A Database Rule Language Compiler Supporting Parallelism

J.-P. Cheiney
ENST, 46 rue Barrault
Paris, France
cheiney@inf.enst.fr

G. Kieman
INRIA Rocquencourt
78153 Le Chesnay, France
kieman@rodin.inria.fr

C.de Maindreville
Alcatel-Alsthom Rocchero
91460 Marcoussis, France
maindrev@aar.alcatel-als Thom.fr

Abstract
This paper presents an extension to a database rule language compiler to support parallelism. The rule language is called RDL/C and is based on an extended version of tuple relational calculus. The grain of parallelism introduced into RDL/C is the production rule. The user is made aware of parallel processing by having to specify in a sub-language of control which sets of rules are to run in parallel and in which order. From the user view point, the parallelism is managed as a partial order over a set of rules. The compiler translates rule programs into C based applications which run over the DBMS. Our implementation is over a set of UNIX workstations which corresponds to a MIMD shared nothing architecture. Parallelism is achieved by having the application connect to different DBMS servers residing on workstations on a local network. Initial performance results are presented. They show the advantages and limitations of parallel execution in this architecture. The main contributions of the paper are (i) the definition of a sub-language of control to specify a parallel execution of a rule program and (ii) to show how parallel extensions can be brought to a rule language compiler using a standard distributed environment with a relational DBMS.

Keywords: Production Rules, Parallelism, Language of Control, Compilation, RDBMS, Implementation.

1. Introduction

Rule languages have long been recognized as candidates for parallel languages [Almasi89, Ganguly90, Wolfson90]. Parallel evaluation of Datalog programs has been studied in [Ganguly90, Wolfson90]. These papers introduce the notion of discriminating predicate which allows the instanciations of rules to be partitioned among different processors. Both papers are concerned with the reduction of communication overhead and load balancing between the processors and discuss the trade-off between non-redundancy and communication costs. One important result stated in [Wolfson90] is the undecidability of the decomposition problem. The parallel evaluation of Datalog-neg programs is discussed in [Wolfson90]. For such programs, in general, the processors have to be synchronized at each stratum. This means that each processor has to wait until all the processors have completed their computation before proceeding to the next stratum.

Experiments with parallelism in production systems such as OPSS [Brownstone85] have been presented in [Gupta89]. In such environments, it is up to the system to detect the possible sources of parallelism in a rule program. It is also the system's job to guarantee that the parallel execution of a rule program is equivalent to the sequential one. Different sources of parallelism have been identified [Almasi89]. The most obvious one is the production rule as the unit of parallelism. Production rules which can run independently are detected and each group is assigned a processor. Three other sources of parallelism are AND-parallelism, OR-parallelism, and Parallel pattern matching. These last three sources of parallelism concern intra-rule parallelism. A single production rule is decomposed into sub-parts which can run independently. The first two types of parallelism come from AND/OR graphs which represent the connection between the predicates and rules which comprise a production system. In the case of Horn clauses, for example, the ANDs are the conjunction of expressions which form the body of a rule; and the ORs are rules with identical heads. Parallel pattern matching is the implementation of a parallel version of the inference engine.

In [Almasi89], the authors report the claim that a production rule application which is written with parallel processing in mind yields a higher degree of useful parallelism than one which is written with a serial machine in mind. Moreover, the performance gained is proportional to the number of rules in the system; and of course to the number of processing elements. They also claim that the useful parallelism and the speed-up from the "rule-per-processor-approach" is yet to be fully established. This indicates that while it is up to the system to detect and implement parallelism, the user must know how the system achieves this parallelism in order to exploit it fully. In [Srivastava89], two types of parallelism are proposed for database production systems. These are user-visible and user-transparent categories. It is the user's responsibility to divide a task into non-interacting subtasks and it is the system's responsibility to execute each element of a subtask in parallel.

The approach presented in this paper supports parallelism at the rule level. The user is made aware of parallel processing by having to specify which sets of rules are to run in parallel and in which order. A control sub-language is used for this purpose. We propose that a parallel algorithm is better than a sequential one on which a certain degree of parallelism might be automatically extracted. We do not address in this paper the automatic detection of parallelism in rule programs.

This paper presents an extension to a rule language compiler to support parallelism. A prototype of the RDL/C compiler is operational on a network of UNIX workstations [Unix84]. Since RDL/C [Kiemann90b, Kiemann91] production rules are based on relational calculus, they can be solved by a relational DBMS without having to extract data during the inference cycle. The inference is driven from the application (generated by the compiler) through a series of calls to the DBMS using an SQL interface. The compiler manages parallelism in production rules by generating a program which connects to several DBMS over a network of workstations. In this framework, each DBMS is a parallel processing unit. The application communicates with each DBMS using standard UNIX interprocess communication facilities called sockets. To run a set of rules in parallel, the application process forks (fork is a UNIX system call which creates a new process) which can run independently. The first two types of parallelism come from AND/OR graphs which represent the connection between the predicates and rules which comprise a production system. In the case of Horn clauses, for example, the ANDs are the conjunction of expressions which form the body of a rule; and the ORs are rules with identical heads. Parallel pattern matching is the implementation of a parallel version of the inference engine.

