UNIVERSITE D'ORLEANS

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Dynamic Data Redistribution for Join Queries on Heterogeneous Shared Nothing Architecture

M. Al Hajj Hassan and M. Bamha
LIFO, Université d'Orléans

Rapport n^o **RR-2008-02**

Dynamic Data Redistribution for Join Queries on Heterogeneous Shared Nothing Architecture

M. Al Hajj Hassan and M. Bamha
LIFO, Université d'Orléans,
B.P. 6759, 45067 Orléans Cedex 2, France.
{alhassan,mbamha}@univ-orleans.fr

Abstract. Owing to the fast development of network technologies, executing parallel programs on distributed systems that connect heterogeneous machines became feasible but we still face challenges: workload imbalance in such environment may not only be due to uneven load distribution among machines as in parallel systems but also due to distribution that is not adequate with the characteristics of each machine. In this paper, we present a new parallel join algorithm for heterogeneous distributed architectures based on a dynamic data distribution and task allocation which makes it insensitive to data skew and ensures perfect balancing properties during all stages of join computation. The performance of this algorithm is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model. We show that our algorithm guarantees optimal complexity and near linear speed-up while reducing communication cost to a minimum.

Keywords: Heterogeneous distributed systems, Parallel join, Dynamic task allocation, Data skew, BSP cost model.

1 Introduction

Join operation is one of the most widely used operations in relational database systems, but it is also a heavily time consuming operation. For this reason it was a prime target for parallelization. Research has shown that it is parallelizable with near-linear speed-up on parallel systems such as Shared Nothing machines [1] but only under ideal balancing conditions where data skew may have disastrous effects on the performance of parallel algorithms on such architectures [2, 4, 15, 8, 14].

In [7] an adaptive parallel distributive join algorithm for evaluating join operation on a cluster of workstations is presented. This algorithm balances the load between processors even for skewed data. However, it is not scalable because all tuples of the smallest relation are sent to a central node which assigns to each processor in the system a range of values of the join attribute. During this step all other processors are idle which limits the performance of the system.
Other parallel algorithms were presented to handle data skew [12, 10]. However, they are not scalable because decisions concerning data buckets allocation are made by a coordinator node and in this time other processors are idle. In addition, data distribution is based on hashing data into buckets and hashing is inefficient in the presence of high frequencies. The communication cost in these algorithms is also very high because all tuples of both relations are redistributed even if they do not participate in the join result and usually the size of relations to be joined is very large.

Optimal skew-handling parallel algorithms for evaluating join on Shared Nothing machines were presented in [4, 2, 1]. These algorithms override the drawbacks of the algorithms explained above because the creation of communication templates that are used for data redistribution among processors is jointly performed in

[1]Shared Nothing machines: a distributed architecture where each processor has its own memory and own disks.

parallel by all processors, *each one not necessarily computing the list of its own messages, so as to balance the overall process*. This makes the algorithm scalable because we do not have idle processors waiting for a coordinator node. In addition, the communication cost is minimal where only tuples that participate in the join operation are redistributed. They also avoid the problem of Join Product Skew (JPS) while balancing the load of different processors even in the presence of Attribute Value Skew (AVS).

Today with the rapid development of network technologies, query processing on distributed systems that may connect heterogeneous pools of machines is gaining the interest of database communities due to their processing power and memory capacity. But the heterogeneity of these systems introduced new challenges which were not present in parallel systems such as shared nothing machines because in order to benefit effectively from the power of distributed systems no processor must be idle while other processors are overloaded [11]. Hence, the actual characteristics of resources such as CPU power, Input/Output speed, connection speed, available memory, etc. must be taken into consideration while assigning jobs to the nodes. It is known that in parallel systems where machines are usually homogeneous, workload imbalance may be due to uneven load distribution, but in heterogeneous distributed systems it may be due to load distribution which is not proportional to the actual capabilities of each machine [9]. Moreover, in multi-user systems the capacity of a machine may highly vary from one instant to another.

Algorithms used for join processing on a homogeneous parallel systems may not be effective on heterogeneous systems even if they efficiently handle the problem of data skew because they use a static strategy in load allocation. Hence, in order to obtain an acceptable performance on such systems an adaptive or dynamic load distribution strategy that takes in consideration the power of each machine and reacts with the system state must be used [11]. Zhang et al. [19] proposed three adaptive hash-based algorithms for evaluating join operations on such environments where a non-static redistribution strategy is used. In these algorithms, they addressed the problem of buffer overflow while building the hash tables by dynamically allocating additional machines. The algorithms start by creating the hash table of tables on a specific number of nodes. During this phase, if the memory of a node is used up, additional nodes will be employed in order to maintain the buckets in memory. The redistribution phase in this algorithm is based on hashing data into buckets using a hash function. However, it's known that hash functions are not efficient in the presence of skewed data. In addition, its communication cost is high because all tuples of both relations are redistributed even those which will not participate in the final join result.

In this paper, we propose and present the performance results of an optimal parallel algorithm to evaluate join operations on heterogeneous distributed systems. In this algorithm, we balance the load between processors in a manner that no processor in the system may be idle while other processors are overloaded even in the presence of highly skewed data. To this end, we use a two-step (static and dynamic) load assignment approach, where in the first static step each processor receives number of buckets whose total join size is proportional to its actual capacity. However, during the join phase, overloaded processors can forward some of the non-treated buckets in their local buffers to underloaded processors. Using this combination of static and dynamic approach allows us to efficiently handle the problem of AVS on heterogeneous systems due to the fact that total processing time is approximately the same on all processors. Join result size may differ between processors but this is due to the different processing capacities of machines. In addition, we use histograms to find tuples that participate in the result of the join operation and only these tuples will be redistributed, thus the communication cost is decreased to minimum. It is proved in [3, 4], using the BSP cost model, that histogram management has a negligible cost when compared to the gain it provides in semi-join computation and in reducing the communication cost and balancing load between processors. The performance of this algorithm was analyzed using the BSP cost model which shows that our algorithm guarantees optimal performance even for highly skewed data on heterogeneous distributed multi-processors architectures. These results were confirmed by a series of tests.
The cost model and the performance evaluation tests show that this algorithm has an optimal performance even for highly skewed data.

