Transparent Parallelism in Query Execution

Stefan Manegold Florian Waas Martin L. Kersten

CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
⟨firstname.lastname⟩@cwi.nl

Abstract

A key assumption underlying query optimization schemes for parallel processing is that their cost models can anticipate the multitude of effects encountered during the execution phase. Unfortunately, this is rarely the case and the optimal processing is only achieved in a few situations. However, enriching cost models with further parameters increases likelihood and extent of estimate errors, thus, does not guarantee better results in general.

In this paper we address the question how to de-couple optimization and execution by transparent means of parallelism, i.e. once the optimizer determined the degree of parallelism for a group of operators, the underlying execution engine ensures optimal parallel execution without demanding any static schedule by the optimizer.

Based on an analytical framework we model both dataflow and processing environment for parallel query execution. On this ground we develop the notion of *Non-look-ahead Optimality* which reflects an execution strategy's ability of ad-hoc resource utilization. We prove a tight upper bound for the processing time of such strategies and show that they are insensitive to skew.

Finally, we model several different strategies and present an execution strategy that fulfills the desired optimality criteria. The new algorithm outperforms conventional pipeline execution substantially and is resistant against various kinds of skew as our experiments confirm.

Keywords: Parallel databases, parallel query processing, dynamic load balancing.

1 Introduction

Query optimization in parallel database systems is, following a common approach, split into two phases [9, 6]: sequential optimization and parallelization. The former involves query rewrites and join ordering to arrive at an optimal sequential *query evaluation plan (QEP)*. The latter deals with mapping a sequential QEP to a parallel execution environment leading to a parallel query execution plan, the final result (cf. Fig. 1).

So far, much research has been devoted to achieve the best possible parallelization of a given sequential plan [7, 8, 12, 2, 10]. A common approach is to incorporate as many features of the target architecture as possible into the cost model—e.g. communication costs or hardware description—and to derive a static parallel schedule based on this information [7, 3]. However, from a validation point of view increasing the number of features considered during optimization is risky and may prove a pitfall since errors in the estimates are propagated exponentially [11]. The consequence are suboptimal parallel schedules. A situation that is exacerbated by data skew—a factor that is hardly to capture sufficiently by a cost model and, therefore, deliberately ignored in most investigations.

To sum up, the execution of an optimized parallel plan is very likely to diverge from the envisioned one. The straight forward solution to enrich the optimizer with better statistics is not desirable as the maintenance costs are too high, the complexity of the optimizing process increases severely, and the quality of the result is of debatable value only since the cost estimates become more sensitive to estimate errors.

To overcome these drawbacks a transparent way of using a parallel execution environment is sought—i.e. the optimizer determines merely the degree of parallelism for a certain QEP, or part of a QEP, while the execution strategy ensures optimal resource utilization, including appropriate speedup and scaleup.

In this paper we develop such a means to transparently exploit parallelism. We give a formal model consisting of two components: a dataflow model and a processing model. The dataflow model enables a simple and uniform handling of relational algebraic operators and their data dependencies. The processing model abstracts a parallel environment as queues and processing units. Guided by this model we discuss the principles to find an optimal solution and introduce the notion of *Non-look-ahead Optimality (NLA-optimality)* as a measure of ad-hoc exploitation of pipeline parallelism: We call a processing strategy NLA-optimal if a processing unit is only idle when no unit of work can be assigned to it at this very moment. We show that the execution time of a NLA-optimal strategy is less than twice the theoretical optimum.

We use this framework to model and analyze conventional pipeline execution strategies, as given in [7], in comparison with DTE, a data parallel algorithm [14, 13]. DTE outperforms PE in most cases significantly. However, it is not yet NLA-optimal: in typical skew situations, some processors may become overloaded while others remain idle. We present DTE/R an NLA-optimal strategy which overcomes this problem by redistribution of the intermediate processing results. The redistribution adds only little overhead, but pays back significantly in almost every case of skew.

The final contribution of this work is a feasibility study to see whether the discussed concepts can be carried into effect. We present a comprehensive quantitative assessment of a prototype implementation which confirms the effectiveness of NLA-optimality. It shows that our analytical worst-case bound is rather pessimistic compared to the average performance observed.

Related Work. The problem of scheduling queries on parallel environments has attracted a lot of attention. Hasan and Motwani point out the importance of pipeline parallelism and develop near optimal scheduling heuristics with respect to minimize communication overhead [7]. Their techniques apply to a restricted class of query plans, such as star queries and paths of pipelined operators. The heuristics proposed ignore skew handling as well as intra-operator parallelism. Chekuri et al. develop a more general treatment and allow for arbitrary query plans using the same cost model [1]. Again, skew is not considered. Garofalakis and Ioannidis discuss a richer cost model and focus on shared-nothing architectures [3, 4]. Their scheduling heuristics are also based on the assumptions that no skew affects the execution. Lo et al. study constraint processor allocation for pipelined hash joins [12, 2] and extend this approach in [10] to scheduling of separate pipeline segments. For their simulation model, they assume uniformly distributed attributes and exclude skew situations.

Figure 1: Two-phase optimization approach

Road-Map. The next section contains a detailed motivation for our focus on pipeline parallelism. In Section 3, we develop a dataflow model that reflects the relational algebraic structure of the query to be processed. Section 4 describes the processing model to capture the parallel processing environment. The basic principles of the algorithms and an analysis, based on the two models, is given in Section 5. The results of our experiments are presented and discussed in Section 6. Section 7 concludes the paper.

2 Why Pipeline Parallelism?

In a pipeline parallelism scenario, only linear data-streams are considered, i.e. operators with a chained consumer-producer relationship between them. For binary operators like joins we have to materialize the second input relation beforehand and store it e.g. in the appropriate hash-table[1]. Once this is achieved the tuples of the right-most input relation can be "piped" through (cf. 2).

Our decision to strictly deploy pipeline parallelism and disregard other means of parallelism was guided by the following considerations.

1. The control-flow within a pipeline is rather simple to handle. Depending on the evaluation paradigm used, data is either processed on demand (demand-driven) or used to activate the subsequent operators (data-driven). Both cases can be handled in a *closed* way, i.e. no interrupt handling is required to jump between different processing steps like other execution strategies demand [5].

