This is a post-peer-review, pre-copyedit version of an article published in Cluster Computing. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10586-021-03261-z

ScalaParBiBit: Scaling the Binary Biclustering in Distributed-Memory Systems

Basilio B. Fraguela · Diego Andrade · Jorge González-Domínguez

Pubished online: 19 March 2021

Abstract Biclustering is a data mining technique that allows us to find groups of rows and columns that are highly correlated in a 2D dataset. Although there exist several software applications to perform biclustering, most of them suffer from a high computational complexity which prevents their use in large datasets. In this work we present *ScalaParBiBit*, a parallel tool to find biclusters on binary data, quite common in many research fields such as text mining, marketing or bioinformatics. *ScalaParBiBit* takes advantage of the special characteristics of these binary datasets, as well as of an efficient parallel implementation and algorithm, to accelerate the biclustering procedure in distributed-memory systems. The experimental evaluation proves that our tool is significantly faster and more scalable that the state-of-the-art tool *ParBiBit* in a cluster with 32 nodes and 768 cores. Our tool together with its reference manual are freely available at https://github.com/fraguela/ScalaParBiBit.

Keywords Biclustering · High Performance Computing · Multicore Clusters · MPI · Master-Slave

1 Introduction

Data mining techniques are extensively used in many scientific fields to discard non interesting data from large input datasets, extract valuable information, and transform it into an understandable structure. In these fields data is usually stored in 2D matrices where rows and columns represent attributes and individuals, respectively. Moreover, many scientific areas only need to work with binary data, where the possible values are 0 or 1 and only need one bit to be represented. Some examples of research fields that can work with binary data are gene expression analyses [13, 30], where a value equal to one indicates that a gene is differentially expressed in an individual; marketing [9], to represent whether people have access to a certain product or shop; or social networks [10], where those values equal to one indicate relationships among users.

Clustering techniques have been traditionally widely used to extract useful information from large datasets [23, 26]. For instance, clustering is useful in genetic and genomic analyses in order to find groups of genes with influence in the phenotype of all samples. However, clustering techniques fail when trying to find patterns that are only present in a group of individuals. In this common scenario we should use biclustering algorithms, which discriminate not only by rows but also by columns [19, 22].

There exist many alternatives for biclustering, each one with its own advantages and drawbacks, but they have in common a computational complexity significantly higher than that of clustering techniques. Although some approaches are able to reduce the runtime necessary for biclustering thanks to taking into account the binary representation of their data [20, 24], their execution time is still prohibitive for large datasets. In this paper we present *ScalaParBiBit*, a parallel tool to accelerate binary biclustering in multicore clusters that exploits both thread-level and process-level parallelism in order to provide high scalability when increasing the number of nodes. Our software package

Universidade da Coruña, CITIC, Grupo de Arquitectura de Computadores
Facultade de Informática, Campus de Elviña, S/N. 15071. A Coruña, Spain
Phone: +34-881-011-219. Fax: +34-981-167-160
E-mail: {basilio.fraguela, diego.andrade, jorge.gonzalezd}udc.es

is derived from *ParBiBit* [6], a high performance tool also targeting these kinds of clusters, but offering much lower performance and scalability while having much greater memory limitations, as our experiments show. As we will see, the development of *ScalaParBiBit* not only involved typical sequential and parallel optimizations, but also a strong redesign of the algorithm. This way, the key contributions and observations from this paper are:

- A sequential optimization of the biclustering algorithm that increases its performance by 78.6% in single core executions.
- A novel strategy for the storage of the initialized biclusters that reduces by one half the memory requirements of the algorithm, thus enabling the processing of much larger datasets.
- A novel full redesign of the algorithm that allows overlapping its most expensive stages, so that it is on average 5.93 times faster than the original design in executions with 764 cores.
- We identify the best values for most critical parameters in terms of performance within our new design (the number of processes per node and the size of the chunks of initialized biclusters), so that users can apply them directly.
- The code is made publicly available under an open source license so that all the scientific community can benefit from it.

Altogether, these contributions turn our proposal into not only the fastest cluster-based alternative for biclustering but also the one with the capacity to process the largest datasets, thus resulting in a strong advancement over the state of the art.

The remainder of the paper is organized as follows. The related bibliography is discussed in Sect. 2, which is followed by a description of the *ParBiBit* tool on which our package is inspired in Sect. 3. The development of *ScalaParBiBit* went through several stages of optimization that gave place to several versions, all of which are described in Sect. 4. Our developments are then evaluated together with the original *ParBiBit* implementation in Sect. 5. Finally, Sect. 6 is devoted to our conclusions and future work.

2 Related work

Data mining has become a target field for acceleration through High Performance Computing (HPC) [3, 29]. Among the different data mining algorithms, there has been a special focus on biclustering methods due to their high computational cost and wide adoption. Therefore, there already exist parallel versions of biclustering algorithms, and some of them can be executed on the same hardware as *ScalaParBiBit* (multicore clusters). For instance, *P-bicluster* [28] presents an MPI implementation of a biclustering method for gene expression data based on the anti-monotones property of these datasets. Another MPI-based work, but using the barycenter heuristic, was described in [1]. Other alternatives are implemented with MapReduce approaches. Although they can be executed on multicore clusters, they are more focused on distributed hardware such as grid or cloud systems. Some examples are the biclustering with topological map organization implemented with Spark and presented in [25], the Parallel Large Average Submatrix (*PLAS*) biclustering method developed with Hadoop [14], the Evolutionary Biclustering Approach using MapReduce (*EBA-MR*) [21], or a Spark-based version of the Large Sum Submatrix (*LSS*) biclustering algorithm [15].

As mentioned in Sect. 1, many researchers and scientists work with binary data. Knowing this information in advance can help to accelerate and make more accurate the biclustering of these datasets [2]. Moreover, a recent experimental evaluation of several biclustering tools has shown that binary-based approaches can also be useful for quantitative data if a binary discretization has been previously applied [18]. However, all the previously mentioned works are focused on quantitative datasets and they are not able to take advantage of the binary representation of the information. Up to our knowledge, *ParBiBit* [6] is the only MPI-based biclustering tool specifically designed for binary data and publicly available to download. It is a parallel version of *BiBit* [24], a tool that obtains accurate results for gene expression data, especially on cases with many large biclusters [18]. *ParBibit* provides the same accurate results as the original tool but with significantly lower time. However, it presents two drawbacks that prevent its use on large infrastructures. For one, the memory requirements are high and it runs out of memory in many modern clusters when working on large datasets. Second, the scalability is quite low when increasing the number of nodes, as there are important parts of the code that can only be executed in one node. *ScalaParBiBit* overcomes these problems using a novel hybrid MPI/multithreading master-slave approach that overlaps sequential and heavily-parallel parts of the code.

Additionally, there are other parallel tools for binary biclustering but focused on cloud systems [17] (not publicly available) or GPUs [7, 16].

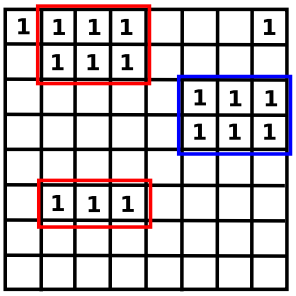


Fig. 1 Example of a binary matrix with two biclusters.

3 Background: the ParBiBit approach

As previously mentioned, *ScalaParBiBit* is an extensive redesign and rewrite of *ParBiBit* that provides higher performance and scalability in multicore clusters thanks to a modification of its parallel structure and implementation. In this section we provide details about the algorithm and implementation of the previous tool, as they are necessary to understand these modifications.

ParBiBit is a command line tool that receives as arguments some configuration parameters. The most important ones are the path to the input and output files, the minimum number of rows (mnr) and columns (mnc) required for the biclusters, and the number of processes and threads. The main component of the input file is the matrix on which the biclustering is performed. Without loss of generality, we will refer in this paper to the rows of the input matrix as attributes or characteristics, while we will consider the columns to be the samples or individuals. *ParBiBit* identifies as biclusters the subsets of rows (attributes) that show a common pattern with respect to a subset of the columns (samples) of the input matrix. This way, since the application operates on binary data, if we consider a binary matrix M of size $m \times n$, a bicluster is a set of rows R and columns C such that $\forall i \in R, \forall j \in C, M_{ij} = 1$. Figure 1 shows an example of binary matrix with two biclusters. The program relies on the concept of bit pattern to identify the biclusters of a binary matrix that have a minimum number of rows mnr and columns mnc specified by the user. The common pattern p to a subset of rows $r_1, \ldots, r_n$ is given by the binary boolean operation AND on them, that is, $p = r_1 \wedge \ldots \wedge r_n$, where $\wedge$ represents the AND operation. The pattern of a bicluster is the common pattern to all the rows that constitute it.

The execution of *ParBiBit* is divided into the following five phases illustrated in Fig. 2:

1. **Input reading**. All processes read the 2D input expression matrix from a file and discretize the data in case that it is not binary (in this case the user must specify the number of bins for discretization). Each process stores a copy of the binary matrix, as all of them need the whole dataset to complete the biclusters (Step 4). Nevertheless, this memory overhead is almost negligible.
2. **Data compression**. Instead of storing in memory a matrix with one integer per element during the whole execution, every process compresses this information using only one bit per element. Every integer (32 bits) stores 32 elements. This compression allows saving memory and to accelerate later calculations, as every integer operation works in parallel over 32 elements.
3. **Bicluster initialization**. This step evaluates all possible gene-pairs and stores in a C++ set structure, namely a `std::set` object of the standard library, the initial information of a bicluster only for those pairs whose genes have at least mnc bits equal

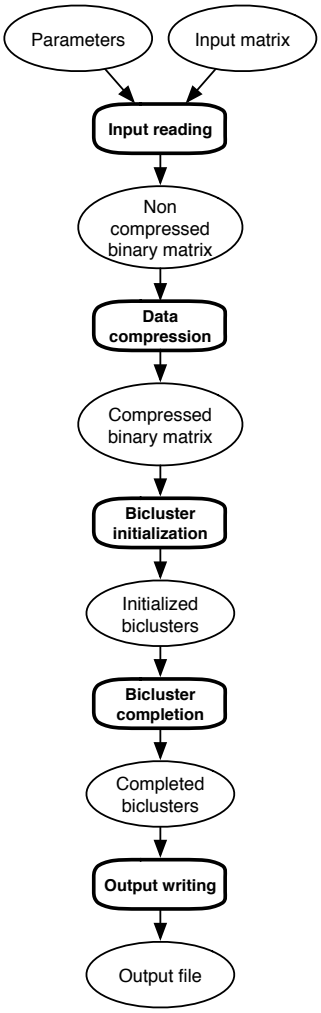


Fig. 2 ParBiBit algorithm. Stages are shown as rounded boxes, while datasets are represented as ellipses.

