Parallel Feature Selection for Distributed-Memory Clusters

Jorge González-Domínguez[a,*], Verónica Bolón-Canedo[b], Borja Freire[a], Juan Touriño[a]

[a]*Computer Architecture Group, Universidade da Coruña, Campus de Elviña, 15071 A Coruña, Spain*
[b]*Department of Computer Science, Universidade da Coruña, Campus de Elviña, 15071 A Coruña, Spain*

Abstract

Feature selection is nowadays an extremely important data mining stage in the field of machine learning due to the appearance of problems of high dimensionality. In the literature there are numerous feature selection methods, mRMR (minimum-Redundancy-Maximum-Relevance) being one of the most widely used. However, although it achieves good results in selecting relevant features, it is impractical for datasets with thousands of features. A possible solution to this limitation is the use of the fast-mRMR method, a greedy optimization of the mRMR algorithm that improves both scalability and efficiency. In this work we present fast-mRMR-MPI, a novel hybrid parallel implementation that uses MPI and OpenMP to accelerate feature selection on distributed-memory clusters. Our performance evaluation on two different systems using five representative input datasets shows that fast-mRMR-MPI is significantly faster than fast-mRMR while providing the same results. As an example, our tool needs less than one minute to select 200 features of a dataset with more than four million features and 16,000 samples on a cluster with 32 nodes (768 cores in total), while the sequential fast-mRMR required more than eight hours. Moreover, fast-mRMR-MPI distributes data so that it is able to exploit the memory available on different nodes of a cluster and then complete analyses that fail on a single node due to memory constraints. Our tool is publicly available at https://github.com/borjaf696/Fast-mRMR.

Keywords: Machine Learning, Feature Selection, High Performance Computing, Parallel Computing

1. Introduction

Over the last few years we have witnessed an explosion in the size of datasets related to different fields such as biology, physics, mathematics, or engineering, to name just a few. This increasing growth has not only occurred in the number of patterns or samples, but also regarding the number of features. If a feature can be understood as an individual property of a phenomenon which can be measurable and observed, it is reasonable to think that

---

*Corresponding author
Email addresses: jgonzalezd@udc.es (Jorge González-Domínguez), veronica.bolon@udc.es (Verónica Bolón-Canedo), borja.freire1@udc.es (Borja Freire), juan@udc.es (Juan Touriño)

having more features could give us better results. However, this is not the case because of the "curse of dimensionality", a term that was coined by Richard Bellman back in 1957 to describe the difficulty of optimization by exhaustive enumeration on product spaces [2]. This colorful term refers to the different phenomena that appear when analyzing high-dimensional datasets (with thousands of features) that do not occur in low-dimensional settings.

If we assume that datasets are usually represented by matrices in which the rows are the patterns (or samples) and the columns are the features, a possible solution to the afore-mentioned problem is to find "narrower" matrices which do not incur a significant loss of information through data mining. This process is known as dimensionality reduction and one of the most popular representative methods is called Feature Selection (FS). FS can be defined as the process of selecting the relevant features and discarding the irrelevant ones, without incurring a degradation of performance [4, 11]. Moreover, FS has also the advantage of its explainability, since it maintains the meaning of the original features, in contrast to feature extraction methods which perform a transformation of the original features to reduce the dimensionality of the problem. However, with the advent of Big Data, and the appearance of datasets with thousands or millions of features, existing state-of-the-art FS methods have become inapplicable, and it has been necessary to accelerate them.

Acceleration of FS algorithms to be able to explore large datasets within a reasonable time has been addressed in some previous works. On the one hand, the acceleration can come from reducing the complexity of the algorithm itself. One example is the acceleration of the popular method minimum-Redundancy-Maximum-Relevance (mRMR) [20] which, although widely used, suffers from high complexity when dealing with a large number of features. A greedy variation, called fast-mRMR [22], was recently proposed to reduce runtime by assessing the redundancy of the candidate features to only those already selected, instead of the whole dataset. Another greedy optimization consists in the simplification of the score metric calculated for each feature [23]. Approaches such as memetic frameworks [29] or particle swarm optimization [8] have also been applied to reduce the number of features to explore during FS.

On the other hand, efficient exploitation of High Performance Computing (HPC) hardware resources can also be employed to accelerate applications without reducing the overall computational requirements of the algorithms. For instance, the popular WEKA toolkit [13] already includes multithreaded support for some FS algorithms in order to exploit the several cores available in current processors. The adequacy of parallel computing to accelerate some WEKA FS algorithms has been addressed in [6], analyzing the use of multiple threads on a multicore system as well as Spark instances on the cloud. FS on cloud-based systems using MapReduce has been extensively studied, including implementations of ReliefF [28], evolutionary algorithms [21] or positive approximation [14]. Another popular platform is the GPU, with a relatively low price and good energy efficiency. For instance, GPU implementations of FS algorithms exist using a non-parametric evaluation criterion [1] or the delta test [10].

In this work, we present fast-mRMR-MPI, a novel parallelization of fast-mRMR that exploits the hardware resources of modern distributed-memory systems to accelerate FS. It is implemented with a hybrid approach that uses MPI [26] to work on different nodes connected

2

through a network, with multiple OpenMP [19] threads to exploit several cores within the same node. Our goal is to gather in the same implementation the performance advantages of greedy optimizations and parallel computing. This idea is already present in [22], with parallel versions of fast-mRMR for GPUs and cloud systems (based on Spark). However, with the popularization of HPC and Big Data, many scientists have access to powerful computational resources such as distributed-memory clusters. These systems are installed in institutes or supercomputing centers and are available to researchers through remote connection. Thus, scientists can have easy access to several multicore nodes connected through a low-latency and high bandwidth network, and they require applications that scale with a large number of nodes. Although the Spark implementation of fast-mRMR can be used in such clusters, message-passing codes usually obtain better performance on them. Therefore, our proposed fast-mRMR-MPI consists in a novel implementation of fast-mRMR using MPI and OpenMP that significantly increases its speedup on clusters.