2. Intermediate results do not have to be fully materialized, i.e. stored in main-memory or on secondary storage, but are immediately processed by the succeeding operator.

3. Optimizers used in sequential databases can be re-used since the previous point also applies to sequential optimization and execution. Thus, no new optimizer design has to be undertaken.

4. Every tree-shaped QEP can be decomposed into linear (=pipeline) segments easily [10]. Hence, pipeline parallelism does not restrict to certain kinds of QEPs only.

5. For certain classes of queries, e.g. star queries, all evaluation plans are linear trees, which makes pipeline parallelism the *only* possibility to achieve parallel processing [7], at all.

6. Resource management is easy to control. Since a pipeline can grow only in one direction the memory requirements need not to be known *a priori*. Suitable length, and thereby memory requirements, are automatically determined during the construction of the single hash tables.

[1] We focus on QEPs consisting of hash-join operators only since joins are of particular interest in query processing. Note, all techniques developed in the remainder of this paper apply to arbitrary relational algebraic operators as well.

For the remainder of this paper we assume a join-QEP generated and decomposed by an optimizer into pipeline segments which will be executed one by one. Moreover, all hash-tables of the particular segment are already loaded into main-memory. These assumptions are in compliance with related work, e.g. [18, 12, 7].

3 Dataflow Model

In this Section, we develop an abstract *dataflow model* to reflect tuple streams between algebraic operators and their dependencies in a QEP. We are especially interested in a *uniform* treatment that helps avoid tedious distinctions by which operator the particular data was, and by which it is to be processed.

Let us assume a tree-shaped QEP generated by an optimizer as illustrated in Figure 2. The basic unit of data transport considered is a *tuple*. The set of all tuples occurring in a query tree is denoted by D. All tuples are unique, regardless of their contents, i.e. an algebraic operator applied to one tuple may entail none, one, or several new tuples all distinct from the original. Consequently, each tuple $d \in D$ is processed by exactly one operator.

Let $J = \{\bowtie_1, \ldots, \bowtie_m\}$ contain the join operators of a QEP. In pipeline processing, joins can be seen as unary operators [13]: A join maps each tuple of its input to a set of output tuples, i.e. $\bowtie$: $D \to 2^D$ where 2^D denotes the powerset of D. J can be ordered and thus we refer to single operators by their indexes $\{1, \ldots, m\}$. Together with tuple uniqueness a mapping $\rho : D \to \{1, \ldots, m\}$ can be defined to indicate the target operator for a tuple. The *join product* of a tuple d is the set of tuples that are entailed by d $\Phi : D \to 2^D$ with $\Phi(d) = \bowtie_{\rho(d)} (d)$.

These definitions enable a general treatment of the dataflow without distinction of cases as well as relationships between operators.

Definition 3.1 A pair (E, D_0) with $E \subset J$ and $D_0 \subset D$ is called *Pipeline Segment* of length n if

$$\bowtie_i (\Phi^{i-1}(D_0)) = \Phi^i(D_0)$$

for all $1 \leq i \leq n$. ◇

D_0 is called *input relation* and $\Phi^n(D_0)$ *result relation* of (E, D_0). Moreover, $D = \bigcup_i \Phi^i(D_0)$ holds.

4 Processing Model

This section describes a processing model for a parallel environment on the basis of two components: a set of *queues* Q, and a set of *processing units* P. Queues implement tuple sets and a partial temporal order reflecting the arrival order. Queues are not limited to FIFO-structures, since we do not make use of the queue's particular order. The cardinality—i.e. the length—of a queue q is denoted by $|q|$. Two queues are distinguished: q_{in} which implements the initial tuple set D_0 and q_{out} that contains the result relation after the processing.

4.1 Assumptions

Our model is based on the following assumptions.

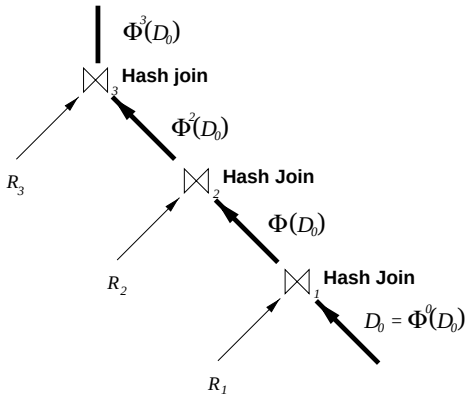

Figure 2: Pipeline Segment

A1. Uniformity of tuple loading. Each tuple takes the same time for loading when assigned to a processing unit p_i—independently of the tuple that was assigned previously to p_i, i.e. the particular tuple arrival order is irrelevant. Tuples differ only in size of the output they entail.

A2. Exclusive processing. A tuple can only be assigned to one single processing unit. A tuple that is released by a processing unit does not re-appear at any later point of time t.

A3. Non-stop processing. Once a tuple d is assigned to a processing unit, $\Phi(d)$ is generated without any delay, break or interrupt.

A4. Homogeneity of processing units. The time to generate $\Phi(d)$ is independent of the particular processing unit, i.e. all processing units achieve the same speed.

A5. Isolation of processing units. Processing units do not affect each other, even when they consume tuples from the same queue or append to the same queue at the same time.

We further assume that (1) all hash tables of a pipeline segment fit into main memory and (2) every hash-look up gives a definite answer, i.e. no further comparison is needed. These two latter assumptions are justified by our considerations of Section 2. Furthermore, similar assumptions are common in this area of research [18, 12, 2, 10]. In Section 6, we shall assess these points experimentally.

4.2 Definitions

All relationships between D, Q and P are temporal. To indicate this fact, we use a time variable index, e.g. $a \in_t A$ means that a is element of set A at t, while $|q|_t$ denotes the length of queue q at t. For convenience, we write $a \in_{[t_1,t_2]} A$ instead of $a \in_t A$, $\forall t \in [t_1, t_2]$.