2 BSP Cost Model

Bulk-Synchronous Parallel (BSP) cost model is a programming model introduced by L. Valiant [17] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of multi-processor architectures [5, 16]. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization, the time g for collectively delivering a 1-relation (communication phase where each processor receives/sends at most one word). The network is assumed to deliver an h-relation in time $g * h$ for any arity h.

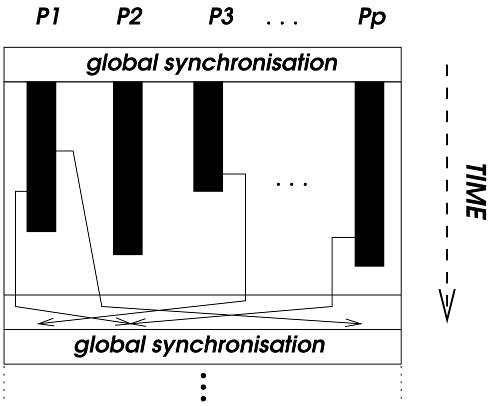


Figure 1: A BSP superstep.

A BSP program is executed as a sequence of *supersteps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses only its local data to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

where $w_i^{(s)}$ is the local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time, $\sum_s \text{Time}(s)$, of a BSP program composed of S supersteps is therefore a sum of 3 terms: $W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or (as in the present application) of more complex parameters like data skew and histogram sizes. To minimize execution time of a BSP algorithm, design must jointly minimize the number S of supersteps and the total volume h (resp. W) and imbalance $h^{(s)}$ (resp. $W^{(s)}$) of communication (resp. local computation).

3 DFA-join: New Approach for Evaluating Join Queries on Heterogeneous Distributed Systems

In this section, we describe in detail the steps of DFA-join (Dynamic Frequency Adaptive join) a new parallel algorithm for processing join operation on heterogeneous distributed systems formed of p processors having

different characteristics of memory, disk Input/Output speed, CPU power, etc.
We assume that the relations to be joined (R and S) are partitioned among processors by horizontal fragmentation.
In the rest of this paper, we use the following notation for each relation $T \in \{R, S\}$:

- T_i denotes the fragment of relation T placed on processor i, a sub-relation of T,
- $Hist(T)$ denotes the histogram [2] of relation T with respect to the join attribute, i.e. a list of pairs (v, n_v) where n_v is the number of tuples of relation T having the value v of the join attribute,
- $Hist(T_i)$ denotes the histogram of fragment T_i,
- $Hist_i(T)$ is processor i's fragment of the histogram of T,
- $Hist(T)(v)$ is the frequency (n_v) of value v in relation T while $Hist(T_i)(v)$ is its frequency in sub-relation T_i and $Hist_i(T)(v)$ is its frequency in processor i's fragment of the histogram of T,
- $\|T\|$ denotes the number of tuples of relation T, and $|T|$ denotes the size (expressed in bytes or number of pages) of relation T.

Basic parallel join processing usually proceeds in two phases: a redistribution phase then a join processing phase. In the first phase, tuples of both relations are redistributed among processors using a hash function. This function is applied on the values of the join attribute in order to send tuples having same values to the same processor. In the second phase, a sequential join of local fragments is performed. This induces high cost of communication because all tuples are redistributed over the network even if they do not participate to join result. In addition, the performance of such algorithms is very sensitive to data skew because the size of buckets assigned to different processors may highly vary. This degrades the performance of the algorithm because the total computation cost is the time taken by the slowest processor. Finally, the size of the join result on different processors may be highly skewed (JPS) which degrades the performance of parallel algorithms when treating complex queries.

To reduce communication costs and to evenly divide load between processors during all stages of join computation, we use the DFA-join algorithm that can be divided into the following five phases. We give for each phase an upper bound of computation time using BSP cost model [16, 18]. The notation $O(\dots)$ only hides small constant factors: they depend only on the implementation, but neither on data nor on the BSP machines parameters.

Phase 1. Creating local histograms:

In this phase, we create in parallel, on each processor i, the local histogram $Hist(R_i)$ (resp. $Hist(S_i)$) $(i = 1, \dots, p)$ of block R_i (resp. S_i) by a linear traversal of R_i (resp. S_i) in time $\max_{i=1,\dots,p}(c^i_{r/w} * |R_i|)$ (resp. $\max_{i=1,\dots,p}(c^i_{r/w} * |S_i|)$) where $c^i_{r/w}$ is the cost to read/write a page of data from disk on processor i.
The cost of this phase is: $Time_{phase1} = O\big(\max_{i=1,\dots,p} c^i_{r/w} * (|R_i| + |S_i|)\big)$.

Phase 2. Computing local semi-joins:

In order to minimize the redistribution cost, only tuples that participate in the final join result will be redistributed. These are tuples of the local semi-joins $\overline{R_i} = R_i \ltimes S$ (resp. $\overline{S_i} = S_i \ltimes R$) on each processor i.
To compute the semi-joins, we proceed by applying the following steps.

a. Creating $Hist_i(R)$ and $Hist_i(S)$ of R and S:
We firstly compute the global histograms $Hist_i(R)$ and $Hist_i(S)$ by redistributing the tuples of the local histograms using a hash function. The application of hash function, in parallel on each processor i, partitions $Hist(R_i)$ (resp. $Hist(R_i)$) into p buckets $Hist^j(R_i)_{j=1,\dots,p}$ $(Hist^j(S_i)_{j=1,\dots,p})$ where $Hist^j(R_i)$ (resp. $Hist^j(S_i)$)

[2] Histograms are implemented as balanced trees (B-tree): data structure that maintains an ordered set of data to allow efficient search and insert operations.

is the fragment of $Hist(R_i)$ (resp. $Hist(S_i)$) that will be sent to processor j.
The chosen hash function distributes the values of the join attribute in a manner that respects the processing capacity of each processor. Thus, the number of pages received by each processor j corresponding to the histogram of R is: $|\bigcup_i Hist^j(R_i)| \leq \min\left(\omega_j * p * |Hist(R)|, \omega_j * |R|\right)$ (this also applies to S). where ω_j is the fraction of the total volume of data assigned to processor j such that

$$\omega_j = \frac{\frac{1}{\gamma_j}}{\sum_{k=1}^{p} \frac{1}{\gamma_k}},$$

where γ_j is the execution time of one operation on processor j.

