus on global exploration of the search space, while the runners concentrate on local exploitation. This dual approach allows the algorithm to perform exploration and exploitation simultaneously in every iteration, thereby overcoming the drawbacks of PSO related to local minima and convergence inefficiencies. A key feature of the Bison Algorithm is its implementation using PySpark, a robust framework for big data processing that allows for parallel computation across distributed systems.

Through this study, we aim to contribute to the growing body of knowledge in machine learning by introducing an efficient and scalable solution for classification tasks. The Bison Algorithm not only bridges the gap between regression and classification but also exemplifies the power of parallel computing in enhancing algorithmic performance.

The structure of this paper is as follows: Section 2 provides a detailed overview of related work in the fields of classification algorithms and parallel processing. Section 3 describes the theoretical foundation of the Bison Algorithm and its implementation details. Section 4 presents experimental results demonstrating the algorithm's performance on various benchmark datasets. Finally, Section 5 discusses the implications of these results and potential avenues for future research.

2. Related Work

Parallelization techniques for nature-inspired algorithms are crucial for handling large datasets and computationally intensive tasks. MPI (Message Passing Interface) is one of the early frameworks used for this purpose. It provides a standard for communication between multiple nodes in a distributed system, enabling efficient parallel computations. Several studies have explored using MPI to parallelize nature-inspired algorithms. For instance, a parallel GA based on the MPI environment was developed, adapting the serial GA to a parallel implementation [7]. Another example is the MPI-OpenMP hybrid approach used to implement a multi-objective Particle Swarm Optimization (PSO) algorithm, combining distributed memory (MPI) and shared memory (OpenMP) parallelization [8]. Additionally, an MPI-based parallel GA was proposed for multiple geographical optimization problems, using a hybrid of fixed-position and sliding models.

MapReduce, introduced by Dean and Ghemawat [9], revolutionized the way large-scale data processing tasks are handled by abstracting the data processing into two fundamental operations: Map and Reduce. This framework has been effectively used to parallelize algorithms like Genetic Programming and Particle Swarm Optimization. For example, a MapReduce-based Particle Swarm Optimization algorithm was developed to handle large-scale clustering tasks, showing notable improvements in processing time and scalability [10].

Apache Spark, an advancement over MapReduce, offers in-memory computation, which significantly speeds up data processing tasks. For instance, Ludwig and Miryala [11] compared the performance of Glow-worm Swarm Optimization (GSO) when implemented using Spark and MapReduce. Their findings indicated that Spark's in-memory computation capabilities provided faster and more efficient processing, making it more suitable for high-dimensional function optimization tasks.

Ludwig involved the implementation of a parallel fuzzy clustering algorithm using Spark. This study demonstrated that the Spark-based algorithm outperformed traditional clustering methods, providing better scalability and handling of real-time data streams [12]. This research highlights Spark's capability to efficiently process and analyze large datasets, making it a preferred choice for implementing nature-inspired algorithms in a parallel computing environment.

Al-Sawwa implemented a scalable design of an artificial bee colony for big data classification using Apache Spark [13]. The performance results reveal that the proposed fitness algorithm can efficiently deal with unbalanced datasets and can scale very well, achieving excellent speedup and scaleup results. Similar implementations were introduced for the Differential Evolution (DE) [14] and the PSO algorithms [15].

GPU-based parallelization, although not specifically MPI, MapReduce, or Spark, is another common approach. Tan conducted a survey focused on GPU-based parallel implementations of Swarm Intelligence algorithms, including GA and Differential Evolution [16]. Similarly, Kroemer et al. presented a survey specifically on GPU-based parallel implementations of Particle Swarm Optimization (PSO) [17]. Lalwani et al. surveyed parallel swarm optimization algorithms, covering CPU- and GPU-based implementations using frameworks like OpenMP, MPI, and R, though focused specifically on PSO [18]. A review of parallelization strategies for Swarm Intelligence algorithms identified particle-level parallelization as a common approach, easily implemented using OpenMP for multi-core CPUs.

Overall, the integration of MPI, MapReduce, and Spark into the parallelization of nature-inspired algorithms has shown significant improvements in performance and scalability. These frameworks provide the necessary tools to handle the growing complexity and size of data in various computational intelligence applications.

In this paper, we have parallelized the Bison Algorithm [6], which is another nature-inspired algorithm applied to optimization problems. We have used the Bison Algorithm and first of all, applied it to the classification problem in data mining, and secondly, parallelized the code using the Spark framework to investigate its efficiency.

3. Proposed Approach

This section starts with an overview of the Bison Algorithm, followed by an introduction to Apache Spark, and the details about the parallel Bison Algorithm implementation provided at the end of this section.

3.1. Overview of Bison Algorithm

The Bison Algorithm is inspired by the collective behavior of bison herds, which exhibit efficient exploration and exploitation strategies during migration. The algorithm divides the bison herd into elite, swarmer, and runner groups, each contributing uniquely to the optimization process.

The following is the overview description of the Bison Algorithm.