Definition 4.1 Let (E, D_0) be a pipeline segment of length n, and P and Q as defined above. A *Pipeline Processing Strategy (PPS)* is a mapping

$$\mu : D \times T \rightarrow Q \cup P \cup \{\perp\}$$

with $T = [t_0, t_{stop}^{(\mu)}]$ and

1. $\forall d \in D_0 : \quad d \in_{t_0} q_{in}$,
2. $\forall d \in \Phi^n(D_0) : \quad d \in_{t^{(\mu)}_{stop}} q_{out}$,
3. $\forall d \in D : \quad \exists t_1, t_2, t_3 \in T : \mu_{[t_1,t_2]}(d) \in Q \quad \wedge \quad \mu_{[t_2,t_3]}(d) \in P$,
4. $\mu_{[t_1,t_2]}(d) \in P \quad \wedge \quad \mu_{[t'_1,t'_2]}(d') \in P,\, d' \in \Phi(d) \quad \Rightarrow \quad t_1 < t'_1$,
5. $\mu_t(d_1) = p \quad \wedge \quad \mu_t(d_2) = p \quad \Rightarrow \quad d_1 = d_2$

where $\mu_t(d) = \perp$ means, the location of d is not defined at time t. The length of T is called *execution time* of μ. ◇

The first two constraints demand, that all tuples of the initial relation D_0 are in the input queue before the processing starts, and all tuple of the result relation are elements of the output queue after the processing terminates. The third point requests the existence of an interval of time for each tuple, when it is to be processed—consisting of an interval $[t_1, t_2]$ it is stored in one of the queues and an interval $[t_2, t_3]$ it is located on a processing unit. There is no delay in between these intervals, i.e. the tuple directly goes from the queue to the processing unit. Next, a tuple d' part of the join product of tuple d cannot be processed before d. Finally, a tuple is exclusively processed, as demanded by A2.

The next issue to discuss is the work needed to process a tuple d, i.e. to generate $\Phi(d)$. It consists of two parts: probing against the hash table and generating the new result tuples. The latter includes allocation of memory or buffer space and passing the result tuple on to the next processing unit. Thus, the work for a single tuple d totals

$$w(d) = w_{probe}(d) + \sum_{d_j \in \Phi(d)} w_{pass}(d_j).$$

According to A1 and A3 we can substitute $w_{probe}(d)$ by w_{probe} and $w_{pass}(d_j)$ with w_{pass}, yielding

$$w(d) = w_{probe} + \sum_{i=1}^{|\Phi(d)|} w_{pass} = w_{probe} + |\Phi(d)| w_{pass}.$$

Because of A3 and A4 work is equal to time. Thus, the time $t^{(seq)}$ a sequential system needs for processing a particular pipeline segment is

$$t^{(seq)} = w(D) = \sum_{d \in D} w(d).$$

To handle the processor allocation at a certain time, we introduce the notion of *load*. The load of a processing unit p at time t is expressed by

$$l_t^{(\mu)}(p) = \begin{cases} 1 & \text{if } \exists d \in D : \mu_t(d) = p, \\ 0 & \text{else.} \end{cases}$$

If no ambiguity appears, the index μ is omitted in the following.

As the next example illustrates, two phases of the execution are distinguished since producer-consumer relationships between processing units may lead to idle times of some processing units:

Example 4.2 Consider a pipeline segment with only two join operators ($J = \{\bowtie_1, \bowtie_2\}$) and two processing units ($P = \{p_1, p_2\}$). Furthermore, let p_1 process $\bowtie_1$ and p_2 process $\bowtie_2$, respectively. p_2—processing only the output of p_1—stays idle until p_1 produced its first output tuple. However, p_1 may finish its work before p_2 does. •

We can generalize this case. For each processing unit p_i there exist $t_{su}^{(\mu)}(i)$ and $t_{sd}^{(\mu)}(i)$ such that

$$t \in [t_0, t_{su}^{(\mu)}(i)] \quad \Rightarrow \quad l_t^{(\mu)}(p_i) = 0$$

i.e. some processors may be idle before $t_{su}^{(\mu)}(i)$, and

$$t \in [t_{sd}^{(\mu)}(i), t_{stop}^{(\mu)}] \quad \Rightarrow \quad l_t^{(\mu)}(p_i) = 0$$

i.e. some processors finish their work earlier than others. This leads to the following definition:

Definition 4.3 For a PPS μ with D, T, P and Q as defined above the interval $[t_0, t_{su}^{(\mu)}]$ with

$$t_{su}^{(\mu)} = \max_i \{t_{su}^{(\mu)}(i)\}$$

is called *startup delay* of μ. Analogously, the interval $[t_{sd}^{(\mu)}, t_{stop}^{(\mu)}]$ with

$$t_{sd}^{(\mu)} = \min_i \{t_{sd}^{(\mu)}(i)\}$$

is called *shutdown delay.* ◇

As the Example shows, shutdown delay cannot be avoided, in general. It rather depends on the particular data. It furthermore suggests that a strategy can only be optimal if all work entailed per tuple can be assessed *a priori*, precisely. Knowledge that is not available beforehand but only after the join evaluation. Without this look-ahead a strategy can only schedule the currently available units of work to the best of its abilities.

Definition 4.4 A PPS μ is *NLA-optimal* in an interval T iff for all $[t_1, t_2] \subseteq T$

$$\sum_i l_{[t_1,t_2]}(p_i) < |P| \quad \Rightarrow \quad \int_{t_1}^{t_2} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \, dt = 0. \tag{1}$$

◇

The left expression holds if during the entire interval at least one processing unit is idle—not necessarily the same one all the time. The right expression allows no interval of length greater than zero during which a tuple is element of a queue, otherwise the integral cannot equal 0.

Theorem 4.5 Let μ be a NLA-optimal PPS and (E, D_0) a pipeline segment. The following holds[2]:

a) The execution time $t^{(\mu)}$ is bound by

$$t^{(\mu)} = 2 - \frac{1}{r} t^{(opt)}$$

[2]The proofs of all propositions and theorems presented in this paper can be found in the appendices.

where r denotes the number of available processing units, and $t^{(opt)}$ is the optimal processing time.

b) Both $t^{(opt)}$ and $t^{(\mu)}$ are bound by $t^{(seq)}$ only.