The cost of this step is: $Time_{phase2.a} =$

$$O\Big(\min\Big(\max_{i=1,\dots,p} \omega_i * p * \big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big), \max_{i=1,\dots,p} \omega_i * \big(g * |R| + \gamma_i * ||R||\big)\Big) +$$

$$\min\Big(\max_{i=1,\dots,p} \omega_i * p * \big(g * |Hist(S)| + \gamma_i * ||Hist(S)||\big), \max_{i=1,\dots,p} \omega_i * \big(g * |S| + \gamma_i * ||S||\big)\Big) + l\Big)$$

where g is the BSP communication parameter and l the cost of synchronization [17] (review *proposition 1* of appendix A for the proof of this cost).
We recall that the size of a histogram is, in general, very small compared to the size of base relations.

b. Creating $Hist_i(R \bowtie S)$ on each processor i:
Now we can easily create $Hist_i(R \bowtie S)$ by computing in parallel, on each processor i, the intersection of $Hist_i(R)$ and $Hist_i(S)$ in time of order:

$$Time_{phase2.b} = O\Big(\max_{i=1,\dots,p} \big(\gamma_i * \min(||Hist_i(R)||, ||Hist_i(S)||)\big)\Big).$$

While creating $Hist_i(R \bowtie S)$, we also store for each value $v \in Hist_i(R \bowtie S)$ an extra information $index(v) \in \{1, 2\}$ such that:

$$\begin{cases} index(v) = 1 & \text{if } Hist(R)(v) \geq f_o \text{ or } Hist(S)(v) \geq f_o, \\ index(v) = 2 & \text{elsewhere.} \end{cases}$$

The threshold frequency used in this algorithm is $f_o = p * log(p)$.
Each processor i also retains a trace of the network layout of values v in its $Hist_i(R)$ (resp. $Hist_i(S)$). This will be useful in the phase of creating the communication templates.
We also compute, on the fly on each processor i, the number of tuples of R (resp. S) that will participate in the join result. These are tuples having values v of the join attribute such that $v \in Hist_i(R \bowtie S)$. So the computed value is simply $\sum_{v \in Hist_i(R \bowtie S)} Hist_i(R)(v)$ $\big($ resp. $\sum_{v \in Hist_i(R \bowtie S)} Hist_i(R)(v)\big)$.

c. Now we need to determine, on each processor j, the values v of the join attribute in each partition $Hist^j(R_i)$ (resp. $Hist^j(S_i)$) that will be present in the join result.
So we compute, on each processor j, the intersections:

- $\overline{Hist}^j(R_i) = Hist^j(R_i) \bigcap Hist_j(R \bowtie S)$ for $i = 1, \dots, p$ and
- $\overline{Hist}^j(S_i) = Hist^j(S_i) \bigcap Hist_j(R \bowtie S)$ for $i = 1, \dots, p$,

in time of order: $O\Big(max_j \big(\gamma_j * \sum_i ||Hist^j(R_i)||\big) + max_j \big(\gamma_j * \sum_i ||Hist^j(S_i)||\big)\Big)$.
However, due to applying in stage a, a hashing function which partitions the buckets in a manner that takes into consideration the processing capacities of each machine, we have:

$$\begin{cases} \sum_i ||Hist^j(R_i)|| & \leq & \min(\omega_j * p * ||Hist(R)||, \omega_j * ||R||), \\ \sum_i ||Hist^j(S_i)|| & \leq & \min(\omega_j * p * ||Hist(S)||, \omega_j * ||S||). \end{cases}$$

Thus the total cost of this step is:
$Time_{phase2.c} = O\Big(\min\big(\max_i \omega_i * \gamma_i * p * ||Hist(R)||, \max_i \omega_i * \gamma_i * ||R||\big)$

$$+\min\big(\max_i \omega_i * \gamma_i * p * ||Hist(S)||, \max_i \omega_i * \gamma_i * ||S||\big)\Big).$$

d. In this step, each processor j sends each fragment $\overline{Hist}^j(R_i)$ (resp. $\overline{Hist}^j(S_i)$) to processor i. So the number of data pages received by each processor i from the other processors is: $\sum_j |\overline{Hist}^j(R_i)| + \sum_j |\overline{Hist}^j(S_i)|$. Hence, the cost of this communication step is of order: $\sum_j g * |\overline{Hist}^j(R_i)| + \sum_j g * |\overline{Hist}^j(S_i)|$, but $Hist(R_i) = \bigcup_j Hist^j(R_i)$ and $|Hist(R_i)| = \sum_j |Hist^j(R_i)| \geq \sum_j |Hist^j(R_i) \bigcap Hist(R \bowtie S)|$ (this also applies to S), thus $|Hist(R_i)| \geq \sum_j |\overline{Hist}^j(R_i)|$ (resp. $|Hist(S_i)| \geq \sum_j |\overline{Hist}^j(S_i)|$).
So the cost of this step is at most: $Time_{phase2.d} = O\Big(g * \big(|\overline{Hist}(R_i)| + |\overline{Hist}(S_i)|\big) + l\Big)$.

e. Finally, the semi-joins $\overline{R}_i = R_i \ltimes S$ and $\overline{S}_i = S_i \ltimes R$ are computed on each processor i. Tuples of $R_i \ltimes S$ (resp. $S_i \ltimes R$) are those whose join attribute values belong to $\overline{Hist}(R_i)$ (resp. $\overline{Hist}(S_i)$).
Thus $R_i \ltimes S$ (resp. $S_i \ltimes R$) can be computed, in parallel on each processor i, by a linear traversal of R_i (resp. S_i) and consulting $\overline{Hist}(R_i)$ (resp. $\overline{Hist}(S_i)$) in time of order: $O\big(\max_i \gamma_i * ||R_i|| + \max_i \gamma_i * ||S_i||\big)$.
$\overline{R}_i$ (resp. $\overline{S}_i$) is divided on the fly into two sub-relations: $\overline{R}_i^{'}$ and $\overline{R}_i^{''}$ ($\overline{R}_i = \overline{R}_i^{'} + \overline{R}_i^{''}$) where:

- tuples of $\overline{R}_i^{'}$ (resp. $\overline{S}_i^{'}$) are associated to values v such that $index(v) = 1$.
- tuples of $\overline{R}_i^{''}$ (resp. $\overline{S}_i^{''}$) are associated to values v such that $index(v) = 2$.