Table 1 First patterns created for the matrix of Fig. 1.

Pair id	r_x	r_y	Pattern	N. of ones
0	0	1	01110000	3
1	0	2	00000001	1
2	0	3	00000001	1
3	0	4	00000000	0
4	0	5	01110000	3
5	0	6	00000000	0
6	0	7	00000000	0
7	1	2	00000000	0
8	1	3	00000000	0
9	1	4	00000000	0
10	1	5	01110000	3
11	1	6	00000000	0
12	1	7	00000000	0
13	2	3	00000111	3
14	2	4	00000000	0
15	2	5	00000000	0
16	2	6	00000000	0
17	2	7	00000000	0
...	...	...	...	...
27	6	7	00000000	0

to one. For this purpose the application considers every possible different pair of rows r_x y r_y, and

(a) It creates a bicluster with the pattern $p = r_x \wedge r_y$.
(b) If the number of ones in the pattern p is equal to or greater than *mnc*, and the pattern has not yet been used by another bicluster, then it is inserted into the set of initial biclusters. Otherwise it is discarded.

For instance, when applying this step to an eight-row matrix as the one in Fig. 1, *ParBiBit* creates 28 patterns (one for each possible pair of rows). Table 1 shows the $\wedge$ patterns of some pairs.

The tool searches by default for biclusters with at least two rows and two columns. As the number of ones in the pattern represents the number of columns shared by the rows, only four pairs in our example fulfill that condition (pairs with ids 0, 4, 10 and 13). Moreover, pairs 4 and 10 are discarded as the pattern is exactly the same as that of pair 0. Consequently, two biclusters are initialized in this example: those with ids 0 and 13, which represent pairs of rows (0,1) and (2,3), respectively.

As the information of all the initial biclusters must be stored in a central C++ set in order to efficiently verify the condition of non-repetition of bicluster patterns, only one process works in this step. This process can however launch several threads and distribute the evaluation of the different gene pairs among them, carefully coordinating their operations on the shared set using a mutex, as it is not a thread safe data structure.

4. **Bicluster completion**. For all the biclusters initialized in the previous step, *ParBiBit* has to find which other matrix rows besides the pair (r_x, r_y) with which it was discovered may belong to the bicluster. For this, the application has to check every row r_z different from r_x and r_y, comparing it with the pattern p. Those for which $p \wedge r_z = p$ are included in the bicluster. If after this process the bicluster ends up with fewer than *mnr* rows, it is discarded. Otherwise it is stored as part of the result.

   Let us use again the example of Fig. 1, where two biclusters had been initialized. Since the pattern 01110000 had been identified when considering the pair of rows (0,1), all rows different to 0 and 1 must be compared to this pattern in this step. The outcome of the completion step for this pattern is that row 5 is included in the bicluster, as it is the only one in Fig. 1 that fulfills the condition. In the case of the second bicluster, no other row is included. Therefore, *ParBiBit* provides the two biclusters that are illustrated in Fig. 1.

   This was originally the most computationally demanding step, and thus *ParBiBit* applies a hybrid parallelization to accelerate it. Namely, once the bicluster initialization stage finished in the master, the initial biclusters were distributed among all the processes using MPI 3 remote memory access facilities, as they have been shown to provide better performance than traditional two-sided MPI communications [8]. In order to try to maximize the load balancing, the biclusters were distributed as evenly as possible among the processes. Having received the initial biclusters, each process then created threads in order to process them in parallel.
5. **Output writing**. Once all processes have finished the completion of their assigned biclusters, they send the information to Process 0, which writes it into the output file.

It deserves to be mentioned that the multithreading mechanism used in stages 3 and 4 of the application relies on the native C++11 threading facilities [27] in order to maximize portability and performance.

As mentioned in the previous section, the scalability of *ParBiBit* is not high, especially because the third step (bicluster initialization) is only parallelized in a single process with C++11 threads. In sequential computation, most of the runtime is spent in the fourth step (bicluster completion). However, when increasing the number of nodes, the runtime of this step is drastically reduced, and the bottleneck moves to the bicluster initialization step. Furthermore, the thread-based parallelization of this third step in *ParBiBit* is not able to fully exploit the resources of a one node because it requires many thread synchronizations to add biclusters

to the C++ set, which is a non thread-safe structure. More information about the *ParBiBit* approach and its bottlenecks is available in [6].

4 ScalaParbibit implementation

While the performance of *ParBiBit* was much better than that of its predecessor *BiBit*, we suspected that it was possible to improve not only its absolute performance, but also its scalability. Another objective of our work was to reduce its memory requirements in order to enable the processing of larger datasets. This second goal is as important as the first one given that main memory is often becoming the performance bottleneck of current systems because of the large growth of the problem sizes compared to the capacity of main memory [12].

For this, we had to choose the right programming tools for the development of the application and then we had to focus on stages 3 and 4, as they are the ones that consume the overwhelming majority of the runtime of the application, which is in fact the reason why they were the steps parallelized in [6]. From the results in that paper it was also clear that while stage 4 enjoyed a scalable parallelization able to exploit all the resources in a multicore cluster, this was not the case for stage 3, which despite consuming a relevant portion of the runtime, could only be performed by a single MPI process. In addition, our own analysis of the code revealed that there were several optimization opportunities missing that could be exploited. This way, after selecting the best components, we worked in two stages. First, we applied optimizations at the process level, in which the structure of the algorithm and the parallelization strategy remains unchanged. In the second stage we introduced changes to the parallelization strategy, which in addition to largely changing the structure of the algorithm, led themselves to new additional optimizations. We now justify the components used in the development of our application and then describe the work developed in the two optimization stages.

4.1 Choosing the right tools

A first element to decide when developing an application is the programming language to apply. Since we were seeking high performance, low level control and portability, interpreted languages and languages that run on virtual machines were less attractive than traditional compiled languages. Among the latter, C and C++ were the obvious choices because of their capability to manipulate the machine at low level and the large set of parallel programming tools available for them. Among them, C++ was chosen because (a) it provides much higher level expressivity than C at a negligible cost and (b) the C++ language includes direct support for multithreading since the C++11 standard.

The second decision to take regarded the mechanism used to exploit shared-memory parallelism within each process on top of multithreading. The two most popular approaches in this field in C++ are by far OpenMP and the direct management of threads on top of the facilities provided by the language mentioned before. This latter approach was chosen for two reasons. First, the manual manipulation and synchronization of the threads ensures the minimum possible overheads and total control of the execution compared to the higher level approach represented by OpenMP. Second, relying on OpenMP would imply requiring users to install a compiler that supports this standard, and while OpenMP is widely adopted, as observed in [5], there are important toolsets that do not implement it. As a result, the manual manipulation of threads based on the language facilities improved both the portability and the potential performance to attain.

The last decision involved choosing the mechanism for launching and communicating the processes in a distributed memory environment. The standard in this field for languages such as C and C++ has been MPI for many years, the result being that it enjoys both large portability and highly optimized implementations for every platform. This way, user surveys such as [11] regularly show that MPI is clearly the most widely used approach in HPC for this purpose. As a result, MPI was the component chosen for this purpose.

Let us finally remark that given these decisions, the components chosen for our implementation were also the ones used in *ParBiBit*. This has the additional benefit that it ensures that the performance and memory usage gains observed in *ScalaParBiBit* are only the result of our work, and not due to using potentially better software tools. This makes the straight comparison between both approaches fair.

4.2 Optimizations that do not alter the top level structure of the algorithm

In this stage we can distinguish three substages, one centered on purely sequential optimizations and two focused on reducing the contention among the threads that parallelize the biclustering initialization stage. These three substages are now described in turn.

```
1 if(!patternsSet.count(v)) {
2   mutex.lock();
3   patternsSet.insert(std::move(v));
4   m.unlock();
5 }
```

Listing 1 Original insertion.

4.2.1 Sequential optimizations

We found many typical sequential optimizations that could be applied in the code. They can be summarized as follows:

- Several variables that were repetitively created and destroyed within loops were changed so that a single creation and destruction per loop is needed. This also reduces the number of dynamic memory allocations and deallocations.
- Functions whose invocations showed non negligible overheads were changed to enable their inlining.
- An integer protected by a mutex used in the bicluster completion stage was replaced by an atomic integer.
- The insertion of new unique patterns in the set used in the initial bicluster creation stage was changed to use C++11 move semantics rather than copy semantics. This reduces the insertion cost, as rather than making a deep copy of the object and then destroying the original object, now the application moves the pointers that hold the data of the original object, which becomes empty.
- The insertion of completed biclusters from several parallel threads was originally performed on a shared vector protected by a mutex. Now each thread stores the completed biclusters in a private vector, which thus requires no synchronizations. When a thread completes its computations, it synchronizes on a mutex only once in order to add its completed biclusters to the shared vector.

4.2.2 Reducing the contention on the shared `std::set`

In the second substage we turned our attention to the main source of contention among the threads during the bicluster initialization. The reason for focusing on this issue is that, since contrary to the bicluster completion, bicluster initialization is performed in a single process and its performance is critical for the scalability of the application. The contention happens in the insertions of new bicluster patterns in the shared `std::set`, which must be protected by a mutex to avoid corrupting this non thread safe data structure. When *ParBiBit* generates a new bicluster, it first checks whether it has at least *mnc* ones, as otherwise it is discarded. Nevertheless, if this condition holds, the thread proceeds to search for the bicluster pattern in the shared pattern set in order to check whether the pattern has already been found, in which case the bicluster is also discarded. Only if the pattern does not appear in the set does the thread synchronize on the mutex that protects the set before inserting the pattern. Of course the insertion makes sure there are no replicated patterns in the set, which is automatically provided by the semantics of the C++ `std::set` class. Listing 1 shows a simplified version of this latter part of the process where `v` is the pattern and `patternsSet` is the shared set of patterns. Here member function `count` returns the number of elements in the set with the given value. Since a `std::set` does not hold replicated values, it can only return either 0, if not present, or 1 if present. Notice how the insertion in line 3 uses the move semantics optimization commented in the first substage.

