Frequent Itemset Mining Problem

Juliette Fournis d'Albiat

August 2022

1 Introduction

Frequent Itemset Mining is a method for market basket analysis, a data mining technique used by retailers to increase sales by better understanding customer purchasing patterns. The idea is to find products that are frequently bought together. This technique is very popular among mail-order companies or online shops. It can also be applied to improve arrangement of products in shelves or on a catalog's pages, support cross-selling, detect fraud, etc. In this work, we present our approach to design efficient distributed algorithms to solve the frequent itemize problem. In section 1, we gives the main definitions for understanding the problem and we also rewrite the problem mathematically, in section 2 we define a generic algorithm for solving the problem, in section 3 we present the algorithms that we have implemented to solve the problem, in section 4 we present the data structures that we used, in section 5 we present how we have parallelized the algorithms and in section 6 we describe our experiments.

2 Problem Definitions

Let's assume we are given a set of items I_n of size n.

- An itemset x is a set of items.
- A transaction is a couple $t = (id, u)$ where id is the transaction identifier and u is an itemset.
- A transaction (id, u) supports an itemset x if all items in x appear in u.
- Given a set of transactions D and a threshold S, an itemset is frequent if it is supported by at least S transactions in D.

The Frequent Itemize Mining Problem is

Given a set of transactions D, find itemsets which are frequent.

In the following we will use these notations.

- I_n is the segment $[|0, n-1|]$ which represents the set of items, item i represents the i-th most frequent item.
- $P(I_n)$ is the set of all parts of I_n, $P(I_n)$ represents the set of itemsets.
- The lattice $(P(I_n), \subset)$ represents the search space.
- D is a subset of $P(I_n)$ which represents the database.
- f is a function that gives for x in $P(I_n)$, D a database, the number elements in D that supports x.
- S is a positive integer, defining the frequent elements, a element x is frequent if $f(x, D) \geq S$
- $F = \{x \in P(I_n) | f(x, D) \geq S\}$ is the set of frequent itemsets

With these notations, we can express the frequent itemset mining problem as, let's D a database, S a positive integer, f a function that gives for an itemset x, the number elements in D that supports x,

find $F = \{x \in P(I_n) | f(x, D) \geq S\}$

2.1 Solving the Frequent Itemize Mining (FIM) Problem

2.1.1 A basic approach

There exists a very simple algorithm which solves this problem,

Algorithm 1 A simple algorithm which solves the FIM problem

```
$F \leftarrow \emptyset$
for every $x$ in $P(I_n)$ do
    if $f(x, D) \geq S$ then
        add $x$ to $F$
return $F$
```

But this algorithm has a very bad complexity, indeed given a set of item I_n, it will enumerate 2^n itemsets.

2.1.2 Defining a better approach

"All subsets of a frequent itemset must be frequent."

This property is the key behind every efficient solution since it enables to prune the space of itemsets. With the notation defined previously, this property says that f is increasing on the lattice $(P(I_n), \subset)$.

Given a subset X of $P(I_n)$, we define the border of X as the minimal elements not in X. Let's define B as the minimal elements not in F,

$$B = \{y | f(y, D) < S \text{ and if } f(x, D) < S \text{ then } y \subseteq x\}$$

B is the border of F. From B we can compute F in a complexity linear in the size of F using that

$$F = \{x | x \subset y, y \in B\}$$

Now the ideas of the algorithms are to discover B, beginning the search from the smallest itemsets and gradually compute greater itemsets. In order to do so we maintain a set of frequent itemsets and explore itemsets in the border of this set till reaching elements of B.

Algorithm 2 A generic algorithm which solves the FIM problem

```
$F \leftarrow \emptyset$
$B \leftarrow \emptyset$
while it exists unvisited itemset $x$ in the border of $F$ do
    if $f(x, D) \geq S$ then
        add $x$ to $F$
    else
        add $x$ to $B$
return $F$
```

The complexity of this algorithm will be really better than the one presented before since F is really smaller than $P(I_n)$ and with this algorithm we will only iterate over $F \cup B$.

2.2 Browsing the space of itemsets

The two main approaches used to browse the space of itemsets are the apriori and eclat algorithms. The search space, $(P(I_n), \subseteq)$ is a lattice, thus it has a graph structure, its nodes are the itemsets and for two itemset a and b, there is an edge from a to b if it exists an item $i \in I_n$, such that $b = a \cup \{i\}$.
Thus $(P(I_n), \subseteq)$ can be explored incrementally using a depth first search or a breadth first search. The apriori algorithm uses the depth search while the eclat algorithm uses the breadth first search approach.