5 Pipeline Processing Strategies

This section is devoted to the description and analysis of different parallelization strategies. To facilitate the use of μ, we introduce the macro *occupyProc* as a short hand for the following situation:

$$\begin{aligned} occupyProc(t,p,q',q'') \;:=\; \exists d \in D: \quad & \exists t', \mu(d,[t',t]) = q', \\ & \mu(d,[t,t+w(d)]) = p, \\ & \forall d' \in \Phi(d), \exists t'' : \mu(d',t'') = q'' \end{aligned}$$

The right hand side comprises three parts: First, there is some data in queue q' before t. Second, d is allocated to processing unit p, thus the join for d is computed. And last, all output data is appended to queue q''.

5.1 Conventional Pipeline Execution

In conventional *Pipeline Execution (PE)*, groups of processing units are assigned to single join operators statically [7]. So, each processing unit can process tuples for one particular join operator only (cf. Expl. 4.2).

We distinguish the two cases where either the number of joins n equals the number of processing units r, or n exceeds r.

Case 1: $n = r$

Every processing unit p_i is assigned to exactly one join, consumes tuples from an exclusive input queue q_i, and appends its entire output to the input queue q_{i+1} of the subsequent processing unit. In case there is no successor, the output is appended to the output queue. For this case, PE is specified by

$$\exists t, i : l_t(p_i) = 0 \wedge |q_i| > 0 \;\Rightarrow\; occupyProc(t, p_i, q_i, q_{i+1}) \tag{I.1}$$

with $Q = \{q_{in} = q_1, \ldots, q_{n+1} = q_{out}\}$.

Case 2: $n < r$

In this case, processing units are grouped and every group is assigned to one join. We address processing units as p_{ij} where i denotes the group and j is the index within the group. r_i is the number of processing units in group i with $r_i \geq 1$ for all i. Queues are defined as before, i.e. $Q = \{q_{in} = q_1, \ldots, q_{n+1} = q_{out}\}$.

For the choice of $(r_1, \ldots, r_n) \in \mathbb{N}^n$—i.e. how to distribute the $r-n$ surplus processing units—the corresponding load balancing problem has to be considered: The work ν_i of group i is given by

$$\nu_i = w(\Phi^{i-1}(D_0)).$$

Optimal load balancing demands that each processor is doing the r-th part of the total work, i.e. for two groups i and k

$$\frac{\nu_i}{r_i} = \frac{\nu}{r} \quad \Leftrightarrow \quad \frac{\nu_i}{\nu_k} = \frac{r_i}{r_k}$$

with $r = \sum_i r_i$ and $\nu = \sum_i \nu_i$ holds. This load balancing problem has no integer solution, in general. Thus, only an approximation can be given. The problem of finding a vector $(r_1, \ldots, r_n) \in \mathbb{N}^n$ with $r_i \geq 1$ and $\sum_i r_i = r$ such that

$$F = \max_i\{\frac{\nu_i}{r_i}\} - \frac{\nu}{r}$$

is minimal is called *Processing unit Assignment Problem (PAP)*. As $t^{(PE)}$ is dominated by the execution time of the slowest operator in the pipeline, i.e. $\max_i\{\frac{\nu_i}{r_i}\}$, minimizing F means minimizing $t^{(PE)}$. In Appendix B we give an algorithm that provably minimizes F for a given pipeline segment.

For the remainder of this Section we assume a grouping of processing units such that F is minimal. We can describe PE by:

$$\exists t, i, j : l_t(p_{ij}) = 0 \wedge |q_i| > 0 \;\Rightarrow\; occupyProc(t, p_{ij}, q_i, q_{i+1}). \tag{I.2}$$

For simplicity we omit a treatment of the case that more than one processing unit/queue pair matches the condition. For more detailed invariants capturing this ambiguities, also for further invariants presented below, we refer the reader to [15].

Corollary 5.1 PE is not NLA-optimal if there is no integer solution to the associated PAP.

In this case, the performance of PE suffers from the *discretization error* [18, 16].

Theorem 5.2 Let (E, D_0) be a pipeline segment of length n. PE is NLA-optimal iff $n = 1$.

However, cutting a QEP into segments of length 1 would give away all advantages of pipeline processing as pointed out in Section 2.

5.2 Data Threaded Execution

With *Data Threaded Execution (DTE)* every processing unit p_i has shared access to q_{in} and exclusive access to a local queue q_i. In contrast to PE the p_i are not only performing one but every join operator in a pipeline segment, as necessary. Every processing unit appends all the output to its local queue—unless no further processing is required—from which it preferably consumes tuples. Only in case the local queue is empty, tuples from q_{in} are consumed. DTE is described by the following two invariants:

$$\exists t, i : l_t(p_i) = 0 \wedge |q_i| > 0 \;\Rightarrow\; occupyProc(t, p_i, q_i, q') \tag{I.3}$$

and

$$\exists t, i : l_t(p_i) = 0 \wedge |q_i| = 0 \wedge |q_{in}| > 0 \;\Rightarrow\; occupyProc(t, p_i, q_{in}, q') \tag{I.4}$$

where $q' = q_i$ if $\rho(d) \leq n$ and q_{out} otherwise.

Example 5.3 Let (E, D_0) be a pipeline segment of length $n = 2$, $|P| = 4$, and $|D_0| = 3$. Furthermore, $|\Phi(d)| = 10^2, d \in D_0$ and $|\Phi(d')| = 10^2, d' \in \Phi(D_0)$, i.e. in each join, every tuple finds 10^2 partners.

Processing this pipeline segment with DTE leads to the following situation: the 3 tuples of q_{in} are assigned to 3 out of 4 processing units, say, p_1, p_2 and p_3. All further processing is performed by these three processing units because of their local queues, i.e. $|q_i|, i \in \{1,2,3\}$ increases to 10^2 while $|q_4| = 0$ and p_4 is idle throughout the entire processing. So, 25% of the available processing resources are wasted. •

Example 5.3 shows that even in situations that offer a high potential of parallelism, DTE may yield results of only poor quality.