We realized that in addition to the `insert(value)` member function that just inserts a new value in a `std::set` if it does not exist, this C++ class also provides a member function `insert(hint, value)`, where the `hint` is an iterator (basically a pointer inside the set object, in C++ terminology) that is used as hint for the place where the new element is to be inserted. Prior to C++11, for the hint to be effective it had to point to the element that would precede the inserted element. Nevertheless, since C++11, which is the standard we are targeting, this member function optimizes the insertion time if the hint iterator points to the element that would follow the inserted element, or to the end of the container, if the new element would be the last one. As a result, in order to minimize the insertion time, and thus the contention among the threads, we changed the original `count` invocation used to decide whether an insertion should be attempted by a `lower_bound` invocation. This kind of search returns an iterator pointing to the first element that is not less than the one searched. This way, it can point either to an identical element or to the immediately following one in the order of the set, which is appropriate as hint for `insert(hint, value)` since C++11. While this can speedup the insertion, the usage of this search also has a disadvantage. Namely, in order to verify whether the considered pattern exists in the set or not, we not only have to test whether the search was unsuccessful, which happens when the returned iterator points to the end of the set, but also whether the element obtained in a successful search is identical to the one provided for the search. This situation is reflected in the code in Listing 2, which illustrates the application of this optimization.

```
1 auto its = patternsSet.lower_bound(v);
2 if( ( its == patternsSet.end() ) || ( *its != v ) ) {
3   mutex.lock();
4   patternsSet.insert(its, std::move(v));
5   m.unlock();
6 }
```

Listing 2 Improved insertion using a hint.

4.2.3 Replacing `std::set` *by a highly optimized fast set*

While, as we will see in Sect. 5, the results of the optimization just discussed were positive, this part of the application continued to limit the maximum performance for two reasons. The most obvious one is the huge restriction that only one thread at a time can modify the set, which also means that all other threads must wait on the mutex before being able to try to insert a new value. The second cause is that while C++ standard allocators are highly optimized for sequential processing, these mechanisms are inefficient in multi-threaded environments [4]. The main reasons for this are that allocators suffer from heavy internal locking and that the cache lines that hold the main elements of their structure often have to be transferred from cache to cache in order to keep them in a coherent state for all the cores involved.

Another limitation is that both the standard data structures and the standard allocator mechanisms of C++ must support a wide variety of operations, which forces their implementation to be necessarily generic. For example, the fact that *ParBiBit* only needs to search or insert elements in the shared set but never erase any item can be exploited to implement a simpler more efficient data structure and also a simple associated allocator. If in addition these elements are written in a careful way based on C++11 atomic operations rather than mutexes and locks, it will be possible to reduce the waiting times associated to these synchronization primitives and thus maximize concurrency. Therefore, as final and third optimization substage before modifying the high level structure of the application, we wrote our own set class and an associated allocator. Both are written in a generic way, so that they can be used for different kinds of datatypes, their only noticeable limitation is that they do not support the removal of elements, in the case of the set, or their deallocation, in the case of the allocator.

Our optimized set internal implementation is based on a hash table rather than on an ordered tree, which is the structure used by `std::set`. This allows to have more control on the number of comparisons used in the set searches, as the larger the hash table, the fewer the potential comparisons. Indeed, while the complexity of insertions and searches in an `std::set`, which is usually implemented as a red-black tree, is $O(log(n))$, the average complexity in a hash table with enough entries and a reasonable hash function is just $O(1)$. Each bucket of our table contains a list that links all the elements that collide in that bucket. This results in a sequential search on collisions in the same bucket. Also, our class is thread-safe by construction, so that the user does not need to worry about concurrent operations on it. As suggested above, our class achieves its thread-safeness exclusively relying on atomic operations, which strongly reduces the potential for synchronization overheads compared to locks and mutexes.

The policy on thread-safeness based on atomic operations was also followed in the design and implementation of our memory allocator, which provides the storage for the set elements. As an additional optimization, in our allocator each thread obtains chunks of memory from the system periodically and then stores the new entries it creates in those locally owned chunks. This has three advantages. First, it avoids requiring synchronization with other threads in the allocation of each bicluster pattern because the chunk is exclusive to the thread. In fact, the fast atomic-based synchronizations only take place when a thread fills its chunk and it has to allocate a new one, which seldom occurs. Chunk allocation is also extremely efficient, as it is based on direct interaction with the OS trough `mmap`. Second, this strategy also minimizes cache line transfers between cores, as each thread stores contiguously in sequence in its memory chunks the new patterns it generated. The third benefit is that this storage, thanks to being strictly sequential, favors locality and has minimal overheads.

It should be mentioned that while, as discussed above, the consecutive storage of the bicluster patterns within large chunks was key to obtain the large memory savings that will be seen in Sect. 5, there are other two factors that explain it. The first is related to the linking of the elements in the container. In a tree structure such as the one typically used for the implementation of a `std::set` every node needs to store two pointers, so that it can point to its potential left and right children. In our hash table, however, each node only needs a single pointer in order to link the items that collide in the same bucket. The second is that the usage of our own allocator for the objects of our fast set allowed us to replace the `std::vector<uint32_t>` used by *ParBiBit* to represent the patterns stored in the set by just the raw sequences of `uint32_t` elements that encode each pattern. This implies saving the storage associated to three pointers per pattern because a `std::vector`, in

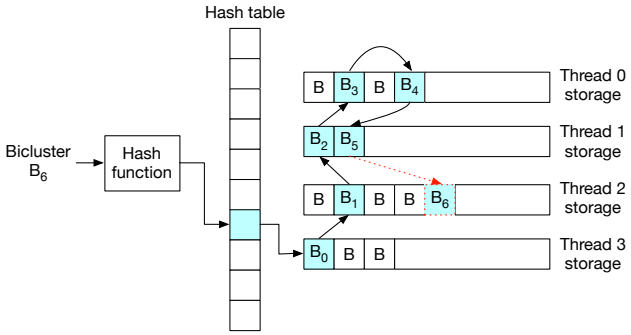


Fig. 3 Fast set implementation and example of usage. A new bicluster pattern B_6 is inserted.

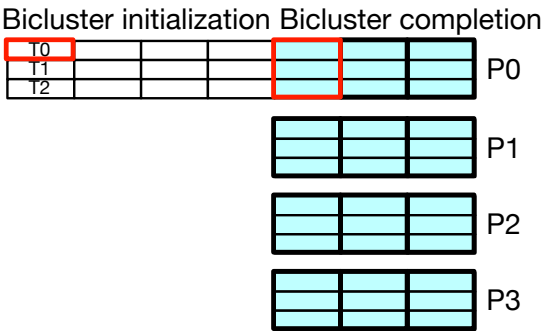


Fig. 4 High level execution of `ParBibit`.

addition to the raw elements stored, needs a pointer to the first one of these elements, another one to the last element stored, and a third one to the end of the region allocated where the data resides. The reason for this latter pointer is that `std::vector` objects allow allocating space for more objects than those actually stored at a given point so that the insertion of new elements in the vector does not necessarily require a memory reallocation. Notice that this never happens in our application because once initialized, the bicluster patterns have a fixed length. Finally, it is worth mentioning that since in the typical 64 bit machines used nowadays each pointer is 8 bytes wide, these last two optimizations imply savings of 32 bytes per pattern.

The structure of our fast set and related allocator and an example of its usage in the insertion of a new bicluster pattern B_6 created by the thread with id 2 out of 4 threads sharing the set are shown in Fig. 3. As first step, the hash function points to a given element in the hash table, which happens to be full, so that there is a collision. This implies that the implementation must follow the linked list conformed by the elements that have collided in that entry. Our example assumes that there are 4 threads in our application, and thus the figure shows on the right the pre-allocated chunk owned by each thread in which it allocates consecutively the entries it pushes into the set. The entries are linked in the order in which the different threads tried to insert them in this same entry of the hash table. If while following the list one of the entries happens to be identical to B_6, it is not inserted. In this example this does not happen, and thus the inserting thread 2 allocates space in its local chunk for the bicluster pattern and it links it at the end of the list followed. This is represented by the dotted link line and box for B_6 in the thread 2 local storage.

4.3 Optimizations to the high level structure of the algorithm

The optimizations discussed in Sect. 4.2 remove many inefficiencies in the application, focusing mainly in the most expensive part of the tool: the bicluster initialization. This makes a lot of sense since in *ParBiBit* this algorithm stage is only performed in a single process, and all the other processes remain idle until its completion. This is illustrated in Fig. 4, which represents an execution with four processes, P0 to P3, and three threads per process, T0 to T2. Each empty box to the left represents the time required by a thread to generate a given amount of initialized biclusters in the master process P0, while each colored box surrounded by a thick line to the right represents the time taken by the 3 threads in a process to complete the biclusters in that chunk. A red line identifies the generation of a first group of initialized biclusters and its later completion. As we can see, this strategy in which initializations and completions cannot be overlapped severely limits the absolute performance and the scalability of the application when more than one process is used in the execution. This is the situation whenever the application uses more than one node in the cluster, either for performance reasons or simply because a single node does not have enough memory to hold the working set of the problem. Therefore, in this second high level stage of the optimization we focus on raising this limitation, which involved changing the structure of the algorithm.

Rather than first initializing all the biclusters, and then scattering them among all the processes in order to complete them, our proposal consists in overlapping both stages. In our proposal, just as in the original *ParBiBit*, a single process, which we call the master, performs the bicluster initialization stage, described as stage 3 in Sect. 3. The difference is that whenever a thread has generated a chunk of biclusters, it tries to send this chunk to another process for its completion,

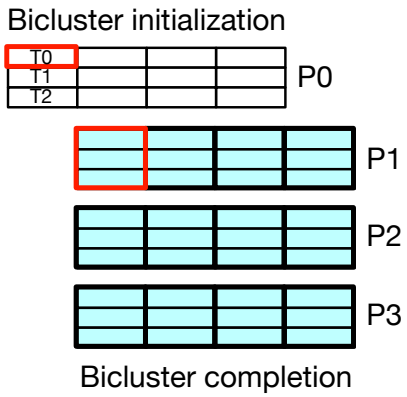


Fig. 5 High level execution of `ScalaParBibit`.

described as stage 4 in Sect. 3, before continuing with the generation of new biclusters. For this, the master keeps the state of each one of the other processes, called slaves. When the master sends a chunk of biclusters for completion to a slave, it marks the slave as busy. Slaves are responsible for notifying the master when they have finished the processing of the assigned chunk so that they can be have new chunks assigned. If the master finds no idle slaves, the simplest solution is that the thread that generated the chunk performs the completion of the biclusters involved before resuming the initialization of new biclusters. Fig. 5 shows the execution of the same problem considered in Fig. 4 using this new strategy, thus illustrating the large potential for performance and scalability improvement that it offers.

This new strategy gives place to interactions between the threading and the MPI components of the application that did not occur in *ParBiBit*. The reason is that it introduces overlaps between these elements, which was not the case in [6]. Indeed, in *ParBiBit* all the bicluster initializations are first performed by multiple threads in the master, and only once they finish the MPI communications are performed by the master thread of each process. The completion of these communications leads to the last stage, parallelized by multithreading within each process. With our proposal, however, any thread in the master can create a chunk of biclusters in any moment and then require to communicate with a slave in order to assign to it the completion of the chunk. And this will happen while the other threads of the master work on their chunks, which can give place to other chunk creations, thus also requiring the usage of MPI.