2.2.1 The apriori Algorithm

The apriori algorithm explores $(P(I_n), \subset)$, using a depth search approach. It proceeds by iteratively incrementing the depth, for a fixed depth k, it maintains F_k, the set of frequent itemsets of size k and it computes a set of candidates C_{k+1}, the set of itemsets which are in the border of F_k (i.e. the elements at depth $k + 1$ that may be frequent).

2.2.2 The eclat Algorithm

The eclat algorithm explores $(P(I_n), \subset)$, uses a depth first search. It starts from the most frequent itemset (which is $\{0\}$) From a frequent itemset $x = \{i_0, i_1, ..., i_m\}$, it distinguish tree cases:

Algorithm 3 The apriori algorithm

```
$F_1 \leftarrow \{$ frequent itemset of size 1$\}$
$k \leftarrow 1$
while $F_{k-1} \neq \emptyset$ do
    $C_k \leftarrow$ border of $F_{k-1}$
    $k \leftarrow k+1$
    for $x \in C_k$ do
        if $f(x, D) \geq S$ then
            add $x$ to $F$
return $\cup_k F_k$
```

- x is a root, then it reccurses by removing the last added item i_m from x and and explores the next unvisited node in the DFS, which is obtained by removing i_{m-1} and adding $i_{m-1}+1$.
- x is infrequent then it reccurses by removing the last added item i_m and explores the next unvisited node in the DFS, which is obtained by adding i_{m+1}.
- x is a frequent inner node, it explores $x \cup \{i_{m+1}\}$.

Algorithm 4 The eclat algorithm

```
We use the notation $x = \{i_0, i_1, ..., i_m\}$ where $i_0 \leq i_1 \leq ... \leq i_m$
  $x \leftarrow \{0\}$
  while x $\neq \{n-1\}$ do
    if (n - 1) $\in x$ then
        $x \leftarrow (x/\{i_{m-1}, i_m\}) \cup \{i_m - 1 + 1\}$
    else if $f(x, D) < S$ then
        $x \leftarrow (x/\{i_m\}) \cup i_{m+1}$
        add $x$ to $F$
    else
        $x \leftarrow x \cup i_{m+1}$
        add $x$ to $F$
    return $F$
```

2.3 Finding efficient data structures

We need to define efficient data structures for representing our database and mine itemsets. Efficient representations should

- enable to compute $f(x, D)$ fastly
- be generated in a reasonable time
- have a reasonable size (the size should fit in the memory)

2.3.1 Filtering the database

To construct a efficient representation of itemsets, first we have removed every non frequent item of the database, since they will not appear in frequent itemsets. We have sorted the n frequent items by increasing frequency and mapped them to I_n.

Thus from a file where each line represents an itemset and each item is encoded as an integer, we have computed a sorted list of list of integers, and two maps from a representation to the other. The size of this representation is closed to $2m$ bytes, where m is the number of appearances of a frequent item in the database.

2.3.2 Encoding itemsets

We have tried two different representations

- Encode an itemset as a sorted list of integers.
- Encode an itemset as a bitmap of size n.

Since in practice n is smaller than 64, a bitmap can be encoded as an integer, then the first representation uses $2k$ bytes for an itemset of length k, while the second uses only 2 bytes.

2.3.3 Encoding the Database

The vertical representation

The database is represented as a table, which gives for a transaction t, the items supported by t. $f(x, D)$ can be calculated by counting the number of rows which support x. This involved a loop over elements of D of size m. If we have m transactions and N frequent items, and we use bitmaps to encode itemsets, this representation can use only mN bits. With python we encoded bitmap with integers (2 bytes), and the table as a list. The size of this representation is $o(2m)$ bytes assuming that N is smaller than 64 and $o(mN/64)$ if N is bigger than 64.

The horizontal representation

The database is represented as a dictionary which gives for an itemset x, the transactions that support x. This dictionary is updated dynamically, every time we need to compute $f(x, D)$, we add an entry for $D[x]$ containing the transactions that support x. $f(x, D)$ can be calculated by counting the number of items in $D[x]$. To compute $D[x]$, we use that $D[uv] = D[u] \wedge D[v]$. If we have m transactions and N frequent items and we use bitmaps to encode itemsets, this representation can use only mN bits. With python, we encode this table as a list of list of integers. Each transaction is represented by a sequence of bitmaps and bitmaps are represented as int. The size of this representation is $o(Nm/64)$ bytes.

