From the modeling of parallel query processing to the query optimization and simulation

Lionel Brunie, Harald Kosch and Wolfgang Wohner
(Lionel.Brunie,Harald.Kosch,Wolfgang.Wohner)@lip.ens-lyon.fr
LIP – CNRS
Ecole Normale Supérieure de Lyon
46, allée d'Italie, F - 69364 Lyon Cédex 07, France
tel +33/72 72 85 03 – fax +33/72 72 80 80

Abstract

This paper presents a novel theoretical model for representing parallel relational query processing. It is based on a hierarchical approach. First, a scheme graph, called *DPL graph*, describes all possible execution dependencies between operators, including communication and run-time control mechanisms. Second, a *high-level Petri net* is used for modeling the data- and control flow. Our model provides the framework for building and studying parallel database tools and applications. Thus, we implemented a parallel query optimizer based on *DPL graphs*, which is able to access sub-search spaces not yet considered. Furthermore, based on this model, a simulation environment has been designed and implemented for testing run-time control strategies as well as query optimization methods.

Key words : Parallel databases, theoretical model for representing parallel query processing, query optimization, run-time simulations.

1 Introduction

With the emergence of decision support systems, relational queries tend to become more and more complex. For instance, latest results (July 1996) of Teradata for the TPC-D benchmark [1], implementing complex queries until a 6-ways join with a database size of 100GB and using 5 processors, showed response times over two thousands of a second.

This complexity renders the classical query representation models obsolete because they do not provide a sufficiently accurate view of the actual parallel execution. In that context, this paper presents a novel theoretical model for representing parallel relational query processing, allowing a very precise description of the implementation techniques and control strategies used in parallel query processing.

This model is based on a hierarchical approach. First, a scheme graph called **DPL graph** is introduced, which represents the "scenario" of the parallel execution. Based on an analysis of the execution dependencies between operators, this graph structure allows one to deal with most parallel architectures and most parallel execution strategies. Stress has especially been put on the modeling of communications and run-time constraints. Second, a **High-level Petri net** is used for modeling the dynamics, i.e. the data and control flows, of *DPL graphs*.

DPL graphs, completed by a *high-level Petri nets* description, can provide the framework for building and studying parallel database tools and applications. Thus, we implemented a parallel query optimizer based on *DPL graphs*.

We chose this formalism because *DPL graphs* allow a very accurate representation of most strategies for the handling of data, task or pipeline parallelism, integrating communication and run-time considerations into the graph itself. Therefore, our parallel query optimizer can access sub-search spaces, containing low-cost processing strategies, not yet considered in this context (parallel optimizer described in section 4). In particular we introduce an original class of execution strategies : the *serialized bushy trees*. Furthermore, we designed a simulation environment for testing run-time control strategies as well as query optimization methods (simulation tool described in section 6). This simulator implements *DPL graphs* and the related *High-level Petri net* on top of a Petri-net simulation software.

This paper is organized as follows. Section 2 proposes a brief overview of previous works. Section 3 presents the basic concept of *DPL graphs*. Then, section 5 describes how *high-level Petri nets* can model the data and control flows in *DPL graphs*. Section 4 presents how the *DPL graphs* have been used for implementing our parallel query optimizer. In section 6 we show how the flexibility of the *DPL graphs* and their related *High-level Petri nets* have been used for implementing a parallel query processing simulation environment. Finally, section 7 concludes this paper and points out future developments.

2 Related works

Query parallelization strategies are usually represented by so called **query processing trees** [2] : *The leaves of a* ***query processing tree*** *represent the base relations that participate in the query and intermediate nodes model operations. These latters receive their input relations via the incoming edges and send the result relation through the outgoing edge to the next operation. The root of the tree produces the result of the whole query.*

Processing trees have been chosen as models for representing parallel query execution since they allow expressing different kinds of parallelism :

inter-operation parallelism : Operations lying on different paths of a query processing tree can be executed concurrently.

intra-operation parallelism : Each relational operation can be decomposed into several sub-operations to be executed on different partitions of the same relation.

pipeline parallelism : Two nodes lying on the same edge can be executed concurrently. Suppose for instance that a selection operator is to be performed on the output of a join operator. Clearly the selection operator can start its job as soon as the first tuple has been processed by the join operator and then work in parallel with that latter.

The models based on this approach suffer from the fact that they do not provide appropriate representations of all the kinds of parallelization strategies. Thus, most models, e.g. [3, 4, 5, 6], only consider data or simple precedence dependencies between operators. This prevents from taking into account some possible processing strategies like, for instance, those in which operators are ordered without reference to a data streams. Furthermore, none of these models deal correctly with the algorithms based on bucket processing. This technique is applied when relation partitions cannot fit into the main memory of the processors. Instead of working on the whole partitions, algorithms implementing this approach work on portions of partition called *buckets* [7]. In section 3.2 we will see that a correct modeling of such algorithms requires to introduce a novel form of dependency, the *loop dependency*.

As far as we know, only two major works have proposed to model the control- and data flows associated with an execution scenario.

In the Bubba project [8], a control protocol is associated with every operator. This protocol, compiled into the execution scenario, specifies the run-time execution of the operator. However, this framework only allows to represent simple data-dependency between operators. For instance, pipeline parallelism cannot be considered.

The modeling of the control schema in DBS3 ([9, 10]) is based on the propagation of termination messages and triggers of operators. Two kinds of triggers are considered, sequential and pipeline triggers. Although this framework is more sophisticated than Bubba's, it suffers from the fact that it is completely static, i.e. it does not integrate dynamic control schemes.

In our approach we represent the control and data flow with the help of *high-level Petri nets*. This representation model is much more compact and is based on a well-founded theoretical background. Furthermore, the notion of conditioned places allows the adaption of the control flow to the actual run-time context.

3 DPL graphs : a novel representation model for the execution scenario

This section introduces the complete notion of *DPL graphs*. First, the operator vertices are presented. Then we illustrate why it is necessary to generalize the notion of the related works, in order to achieve a more accurate representation of most parallelization strategies. *DPL graphs* were partially introduced in [11], but not yet fully developed.