Theorem 5.4 Let (E, D_0) be a pipeline segment of length n. DTE is NLA-optimal

a) if $n = 1$.

b) during $[t_0, t_{sd}^{(DTE)}]$ or

For $t > t_{sd}^{(PE)}$ a situation as described in Example 5.3 may occur.

5.3 Data Threaded Execution with Redistribution

DTE/R overcomes this drawback by redistribution of intermediate results. Every processing unit appends all its output to the input queue q_{in} unless no further processing is required. The following invariant specifies DTE/R

$$\exists t, i : l_t(p_i) = 0 \wedge |q_{in}| > 0 \Rightarrow occupyProc(t, p_i, q_{in}, q') \tag{I.5}$$

where $q' = q_{in}$ if $\rho(d) \leq n$ and q_{out} otherwise.

Theorem 5.5 DTE/R is NLA-optimal for any pipeline segment (E, D_0).

6 Quantitative Assessment

To verify the analytical results, we implemented the above mentioned strategies and carried out experiments on an SGI Origin2000 shared-memory machine with 16 processors and 8 GB main memory. The control-flow of each processing unit is realized by a thread.

Our hash table implementation reaches a performance comparable to the Berkeley Hash/DB-package, as preliminary experiments showed.

As the description of the algorithms suggests, different processing units share common queues, which may lead to locking conflicts if implemented in a naive way. We circumvented such shortcomings by a more sophisticated queue design including evasion mechanisms to avoid lock conflicts largely[3]. The more complex queue implementation adds some extra overhead, but pays back by avoiding unnecessary locks and enabling a higher degree of parallelism instantly.

For each experiment, we first generate the base relations according to the query specifications, then build the hash tables sequentially and after that, execute the strategy considered. To obtain stable results we took the median of 10 runs.

[3]For a detailed description of this issue see [15].

REGION	NATION	SUPPLIER	CUSTOMER	PART	PARTSUPP	ORDER	LINEITEM
5	25	10k	150k	200k	800k	1500k	6000k

Table 1: TPC-D Benchmark: relations & cardinalities

6.1 TPC-D Benchmark

The initial set of experiments is a subset of the TPC-D Benchmark [17]. According to the focus of this paper, we execute only the join pipelines leaving out the aggregations. For each query, we generated several right-deep query trees using different join orders.

Figure 3 depicts the relative execution times $\frac{t^{(\mu)}(r)}{t^{(DTE/R)}(r)}$ (where μ is PE, DTE, or DTE/R) using $1 \leq r \leq 16$ processors for query Q5. The join order is (REGION $\bowtie$ (NATION $\bowtie$ (SUPPLIER $\bowtie$ (LINEITEM $\bowtie$ (ORDER $\bowtie$ CUSTOMER))))), i.e. CUSTOMER is the right-most input relation. DTE performs slightly ($\leq 5\%$) better than DTE/R, as the simple local queues of DTE cause less overhead than the complex global queue of DTE/R. PE performs significantly worse than the other strategies.

Figure 4 depicts the speedup behavior $\frac{t^{(DTE/R)}(1)}{t^{(\mu)}(r)}$ of all strategies for query Q5. DTE and DTE/R both achieve near-linear speedup, while PE performs significantly worse suffering from discretization errors.

Figures 5 and 6 depict relative execution times and speedup behavior for the same query (Q5), but with reverse join order (CUSTOMER $\bowtie$ (ORDER $\bowtie$ (LINEITEM $\bowtie$ (SUPPLIER $\bowtie$ (NATION $\bowtie$ REGION))))), i.e. with REGION as right-most input relation. PE—still suffering from discretization errors—performs significantly worse than DTE/R that again achieves near-linear speedup. The behavior of DTE, however, is completely different than in the previous experiment. Using up to 5 processors, the execution time is similar to that of DTE/R, but then it gets the worse the more processors are used. The speedup is limited to 5 processors, which means that the execution time does not decrease using more than 5 processors. The reason for this is that there are only 5 tuples in relation REGION. Thus, at most 5 processors can participate in executing the pipeline segment while any additional processor stays idle. This situation is similar to the one described in Example 5.3.

Finally, Figures 7 and 8 depict relative execution times and speedup behavior for query Q7 with join order (NATION $\bowtie$ (SUPPLIER $\bowtie$ (LINEITEM $\bowtie$ (ORDER $\bowtie$ (CUSTOMER $\bowtie$ NATION))))). Here, DTE achieves nearly the same performance and speedup as DTE/R whenever $r = \lceil \frac{25}{x} \rceil, x \in \mathbb{N}$ holds, but performs worse in any other case.

Further experiments showed, that the different behavior of DTE and DTE/R as described above is triggered only by the cardinality of the right-most input relation, here. The remaining join ordering has the same impact on both DTE and DTE/R as all intermediate results consist of more tuples than there are processors. We will discuss this in more detail in the next section.

6.2 Extreme Skew

Example 5.3 and the experiments above demonstrate that DTE may perform worse than DTE/R as DTE is not NLA-optimal on $[t_{sd}^{(DTE)}, t_{stop}^{(DTE)}]$. Of course, the difference between both strategies depends on the severity of skew. To examine the impact of skew quantitatively, we used a two-join-query with base relations of equal size (240k tuples). We varied the skew by choosing attribute value distributions and ranges of join attributes so that:

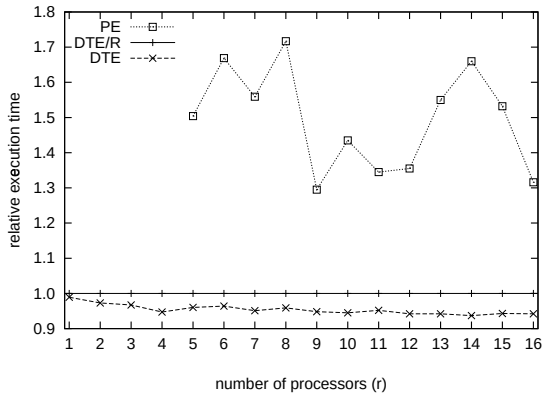

Figure 3: Relative performance $Q5_{(CUSTOMER)}$

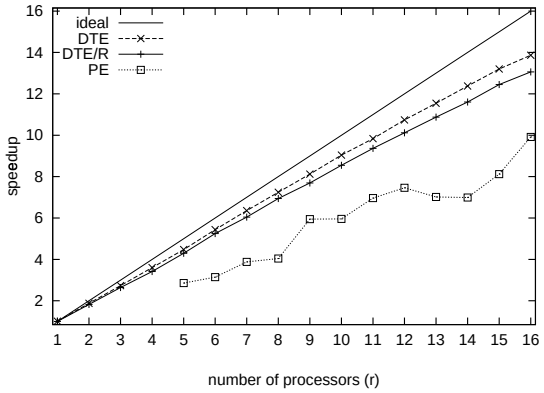

Figure 4: Speedup $Q5_{(CUSTOMER)}$

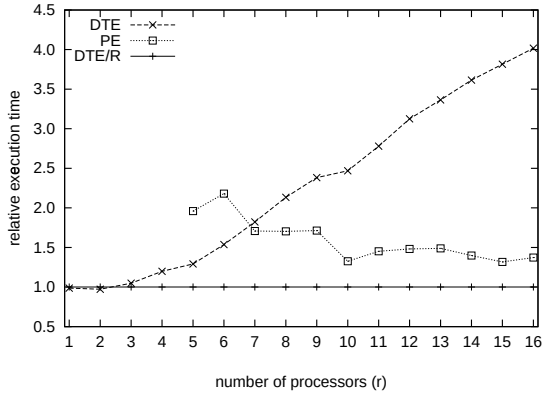

Figure 5: Relative performance $Q5_{(REGION)}$

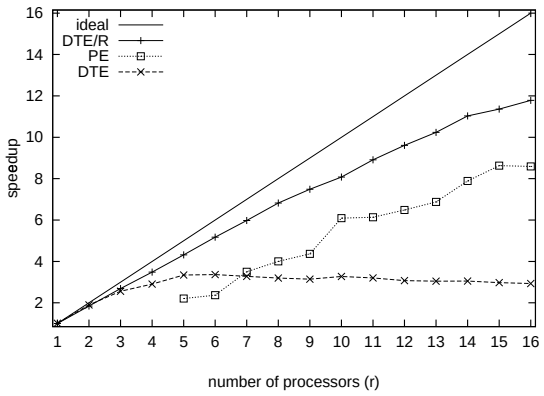

Figure 6: Speedup $Q5_{(REGION)}$

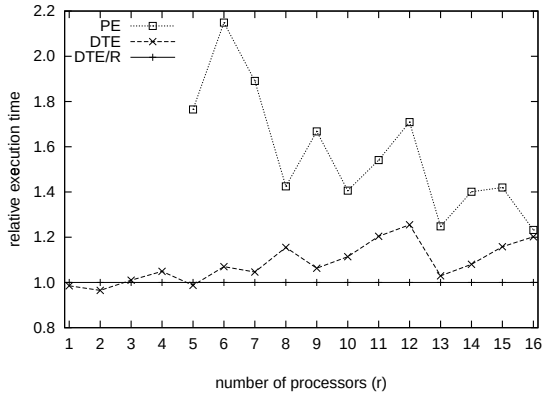

Figure 7: Relative performance $Q7_{(NATION)}$

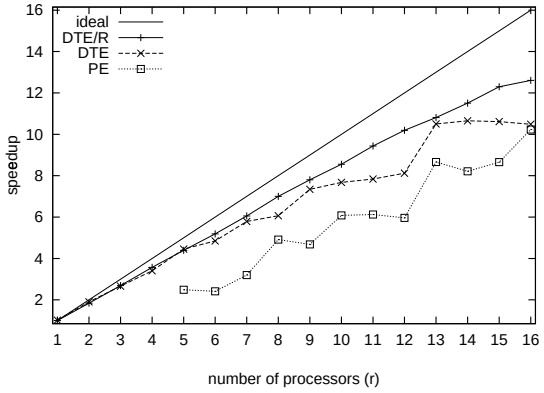

Figure 8: Speedup $Q7_{(NATION)}$

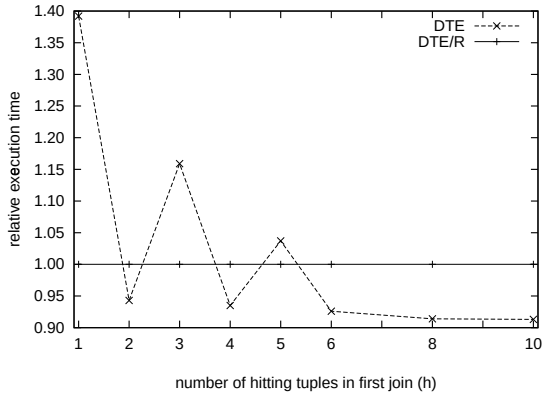

Figure 9: Skew on 2 processors

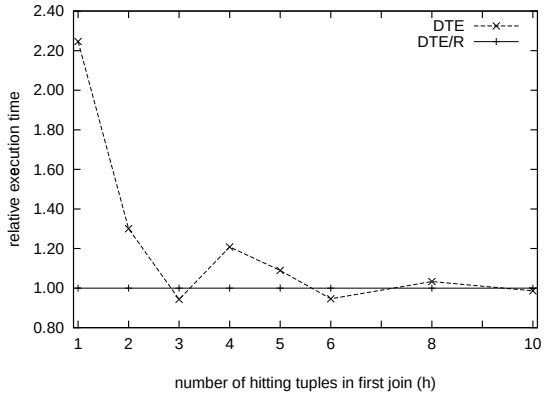

Figure 10: Skew on 3 processors

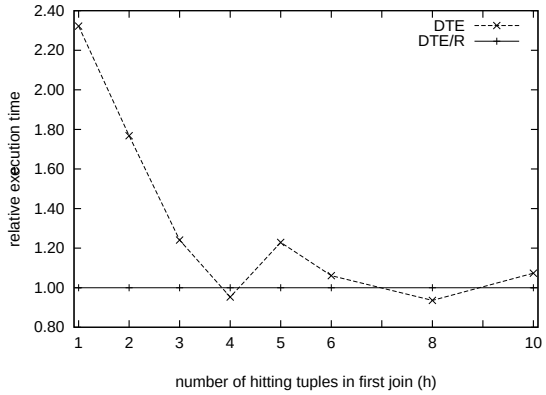

Figure 11: Skew on 4 processors

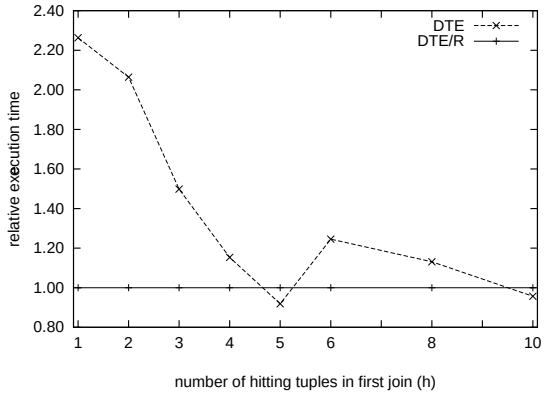

Figure 12: Skew on 5 processors

- In the first join, h input tuples hit and find $f = \frac{240,000}{h}$ tuples each. Between two subsequent hitting tuples $f - 1$ tuples find no partner, i.e. every f-th input tuple finds f partners. For instance, with $h = 2$ the 120,000th and the 240,000th, i.e. last, tuple hit and entail 120,000 output tuples each.

- The second join produces exactly one output tuple from each input tuple.

The parameter h provides a kind of metric for the skew: the smaller h is, the greater the skew is. In our experiments, we varied h from 1 to 10. Figures 9 through 12 show the relative execution times on two to five processors, respectively.

With $h = 1$, DTE performs significantly worse than DTE/R. First, all processors are involved in executing the first join, i.e. just probing the input tuples against the first hash table without finding any partner. Only the last input tuple finds partners, which then have to be processed through the second join by one thread only. With DTE/R, however, the tuples are re-distributed and all processors participate in executing the second join.

With increasing h, i.e. decreasing skew, the differences between the strategies become smaller as DTE achieves better load balancing, then. DTE even beats DTE/R, whenever h is a multiple of the

number of processors.

7 Conclusion

In this paper we presented a formal framework to model and analyze pipeline processing strategies for parallel query execution. The key notion developed is the concept of NLA-optimality which reflects the execution strategy's ability of ad-hoc resource utilization. Strategies fulfilling the stated optimality criteria guarantee a near-optimal query execution. We presented a thorough analysis of the standard execution technique and data parallel algorithms of which one achieves NLA-optimality.

The analytical approach taken has been verified by a comprehensive quantitative assessment of the different techniques. The experiments distinctly exhibit the effectiveness of our approach showing that NLA-optimal strategies yield near-linear speedup and are resistant against various kinds of skew.