1. Initialization: A population of bisons is initialized with random positions in the search space.
2. Fitness Evaluation: The fitness of each bison is evaluated using a predefined fitness function.
3. Elite Group: The best-performing bisons are selected as the elite group, representing the best solutions found so far.
4. Swarmers: The positions of swarmer bisons are adjusted towards the elite center, promoting exploitation.
5. Runners: Runner bisons explore the search space randomly, promoting exploration.
6. Iteration: The process is repeated for a predetermined number of iterations or until convergence criteria are met.

The detailed algorithm description is the following applied to the classification task:

1. Initialization:
   - Generate a random population of bisons within the defined search space bounds.
   - Define the number of iterations (*max_iter*) and the size of the bison population (*swarm_size*).
2. Fitness Function:
   - The fitness function evaluates the accuracy of the classification model. In this case, it is defined as the negative accuracy to facilitate minimization.
3. Iteration Loop:
   - For each iteration:
     (a) Evaluate the fitness of each bison.
     (b) Update the best-known positions (elite group).
     (c) Calculate the elite center as the mean position of the elite group.
     (d) Adjust the positions of swarmer bisons towards the elite center with a random step size.
     (e) Allow runner bisons to explore randomly within the search space bounds.
4. Convergence:
   - The process continues until the maximum number of iterations is reached or convergence criteria are met.

3.2. Apache Spark

Apache Spark is a popular open-source analytics engine used for large-scale data processing [19]. Spark uses parallelization, which is a computation technique used to divide a task into smaller, independent sub-tasks that can be executed simultaneously across multiple processors, or cores. Parallelization aims to increase computational speed by reducing the overall time required to complete a task. By distributing a workload across multiple cores, performing operations such as classification can be conducted in parallel to ease computational load.

The main program that runs the user application is called the driver program, which defines the high-level logic of the application, including details about the execution of tasks. The cluster manager manages the cluster of machines, or nodes, and allocates resources to the Spark application.

Worker nodes are the machines in the cluster that execute the tasks assigned by the driver program. Each worker node runs an executor that executes the individual tasks. Apache Spark leverages distributed computing by breaking down large data processing tasks into smaller ones that are executed in parallel across clusters of computers, or cores, (seen in Figure 1), enabling efficient handling of large amounts of data processing, improving speed and scalability [20].

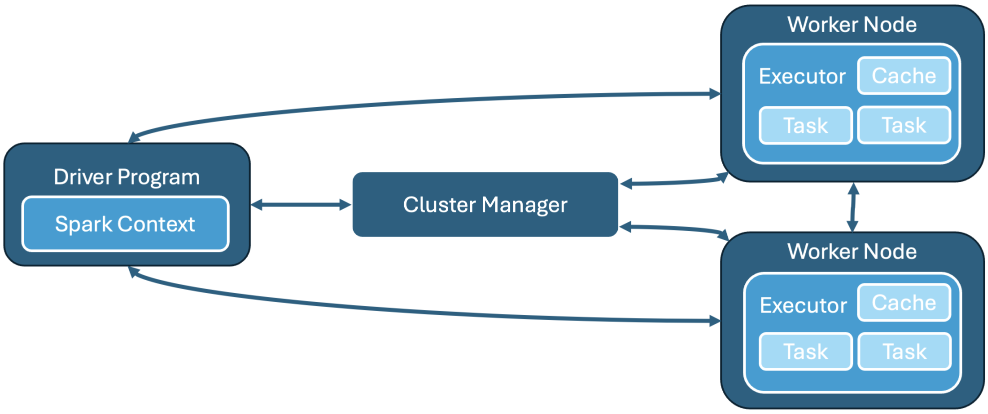

Figure 1. Spark's data processing overview.

3.3. Bison Algorithm for Classification Implemented with Spark

The Bison Algorithm for classification implemented with Spark can be thought of an iterative algorithm that uses a swarm of solutions (bisons) to find the best way to classify data points in a dataset. The algorithm improves over time by learning from the best-performing bisons, adjusting the positions of the others, and testing the resulting model on unseen data, thus, achieving a high level of accuracy in clustering and classifying data.

In order to enable the Bison Algorithm to perform the classification task, a few steps need to be taken. The algorithm combines elements of classification and clustering to classify data points. The process starts with a setup phase where a configuration file is read to determine key parameters, such as how many "bisons" (representing possible solutions) will be used, how many iterations the algorithm should run, and the paths to the necessary input data files. These parameters are essential as they dictate the behavior of the algorithm, including how it processes data and how much computational power it allocates.

After the setup is complete, the algorithm moves into the data preparation phase. Here, the dataset is loaded, and these data consist of features and the classification label, which classifies each point. To ensure that all features contribute equally during processing, the algorithm scales them so that no single feature can dominate due to its magnitude. The labels are also converted into numerical form. Then, the dataset is split into two parts: one part will be used for training, and the other will be used for testing.

Then, the swarm of bisons is initialized. Please note that each bison represents a random point in the search space, and these points act as potential centroids around which data points can be clustered. Essentially, each bison is a "guess" for where groups of similar data points might gather in the feature space. The goal of the algorithm is to improve these centroid locations over time converging to the optimal centroid position.

During the training phase, the algorithm goes through several iterations where it evaluates how well the bisons are classifying the data points. For each data point, the algorithm calculates which bison's centroids are closest to the point using a distance measure.

