PIPELINED PARALLELISM IN MULTI-JOIN QUERIES ON HETEROGENEOUS SHARED NOTHING ARCHITECTURES

Mohamad Al Hajj Hassan and Mostafa Bamha
LIFO, University of Orléans, BP 6759, 45067 Orléans cedex 2, France

Keywords: PDBMS (Parallel Database Management Systems), Intra-transaction parallelism, Parallel joins, Multi-joins, Data skew, Dynamic load balancing.

Abstract: Pipelined parallelism was largely studied and successfully implemented, on shared nothing machines, in several join algorithms in the presence of ideal conditions of load balancing between processors and in the absence of data skew. The aim of pipelining is to allow flexible resource allocation while avoiding unnecessary disk input/output for intermediate join results in the treatment of multi-join queries.
The main drawback of pipelining in existing algorithms is that communication and load balancing remain limited to the use of static approaches (generated during query optimization phase) based on hashing to redistribute data over the network and therefore cannot solve data skew problem and load imbalance between processors on heterogeneous multi-processor architectures where the load of each processor may vary in a dynamic and unpredictable way.
In this paper, we present a new parallel join algorithm allowing to solve the problem of data skew while guaranteeing perfect balancing properties, on heterogeneous multi-processor Shared Nothing architectures. The performance of this algorithm is analyzed using the scalable portable BSP (Bulk Synchronous Parallel) cost model.

1 INTRODUCTION

The appeal of parallel processing becomes very strong in applications which require ever higher performance and particularly in applications such as: data-warehousing, decision support, On-Line Analytical Processing (OLAP) and more generally DBMS (Liu and Rundensteiner, 2005; Datta et al., 1998). However parallelism can only maintain acceptable performance through efficient algorithms realizing complex queries on dynamic, irregular and distributed data. Such algorithms must be designed to fully exploit the processing power of multi-processor machines and the ability to evenly divide load among processors while minimizing local computation and communication costs inherent to multi-processor machines.

Research has shown that the join operation is parallelizable with near-linear speed-up on Shared Nothing machines only under ideal balancing conditions. Data skew can have a disastrous effect on performance (Mourad et al., 1994; Hua and Lee, 1991; DeWitt et al., 1992; Bamha and Hains, 2000; Bamha and Hains, 1999) due to the high costs of communications and synchronizations in this architecture.

Many algorithms have been proposed to handle data skew for a simple join operation, but little is known for the case of complex queries leading to multi-joins (Lu et al., 1994; DeWitt et al., 1992; Hua and Lee, 1991).
However, these algorithms cannot solve load imbalance problem as they base their routing decisions on incomplete or statistical information.
On the contrary, the algorithms we presented in (Bamha and Hains, 1999; Bamha and Hains, 2000; Bamha, 2005) for treating queries involving one join operation use a total data-distribution information in the form of histograms. The parallel cost model we apply allows us to guarantee that histogram management has a negligible cost when compared to the efficiency gains it provides to reduce the communication cost and to avoid load imbalance between processors.
However the problem of data skew is more acute with multi-joins because the imbalance of intermediate results is unknown during static query optimization (Lu et al., 1994).

To this end, we introduced in (Bamha and Exbrayat, 2003) a pipelined version of OSFA-Join

Al Hajj Hassan M. and Bamha M. (2008).
PIPELINED PARALLELISM IN MULTI-JOIN QUERIES ON HETEROGENEOUS SHARED NOTHING ARCHITECTURES.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 127-134
DOI: 10.5220/0001889901270134
Copyright © SciTePress

based on a dynamic data redistribution approach allowing to solve the problem of Attribute Value Skew (AVS) and Join Product Skew (JPS) and guaranteeing perfect load balancing on homogeneous distributed Shared Nothing architecture.

