UNIVERSITY OF SALENTO
College of Engeneering
Degree in Computer Engineering

Parallel Algorithms

Implementation CHARM algorithm in MPI

PROFESSORS:
Massimo CAFARO
Italo EPICOCO

STUDENT:
Danilo Giovannico
Serial number n° 20039574

Academic year 2021 – 2022

INDEX

1 Sequential CHARM Algorithm

Before describing the CHARM algorithm, let's give some definitions that will be useful for understanding what Itemset mining is about. In many applications one is interested in how often two or more objects of interest co-occur. For example, consider a popular website, which logs all incoming traffic to its site in the form of weblogs. Weblogs typically record the source and destination pages requested by some user, as well as the time, return code whether the request was successful or not, and so on. Given such weblogs, one might be interested in finding if there are sets of web pages that many users tend to browse whenever they visit the website. Such "frequent" sets of web pages give clues to user browsing behavior and can be used for improving the browsing experience. The prototypical application is market basket analysis, that is, to mine the sets of items that are frequently bought together at a supermarket by analyzing the customer shopping carts (the so-called "market baskets"). Once we mine the frequent sets, they allow us to extract association rules among the item sets, where we make some statement about how likely are two sets of items to co-occur or to conditionally occur. For example, in the weblog scenario frequent sets allow us to extract rules like, "Users who visit the sets of pages main, laptops and rebates also visit the pages shopping-cart and checkout", indicating, perhaps, that the special rebate offer is resulting in more laptop sales. In the case of market baskets, we can find rules such as "Customers who buy milk and cereal also tend to buy bananas," which may prompt a grocery store to co-locate bananas in the cereal aisle.

Example

Suppose we want to know which items are bought together in an electronics store. An example of such a rule acquired from an electronics transactional database could be the following:

$$\text{buys}(X, \text{“computer”}) \Rightarrow \text{buys}(X, \text{“software”}) \text{ [support = 1\%,confidence = 50\%]},$$

where X is a variable representing the customer. The confidence (or certainty) value of 50% means that, if a customer buys a computer, there is a 50% chance that they will also buy the software. A support of 1% means that 1% of all transactions under analysis show that the computer and software are purchased together. This association rule involves a single attribute or predicate (e.g., purchases) that repeats. Association rules that contain a single predicate are called single-dimensional association rules.
Suppose, instead, that we are provided with a database related to purchases. A data mining system can find association rules such as:

$$\text{age}(X, \text{“20..29”}) \wedge \text{income}(X, \text{“40K..49K”}) \Rightarrow \text{buys}(X, \text{“laptop”}) \text{ [support = 2\%, confidence = 60\%]}.$$

The rule indicates that 2% ages 20-29 with income between $ 40,000 and $ 49,000 have purchased a computer in the electronics store. There is a 60% chance that a customer in that age and income range will purchase a laptop. Note that this association involves more than one attribute or predicate (ie age, income and purchases), this associative rule is called the multidimensional association rule.
Typically, association rules are discarded if they do not meet a minimun support threshold and a minimun confidence threshold.

With the term Frequent Patterns we mean patterns that occur frequently in the data. There are many types of frequent patterns, including frequent itemsets, frequent sub-sequences, and frequent substructures. A frequent itemset typically refers to a set of objects that often appear together in a transactional data set; a classic example would be milk and bread, which are often bought together in supermarkets by customers. A frequently occurring subsequence is called a (frequent) sequential pattern; a classic example would be where a customer first buys a laptop, followed by a digital camera and then a memory card. A substructure can

refer to several structural shapes (e.g. graphs, trees, or lattices) that can be combined with itemsets or subsequences. If a structure occurs frequently, it is called a (frequent) structured pattern. The extraction of frequent patterns leads to the discovery of interesting associations and correlations within the data.

Of course, in order to come up with frequent patterns, the data must be represented in some way, the types are different and you can see them here in the figure:

D	*A*	*B*	*C*	*D*	*E*
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

(a) Binary database

t	**i**(*t*)
1	*ABDE*
2	*BCE*
3	*ABDE*
4	*ABCE*
5	*ABCDE*
6	*BCD*

(b) Transaction database

x	*A*	*B*	*C*	*D*	*E*
t(*x*)	1	1	2	1	1
	3	2	4	3	2
	4	3	5	5	3
	5	4	6	6	4
		5			5
		6			

(c) Vertical database

As we can see, there is a binary relationship between what are the tids and the items, that is $D \subseteq T$ x I (we can say that a tid t ∈ T contains an item x ∈ I if (t,x) ∈ D), where $I = \{x_1, x_2, \ldots, x_m\}$ represents a set of elements called items and $T = \{t_1, t_2, \ldots, t_n\}$ represents another set of elements called tids (transaction identifiers). In particular, in the examples in the figure it is possible to see how the ABDE items are associated with tid 1 and so on for the rest, it is therefore possible to give the following additional definitions, where some of them will be useful in the analysis of the CHARM algorithm:

- a set X ⊆ I is called itemset (for example, a collection of products sold at the supermarket);
- a set T ⊆ T is called tidset (set of identifiers);
- a transaction is a tuple of the form <t,X>, where t ∈ T is a unique identifier of the transaction, and X is an itemset; for example the set of transactions T could denote the set of all the supermarket customers (by convention, we denote the transaction <t,X> like t).

We have also previously introduced a minimum support threshold useful for identifying Frequent Itemsets, allowing filtering on the association rules; more in detail we can give the following definitions:

- the support of an itemset X in a dataset D, denoted with sup(X,D), is the number of transactions in D that contain X

$$sup(X, \mathbf{D}) = \left|\{t \mid \langle t, \mathbf{i}(t)\rangle \in \mathbf{D} \text{ and } X \subseteq \mathbf{i}(t)\}\right| = |\mathbf{t}(X)|$$

The relative support of X is the fraction of the transactions that contain X

$$rsup(X, \mathbf{D}) = \frac{sup(X, \mathbf{D})}{|\mathbf{D}|}$$

It is an estimate of the joint probability of the elements that make up X.

- An itemset X is said to be frequent in D if $\sup(X,D) \geq$ minsup, where the minsup is a minimun support threshold. We use the set F to denote the set of all frequent itemsets, and $F^{(k)}$ to denote the set of all frequent k-itemsets.

From the definition of rule support and confidence, we can observe that to generate frequent and high confidence association rules, we need to first enumerate all the frequent itemsets along with their support values. Formally, given a binary database D and a user defined minimum support threshold minsup, the task of frequent itemset mining is to enumerate all itemsets that are frequent, i.e., those that have support at least minsup. Next, given the set of frequent itemsets F and a minimum confidence value minconf, the association rule mining task is to find all frequent and strong rules.
The search space for frequent itemsets is usually very large and grows exponentially with the number of items. In particular, a minimum value of the support value could result in an intractable number of frequent itemsets. An alternative approach is to determine condensed representations of frequent itemsets that summarize their essential characteristics. We therefore give the definition of frequent and maximal itemsets.

Given a binary database $D \subseteq T \times I$, on tids T and the items I, let F be the set of all frequent itemsets, that is:

$$\mathcal{F} = \{X \mid X \subseteq \mathcal{I} \text{ and } sup(X) \geq minsup\}$$

A frequent itemset $X \in$ F il called maximal if does not have frequent supersets. Let M be the set of all maximal frequent itemsets, expressed as

$$\mathcal{M} = \{X \mid X \in \mathcal{F} \text{ and } \nexists Y \supset X, \text{ such that } Y \in \mathcal{F}\}$$

The set M is a condensed representation of the set of all sets of frequent elements F, because we can determine if a set of elements X is frequent or not using M. If there exists a set of maximal Z elements such that $X \subseteq Z$, then X must be frequent; otherwise X cannot be frequent. On the other hand, we cannot determine $\sup(X)$ using only M, even if we can limit it to the minimum, that is $\sup(X) \geq \sup(Z)$ if $X \subseteq Z \in M$. Below we represent these concepts in an image:

Tid	Itemset
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

(a) Transaction database

sup	Itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

(b) Frequent itemsets ($minsup = 3$)

Given the definition of Frequent Itemsets and Maximal Itemsets, we therefore come to give the definition of Closed Frequent Itemsets that the CHARM algorithm will generate from its execution:

A frequent closed itemset is a frequent itemset that is not included in a proper superset having exactly the same support. The set of frequent closed itemsets is thus a subset of the set of frequent itemsets.

But to better understand what this means we must introduce what is called a closing operator:

- the closing operator c maps element sets into element sets and satisfies the following three properties:
  - Extensive: $X \subseteq c(X)$
  - Monotonic: Se $X_i \subseteq X_j$, then $c(X_i) \subseteq c(X_j)$
  - Idempotent: $c(c(X)) = c(X)$.

An itemset X is called closed if c(X) = X, that is, if X is a fixed point of the closure operator c. On the other hand, if $X \neq c(X)$, then X is not closed, but the set c(X) is called its closure. From the properties of the closure operator, both X and c(X) have the same tidset. It follows that a frequent set $X \in F$ is closed if it has no frequent superset with the same frequency because by definition, it is the largest itemset common to all the tids in the tidset t(X). The set of all closed frequent itemsets is thus defined as

$$\mathcal{C} = \{X \mid X \in \mathcal{F} \text{ and } \nexists Y \supset X \text{ such that } sup(X) = sup(Y)\}$$

In other words, *X* is closed if all supersets of *X* have strictly less support sup(*X*) > sup(Y), for all $Y \supset X$. The set of all closed frequent itemsets C is a condensed representation, since we can determine if a set *X* is frequent, as well as the exact support of *X* using C alone. The set of elements *X* is frequent if there is a closed frequent itemset $Z \in C$ such that $X \subseteq Z$. In addition, the support of *X* is defined as:

$$sup(X) = \max\{sup(Z) | Z \in \mathcal{C}, X \subseteq Z\}$$

The following relationship holds between the set of all sets of elements, closed and at maximum frequency:

$$\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$$

So let's talk about the functioning of the CHARM algorithm.
The task of mining association rules consists of two main steps. The first involves finding the set of all frequent itemsets. The second step involves testing and generating all high confidence rules among itemsets. In this document**,** we will present the CHARM algorithm**,** an efficient algorithm for extracting all frequent closed itemsets**,** showing that it is not necessary to mine all frequent itemsets in the first step but it is sufficient to mine the set of closed frequent itemsets, which is much smaller than the set of all frequent itemsets (for more details see **Bibliography 1** and **2**). Mining closed frequent itemsets requires that we perform closure checks, that is, whether X = c(X). Direct closure checking can be very expensive, as we would have to verify that X is the largest itemset common to all the tids in t(X), that is, $X = \bigcup_{t \in t(X)} i(t)$.