This new situation implies that *ScalaParBiBit* must initialize MPI in its processes with `MPI_Init_thread` in order to require a certain level of thread support from this library. The easiest alternative would have been to require a `MPI_THREAD_MULTIPLE` level of support, which allows multiple threads to invoke MPI in parallel. However this is the most demanding level of support for MPI and, as observed in [5], important MPI distributions are not compiled by default to support this level, while others do not even support it, at least in all the environments. Therefore we chose to require a `MPI_THREAD_SERIALIZED` multithreading level, in which multiple threads may make MPI calls, but only one thread can make it at a time, so that there can be no concurrent invocations from different threads. This latter condition is satisfied by encapsulating the MPI invocations that can be performed simultaneously by different threads in critical sections implemented by means of C++11 mutexes.

Two further optimizations have been explored for this new implementation, achieving positive results as we will see:

- Rather than forcing the master to compute the chunks whenever there are no idle slaves, the master could keep a buffer of unprocessed chunks and only complete the chunks if (1) there are no idle slaves and (2) this buffer is also full. If all the slaves were busy but the buffer were not full, the thread would just leave the chunk in the buffer and it would proceed to generate new biclusters. In order to empty the buffer, whenever a thread succeeds in assigning a chunk of biclusters to a slave, it also tries to assign chunks stored in the buffer to other potentially available idle slaves.
- Whenever a slave obtains a chunk, it launches one thread per each core assigned to the slave in order to parallelize the processing of the chunk. However, thread creation and destruction is known to be expensive. A better although more complex alternative consists in using a thread pool that allows building and destroying the set of threads once only, dynamically reusing and synchronizing these threads as many times as desired.

5 Evaluation

The tests were performed with configurations analogous to the ones used in [6] in order to facilitate the comparison. Namely, we used sets of 12800, 25600 and 51200 attributes, considering sets of 100 and 200 samples and rates of ones of 10% and 15% (percentage of elements equal to one within the input binary dataset). Similarly, we searched for biclusters with at least 2 samples and 1% of the attributes, that is, 128, 256 and 512 for

Table 2 Experimental environment.

Feature	Value
#Nodes	32
CPUs/Node	2 x Intel Xeon E5-2680 v3
CPU Family	Haswell
CPU Frequency	2.5 GHz
#cores/CPU	12
Total #cores	$32 \times 2 \times 12 = 768$
Memory/node	64GB DDR4
Network	Infiniband FDR@56Gbps
Compiler	g++ 6.4
MPI	Intel MPI 2018.4.274

the datasets with 12800, 25600 and 51200 attributes, respectively.

The experiments were performed in the Linux cluster described in Table 2, which consists of 32 nodes with 24 cores each, totaling 768 cores. The optimization level `O3` was used in all the compilations.

In the graphs we will follow this terminology:

- "original" is the *ParBiBit* implementation.
- "basics" corresponds to that implementation after applying the sequential optimizations explained in Sect. 4.2.1.
- "insertionhint" applies in addition the hint optimization described in Sect. 4.2.2 when inserting in the set.
- "fast set" replaces altogether the standard C++ set with its allocator by the ones explained in Sect. 4.2.3. This way, this version covers all the optimization described in Sect. 4.2, except the insertion hint, which does not apply in our fast set.
- "dyn+xx" is the label for the versions that dynamically scatter chunks of initial biclusters to complete to the slaves proposed in Sect. 4.3. These versions are suffixed by combinations of the abbreviations `bf` and `tp` depending on whether they include the optimizations based on buffering and thread pools, respectively. Unless otherwise stated, the dynamic versions use chunks of 1024 initialized biclusters, and when buffering is used, they reserve a buffer position per slave available. This configuration provides a more than acceptable performance in all scenarios, as will be shown later.

5.1 Evaluation in a single core

In this Section we will first evaluate the performance improvements obtained by the different optimization stages applied in sequential executions in a single core. In a second step, we will analyze these versions from the point of view of their memory consumption.

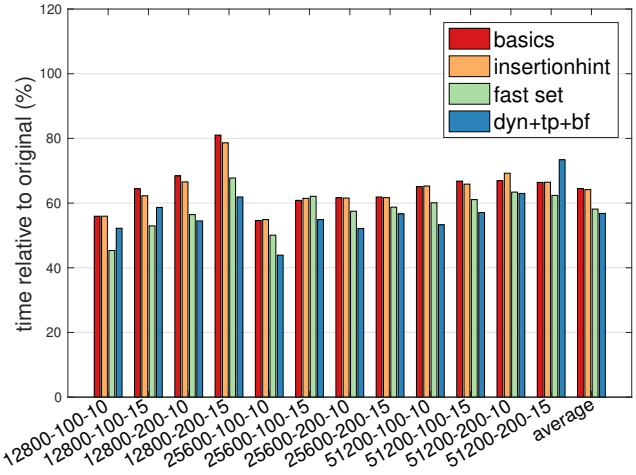


Fig. 6 Sequential execution time of the versions developed as a percentage of the runtime of the original implementation.

5.1.1 Performance evaluation

The influence of the optimizations proposed on the sequential runtime is illustrated in Fig. 6. Namely, the graph shows the sequential execution time for all the versions and problem sizes considered as the fraction of the runtime of the original version for the same problem size. The last group of bars represents the average obtained for each version. Each problem size is labeled as $a - s - p$ in the figure, where a is the number of attributes, s is the number of samples, and p is the percentage of one values. Only one bar is provided for the whole family of dynamic implementations, as its optimizations only come into play when more than one process is in use, thus when using a single process they yield the same runtime regardless of whether the thread pool and the buffering optimizations are used or not.

This experiment reflects that the basic optimizations applied in the first step are by far the most critical ones for the sequential performance, as they remove on average 35.5% of the execution time. The other versions are also useful, as the versions that also apply the insertion hint optimization, the usage of the fast set and the dynamic completion of biclusters interleaved with their initialization have on average 35.85%, 41.85% and 43.2% shorter execution times than the original version, respectively. Nevertheless, as we can see, while they also benefit the runtime, their impact is noticeably smaller than that of the basic optimizations at this point. This makes sense because they were designed mostly with thread contention reduction and scalability in mind.

5.1.2 Memory evaluation

Despite our previous conclusions, the usage of our fast set class may have a crucial impact on the algorithm

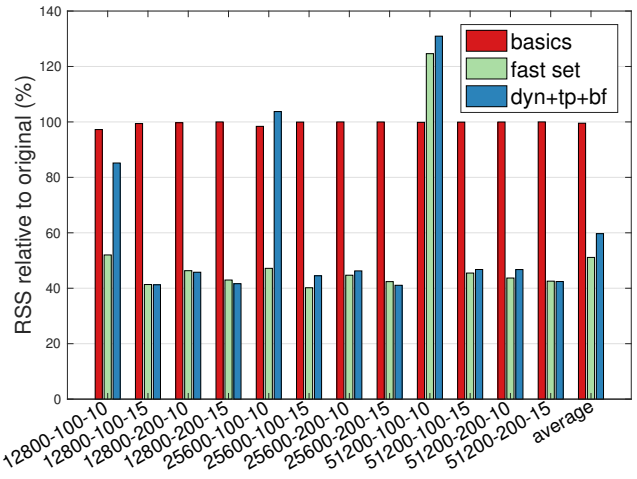


Fig. 7 Resident set size (RSS) of several program versions as a percentage of the RSS of the original implementation.

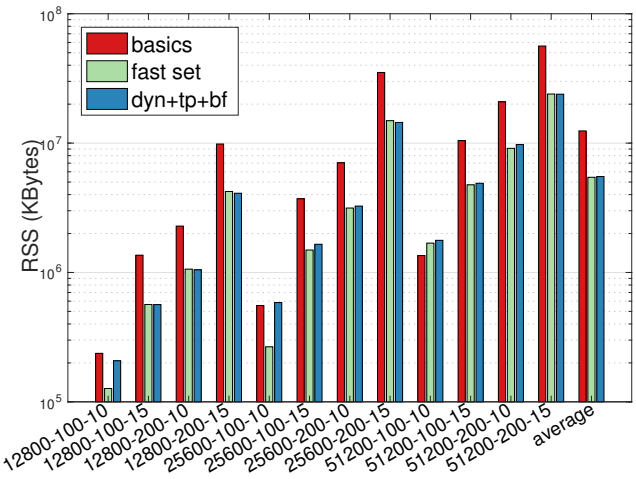


Fig. 8 Resident set size (RSS) of the sequential execution of three program versions.

even in sequential executions. The reason is that it requires significantly less memory than the original version, thus enabling the processing of much larger datasets in memory. Figure 7 shows the process Resident Set Size (RSS) of the basic, fast set and dynamic implementation with its two optimizations as a percentage of the RSS of the original version when using a single process. All our sets fit in memory using the original *ParBiBit* version, as otherwise the disk accesses would totally distort the time measurements. Also, this way measuring the RSS, which can be done with extremely low overhead, provides a good estimation of the total memory required by the processes. The version that relies on the insertion hint optimization uses exactly the same data structures as the version with basic optimizations, and thus it has not been included in the plot. We can see that, despite their large impact on the sequential runtime, the basic optimizations do not modify the memory footprint of the algorithm. The usage of our highly optimized fast set and related allocator, however, reduces on average by a whopping 50% the memory required by the algorithm, the reduction reaching 60% in the best cases.

The 51200−100−10 problem presents a different behavior, as the use of our fast set requires more memory than the original implementation. The reason is related to the fact that, as explained in Sect. 4.2.3, our fast set is based on a hash table, and creating this table with a sizable number of entries helps reduce the number of conflicts, and thus also the search times within our set. Our implementation currently heuristically creates $a^2/20$ entries in the table where a is the number of attributes in the problem, thus estimating a $\sim 10\%$ of attribute combinations leading to initialized biclusters before generating conflicts. In the case of the experiments with 100 samples and 10% of ones the actual number of different biclusters that are initialized is much smaller than the heuristic estimation because of the small number of samples and ones, and thus the preallocation of the fast set table wastes space. However, this only requires more memory than the initial implementation in the test with 51200 attributes because the heuristic is quadratic on this parameter, and in the tests with fewer attributes the other storage advantages of our implementation, such as the optimized consecutive storage of the patterns, offset this fact. This way, we only consume more memory in some scenarios that initialize relatively small numbers of biclusters and, as Fig. 7 shows, in the vast majority of the cases the fast set requires noticeably less memory. This is particularly true in the case of problems that require the largest working sets. This is illustrated in Fig. 8, which shows the RSS using a logarithmic scale so that they are all visible. Notice that the fact that the scale is logarithmic also implies that a short relative visual difference actually implies a large practical one in terms of memory. Also, more advanced heuristics could be followed to estimate the size of the fast set hash table.