The idea of using message-passing to accelerate FS on multicore clusters has already been addressed in previous works. Some examples are DWFS [24], with a parallel genetic algorithm, as well as an MPI version of online FS [27]. Nevertheless, fast-mRMR-MPI provides the following advantages over them:

- The FS filter used as base for our work (mRMR) is extremely popular and has been proved highly accurate.

- The MPI implementation of the two previous works is based on a master-slave paradigm, which usually limits the scalability to only tens of processes. Instead, fast-mRMR-MPI employs a static and completely decentralized parallelization that, as will be shown in Section 4.2, scales up to several hundreds of cores.

- fast-mRMR-MPI integrates multiple threads and processes with a hybrid MPI/OpenMP implementation to efficiently exploit the computational capabilities of modern multi-core clusters. This hybrid approach has shown good scalability in other fields such as bioinformatics [9] or molecular dynamics [5].

- The code is publicly available at https://github.com/borjaf696/Fast-mRMR

The rest of the paper is organized as follows. Section 2 explains the background about the FS algorithm and HPC technologies necessary to understand the implementation. Our parallel implementation is described in Section 3. Section 4 provides the experimental evaluation in terms of runtime and scalability. Finally, concluding remarks and future work are presented in Section 5.

2. Background

In this section we introduce some basic concepts associated to the FS method chosen to be parallelized (fast-mRMR), as well as to the parallelization technologies employed in our implementation (OpenMP and MPI).

2.1. Feature selection

As mentioned in the Introduction, feature selection is a popular preprocessing technique that allows us to reduce the dimensionality of the problem. A correct selection of the features can lead to an improvement of the inductive learner, either in terms of learning speed, generalization capacity or simplicity of the induced model.

FS methods are typically divided into three major approaches according to the relationship between a FS algorithm and the inductive learning method used to infer a model [11]: *filters*, which rely on general characteristics of the data and are independent of the induction algorithm; *wrappers*, which use the prediction provided by a classifier to evaluate subsets of features; and *embedded methods*, which perform FS in the process of training and are specific to given learning machines.

Filters are probably the most popular approach for feature selection, because of their simplicity but good results, the following being well-known representative methods: Correlation-based Feature Selection (CFS), which selects features highly correlated with the class but uncorrelated between them; ReliefF, which is based on the idea of nearest neighbors; Mutual Information Minimization (MIM), which just selects those features with high mutual information with the class; or mRMR, which is the focus of this work. Among embedded methods, Recursive Feature Elimination for Support Vector Machines (SVM-RFE) is a popular method which decides the relevant features according to the weights of an SVM classifier, while LASSO and RIDGE are regularization methods that perform feature selection and classification at the same time. Finally, the most common approach for wrappers is forward or backward selection, in combination with an induction method. It is worth mentioning the popular dimensionality reduction Principal Component Analysis (PCA), which identifies a smaller number of uncorrelated variables known as principal components from a larger set of data. Note, however, that PCA is a feature extraction method, since it transforms the original features into a set of new ones. On the contrary, FS methods select a subset of the original features. The interested reader can find more general information about FS in the specialized literature [4, 12, 18, 25].

In this work we have focused on improving the performance of mRMR, which is a FS method from the family of filters which returns an ordered ranking of all the features. It will be further described in the next subsection.

2.2. minimum Redundancy Maximum Relevance

The mRMR method [20] is a popular filter for FS which in essence estimates the Mutual Information (MI) to detect those features highly relevant to the class but not redundant among them. It has been the focus of much attention by the research community. It was originally developed to deal with DNA microarray data classification although, nowadays, the method has also been applied to many other fields such as anomaly detection [3], human activity recognition [16], or image classification [15].

However, mRMR can present computational problems when dealing with a high number of features because of its complexity $O(n^2 * m)$, where n is the number of features and m is the number of samples. This high complexity is the result of an expensive calculation process. It consists of calculating high and low MI in the form of relevance and redundancy,

respectively. Equation 1 shows how to calculate the MI I between two variables A and B, using the marginal probabilities $p(a)$ and $p(b)$, and the joint probability $p(a, b)$.

$$I(A; B) = \sum_{b}^{B} \sum_{a}^{A} p(a, b) * log(\frac{p(a, b)}{(p(a) * p(b))} \tag{1}$$

Next, Equation 2 shows how to compute the mRMR value for a given feature A, where C is the class and F is the complete set of features.

$$mRMR(A) = I(A; C) - \sum_{f}^{F - (A, C)} I(A, f) \tag{2}$$

In the original implementation of the method, the high complexity comes from the fact that the MI is calculated for each feature and class, and then between all feature pairs, which requires an extremely long runtime when analyzing datasets with a large number of features. In order to alleviate this issue, a modification of the original mRMR called fast-mRMR was recently proposed [22], introducing multiple optimizations over mRMR. The most important one is the explicit separation of the calculation of the high MI (relevance) and low MI (redundancy), as follows:

- The relevance is only calculated once for each feature and saved in memory to be reused. The score that determines the first feature selected is the relevance, while the mRMR value is seen as the score for the next features.