In homogeneous multi-processor architectures, these algorithms are insensitive to data skew and guarantee perfect load balancing between processors during all the stages of join computation because data redistribution is carried out jointly by all processors (and not by a coordinator processor). Each processor deals with the redistribution of the data associated to a subset of the join attribute values, not necessarily its "own" values. However the performance of these algorithms degrades on heterogeneous multi-processor architectures where the load of each processor may vary in a dynamic and unpredictable way.
To this end we introduced in (Hassan and Bamha, 2008) a parallel join algorithm called *DFA-Join* (Dynamic Frequency Adaptive parallel join algorithm) to handle join queries on heterogeneous Shared Nothing architectures allowing to solve the problem of data skew while guaranteeing perfect balancing properties.
In this paper, we present a pipelined version of *DFA-Join* algorithm called *PDFA-Join* (Pipelined Dynamic frequency Adaptive join algorithm). The aim of pipelining in *PDFA-Join* is to offer flexible resource allocation and to avoid unnecessary disk input/output for intermediate join result in multi-join queries. We show that *PDFA-Join* algorithm can be applied efficiently in various parallel execution strategies making it possible to exploit not only intra-operator parallelism but also inter-operator parallelism. These algorithms are used in the objective of avoiding the effect of load imbalance due to data skew, and to reduce the communication costs due to the redistribution of the intermediate results which can lead to a significant degradation of performance.

2 LIMITATIONS OF PARALLEL EXECUTION STRATEGIES FOR MULTI-JOIN QUERIES

Parallel execution of multi-join queries depends on the execution plan of simple joins that compose it. The main difference between these strategies lies in the manner of allocating the simple joins to different processors and in the choice of an appropriate degree of parallelism (i.e. the number of processors) used to compute each simple join.

Several strategies were proposed to evaluate multi-join queries (Liu and Rundensteiner, 2005; Wilschut et al., 1995). In these strategies intra-operator, inter-operator and pipelined parallelisms can be used. These strategies are divided into four principal categories presented thereafter.

Sequential Parallel Execution is the simplest strategy to evaluate, in parallel, a multi-join query. It does not induce inter-operator parallelism. Simple joins are evaluated one after the other in a parallel way. Thus, at a given moment, one and only one simple join is computed in parallel by all the available processors.

This strategy is very restrictive and does not provide efficient resource allocation owing to the fact that a simple join cannot be started until all its operands are entirely available, and whenever a join operation is executed on a subset of processors, all the other processors remain idle until the next join operation. Moreover this strategy induces unnecessary disk Input/Output because intermediate results are written to disk and not immediately used for the next operations.

To reach acceptable performance, join algorithms used in this strategy should reduce the load imbalance between all the processors and the number of idle processors must be as small as possible.

Parallel Synchronous Execution uses in addition to intra-operator parallelism, inter-operator parallelism (Chen et al., 1992b). In this strategy several simple join operations can be computed simultaneously on disjoint sets of processors.

The parallel execution time of an operator depends on the degree of parallelism. The execution time decreases by increasing the number of processors until the arrival at a point of saturation (called optimal degree of parallelism) from which increasing the number of processors, increases the parallel execution time (Rahm, 1996; Chen et al., 1992b). The main difficulty in this strategy lies in the manner of allocating the simple joins to the available processors and in the choice of an appropriate degree of parallelism to be used for each join.

In this strategy, the objective of such allocation is to reduce the latency where the global execution time of all operators should be of the same order. This also applies to the global execution time of each operator in the same group of processors where the local computation within each group must be balanced.

This Strategy combines only intra- and inter operator parallelism in the execution of multi-join queries and does not introduce pipelined parallelism and large number of processors may remain idle if aren't used in inter-operator parallelism. This constitutes the main limitations of this strategy for flexible resource allocation in addition to unnecessary disk/input oper-

ation for intermediate join result.

Segmented Right-Deep Execution. Contrary to a parallel synchronous strategy, a *Segmented Right-Deep execution* (Chen et al., 1992a; Liu and Rundensteiner, 2005) employs, in addition to intra-operator parallelism, pipelined inter-operator parallelism which is used in the evaluation of the right-branches of the query tree.
This strategy offers more flexible resource allocation than parallel synchronous execution strategy : many joins can be computed on disjoint sets of processors to prepare hash tables for pipelined joins. Its main limitation remains in the fact that pipelined parallelism cannot be started until all the hash tables are computed. Moreover no load balancing between processors can be performed whenever pipelined parallelism begins.

Full Parallel Execution. (Wilschut et al., 1995; Liu and Rundensteiner, 2005) uses inter-operator parallelism and pipelined inter-operator parallelism in addition to intra-operator parallelism. In this strategy, all the simple joins, associated to the multi-join query, are computed simultaneously in parallel using disjoint sets of processors. Inter-operator parallelism and pipelined inter-operator parallelism are exploited according to the type of the query tree.
The effectiveness of such strategy depends on the quality of the execution plans generated during the query optimization phase and on the ability to evenly divide load between processors in the presence of skewed data.