Instead, we will describe a vertical tidset intersection based method called CHARM that performs more efficient closure checking. Given a collection of IT-pairs $\{<X_i, t(X_i)>\}$, the following three properties hold:

- Property 1: If t(Xi) = t(Xj), then c(Xi) = c(Xj) = c(Xi ∪ Xj), which implies that we can replace every occurrence of Xi with Xi ∪ Xj and prune the branch under Xj because its closure is identical to the closure of Xi ∪ Xj.
- Property 2: If t(Xi) ⊂ t(Xj), then c(Xi) ≠ c(Xj) but c(Xi) = c(Xi ∪ Xj), which mean that we can replace every occurrence of Xi with Xi ∪ Xj, but we cannot prune Xj because it generates a different closure. Note that if t (Xi) ⊃ t(Xj) then we simply interchange the role of Xi and Xj.
- Property 3: If t(Xi) ≠ t(Xj), then c(Xi) ≠ c(Xj) ≠ c(Xi ∪ Xj). In this case we cannot remove either Xi or Xj, as each of them generates a different closure.

The CHARM algorithm is based on a similar algorithm called Eclat. Here the pseudocode of CHARM:

Algorithm 9.2: Algorithm CHARM

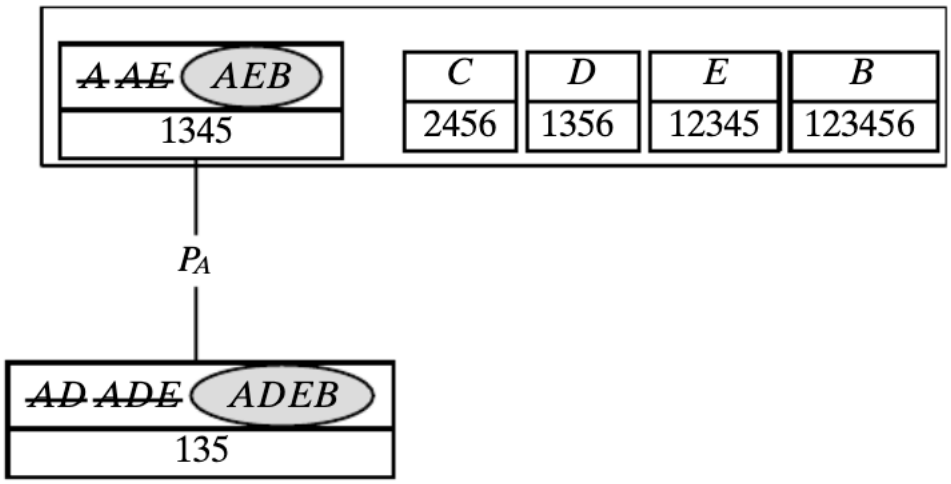
```
// Initial Call: C ← ∅, P ← {⟨i, t(i)⟩ : i ∈ I, sup(i) ≥ minsup}
CHARM (P, minsup, C):
1  Sort P in increasing order of support (i.e., by increasing |t(X_i)|)
2  foreach ⟨X_i, t(X_i)⟩ ∈ P do
3      P_i ← ∅
4      foreach ⟨X_j, t(X_j)⟩ ∈ P, with j > i do
5          X_ij = X_i ∪ X_j
6          t(X_ij) = t(X_i) ∩ t(X_j)
7          if sup(X_ij) ≥ minsup then
8              if t(X_i) = t(X_j) then // Property 1
9                  Replace X_i with X_ij in P and P_i
10                 Remove ⟨X_j, t(X_j)⟩ from P
11             else
12                 if t(X_i) ⊂ t(X_j) then // Property 2
13                     Replace X_i with X_ij in P and P_i
14                 else // Property 3
15                     P_i ← P_i ∪ {⟨X_ij, t(X_ij)⟩}
16     if P_i ≠ ∅ then CHARM (P_i, minsup, C)
17     if ∄Z ∈ C, such that X_i ⊆ Z and t(X_i) = t(Z) then
18         C = C ∪ X_i   // Add X_i to closed set
```

It takes as input the set of all frequent single items along with their tidsets. Also, initially the set of all closed itemsets, C, is empty. Given any IT-pair set P =$\{<X_i, t(X_i)>\}$, the method first sorts them in increasing order of support. For each itemset Xi we try to extend it with all other items Xj in the sorted order, and we apply the above three properties to prune branches where possible. First we make sure that Xij = Xi ∪ Xj is frequent, by checking the cardinality of t(Xij). If yes, then we check properties 1 and 2 (lines 8 and 12). Note that whenever we replace Xi with Xij = Xi ∪ Xj, we make sure to do so in the current set P, as well as the new set Pi. Only when property 3 holds do we add the new extension Xij to the set Pi (line 14). If the set Pi is not empty, then we make a recursive call to Charm. Finally, if Xi is not a subset of any closed set Z with the same support, we can safely add it to the set of closed itemsets, C (line 18).

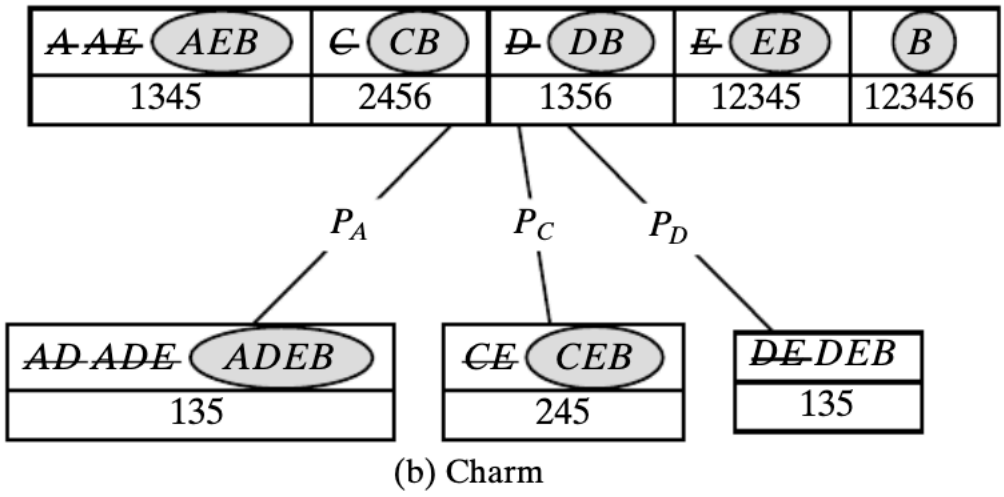
An application example of the algorithm is the following:

Tid	Itemset
1	*ABDE*
2	*BCE*
3	*ABDE*
4	*ABCE*
5	*ABCDE*
6	*BCD*

(a) Transaction database



(a) Process *A*



(b) Charm

We illustrate the Charm algorithm for mining frequent closed itemsets from the example database in Figure (a), using as threshold minsup = 3. The image above shows the sequence of steps. The initial set of IT-pairs, after support based sorting, is shown at the root of the search tree. The sorted order is A, C, D, E, and B. We first process extensions from A, as shown in Figure (a). Because AC is not frequent, it is pruned. AD is frequent and because t(A) ≠ t(D), we add <AD, 135> to the set P_A (property 3). When we combine A with E, property 2 applies, and we simply replace all occurrences of A in both P and P_A with AE, which is illustrated with the strike-through. Likewise, because t(A) ⊂ t(B) all current occurrences of A, actually AE,

in both P and P_A are replaced by AEB. The set P_A thus contains only one itemset {<ADEB, 135>}. When CHARM is invoked with P_A as the IT-pair, it jumps straight to line 18, and adds ADEB to the set of closed itemsets C. When the call returns, we check whether AEB can be added as a closed itemset. AEB is a subset of ADEB, but it does not have the same support, thus AEB is also added to C. At this point all closed itemsets containing A have been found.
The Charm algorithm proceeds with the remaining branches as shown in Figure (b). For instance, C is processed next. CD is infrequent and thus pruned. CE is frequent and it is added to P_C as a new extension (via property 3). Because $t(C) \subset t(B)$, all occurrences of C are replaced by CB, and P_C = {<CEB, 245>}. CEB and CB are both found to be closed. The computation proceeds in this manner until all closed frequent itemsets are enumerated. Note that when we get to DEB and perform the closure check, we find that it is a subset of ADEB and also has the same support; thus DEB is not closed.

2 Implementation Details

Some implementation details will be described below and how the CHARM algorithm differs from the pseudocode for implementation and performance reasons.