- The redundancy is calculated only for those features which have already been selected, instead of calculating the MI for every feature pair.

fast-mRMR is able to reduce the complexity from $O(n^2 * m)$ to $O(m * n)$, which results in a high improvement in efficiency without affecting the effectiveness of the algorithm to select the adequate features. We refer to [22] for more details about these algorithms. However, the fast-mRMR implementation is still prohibitive when dealing with large datasets.

2.3. Parallel programming models

The target parallel architectures of this work are distributed-memory clusters that consist of several nodes (each of them with several CPU cores and memory modules) interconnected through a network. The basis of parallel computing on this kind of systems is to split the workload into different tasks that are executed on multiple nodes. The most common programming model for these systems is message-passing, where different processes (each one with an associated local memory) are in charge of different tasks.

MPI [26] is a portable, efficient, and flexible standard interface for message-passing. It allows programmers to not be concerned about the hardware architecture or the memory system. All processes are connected and synchronized through a logical abstraction and use messages to communicate among them. Nevertheless, programmers are in charge of data distribution and communication, which means better optimization opportunities at

the expense of hard programming work. We should remark that each process has its own memory address space that cannot be directly accessed by other processes. MPI offers a set of primitives to access remote data. Data communication among processes is one of the main performance overheads in MPI programs, so programmers must try to minimize it by placing on the local memory of each process the data required to complete its tasks.

In a pure MPI program each process is linked to one core, while in a hybrid approach each process is usually mapped to one node and it has several associated threads launched using OpenMP [19] (often the same number of threads as cores within the node). OpenMP is a C/C++/Fortran API for platform-independent shared-memory parallel programming. It provides a high-level abstraction layer over low-level multithreading primitives that are portable to different compilers and hardware architectures, scalable over an arbitrary number of CPU cores and flexible in terms of writing compact and expressive source code.

The basic philosophy of OpenMP is to augment sequential code by using compiler directives (called pragmas) to give hints to the compiler on how the code can be parallelized. When a program starts its execution, only one master thread exists. Parallel sections or parallel loops are defined with the corresponding pragma, and an arbitrary number of software threads are spawned by the master thread. All threads share the resources of the parent system process, i.e., they can access the same memory space. This is advantageous since threads can be spawned with low latency and benefit from lightweight inter-thread communication using shared registers and arrays. However, it can provoke race conditions, i.e., situations where two or more threads access the same data simultaneously. As the thread scheduling algorithm can swap between threads at any time, we do not know the order in which the threads will attempt to access the shared data. Therefore, the result of a data modification would depend on the thread scheduling algorithm. The programmer must ensure that data modifications are performed in the correct order, either with mutexes that serialize shared memory accesses or creating copies of the data for each thread (private data).

3. Parallel implementation

fast-mRMR-MPI is a command line tool that receives as arguments some configuration parameters such as the path to the input file, the number of wanted features or the number of classes. An explanation of all the arguments, as well as installation and execution instructions, are included in the website of the tool. Note that fast-mRMR-MPI was designed as a parallel version of fast-mRMR (it returns the same results) and thus requires the same type of data as input:

- Data must be discrete in order to calculate the mRMR value as shown in Section 2.2.

- The input dataset must follow a binary format that helps to reduce the disk quota requirements on the testbeds.

The website of the tool also includes programs to discretize data and convert it to the binary format in the event that the input data format does not fulfill these conditions.

3.1. MPI parallelization

As mentioned in Section 2.3, an MPI parallelization consists in splitting the computation into several tasks that are assigned to different processes. Every time that a new feature must be selected each process is in charge of calculating the mRMR value for different features. All processes ($numP$) can work simultaneously without interruptions as the mRMR calculation is independent for different features (n). All static distributions assigning the same number of features to every process lead to a well-balanced workload (similar runtime for every process) as the time required to calculate the mRMR value is constant for every feature. Specifically, we have used a pure block distribution where the first $\frac{n}{numP}$ features are assigned to Process 0, next $\frac{n}{numP}$ to Process 1 and so on. Splitting the work by samples was discarded as it would require n reduction operations involving all processes for every time that we select a new feature in order to calculate the mRMR value of all features.

Algorithm 1 describes the MPI parallelization to select the most relevant features. In order to reduce the overhead due to I/O operations each process starts reading from the input file only its assigned features (`ReadMyFeatures` in Line 1), while the information of the classes is read by all processes (`ReadClass` in Line 2). This approach also reduces the memory requirements as each process only stores the data of the associated features (no feature information is replicated in different local memories). As will be shown in an example of the experimental evaluation (Section 4.2), feature distribution among different nodes makes fast-mRMR-MPI able to analyze big datasets that do not fit in the memory of a single desktop computer. An alternative approach where only Process 0 reads the whole information and then distributes it among the processes was also designed, but was discarded because it increased communication and provided poorer performance.

Once the local memory of each process contains the data of the associated features the procedure to select each feature consists in the following steps:

1. Each process calculates the score of its features using the information stored in its local memory. No communications are necessary at this point thanks to the appropriate data distribution. Just like the original fast-mRMR, the score for the first feature is the relevance to the class, while the mRMR value is used as a score for the other `numFeaturesWanted`−1 features (Lines 6 and 24 in Algorithm 1, respectively).

2. Each process calculates the feature that obtained the highest score (`mySelFeature` in Lines 7 and 25), as well as the partial maximum value (`myMaximum` in Lines 8 and 26).

3. The partial maximum scores of all processes and the features that obtained them are collected by Process 0 using the MPI collective `MPI_Gather` (Lines 10 and 11). Concretely, these gathers are grouped into only one call to the MPI routine with two elements per process in order to reduce communication latency.

4. As soon as Process 0 has all the partial maximums in its local memory, it looks for the global maximum and includes it in the list where the selected features are saved (Lines 12-14). If the number of already selected features `numSelectedFeatures` is equal to `numFeaturesWanted` the computation finishes.