The section is based on the study of the optimization of a sample query $R_1 \bowtie R_2 \bowtie R_3$ executed on a shared nothing system. Relations are supposed to be already partitioned on the join attribute over all the disks. The intermediate result relation $R_2 \bowtie R_3$ must be repartitioned. The two join operators are implemented using hash-based algorithms, whereby the two hash tables are built on the base relations R_2 and R_1.

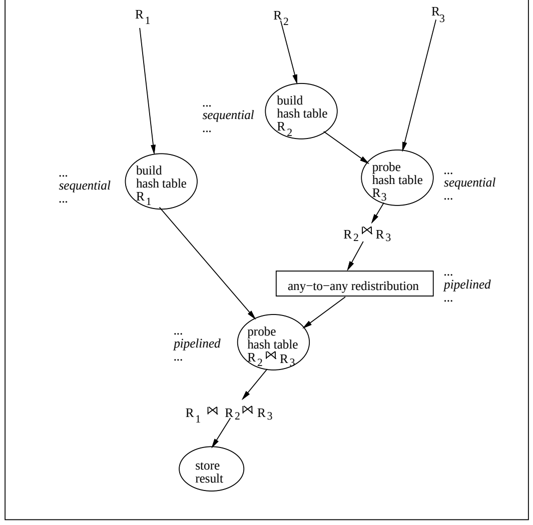
3.1 Operator vertex : basic, communication and control operators

The vertices within a *DPL graph* represent various operators which can be divided into three different categories. First the **basic operators** are atomic operators working on relation partitions. They are part of the implementation of a relational operator. These operators work independently on each processor holding a part of the implicated relations. Basic operators are graphically represented by circles whose inscriptions detail their functionality (e.g. *build hash table*).

Then, the **communication operators** implement data redistribution. They are graphically represented as boxes whose inscriptions state the kind of repartition to be done (e.g. all-to-all repartition).
Example : suppose the join $R_1 \bowtie R_2$ is to be executed using a hash based method. If the input relation R_1 is not distributed on the same processors as R_2 is, at least one of the two relations must be repartitioned. This can be specified using a *communication operator* `any-to-any repartition`, with its related annotations specifying e.g. the processors before and after redistribution and the relation access method.

Third, the **control operators** are used to control the query processing, their inscriptions state the the kind of control to be performed. They are graphically represented by lozenges.
Example: The control operator *choose* [12] decides dynamically between several alternative execution strategies. Its related annotations specify e.g. the processors in charge of this control and the way the run-time machine has to determine the best alternative (see also the Petri net description for the control operator in section 5.1).

All operators are enriched with annotations depending on the characteristics of a parallel environment. Those annotations specify for example the method how the stored data is accessed and the kind of data dependency that exists between relational operations i.e. sequential or pipelined. Additionally, the different types and degrees of parallelism, namely pipeline, task and data parallelism, can be referred to.



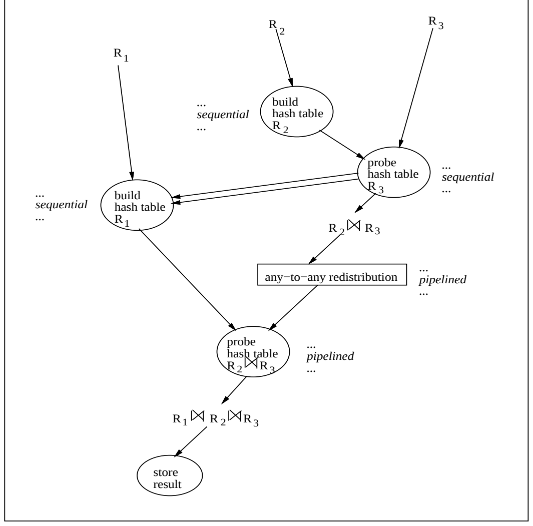


Figure 1: Left scheme : *D graph* for the sample query. Right scheme: *DP graph* for the same query.

3.2 From D graphs to DPL graphs

In order to express fine-grain parallelism, most works [4, 5, 2] introduce low-level implementations into the existing model of query processing trees (see section 2). The resulting graphs will furthermore be called **D graphs**. The *D* stands for the **D**ata dependencies between operators represented by the connecting edges. See fig. 1 left scheme for a *D graph* for the sample query[1].

D graphs cannot model all possible parallel execution strategies. To illustrate this, let us suppose that the available memory is limited and that the hash-table on R_1 can be built only when the $R_2 \bowtie R_3$ has been terminated. Such a situation can not be represented using *D graphs* (fig. 1, left scheme). Indeed no *data* dependency exists between the `probe hash table` R_3 operator and the `build hash table` R_1. Therefore, these two operations should be executed in parallel.

Representing such an execution strategy requires to introduce **precedence dependencies** stating that an operator must be terminated before another operator can start, though no *data* dependency is involved. Graphs including *precedence dependencies*, graphically represented as a double directed edge, will be called **DP graphs** (for **d**ata and **p**recedence dependencies). Thus, the *DP graph* of fig. 1, right scheme, indicates that the `build hash table` R_1 can start only when the redistribution of $R_2 \bowtie R_3$ (`probe hash table` R_3 operator) has terminated.

Let us consider now a more realistic processing situation in which the relation partitions cannot fit into the main memory of the processors. In such situations some overflow processing has to be carried out. A common way to do that is to work on portions of partitions, called *buckets* [7]. Now, let us take a look at the simplest hash based join algorithm working on buckets, the *Grace hash join* [7]. In the *split phase*, a first hash function is applied to the tuples in order to determine the processor they must be assigned to; then a second hash function is applied to determine the number of buckets. In the local *join phase*, the buckets of the relations to be joined are successively loaded into main memory and a classical hash-based algorithm is applied.

[1]In order to keep the figures readable, only the data dependencies (see section 3.1) between operators (*sequential* or *pipelined*) are mentioned. Other annotations are hidden.