2.4 Parallelizing the algorithm

2.4.1 Splitting the database

The first possibility to parallelize the algorithm is to parallelize the computation of f, the function that counts the occurrence of an itemset in the database. This can be done by splitting the database, calling f on each portion of the database and summing the results. But if we create a task every time we must call f, we will create a huge number of tasks and each task creation is costly. Thus we should define a task as the computation of f for a set of itemsets X, and not only for one itemset x. To minimize the number of tasks we decided to define the set X as big as possible and take X as exactly the points in the border of the set of itemsets known as frequent. This correspond to define X as the set of candidates used in the apriori algorithm. In the following we provide the pseudo code for a task and for the general procedure.

Algorithm 5 the task $@F(X, D)$ that should be executed remotely

```
Require: $X$ a set of itemsets, $D$ a database
  return $\cup_{x \in X} f(x, D)$ =0
```

Algorithm 6 the parallel version of $F(X, D)$

```
Require: $X$ a set of itemsets, $D$ the database, $k$ the number of task to create
  $D_1, D_2, ..., D_k \leftarrow$ a partition of $D$
  for $i$ in $1, ..., k$ do
      $f_i^{x_1}, f_i^{x_2}, ..., f_i^{x_m} \leftarrow @F(X, D_i)$
  return $\Sigma_{i=1k} f_i^{x_1}, \Sigma_{i=1k} f_i^{x_2}, ..., \Sigma_{i=1k} f_i^{x_m}$
```

Algorithm 7 the parallel version of $FIM(D, S, k)$

```
Require: $X$ a set of itemsets, $D$ the database, $k$ the number of task to create
  $F \leftarrow \emptyset$
  $B \leftarrow \emptyset$
  $X \leftarrow$ the border of $F$
  for $f_x$ in $F(X, D, k)$ do
      if $f_x \geq S$ then
          add $x$ to $F$
      else
          add $x$ to $B$
  return $F$
```

2.4.2 Dividing the sets of candidates

Another solution is instead of dividing the database, dividing the candidates and process the set of candidates in parallel.

Algorithm 8 the parallel version of $F(X, D, k)$

```
Require: $X$ a set of itemsets, $D$ the database, $k$ the number of task to create
  $X_1, X_2, ..., X_k \leftarrow$ split $X$ into $k$ chunks of equal size
  for $i$ in $1, ..., k$ do
      $f_i \leftarrow @F(X_i, D)$
  return $\cup_{i=1}^{k} f_i$
```

An advantage of this solution is that we will not need to divide the database and sum the results. But a disadvantage is that we will have to send more data, and use more memory since we have to send the entire database to every node. Thus this solution seems relevant, for distributed resources that share memory.

2.4.3 Dividing the search space

We have seen that we can parallelize the solution, by parallelizing the computation of f. Another idea is to split the search space and solve the problem on each subspace of the search space. That is to say

- defining a partition of $P(I_n)$
- solving the FIM problem for each partition P and merging the results (i.e. finding the frequent itemsets of P)
- defining a task that solve the FIM problem on a subset of $P(I_n)$

The Algorithms 9 and 3 give the generic procedure to follow to solve the problem.

Algorithm 9 task $@FIM(D, S, P)$, it implements version of the FIM algorithm which working on a subset of the search space $P(I_n)$

```
Require: $X$ a set of itemsets, $D$ the database, $P$ a set of itemsets
  $F \leftarrow \emptyset$
  $B \leftarrow \emptyset$
  for $x$ in (the border of $F$) $\cap P$ do
      if $f(x, D) \geq S$ then
          add $x$ to $F$
      else
          add $x$ to $B$
  return $F$
```

2.4.4 Defining a partition of the search space

We need to define a good partition to explore. To solve this problem, we have supposed that the number of partitions k was a parameter, and decided to try to find a partition that minimizes the size of the biggest task. Finding a good partition is difficult since, the time of $FIM(D, S, P)$, the function that solve

Algorithm 10 the parallel version of $FIM(X, D, k)$

```
Require: $X$ a set of itemsets, $D$ the database, $k$ the number of tasks to create
  $P_1, P_2, ..., P_k \leftarrow$ a partition of $P(I_n)$
  for $i$ in $1, ..., k$ do
      $F_i \leftarrow @FIM(D, S, P_i)$
  return $\cup_{i=1}^{k} F_i$
```

the FIM problem for D and S, on the search space P, mainly depends on the size of $F \cap P$ (the number of frequent itemsets in P) but since we don't know F we can't know this time.