Data format

Since itemset-tidset pairs are manipulated and after noting that the fundamental operation is the intersection between two tidsets, the choice of the type of database to use fell on the vertical one. Specifically, the data reading will be performed using a file called itemset.txt, which will contain the data expressed in the following way:

itemset.txt

```
1-74 51 61 88 81 1 48 42 54 73 25 97 10 93 44 9 70 39 67 20 34 8 23 17 18
2-54 33 41 59 99 29 13
3-15 89
4-5 40 52 58 30 27 68 29 88 13 79 73 33 35 4 98 3 38 46 9 47 32 77 93
5-11 5 43
6-44 25 10 32 53 66 71 36 98 95 48 54 28
7-88 32 2 50 25
8-54 13 27 37 76 93 69 9 98 36 16 26 97 40 77 29 58 5
9-52 92 40 71 61 89 79 11 78 42 60 76 91 94 8 90 75 38 70 25
10-21 42 86 8 32 91 78 13 76 87 39 94 4 23 95 18 34 37 98
11-2 92 89 73 77 13 54 52 34 51 62 12 4 37 27 98 7 63 33 96 91 32 39
12-49 80 19 28 47 41 48 93 45 11 73 72 42 83 68 53 89 51 40 56 35 52
13-16 35 63 56 84 61 74 14 99 64 12 93 86 57 75 48 83 32 34 45 73
14-62 40 65 23 41 74 6 55 94 2 78 47 57 61 7 8 80
15-95 98 33 68 34 56 99 7 39 57 9 80 64 10 82
16-80 63 83 43 47 44 40 11 73 36 41 78 82 1 93 61 8
17-27 47 58 80 29 95 32 71 36 20 98 78 82 75 12 28 24 55 44 25 5 91
18-34 23 57 36 19 6 52 65 99 15 38 63 11 14 29 9 81 47 43 42 17
19-90 86 32 46 47 9 88 10 58
20-86 88 58 25 80 75 96 31 8 29 84 55 49 59
21-27 75 13 92 61 24 34 21 89 84 37 82 63 12 86
22-38 90 96 10 13 2 11 82
23-64 99 65 66 89 4 42
24-20 72 34 53 12 19 11 41 88 17 64 46 96 74 76 40 5 67 21 1 63 59
25-57 7 14 12 74 97 86 79 40 36 73 41 33 47
26-68 58 39 98 38 37 13 95 78 36
27-25 29 84
28-97 11 26 89 94 15 78 69 23 63 35 3 74 38 24 72 49 85
29-30 93 52 62 14 42 73 94 27 90 25 49 31 47 6
30-95 91 55 39 74
31-38 66 57 67 12 36 77 96 17 27 37 40 82 78 93 56 19 28 60 21 11 43 18
32-97 73 58 95 13 6 66 40 79 34 71 11 84 80 63 60 49 94 46 98 3 77 93
33-5 92 44 28 80 70 49 98 8 83 1 89 4 59 97 18 61
34-47 95 96 2 3 53 34 79 69 82 11 13 88 31 87 18 90 12
35-26 24 34 37 87 41 58 5 78 68 71 6 13 99 25 1 90 85 9 45 70 19 23
36-24 99 29 17 8 22 79 15 6 34 76 40 93 16 27 69 51 13 11 90 25 52 70 18 55 82 64 3 39
37-71 79 64 35 55 62 99 33 10 89 13 36 19 46 69 52 24 54 14 74
38-35 66 51 33 67 48 77 7 42 83 41 38 47 5 13 25 2 90 97 53 65 89 52
39-80 35 96 42 60 41 72 48 58 32 63 46 84 54 13 55 52 31 76 40 24 1 65 4 12 90 61 74
40-68 26 14 89
```

Where the first column represents the id of the item and the subsequent values, after the delimiter -, correspond to the id of the tids; both ids will start from id 1 and go up. The choice to use integers

instead of characters is due to the fact that the characters follow the letters of the alphabet (max 26, 52 if considered case-sensitive), and this strongly limits the number of possible items, for this reason they will be used integers that allow a wide management of the number of items to be analyzed.

Generation Candidates
This step allows you to generate all the subsets of I, called candidates, of course, each itemset is potentially a candidate frequent pattern. The search space of candidate itemsets is exponential since there are $2^{|I|}$ potential frequent itemsets, which would be generated in the worst case corresponding to the case in which the intersection of tids t(Xi) and t(Xj) falls into Property 3 corresponding to line 15 of the pseudocode.

Observation
Another important observation falls on the structure of the itemset search space: the set of all itemsets constitutes a lattice where two itemsets, e.g. X and Y are connected if X is an immediate subset of Y, that is, $X \subseteq Y$ and $| X | = | Y | -1$. In terms of a practical search strategy, the itemsets in the lattice can be enumerated using either a breadth-first search (BFS = breadth-first search) or a depth-first search (DFS = depth first search) on the prefix tree, where two itemsets X, Y are connected if X is an immediate subset and prefix of Y. This allows itemsets to be enumerated starting with an empty set, then adding one item at a time.

Intersections and Subset Testing
Given the availability of vertical tidsets for each itemset, the calculation of the intersection of the tidsets to generate a new combination is simple. It corresponds to having a simple linear scan on the two tidsets and the matches obtained are stored in a new tidset. For example, we have t(A) = 1345 and t(D) = 2456, then $t(AD) = 1345 \cap 2456 = 45$.
This operation might seem expensive but in reality the vertical database offers the following advantage: when two tidsets intersect, the discrepancies of both the tids lists t(Xi) and t(Xj) are tracked; denote by m(Xi) and m(Xj) the number of discrepancies of the aforementioned lists, we will have three cases to consider:

- $m(Xi) = 0$ and $m(Xj) = 0$, then $t(Xi) = t(Xj)$ - Property 1
- $m(Xi) = 0$ and $m(Xj) \neq 0$, then $t(Xi) \subset t(Xj)$ - Property 2
- $m(Xi) \neq 0$ and $m(Xj) \neq 0$, then $t(Xi) \neq t(Xj)$ - Property 3

N.B. There would also be the case $m(Xi) \neq 0$ and $m(Xj) = 0 \Longrightarrow t(Xi) \supset t(Xj)$, but in this version of CHARM it is not treated due to the sorting in incremental order of support present at line 1 visible in our pseudocode; this limits us to consider only the case $t(Xi) \subset t(Xj)$.

Eliminating Non-Closed Itemsets
In line 17-18 of our pseudocode, the inclusion of closed frequent itemset in set C appears; such lines of code could lead to having a number of CFI higher than the real number, this is because possible subsets could occur (in the example above it is possible to verify that DEB is a subset of ADEB). Clearly it is preferable to avoid a check on all the elements present in C, since there would be a complexity equal to $O(|C|^2)$. To overcome this problem, C could be saved in a hash table where the hash function could be applied to the sum of the tids (it will be seen later that this procedure leads to collisions, it was therefore decided to adopt another variant using the tids), this would coincide with having a simple linear scan, greatly reducing execution times. The following is the code relating to the hash function used by us:

```c
int hash_function(struct itemset_tids itemset) {
    int support = itemset.tids[0];
    char str[ELEMENTS_SIZE];
    int index = 0;
    for(int i=0; i<=support; i++) {
        index += sprintf(&str[index], "%d", itemset.tids[i]);
    }
    unsigned hashval = 5381;
    int k = 0;
    while (str[k] != '\0') {
        //hashval = str[k] + 33*hashval;
        hashval = ((hashval << 5) + hashval) + str[k];
        k++;
    }
    return hashval % FCI_SIZE_ARRAY;
}
```

The idea was to concatenate the tids by building a string and then applying the hash function djb2 written by Daniel J. Bernstein in 1991. This hash function starts by setting the hashval variable to the value 5381, then iterates over the character string to execute the following operations for each character:

1. We multiply the hash variable by the value 33;
2. We add the ASCII value of the character in question to hashval.

After finishing the for loop, the module of hashval is returned divided by the size of the array of elements of C. The multiplication of the hashval could be performed normally hashval * 33; however we used (hashval << 5) + hashval bit shifts whose operation is often more efficient on many CPUs.
hash << 5 allows you to shift 5 bits to the left by 5 positions, multiplying the number by 32 (2 ^ 5) and + hashval adds another hashval value, transforming it into a multiplication by 33.
The starting number 5381 and the value 33 were chosen by DJB because having performed numerous tests, he was able to demonstrate that the number of collisions generated is very limited (in the datasets we used for the simulations the collisions were equal to 0).
N.B. Our code has a slight modification in the pseudocode: in line 17 the verification check that allows the addition of the FCI has been removed, this operation will be performed only by the Master process (the idea of parallelization will be described later) after having obtained all FI (Even with subsets present) generated by the processes. From the tests performed we have noticed that the performances are better, the only limit defined by the following hash function is the memory allocation for set C; the indexes could also be very large, this causes a greater memory allocation to minimize the number of collisions.

3 Correctness and Efficiency

Correctness

The CHARM algorithm correctly identifies all and only frequent closed itemsets, since its search is based on a complete search of the network of subsets. The only branches pruned are those that do not have sufficient support or those that cause non-closure based on the properties of the itemset-tidset pairs seen previously. Finally, CHARM eliminates any subsets from set C thanks to a preventive check.

Computational cost

The CHARM algorithm has a **Running Time equal to O(t·$2^{|I|}$)**, where t represents the average length of the tidset. Starting from the single items and associated tidsets, while we are processing a branch (as we have already seen from the properties expressed) the following cases may occur:

- We execute a prune of the branch if t(Xi) = t(Xj) (see Property 1);
- We execute a prune of Xj if t(Xi) ⊃ t(Xj) (see Property 2);
- We do not execute any prune if t(Xi) ≠ t(Xj) and we generate a new node (see Properties 3).

We can see that each new node generated indirectly represents a closed itemset, since there is a closed itemset for each closed tidset. Therefore CHARM works in the order of O($2^{|I|}$) intersections, the only exceptions are those relating to any subsets (see line 17-18). If each tidset has an average length t, an intersection will cost at most 2·t. Therefore the running time of CHARM is 2·t·$2^{|I|}$ or O(t·$2^{|I|}$).