5. Process 0 communicates to all processes which has been the last selected feature, using the collective `MPI_Bcast` (Line 17).

Algorithm 1: fast-mRMR-MPI: most computationally expensive steps for every MPI process.

Data: rank, file, numFeaturesWanted

```
/* rank is the id of the process, file the input file with the data,
   and numFeaturesWanted is the number of selected features     */
```

1 ReadMyFeatures(file, myCandidates);

2 ReadClass(file, classInfo);

3 numSelectedFeatures = 0;

4 **for** *every feature f in myCandidates* **do**

5   accumRedundancy[f] = 0;

6   myRelevancesVector[f] = MutualInfo(f, classInfo);

7 mySelFeature = getMaxPosition(myRelevancesVector);

8 myMaximum = myRelevanceVector[mySelFeature];

9 **while** *numSelectedFeatures ≤ numFeaturesWanted* **do**

10   MPI_Gather(allMaximums, myMaximum, 0);

11   MPI_Gather(allSelFeatures, mySelFeature, 0);

12   **if** *rank == 0* **then**

13   lastFeatureSelected = allSelFeatures[getMaxPosition(allMaximums)];

14   selectedFeatures.add(lastFeatureSelected);

15   numSelectedFeatures += 1;

16   **if** *numSelectedFeatures < numFeaturesWanted* **then**

17   MPI_Bcast(lastFeatureSelected, 0);

18   MPI_Bcast(getInfo(lastFeatureSelected), getOwner(lastFeatureSelected));

19   **if** *rank == getOwner(lastFeatureSelected)* **then**

20   myCandidates.remove(myFeatureSelected);

21   **for** *every feature f in myCandidates* **do**

22   accumRedundancy[f] += mutualInfo(f, infoSelFeature);

23   redundancy = accumRedundancy[f] / numSelectedFeatures;

24   mRMR[f] = myRelevancesVector[f] - redundancy;

25   mySelFeature = getMaxPosition(mRMR);

26   myMaximum = mRMR[mySelFeature];

---

6. In order to calculate the next mRMR values for their associated candidates, all processes need the data of the samples related to the last selected feature. However, as previously explained, the data is not replicated, and thus only one process has read from the input file and stored in its local memory the information of the last selected feature. An `MPI_Bcast` of the last feature data from the owner process is thus required before starting to calculate the next mRMR values (Line 18).

7. The owner of the last feature must remove it from the candidates list to avoid checking it again (Line 20) and all processes return to step 1.
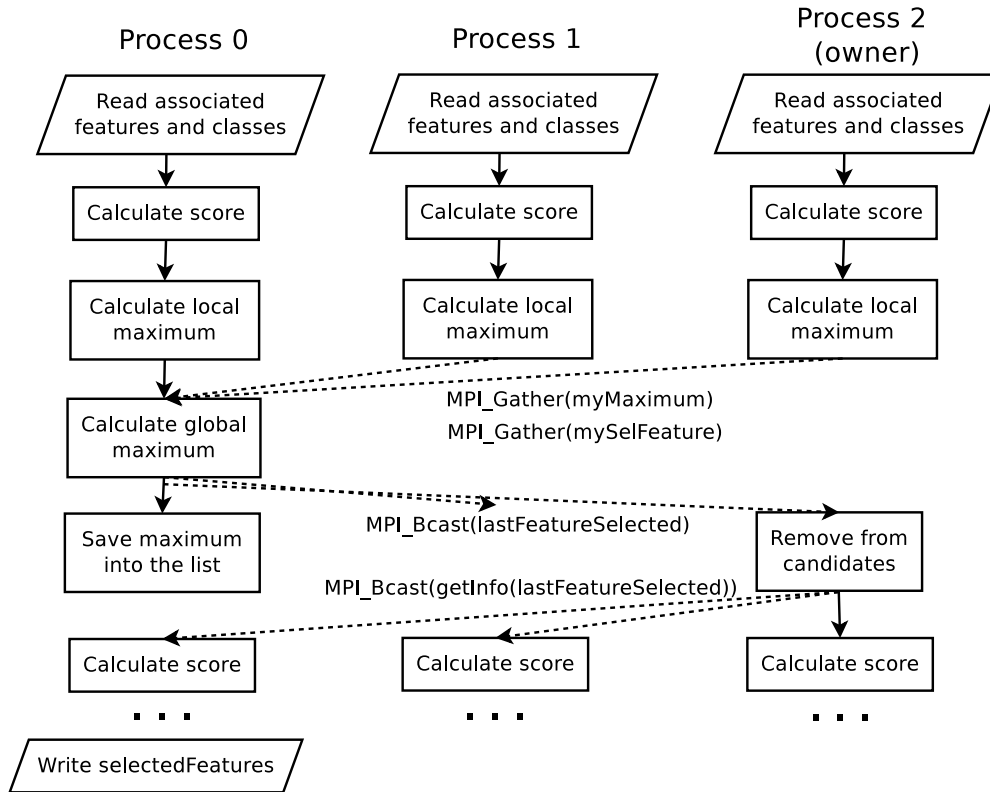
Figure 1: Abstraction of the communications during the selection of each feature for an example with three processes (Process 2 has the features with the highest scores).

Figure 1 illustrates the communications that are present in each iteration for an example with three processes. For simplicity, we assume the unlikely scenario where Process 2 is the owner of all the selected features. At the end of the algorithm Process 0 has the list of features that have been selected (`selectedFeatures`).