Total cost of this phase is the sum of the above five steps: $Time_{phase2} =$

$$O\Big(\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |R| + \gamma_i * ||R||\big)\Big)$$
$$+\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(S)| + \gamma_i * ||Hist(S)||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |S| + \gamma_i * ||S||\big)\Big)$$
$$+\max_i \gamma_i * ||R_i|| + \max_i \gamma_i * ||S_i|| + l\Big).$$

Phase 3. Creating the communication templates:

In heterogeneous systems, the actual capacity of each machine must be taken into account while assigning data or tasks to each processor in order to achieve an accepted performance. In addition, available capacities of machines in multi-user systems may rapidly change after load assignment, and the state of an overloaded processor may fastly become underloaded during join computation. Thus to benefit from the processing power of such systems, we must not have idle processors while others are overloaded through out all the join computation phase.

To this end, we use a two-step load assignment approach,

- the first step is a static one in which a coordinator node assigns to each processor a load proportional to its actual capacities.
- the second step is a dynamic one and is executed throughout the join computation phase. When a processor finishes join processing of its assigned buckets, it asks the coordinator node to forward to it some of untreated buckets on other processors. To ensure the extensibility of the algorithm, processors are partitioned into disjoint sets of processors. Each set of processors has a designed local coordinator node. Load is first balanced inside each set of processors and whenever a set of processors finishes its assigned tasks, it asks other set of processors for additional tasks.

This combination of static and dynamic approaches allows us to reduce the join processing time because, in parallel and distributed systems, total executing time is time taken by the slowest processor to finish its task.

a. Static load assignment step:
In the static load assignment step we compute, in parallel on each processor i, the size of the join of all tuples related to values v that belong to $Hist_i(R \bowtie S)$. This is simply the sum of the frequencies $Hist_i(R \bowtie S)(v)$ for all values v of the join attribute in $Hist_i(R \bowtie S)$. This value is computed by a linear traversal of $Hist_i(R \bowtie S)$ in time: $O\big(\max_{i=0,\ldots,p} \gamma_i * ||Hist_i(R \bowtie S)||\big)$.

After that, all processors send the computed values ($\sum_{v \in Hist_i(R \bowtie S)} Hist_i(R)(v)$, $\sum_{v \in Hist_i(R \bowtie S)} Hist_i(S)(v)$, $\sum Hist_i(R \bowtie S)(v)$) to a designated coordinator node in time of order $O(3 * p * g + l)$ which in its turn calculates the number of tuples of R (i.e. $||\overline{R}||$) and S (i.e. $||\overline{S}||$) that participate in the final join result. In addition to the total number of tuples in $R \bowtie S$ ($||R \bowtie S||$) by respectively computing the sum of values received from all the processors.
Now, the coordinator node uses the value of $||R \bowtie S||$ and information concerning the available capacities to assign to each processor i a join volume ($vol_i * ||R \bowtie S||$) proportional to its resources where the value of vol_i is determined by the head node depending on the actual capacity of each processor i such that $\sum_{i=1}^{p} vol_i = 1$.
The coordinator node will also assign to each processor i, the number of tuples $vol_i * ||\overline{R}||$ (resp. $vol_i * ||\overline{S}||$) of R (resp. S) that it can receive during the static phase. This is important to assure that the global size of data received by each processor is not highly skewed.
Finally, the head node sends to each processor i a message having the form $(vol_i * ||\overline{R}||, vol_i * ||\overline{S}||, vol_i * ||R \bowtie S||)$ in time of order: $O(3 * p * g + l)$.
The cost of this step is: $Time_{phase3.a} = O(3 * p * g + l)$.

b. Communication templates creation
Communication templates are list of messages that constitute the relations redistribution. We will only create them for join attribute values v having high frequencies (i.e. $index(v) = 1$), because they can lead to AVS and cause JPS. Thus they need a special treatment in the contrary to tuples with low frequencies (i.e. $index(v) = 2$) that do not have effect neither on AVS nor JPS. So these tuples will be simply hashed into buckets in their source processors using a hash function and their treatment will be postponed to the dynamic phase.
So first of all, $Hist_i(R \bowtie S)$ must be partitioned on each processor i into two sub-histograms: $Hist_i^{(1)}(R \bowtie S)$ and $Hist_i^{(2)}(R \bowtie S)$ such that:

- $v \in Hist_i^{(1)}(R \bowtie S)$ if $index(v) = 1$ i.e. join attribute value with high frequency in R and/or S,
- $v \in Hist_i^{(2)}(R \bowtie S)$ if $index(v) = 2$ i.e. join attribute value with low frequency in R and S.

This partitioning step is performed while computing $\sum_v Hist_i(R \bowtie S)(v)$ in step 3.a in order to avoid reading the histogram two times.
We can start now by creating the communication templates which are computed, in a fist time, on each processor i for only values v in $Hist_i(R \bowtie S)$ such that the total join size related to these values is inferior or equal to $vol_i * ||R \bowtie S||$ starting from the value that generates the highest join result and so on. In addition, the number of received tuples of R (resp. S) must not exceed $vol_i * ||\overline{R}||$ (resp. $vol_i * ||\overline{S}||$). For each value v in $Hist_i(R \bowtie S)$, processor i creates communicating messages $order\_to\_send(j, i, v)$ asking each processor j that holds tuples of R or S having values v for the join attribute to send them to it.
If the processing capacity of a processor i doesn't allow it to compute the join result associated to all values v of the join attribute in $Hist_i(R \bowtie S)$ [3], then it will not ask the source processors j holding the remaining values v to redistribute their associated tuples but to partition them into buckets using a hash function and save them locally for farther join processing step in the dynamic phase. Hence it sends an *order_to_save(j,v)* message for each processor j holding tuples having values v of the join attribute.
The maximal complexity of creating the communication templates is:

$$O\Big(\max_i \left(\omega_i * p * \gamma_i * ||Hist^{(1)}(R \bowtie S)||\right)\Big),$$

because each processor i is responsible of creating the communication template for approximately $\omega_i * ||Hist^{(1)}(R \bowtie S)||$ values of the join attribute and for a given value v at most $(p-1)$ processors can send data.
After creating the communication templates, on each processor i, messages: $order\_to\_send(j, i, .)$ and $order\_to\_save(j, .)$