Cost I/O

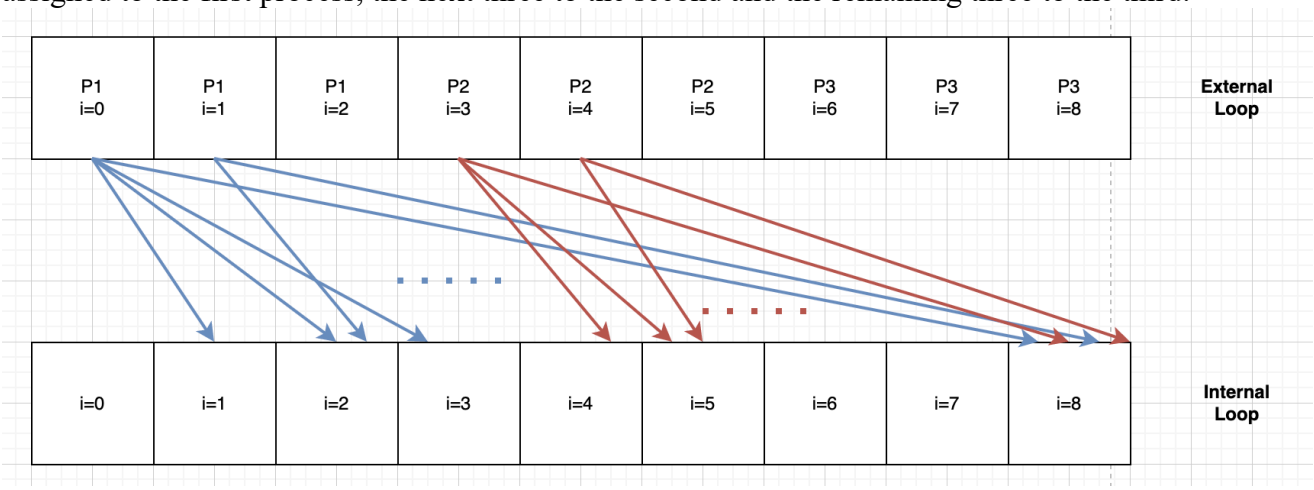
The number of database scans performed by CHARM is given by $O\left(\frac{2^{|I|}}{|I|}\right)$, where $2^{|I|}$ is the set of all closed frequent itemsets and I is the set of items. The number of scans that the database requires is given by the total memory consumption of the algorithm divided by the fraction of the database saved in memory. **Since CHARM works in the order of O($2^{|I|}$) intersections, the total memory required by CHARM is O(t·$2^{|I|}$), where t is the average length of the tidset**. We note that as we intersect the size of the tidsets of longer element sets shrinks rapidly, but we ignore those effects in our analysis (as it is a pessimistic limit).

4 Parallelized CHARM algorithm with MPI

The aim of the project is to transform the sequential CHARM algorithm into a parallel version thanks to the use of the MPI library. For this purpose, we simulated the classic Market basket analysis model used by retailers to increase sales by better understanding customers' purchasing patterns. The dataset was created randomly and is composed of 10000 products (items) with an incremental id from 1 to 10000 (e.g. id = 1 for bread and so on) and for each product a random number of cart/sales ids (tids) variable on 10000 carts/sales (e.g. product 3 appears in carts 15 and 89).

Block decomposition parallelization approach

Initially the idea was to use a block decomposition approach. The data was divided into blocks by process, a practical example is shown in the figure where the first three elements of the array are assigned to the first process, the next three to the second and the remaining three to the third.



The performances were very bad given that the first process found itself carrying out a number of comparisons on the whole cycle much higher than the subsequent processes, the same thing for the

second process compared to its subsequent ones and so on. To fully understand this statement, let's analyze the process P1 in the figure, for the elements of the external for there will be the following situation:

- for the element of the array with index i = 0 we will have n-1 comparisons on the elements of the internal for,
- for the element of the array with index i = 1 we will have n-2 comparisons on the elements of the internal for,
- for the element of the array with index i = 2 we will have n-3 comparisons on the elements of the internal for.

For process P2 we will have:

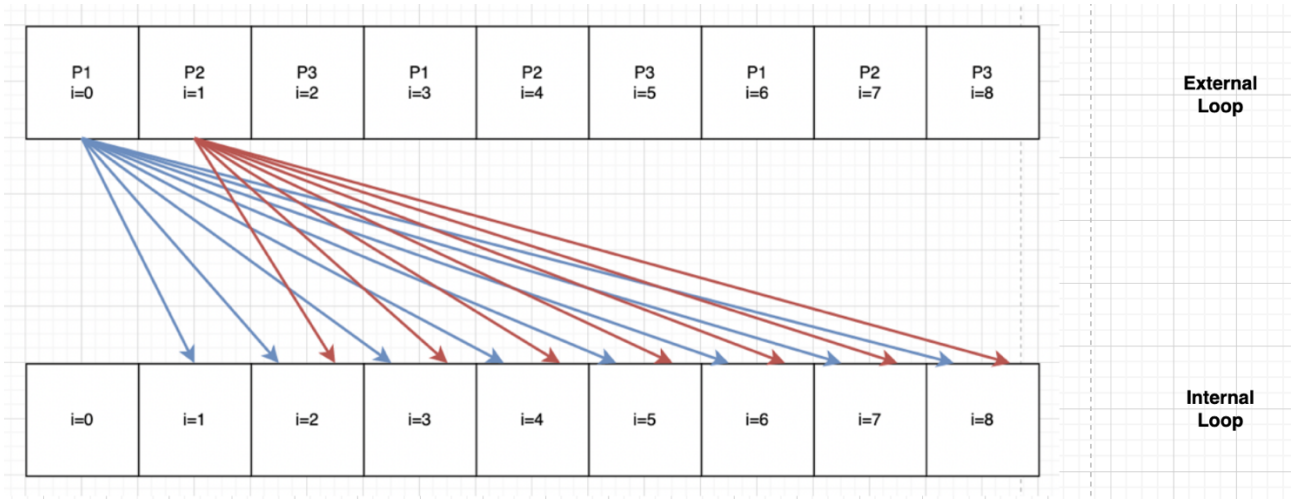
- for the element of the array with index i = 3 we will have n-4 comparisons on the elements of the internal for,
- for the element of the array with index i = 4 we will have n-5 comparisons on the elements of the internal for,
- for the element of the array with index i = 5 we will have n-6 comparisons on the elements of the internal for.

And so on for the third process.
It is therefore possible to understand the imbalance on the assignment of data per process, to remedy this situation a Round Robin distribution approach has been used which allows a data balancing almost equal per process.

Round Robin Scheduling parallelization approach
The idea behind code parallelization is to distribute the workload among the processors in the most balanced way; it was therefore decided to use an assignment of the indices relative to the blocks of the array, relative to the external for loop of the pseudocode (line 2), alternately and repeatedly. This will involve a distributed balancing of the unions on the itemsets (line 5) and of the intersections on the tidset (line 6), and therefore on the number of comparisons, due to the two nested for loops according to the logic that follows CHARM.

A visual example of what happens is the following:

> Suppose we have an array consisting of 9 blocks and 3 processors, the subdivision of the indices on the external for loop will take place as follows in the image. This will involve comparing the element with index 0 with the subsequent n-1 elements (the same as both for iterate on the same array); the element with index 1 will perform the comparison with the remaining n-2 elements and so on. In this way each processor will work approximately equally; the number of possible candidates generated by each processor will also be balanced in most cases.



Description of the code and operation of the parallel algorithm
Here we will describe in more detail how the parallel version of the algorithm was implemented.

The algorithm starts from the reading of the data inserted in a .txt file by the process with rank zero. The algorithm starts from the reading of the data inserted in a .txt file by the process with rank zero. This process will then take care of the execution of a broadcast of the complete dataset towards all the other processes:

```
//Function for broadcasting from master to workers.
void bcast_dataset(struct itemset_tids *P, char* namefile, MPI_Datatype item_type, int rank, int processes, int num_el_P) {
    //If the process has rank 0 (master) I read the data.
    if(rank == 0) {
        if ((processes - 1) > num_el_P) {
            fprintf(stderr, "There are a number of process greater than the number of data present.\n");
            exit(0);
        }
        read_data(P, namefile, num_el_P);
        //I sort the data in order of support.
        sort_by_order_support(P, num_el_P);
    }

    //Broadcasts of the data read and inserted in the array from the process with rank "0" (master) to all other processes (workers)
    //of the communicator MPI_COMM_WORLD.
    int res = MPI_Bcast(P, num_el_P, item_type, 0, MPI_COMM_WORLD);
    if (res != MPI_SUCCESS) {
        fprintf(stderr, "MPI_Bcast failed\n");
        exit(0);
    }
}
```

The transmitted data array will contain structs composed as follows:

```
/************************ STRUCT ************************/
struct itemset_tids {
    int itemset[ITEMSETS_SIZE];
    int tids[TIDS_SIZE];
    int index;
};
```

Where itemset corresponds to the array containing the ids of the items present in the set, tids corresponds to the array of tidsets containing the ids of the tids and index will contain the index calculated by the hash function.