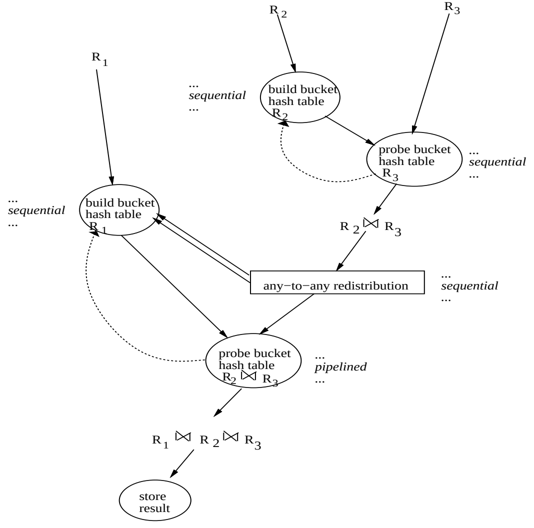

R1
R2
R3
build bucket hash table R2
probe bucket hash table R3
build bucket hash table R1
R2 ⋈ R3
any−to−any redistribution
probe bucket hash table R2 ⋈ R3
R1 ⋈ R2 ⋈ R3
store result
... sequential ...
... sequential ...
... sequential ...
... sequential ...
... pipelined ...

Figure 2: *DPL graph* for the sample query.

Practically, the first bucket of the first relation is loaded and a `build hash table` operation performed. Then, the first bucket of the second relation is loaded and a `probe hash table` operator applied. After that, the second bucket of the first relation is loaded and the hash table built before the second bucket of the second relation is probed against it. And so on and so forth.

Unfortunately a *DP graph* cannot model such bucket based join algorithms. Indeed, modeling the local join process with only one `build hash table on bucket` operator connected by a sequential data dependency to a `probe hash table on bucket` operator is not correct, as the **loop** phenomenon between these two operators is not represented. Therefore, it is necessary to introduce a new kind of dependency called **loop dependency**:

*A **loop dependency** is based on a sequence of **basic** and **communication** operators lying on a same data dependency path. It indicates that this sequence must be repeated as many times as there are available buckets (or tuples). **Loop dependencies** are represented by dotted directed edges.*

A *DP graph* including loop dependencies will be called **DPL graph** (for **d**ata, **p**recedence and **l**oop dependencies). Fig. 2 shows the *DPL graph*, modeling our sample join, $R_1 \bowtie R_2 \bowtie R_3$, in which all the join operations are executed using a Grace-join method.

2

4 Using *DPL graphs* for designing a parallel query optimizer

As noted above, an important contribution of *DPL graphs* is the introduction of novel dependencies : **loop dependencies** which model bucket processing and **precedence dependencies** which allow the integration of scheduling considerations into the execution plan.

In this section, we analyze how a parallel query optimizer can take advantage of *DPL graphs* to improve its optimization effectiveness. We developed a *DPL graphs* based parallel query optimizer implementing a one-phase randomized search strategy (see [14] for more details on the implementation). In the following, we aim to show, on an example, the benefits the optimizer takes from the *DPL graphs*.

Let us concentrate on **precedence dependencies**. Precedence dependencies allow the

[2]There is no "official" definition of a high-level Petri nets. However, a high-level net is usually considered [13] to be derived from a low-level-net by adding labels and annotations to the tokens, transition and places.

modeling of parallel processing strategies in which operators are ordered without reference to a data stream. This typically occurs if resources must be optimized, e.g. if CPU contention appears or if the available memory is limited. Let us consider a three-way join query: $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$. Suppose the target machine is based on a shared nothing architecture, and all base and intermediate relations are declustered over all available disks. Furthermore, assume that only one database operation at most can be run on the same processor[3].

In such a context, most classical optimizers would only consider linear execution trees. So they would actually **discard** all bushy trees. However, let us consider the *DPL graph* in fig. 3. The *P-edge* from the `probe hash table` R_2 to the `build hash table` R_3 operator implies that the join J_2 can only start when the join J_1 has terminated its execution. So we are in front of a "bushy like" execution plan, in which the execution of the operations is serialized (we will call such plans **serialized bushy trees**).

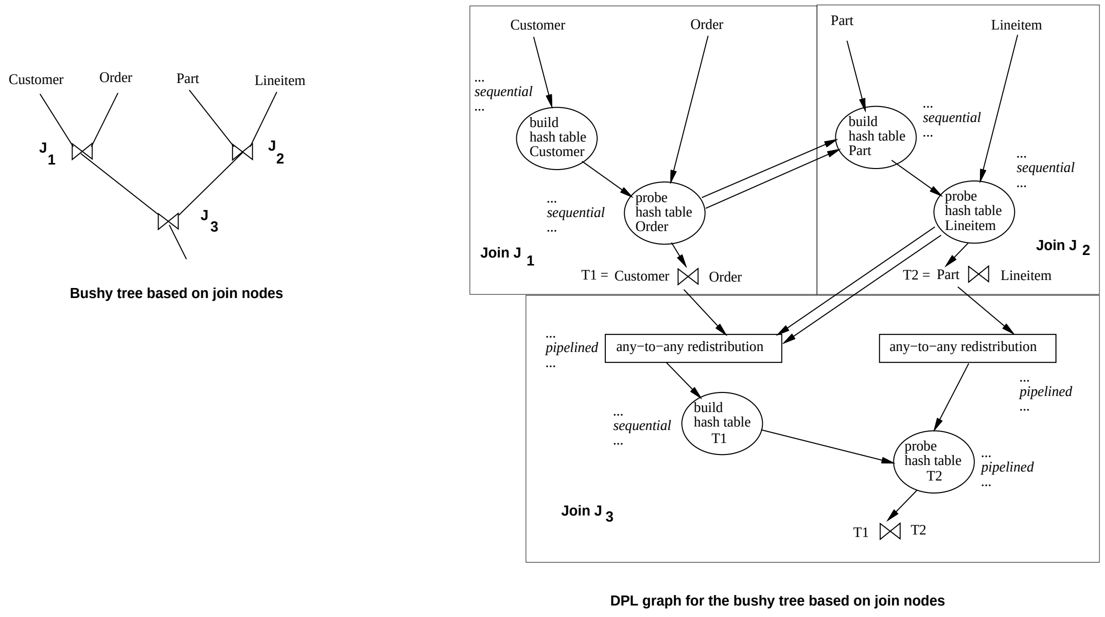


Figure 3: Linear ordering tree with a precedence dependency (*serialized bushy tree*).

We ran the optimizer with the following parameters : the target machine is a cluster of four workstations ; the database schema of the four relations allows one to build three possible joins combinations with a join selectivity[4] between 0.01 and 0.02 ; the number of tuples in each relation varies from 0.5MB to 1MB ; finally the tuple size varies from 30 Bytes until 100 Bytes.