If the centroids associated with a bison misclassify a point, that bison receives a penalty. Over time, bisons with fewer penalties are considered better classifiers since they better assign points to clusters. Once the algorithm has evaluated the bisons in an iteration, it updates their positions. The bisons that performed best in this round are designated as elite bisons. These elite bisons represent the best solutions at this stage. The remaining bisons adjust their positions based on these elite bisons. Some bisons, called swarmers, move closer to the elite bisons, trying to emulate their success. Others (runners), move in random directions, exploring new parts of the solution space that may provide better results. This balance between exploration and exploitation allows the algorithm to avoid getting stuck in local optima and helps to search more broadly for better centroids.

After completing the training phase, the algorithm selects the best-performing bison, which had the lowest classification error across the iterations. This best bison represents the most accurate model for classifying the dataset. The next step is the testing phase where the best bison is applied to the testing dataset in order to test the model in classifying new data points. After the classification is completed, the algorithm evaluates its performance using several metrics (more description in Section 4.3).

The algorithm leverages Spark's distributed computing capabilities to perform data preprocessing, training, and testing in parallel. It distributes the data across multiple worker nodes, processes it simultaneously, and aggregates the results using accumulators and broadcast variables. This parallelization allows the algorithm to scale efficiently and handle large datasets without sacrificing speed or accuracy. Spark ensures that the algorithm can take advantage of cluster resources, making the training and testing phases much faster than they would be on a single machine.

The algorithm uses Apache Spark to parallelize both the data processing and the training of the Bison model by distributing tasks across multiple nodes in a cluster. When the dataset is loaded, it is stored as a DataFrame, which Spark automatically splits and distributes across different worker nodes. Each node processes a portion of the data in parallel. For example, feature scaling, label encoding, and data splitting are all performed concurrently on each partition of the dataset. After the initial data preparation, the DataFrame is converted into an RDD (Resilient Distributed Dataset), which allows the algorithm to perform custom operations such as mapping and reducing in parallel.

In the training phase, the swarm of bisons (candidate solutions) is broadcast to all worker nodes, ensuring that every node has access to the current bison positions without redundant data transfers. Each worker node evaluates how well these bisons classify the data points in its partition and updates the accumulator, a Spark variable that aggregates the misclassification counts from all nodes. This allows the algorithm to compute fitness (classification accuracy) in parallel, as each node processes its data independently. Once the fitness of each bison is evaluated, the bisons are updated, and the next iteration begins. This parallel approach allows the algorithm to scale efficiently, making it capable of handling large datasets and complex computations faster than if it were run on a single machine. The algorithm description is provided in Algorithm 1.

Algorithm 1: Bison Algorithm with RDD.

Initialize the Bison group randomly
Initialize the Runner group around x_{best}
Initialize the run direction vector
Read the dataset into an RDD
Initialize Accumulator Accu
loop
    Broadcast Bison group
    for each element in RDD **do**
        Assign the element to the closest centroids
        Update Accu
    end for
    Update the fitness of the bisons based on the values in Accu
    Update x_{best}
    Update Bison Group according to the original algorithm
    Reset Accu
end loop

Figure 2 shows how the dataset is partitioned and then used by the Spark framework. Each partition is sent to an executor, which performs the closest centroid calculations.

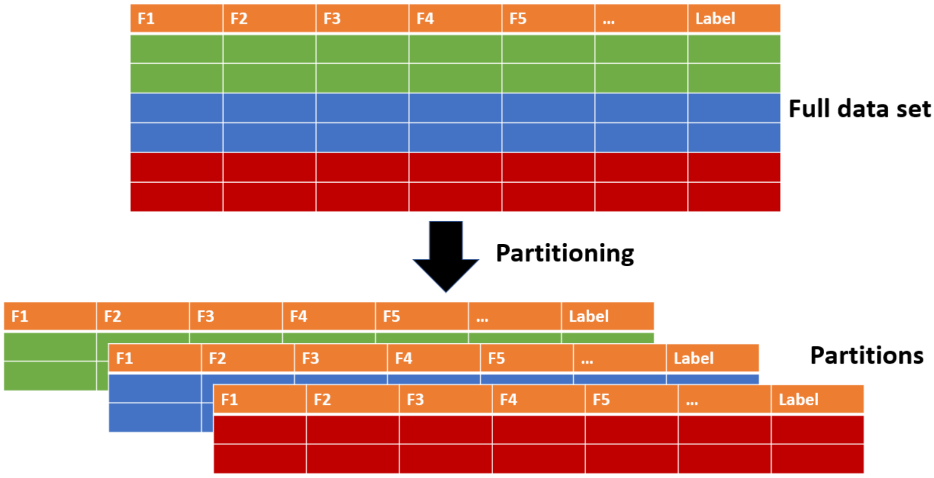

Figure 2. Partitioning of dataset.

3.4. Time Complexity Analysis

The time complexity of the Spark Bison Algorithm when run in a serial manner is

$$\mathcal{O}(T \cdot (|B| + n \cdot k)).$$

The time complexity of the Spark Bison Algorithm when run in parallel is

$$\mathcal{O}\left(T \cdot \left(|B| + \frac{n \cdot k}{p}\right)\right),$$

where

T is the number of iterations of the algorithm
$|B|$ is the number of bisons (or centroids) in the Bison group
n is the number of elements in the dataset
k is the number of centroids (or bisons) to which elements are assigned
p is the number of partitions (or parallel workers in the cluster) used by Apache Spark.