Regarding the dynamic implementation, it uses somewhat more memory than the fast set version. The main reason is the partition in chunks of the storage of the initialized biclusters, as this generates more fragmentation, and thus more memory consumption, than the consecutive sequential storage in a single buffer used by the non-dynamic versions. It deserves to be mentioned that the measurements were performed using chunks of 1024 initialized biclusters, which is a heuristic value derived from extensive experimentation as we will see soon. A straightforward way to reduce the fragmentation and thus the memory consumption would be to

simply increase the chunk size. Still, as in the case of our fast set, the additional memory overheads of the dynamic implementation only play a relevant role for small problem sizes, being minimal for the large ones.

The two stages of the algorithm that consume more memory are, with a very large difference over the other ones, the bicluster completion, followed by the bicluster initialization. Nevertheless, only the memory consumed in the bicluster initialization limits the applicability of the tool to larger problems. One reason is that, while the whole set of completed biclusters can require more memory than the set of initialized biclusters and different bit patterns found, the former are distributed among all the processes used in the execution, while the latter only reside in the master process. Another and even more important reason is that while the master process needs to have access to all the bit patterns in order to verify whether a newly initialized bicluster provides a new bit pattern not previously found, there is no need at all for the completed biclusters to be kept in the memory of the processes that complete them. Rather, since the completed biclusters are just the output of the algorithm, it is perfectly feasible to periodically dump them to disk and then deallocate them in each process. This could be easily done using a separate file per MPI process and joining all the files in a single one in a post-processing stage once the algorithm finishes. For this reason, the reduction of the memory footprint provided to the bicluster initialization stage by our fast set and its allocator is indeed the critical improvement needed to enable the application of the algorithm to larger datasets.

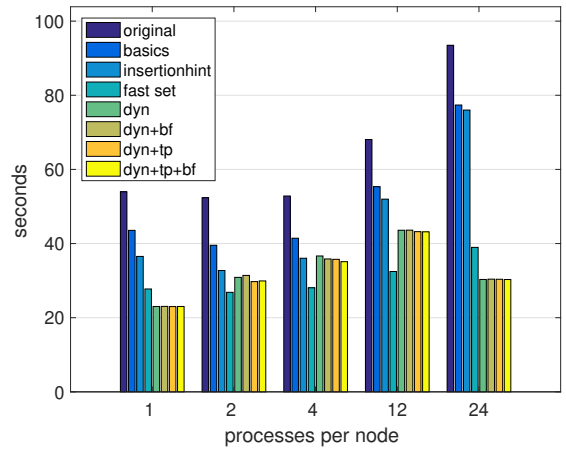
5.2 Multicore and multinode evaluation

When performing parallel executions our application requires choosing two critical parameters. The first one, which is in common with *ParBiBit*, is the number of processes per node. The second one is the chunk size used in the new proposed dynamic high level structure of the algorithm described in Sect. 4.3, which overlaps the initialization and the completion of the biclusters. For this reason, this experimental section is divided into two parts. While the first one is primarily devoted to the analysis and selection of these parameters, it will also allow us to make important observations on the performance in parallel executions of the different versions developed, since it will be the first contact with these kinds of executions in this paper. The second part of the evaluation consists in a detailed parallel performance evaluation of all the versions developed in this manuscript once the critical parameters mentioned above have been settled.

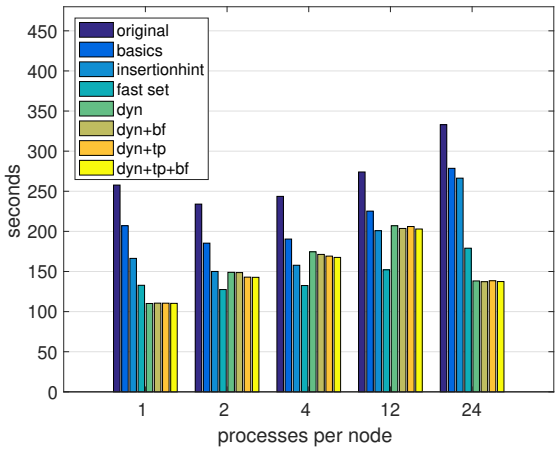
5.2.1 Selection of parameters for the parallel executions

In order to learn the best number of processes per node, we started with experiments based on the two datasets of 12800 attributes and 200 samples, which differ in the ratio of one values (10% or 15%), and we considered executions using a single node, four and sixteen nodes. For each combination we run all our versions measuring the execution time when using $p = 1, 2, 4, 12$ and 24 processes per node. Also, since our nodes have 24 physical cores, $24/p$ threads were always assigned to each process. The results are displayed in Fig. 9, where the program versions are labelled according to the terminology introduced at the beginning of Sect. 5. The family of non-dynamic implementations, i.e., those that operate at the high level as the original *ParBiBit* application, present the behavior observed in [6]. Namely, they consistently offer the best performance when two processes are used per node. The behavior of the dynamic versions (dyn+xx) is clearly different from that of the first family of versions with respect to the number of processes per node:

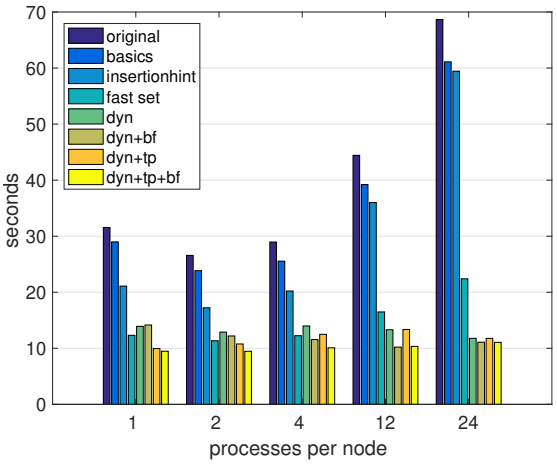
- When a single node is used, they always achieve the shortest runtime using a single process. There are two reasons for this. First, the dynamic implementation never uses MPI communications when a single process is used, as the process detects that there are no slaves and thus the threads that generate the chunks of initialized biclusters also perform their completion. Second, the threads that are busy completing biclusters do not compete for the use of the shared fast set, thus reducing the contention during the initialization stage.
- When four nodes are used, however, the versions that do not reuse threads (dyn and dyn+bf) perform better with the largest numbers of processes per node, which allow them to create and join fewer threads, while the versions that incorporate a thread pool (dyn+tp and dyn+tp+bf) continue to perform better using a single process. We must realize that when we have 4 nodes with p processes per node, the master has $24/p$ parallel threads generating chunks that must be processed as soon as possible by $4p-1$ slave processes, as otherwise we run the risk of having to process them in the master. As a result, for small values of p the response time of the slaves, which can only accept new chunks when all their threads have finished processing the previous assigned chunk, is critical in order to have slaves available whenever the master threads generate new work.
- Finally, at sixteen nodes the best actual runtime is always achieved using 2 processes per node, al-



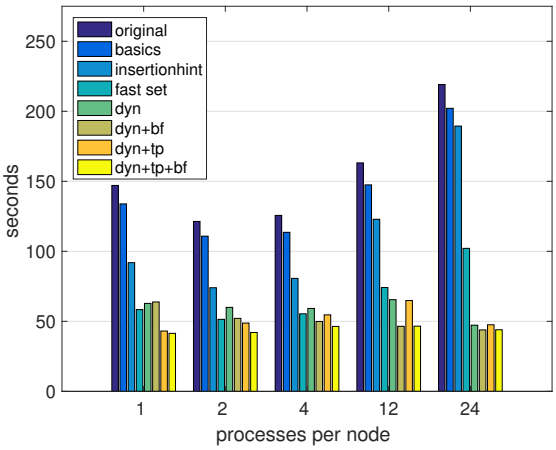
(a) 10% one values - 1 node



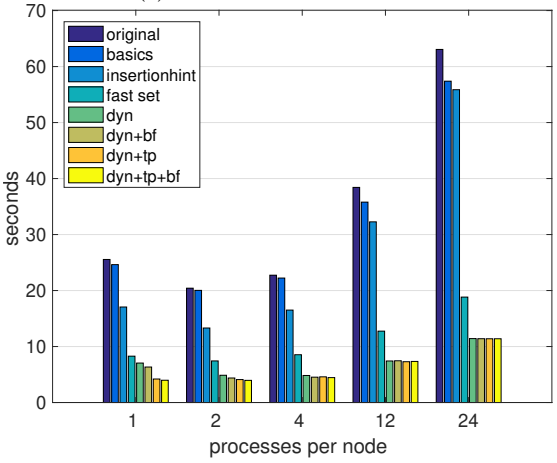
(b) 15% one values - 1 node



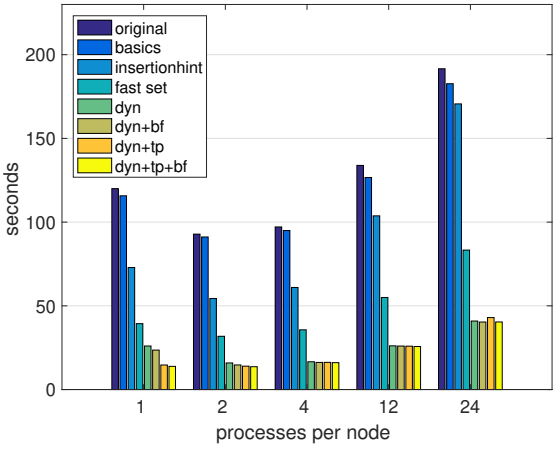
(c) 10% one values - 4 nodes



(d) 15% one values - 4 nodes



(e) 10% one values - 16 nodes



(f) 15% one values - 16 nodes

Fig. 9 Influence on the number of processes per node for the execution on datasets of 12800 attributes and 200 samples considering datasets with 10% and 15% one values on 1, 4 and 16 nodes. Each process uses 24/#processes_per_node threads.

though using only one provides a remarkably similar performance to the versions that incorporate the thread pool. The change in the behavior comes from the fact that now the threads of the master have $16p - 1$ slaves available, so that now it is much easier to find a slave available.