3.2. Hybrid MPI/multithreaded parallelization

There is a trend in HPC to develop hybrid MPI and multithreaded programs to achieve an optimal balance between explicit and implicit parallelism in shared and distributed memory architectures. Each process is associated to a group of cores and it launches several threads to map its tasks among the cores of the group. This approach joins advantages of each paradigm such as high speed access to shared memory for the threads which belong to the same process, or the possibility of using multiple nodes to carry out executions with high demand of resources. The best configuration of the number of MPI processes and number of threads per process depends on characteristics of the code such as the number and size of messages, as well as on hardware characteristics such as network latency and bandwidth. Our implementation is flexible enough to allow the users to specify the desired number of processes and threads in order to optimize performance on different systems.

OpenMP is used to generate several threads per process. Concretely, several threads are

Table 1: Dataset description.

Dataset	Features	Samples
BreastCancer	47,293	128
Epsylon	2,000	400,000
ECBDL14	631	32,000,000
News20	1,355,191	19,996
E2006Log1	4,272,227	16,087

launched to collaborate in the for loops where the scores of the candidates are calculated (Lines 4 and 21 in Algorithm 1), using a `pragma parallel for` directive. A dynamic distribution with a block size of 2% of the candidates per thread is used, as this was assessed as the optimal distribution in terms of workload balance.

As mentioned in Section 2.3, one of the main problems usually found in multithreaded codes are the race conditions, where several threads perform concurrent accesses to shared variables. In our fast-mRMR-MPI having a shared variable to indicate the partial maximum of the process would lead to race conditions, as all threads should access and modify it. Instead, each thread stores the scores of its associated features on the proper positions of the vector `mRMR`, and the maximum is calculated by the process after the parallel for loop (Lines 7-8 and 25-26 in Algorithm 1).

4. Experimental evaluation

The experimental evaluation is focused on performance in terms of execution time, as the results of fast-mRMR-MPI were identical to those of the original fast-mRMR, and thus their accuracy has already been proved in [22]. Table 1 summarizes the main characteristics of five representative datasets used for the performance evaluation, which were obtained from the LIBSVM website [17]. They present a different number of features and samples in order to analyze the adequacy of the parallel approach to different scenarios.

Two platforms with different characteristics were used to evaluate the scalability of fast-mRMR-MPI:

- A private cluster of the Universidade da Coruña called Pluton, with eight nodes containing 64GB of memory and two 8-core Intel Xeon E5-2660 Sandy Bridge-EP processors each (i.e., 16 cores per node and 128 in total).

- 32 nodes of the Finis Terrae II supercomputer, installed at the Galician Supercomputing Center (CESGA) [7]. It comprises a total of 768 cores as each node consists of 24 cores (two 12-core Intel Haswell 2680 processors) and 128 GB of memory.

Both clusters have a low-latency and high bandwidth InfiniBand FDR network to connect the nodes. Regarding software, fast-mRMR-MPI was compiled and executed with the open source OpenMPI library (versions 1.8.8 and 1.10.2 for Pluton and Finis Terrae, respectively) and the -O3 option.
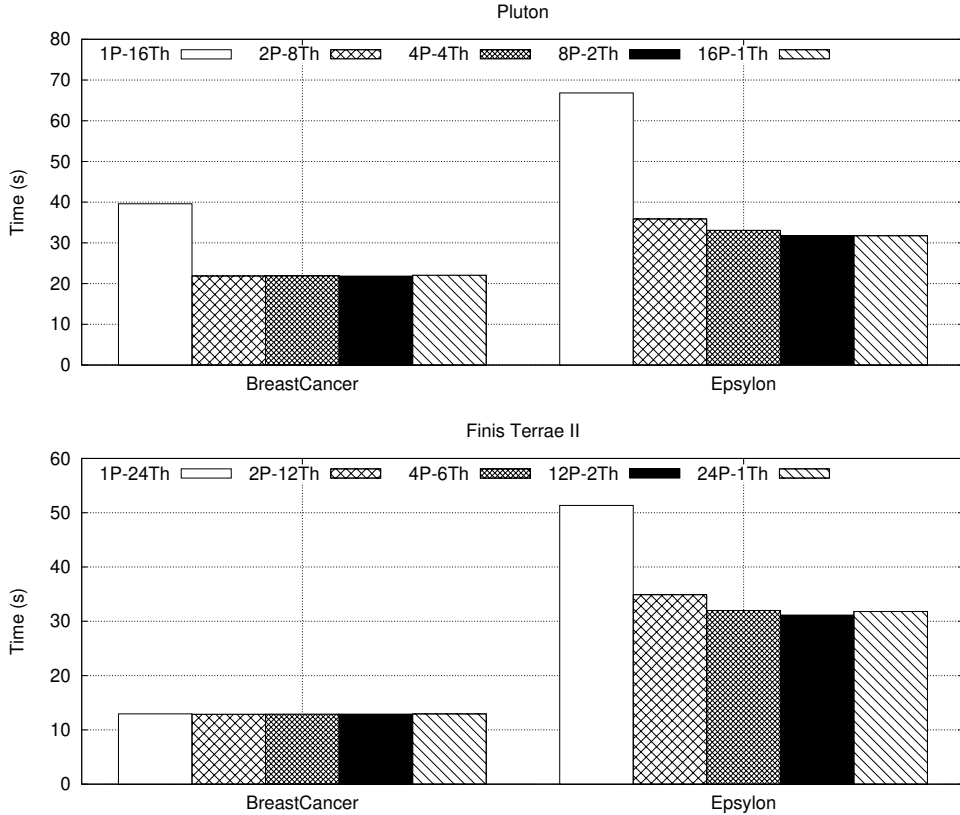
Figure 2: Runtime of fast-mRMR-MPI on one node of each platform when selecting 200 features of `BreastCancer` and `Epsylon` (P=#processes; Th=#threads per process).