All existing algorithms using this strategy are based on static hashing to redistribute data over the network which makes them very sensitive to data skew. Moreover pipelined parallelism cannot start until the creation of hash tables of build relations. We recall that all join algorithms used in these strategies require data redistribution of all intermediate join results (and not only tuples participating to the join result) which may induce a high cost of communication. In addition no load balancing between processors can be performed when pipelined parallelism begins, this can lead to a significant degradation of performance.

In the following section we will present *PDFA-Join* (Pipelined Dynamic Frequency Adaptive Join): a new join algorithm which can be used in different execution strategies allowing to exploit not only intra-operator but also inter-operator and pipelined parallelism. This algorithms is proved to induce a minimal cost for communication (only relevant tuples are redistributed over the network), while guaranteeing perfect load balancing properties in a heterogeneous multi-processor machine even for highly skewed data.

3 PARALLELISM IN MULTI-JOIN QUERIES USING PDFA-JOIN ALGORITHM

Pipelining was largely studied and successfully implemented in many classical join algorithms, on Shared Nothing (SN) multi-processor machine in the presence of ideal conditions of load balancing and in the absence of data skew (Liu and Rundensteiner, 2005). Nevertheless, these algorithms are generally based on static hash join techniques and are thus very sensitive to AVS and JPS.

The pipelined algorithm we introduced in (Bamha and Exbrayat, 2003) solves this problem and guarantees perfect load balancing on homogeneous SN machines. However its performance degrades on heterogeneous multi-processor architectures.

In this paper, we propose to adapt *DFA-Join* to pipelined multi-join queries to solve the problem of data skew and load imbalance between processors on heterogeneous multi-processors architectures.
DFA-Join is based on a combination of two steps :

- a static step where data buckets are assigned to each processor according to its actual capacity,
- then a dynamic step executed throughout the join computation phase to balance load between processors. When a processor finishes join processing of its assigned buckets it asks a local head node for untreated buckets of another processor.

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

To ensure the extensibility of the algorithm, processors are partitioned into disjoint sets. Each set has a designed local head node. Load is first balanced inside each set of processors and whenever a set of processors finishes its assigned tasks, it asks a head node of another set of processors for additional tasks.

3.1 Detailed Algorithm

In this section, we present a parallel pipelined execution strategy for the multi-join query, $Q = (R \bowtie_{a_1} S) \bowtie_{b_1} (U \bowtie_{a_2} V)$, given in figure 1 (this strategy can be easily generalized to any bushy multi-join query) where R, S, U and V are source relations and a_1, a_2 and b_1 are join attributes.

We will give in detail the execution steps to evaluate the join query $Q_1 = R \bowtie_{a_1} S$ (the same technique is used to evaluate $Q_2 = U \bowtie_{a_2} V$).

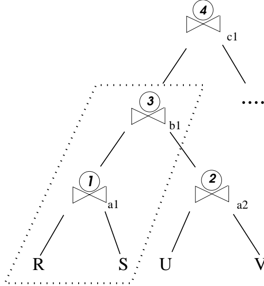

Figure 1: Parallel execution of a multi-join query using PDFA_join algorithm.

We first assume that each relation $T \in \{R, S, U, V\}$ is horizontally fragmented among p processors and:

- T_i is the fragment of T placed on processor i,
- $Hist^x(T)$ denotes the histogram[1] of T with respect to the join attribute x, i.e. a list of pairs (v, n_v) where $n_v \neq 0$ is the number of tuples of T having value v for x. The histogram is often much smaller and never larger than the relation it describes,
- $Hist^x(T_i)$ denotes the histogram of fragment T_i placed on processor i,
- $Hist_i^x(T)$ is processor i's fragment of the global histogram of relation T,
- $Hist^x(T)(v)$ is the frequency (n_v) of value v in T,
- $Hist^x(T_i)(v)$ is the frequency of value v in T_i,
- $\|T\|$ denotes the number of tuples of T, and
- $|T|$ is the size (in bytes or number of pages) of T.