4. Experimental Setup

This section starts off with a description of the dataset and its pre-processing, followed by the experimental setup as well as the results obtained from the experiments.

4.1. Dataset Description

The dataset used for our experiments was carefully pre-processed to ensure it was suitable for the Bison Algorithm and to maintain the integrity of the classification task. The Mice Protein Expression dataset contains measurements from an experiment conducted on mice to study the effects of trisomy (Down syndrome) on protein expression in the cerebral cortex [21]. The dataset used was the Mice Protein Expression dataset, which contains measurements from an experiment conducted on mice to study the effects of trisomy (Down syndrome) on protein expression in the cerebral cortex. Seventy-two mice were studied and their measurements were recorded. The goal for the collection of this dataset was to analyze protein expression and behavior, predict whether a mouse underwent behavioral testing or received drug treatment, and assess the impact of trisomy on protein levels. We have opted to use the binary outcome behavior as the classification variable. Table 1 contains more descriptions.

Table 1. Dataset characteristics for the study on expression levels of proteins in the cerebral cortex.

Characteristic	Details
Expression levels	77 proteins measured
Region	Cerebral cortex
Classes	8 (control and Down syndrome mice)
Exposure	Context fear conditioning
Task	Assessing associative learning
Dataset Characteristics	Multivariate
Subject Area	Biology
Associated Tasks	Classification, clustering
Feature Type	Real
Instances	1080
Features	80

The following steps show the pre-processing steps taken.

4.1.1. Data Cleaning

Initially, any row with a missing value was removed from the dataset. This step was crucial to prevent the introduction of biases and inaccuracies that can arise from imputing or otherwise handling missing data improperly. The resulting dataset was free of any missing values, ensuring a clean baseline for further processing.

4.1.2. Data Normalization

To facilitate the optimization process and enhance the performance of the algorithm, the entire dataset was normalized within the range of 0 to 1. This scaling process was conducted to ensure that all features contributed equally to the model, preventing any single feature from dominating due to its scale.

4.1.3. Class Selection

For the purpose of this study, only the instances belonging to the class 'behavior' were used. This choice was driven by the specific objectives of our classification task, focusing on accurately predicting behavioral outcomes based on the provided features.

4.1.4. Dataset Folding

In order to conduct scaling experiments, we utilized a folding technique to artificially expand the dataset. This method involved repeating the entire set of instances multiple times. After the initial pre-processing steps, the number of instances in the dataset was reduced to 552. To simulate larger datasets and assess the performance of our algorithm under varying scales, we created folds of the dataset at different scales. Specifically, we created datasets with 1000, 2000, 3000, 4000, 5000, 8000, 16,000, 32,000, and 64,000 folds of the original 552 instances. This approach allowed us to evaluate the scalability and efficiency of the Bison Algorithm in handling larger datasets systematically. Table 2 shows the generated datasets with name, number of folds, number of instances and size in GB.

4.2. Setup of Experiments

The pre-processed and folded datasets were then used to run a series of experiments aimed at testing the Bison Algorithm's performance. Each experiment involved initializing the Bison Algorithm with the same parameters and running it on datasets of increasing size to observe how well the algorithm scaled and how quickly it could produce accurate classifications. The results from these experiments provide valuable insights into the algorithm's ability to handle large-scale data and its potential application in real-world scenarios.

By carefully pre-processing the data and employing a methodical approach to scaling the dataset, we ensured that our experiments were both robust and relevant, providing meaningful results that contribute to the understanding and improvement of the Bison Algorithm for classification tasks.

Table 2. Generated datasets.

Dataset Name	Number of Folds	Number of Instances	Size in GB
1000	1000	552,000	0.5
2000	2000	1,104,000	1.0
3000	3000	1,656,000	1.5
4000	4000	2,208,000	2.0
5000	5000	2,760,000	2.5
8000	8000	4,416,000	4.0
16,000	16,000	8,832,000	8.0
32,000	32,000	17,664,000	16.0
64,000	64,000	35,328,000	32.0

4.3. Performance Metrics

The performance of the algorithm was assessed using the following metrics:

Accuracy: The proportion of correct predictions among the total number of examples.
Precision: The proportion of true positive predictions among the total positive predictions.
Recall: The proportion of true positive predictions among the total actual positives.
F1-score: The harmonic mean of precision and recall, providing a single metric that balances both concerns.

In addition, the speedup measures the parallelization ability of the algorithm by taking the ratio of the running time on a single node to the running time on parallel nodes n. The speedup is calculated as follows, where the dataset size is fixed while the number of nodes is increased in a certain ratio.

$$Speedup = \frac{T_1}{T_n}, \tag{1}$$

where T_1 is the running time using a single node, and T_n is the running time using n nodes.