Using the cost model proposed by Ganguly et al. [4], the best *left-deep tree* achieves a cost of 324, the best *right-deep tree* a cost of 323, the best *Zig-Zag* tree [15] a cost of 320, while the *serialized bushy trees* of fig. 3 achieves a cost of only 312.

So this example shows that a parallel optimizer limited to data dependencies by excluding sub-search spaces, risks to miss optimal processing strategies. Oppositely, the representation power of *DPL graphs* allows one to take into account original execution strategies (e.g. serialized bushy trees) while keeping the representation model compact.

[3]Such requirement is not unusual, for instance if the target machine has to face a high CPU charge.

[4]The join selectivity is defined as $\frac{card(join(R,S))}{card(R)*card(S)}$. This formula gives the ratio of the number of tuples in the result of the join to the tuples in a Cartesian product of the two relations.

5 Using high-level Petri nets for modeling control- and data flows in DPL graphs

Timed Petri nets are a powerful and simple theory to model the data flow in distributed and parallel systems [16, 17, 18]. This because this formalism allows the representation of the control and processing strategies while being independent from an underlying parallel machine. However, to our knowledge, Petri nets have not yet been considered for modeling the control- and data flows of parallel relational query processing neither as the base for any simulation tool.

In this context we apply the theory of Timed Environment/Relationship Petri nets (TER-nets) [16], one of the models in the class of timed high-level Petri nets (HLTPNS) [**?**], for representing the control- and data-flow in an *DPL graph* and to enable such net to be the simulation model for parallel execution of the DPL graph. The class of high-level timed Petri-nets (HLTPNS) seems to us the most appropriatest model, because it represent data and functionality aspects in addition to the control aspects of flat and colored Place/Transition Petri Nets [**?**]. Data is associated with tokens and functionality of the operators is associated with transitions. Transition firings are used to model events, whose occurrence is enabled with the availability of suitable data, represented by tokens in the input places e.g. the arriving of the relation pages for the *build hash table* operator. The event consumes the enabling tokens and produces new data in the output places, e.g. the constructed hash table. Among the data associated with tokens, a special timestamp can be used to represent the time at which the token is created. The timestamp associated with the tokens produced by the firing of a transitions depends on the time stamps associated with the removes tokens. The dependency is modeled by the time consuming function associated to the transitions. The transformation of an *DPL graph* to a timed high-level Petri nets requires especially the following features :

Actions must be associated to the transitions, which represent the operators functionality.

Predicates must be added to the transitions, which controls the firing of an transition. This is necessary, because in an *L-edge*, the first operator of the loop should be only reexceuted if there are still buckets to process, otherwise the continuing transitions is to be fired. Thus, the tokens must be attached with a boolean variable whose value indicate if their are still available buckets, i.e. `TRUE` when there are still buckets to process or `FALSE` otherwise. The predicate of the first loop transition will only enable the firing when the value of those variable is `TRUE` and the predicate of the continuing transitions only when it is `FALSE`.

The timestamp should represent **simulation time**. The manipulation of the timestamp is made by the actions associated to the transitions based on the timestamp of all input tokens. The timestamp of the output tokens should be computed as the maximum of all timestamp of the input tokens plus the estimated local operator execution .

When seeking the proposed HLTPNS, the **TER-nets** (Time Environment/Relationship Petri nets)[] developed by the Politecnico di Milano appeared the most adapted to the previous requirements. This because the related HLTPNS based on timed Predicate/Transition net [] offer no way to represent both actions and predicates as associations to a transitions.

TER-nets are an extension of Place/Transitions nets, where the tokens are environments on a set of variables identifiers ID and a set of values V, i.e. functions $ID \rightarrow V$. Let $ENV = V^{ID}$ be the set of possible environments. Furthermore for a given transition t, let ?? denotes the preset of the transition t,i.e. the set of places connected to t by an arc entering t and ?? be the post set of transition t, i.e. the set of places connected to transition t by an arc existing transition t.

The *action* associated to transition t is defined as the relationship :

$$a(t) ?? ENV^{k(t)} \times ENV^{h(t)} \; where \; k(t) = \| ?? \| \; and \; h(t) = \| ?? \|$$

The projection of $a(t)$ on $ENV^{k(t)}$ is called the *predicate* of the transition t.

Furthermore, each token is associated with a timestamp field, representing the time at which the token has been created by a firing, Each transition is associated with a time-function, which described the relation between the time stamps of the tokens removed by the firing and the timestamps of the token produced by the firing. The actions associated with transitions, responsible for producing time stamps must satisfy the two following axioms:

1. *Local monotonicity :* For any firing, the timestamp in the environment produced by the firing cannot be less than the value of the timestamp in any environment removed by the firing.
2. *Firing sequence monotonicity :* For any firing sequence s, the times of firings should be monotonically nondecreasing with respect to their occurrence in s.

Obviously, our proposed time-function : 'the timestamp of the output tokens is computed as the maximum of all timestamp of the input tokens plus the estimated local operator execution' satisfies the previous axioms.

The definition of a *TER-net* fulfills all the requirements for modeling the data and control flow of an DPL graph. Furthermore thay are attractive, as an simulation tool of *TER-nets* has been implemented and made free-ware : *Cabernet* (Computer Aided software engineering environment Based on ER NETs).

The next section develops the transformation rules of an DPL graph to an TER-net. Then in section **??** we will describe how we combined the DPL graphs with *Cabernet* in order to build a powerful tool to simulate the behavior of their parallel execution.

5.1 Transformation of the operator vertices

Recall that we let the transitions correspond to the operator vertices of *DPL graphs*. Attention must be paid only to control operators which need supplement transformation effort. We will treat them a little bit later.

The transformation of the *D*- or *P-edges* connecting two *operator vertices* will be modeled by two transitions, which represent the two *operator vertices* and a place, which is connected to the two transitions.