Our algorithm (*Algorithm 1*) can be divided into the following five phases. We give for each phase an upper bound of the execution time using BSP (Bulk Synchronous Parallel) cost model (Skillicorn et al., 1997; Valiant, 1990). Notation $O(\ldots)$ only hides small constant factors : they depend only on the implementation, but neither on data nor on the BSP machine parameters.

Phase 1. Creating Local Histograms
In this phase, we create in parallel, on each processor i, the local histogram $Hist^{a_1}(R_i)$ (resp. $Hist^{a_1}(S_i)$) $(i = 1, \ldots, p)$ of block R_i (resp. S_i) by a linear traversal of R_i (resp. S_i) in time $\max_{i=1,\ldots,p}(c^i_{r/w} * |R_i|)$ (resp. $\max_{i=1,\ldots,p}(c^i_{r/w} * |S_i|)$) where $c^i_{r/w}$ is the cost to read/write a page of data from disk on processor i.
While creating the histograms, tuples of R_i (resp. S_i) are partitioned on the fly into N buckets using a hash function in order to facilitate the redistribution phase. The cost of this phase is:

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

Algorithm 1. Parallel PDFA-Join computation steps to evaluate the join of relations R and S on attribute a_1 and preparing the next join on attribute b_1.

In Parallel (on each processor) $i \in [1, p]$ **do**
1 ▶ Create local histograms $Hist^{a_1}(R_i)$ of R_i and, on the fly, hash the tuples of relation R_i into different buckets according to the values of join attribute a_1,
▷ Create local histograms $Hist^{a_1}(S_i)$ of S_i and, on the fly, hash the tuples of relation S_i into different buckets,
2 ▶ Create global histogram fragment's $Hist_i^{a_1}(R)$ of R,
▷ Create global histogram fragment's $Hist_i^{a_1}(S)$ of S,
▷ Merge $Hist_i^{a_1}(R)$ and $Hist_i^{a_1}(S)$ to create $Hist_i^{a_1}(R \bowtie S)$,
3 ▶ Create communication templates for only relevant tuples,
▷ Filter generated buckets to create tasks to execute on each processor according to its capacity,
▷ Create local histograms $Hist^{b_1}(R_i \bowtie S_i)$ of join result on attribute b_1 of the next join using histograms and communication templates (See Algorithm 2.),
4 ▶ Exchange data according to communication templates,
5 ▶ Execute join tasks and store join result on local disk.
Loop until no task to execute
▷ Ask a local head node for jobs from overloaded processors,
▷ Steal a job from a designated processor and execute it,
▷ Store the join result on local disk.
End Loop
End Par

$$Time_{phase1} = O\big(\max_{i=1,\ldots,p} c^i_{r/w} * (|R_i| + |S_i|)\big).$$

Phase 2. Computing the Histogram of $R \bowtie S$
In this phase, we compute $Hist_i^{a_1}(R \bowtie S)$ on each processor i. This helps in specifying the values of the join attribute that participate in the join result, so only tuples of R and S related to these values are redistributed in a further phase which allows us to minimize the communication cost. The histogram of $R \bowtie S$ is simply the intersection of $Hist^{a_1}(R)$ and $Hist^{a_1}(S)$, so we must first compute the global histograms $Hist_i^{a_1}(R)$ and $Hist_i^{a_1}(S)$ by redistributing the tuples of the local histograms using a hash function that distributes the values of the join attribute in a manner that respects the processing capacity of each processor.
The cost of this step is:

$$Time_{phase2.a} = O\Big(\min\big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist^{a_1}(R)| + \gamma_i * \|Hist^{a_1}(R)\|), \max_{i=1,\ldots,p} \omega_i * (g * |R| + \gamma_i * \|R\|)\big) + \min\big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist^{a_1}(S)| + \gamma_i * \|Hist^{a_1}(S)\|), \max_{i=1,\ldots,p} \omega_i * (g * |S| + \gamma_i * \|S\|)\big) + l\Big).$$

where ω_i is the fraction of the total volume of data assigned to processor i such that: $\omega_i = (\frac{1}{\gamma_i})/(\sum_{j=1}^{p} \frac{1}{\gamma_j})$, γ_i is the execution time of one instruction on processor i, g is the BSP communication parameter and l the cost of synchronization (Skillicorn et al., 1997;

Valiant, 1990) (review *proposition 1* of appendix A for the proof of this cost (Hassan and Bamha, 2008)).