The scaleup measures how the cluster of nodes are utilized efficiently by the parallel algorithm. The scaleup is calculated as follows, where the dataset size and the number of nodes are increased by the same ratio.

$$Scaleup = \frac{T_{sn}}{T_{Rsn}}, \tag{2}$$

where T_{sn} is the running time for the dataset with size s using n nodes, R is the increasing ratio, and T_{Rsn} is the running time for the dataset with size Rs using Rn nodes.

4.4. Infrastructure for Running Experiments

The nodes used for the experiments were the Pittsburgh Supercomputing Center's Bridges-2 system, which is equipped with two AMD EPYC 7742 CPUs, each featuring 64 cores (128 threads total per node) with a clock speed between 2.25 GHz and 3.40 GHz. These nodes have either 512 GB of RAM and 3.84 TB of local NVMe SSD storage. The nodes are connected via Mellanox ConnectX-6 HDR InfiniBand with 200 Gbps bandwidth, providing efficient communication across the system, which is key for high-performance computing tasks such as used in this study.

5. Results

This sections shows the results of the experiments. First, we run the experiments measuring the execution time to calculate the speedup for datasets 1000, 2000, 3000, 4000, and 5000. Afterwards, we run experiments to measure the execution time to calculate the scaleup for datasets 1000, 2000, 4000, 8000, 16,000, and 32,000. The parameters for the Bison Algorithms were set to number of bisons = 100 and the number of iterations = 100.

Table 3 shows the results in terms of the performance measures for the classification task. Accuracy, precision, recall, and F1-Score are listed. As can be seen, very good results for each measure are achieved. Also, the increasing dataset size does not influence any of the measures negatively.

Table 3. Accuracy, precision, recall, and F1-score.

Dataset	Accuracy	Precision	Recall	F1-Score
1000	0.97826087	0.98850575	0.96629214	0.97727273
2000	0.97435987	0.98671365	0.96598933	0.97628493
3000	0.97963341	0.98754650	0.99678469	0.97863215
4000	0.97571463	0.98943264	0.97542914	0.97498216
5000	0.97738793	0.98547925	0.96749315	0.97769850
8000	0.97884539	0.98896458	0.98326430	0.97849352
16,000	0.97554697	0.98478950	0.97394583	0.97978316
32,000	0.97832146	0.98736512	0.98323540	0.97769815
64,000	0.97634821	0.98793543	0.98674513	0.97856435

In terms of execution time and speedup for the 1000 dataset, the execution time and speedup are provided in Table 4 and Figure 3. As can be seen, an exponential trend for the execution time is achieved, with the speedup showing closer alignment to the ideal speedup (shown as the red-dashed line) up to 64 nodes. For more than 64 nodes we can see that the speedup drifts off from the ideal speedup.

Table 4. Execution time and speedup for the 1000 dataset.

Number of Nodes	Execution Time (s)	Speedup
1	25,946.03	1.00
2	13,571.13	1.91
4	6632.55	3.91
8	3197.00	8.11
16	1592.43	16.29
32	814.96	31.83
64	418.65	61.97
128	270.63	95.87

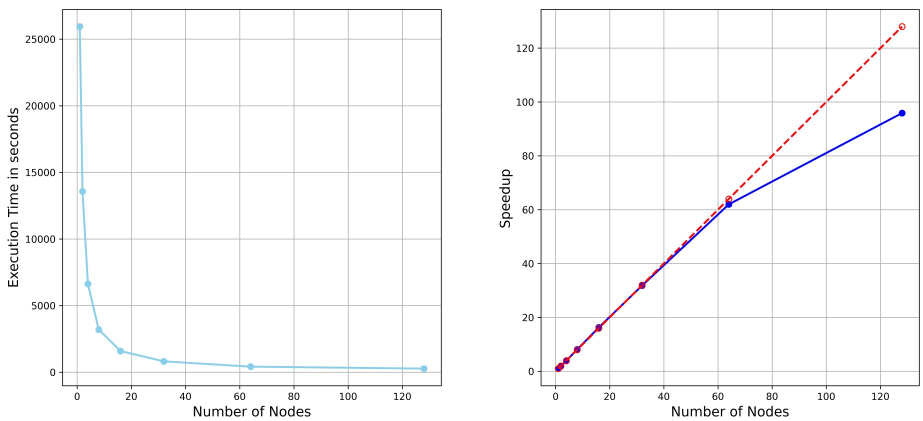

Figure 3. Execution time and speedup for the 1000 dataset.

Similar trends can be observed for the execution time and speedup for the 2000 dataset. The execution time and speedup are provided in Table 5 and Figure 4.

Table 5. Execution time and speedup for the 2000 dataset.