The main contribution of the paper is to show how a parallel production system can be implemented over a network of workstations. Initial performance results indicate the advantages of this approach for high performance production rule systems. Section 2 presents an overview of the production rule language which is used throughout this paper. Production Compilation Networks (PCN) [Maindreville88] as a model of parallelism are introduced in Section 3. PCN are the computational model for the RDL/C production rule system. Section 4 describes the syntactic extensions that were brought to the serial version of the compiler.
to support parallelism. Section 5 concerns the architecture. The compiler and the run-time support are presented. Section 6 details the implementation and Section 7 gives performance results and discuss the advantages and limitations of this approach.

2. Overview of the Language

RDL/C is derived from RDL1 [Maindreville88, Kiernan90a], a production rule language which has been integrated in the Sabrina RDMS [Gardarin89J. The RDL/C language supports declarative programming based on RDL1 and procedural programming based on C code. In this section, we present an overview of the language through some examples. The syntax and semantics of RDL/C are given in [Kiernan90b, Kiernan91J.

2.1. Illustrative Example

Consider the base relation Parent having the schema Parent (asc integer, desc integer). The following rule program computes the transitive closure of the Parent relation :

A parallel version of this program will be presented in Section 7, when performance issues are discussed.

2.2. The Kernel Language

2.2.1. The Syntax

The rule part of an RDL/C program is composed of a set of if-then rules. The IF part of a rule is a tuple relational calculus expression. Its syntax is very close to the syntax of a WHERE clause in the SQL language. The THEN part of a rule is a set of actions that are insertions, deletions of tuples in relations, procedural side-effects and variable assignments. A discussion of the latter two types of actions are beyond the scope of this paper. However, they are described in [Kiernan90b]. The action part is very close to the SELECT clause of an SQL statement.

Let Person and Worker, be two relations having the same schema (id integer, name char, age integer). The following expressions are valid Left-Hand-Sides (LHS) of rules :

The following expressions are valid RHS of rules :

Following is a set of valid rules :

2.2.2. The Semantics

The semantics adopted for the RDL/C language is a set oriented one. When a condition is evaluated against the database, it returns the set of instances which make the condition valid. When an action is executed against the database, it is executed for all the values which appear as arguments in the action. In the following, we present examples of rule execution.

Let us consider the following rules :

Firing the following rule causes the insertion into the Worker relation of the contents of the Person relation :

Firing the following rule causes the deletion from the R1 relation of the contents of the Person relation :

Firing the following rule leads to a null action :

Firing the following rule causes the insertion into the Q1 relation the set of tuples :

causes the insertion into the Q1 relation the set of tuples :

and the deletion from R1 the set :

2.3. Partial Ordering of Rules

The mixing of declarative reasoning and imperative control has been shown necessary for many applications. A control language has been designed to specify an application mode over the rules. It induces a partial ordering over the rules. Each expression of the sub-language is declared in the module. The basic terms of control language are rule names. A general expression exp in the control language is :

If a rule name does not appear in the control part of the program, its firing is chosen at random by the inference engine. If there is no CONTROL section, the rule interpreter applies a default strategy. The priority of rules is according to the order of their appearance in the module.
Let us consider a rule program \((r_1, r_2, r_3, r_4, r_5)\).

The PCN model supports parallelism at the rule level as it is shown in the following example: consider the PCN,

![Figure 3.2: PCN](image)

The firing of the transitions \(t_2\) and \(t_3\) is easily parallelized since they do not share any post places in the PCN structure. On the contrary, the transitions \(t_4\) and \(t_5\) are not easily parallelized since the firing of \(t_5\) needs the contents of the place named \(f\) which is given by the firing of \(t_4\). Such a restriction is due to the set oriented computation of each transition (rule). A dataflow execution would have allowed the relaxation of this kind of limitation.

We now give syntactic restrictions over the PCN structure that allow the parallelization of a rule program. The limitations are of two different kinds. The first one is due to the parallel granularity we chose, i.e. the rule. The second one is for semantic considerations. As discussed in [Wolfson90], for non deterministic languages such as Datalog-neg, the processors have to be synchronized at each stratum. As RDL//C is an extension of Datalog-neg, the same limitation applies: a non recursive transition is fired iff its input places have reached a stable marking. As shown in [Maindreville88], this induces a partial ordering on the rule program and, for instance, implements the notion of stratification. This limitation synchronizes read/write operations between rules. That is, a rule is not allowed to write data in a relation which is read by another rule in parallel (see rules \(t_4\) and \(t_5\) in the previous section).