Now we can easily create $Hist_i^{a_1}(R \bowtie S)$ by computing in parallel, on each processor i, the intersection of $Hist_i^{a_1}(R)$ and $Hist_i^{a_1}(S)$ in time of order: $Time_{phase2.b} = O\Big(\max_{i=1,\dots,p}\big(\gamma_i * \min(||Hist_i^{a_1}(R)||, ||Hist_i^{a_1}(S)||)\big)\Big)$.

While creating $Hist_i^{a_1}(R \bowtie S)$, we also store for each value $v \in Hist_i^{a_1}(R \bowtie S)$ an extra information $index(v) \in \{1,2\}$ such that:

$$\begin{cases} index(v) = 1 & \text{if } Hist^{a_1}(R)(v) \geq f_o \text{ or } Hist^{a_1}(S)(v) \geq f_o \\ & \text{(i.e. values associated to high frequencies)} \\ index(v) = 2 & \text{elsewhere (i.e. values of low frequencies).} \end{cases}$$

The used threshold frequency is $f_o = p * log(p)$.
This information will be useful in the phase of the creation of communication templates.
The total cost of this phase is the sum of $Time_{phase2.a}$ and $Time_{phase2.b}$.

We recall that the size of a histogram is, in general, very small compared to the size of base relations.

Phase 3. Creating the Communication Template
In homogeneous systems workload imbalance may be due to uneven distribution of data to be joined among the processors, while in heterogeneous systems it may be the result of allocating to processors an amount of tuples that is not proportional to actual capabilities of used machines (Gounaris, 2005).

So in order to achieve an accepted performance, the actual capacity of each machine must be taken into account while assigning the data or tasks to each processor. Another difficulty in such systems lies in the fact that available capacities of machines in multi-user systems may rapidly change after load assignment: the state of an overloaded processor may fastly become underloaded while computing the join operation and vice-versa. Thus to benefit from the processing power of such systems, we must not have idle processors while others are overloaded throughout all the join computation phase.

To this end, we use, as in *DFA-join* algorithm, a two-step (static then dynamic) load assignment approach which allows us to reduce the join processing time.

3.a. Static Load Assignment Step. In this step, we compute, in parallel on each processor i, the size of the join which may result from joining all tuples related to values $v \in Hist_i^{a_1}(R \bowtie S)$. This is simply the sum of the frequencies $Hist_i^{a_1}(R \bowtie S)(v)$ for all values v of the join attribute in $Hist_i^{a_1}(R \bowtie S)$. This value is computed by a linear traversal of $Hist_i^{a_1}(R \bowtie S)$ in time: $O\big(\max_{i=1,\dots,p} \gamma_i * ||Hist_i^{a_1}(R \bowtie S)||\big)$.

After that all processors send the computed value to a designated head node which in its turn calculates the total number of tuples in $R \bowtie S$ (i.e. $||R \bowtie S||$) by computing the sum of all received values in time of order: $O(p * g + l)$.
Now the head node uses the value of $||R \bowtie S||$ and the information received earlier 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 $\sum_{i=1}^{p} vol_i = 1$.
Finally, the head node sends to each processor i the value of $vol_i * ||R \bowtie S||$ in time of order: $O(g * p + l)$.
The cost of this step is:

$Time_{phase3.a} = O\big(\max_{i=1,\dots,p} \gamma_i * ||Hist_i^{a_1}(R \bowtie S)|| + p * g + l\big)$.

3.b. Communication Templates Creation Step. Communication templates are list of messages that constitute the relations redistribution. Owing to fact that values which could lead to AVS (those having high frequencies) are also those which may cause join product skew in "standard" hash join algorithms, we will create communication templates for only values v having high frequencies (i.e. $index(v) = 1$). Tuples associated to low frequencies (i.e. $index(v) = 2$) don't have effect neither on AVS nor on 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, $Hist_i^{a_1}(R \bowtie S)$ is 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 $index(v) = 1$ and $v \in Hist_i^{(2)}(R \bowtie S)$ if $index(v) = 2$.

This partitioning step is performed while computing $\sum_v Hist_i^{a_1}(R \bowtie S)(v)$ 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 $Hist_i^{a_1}(R \bowtie S)$ 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.