Number of Nodes	Execution Time (s)	Speedup
1	51,449.84	1.00
2	27,051.38	1.90
4	13,148.17	3.91
8	6442.22	7.98
16	3292.42	15.62
32	1673.75	30.73
64	864.14	59.53
128	525.92	97.82

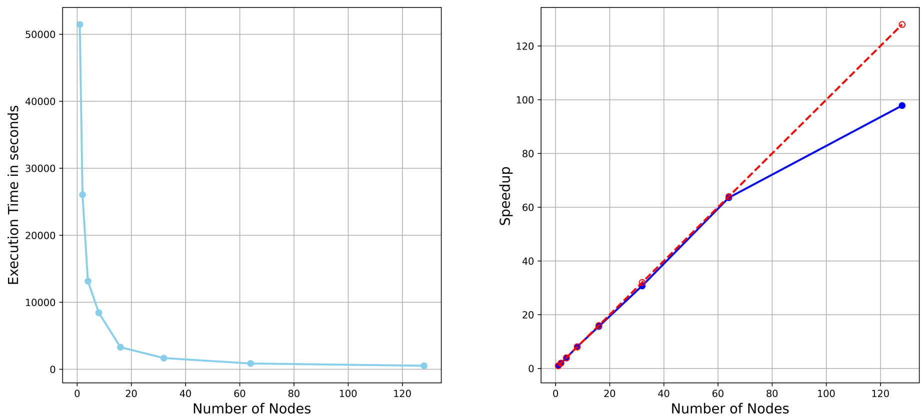

Figure 4. Execution time and speedup for the 2000 dataset.

Moreover, for the 3000 dataset, the execution time and speedup are provided in Table 6 and Figure 5.

Table 6. Execution time and speedup for the 3000 dataset.

Number of Nodes	Execution Time (s)	Speedup
1	78,477.07	1.00
2	40,198.49	1.95
4	21,006.39	3.73
8	10,542.96	7.44
16	5343.68	14.68
32	2498.81	31.40
64	1258.43	62.36
128	765.07	102.57

The execution time and speedup for the 4000 dataset are provided in Table 7 and Figure 6.

Table 7. Execution time and speedup for the 4000 dataset.

Number of Nodes	Execution Time (s)	Speedup
1	103,606.60	1.00
2	52,226.85	1.98
4	26,583.26	3.89
8	13,039.79	7.94
16	8491.96	12.20
32	3245.16	31.92
64	1661.19	62.36
128	991.96	104.44

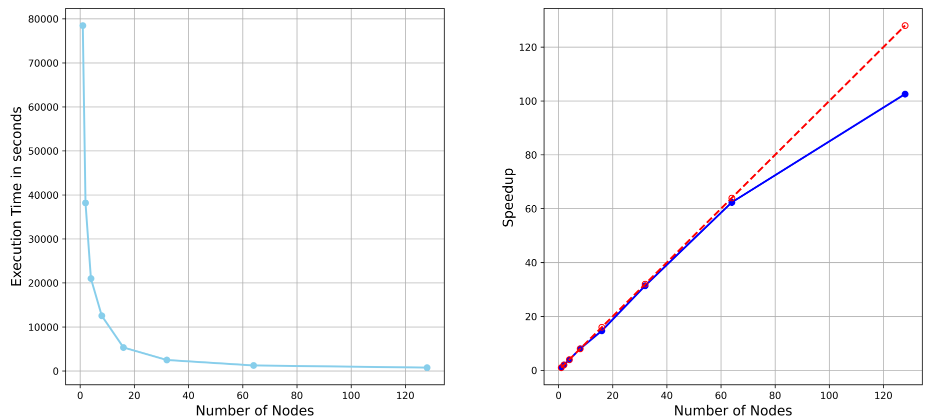

Figure 5. Execution time and speedup for the 3000 dataset.

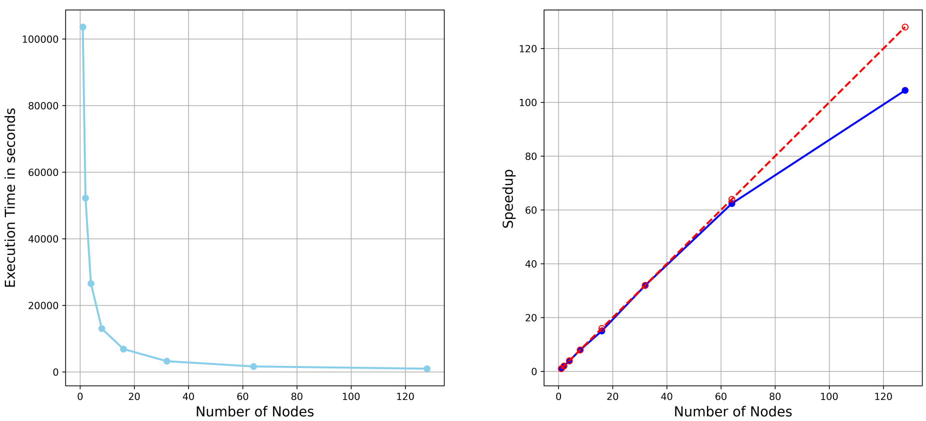

Figure 6. Execution time and speedup for the 4000 dataset.

The execution time and speedup are provided in Table 8 and Figure 7 for the 5000 dataset. What we can see is that with increasing dataset sizes, the speedup slightly improves, as expected.

Table 8. Execution time and speedup for the 5000 dataset.