Given a minimal support value S, a database D containing only frequent itemsets, it seems reasonable to consider that FIM is an increasing function of the size of D. Indeed the complexity of FIM is $o(|S||D|)$ and it seems reasonable to consider that the size of S is an increasing function of the size of D. Considering two databases A and B, such that $|A| \geq |B|$ we know that the probability for F_A (the frequent itemsets in A) to be greater than F_B (the frequent itemsets in B) is greater than $\frac{1}{2}$.

2.4.5 Defining two tasks

A natural way to split the search space in two is to choose an item x and define two tasks

- mine frequent itemsets which contain x
- mine frequent itemsets which don't contain x

For mining frequent itemsets which contain x we can remove x from the database the itemsets and apply the FIM algorithm on transactions of the database which contains x. For mining frequent itemsets which don't h contain x we can remove x from the database. This way of partitioning $P(I_n)$ seems very interesting since if we mine frequent itemsets remotely, we don't need to send all the database to each task, thus we will have less transfer of data.

Thus the main work consist in

- constructing D_x by removing x from D and removing the transactions that don't contain x, this will results in a database of size $(|D|f_x - f_x)$ where f_x is the number of occurrences of x in D.
- constructing $D_{\neg x}$ by removing x from D, this will results in a database of size $|D| - f_x$. We decided to choose the most frequent item for x since x is removed from $D_{\neg x}$ and D_x thus more x appears in D smaller the size of D_x and $D_{\neg x}$ will be and choosing the most frequent item x enables us to obtain the smallest databases.

2.4.6 Defining a given number of tasks

The idea for defining several tasks was to apply the procedure in 2.4.5 recursively, i.e. choosing a task and splitting the task chosen in two. Since our goal is to minimize the size of the biggest task we always split the biggest task.

We can encode a way to split the database as a word w of $\{0,1\}^*$, such that the database D_w is defined by recursively as

- $D_\epsilon = D$
- for a word $w = au$, $a \in \{0,1\}$ and $u \in \{0,1\}^*$, D_w is obtained from D_u by
  - removing the most frequent item from the transactions
  - removing the transactions that don't contain the most frequent item if a is 1

The sizes of the resulting database d_{au} is given by the recurrence formula

- $d_{au} = d_u - f_u$ if a is 1
- $d_{au} = d_u f_u - f_u$ if a is 0

where f_u is the number of occurrences of the most frequent item of D_u in D_u. These results enabled us to write the following procedure to define a partition of D.

Algorithm 11 Split the database D to create k tasks that work on distinct portion of $P(I_n)$

```
Require: $D$ the database, $k$ the number of task to create
  return A partition of the database $D$
  if $k$ is 1 then : return $\{(\epsilon, |D|)\}$
  else
    $x \leftarrow$ the most frequent item of $D$
    $\Omega \leftarrow \{(1, x, D, |D|f_0 - f_0)), (0, x, D, |D| - f_0)\}$
    for $j$ in $\{0...k-2\}$ do
      $(w, v, D_w, d) = max_d\{(a, b, c, d)|(a, b, c, d) \in \Omega\}$
      $x \leftarrow$ the most frequent item of $D_w$
      $f_w \leftarrow$ the number of occurrence $x$ in the database $D_w$
      $\Omega \leftarrow \{(1w, xv, D_{1w}, df_w - f_w)), (0w, xv, D_{0w}, d - f_w)\} \cup \Omega$
  return $\Omega$
```

2.5 Experiments

We have implemented the algorithms defined previously and run them on several platforms that are described in the table below. The platforms include a simple computer, a supercalculateur MareNostrum4, and Grid5000.

For running our experiments we have used databases provided by the FIMI'03 workshop http://fimi.uantwerpen.be/fimi03/.

Algorithm 12 the parallel version of $FIM(X, D, k)$

```
Require: $X$ a set of itemsets, $D$ the database, $k$ the number of tasks to create
  $(D_1, u_1, w_1, \_), (D_2, u_2, w_2, \_), ..., (D_k, u_k, w_k, \_) \leftarrow split(D, k)$
  for $i$ in $1, ..., k$ do
    $F_i \leftarrow @FIM(D_i, S)$
    $F_i \leftarrow$ find frequent itemsets of $D$ using $F_i, u_i$ and $w_i$
  return $\cup_{i=1}^{k} F_i$
```

Table 1: Database Characteristics

Database	#Items	Avg Length	#Transactions
Chess	75	37	3,196

We have mined frequent itemsets using different values for the threshold S, different number of tasks k and different number of threads.