It is important to mention here that only tuples that effectively participate in the join result will be redistributed. These are the tuples of $\overline{R_i} = R_i \ltimes S$ and $\overline{S_i} = S_i \ltimes R$. These semi-joins are implicitly evaluated due to the fact that the communication templates are only created for values v that appear in join result (i.e. $v \in Hist^{a_1}(R \bowtie S)$).
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 $Hist_i^{a_1}(R \bowtie S)$, processor i creates communicating messages $order_to_send(j,i,v)$ asking each processor j holding tuples of R or S with

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 $Hist_i^{a_1}(R \bowtie S)$ [2], 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 further 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:$O\Big(\max_i(\omega_i * p * \gamma_i * ||Hist^{(1)}(R \bowtie S)||)\Big)$, 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, $order_to_send(j,i,.)$ and $order_to_save(j,.)$ messages are sent to their destination processors j when $j \neq i$ in time:

$$O\Big(\max_i(g * \omega_i * p * |Hist^{(1)}(R \bowtie S)|) + l\Big).$$

The total cost of this step is the sum of the above two costs.
3.c. Task Generation Step. After the communication templates creation, each processor i obeys the $order_to_send(.,.,.)$ messages that it has just received to generate tasks to be executed on each processor. So it partitions the tuples that must be sent to each processor into multiple number of buckets greater than p using a hash function. This partition facilitates task reallocation in the join phase (phase 5) from overloaded to idle processors if a processor could not finish its assigned load before the others. After this partition step, each bucket is sent to its destination processor. The cost of this step is : $O\big(max_{i=1,...,p}\gamma_i * (||R_i|| + ||S_i||)\big)$.

In addition, each processor i will partition tuples whose join attribute value is indicated in $order_to_save()$ messages into buckets using the same hash function on all the processors. However, these buckets will be kept for the moment in their source processors where their redistribution and join processing operations will be postponed till the dynamic phase.

During this step, local histogram of the join result, ($Hist^{b_1}(R \bowtie S)$, on attribute b_1 is created directly from $Hist^{a_1}(R_i)$ and $Hist^{a_1}(S_i)$ using Algorithm 2.

Owing to the fact that the access to the histogram (equivalent to a search in a B-tree) is performed in a constant time, the cost of the creation of the histogram of join result is : $O\big(max_{i=1,...,p}\gamma_i * ||R_i||\big)$.
The global cost for this step:

$$Time_{phase3.c} = \big(max_{i=1,...,p}\gamma_i * (||R_i|| + ||S_i||)\big).$$

Algorithm 2. Join result histogram's creation algorithm on attribute b_1.

Par (on each node) $i \in [1,p]$ **do**
$\triangleright$ $Hist^{b_1}(R_i \bowtie S_i)$=NULL (Create an empty B-tree)
For each tuple t of each bucket of relation R_i **do**
  $\triangleright$ $freq1 = Hist^{a_1}(S)(t.a_1)$ (i.e. frequency of $t.a_1$ of tuple t)
  If ($freq1 > 0$) (i.e. tuple t will be present in $R \bowtie S$) **Then**
    $\triangleright$ $freq2 = Hist^{b_1}(R_i \bowtie S_i)(t.b_1)$
    If ($freq2 > 0$) (i.e value $t.b_1 \in Hist^{b_1}(R_i \bowtie S_i)$) **Then**
      $\triangleright$ Update $Hist^{b_1}(R_i \bowtie S_i)(t.b_1) = freq1 + freq2$
    Else
      $\triangleright$ Insert a new couple $(t.b_1, freq1)$ into $Hist^{b_1}(R_i \bowtie S_i)$
    End If
  End If
End For
End Par

Phase 4. Data Redistribution
According to communication templates, 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 the global cost of this phase is:

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

Phase 5. Join Computation
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:

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

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 head node a message asking for more work. Hence, the head 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

[2] This is the case if $vol_i * ||R \bowtie S|| < \sum_v Hist_i^{a_1}(R \bowtie S)(v)$ on processor i.