After the broadcast each process will launch the charm function, which at the first non-recursive launch will proceed to the automatic assignment of the indices on the external for thanks to a check on the integer i and a flag called flag_starting_index that allows you to define when the function is launched the first time,

setting it to 1, and 0 when the recursion is performed instead. The same thing happens for the sort by order of support (line 1 pseudocode) since during the reading phase it will be the process with rank 0 to execute it for the first time.

```
//CHARM algorithm
int charm(struct itemset_tids *P_ptr, int minsup, struct itemset_tids *C, int flag_starting_sort, int flag_starting_index, int num_el_P, int start_index) {
    int *Xij = (int*) malloc(ITEMSETS_SIZE*sizeof(int));
    int *tij = (int*) malloc(TIDSETS_SIZE*sizeof(int));
    struct itemset_tids *Pi = malloc(num_el_P*sizeof(struct itemset_tids));
    //Check the sort in support order. The first time workers CHARM this check is not performed.
    if(flag_starting_sort == TRUE) {
        sort_by_order_support(P_ptr, num_el_P);
    }
    int i = start_index;
    //Cycle to the final index identified by the master process. This for loop is the parallelized one.
    while(i < num_el_P) {
        int j = i+1, num_el_Pi = 0;
        //Internal for loop on all elements following the one in use by the external for.
        while(j < num_el_P) {
            //I create the union set of the itemset relative to the external and internal for using the union_itemset method.
            //This union does not have duplicate items.
            union_itemset(P_ptr[i].itemsets, P_ptr[j].itemsets, Xij);
            //I create the intersection set of the relative tids relative to the itemset of the external and internal for.
            int dis_i = 0, dis_j = 0, c_i=1, c_j=1, common_element_counter = 1, c1, c2, val1, val2;
            while(P_ptr[i].tidsets[0] >= c_i && P_ptr[j].tidsets[0] >= c_j){
                c1 = P_ptr[i].tidsets[0];
                c2 = P_ptr[j].tidsets[0];
                val1 = P_ptr[i].tidsets[c_i];
                val2 = P_ptr[j].tidsets[c_j];
                if (P_ptr[i].tidsets[c_i] < P_ptr[j].tidsets[c_j]) {
                    c_i++;
                    dis_i++;
                }else if(P_ptr[j].tidsets[c_j] < P_ptr[i].tidsets[c_i]){
                    c_j++;
                    dis_j++;
                } else {
                    tij[common_element_counter] = P_ptr[i].tidsets[c_i];
                    c_i++;
                    c_j++;
                    common_element_counter++;
                }
                if(c_j > P_ptr[j].tidsets[0] && P_ptr[i].tidsets[0] >= c_i) {
                    dis_i++;
                }
                if(c_i > P_ptr[i].tidsets[0] && P_ptr[j].tidsets[0] >= c_j) {
                    dis_j++;
                }
            }
            tij[0] = --common_element_counter;
```

```
            //I enter the if only if the number of tids of the intersection set is greater than MIN_SUPPORT. The total number of tids is stored in index 0 for easier code management.
            if(tij[0] >= minsup) {
                //Property 1
                //I check if the tids relative to the element under analysis of the internal and external for are equal. Use the methods tids_are_equals
                if(dis_i == 0 && dis_j == 0) {
                    //I perform the substitution on P and Pi and finally the removal on P.
                    replace_P(i, P_ptr, Xij);
                    replace_Pi(Pi, Xij, num_el_Pi);
                    remove_itemsets(j, P_ptr, P_ptr[j]);
                    j--;
                } else {
                    //Property 2
                    //I check if the tids relative to the element under analysis of the external for are a subset of the internal one. Use the methods tids_are_subset.
                    if(dis_i == 0 && dis_j > 0) {
                        //I perform the substitution on P and Pi.
                        replace_P(i, P_ptr, Xij);
                        replace_Pi(Pi, Xij, num_el_Pi);
                    } else { //Property 3
                        //I insert the element into the Pi array.
                        memcpy(Pi[num_el_Pi].itemsets, Xij, ITEMSETS_SIZE * sizeof(int));
                        memcpy(Pi[num_el_Pi].tidsets, tij, TIDSETS_SIZE * sizeof(int));
                        num_el_Pi++;
                    }
                }
            }
            j++;
        }

        //If I have elements in Pi I run CHARM recursively.
        if(num_el_Pi > 0)
            charm(Pi, minsup, C, 1, 1, num_el_Pi, 0);
        //I check if the possible frequent closed itemset generated are a subset of some element present in the array containing the FCIs,
        //we also carry out the verification on the support.
        if(P_ptr[i].tidsets[0] >= minsup) {
            memcpy(C[counter_cfi].itemsets, P_ptr[i].itemsets, ITEMSETS_SIZE * sizeof(int));
            memcpy(C[counter_cfi].tidsets, P_ptr[i].tidsets, TIDSETS_SIZE * sizeof(int));
            C[counter_cfi].index = hash_function(C[counter_cfi]);
            counter_cfi++;
        }
        if(flag_starting_index == TRUE) {
            i++;
        } else {
            i=i+processes;
        }
    }
    free(Xij);
    free(tij);
    free(Pi);
    return counter_cfi;
}
```

At the end of CHARM each process will have generated its possible candidates which will have to be transmitted via Gather to the process with rank 0 (which could also be called the master process). To do this, an MPI_Gather is performed on the number of elements generated by each single process at process 0, which will then perform the sum so as to be able to allocate the right amount of memory for receiving the array of structs through an MPI_Gatherv:

```
int num_CI_send = charm(P, MIN_SUP, closedItemset, 0, 0, NUM_EL_P, rank);
int *num_CI_recv = malloc(processes*sizeof(int));
MPI_Gather(&num_CI_send, 1, MPI_INT, num_CI_recv, 1, MPI_INT, 0, MPI_COMM_WORLD);
int tot_CI_el = 0;
int *displ = malloc(processes*sizeof(int));
if(rank==0) {
    for (int i = 0; i < processes; i++)
    {
        displ[i] = tot_CI_el;
        tot_CI_el = tot_CI_el + num_CI_recv[i];
    }
}
struct itemset_tids *CI_temp = calloc(tot_CI_el, sizeof(struct itemset_tids));
MPI_Gatherv(closedItemset, num_CI_send, item_type, CI_temp, num_CI_recv, displ, item_type, 0, MPI_COMM_WORLD);
```

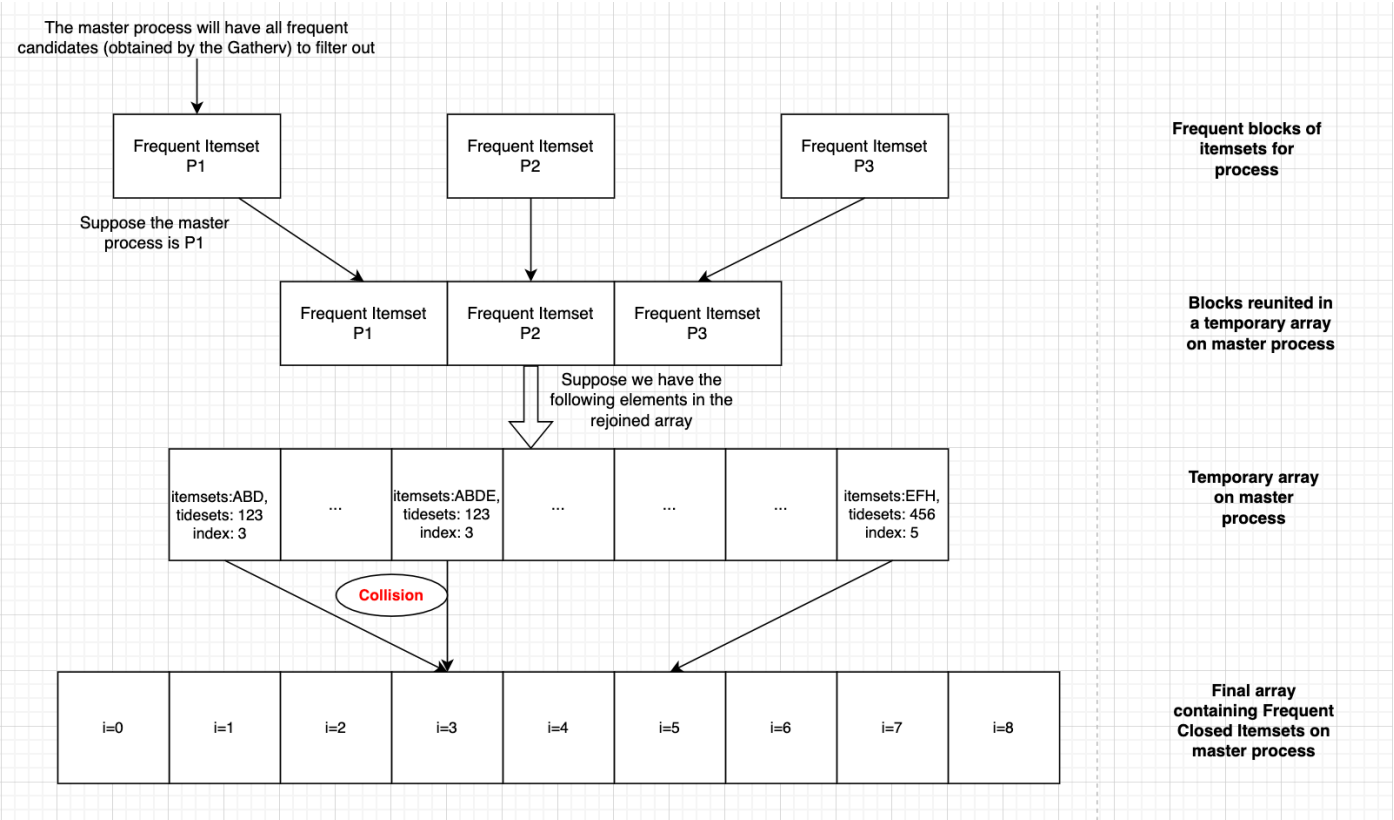
Finally, the process with rank 0 will try to filter the complete array of candidates by removing any subsets (function get_frequent_closed_itemset). It was preferred to remove the control present on line 17 of the pseudocode, since keeping it on each process would cause an increase in the execution time of the parallel algorithm, it was therefore preferred to limit linear scanning to only process 0. The certainty that only the

subsets and not the supersets are removed is given by a check on the number of items present in the itemset set; this implies that on elements of the same index, the one that will be kept in the filtered array will be the one with a greater number of items.

```
for (int i = 0; i < num_el_CI; i++) {
    int index = closedItemsetProcess[i].index;
    if(frequentClosedItemset[index].index == 0) {
        memcpy(frequentClosedItemset[index].itemsets, closedItemsetProcess[i].itemsets, ITEMSETS_SIZE * sizeof(int));
        memcpy(frequentClosedItemset[index].tidsets, closedItemsetProcess[i].tidsets, TIDSETS_SIZE * sizeof(int));
        frequentClosedItemset[index].index = index;
        counter++;
    } else {
        if(frequentClosedItemset[index].itemsets[0] < closedItemsetProcess[i].itemsets[0]) {
            memcpy(frequentClosedItemset[index].itemsets, closedItemsetProcess[i].itemsets, ITEMSETS_SIZE * sizeof(int));
        }
    }
}
```

The final results will be contained in an array of structs called frequentClosedItemset; it is also possible to print the results in a file called frequent_closed_itemsets.txt by setting the variable FLAG_WRITE_FCI equal to TRUE in the common.h file. Further flags that can be set are the flag for sorting the ids in the tidsets set (FLAG_SORT_TIDS) and the one relating to the itemsets (FLAG_SORT_ITEMSETS).