If the data dependency is pipelined, the transition representing the initial operator vertex will fire as soon as the first tuple is produced. On the contrary, for an sequential data dependency, it will fire only when all tuples are produced. Notice this exception, however, if the terminal vertex of a *D-edge* is the initial vertex of a *L-edge*, a special net construction is necessary, which will be introduced in the next section

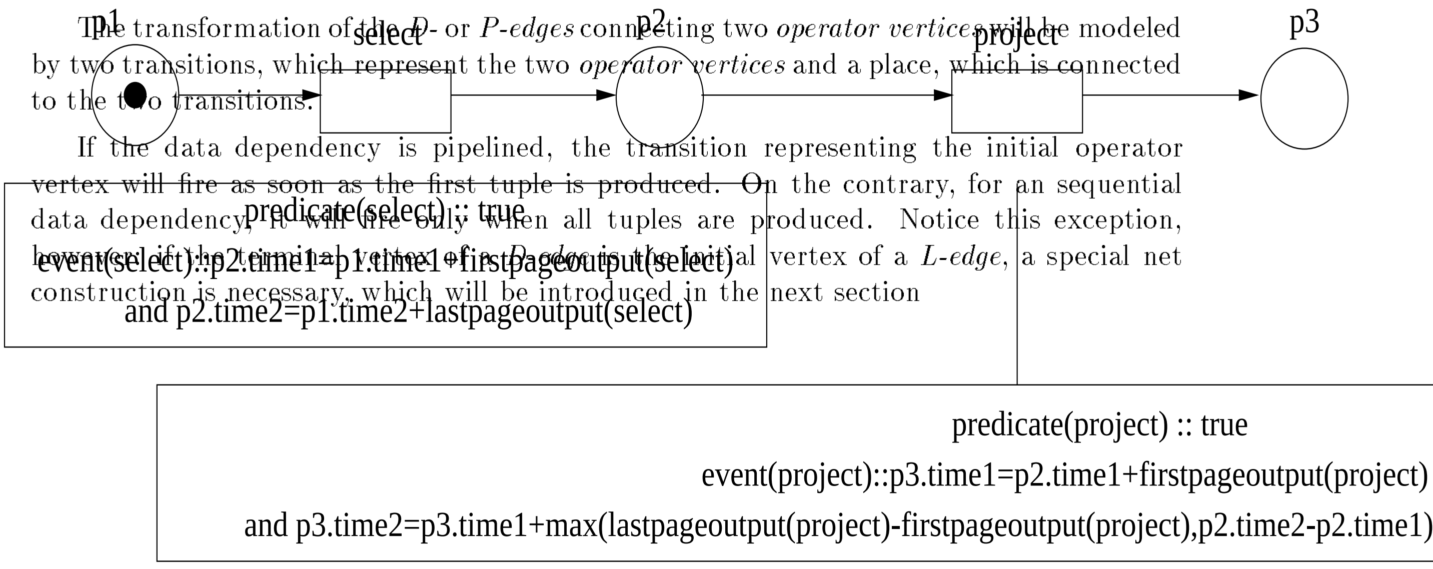

p1
select
p2
project
p3
predicate(select) :: true
event(select)::p2.time1=p1.time1+firstpageoutput(select)
and p2.time2=p1.time2+lastpageoutput(select)
predicate(project) :: true
event(project)::p3.time1=p2.time1+firstpageoutput(project)
and p3.time2=p3.time1+max(lastpageoutput(project)-firstpageoutput(project),p2.time2-p2.time1)

Figure 4: Transformation of a sample *DPL graph* of two basic operators to a *TER-net*.

Let us consider the example of an *DPL graph* consisting of two basic operators *select* and *project*, connected by *D-edges*. Its related *TER-net* is shown in fig. **??**. In the graphical representation of an *TER-net*, the values associated with tokens are described as structured data. To denote the field x of any token stored in place p, the notation $p.x$ is used. The only field required for the token of the example is the simulation time, denoted by *p.time*. Its value is modified in the *select* and *project* transition. Furthermore no predicate condition has to be specified.

The integration of a control operator vertices is more complicated, as its execution can imply a decision between two execution alternatives. For example consider the *choose operator* [12] shown in fig. 5 left scheme, for an alternative execution between an *index-scan* or a *seq-scan* access of a relation R. The *choose operator* is inserted in the query execution scenario at compile-time and the run-time processing control decides on the actual execution environment which operator to execute.

This run-time decision is modeled in the *TER-net* with the introduction of an supplement field *scan* of type integer to the token. It is initialized in the transition representing the *choose* control operator, i.e. *scan* is set to 1, if an *index-scan* is best strategy for the current execution environment (performed by a *decide*-function) or set to 0 otherwise. Based on the token value of *scan* the predicate associated to the *index-scan* and *seq-scan* transitions decides if the transition is firing, i.e. `index-scan` will fire if the value of the field *scan* of the input token is 1, otherwise `seq-scan` will fire. Fig. 5, right scheme shows the corresponding *TER-net*.

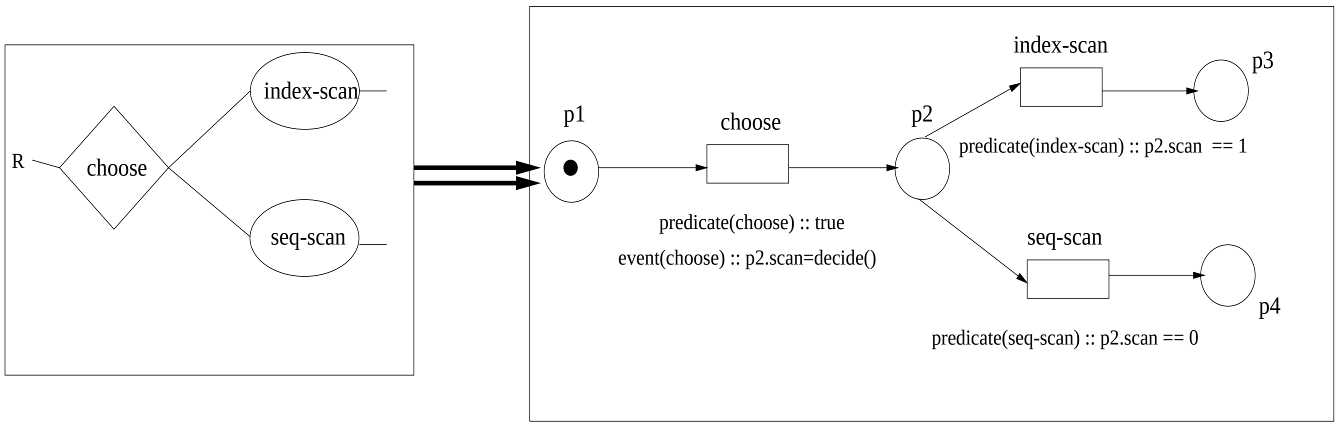

R
choose
index-scan
seq-scan
p1
choose
p2
index-scan
p3
predicate(index-scan) :: p2.scan == 1
predicate(choose) :: true
event(choose) :: p2.scan=decide()
seq-scan
p4
predicate(seq-scan) :: p2.scan == 0

Figure 5: Left scheme : *DPL graph* for the *choose operator*. Right scheme: Corresponding *TER-net* control schema.