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 *PDFA-Join* algorithm is:

$$Time_{PDFA-join} = O\Big(\max_{i=1,\ldots,p} c^i_{r/w} * (|R_i| + |S_i|) + l + \min\Big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist^{a_1}(R)| + \gamma_i * ||Hist^{a_1}(R)||), \max_{i=1,\ldots,p} \omega_i * (g * |R| + \gamma_i * ||R||)\Big) + \max_{i=1,\ldots,p} \gamma_i * (||R_i|| + ||S_i||) + \min\Big(\max_{i=1,\ldots,p} \omega_i * p * (g * |Hist^{a_1}(S)| + \gamma_i * ||Hist^{a_1}(S)||), \max_{i=1,\ldots,p} \omega_i * (g * |S| + \gamma_i * ||S||)\Big) + g * \max_{i=1,\ldots,p} (vol_i * (|\overline{R}| + |\overline{S}|) + \max_{i=1,\ldots,p} (\omega_i * p * (g * |Hist^{(1)}(R \bowtie S)| + \gamma_i * ||Hist^{(1)}(R \bowtie S)||)) + \max_{i=1,\ldots,p} (c^i_{r/w} * vol_i * (|\overline{R}| + |\overline{S}| + |\overline{R} \bowtie \overline{S}|))\Big).$$

Remark
Sequential evaluation of the join of 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:

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

PDFA-Join algorithm has optimal asymptotic complexity when:

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

this is due to the fact that the other terms in $Time_{PDFA-join}$ 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 $f_o = p * log(p)$).

3.2 Discussion

To understand the whole mechanism of *PDFA-Join* algorithm, we compare existing approaches (based on hashing) to our pipelined join algorithm using different execution strategies to evaluate the multi-join query $Q = (R \bowtie_{a_1} S) \bowtie_{b_1} (U \bowtie_{a_2} V)$.

A *Full Parallel* execution of DFA-Join algorithm (i.e. a basic use of DFA-Join where we do not use pipelined parallelism) requires the evaluation of $Q_1 = (R \bowtie_{a_1} S)$ and $Q_2 = (U \bowtie_{a_2} V)$ on two disjoint set of processors, the join results of Q_1 and Q_2 are then stored on the disk. The join result of query Q_1 and Q_2 are read from disk to evaluate the final join query Q.

Existing approaches allowing pipelining first start by the evaluation of the join queries Q_1 and Q_2, and then each generated tuple in query Q_1 is immediately used to build the hash table. However the join result of query Q_2 is stored on the disk.
At the end of the execution of Q_1, the join result of query Q_2 is used to probe the hash table. This induces unnecessary disk input/output. Existing approaches require data redistribution of all intermediate join result (not only relevant tuples) this may induce high communication cost. Moreover data redistribution in these algorithms is based on hashing which make them very sensitive to data skew.

In PDFA-Join algorithm, we first compute in parallel the histograms of R and S on attribute a_1, and at the same time we compute the histograms of U and V on attribute a_2. As soon as these histograms are available, we generate the communication templates for Q_1 and Q_2 and by the way the histograms of the join results of Q_1 and Q_2 on attribute b_2 are also computed. Join histograms on attribute b_2 are used to create the communication templates for Q which makes it possible to immediately use the tuples generated by Q_1 and Q_2 to evaluate the final join query Q.

PDFA-Join algorithm achieves several enhancements compared to pipelined join algorithm presented in the literature: During the creation of communication templates, we create on the fly the histograms for the next join, limiting by the way the number of accesses to data (and to the disks). Moreover data redistribution is limited to only tuples participating effectively to join result, this reduces communication costs to a minimum. Dynamic data redistribution in *PDFA-Join* makes it insensitive to data skew while guaranteeing perfect load balance during all the stages of join computation.

PDFA-Join can be used in various parallel strategies, however in the parallel construction of the histograms for source relations, we can notice that the degree of parallelism might be limited by two factors : the total number of processors available, and the original distribution of data. A simultaneous construction of two histograms on the same processor (which occurs when two relations are distributed, at least partially, over the same processors) would not be really interesting compared to a sequential construction. This intra-processor parallelism does not bring acceleration, but should not induce noticeable slowdown : histograms are generally small, and having several histograms in memory would not necessitate swapping. On the other hand, as relations use to be much bigger than the available memory, we have to access them by blocks. As a consequence, accessing one or several relations does not really matter. Our pipeline strategy will really be efficient if different join operators are executed on disjoint (or at least par-

tially disjoint) sets of processors. This brings us to limit the number of simultaneous builds. As a consequence, we have to segment our query trees, similarly to segmented right-deep trees, each segment (i.e. a set of successive joins) being started when the former is over. Once the histograms are produced for both tables, we can compute the communication template, then distribute data, and finally compute the join. Unfortunately, the computation of the communication template is the implicit barrier within the execution flow, that prohibits the use of long pipeline chains.