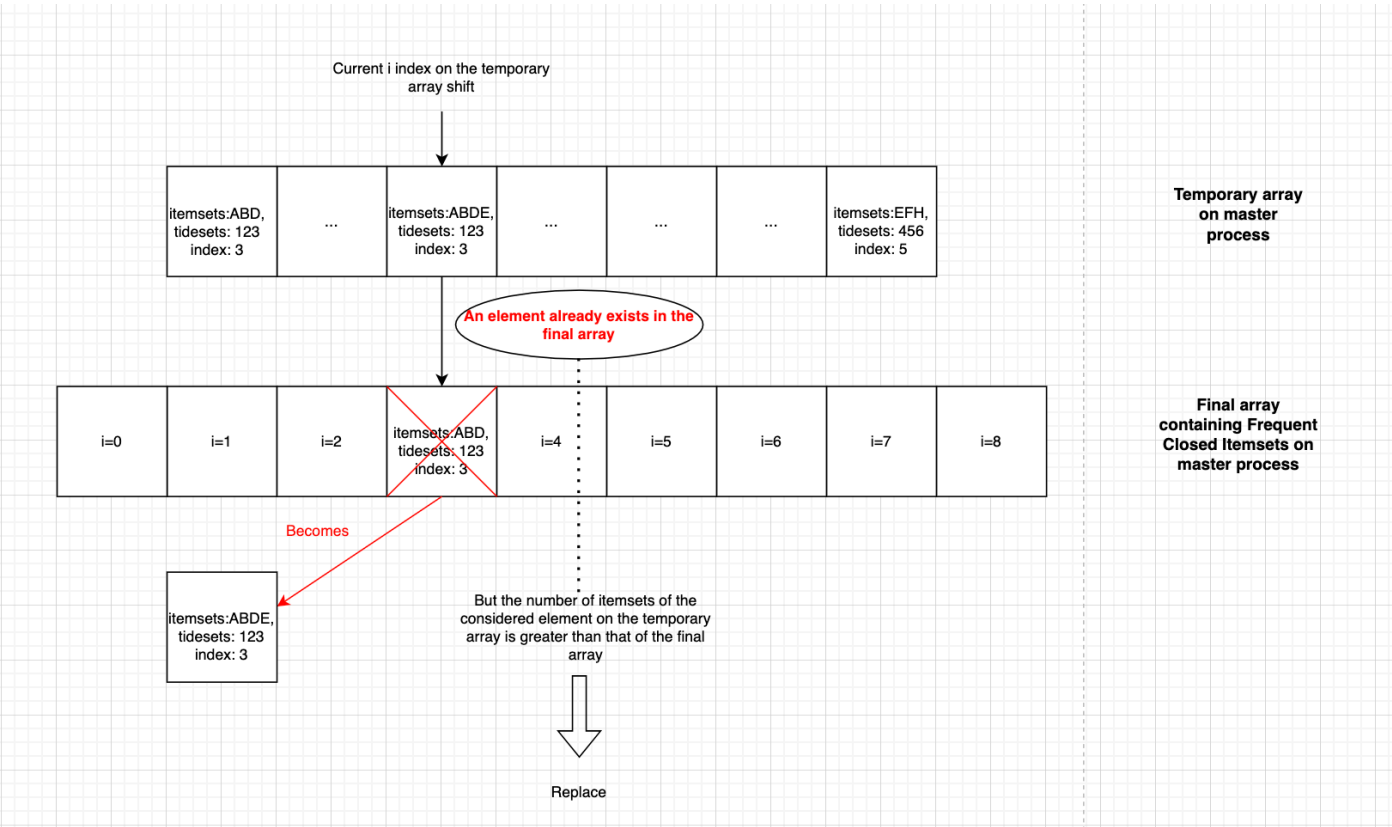
To better understand this filtering phase, we provide the following graphic example:



In this example we can see how in the temporary array (which contains the different rejoined blocks from each process) there are two elements with index 3 (calculated using the hash function). What happens on the master process is the following logic:

- iterates over all elements of the temporary array

- for each element, it takes the index and checks if there is already a Frequent Itemsets in the bucket with index i = index of the final array
    - or if there is no such element it is added (case itemsets: EFH, tidsets: 456, index = 5)
    - if present, a check is made between the number of itemsets present in the final array at the index element i = index and the one taken into consideration in the iterative form of the temporary array
        - if the element of the temporary array has a number of items higher than that of the final filtered array, the itemsets are replaced
        - otherwise the final array remains unchanged.



Remarks

On line 9, 10 and 13 of the pseudocode there are respectively the functions for replace and remove: in the parallel version of the algorithm we could think of applying communications between processes in order to perform a synchronization on the pruning of the lattice for the removal and replacement of the 'set itemsets on a corresponding element of the array P at a certain index i; but can it actually be applied?
Let's analyze the following practical cases:

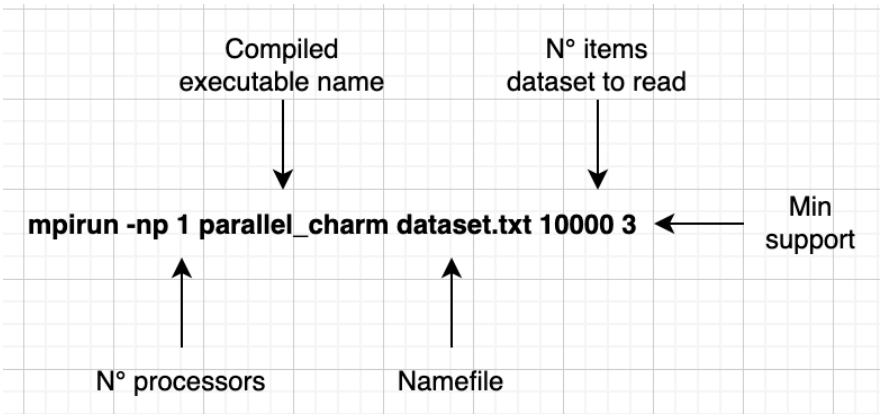
- P1 removes the element i = 2 from the array P, but P2 has exceeded i = 2 and P3 has already taken it in analysis since it is its first => it is not possible to take advantage of the number of comparisons which would remain unchanged anyway;
- P1 performs a replace on i = 3 and communicates it to P1 and P2 (which in any case could have exceeded this index) who perform it => the number of comparisons would not vary.

Furthermore, this logic could make sense only at the first launch of CHARM but not on the recursion performed on the set Pi of generated elements, which will turn out to be different for each process given the distribution of the array of elements. We therefore conclude that the only communication that could make sense would be the remove, but it is not advisable to manage its synchrony as each process would find itself

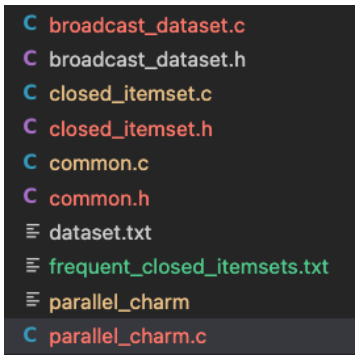
waiting for the others after each internal for loop, increasing the overhead due to communications and synchrony.

Description of the implemented functions and MPI functions used
To compile and launch the application, follow the instructions below:

- Compiles and links MPI programs written in C: **mpicc -O3 parallel_charm.c common.c broadcast_dataset.c closed_itemset.c -o parallel_charm**
- Launch of the application:



The project consists of the following files.



whose usefulness and composition is as follows:

- **parallel_charm.c**: file containing the main launch method and in which the code relating to the CHARM algorithm and the hash function is implemented:
  - **int hash_function (struct itemset_tids itemset),**
  - **int charm (struct itemset_tids *P_ptr, int minsup, struct itemset_tids *C, int flag_starting_sort, int flag_starting_index, int num_el_P, int start_index).**
- **broadcast_dataset.c**: file containing the methods for reading data from file and broadcasting the dataset:
  - **void read_data (struct itemset_tids *P, char *namefile, int num_el_P),**
  - **void bcast_dataset (struct itemset_tids *P, char *namefile, MPI_Datatype item_type, int rank, int processes, int num_el_P).**

- **common.c**: file containing the common methods used by CHARM including the method for sorting in support order, the method for sorting tids and items by incremental id, the method for removing an itemsets from the array, method for the union of itemsets, methods for replacing an itemset on P and Pi:
    - **void sort_by_order_support (struct itemset_tids *P_ptr, int num_el),**
    - **void sort_tids_itemsets (int *tids, int n),**
    - **void remove_itemsets (int index, struct itemset_tids *P_ptr, struct itemset_tids Xj),**
    - **void union_itemset (const int *Xi, const int *Xj, int *Xij),**
    - **void replace_P (int index, struct itemset_tids *P_ptr, int *Xij),**
    - **void replace_Pi (struct itemset_tids *P_ptr, int *Xij, int n).**
- **closed_itemset.c**: contains the method for printing the FCIs in the file and for removing any subsets on the FCIs generated by the different processes:
    - **int get_frequent_closed_itemsets (struct itemset_tids *frequentClosedItemset, struct itemset_tids *closedItemsetProcess, int num_el_CI).**
- **dataset.txt**: file containing the dataset.
- **frequent_closed_itemsets.txt**: file containing the FCIs generated.