Because of lake of time we haven't been able to produce figures summarizing results obtained for the different platforms. Nevertheless, we have make measures with a simple machine equipped with 8 CPUs Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, with a a capacity of 3,4 GHz and 7844 MiB of memory. For the experiments, each algorithm was allocated a maximum of 200 seconds to finish execution, after which point it was killed. Table 1 summarizes the main characteristics of the chess database, the database that we used for figure 1, 2 and 3.

Figure 1 shows the results obtained for the sequential version of our algorithm using different representations of the database (vertical and horizontal) and different algorithms (eclat and apriori).

Figure 2 shows the results obtained for the distributed version of our algorithm using different representations of the database (vertical and horizontal) and different algorithms (eclat and apriori). For running these measures we have split the database in four, using the algorithm described in section 2.4.3.

Figure 3 compares the results for the best implementation (the best implementation uses the apriori algorithm with an vertical representation of the database) of the parallel and distributed version. The graph shows that the distributed version enables to speed the algorithm by a factor of 1.5 when we use 4 threads instead of a single thread.

2.6 Conclusion

In this work, we have worked on the FIM problem, provided a theoretical background to understand the problem and some of the known solutions (the eclat and apriori algorithm), then focused on identifying the right implementation choices to achieve good performance, and finally on parallelizing the solution. We identified three different ways of parallelizing the algorithms, and compared

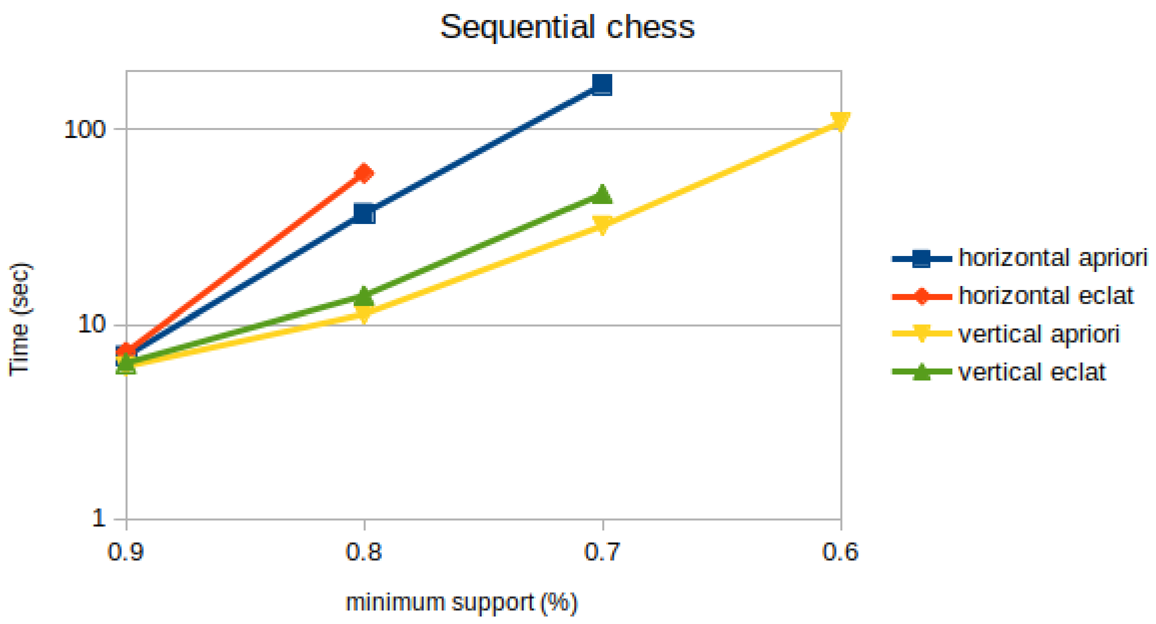

Figure 1: Comparative performance of the sequential implementations

the different approaches, highlighting the advantages and disadvantages. Finally, we experimented with our implementations and measured their performance. Although we have experimented with our algorithms on platforms providing large-scale resources such as Marenostrum4, a supercomputer located in the Centro Nacional de Supercomputación in Barcelona, or the Grid5000, a large-scale and flexible testbed for experimental research, due to time constraints we have only recorded part of the results obtained and we would like to carry out a complete and comparative study of the performance of our different implementations in the future.