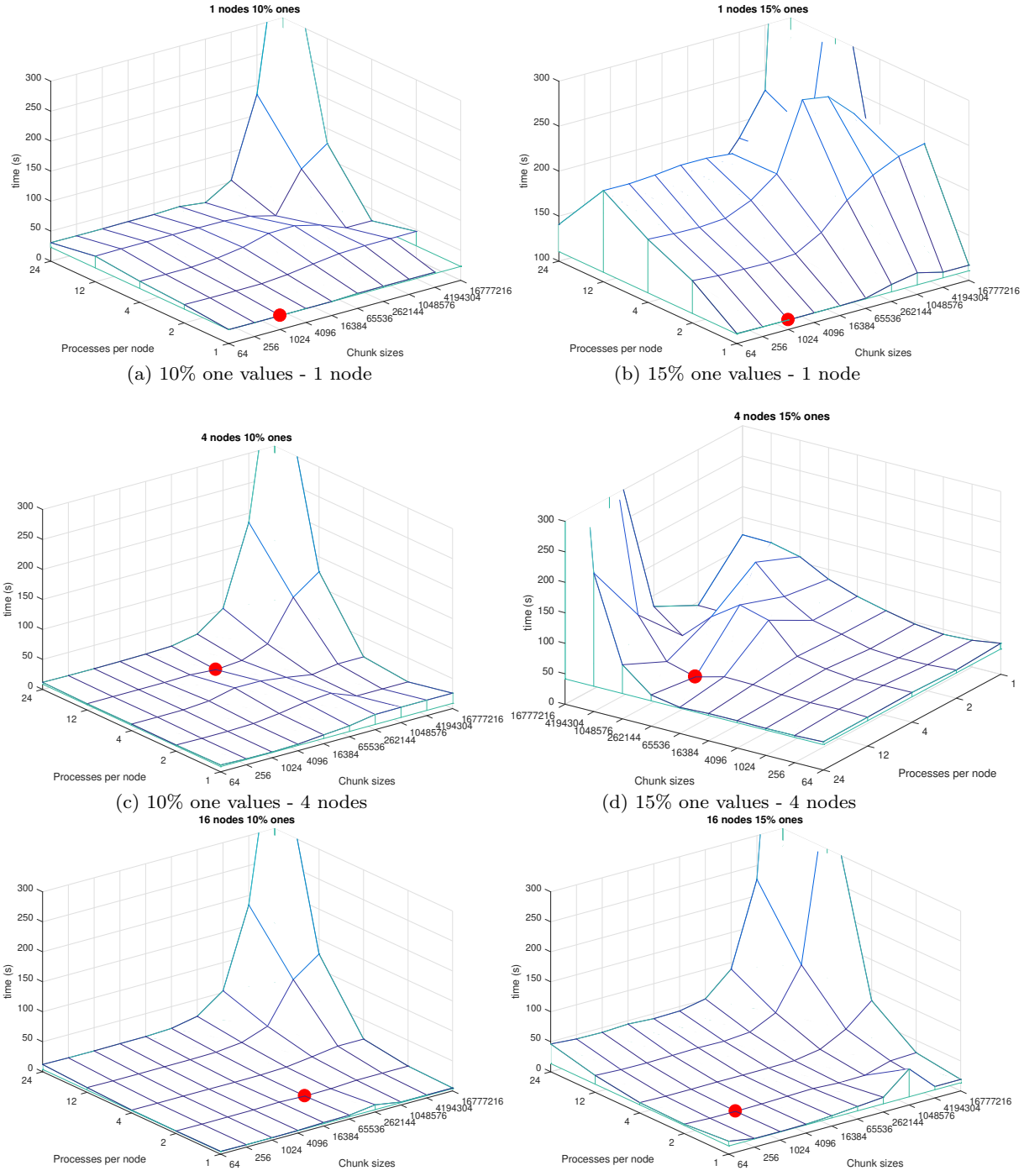
The results displayed in Fig. 9 also show that the insertion hint, fast set and dynamic versions of the algorithm present much clearer improvements than those

observed in Fig. 6 with respect to the implementation that only relies on our initial basic optimizations, now that we consider parallel executions. For example, we can see how the insertion hint optimization was indeed a very good idea given its very reduced programming cost, often impacting on the parallel runtimes much more than all the basic optimizations together. As a final comment on the non-dynamic versions, Fig. 9 also demonstrates that, despite the good results achieved by the insertion hint optimization, the complex work involved in the development of our own fast set class and its associated memory manager paid off with large additional performance gains in the parallel runs. In fact this change is the one that contributes the most to the improvement of the original implementation of the algorithm. Also, as one would expect, the dynamic versions can achieve even better performance thanks to the ability to overlap the initialization and the completion stage of the biclusters. The two optimizations proposed for this second family of implementations (buffering and thread pool) are of interest, as the version that incorporates both is the one that systematically achieves the best performance.

Figure 10 explores the selection of the chunk size of the dynamic versions relying on the most optimized implementation of this family, the one that uses both a thread pool and buffering for the chunks (dyn+tp+bf). We used the same problem sizes as in Fig. 9 and we measured the performance achieved when using 1, 4 and 16 nodes and 1, 2, 4, 12 and 24 processes per node considering chunks of 2^{2i} initialized biclusters, for $3 \leq i \leq 12$. The figures mark with a thick point the best runtime achieved, and their z axis is cut at 300 seconds in order to focus on the area of interest. Figure 10(d) had to be rotated in order to avoid hiding the best time behind the peaks in the graph. We see that a chunk size of 1024 initialized biclusters combined with the usage of a single process per node is the best option when using a single node, and while not optimal, it is a reasonable configuration in the other experiments. In fact, it is at most 10% slower than the optimal point, which occurs in the experiment using 10% one values and 4 nodes.

A general tendency is that, if we dismiss very large chunk sizes, the graphs become flatter as the number of nodes increases. This makes sense because this increases the number of slaves, making increasingly feasible that the master finds idle slaves when it generates a new chunk. Another tendency is that large chunk sizes are negative when combined with large number of processes per node, and thus, slaves. This occurs because using large chunks gives place to infrequent and costly generations of new chunks in the master, and thus these numerous slaves may remain idle more often during execution. Using large numbers of processes per node has two more potentially negative consequences because it reduces proportionally the number of threads per process. First, since only the master process performs bicluster initialization, it reduces the parallelism in this stage, and thus the ability of the master to feed all the slaves, particularly when the chunk sizes are large, as we have just explained. Second, bicluster completion can be quite unbalanced, and while slaves can balance the workload among their threads, each slave is solely responsible for the completions of the chunks it has been assigned. Thus, while the increase of processes per node increases the number of slaves among which to distribute chunks, it can also increase the unbalance in their workload in unpredictable ways. Despite these disadvantages, at moderate number of nodes above 1, such as 4, large numbers of processes per node can give place to the best runtimes, although they are similar to those obtained using the heuristic suggested of one process per node and chunks of 1024 initialized biclusters.

While the previous experiments clearly point that, at least for our dyn+tp+bf version, using one process per node is a reasonable heuristic, we wanted to obtain a more detailed global evaluation of this heuristic on all the dynamic versions and for all the problem sizes. With this purpose we run all the dynamic versions of *ScalaParBiBit* for all the problems considered on 1, 2, 4, 8, 16 and 32 nodes, trying executions with 1, 2, 4, 12 or 24 processes per node and the default chunk size of 1024. Then we measured the slowdown obtained when using a single process per node in each execution with respect to the time associated to the usage of the optimal number of processes per node. The results, shown in Fig. 11, which represents the average slowdown for the executions using different numbers of nodes for each problem considered, are consistent with our previous more limited experiments. This way, once the thread pool implementation is incorporated in the dynamic versions, the heuristic of using one process per node is on average 1.5% slower than using the optimal number of processes per node when the buffering optimization is also incorporated and just 0.6% slower when the buffering optimization is not used. Notice that these values are about the typical runtime variation observed among different runs that use the same configuration or smaller. When thread pools are not used, however, the heuristic is not as good, as it can lead to runtimes that are on average 16% slower than those achieved using the optimal number of processes per node, or 19.8% if the buffering optimization is used.



(a) 10% one values - 1 node

(b) 15% one values - 1 node

(c) 10% one values - 4 nodes

(d) 15% one values - 4 nodes

(e) 10% one values - 16 nodes

(f) 15% one values - 16 nodes

Fig. 10 Influence on the chunk size combined with different numbers of processes per node for the execution on datasets of 12800 attributes and 200 samples considering datasets with 10% and 15% one values on 1, 4 and 16 nodes for the dyn+tp+bf version.

5.2.2 Final parallel evaluation

Figures 12 to 14 show the speedup of all the versions when using 1, 2, 4, 8, 16 and 32 nodes taking as baseline the execution of the original version in a single node for the working sets of 12800, 25600 and 51200 attributes, respectively. These results illustrate not only the scalability, but also the comparison of the performance of the different versions developed in this paper as well as *ParBiBit*. The versions of the first family were run using 2 processes per node, as this always gives place to the best runtime. For the dynamic versions, the left

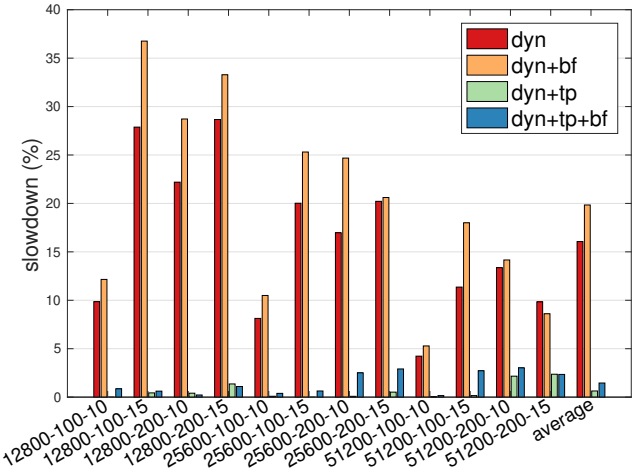


Fig. 11 Average overhead of using a single process per node with respect to the optimal one among using 1, 2, 4, 12 or 24 processes per node in executions on 1, 2, 4, 8, 16 and 32 nodes.

column of each figure plots the best performance obtained using either 1, 2, 4, 12 or 24 processes per node, while the right column plots the performance obtained using one process per node. In all the cases, chunks of 1024 initialized biclusters were used. We have decided to provide the plots of the two columns in order to clearly demonstrate that the dynamic versions that use a thread pool executed with a single process per node are consistently the best ones, and that, as a result, it is safe to recommend the usage of dyn+tp+bf with one process per node as the best version. Without these plots, given the results in the previous experiments it would have been very reasonable to doubt whether the dynamic versions without thread pool could actually be better if the user searches for the optimal number of processes per node for them. In fact we can see that when the best number of processes per node is searched, the performance of these versions trails but is not so far from dyn+tp+bf, and they do sometimes overperform dyn+tp.