[3] This is the case if $vol_i * ||R \bowtie S|| < \sum_v Hist_i(R \bowtie S)(v)$ on processor i.

are sent to their destination processors j when $j \neq i$ in time:

$$Time_{phase3.b} = O\Big(\max_i \left(g * \omega_i * p * |Hist^{(1)}(R \bowtie S)|\right) + l\Big).$$

The total cost of this phase is :

$$Time_{phase3} = O\Bigg(\max_i \left(\omega_i * p * \left(g * |Hist^{(1)}(R \bowtie S)| + \gamma_i * ||Hist^{(1)}(R \bowtie S)||\right)\right) + l\Bigg).$$

Phase 4. Redistribution phase:

In this phase, each processor i obeys the *order_to_send*(.,.,.) messages that it has just received. So tuples that must be sent to each processor are partitioned into multiple number of buckets greater than p using a hash function. Hashing data assigned to a processor into multiple number of buckets facilitates task reallocation in the join phase (phase 5) from overloaded to idle processors. In addition, each processor i will partition tuples whose join attribute value is indicated in *order_to_save*() messages or those of $\overline{R}_i^{''}$ (resp. $\overline{S}_i^{''}$) into buckets using the same hash function on all the processors. However, these buckets will be kept for a moment in their source processors where their redistribution and join processing operations will be postponed till the dynamic phase.

After this partition step, buckets are sent to their destination processors.

It is important to mention here that only tuples of R and S that effectively participate in the join result will be redistributed. So each processor i receives a partition of R (resp. S) whose maximal size is $vol_i * ||\overline{R}||$ (resp. $vol_i * ||\overline{R}||$). Therefore, in this algorithm, communication cost is reduced to a minimum and is of order:

$$Time_{phase4} = O\Big(g * \max_i \left(vol_i * (|\overline{R}| + |\overline{S}|\right) + l\Big).$$

Phase 5. Join computation phase:

Buckets received by each processor are arranged in a queue. Each processor executes successively the join operation of its waiting buckets. The cost of this step of local join computation is:

$$Time_{local\_join} = O\Big(\max_i \left(c^i_{r/w} * vol_i * (|\overline{R}| + |\overline{S}| + |\overline{R} \bowtie \overline{S}|)\right)\Big)$$

If a processor finishes computing the join related to its local data and the overall join operation is not finished, it will send to the coordinator node a message asking for more work. Hence, the coordinator node will assign to this idle processor some of the buckets related to join attribute values that were not redistributed earlier in the static phase. However, if all these buckets are already treated the head node checks the number of non treated buckets in the queue of the other processors and asks the processor that has the maximal number of non treated buckets to forward a part of them to the idle one. The number of sent buckets must respect the capacity of the idle processor.

And thus, the global cost of join computation of two relations R and S using DFA-join algorithm is:

$$\begin{aligned}
Time_{DFA-join} = O\Bigg(& \max_{i=1,\ldots,p} c^i_{r/w} * (|R_i| + |S_i|) + \max_{i=1,\ldots,p} \left(c^i_{r/w} * vol_i * (|\overline{R}| + |\overline{S}| + |\overline{R} \bowtie \overline{S}|)\right) \\
&+ \min\Big(\max_{i=1,\ldots,p} \omega_i * p * \left(g * |Hist(R)| + \gamma_i * ||Hist(R)||\right), \max_{i=1,\ldots,p} \omega_i * \left(g * |R| + \gamma_i * ||R||\right)\Big) \\
&+ \min\Big(\max_{i=1,\ldots,p} \omega_i * p * \left(g * |Hist(S)| + \gamma_i * ||Hist(S)||\right), \max_{i=1,\ldots,p} \omega_i * \left(g * |S| + \gamma_i * ||S||\right)\Big) \\
&+ \max_{i=1,\ldots,p} \Big(\omega_i * p * \left(g * |Hist^{(1)}(R \bowtie S)| + \gamma_i * ||Hist^{(1)}(R \bowtie S)||\right)\Big) \\
&+ g * \max_{i=1,\ldots,p} \left(vol_i * (|\overline{R}| + |\overline{S}|\right) + \max_{i=1,\ldots,p} \left(\gamma_i * ||R_i||\right) + \max_{i=1,\ldots,p} \left(\gamma_i * ||S_i||\right) + l\Bigg).
\end{aligned}$$

Discussion

If DFA-join algorithm is used on a homogeneous mono-user distributed system, then the load assigned to different processors by the coordinator node will be approximately the same. Hence, the cost of applying DFA-join is comparable to OSFA-join [1]. OSFA_join algorithm is a parallel algorithm for treating join queries on Shared Nothing machines based on efficient semi-join technique. It has an optimal complexity even in the presence of highly skewed data on SN machines. However, in OSFA-join algorithm, we have an additional cost $g * (|\widetilde{R''}| + |\widetilde{S''}|)$ which represents the cost of duplicating some partitions of data on all

processors.
DFA-join attains better performance in heterogeneous systems due to the fact that tasks are assigned depending on the available resources of each machine. If OSFA-join is applied, the load will be evenly partitioned among processors, hence fast machines may terminate their assigned tasks before other slower ones. So the processing resources of these processors are not efficiently used. Thus, the performance of OSFA-join algorithm is not optimal on homogeneous multi-user and heterogeneous distributed systems.
We conclude that DFA-join algorithm can be used and has optimal performance on homogeneous multi-user as well as on heterogeneous distributed systems.