Finally, there is a strong relation between the number of features selected and the runtime, as it indicates the number of iterations of the while loop (Line 9 in Algorithm 1). All the experiments in this section were run selecting 200 features of the dataset. This number is high enough to show the impact of discarding the selected features from the candidates list of the owner process, but not too large to avoid selecting an extremely high number of features, which will never be the case in a real scenario. Anyway, the acceleration obtained by our hybrid parallel approach over the original fast-mRMR should be independent of the number of selected features, as the parallelization process is repeated for every iteration of the while loop.

4.1. Best configuration of processes and threads

The experimental evaluation started by finding the best configuration of the number of processes and threads for fast-mRMR-MPI. Figure 2 shows the runtime on a single node of each platform (16 and 24 cores on Pluton and Finis Terrae II, respectively) for different configurations. The smallest datasets (`BreastCancer` and `Epsylon`) were used for this preliminary evaluation.

The worst results are obtained by the configuration with only one MPI process per node

11

that launches one thread per core. As each node has two processors (each one with one memory module), half of the threads are mapped to the cores of a processor different than the host of the MPI process. This leads to some threads accessing the memory module of the other processor, which generates a performance degradation.

When executing with more than one process per node, we use the proper runtime configuration so that all threads are mapped to cores that belong to the same processor as their associated MPI process. Therefore, all memory accesses are performed within the same processor and, as can be seen in Figure 2, the performance of all configurations with more than one process per node is very similar. Note that these conclusions are the same for one dataset with more features than samples (`BreastCancer`) and another one with more samples than features (`Epsylon`). From now on all the results are obtained with the configuration that showed on average slightly better performance on these experiments: eight processes and two threads per node on Pluton, and twelve processes and two threads per node on Finis Terrae II.

4.2. Scalability analysis on several nodes

The speedup S is used as a measure of performance scalability. It is calculated as the acceleration by comparing the sequential time T_s and the parallel time using n cores (T_n): $S(n) = \frac{T_s}{T_n}$. A parallel application is scalable when the speedup increases with the number of cores. The closer $S(n)$ is to n, the better parallel scalability the algorithm presents.

Figure 3 shows the speedups of fast-mRMR-MPI over the original sequential implementation for a different number of nodes. The use of powerful HPC resources is especially relevant for large datasets, so the graphs only show the speedups for the three largest datasets. Results for `E2006Log1` on Pluton are not shown as the original fast-mRMR requires around 70GB of memory, while one node of this cluster only provides 64GB. These graphs prove that the scalability of fast-mRMR-MPI is high (close to the maximum theoretical speedup of the machine), especially for those datasets with a significantly higher number of features than samples (`News20` and `E2006Log1`). For instance, the experiments using `News20` obtain accelerations of 121.91x and 711.52x on the 128 and 768 cores of Pluton and Finis Terrae II, respectively (very high parallel efficiencies of 94.88% and 92.65%). In order to illustrate this trend Figure 4 represents the speedups for all datasets on both platforms (using their largest configuration), according to the number of features in the original dataset. Note that datasets with much more features than samples are nowadays very common in fields such as bioinformatics or genetics. In Section 1, a Spark-based approach of fast-mRMR was mentioned as a possible counterpart of fast-mRMR-MPI on distributed-memory clusters [22]. However, this implementation is focused on other kinds of systems (a cloud infrastructure) and its scalability is very low compared to our tool: according to [22] it achieves accelerations always lower than 5x over fast-mRMR using 228 cores.