Number of Nodes	Execution Time (s)	Speedup
1	129,357.60	1.00
2	64,471.76	2.00
4	33,343.78	3.87
8	17,415.78	7.42
16	8620.33	15.00
32	4139.66	31.24
64	2129.46	60.74
128	1237.78	104.50

Last but not least, the scaleup results are are provided in Table 9 and Figure 8. Scaleup evaluates how well our parallel implementation of the Bison Algorithm handles increasing dataset sizes as more processing nodes (executors) are added. It basically measures the algorithm's ability to handle proportionally larger dataset sizes when the processing nodes are scaled up. As can be seen from the table and the figure, we observe a stable scaleup for up to dataset size of 32,000-fold using 64 processing nodes (scaleup = 0.96), with a scaleup declining for the 64,000-fold dataset using 128 processing nodes (scaleup of 0.85). The scaleup using four nodes and eight nodes is slightly higher than 1.0, with values of

1.3 and 1.08, respectively. This could be due to a few reasons, such as improved resource utilization, decreased contention, and more optimal task scheduling.

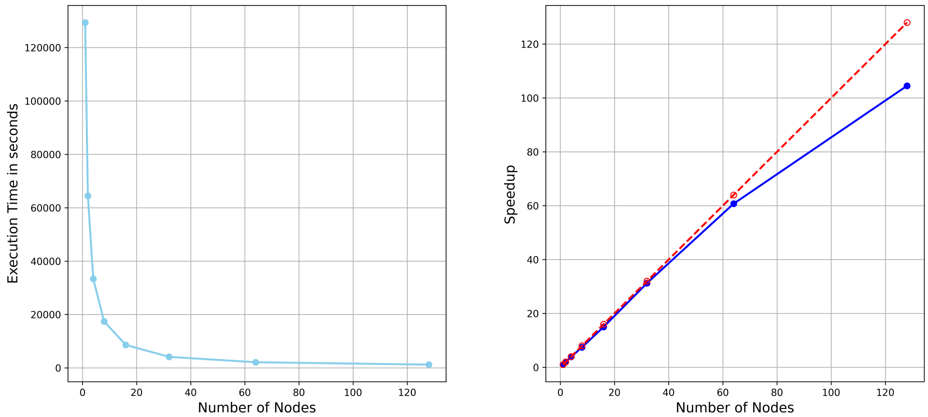

Figure 7. Execution time and speedup for 5000 dataset.

Table 9. Scaleup for all datasets.

Dataset	Number of Nodes	Execution Time (s)	Scaleup
1000	2	13,571.13	1.00
2000	4	13,148.17	1.03
4000	8	12,542.96	1.08
8000	16	13,724.50	0.99
16,000	32	13,758.79	0.99
32,000	64	14,148.23	0.96
64,000	128	15,966.44	0.85

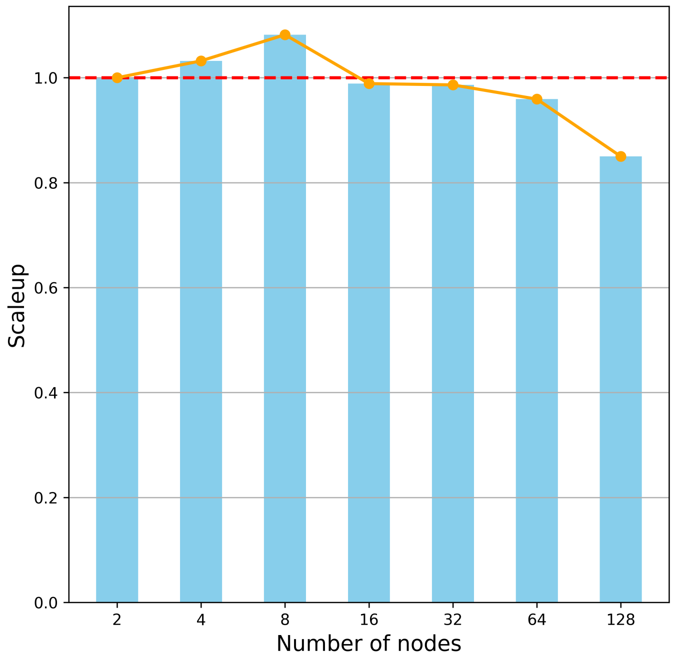

Figure 8. Scaleup experiments.

6. Conclusions

This paper presented an implementation of the parallelization of the Bison Algorithm, applied to classification problems. The Bison Algorithm is implemented using PySpark in order to leverage the distributed computing power and to handle large datasets efficiently.

As well as evaluating the classification measures, such as accuracy, precision, recall, and F1-Score, this study compared the performance of the Bison Algorithm on several dataset sizes using speedup and scaleup as performance measures. As for the results, very good accuracy, precision, recall, and F1-scores were achieved, with average values over the nine datasets being 0.97714764, 0.98741471, 0.97768657, and 0.97771218, respectively. Furthermore, regarding the scalability analysis, very good speedup results were achieved when increasing the the dataset sizes. Ideal speedup values were obtained on all datasets running the algorithm up to 64 processing nodes. The dataset sizes used for the experiments ranged from 0.5 GB to 32.0 GB.

Future research will explore even larger dataset sizes and also explore Apache Spark's other ways of parallelizing the data processing task. In addition, a potential comparison of the different parallelization techniques could be conducted. Moreover, it would be interesting to investigate whether the programming language used for the algorithm implementation factors into the overall performance.