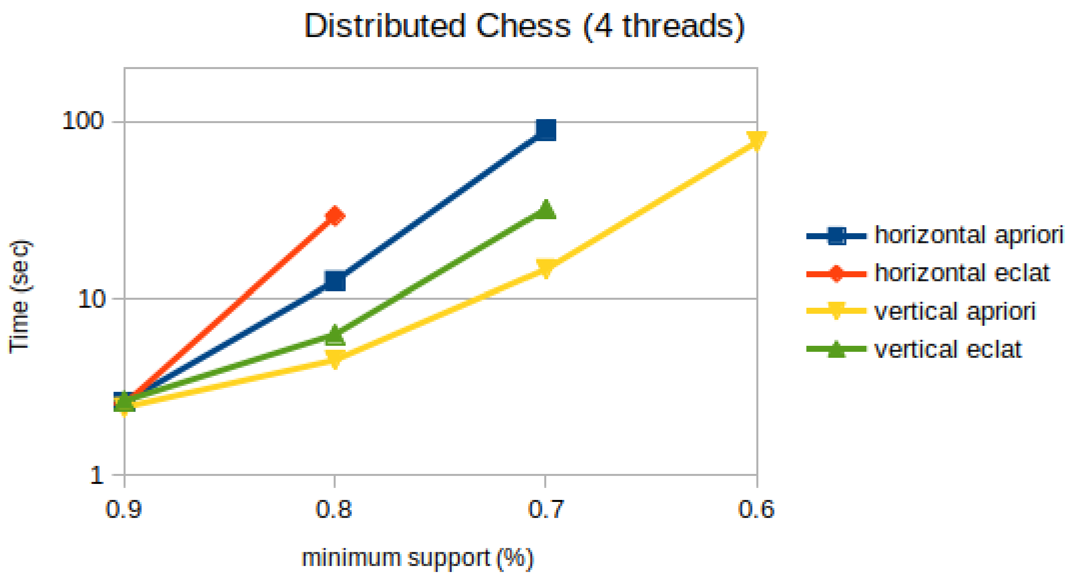

Figure 2: Comparative performance of the distributed implementations

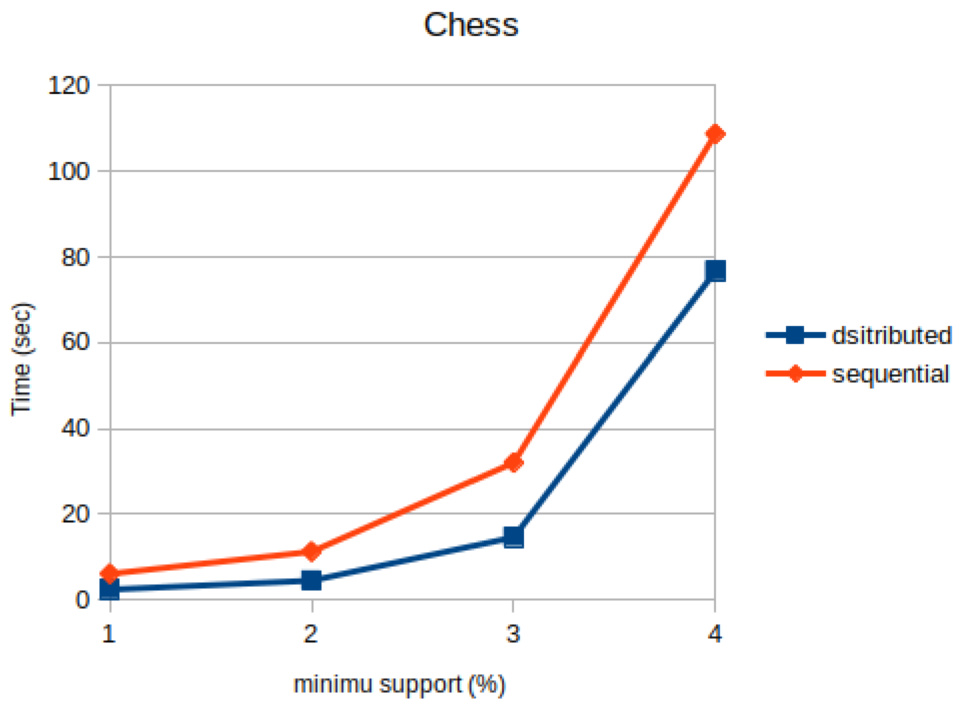

Figure 3: Comparative performance of the distributed and sequential best implementations