To design this algorithm and make it parallel through the use of MPI, we made use of some useful functions in the library:

- *Execution Environment*: One of MPI's goals is to achieve source code portability. By this we mean that a program written using MPI is portable, requiring no changes to the source code when migrated from one system to another.
    - **MPI_Init**: Initialize the MPI execution environment.
        - **int MPI_Init(int *argc, char ***argv)**
            - **argc:** Pointer to the number of arguments
            - **argv:** Pointer to the argument vector
    - **MPI_Finalize**: terminates the MPI execution environment.
        - **int MPI_Finalize (void)**
    - **MPI_Wtime:** is a function that returns the walltime elapsed from an arbitrary time in the past.
        - **double MPI_Wtime (void)**
- *Communication and Communicator*: When a program is running with MPI, all processes are grouped into what we call a communicator. A communicator can be seen as a box that groups processes together, allowing them to communicate. Each communication is connected to a communicator, allowing the exchange of messages between different processes. Communications can be Point-to-Point or Collective, respectively if the exchange of messages occurs between two distinct processes or between all those present. The default communicator is called MPI_COMM_WORLD; it allows you to group all processes when the program is started. Collective communications are a communication method that involves all processes in a communicator operating in secure mode. Communications involving a group of processes are of three types: Synchronization (e.g., Barrier), Data Movement (e.g., Broadcast or Gather / Scatter) and Global Computation (e.g., Reductions).
    - **MPI_Comm_rank**: This function gets the rank of the calling MPI process in the communicator provided.

- **int MPI_Comm_rank(MPI_Comm comm, int *rank)**
  - **comm:** communicator (handle)
  - **rank**: A pointer on the variable in which write the rank of the calling MPI process in the MPI communicator given.

- **MPI_Comm_size**: this function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available.
  - **int MPI_Comm_size (MPI_Comm comm, int * size)**
    - **comm**: communicator (handle)
    - **size**: number of processes in the group of comm (integer)
- **MPI_Bcast**: a broadcast is one of the standard collective communication techniques. During a brodcast, a process sends the same data to all processes in a communicator. One of the primary uses of broadcast is to send user input to a parallel program or to send parameter configuration to all processes.
  - **int MPI_Bcast (void * buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**
    - **buffer:** starting address of buffer (choice)
    - **count:** number of entries in buffer (integer)
    - **datatype:** data type of buffer (handle)
    - **root:** rank of broadcast root (integer)
    - **comm:** communicator (handle)
- **MPI_Barrier**: blocking function that stops all processes until they reach this routine.
  - **int MPI_Barrier (MPI_Comm comm)**
    - **comm:** communicator (handle)
- **MPI_Gather**: gathers together values from a group of processes (equal size).
  - **int MPI_Gather (const void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)**
    - **sendbuf**: starting address of send buffer (choice)
    - **sendcount**: number of elements in send buffer (integer)
    - **sendtype**: data type of send buffer elements (handle)
    - **recvcount**: number of elements for any single receive (integer, significant only at root)
    - **recvtype**: data type of recv buffer elements (significant only at root) (handle)
    - **root**: rank of receiving process (integer)
    - **comm**: communicator (handle)
    - **recvbuf**: address of receive buffer (choice, significant only at root)
- **MPI_Gatherv**: Gathers into specified locations from all processes in a group (different size).
  - **int MPI_Gatherv (const void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, const int * recvcounts, const int * displs, MPI_Datatype recvtype, int root, MPI_Comm comm)**
    - **sendbuf:** starting address of send buffer (choice)

- **sendcount:** number of elements in send buffer (integer)
- **sendtype:** data type of send buffer elements (handle)
- **recvcounts:** integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
- **displs:** integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
- **recvtype:** data type of recv buffer elements (significant only at root) (handle)
- **root:** rank of receiving process (integer)
- **comm:** communicator (handle)
- **recvbuf:** address of receive buffer (choice, significant only at root)

5 Performance analysis of the algorithm

As previously said CHARM has a computational complexity equal to $T_1 = O\left(t \cdot 2^{|I|} \right)$.

We also remind you that parallelization is divided into two steps:

1. items on which CHARM is performed are associated with each process;
2. the master process (rank = 0) performs a linear scan on the Frequent Itemsets sent by the other processes, through Gatherv, so as to be able to filter obtaining the Frequent Closed Itemsets.

Therefore, in the worst case:

1. in step 1, the parallel computational cost is given by $O\left(t \cdot \frac{2^{|I|}}{p}\right)$, assuming the workload balanced equally between processes;
2. in step 2, since the master process must perform a linear scan on the possible candidates received, and since in the worst case the Frequent Itemset can be at most $2^{|I|}$, the parallel computational cost is given by $O\left(2^{|I|}\right)$.

We conclude by saying that the overall computational complexity of the parallel algorithm is:

$$O\left(t \cdot \frac{2^{|I|}}{p} + 2^{|I|}\right) = O(2^{|I|})$$

assuming that $t \ll 2^{|I|}$.

It can be observed that the theoretical analysis does not take into account the "time saved" in the first step (which will be highlighted in the later experimental analysis).

Now let's talk about the communication time: having an MPI_Gatherv for sending the possible candidates to the master process we have that the communication time is given in the worst case by $O(2^{|I|} \cdot \log p)$.

So T_p is given by the sum of the computation time and the communication time, we have that:

$$T_p = O\left(2^{|I|} + 2^{|I|} \cdot \log p\right) = O\left(2^{|I|} \cdot \log p\right).$$

We can also note here that the worst case parallel asymptotic time is greater than the sequential one, further supporting that the asymptotic analysis fails to appreciate the effort in parallelization.

Having defined this we can calculate the parallel overhead, given by:

$$T_0 = pT_p - T_1 = O\left[\left(t \cdot 2^{|I|} + p \cdot 2^{|I|} + \mathrm{p} \cdot 2^{|I|} \cdot \mathrm{logp}\right) - \ t \cdot 2^{|I|}\right] = O\left(p \cdot 2^{|I|} + \mathrm{p} \cdot 2^{|I|} \cdot \mathrm{logp}\right).$$

We also highlight, in support of the observations previously determined, the isoefficiency relationship which further highlights the incongruity of the theoretical analysis on the practical case in the parallelization process:

$$t \cdot 2^{|I|} \geq C \cdot \left(p \cdot 2^{|I|} + \mathrm{p} \cdot 2^{|I|} \cdot \mathrm{logp}\right)$$
$$t \geq Cp \cdot (1 + \mathrm{logp})$$

6 Functional tests

To test the fidelity of the results, I took the one mentioned above as a practical example; I also tested the results on larger datasets by comparing them with a sequential algorithm (modifying something on the reading) written in java available at the following link https://github.com/SatishUC15/CHARM-Algorithm/blob/master/CharmV2.java; the results were the same and collisions were visible only for a large number of FCI generated, but in a limited number (in the order of tens). Below are the results of the example described:

```
ITEMSET: 1 2 4 5 --> TIDS: 1 3 5
ITEMSET: 1 2 5 --> TIDS: 1 3 4 5
ITEMSET: 2 3 5 --> TIDS: 2 4 5
ITEMSET: 2 3 --> TIDS: 2 4 5 6
ITEMSET: 2 4 --> TIDS: 1 3 5 6
ITEMSET: 2 5 --> TIDS: 1 2 3 4 5
ITEMSET: 2 --> TIDS: 1 2 3 4 5 6

Ove A=1, B=2, C=3, D=4, E=5
```

7 Benchmark

This algorithm was run on a MacOS Monterey OS equipped with a 2.3Ghz Intel Core i9 8 core processor and 16Gb of 2400Mhz DDR4 RAM. Below we represent the table with the related tests performed, both by varying the chosen support, and by increasing the size of our dataset:

Dataset with 2500 items and a variable number of tids from 1 to 10000

	Running Time (seconds)			
N° PROCESSORS	**MIN_SUP = 2**	**MIN_SUP = 3**	**MIN_SUP = 4**	**MIN_SUP = 5**
1	0,630902	0,571236	0,566644	0,582698
2	0,351318	0,307809	0,311724	0,310000
4	0,213388	0,167959	0,169957	0,168565
8	0,151098	0,113118	0,116650	0,103738
N° FCI	**12281**	**2819**	**2361**	**2292**

Dataset with 5000 items and a variable number of tids from 1 to 10000

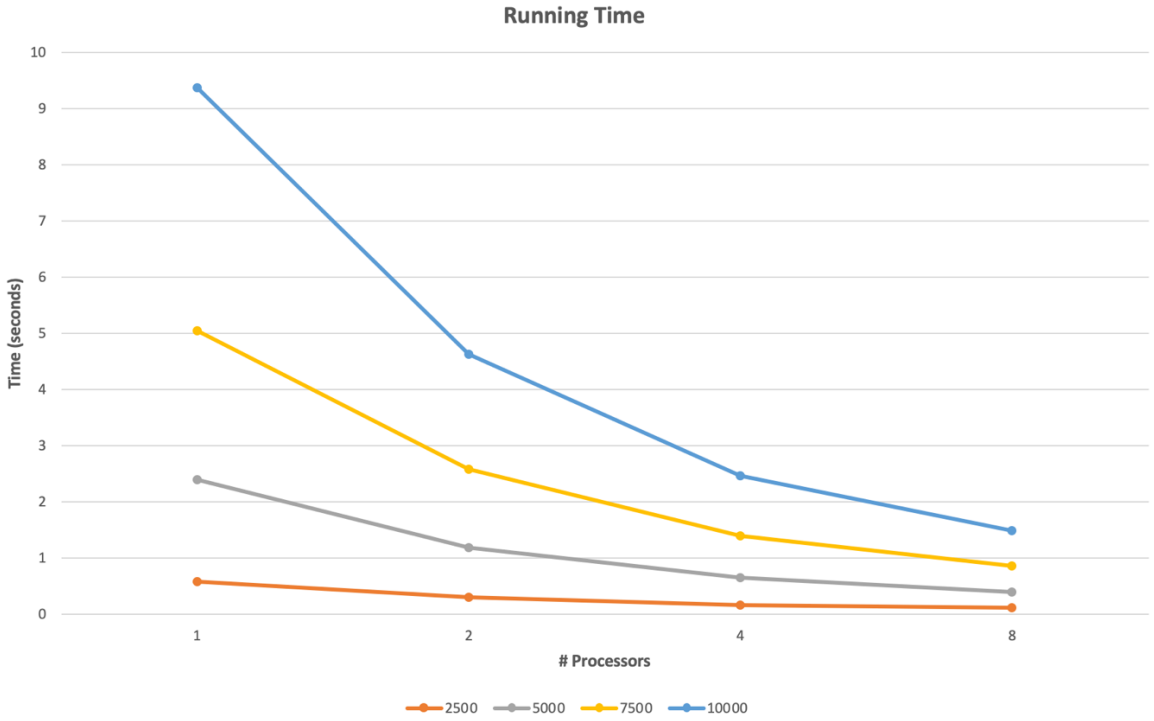
	Running Time (seconds)			
N° PROCESSORS	**MIN_SUP = 2**	**MIN_SUP = 3**	**MIN_SUP = 4**	**MIN_SUP = 5**
1	2,460189	2,390122	2,317613	2,238704
2	1,434550	1,181971	1,193601	1,190757
4	0,847228	0,641227	0,610601	0,653112
8	0,567516	0,387319	0,376997	0,388047
N° FCI	**43235**	**6428**	**4754**	**4597**