These graphs show that, not only the original version, but also the first improved versions presented in this paper present a very limited scalability. They also clearly illustrate how our highly optimized set class based on hash tables, atomic operations and a careful memory management is undoubtedly the most beneficial change performed in the implementations of the first family. While its use alone increases the scalability of the algorithm noticeably, the modification of the structure to enable overlapping of the initialization and the completion of biclusters in the dynamic versions further increases the scalability. Also, although all the versions of this second family scale pretty well, the usage of the thread pool and/or buffering give place to improvements that tend to be more noticeable as the number of nodes used grows.

As mentioned before, Figs. 12 to 14 take the runtime of the original *ParBiBit* in one node as common baseline for all the speedups so that they provide a reliable comparison among all the versions. This also implies however that they are not proper speedup lines for each version, which would have required to take as baseline the runtime of that version. In this regard, it is remarkable that despite being much faster than *ParBiBit* in a sequential execution, our versions also scale much better in an absolute way. This way, while our final version dyn+tp+bf is on average 78.6% faster than *ParBiBit* when executed in a single core, the average speedup (with respect to itself) when using the 764 cores in our cluster across all the problem sizes tested is 244.38, which is roughly 3.7 times the average speedup of 66.2 achieved by *ParBiBit*.

6 Conclusions

As the amount of available data and the possibilities to extract meaningful information from them grow day by day, so does the need for tools that allow us to efficiently perform the related processing. In this paper we have tackled the problem of raising the computational capacity to apply the biclustering data mining technique, widely used in many fields of knowledge, to binary data. For this, we analyzed and improved the high-performance tool *ParBiBit* [6], which already provided very good performance thanks to its development on a compiled language such as C++ and the hybrid parallelization for distributed and shared memory on top of MPI and C++ threads, respectively.

Our new version, called *ScalaParBiBit* not only offers much better scalability on multicore clusters, as its name implies, but it also provides shorter execution times and it makes more efficient usage of memory, thus allowing to process larger datasets. This way, the average speedup achieved by *ScalaParBiBit* in a current multicore cluster with respect to *ParBiBit* across the set of problems considered in our experiments goes from 78.6% when both use a single core to 492.8% when both take advantage of the 764 cores of our system. The source code of our tool, together with building and usage instructions are freely available at https://github.com/fraguela/ScalaParBiBit under an open source license.

Possible future lines of work are porting the most time consuming portions of the application to hardware accelerators in order to further speedup the processing

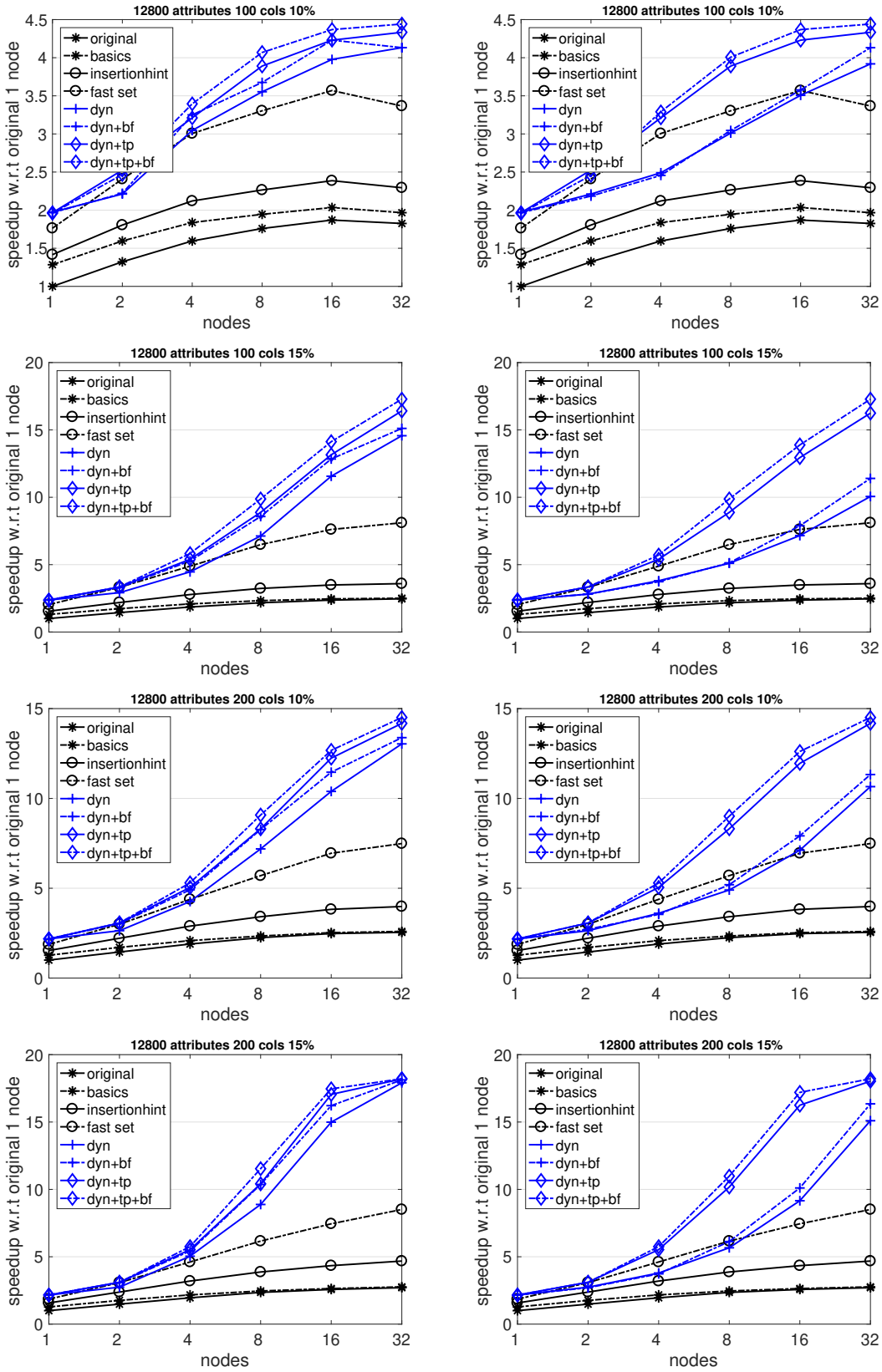


Fig. 12 Speedups of the experiments using 12800 attributes taking as baseline the execution of the original implementation in a single node. The figures in the left column show the best result for the dynamic version when using 1, 2, 4, 12 or 24 processes per node, while the right column reflects the usage of 1 process per node for these versions.

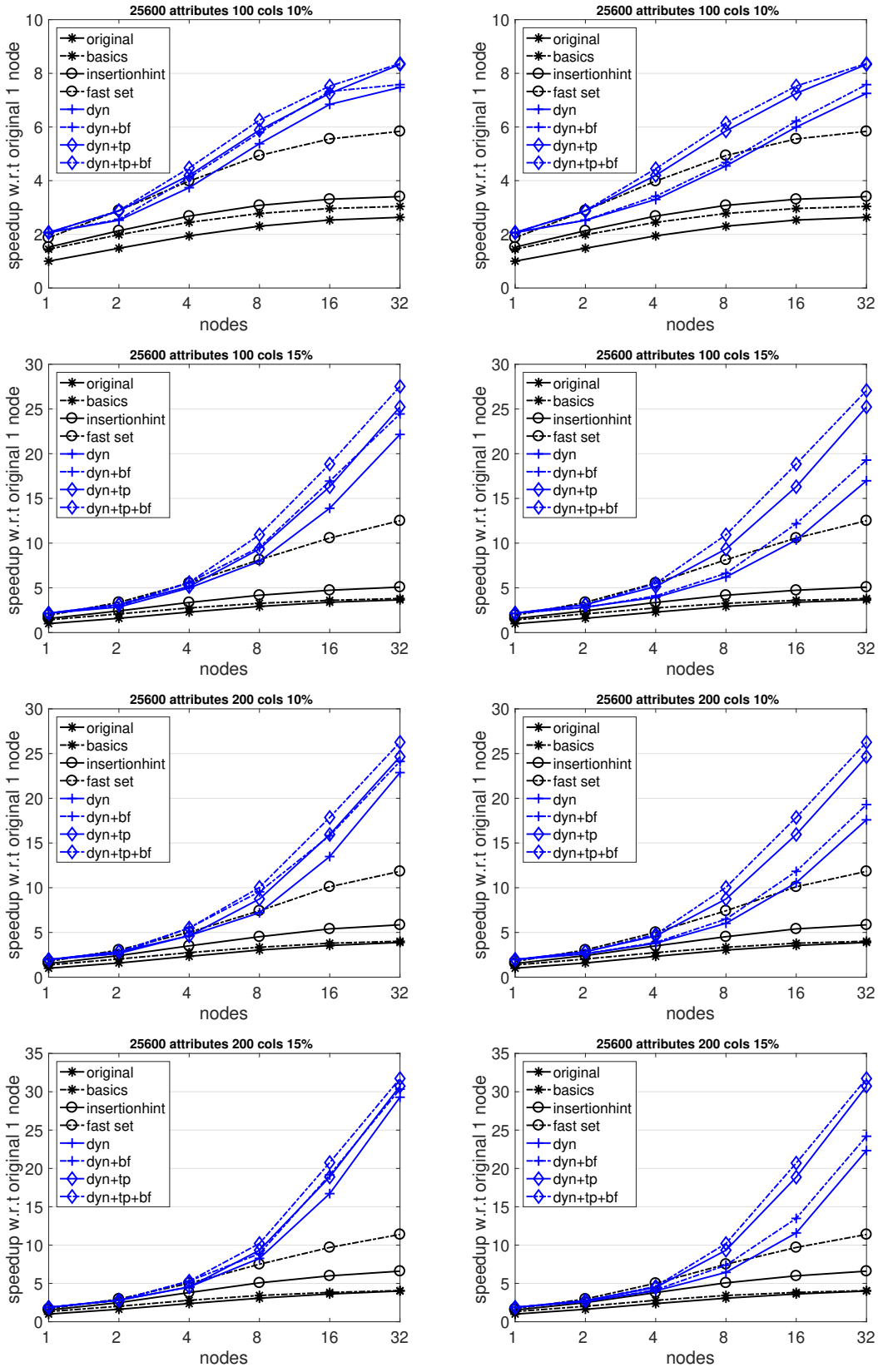


Fig. 13 Speedups of the experiments using 25600 attributes taking as baseline the execution of the original implementation in a single node. The figures in the left column show the best result for the dynamic version when using 1, 2, 4, 12 or 24 processes per node, while the right column reflects the usage of 1 process per node for these versions.