These limitations are expressed over the PCN model as follows:

Let us consider \((t_i)\) a set of transitions, and Post\((t_i)\) and Pre\((t_i)\) the input and output places of \(t_i\). Then \(t_l\) and \(t_k\) can be parallelized (noted by \(t_l // t_k\)), iff:

- \(Post(t_l) \supset Pre(t_k)\) or \(Post(t_k) \supset Pre(t_l)\)
- \(Post(t_l) \neq Post(t_k)\) for each \(l, k\)

These two conditions allow only independent rules to be fired in parallel. They ensure that the semantics of the rule program are preserved by the parallel processing. On the PCN displayed in figure 3.2, only transitions (rules) \(t_2\), \(t_4\) and \(t_3\) can be run in parallel.
4. Extending the Language to Support Parallelism

The RDL/C language has been extended to support inter rule parallelism. The parallelism is specified in the control language as an explicit partial ordering over the set of rules. The language extension used to specify parallelism among the rules is the PAR structure, found in the control string. Each argument is itself a control argument. PAR can be nested. Semantic checks are done over the arguments to ensure that the rules to be run in parallel are independent as defined in the previous section. Consider the following example:

```
SW (To, PAR (~1, BIDCK (r2) ), SEQ (rlh BLtXX (r2B))), r3)
```

This control structure specifies two sets of rules to be run in parallel. The first set contains rules rl and r2; the second set contains rules r1B and r2B. The control structure of the first set states that rule rl fires once and that rule r2 fires up to saturation. Rules rl and r2 will run on the server servA and rules r1B and r2B will run on the server servB. When a set of rules has been fired according to the control structure given in the control string, the process controlling it attempts to synchronize with other processes running other sets in parallel. All rules in the PAR control structure must have fired according to the control string before the engine attempts to fire rule r3. Consider the following control string.

```
SEQ (r0, PAR (rl, r2), r3, PAR (r4, r5), r6)
```

In the above example, the engine will attempt to fire rule r0, and then fire rules r1 and r2 in parallel. When r1 and r2 are no longer firable, the engine will move on to rule r3 and then, attempt to fire rules r4 and r5 in parallel before firing r6. The following control structure is equivalent to the previous one.

```
SEQ (r0, PAR (block (rl), block (r2)), r3, PAR (block (r4), block (r5)), r6)
```

A rule which is run in parallel can also be run sequentially in the same module. This is the case for rule r2 in the following example.

```
SEQ (r0, r1, PAR (r2, r3), r4, r5, r6)
```

The inference engine will attempt to fire rules r2 and r3 in parallel. It will then attempt to fire rule r4 before attempting to fire rule r2 again.

5. Architecture

5.1. Overview of the Compiler

The general architecture of the RDL/C compiler is portrayed in Figure 5.1.

```
RDL source  RDL/C compiler  C program
          Run-time Library
```

The compiler accepts a source program and produces, as output, a C program which implements the rule program. The C program contains code to implement each rule and includes the inference engine which fires rules until a fixpoint is reached. The DBMS does not require any inferencing capabilities to process the program. The extraction of the data from the DBMS to the application is not required during the inference process. This is because rules are based on relational calculus and can thus be solved by the DBMS.

The compiler generates an SQL connection statement for each server specified in the ON statement in the module. The PAR structure is compiled from the control string in basically the same way as are the other two structures SEQ and BLOCK. The run-time library which is linked with the C program produced by the compiler manipulates parallelism. These procedures are the topic of the next section.

5.2. Run-time Environment

The run-time environment is a set of workstations linked to one another through a network. Each workstation has similar computational power. The application resides on one workstation while each workstation is assigned one DBMS process. Each DBMS has its local database. Each relation referenced in a module must be found in at least one site. This allows a horizontal and vertical partitioning of relations in so long as each partition is a named relation existing on a site. The DBMS is a standard relational DBMS. Although the one that we are using supports Abstract Data Types [Gardarin89], this is not relevant for parallel processing. Parallelism is achieved by having the application access each DBMS concurrently.

5.2.1. UNIX System Tools

This architecture is supported by the UNIX operating system. The key facilities used to support parallelism are the process control facilities and network facilities. The UNIX fork statement creates a duplicate process from a running process. Running concurrent processes communicate using sockets and pipes. Sockets and pipes are similar structures used to support interprocess communication. The Network File System (NFS) allows users on different machines to share files.