4 CONCLUSIONS

In this paper, we presented *PDFA-Join* a pipelined parallel join algorithm based on a dynamic data redistribution. We showed that it can be applied efficiently in various parallel execution strategies offering flexible resource allocation and reducing disks input/output of intermediate join result in the evaluation of multi-join queries. This algorithm achieves several enhancements compared to solutions suggested in the literature by reducing communication costs to only relevant tuples while guaranteeing perfect balancing properties on heterogeneous multi-processors shared nothing architectures even for highly skewed data.

The BSP cost analysis showed that the overhead related to histogram management remains very small compared to the gain it provides to avoid the effect of load imbalance due to data skew, and to reduce the communication costs due to the redistribution of the intermediate results which can lead to a significant degradation of the performance.

Our experience with the BSP cost model and the tests presented in our previous papers (Bamha and Hains, 1999; Bamha and Hains, 2000; Hassan and Bamha, 2008) prove the effectiveness of our approach compared to standard hash-join pipelined algorithms.

REFERENCES

Bamha, M. (2005). An optimal and skew-insensitive join and multi-join algorithm for distributed architectures. In *Proc. of DEXA'2005 International Conference. Copenhagen, Danemark*, pages 616–625.

Bamha, M. and Exbrayat, M. (2003). Pipelining a skew-insensitive parallel join algorithm. *Parallel Processing Letters*, 13(3), pages 317–328.

Bamha, M. and Hains, G. (2000). A skew insensitive algorithm for join and multi-join operation on Shared Nothing machines. *Proc. of DEXA'2000 International Conference*, pages 644–653, London, UK.

Bamha, M. and Hains, G. (1999). A frequency adaptive join algorithm for Shared Nothing machines. *PDCP Journal, Volume 3, Number 3, pages 333-345*.

Chen, M.-S., Lo, M. L., Yu, P. S., and Young, H. C. (1992a). Using segmented right-deep trees for the execution of pipelined hash joins. *Proc. of VLDB'92 International Conference, 1992, Vancouver, Canada*, pages 15–26.

Chen, M.-S., Yu, P. S., and Wu, K.-L. (1992b). Scheduling and processor allocation for the execution of multi-join queries. In *International Conference on Data Engineering*, pages 58–67, Los Alamos, Ca., USA.

Datta, A., Moon, B., and Thomas, H. (1998). A case for parallelism in datawarehousing and OLAP. In *Proc. of DEXA 98 International Workshop*, IEEE Computer Society, pages 226–231, Vienna.

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

Gounaris, A. (2005). Resource aware query processing on the grid. Thesis report, University of Manchester, Faculty of Engineering and Physical Sciences.

Hassan, M. A. H. and Bamha, M. (2008). Dynamic data redistribution for join queries on heterogeneous shared nothing architecture. Technical Report 2, LIFO, Université d'Orléans, France.

Hua, K. A. and Lee, C. (1991). Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of VLDB 17th International Conference*, pages 525–535, Barcelona, Catalonia, Spain.

Liu, B. and Rundensteiner, E. A. (2005). Revisiting pipelined parallelism in multi-join query processing. In *Proc. of VLDB'05 International Conference*, pages 829–840.

Lu, H., Ooi, B.-C., and Tan, K.-L. (1994). *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamos, California.

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

Rahm, E. (August 1996). Dynamic load balancing in parallel database systems. *in: Proc. EURO-PAR'96 Conference, LNCS, Springer-Verlag, Lyon*.

Schneider, D. and DeWitt, D. (1989). A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proc. of 1989 ACM SIGMOD International Conference, Portland, Oregon*, pages 110–121, New York, NY 10036, USA.

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

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

Wilschut, A. N., Flokstra, J., and Apers, P. M. (1995). Parallel evaluation of multi-join queries. In *Proc. of ACM-SIGMOD*, 24(2):115–126.