Recall, taht such adaptation capacities can be opposed to the control schemas used in systems like DBS3 [9] (see also related work in section 2), which are static and allow no run-time adaptation.

5.2 Initialization and terminating net constructions

Two special nets : the **input-** and the **output** nets are devoted to model the beginning and the end of the execution.

The **input place** is connected to the **initialization transition**. This latter represents the operations to be done before starting the execution, e.g. set the value of the time field to 0 The *initialization transition* issues as many places as there are base relations. These places are then connected to the transitions representing the first operator vertices working on these base relations. In order to allow the execution to begin, a starting token is put in the *input place*.

The **output place** is connected to the last executed operator vertices (e.g. the store result operator vertex in fig. 2).

5.3 Transforming of the loop dependency

The control schema for the *L-edge* requires a complex *TER-net* construction. Assume, without loss of generality, that only one external operator vertex (noted `operator j`) is connected to an operator vertex lying on the loop.

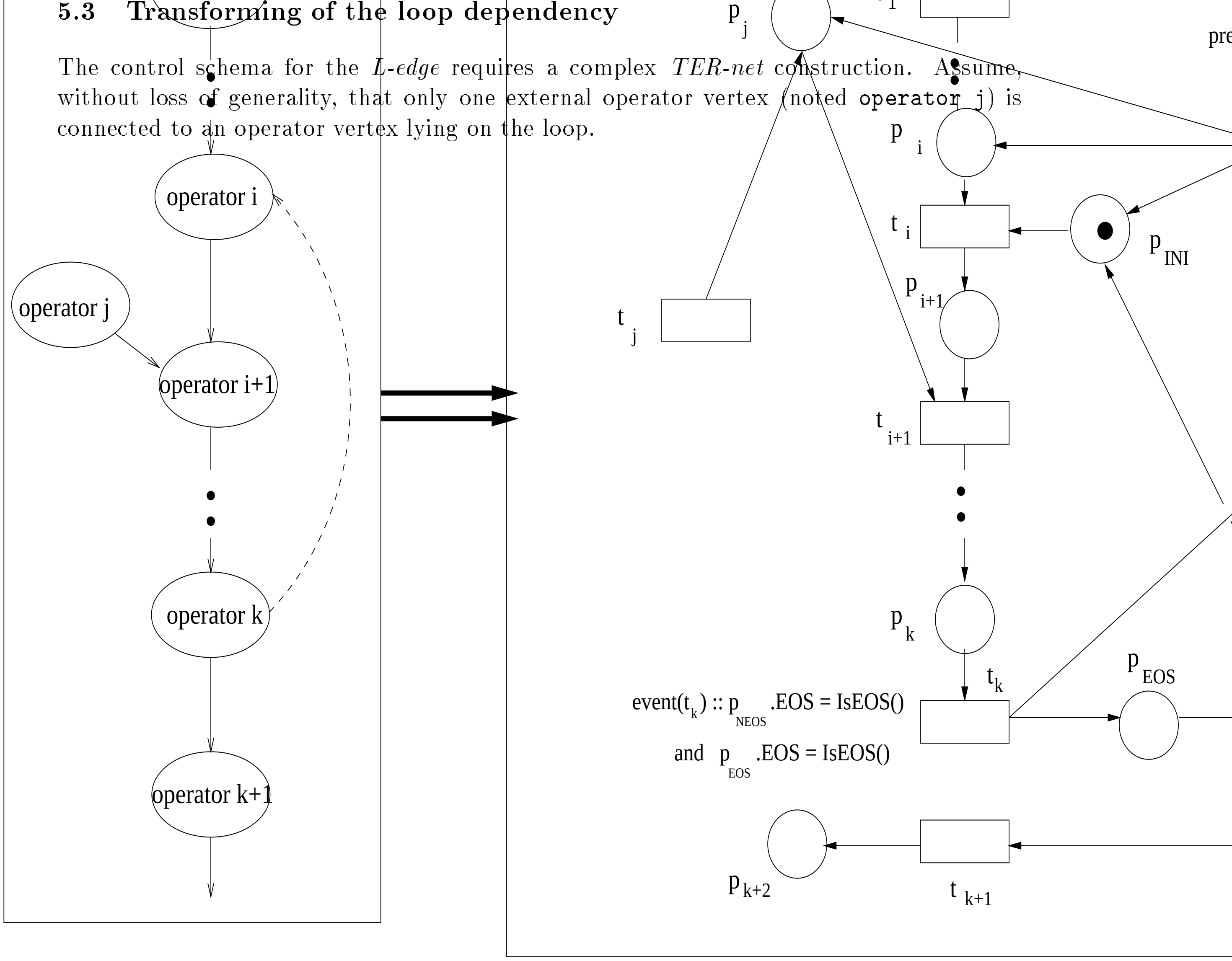


Figure 6: Transformation of an *L-edge* to a *TER-net*.

Fig. 6 shows the *TER-net* construction associated to an *L-edge*. The operators lying on the loop (noted *operator i,..., operator k*) are represented by transitions (noted $t_i, ...t_k$). The *L-edge* is represented by three places ($p_{NEOS}, p_{EOS}, p_{INI}$) and two transitions ($t_{NEOS}, t_{EOS}$). Initially, a token must be put in p_{INI} and p_1. The time field of the token is modified as explained before. In addition, the field EOS of type boolean must be added to the token. It is initialized by the transition t_k corresponding to the last operator lying on the loop. If after processing the actual bucket, no more buckets are available the $IsEOS()$ function returns true, otherwise false. Thus the action associated to t_k expresses as :

$$\text{action}(t_k) :: p.EOS = IsEOS() \text{ and } p.NEOS = IsEOS()$$

If there are still available buckets, i.e. $p.EOS = true$, the predicate condition of t_{EOS} :

$$\text{predicate}(t_{EOS}) :: p.EOS == true$$

is true and will enable the firing of t_{EOS} and then provide the places p_i, p_{INI}, o_j) with tokens in order to reexecute the loop.