Dataset with 7500 items and a variable number of tids from 1 to 10000

	Running Time (seconds)			
N° PROCESSORS	**MIN_SUP = 2**	**MIN_SUP = 3**	**MIN_SUP = 4**	**MIN_SUP = 5**
1	5,361150	5,051593	5,057019	4,989970
2	2,983856	2,578681	2,631433	2,619742
4	1,735653	1,403631	1,395670	1,354395
8	1,178143	0,851239	0,811819	0,820841
N° FCI	**93111**	**10969**	**7187**	**6898**

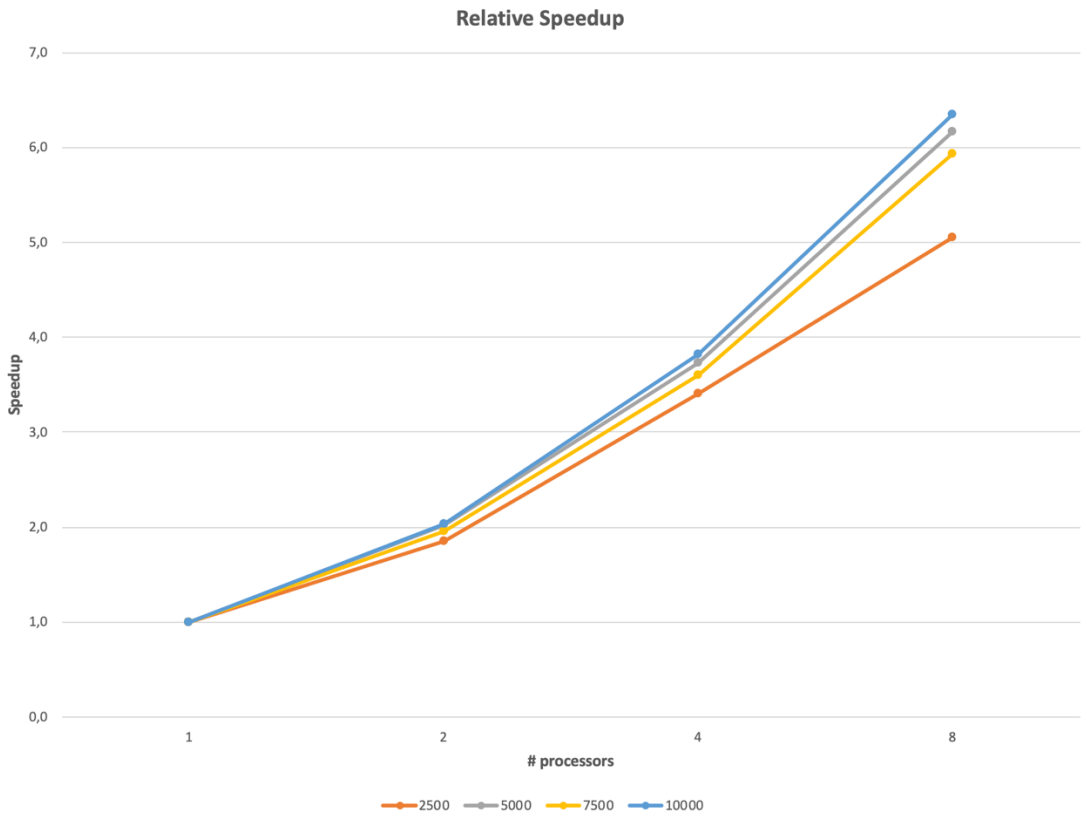
Dataset with 10000 items and a variable number of tids from 1 to 10000

	Running Time (seconds)			
N° PROCESSORS	**MIN_SUP = 2**	**MIN_SUP = 3**	**MIN_SUP = 4**	**MIN_SUP = 5**
1	10,349730	9,381631	8,989737	8,904228
2	5,158376	4,626875	4,530194	4,653737
4	3,044123	2,459037	2,434530	2,481697
8	2,029849	1,476956	1,470256	1,494704
N° FCI	**162011**	**16348**	**9658**	**9230**



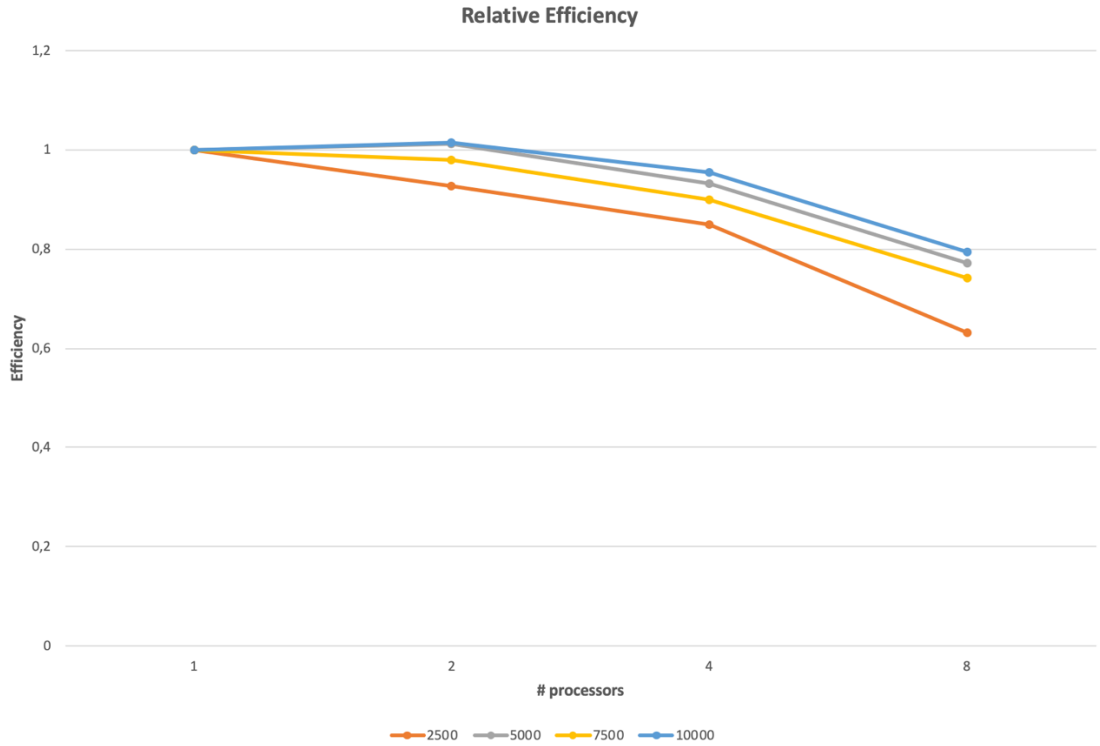
From the following graph it is possible to see how the execution times are reduced as the number of processors increases and the input size of the problem increases:

- for two processors the times become almost half and the slopes of the curves tend to increase as the number of input elements increases (the more the slope is marked, the higher the performance);
- for four processors this phenomenon is still visible with excellent performance;
- for eight processors we are starting to see a stall phase by virtue of the fact that the weight of communications is going to outweigh the reduction in computation time.



Let's see the analysis in terms of speedup:

- on two processors there is a perfect speedup, this implies that both processors are working perfectly and that the overhead has a much lower complexity than the parallelizable part (it grows faster than the intrinsically sequential and overloaded part); furthermore, a slight phenomenon of superlinear speedup can be seen (case 10000 and 5000 elements), which could be immediately connected to cache phenomena, but in reality it is not because it is due to the type of architecture on which the algorithm is running (evidently the cores are distributed in such a way that the communication is immediate for the data to be transmitted);
- on four processors as the input size increases, the speedup increases, slightly deviating from the optimal number;
- on eight processors we begin to see how the contribution due to communications affects, in fact we are very different from optimality.



Also in terms of efficiency, the analysis supports the one mentioned above:

- on two processors, the use of processors is perfect as the input size increases, if the input size is 10,000 or 5000 elements, the theoretical maximum achievable threshold of 1 (superlinearity) is even exceeded;
- out of four we begin to lose efficiency but as the size of the problem increases, it is possible to see how the maximum threshold is reached (the algorithm is working optimally);
- on eight processors it is lost in terms of efficiency in support of the one mentioned above (the algorithm is still good on large input sizes).

Conclusions

We can conclude by saying that the parallel version of the CHARM algorithm achieves better and better performances as the size of the problem increases and scales optimally on the processors up to a certain optimality limit, after which we can see the contribution of the weight due to the communications that leads to a saturation on the times, the meaning of which is that of having exceeded the optimal number of processors which minimizes the execution times in parallel.
We therefore confirmed what was said during the theoretical analysis, which is not reflected in the practical case.

8 Bibliography

1. https://dataminingbook.info/book_html/chap8/book.html
2. https://dataminingbook.info/book_html/chap9/book.html
3. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2956&rep=rep1&type=pdf