As a summary, Table 2 gathers the runtimes to select 200 features of the five datasets on both clusters, and compares them with fast-mRMR runtimes. The performance improvement is significant even using only one node with shared memory (which could be seen equivalent to use a single multicore server). For instance, while fast-mRMR requires more than eight hours to select 200 features of `E2006Log1`, our parallel version reduces it to only
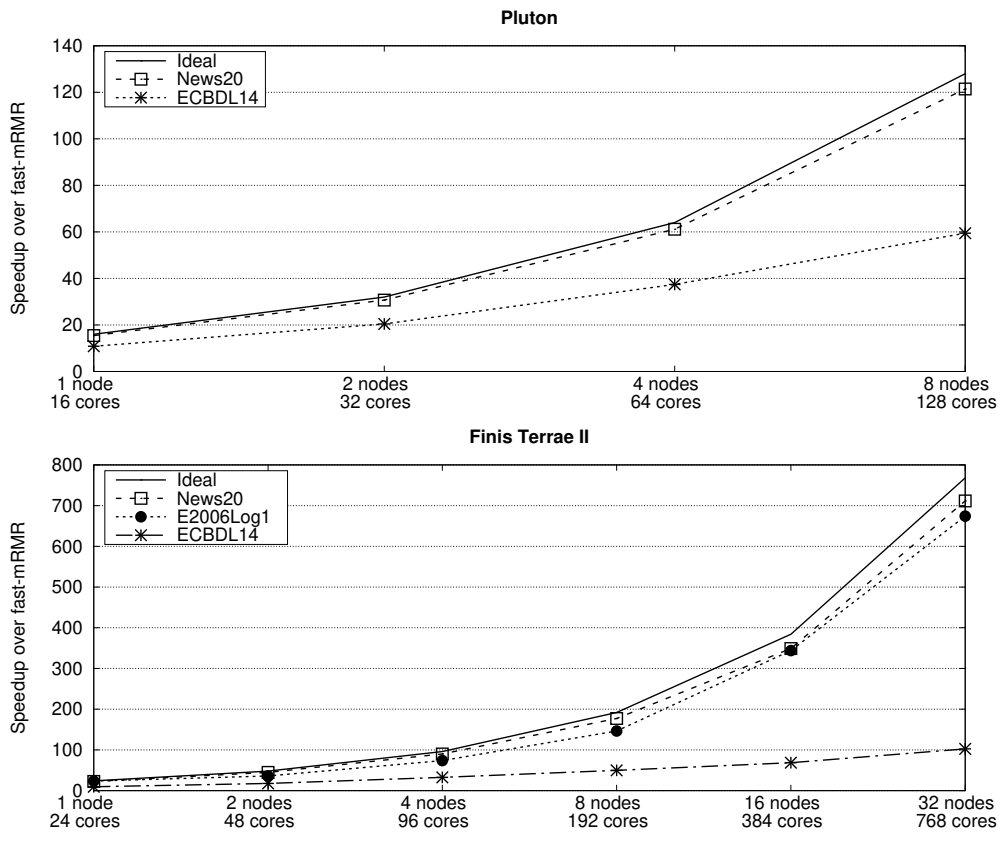
Figure 3: Speedups of fast-mRMR-MPI over fast-mRMR selecting 200 features of the largest datasets.
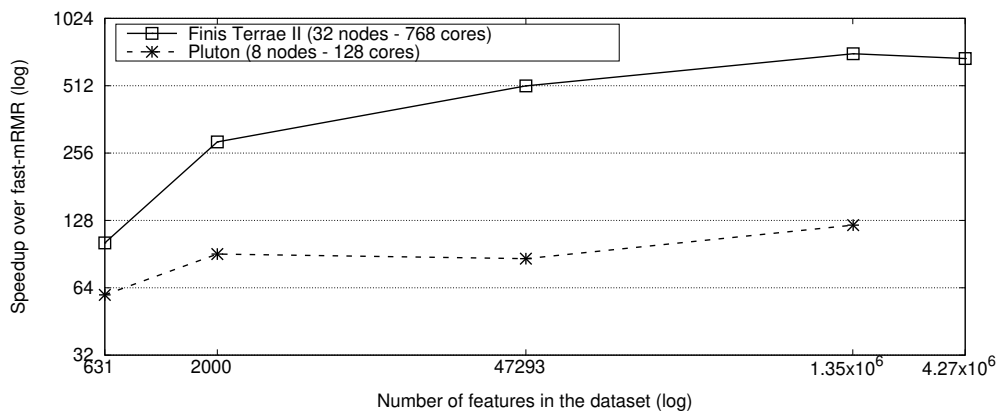

Figure 4: Speedups of fast-mRMR-MPI over fast-mRMR selecting 200 features of the five datasets ordered by their number of features (using the largest configuration of Pluton and Finis Terrae II).

13

Table 2: Runtimes (in minutes) of fast-mRMR-MPI with the best configuration of processes and threads for each platform. The runtimes of the sequential fast-mRMR are also included for comparison purposes. Both algorithms look for 200 features. "−" means that the experiment could not be completed due to memory constraints or not having enough nodes.

Platform	Dataset	fast-mRMR	fast-mRMR-MPI		
			1 node	8 nodes	32 nodes
Pluton	BreastCancer	5.18	0.36	0.06	-
	Epsylon	11.77	1.06	0.13	-
	ECBDL14	89.27	8.20	1.50	-
	News20	224.31	14.53	1.84	-
	E2006Log1	-	-	4.38	-
Finis Terrae II	BreastCancer	5.11	0.33	0.03	0.01
	Epsylon	11.48	0.52	0.10	0.04
	ECBDL14	83.29	8.72	1.67	0.82
	News20	220.57	9.74	1.24	0.31
	E2006Log1	487.47	20.76	3.32	0.72

about 20 minutes on a single node with 24 shared-memory cores. Nevertheless, researchers with access to a cluster will further benefit from the great performance of fast-mRMR-MPI, being able to perform the FS of all the studied datasets in less than one minute on 32 nodes of Finis Terrae II.

Finally, we remark again that the original fast-mRMR algorithm could not be executed on Pluton for the `E2006Log1` dataset, while it took around four minutes for fast-mRMR-MPI using the eight nodes. Therefore, an additional advantage of our parallel version is that it can exploit the memory of several nodes and, with the data distribution explained in Section 3.1, complete the FS of large datasets that cannot be performed on a single node.

5. Conclusions

FS is a data mining technique that is nowadays a common and extremely important step in machine learning, especially with the continuous increase of the average dataset sizes on different fields such as text mining, genetics or bioinformatics. In this paper we have presented fast-mRMR-MPI, a parallel tool designed to accelerate FS by exploiting the computational capabilities of distributed-memory clusters. It is based on fast-mRMR, a greedy optimization of the very popular mRMR algorithm that has proved to be efficient.

Our tool follows a hybrid two-level parallelization approach. First, it includes an MPI implementation that distributes the features among MPI processes so that FS of large datasets can benefit from the memory available on different nodes of a cluster or supercomputer. It is optimized for modern multicore clusters thanks to the second level of parallelism, where each MPI process launches several OpenMP threads sharing memory.

The experimental evaluation on several scenarios (two clusters and five representative input datasets) proved that fast-mRMR-MPI selects the same features as fast-mRMR, but in a significantly lower runtime. Although the magnitude of the acceleration depends on the

characteristics of the dataset (it is more beneficial for those datasets where the number of features is significantly higher than the number of samples), fast-mRMR-MPI performance scales with the number of nodes for all tested datasets. Specifically, it is able to obtain speedups of up to 711.52x using 768 cores.