If no more buckets are available, i.e. $p.NEOS = false$, the predicate condition of :

$$\text{predicate}(t_{NEOS}) :: p.NEOS == false$$

is true and will enable the firing of t_{NEOS}. This transition provides p_{INI} with a token (in order to come back to the initial state) and fires t_{k+1}, which continues the query execution.

6 Designing a simulation tool based on DPL graphs and their related Petri nets

In section 5, we have shown how the *TER-nets* could model the data and control flows in *DPL graphs*. This point is especially interesting because, an *TER net* free-ware simulator exists *Cabernet* [19] (Computer Aided software engineering environment Based on TER NETs)[5].

In this section we analyze how the combination of *DPL graphs* with Cabernet provide a very powerful tool for simulating parallel query processing, different optimization strategies and run-time control mechanism (e.g. load balancing).

As mentioned above, in *TER-nets*, the simulation time is affected by the actions associated to the transitions. For each operation a function `compute_time()` is introduced, which computes the estimation of the first and last tuple output by the operations. This time interval is added to the global simulation time field associated to the token.

Fig. 7 shows the different programs required for the run of an simulation. *Prog_funct_cost.cc* contains the cost functions `compute_time()` for each possible atomic operation, the program *Prog_Petri.cc* contains the $C++$-data-structure of the *TER-net* related to the input *DPL graph*. Finally, *Cabernet.o* is the Cabernet object file. Compiled and linked together they form the executable simulation code for the input *DPL graph*.

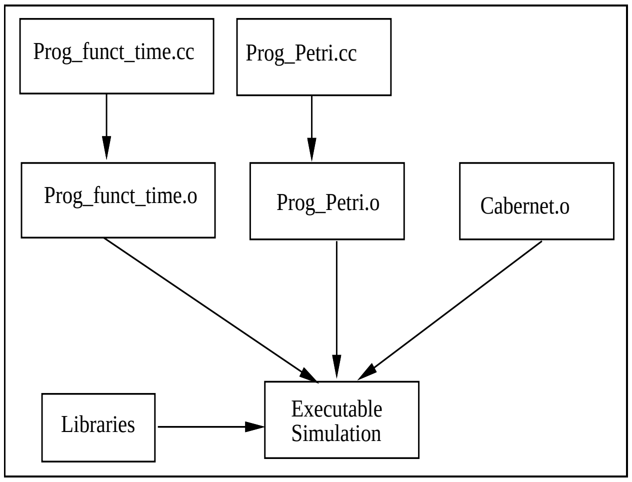


Figure 7: Compiling and linking the simulation tool

.

The transformation of a *DPL graph* to a *TER-net* can be easily automizised by applying the proposed transformation rules successively :

First the operator vertices of the *DPL graph* are replaced by transitions. The actions associated to the newly created transitions must determine the time field of the output tokens, which is computed as the maximum of the value of the time fields of all input tokens plus the `compute_time()`. If the operator is a control operator, a control field must be affected. Based on the value of this field, the successive transitions predicates can determine if they allowed to fire.

Second, the *L-edges* are replaced by the *TER-net* construction, as proposed in subsection 5.3.

Finally, the initialization and terminating nets are added.

The transformation of an *DPL-graph* generated of our parallel query optimizer to the $C++$-data-structure representing the related *TER-net* has been been done by hand for

[5]An overview of all known Petri net tools in the world can be found at the following address :`http://www.crim.ca/se/petri-tools.html`.

several sample *DPL graphs*. The defined *TER-net* must be first loaded.

The simulation can then be ran step by step or in one stream. In the latter case the net executes until it arrives in a stable state, where no more transitions can be fired, or when a deadlock situation occurs. A trace file specifies the sequence of fired transitions and places. Other informations about the simulation run can be shown from the circulated token, e.g. the global simulation time value attached to the token.