Remark:
Sequential evaluation of the join of two relations R and S on processor i requires at least the following lower bound:

$$bound_{inf_1} = \Omega\big(c^i_{r/w} * (|R| + |S| + |R \bowtie S|) + \gamma_i * (||R|| + ||S|| + ||R \bowtie S||)\big).$$

Therefore, parallel join processing on p heterogeneous processors requires:

$$bound_{inf_p} = \Omega\Big(\max_i \big(c^i_{r/w} * \omega_i * (|R| + |S| + |R \bowtie S|) + \gamma_i * \omega_i * (||R|| + ||S|| + ||R \bowtie S||)\big)\Big).$$

DFA-join algorithm has optimal asymptotic complexity when:

$$\max_{i=1,\ldots,p} \big(\omega_i * p * |Hist^{(1)}(R \bowtie S)|\big) \leq \max_i \big(c^i_{r/w} * \omega_i * \max(|R|, |S|, |R \bowtie S|)\big), \tag{1}$$

this is due to the fact that other terms in $Time_{DFA-join}$ are bounded by those of $bound_{inf_p}$.
Inequality (1) holds if the chosen threshold frequency f_o is greater than p (which is the case for our threshold frequency $f_o = p * log(p)$).

4 Performance Evaluation

We compared the performance of DFA-join algorithm to our previous OSFA_join (Optimal symmetric frequency adaptive join) algorithm [1] and the standard algorithm based on pure hashing.
In the tests, we were interested in comparing the speed-up of the three algorithms and the effect of data skew on their performance. The tests were performed on a homogeneous multi-user system [4] formed of 8 processors using MySQL-5.0.26 Database server and MPI-2 as Message Passing Interface for communications.

To study the effect of data skew on the performance, frequencies of join attribute values have been taken to follow the *Zipf* [20, 6, 13] as it is the case in most database tests. The number of tuples of the i^{th} distinct value of the join attribute in a relation R with a domain $\{1, 2, \ldots, D\}$ is given by the following expression:

$$||v_i|| = \frac{||R||}{i^z * \sum_{j=1}^{D} \frac{1}{j^z}}$$

where z is the skew factor. The distribution is considered uniform if $z = 0$ and highly skewed when $z \geqslant 1$.

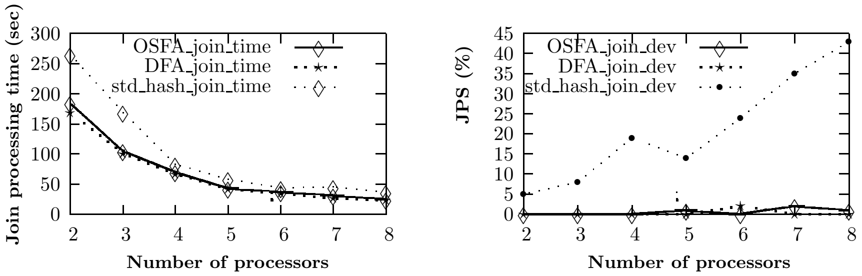
The JPS (Join Product Skew) was measured as the percentage of the maximum deviation of the local join result size with respect to the average over all processors.

4.1 Speed-up test

To perform the speed-up test, we applied the three algorithms on two relations formed of $500K$ tuples each and with a *Zipf* skew factor of value 0.8 which represents a relatively weak degree of skew.
The number of processors was varied from 1 to 8 where the join result was formed of approximately 45 million

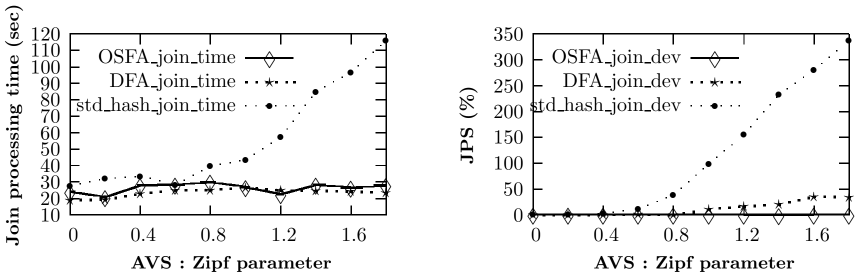
[4] For better results, the simulations must be performed on a heterogeneous distributed systems, but currently we do not have access to such system.

tuples. The results obtained showed that OSFA_join and DFA-join algorithms outperform the standard hash join (std-hash-join) algorithm (fig. 2(a)), The results also showed that DFA-join algorithm is faster than OSFA_join algorithm, although the tests are performed on a multi-user homogeneous distributed system. The cost model assures that the performance of DFA-join is better than OSFA_join algorithm in heterogeneous systems owing to its dynamic task allocation approach for load balancing. We can also see in figure 2(b) that the standard hash algorithm suffers from the JPS, however, the percentage of JPS in DFA-join and OSFA_join is negligible.



(a) (b)

Figure 2: Speed-up and Join Product skew (*JPS*) tests with a low Attribute Value Skew (*AVS*).



(a) (b)

Figure 3: Effect of Attribute Value Skew (*AVS*) on join processing time and Join Product Skew (JPS).

4.2 Attribute value skew test

To study the effect of AVS on DFA-join algorithm, we fixed the number of processors to 8 and the value of the *Zipf* skew factor was varied from 0.0 to 1.8. The test was applied on two relations of size $500K$ tuples each. Figure 3(a), shows that OSFA_join and DFA-join algorithms are much more faster than the standard hash join algorithm and that their processing time remains balanced while varying the skew factor. We can also see that the execution time of DFA-join algorithm is better than OSFA_join algorithm in approximately all the cases.

We can also deduce from figure 3(b) that the standard hash algorithm suffers from a high percentage of skew in the result of the join operation, which is not the case of DFA-join and OSFA-join algorithms. However, we can see that the percentage of skew that results from applying OSFA-join algorithm is smaller than that of DFA-join. This is due to the fact that OSFA-join algorithm, that is dedicated to homogeneous parallel systems, evenly distributes the tuples of the input relation in a manner that balances the size of the result of the join. However, DFA-join algorithm can be used on a heterogeneous distributed systems where we do not want to have idle processors while others are overloaded, hence the size of the join result on each processors depends on its capacity.