The concepts presented have direct application to existing database systems. DTE/R can be used as one single multi-way join operator within a larger QEP: providing a transparent interface to parallel execution that does not require any scheduling. Moreover, DTE/R can also be used in this way to easily boost sequential query evaluation systems put on a parallel platform by using a fixed maximal degree of parallelism.

The promising results obtained raise the question whether parts of the pre-processing phase can also be efficiently integrated with NLA-optimal strategies. In addition, we intend to address transparent parallelism in the context of multi-query optimization.

References

[1] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–265, San Jose, CA, USA, May 1995.

[2] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 15–26, Vancouver, BC, Canada, August 1992.

[3] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 365–376, Montreal, Canada, June 1996.

[4] M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 296–305, Athens, Greece, September 1997.

[5] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 749–764, Atlantic City, NJ, USA, May 1990.

[6] W. Hasan, D. Florescu, and P. Valduriez. Open Issues in Parallel Query Optimization. *ACM SIGMOD Record*, 25(3):28–33, September 1996.

[7] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelining Parallelism. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 36–47, Santiago, Chile, September 1994.

[8] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 239–250, Zurich, Switzerland, September 1995.

[9] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proc. of the Int'l. Conf. on Parallel and Distr. Inf. Sys.*, pages 9–32, Miami Beach, FL, USA, December 1991.

[10] H.-I. Hsiao, M.-S. Chen, and P. S. Yu. On Parallel Execution of Multiple Pipelined Hash Joins. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 185–196, Minneapolis, MN, USA, May 1994.

[11] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–277, Denver, CO, USA, May 1991.

[12] M.-L. Lo, M.-S. Chen, C. V. Ravishankar, and P. S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 69–78, Washington, DC, USA, May 1993.

[13] S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments. In *Proc. of the European Conf. on Parallel Processing*, pages 1117–1124, Passau, Germany, August 1997.

[14] S. Manegold, J. K. Obermaier, F. Waas, and J.-C. Freytag. Data Threaded Query Evaluation in Shared-Everything Environments. Technical Report HUB-IB-58, Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany, April 1996.

[15] S. Manegold, F. Waas, and M. L. Kersten. On Optimal Pipeline Processing in Parallel Query Optimization. Technical Report INS-R9805, CWI, Amsterdam, The Netherlands, February 1998.

[16] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Proc. of the Int'l. Conf. on Parallel and Distr. Inf. Sys.*, pages 84–92, San Diego, CA, USA, January 1993.

[17] Transaction Processing Performance Council, San Jose, CA, USA. *TPC Benchmark D (Decision Support)*, Revision 1.3.1, 1998.

[18] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 115–126, San Jose, CA, USA, May 1995.