Author Contributions: Conceptualization, S.A.L.; methodology, S.A.L. and J.A.-S.; software, J.A.-S.; validation, S.A.L.; data curation, A.M.M. and J.A.-S.; writing—original draft preparation, S.A.L. and A.M.M.; writing—review and editing, S.A.L. and J.A.-S.; visualization, S.A.L.; supervision, S.A.L.; project administration, S.A.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data that was used for this research project can be downloaded from https://archive.ics.uci.edu/dataset/342/mice+protein+expression (accessed on 30 July 2024).

Acknowledgments: The authors gratefully acknowledge the ACCESS resources used for this research supported by the U.S. National Science Foundation (NSF) under the Office of Advanced Cyberinfrastructure awards 2138259, 2138286, 2138307, 2137603, and 2138296.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Dantzig, G.B. *Maximization of a Linear Function of Variables Subject to Linear Inequalities*; Koopmans, T.C., Ed.; Activity Analysis of Production and Allocation; Wiley and Chapman-Hall: New York, NY, USA; London, UK, 1951; pp. 339–347.
2. Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*; Addison-Wesley: Reading, MA, USA, 1989.
3. Kennedy, J.; Eberhart, R. Particle Swarm Optimization. In Proceedings of the IEEE International Conference on Neural Networks, Perth, Australia, 27 November–1 December 1995.
4. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, 2nd ed.*; University of Michigan Press: Ann Arbor, MI, USA ; MIT Press: Cambridge, MA, USA, 1992.
5. Dorigo, M.; Stützle, T. *Ant Colony Optimization*; MIT Press: Cambridge, MA, USA, 2004.
6. Kazikova, A.; Pluhacek, M.; Senkerik, R. Performance of the Bison Algorithm on Benchmark IEEE CEC 2017. In *Artificial Intelligence and Algorithms in Intelligent Systems, CSOC2018*; Silhavy, R., Ed.; Advances in Intelligent Systems and Computing; Springer: Cham, Switzerland, 2019; Volume 764. [CrossRef]
7. Doe, J. Parallel Genetic Algorithms Using MPI. *J. Comput. Optim.* **2015**, *12*, 233–245.
8. Smith, A.; Lee, B.; Wang, C. MPI-Based Particle Swarm Optimization for High-Dimensional Problems. *IEEE Trans. Evol. Comput.* **2017**, *21*, 157–169.
9. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* **2004**, *51*, 107–113. [CrossRef]
10. Ludwig, S.A.; Aljarah, I. MapReduce Intrusion Detection System Based on a Particle Swarm Optimization Clustering Algorithm. In Proceedings of the IEEE Congress on Evolutionary Computation, Cancun, Mexico, 20–23 June 2013.
11. Miryala, G.; Ludwig, S.A. Comparing Spark with MapReduce: Glowworm Swarm Optimization Applied to Multimodal Functions. *Int. J. Swarm Intell. Res.* **2018**, *9*, 1–22. [CrossRef]
12. Ludwig, S.A. MapReduce-based Fuzzy C-Means Clustering Algorithm: Implementation and Scalability. *Int. J. Mach. Learn. Cybern.* **2015**, *6*, 923–934. [CrossRef]
13. Al-Sawwa, J.; Almseidin, M. A Spark-Based Artificial Bee Colony Algorithm for Unbalanced Large Data Classification. *Information* **2022**, *13*, 530. [CrossRef]
14. Al-Sawwa, J.; Ludwig, S.A. Performance Evaluation of a Cost-Sensitive Differential Evolution Classifier using Spark—Imbalanced Binary Classification. *J. Comput. Sci.* **2020**, *40*, 101065. [CrossRef]
15. Al-Sawwa, J.; Ludwig, S.A. Parallel Particle Swarm Optimization Classification Algorithm Variant implemented with Apache Spark. *J. Concurr. Comput. Pract. Exp.* **2020**, *32*, e5451. [CrossRef]

16. Tan, Y. Survey on GPU-based Swarm Intelligence Algorithms. *Swarm Evol. Comput.* **2013**, *7*, 34–47. [CrossRef] [PubMed]
17. Kroemer, O.; Niekum, S.; Konidaris, G. GPU-based Parallelization of Particle Swarm Optimization. *J. Comput. Sci.* **2015**, *16*, 201–211.
18. Lalwani, L.; Roy, R.; Dehuri, S. Parallel Swarm Optimization Algorithms: A Survey. *Int. J. Swarm Intell. Res.* **2020**, *11*, 1–25. [CrossRef]
19. Salloum, S.; Dautov, R.; Chen, X.; Peng, P.X.; Huang, J.Z. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* **2016**, *1*, 145–164. [CrossRef]
20. Karau, H.; Warren, R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017. Available online: https://books.google.cl/books?hl=en&lr=&id=90glDwAAQBAJ&oi=fnd&pg=PP1&dq=apache+spark+&ots=FB4RNXixd&sig=U1cmaDYpXa6l_dEa44fW0pgGiI8&redir_esc=y#v=onepage&q=apache%20spark&f=false (accessed on 24 July 2024).
21. Higuera, C.; Gardiner, K.; Cios, K. Mice Protein Expression Dataset. UCI Machine Learning Repository. 2015. Available online: https://archive.ics.uci.edu/dataset/342/mice+protein+expression (accessed on 31 October 2024). [CrossRef]