5 Conclusion

In this paper, we presented a parallel join algorithm based on a dynamic data redistribution approach allowing to reduce communication cost to a minimum while guaranteeing perfect balancing properties on multi-user homogeneous architecture as well as on heterogeneous distributed architectures.
The BSP cost model showed that the overhead related to histogram's management remains very small compared to the gain it provides in reducing communication time and handling buckets reallocation to balance load between processor. This algorithm proved to have an optimal complexity even for highly skewed data. Future work will be devoted to a pipelined version of this algorithm for treating mult-join queries on multi-user homogeneous and heterogeneous distributed systems. We are also interested in adapting this algorithm to Grid architectures.

References

[1] M. Bamha. An optimal and skew-insensitive join and multi-join algorithm for ditributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *Lecture Notes in Computer Science*, pages 616–625. Springer-Verlag, 2005.

[2] M. Bamha and G. Hains. A skew-insensitive algorithm for join and multi-join operation on shared nothing machines. In *the 11th International Conference on Database and Expert Systems Applications DEXA'2000*, volume 1873 of *Lecture Notes in Computer Science*, London, United Kingdom, 2000. Springer-Verlag.

[3] M. Bamha and G. Hains. An efficient equi-semi-join algorithm for distributed architectures. In *Proceedings of the 5th International Conference on Computational Science (ICCS'2005). 22-25 May, Atlanta, USA*, volume 3515 of *Lecture Notes in Computer Science*, pages 755–763. Springer-Verlag, 2005.

[4] M. Bamha and G. Hains. A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345*, September 1999. Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.

[5] Rob H. Bisseling. *Parallel Scientific Computation : A Structured Approach using BSP and MPI.* Oxford University Press, USA, 2004.

[6] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.

[7] Soon M. Chung and Arindam Chatterjee. An adaptive parallel distributive join algorithm on a cluster of workstations. *J. Supercomput.*, 21(1):5–35, 2002.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference*, pages 27–40, Vancouver, British Columbia, Canada, 1992.

[9] Anastasios Gounaris. Resource aware query processing on the grid. Thesis report, University of Manchester, Faculty of Engineering and Physical Sciences, 2005.

[10] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the 17th International Conference on Very Large Data Bases*, pages 525–535, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[11] H. Karatza and R. Hilzer. Load sharing in heterogeneous distributed systems. In *Proceedings of the 2002 Winter Simulation Conference*, pages 489–496, 2002.

[12] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for skew in the super database computer (SDC). In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Very Large Data Bases: 16th International Conference on Very Large Data Bases, August 13–16, 1990, Brisbane, Australia*, pages 210–221, Los Altos, CA 94022, USA, 1990. Morgan Kaufmann Publishers.

[13] H. Lu and K.-L. Tan. Dynamic and load-balanced task-oriented database query processing in parallel systems. In *Proceedings of Advances in Database Technology (EDBT '92)*, volume 580 of *LNCS*, pages 357–372, Berlin, Germany, 1992. Springer.

[14] A. N. Mourad, R. J. T. Morris, A. Swami, and H. C. Young. Limits of parallelism in hash join algorithms. *Performance evaluation*, 20(1/3):301–316, May 1994.

[15] M. Seetha and P. S. Yu. Effectiveness of parallel joins. *IEEE, Transactions on Knowledge and Data Enginneerings*, 2(4):410–424, December 1990.

[16] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[18] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[19] Xi Zhang, Tahsin Kurc, Tony Pan, Umit Catalyurek, Sivaramakrishnan Narayanan, Pete Wyckoff, and Joel Saltz. Strategies for using additional resources in parallel hash-based join algorithms. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 4–13, Washington, DC, USA, 2004. IEEE Computer Society.

[20] G. K. Zipf. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. *Reading, MA, Adisson-Wesley*, 1949.

Appendix A

Proposition 1: Cost of creating global histogram fragments, $Hist_i(R)_{i=1,\dots,p}$, on heterogeneous systems starting from the local histograms $Hist(R_i)_{i=1,\dots,p}$ of relation R_i of R according to some attribute A is of order [5]:

$$O\Big(\min\Big(\max_{i=1,\dots,p}\omega_i * p * \big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big), \max_{i=1,\dots,p}\omega_i * \big(g * |R| + \gamma_i * ||R||\big)\Big) + l\Big) \qquad (2)$$

if a hash function $\mathcal{H}$ that distributes the distinct values of the attribute A among the p processors in a manner that respects the processing capacity of each machine is used.

In the above equation, γ_i is the time needed for executing one operation on processor i and ω_i is the fraction of the total volume of data that must be assigned to processor i in a manner that respects its available capacity in order to allow all participating processors to terminate their assigned tasks approximately at the same time where ω_i is given by the following formula:

$$\omega_i = \frac{\frac{1}{\gamma_i}}{\sum_{k=1}^{p}\frac{1}{\gamma_k}}.$$

[5] This is the generalized form of *proposition 1* presented in [3] for computing $Hist_i(R)$ in a Shared Nothing architecture.

Proof: In order to prove this cost we will consider four cases where the difference between them lies in the frequency distribution of the attribute A of relation R.

case 1: In this case, we consider that all the values of the attribute A appear in the relation R with a frequency equal to 1 hence $||Hist(R)|| = ||R||$.
So if we use a hash function $\mathcal{H}$ on each processor i that distributes the tuples of R in a manner that respects the processing power of each machine, then each processor will receive $\omega_i * ||Hist(R)||$ tuples from all the processors in the system. So the cost of creating $Hist_i(R)$ is of order:

$$O\Big(\max_{i=1,\ldots,p} \omega_i\big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big) + l\Big) \leqslant O\Big(\max_{i=1,\ldots,p} \omega_i\big(g * |R| + \gamma_i * ||R||\big) + l\Big). \quad (3)$$