Our tool is publicly available at https://github.com/borjaf696/Fast-mRMR. Our future work consists in making fast-mRMR-MPI more flexible so that it can accept more formats for the input datasets as well as in developing parallel versions of other feature selection algorithms.

Acknowledgments

References

[1] Azmandian, F., Yilmazer, A., Dy, J.G., Aslam, J.A., Kaeli, D.R., 2014. Harnessing the Power of GPUs to Speed Up Feature Selection for Outlier Detection. Journal of Computer Science and Technology 29, 408–422.

[2] Bellman, R., 1957. Dynamic Programming. Princeton University Press.

[3] Bhuyan, M.H., Bhattacharyya, D.K., Kalita, J.K., 2014. Network Anomaly Detection: Methods, Systems and Tools. IEEE Communications Surveys and Tutorials 16, 303–336.

[4] Bolón-Canedo, V., Sánchez-Maroño, N., Alonso-Betanzos, A., 2015. Feature Selection for High-Dimensional Data. Springer.

[5] Chorley, M.J., Walker, D.W., 2010. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-Core Clusters. Journal of Computational Science 1, 168–174.

[6] Eiras-Franco, C., Bolón-Canedo, V., Ramos, S., González-Domínguez, J., Alonso-Betanzos, A., Touriño, J., 2016. Multithreaded and Spark Parallelization of Feature Selection Filters. Journal of Computational Science 17, 609–619.

[7] Galician Supercomputing Center (CESGA), Last visited: October 2018. https://www.cesga.es/.

[8] Ghamisi, P., Benediktsson, J.A., 2015. Feature Selection Based on Hybridization of Genetic Algorithm and Particle Swarm Optimization. IEEE Geoscience and Remote Sensing Letters 12, 309–313.

[9] González-Domínguez, J., Liu, Y., Touriño, J., Schmidt, B., 2016. MSAProbs-MPI: Parallel Multiple Sequence Aligner for Distributed-Memory Systems. Bioinformatics 32, 3826–3828.

[10] Guillén, A., García-Arenas, M.I., Heeswijk, M., Sovilj, D., Lendasse, A., Herrera, L.J., Pomares, H., Rojas, I., 2014. Fast Feature Selection in a GPU Cluster Using the Delta Test. Entropy 16, 854–869.

[11] Guyon, I., Elisseeff, A., 2003. An Introduction to Variable and Feature Selection. Journal of Machine Learning Research 3, 1157–1182.

[12] Guyon, I., Nikravesh, M., Gunn, S., Zadeh, L.A., 2006. Feature Extraction. Springer.

[13] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA Data Mining Software: an Update. ACM SIGKDD Explorations Newsletter 11, 10–18.

[14] He, Q., Cheng, X., Zhuang, F., Shi, Z., 2014. Parallel Feature Selection Using Positive Approximation Based on MapReduce, in: 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2014), pp. 397–402.

[15] Jia, X., Kuo, B.C., Crawford, M.M., 2013. Feature Mining for Hyperspectral Image Classification. Proceedings of the IEEE 101, 676–697.

[16] Lara, O.D., Labrador, M.A., 2013. A Survey on Human Activity Recognition Using Wearable Sensors. IEEE Communications Surveys and Tutorials 15, 1192–1209.

[17] LIBSVM Data, Last visited: October 2018. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[18] Liu, H., Motoda, H., 2007. Computational Methods of Feature Selection. CRC Press.

[19] OpenMP Architecture Review Board, 2013. OpenMP Application Program Interface Version 4.0. Http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[20] Peng, H., Long, F., Ding, C., 2005. Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min-Redundancy. IEEE Transactions on Pattern Analysis and Machine Intelligence 27, 1226–1238.

[21] Peralta, D., Río, S., Ramírez-Gallego, S., Triguero, I., Benítez, J.M., Herrera, F., 2015. Evolutionary Feature Selection for Big Data Classification: A MapReduce Approach. Mathematical Problems in Engineering , 1–11.

[22] Ramírez-Gallego, S., Lastra, I., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J.M., Herrera, F., Alonso-Betanzos, A., 2017. Fast-mRMR: Fast Minimum Redundancy Maximum Relevance Algorithm for High-Dimensional Big Data. International Journal of Intelligent Systems 32, 134–152.

[23] Raza, M.S., Qamar, U., 2018. A Parallel Rough Set Based Dependency Calculation Method for Efficient Feature Selection. Applied Soft Computing 71, 1020–1034.

[24] Soufan, O., Kleftogiannis, D., Kalnis, P., Bajic, V.B., 2015. DWFS: A Wrapper Feature Selection Tool Based on a Parallel Genetic Algorithm. PLoS ONE 10, 1–23.

[25] Stańczyk, U., Jain, L.C., 2015. Feature Selection for Data and Pattern Recognition. Springer.

[26] The MPI Forum, 2015. MPI: A Message Passing Interface (version 3.1). Http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[27] Yang, H., Fujimaki, R., Kusumura, Y., Liu, J., 2016. Online Feature Selection: A Limited-Memory Substitution Algorithm and its Asynchronous Parallel Variation, in: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), pp. 1945–1954.

[28] Yazidi, J., Bouaguel, W., Essoussi, N., 2016. A Parallel Implementation of Relief Algorithm Using MapReduce Paradigm, in: 8th International Conference on Computational Collective Intelligence (ICCI 2016), pp. 418–425.

[29] Zhu, Z., Ong, Y., Dash, M., 2007. Wrapper–Filter Feature Selection Algorithm Using a Memetic Framework. IEEE Transactions on Systems, Man, and Cybernetics 37, 70–76.