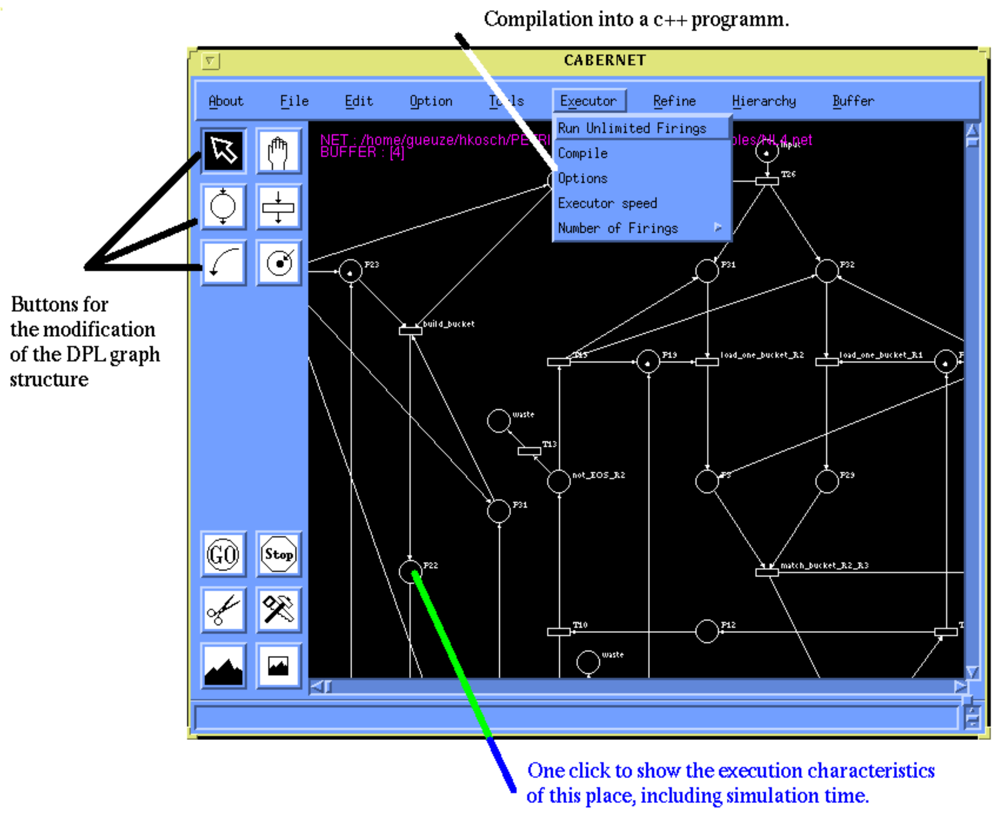

Compilation into a c++ programm.
CABERNET
About
File
Edit
Option
Tools
Executor
Refine
Hierarchy
Buffer
Run Unlimited Firings
Compile
Options
Executor speed
Number of Firings
Buttons for the modification of the DPL graph structure
GO
Stop
One click to show the execution characteristics of this place, including simulation time.

Figure 8: Snapshot of *Cabernet* showing its main functionality.

Running the simulation in the step by step mode allows the redefinition of the simulation environment. Thus load imbalance can be interactively introduced by a modification of the transition/place time value (see fig. 8). Then, several alternative run-time strategies can be tested e.g. a redistribution operator can be integrated. The structure of the *DPL graph* itself can also be easily modified thanks to the graphical representation tool (see fig. 8) e.g. a bushy processing strategy can be changed into a linear just by adapting the arcs in the *ER nets*.

In conclusion, *DPL graphs* combined with *TER-nets* provide a very adaptive and powerful representation model which fulfills very well the requirements of simulations.

7 Conclusion and future work

This paper has described a novel theoretical model for representing parallel relational query executions. In comparison to previous approaches, *DPL graphs* combined with *annotated and inscribed P/T-nets* allow one to very accurately represent most of execution strategies and to precisely model the data- and control flows. Thus, this formalism provides a representation model much closer to the actual execution.

Furthermore, we showed how this formalism could be used for optimizing parallel relational queries and simulating run-time execution and control strategies. In the specific field of the query optimization, *DPL graphs* allowed the introduction of an original class of execution strategies, i.e. the serialized bushy trees.

We are now studying the interfacing of our model with a token driven execution environment, developed at the Budapest Parallel Computer Center [20]. At time this will allow us to propose a complete query processing system, from the query representation model to the actual execution on a target machine.

References

[1] Ramesh Bhashyam. TPC-D The Challenges, Issues and Results. In *Proceedings of the International Conference on Very Large Database Conference*, Bombay, India, May 1996.

[2] R.S.G. Lanzelotte P. Valduriez and M. Zaït. Industrial-Strength Parallel Query Optimization: Issues and Lessons. *Information Systems - An International Journal*, 1994.

[3] M. Spiliopoulou and J.-C. Freytag. Modeling Resource Utilization in Pipelined Query Execution. In LNCS 1124 Springer, editor, *EUROPAR 96*, Parallel Processing, August 1996.

[4] S. Ganguly W. Hasan and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of the ACM SIGMOD International Conference of Managment of Data*, San Diego, California, USA, 1992.

[5] G. Graefe R.L. Cole D.L. Davison W.J. McKenna and R.H. Wolniewicz. *Extensible Query Optimization and Parallel Execution in Volcano*, page 305. Query Processing for Advanced Database Applications. Morgan Kaufman, San Mateo, CA, 1994.

[6] Chekuri C. Hasan W. and Motwani R. Scheduling Problems in Parallel Query Optimization. In *Proceedings of the Principles of Database Sytems*, 1995.

[7] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[8] H. Boral W. Alexander L.. Clay G.. Copeland S. Danforth M. Franklin B. Hart M. Smith P. and Valduriez P. Prottyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):5–24, June 1990.

[9] P. Borla-Salamet C. Chachaty and B. Dageville B. Compiling Control into Databases Queries for Parallel Execution Management. In *Proceedings of the 1st International Conference of Parallel and Distributed Information Systems*, Miami, Florida, December 1991.

[10] L. Bouganum D. Florescu and B. Dageville. Skew Handeling in the DBS3 Parallel Database Sytem. In Springer Verlag LNCS 1127, editor, *Proceedings of the International ACPC Conference*, Parallel Computing, September 1996.

[11] L. Brunie and H. Kosch. DPL graphs - A powerful representation of parallel relational query execution plans. In LNCS 1124 Springer, editor, *EUROPAR 96*, Parallel Processing, August 1996.

[12] L. Brunie and H. Kosch. Control strategies for complex relational query processing in shared nothing systems. *ACM Sigmod Records*, 25(3), September 1996.

[13] K. Jensen and G. Rozenberg. *High-Level Petri-Nets, Theory and Application.* Springer-Verlag, 1991.

[14] L. Brunie and H. Kosch. Parallized query optimization in parallel databases. In Shaker, editor, *4th Workshop on Scientific Computing*, Parallel and Distributed Computing, October 1996.

[15] Ziane M. Zaït M. and Borla Salamet P. Parallel Query Processing with ZigZag Trees. *Very Large Databases Journal*, 2(3), March 1993.

[16] C. Ghezzi D. Mandrioli S. Morasca and M. Pezzè. A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE Transaction on Software Engineering*, 17(2), February 1991.

[17] I. Gorton. Parallel Program Design using High-Level Petri Nets. *Concurrency: Practice and Experience*, 5(2), April 1993.

[18] C.I. Birkinshaw and P.R. Crall. Modelling the Client-Server Behaviour of Parallel Real-Time Sytems Using Petri Nets. In IEEE Computer Society Press, editor, *Proceedings of the International Conference on Systems Sciences*, January 1995.

[19] Sergio Silvia. Cabernet user manual. Technical report, Politecnica di Milano, May 1994. available at ftp: ftp-se.elet.polimi.it.

[20] Peter Kacsuk. Dataflow Model for Handling Cut in the LOGFLOW Parallel Prolog. In IEEE Computer Society Press, editor, *Proceedings of the IEEE Euromicro Workshop on Parallel and Distributed Processing*, San Remo, Italy, 1995.