The term $\omega_i * g * |Hist(R)|$ is the necessary time for communicating data and the term $\omega_i * \gamma_i * ||Hist(R)||$ is the time needed to form $Hist_i(R)_{i=1,\ldots,p}$ starting from the blocks received by each processor i during the distribution of the local histograms $Hist(R_i)_{i=1,\ldots,p}$.
The histogram is formed of couples having the form (v, n_v) where v are the distinct values of attribute A of R and n_v their corresponding frequencies hence $|Hist(R)| \leqslant |R|$ which makes that the above inequality holds.
So for the case where all the values of the attribute A have a frequency equal to 1 in R, the fragments $Hist_i(R)$ of the global histogram can be computed in time of order:

$$O\Big(\min\big(\max_{i=1,\ldots,p} \omega_i * (g * |Hist(R)| + \gamma_i * ||Hist(R)||), \max_{i=1,\ldots,p} \omega_i * (g * |R| + \gamma_i * ||R||)\big) + l\Big)$$
$$\leqslant O\Big(\min\big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist(R)| + \gamma_i * ||Hist(R)||), \max_{i=1,\ldots,p} \omega_i * (g * |R| + \gamma_i * ||R||)\big) + l\Big). \quad (4)$$

case 2: Now we consider that the frequency of all the values of the attribute A in R is lower than p. In order to use the cost obtained in *case 1*, relation R must be partitioned into p relations $R^1, R^2, \ldots, R^p$ such that the frequency of each value v in relation $R^j (j = 1, \ldots, p)$ is exactly equal to 1.
So the cost of creating $Hist_i(R^j)$ of each relation R^j for $j = 1, \ldots, p$ according to *case 1* is of order:

$$O\Big(\min\big(\max_{i=1,\ldots,p} \omega_i * (g * |Hist(R^j)| + \gamma_i * ||Hist(R^j)||), \max_{i=1,\ldots,p} \omega_i * (g * |R^j| + \gamma_i * ||R^j||)\big) + l\Big), \quad (5)$$

hence the necessary time for creating $Hist_i(R)$ of relation R is:

$$O\Big(\sum_{j=1,\ldots,p} \min\big(\max_{i=1,\ldots,p} \omega_i * (g * |Hist(R^j)| + \gamma_i * ||Hist(R^j)||), \max_{i=1,\ldots,p} \omega_i * (g * |R^j| + \gamma_i * ||R^j||)\big) + l\Big)$$
$$\leqslant O\Big(\min\big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist(R)| + \gamma_i * ||Hist(R)||), \max_{i=1,\ldots,p} \omega_i * (g * |R| + \gamma_i * ||R||)\big) + l\Big). \quad (6)$$

The above inequality holds due to the fact that $Hist_i(R)$ can be created by simply fusioning the fragments $Hist_i(R^j)_{j=1,\ldots,p}$ in time of order:

$$O\big(\sum_{j=1,\ldots,p} \max_{i=1,\ldots,p} \omega_i * \gamma_i * ||Hist_i(R^j)||\big) \leqslant O\big(\max_{i,j} \omega_i * \gamma_i * ||Hist_i(R^j)|| * p\big)$$
$$\leqslant O\big(\max_i \omega_i * \gamma_i * ||Hist(R)|| * p\big)$$

and that $\sum_i \min(a_i, b_i) \leqslant \min(\sum_i a_i, \sum_i b_i)$.
In practice, we do not need to partition relation R into p relations where each processor distributes the couples (v, n_v) related to all values v of the join attribute A using a hash function even if $n_v > 1$. However, we used this technique here to demonstrate the validity of the cost model.

case 3: In this case, we consider that the frequencies of all the values of the attribute A are higher than p. Hence, each processor i is responsible of treating $\omega_i * ||Hist(R)||$ distinct values of R where ω_i depends on the processing power of each processor, and if each value v of attribute A is found on all the p processors than each processor i will receive a block of data of maximal size $\omega_i * |Hist(R)| * p$ from all the other processors.

So if an appropriate hash function is used, each processor receives a block of size $\omega_i * |Hist(R)|$ with a communication cost of order:

$$O\Big(\max_i \big(g * \omega_i * |Hist(R)| * p\big) + l\Big) \leqslant O\Big(\max_i \big(g * \omega_i * |R|\big) + l\Big). \quad (7)$$

This inequality holds due to the fact that $||Hist(R)|| \leqslant \frac{||R||}{p}$ because each value v of A appears with a frequency higher than p thus $\omega_i * ||Hist(R)|| * p \leqslant \omega_i ||R||$.

After this step of communication, $||Hist_i(R)||$ is created on each processor i by a simple fusion of all the fragments of $Hist(R_j)_{j=1,\ldots,p}$ received from all processors j in time of order: $O\big(\omega_i * \gamma_i * ||Hist(R)|| * p\big)$. Thus the cost of creating $Hist_i(R)$ is of order:

$$O\bigg(\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |R| + \gamma_i * ||R||\big)\Big) + l \bigg). \tag{8}$$

case 4: The cost of the general case can be deduced from previous three cases where R is partitioned into two disjoint sub-relations R' and $R"$ such that the tuples of R' (resp. $R"$) are associated with frequencies lower (resp. higher) than p for the join attribute A. Thus using cases 1 and 2, the creation of the histogram of R' costs:

$$O\bigg(\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(R')| + \gamma_i * ||Hist(R')||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |R'| + \gamma_i * ||R'||\big)\Big) + l \bigg), \tag{9}$$

and according to case 3 the creation of the histogram of $R"$ costs:

$$O\bigg(\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(R")| + \gamma_i * ||Hist(R")||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |R"| + \gamma_i * ||R"||\big)\Big) + l \bigg). \tag{10}$$

Owing to the fact that $Hist(R) = Hist(R') \cup Hist(R")$ and that $min(a,b) + min(c,d) \leqslant min(a+c, b+d)$, we can deduce that the cost of creating $Hist_i(R)$ is at most of order:

$$O\bigg(\min\Big(\max_{i=1,\ldots,p} \omega_i * p * \big(g * |Hist(R)| + \gamma_i * ||Hist(R)||\big), \max_{i=1,\ldots,p} \omega_i * \big(g * |R| + \gamma_i * ||R||\big)\Big) + l \bigg). \tag{11}$$