A Proofs

A.1 Proof of Theorem 4.5

Let w denote the total amount of work $w = w(D)$. In case all units of work are independent it is easy to see that it takes the optimal PPS the processing time of $\frac{w}{r}$ with r is the number of

processing units. Now, let $\lambda \in [0,1]$ be the largest ratio of work that cannot be parallelized due to data dependencies. Obviously, as long as $\lambda \leq \frac{1}{r}$ holds, the optimal parallelization is still feasible. However, in case $\lambda \geq \frac{1}{r}$, $t^{(opt)}$ is bound by λw:

$$t^{(opt)} = \begin{cases} \lambda w & \text{if } \lambda > \frac{1}{r} \\ \frac{w}{r} & \text{else} \end{cases}$$

For the further considerations let us have a look at an example: For $D = \{d_1, \ldots, d_{16}\}$ we assume $w(d_i) = 1$ and data dependencies be $d_{13}, d_{14}, d_{15}, d_{16}$ such that $d_{14} = \Phi(d_{13})$ etc. Thus, λ computes to $\frac{1}{4}$. A possible schedule with processing time 7 is depicted in Figure 13 (the four tuple that have to be processed one after the other are grey). On the other hand, the shortest possible thus optimal schedule requires only 4 time units for execution as Figure 14 shows. However, without this look-ahead on how much work a tuple will cause a PPS μ is bound by

$$t^{(\mu)} \leq \frac{(1-\lambda)w}{r} + \lambda w.$$

The ratio of $t^{(\mu)}$ to $t^{(opt)}$ in dependency of λ is

$$W(\lambda) = \frac{t^{(\mu)}}{t^{(opt)}} \leq \begin{cases} \frac{1}{r\lambda} - \frac{1}{r} + 1 & \text{if } \lambda > \frac{1}{r}, \\ r\lambda - \lambda + 1 & \text{else.} \end{cases}$$

For λ approaching $\frac{1}{r}$

$$\lim_{\lambda \leq \frac{1}{r}} \frac{1}{r\lambda} - \frac{1}{r} + 1 = 2 - \frac{1}{r}$$

increases monotonically as well as

$$\lim_{\lambda \geq \frac{1}{r}} r\lambda - \lambda + 1 = 2 - \frac{1}{r}$$

does. Thus, W is both continuous and maximal in $\lambda = \frac{1}{r}$. Leading to

$$W(\lambda) \leq 2 - \frac{1}{r}$$

To proof the general upper bound we consider the case $\lambda = 1$. No parallelism can be exploited and the upper bound for $t^{(\mu)}$ as well as for $t^{(opt)}$ is $t^{(seq)}$. □

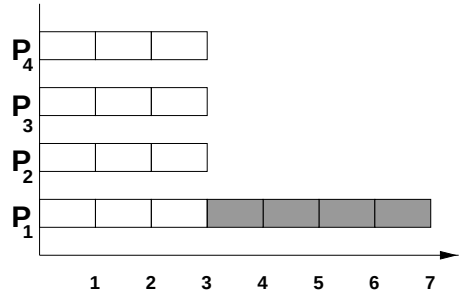

Figure 13: Suboptimal schedule

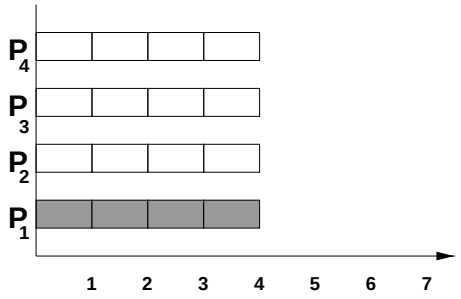

Figure 14: Optimal schedule

A.2 Proof of Theorem 5.2

Case 1: $n = 1$
The situation where $l_t(p_i) = 0$ for some i occurs only if $|q_{in}| = 0$. As q_{in} is the only queue in $Q \setminus \{q_{out}\}$ the theorem holds.

Case 2: $n > 1$
Consider a input relation D_0 such that $\Phi^i(D_0) \neq \emptyset$ for all i. PE has a minimum startup of

$$t_{su}^{(PE)} \geq (n-1)(w_{probe} + w_{pass})$$

during which at least one processing unit is idle but $|q_{in}| > 0$. □

A.3 Proof of Theorem 5.4

For $n = 1$ see Theorem 5.2.
Let $n > 1$. As long as $|q_{in}|_t > 0$ Condition I.3 implies $l_t(p_i) = 1$. The length of q_{in} is monotonically decreasing and

$$t_{sd}^{(DTE)} = \min\{t : |q_{in}|_t = 0\}$$

Which completes the proof. □

A.4 Proof of Theorem 5.5

Assume

$$\sum_i l_{[t_1,t_2]}(p_i) < |P|$$

holds for an interval $[t_1, t_2]$ with $t_2 - t_1 > 0$. For all intervals $[u_i, v_i] \subset [t_1, t_2]$ where $|q_{in}|_t > 0$ with $t \in \cup_i [u_i, v_i]$

$$u_i = v_i$$

immediately follows because of Invariant I.5. Consequently,

$$\int_{t_1}^{t_2} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \; dt = \int_{\cup_i [u_i, v_i]} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \; dt + \int_{T \setminus \cup_i [u_i, v_i]} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \; dt = 0$$

since all intervals $[u_i, v_i]$ are of empty size. □

B Algorithm for the PAP

To obtain a discrete approximation of the PAP, we first assign one single processing unit to each join and then gradually add processing units always to the currently slowest join until all processing units are distributed (see Figure 15). The overall execution time—dominated by the slowest join—is step by step reduced as far as possible.

input: $n, r, (\nu_1, \ldots, \nu_n)$;
output: $(r_1, \ldots, r_n)$;

for $i \leftarrow 1$ **to** n **do**
  $r_i \leftarrow 1$;
od;
$r_{avail} \leftarrow r - n$;
while $r_{avail} > 0$ **do**
  find smallest j with $\frac{\nu_j}{r_j} = \max_i \left\{ \frac{\nu_i}{r_i} \right\}$;
  $r_j \leftarrow r_j + 1$;
  $r_{avail} \leftarrow r_{avail} - 1$;
od.

Figure 15: Algorithm to assign r processing units to n joins

Obviously, this algorithm minimizes $\max_i \{\frac{\nu_i}{r_i}\}$ and hence

$$F = \max_i \{\frac{\nu_i}{r_i}\} - \frac{\nu}{r} \qquad \text{(cf. Section 5.1)}$$

if the PAP has no integer solution. In case the PAP has an integer solution, the algorithm finds the exact solution as the r_i are increased, and hence all $\frac{\nu_i}{r_i}$ are decreased, until $\frac{\nu_i}{r_i} = \frac{\nu}{r}$ holds for all i.