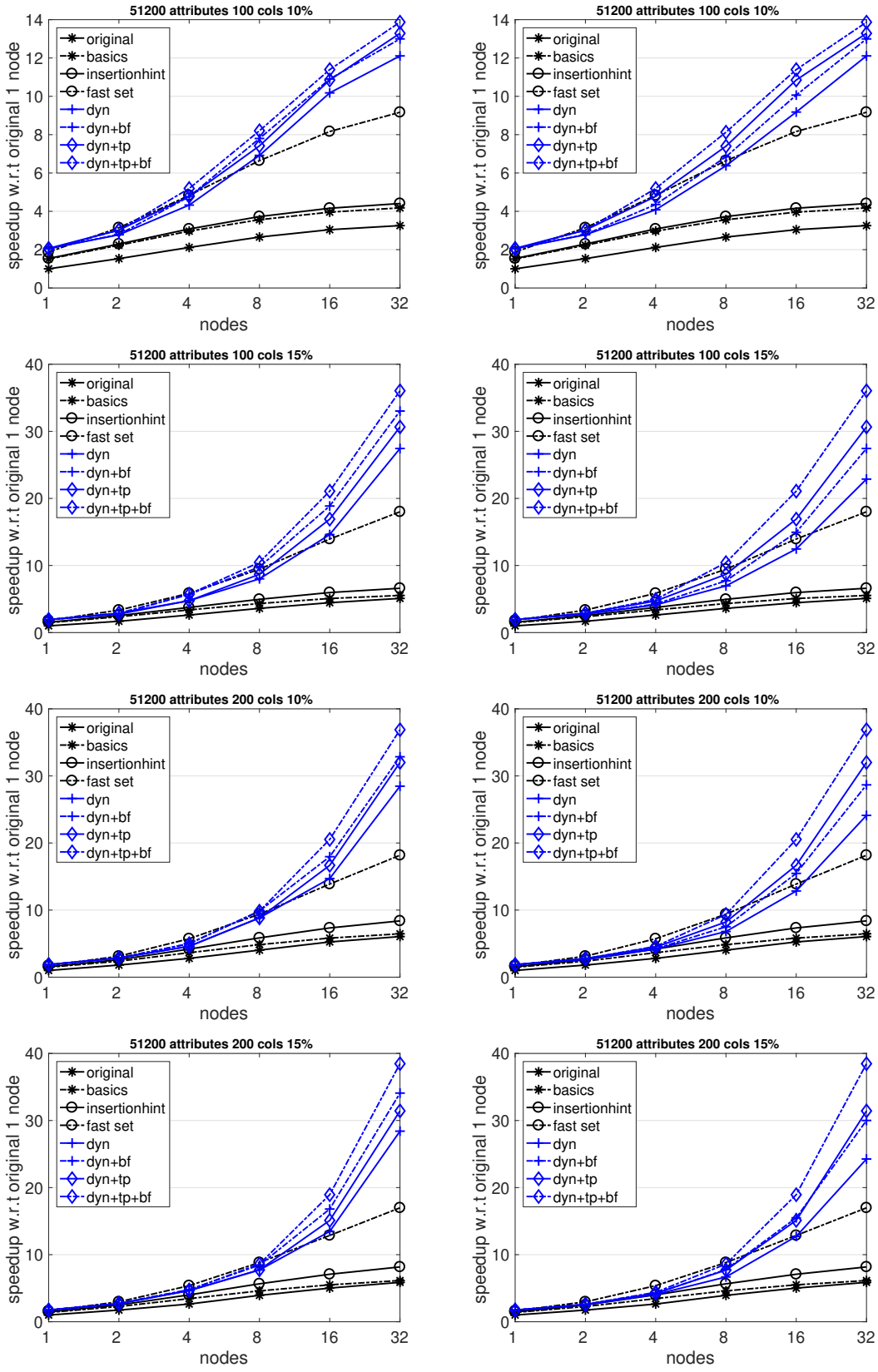


Fig. 14 Speedups of the experiments using 51200 attributes taking as baseline the execution of the original implementation in a single node. The figures in the left column show the best result for the dynamic version when using 1, 2, 4, 12 or 24 processes per node, while the right column reflects the usage of 1 process per node for these versions.

and developing a version of the application for non-binary biclustering.

Data availability statement

The application developed in this manuscript, together with building and usage instructions, as well as the datasets used in the experiments are publicly available under an open source license at https://github.com/fraguela/ScalaParBiBit.

Acknowledgements

This research was supported by the Ministry of Science and Innovation of Spain (TIN2016-75845-P and PID2019-104184RB-I00, AEI/FEDER/EU, 10.13039/501100011033), and by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04). We acknowledge also the support from the Centro Singular de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01. We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of their resources.

References

1. Bhatnagar R, Kumar L (2009) High performance parallel/distributed biclustering using Barycenter heuristic. In: 2009 SIAM International Conference on Data Mining, Sparks, NV, USA, SDM 2009, pp 1050–1061
2. Chen HC, Zou W, Tien YJ, Chen JJ (2013) Identification of bicluster regions in a binary matrix and its applications. PLOS ONE 8(8):e71680
3. Feng G, Li Z, Zhou W, Dong S (2020) Entropy-based outlier detection using Spark. Cluster Computing 23(2):409–419
4. González CH, Fraguela BB (2015) Enhancing and evaluating the configuration capability of a skeleton for irregular computations. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, Finland, PDP 2015, pp 119–127
5. González CH, Fraguela BB (2017) A general and efficient divide-and-conquer algorithm framework for multi-core clusters. Cluster Computing 20(3):2605–2626
6. González-Domínguez J, Expósito RR (2018) ParBiBit: Parallel tool for binary biclustering on modern distributed-memory systems. PLOS ONE 13(4):e019436
7. González-Domínguez J, Expósito RR (2019) Accelerating binary biclustering on platforms with CUDA-enabled GPUs. Information Sciences 496:317–325
8. Hoefler T, Dinan J, Thakur R, Barrett B, Balaji P, Gropp W, Underwood K (2015) Remote memory access programming in MPI-3. ACM Trans Parallel Comput 2(2):9:1–9:26
9. Isokpehi RD, Johnson MO, Campos B, Sanders A, Cozart T, Harvey IS (2020) Knowledge visualizations to inform decision making for improving food accessibility and reducing obesity rates in the United States. International Journal of Environmental Research and Public Health 17(4):1263
10. Jiang F, Leung CKS (2014) Mining interesting following patterns from social networks. In: 16th International Conference on Data Warehousing and Knowledge Discovery, Munich, Germany, DaWaK 2014, pp 308–319
11. Koniges A, Cook B, Deslippe J, Kurth T, Shan H (2016) MPI usage at NERSC: Present and future. In: 23rd European MPI Users' Group Meeting, Edinburgh, United Kingdom, EuroMPI 2016, pp 217–217
12. Lee Y, Kim Y, Yeom HY (2020) Lightweight memory tracing for hot data identification. Cluster Computing 23(3):2273–2285
13. Li Z, Chang C, Kundu S, Long Q (2020) Bayesian generalized biclustering analysis via adaptive structured shrinkage. Biostatistics 21(3):610–624
14. Lin Q, Xue Y, Chen WS, Ye SQ, Li WL, Liu JJ (2015) Parallel large average submatrices biclustering based on MapReduce. In: 11th International Conference on Computational Intelligence and Security, Shenzhen, China, CIS 2015
15. Lin Q, Zhang H, Wang X, Xue Y, Liu H, Gong C (2019) A novel parallel biclustering approach and its application to identify and segment highly profitable telecom customers. IEEE Access 7:28696–28711
16. López-Fernández A, Rodríguez-Baena D, Gómez-Vela F, Divina F, García-Torres M (2021) A multi-GPU biclustering algorithm for binary datasets. Journal of Parallel and Distributed Computing 147:209–219
17. Nisar A, Ahmad W, Liao WK, Choudhary A (2015) An efficient Map-Reduce algorithm for computing formal concepts from binary data. In: 3rd IEEE International Conference on Big Data, Santa Clara,

CA, USA, Big Data 2015, pp 1519–1528
18. Padilha VA, Campello R (2017) A systematic comparative evaluation of biclustering techniques. BMC Bioinformatics 18:55
19. Pontes B, Giráldez R, Aguilar-Ruiz JS (2015) Biclustering on expression data: a review. Journal of Biomedical Informatics 57:163–180
20. Prelic A, Bleuler S, Zimmermann P, Wille A, Bühlmann P, Gruissem W, Hennig L, Thiele L, Zitzler E (2006) A systematic comparison and evaluation of biclustering methods for gene expression data. Bioinformatics 22(9):1122–1129
21. Rathipriya R (2016) A novel evolutionary biclustering approach using MapReduce (EBC-MR). International Journal of Knowledge Discovery in Bioinformatics 6(1):26–36
22. Rocha O, Mendes R (2018) JBiclustGE: Java API with unified biclustering algorithms for gene expression data analysis. Knowledge-Based Systems 155:83–87
23. Rodriguez MZ, Comin CH, Casanova D, Bruno OM, Amancio DR, Costa LdF, Rodrigues FA (2019) Clustering algorithms: A comparative approach. PLOS ONE 14(1):e0210236
24. Rodríguez-Baena DS, Pérez-Pulido AJ, Aguilar-Ruiz JS (2011) A biclustering algorithm for extracting bit-patterns from binary datasets. Bioinformatics 27(19):2738–2745
25. Sarazin T, Lebbah M, Azzag H (2014) Biclustering using Spark-MapReduce. In: 2nd IEEE International Conference on Big Data, Washington DC, USA, Big Data 2014, pp 58–60
26. Saxena A, Prasad M, Gupta A, Bharill N, Patel OP, Tiwari A, Er MJ, Ding W, Lin CT (2017) A review of clustering techniques and developments. Neurocomputing 267:664–681
27. Stroustrup B (2013) The C++ Programming Language, 4th edn. Addison-Wesley Professional
28. Wei L, Ling C (2008) A parallel algorithm for gene expressing data biclustering. Journal of Computers 3(10):71–77
29. Wu H, Cheng S, Wang Z, Zhang S, Yuan F (2020) Multi-task learning based on question–answering style reviews for aspect category classification and aspect term extraction on GPU clusters. Cluster Computing 23(3):1973–1986
30. Yoon S, Nguyen HC, Jo W, Kim J, Chi SM, Park J, Kim SY, Nam D (2019) Biclustering analysis of transcriptome big data identifies condition-specific microRNA targets. Nucleic Acids Research 47(9):e53–e53