5.2.2. Schema of the Run-Time Environment

```
Figure 5.1 : Sketch of the RDL/C compilation environment.
```

```
Figure 5.2 : The Run-Time Environment.
```
5.2.3. The Application and the DBMSs

An application requiring access to the database issues an SQL connect statement to establish a connection to the DBMS. The application and the DBMS then communicate through sockets. The application issues SQL commands to the DBMS and the DBMS returns its results after each command. There is normally one such connection between an application and the DBMS. However, to exploit the power of parallel processing, the compiler generates an application which establishes a connection with DBMS residing on different machines. The application can thus communicate with each DBMS by using the communication link that is assigned to it.

5.2.4. The Parent Application and its Children

Although the application can access each DBMS independently, it cannot do so in parallel without creating extra processes. To run a set of rules in parallel, the parent (or main) process creates a child process for each set of rules which are to be run in parallel. This is achieved with the UNIX fork system call. Each child is assigned a particular DBMS. All application processes reside on the same machine regardless of the fact that they access DBMS which are on different machines. Before creating a child, the parent dumps all the relations which are referenced in the set of rules managed by the child. The child loads the relations before starting to process the rules. Running parallel processes synchronize when they have finished running the set of rules they have been assigned. Each child must return information on the rules that it fired; this communication is done through pipes. One such pipe is created per child process. Relations which were created by a child process must be made available to the parent. This is achieved by having the child dump these relations on a common file system and the parent load them from this file system. The restrictions over the PCN avoid any collisions between relations being dumped. Two rules running in parallel can not reference a same relation in the RHS. After having done this, the child process exits. The parent process synchronizes with the child processes once it has finished running the set of rules that were assigned to it. The parent synchronizes with each child, one after the other. Once it has finished, the parent process can resume firing rules in sequence or in parallel.

6. Implementation

Processing a PAR in the Control String

The algorithm used to process PAR in the control string is the following one. It is given in pseudo C language notation.

The control string is the support for parallelism in the language. The control string is mapped to an in-memory data structure which is used by the inference engine. While processing the control string, if the inference engine encounters a PAR control structure, the engine tries to fork child processes to run rules in parallel. To do this, it goes through the following steps. First, it determines, for a set of rules, if there are pertinent rules in the set. If there are no pertinent rules, the engine forks a new child process for each set of rules which are to be run in parallel. This is achieved with the UNIX fork system call. Each child is assigned a particular DBMS. All application processes reside on the same machine regardless of the fact that they access DBMS which are on different machines. Before creating a child, the parent dumps all the relations which are referenced in the set of rules managed by the child. The child loads the relations before starting to process the rules. Running parallel processes synchronize when they have finished running the set of rules they have been assigned. Each child must return information on the rules that it fired; this communication is done through pipes. One such pipe is created per child process. Relations which were created by a child process must be made available to the parent. This is achieved by having the child dump these relations on a common file system and the parent load them from this file system. The restrictions over the PCN avoid any collisions between relations being dumped. Two rules running in parallel can not reference a same relation in the RHS. After having done this, the child process exits. The parent process synchronizes with the child processes once it has finished running the set of rules that were assigned to it. The parent synchronizes with each child, one after the other. Once it has finished, the parent process can resume firing rules in sequence or in parallel.
7. Performance Analysis

7.1. The Environment

The objective of this section is to measure the performances of the execution of a RDL//C program. The Sabrina DBMS [Gardarin89] we used, implements all the standard features of commercial relational DBMS. The system architecture is a network of Sun SPARK workstations, each one having disk capabilities to store the database. This environment is somewhat different from bus-based backend database processing. Particularly, workstations are not designed for stand-alone processing (as dedicated database servers). Rather, they offer resource sharing to obtain load balancing through a local area network. However, such an environment can provide a larger memory space and faster computation in many cases. Moreover, a network of workstations constitutes a very common environment which has the advantages of parallel processing but without the specialized or dedicated hardware.

7.2. The Application

The application used to measure performance is based on a typical transitive closure operation to solve the ancestor problem. A sequential version of the rule module which calculates ancestors has been given in Section 2. We give a parallel program for the same problem. Parallelization of transitive closure has been widely studied [Valduriez88, Agrawal88, Cheiney90]. Within proposed algorithms, parallelism is included in the operator itself. In our approach, we express the parallelism within the rule language by adding localization predicate to rules [Wolson90] and using the PAR structure within the control string.

The relation Parent is duplicated on n nodes. The processing task is divided into n parts: a first station computes the ancestors of desc value whose (value MOD 2) = 0; a second station computes the ancestors of desc value whose (value MOD 2) = 1; and so on, for n workstations. In the case of total replication of the relation Parent, the ancestors problem can be partitioned among n workstations without intermediate transfer. Only the building of the complete Ancestor relation implies transfers.

We give the RDL//C program for two nodes:

```
MODULE ancestor;
BASE
    Parent (an-int, desc-int) ;
DEDUCED
    Arc [] INE Parent ;
    Ancestor LHS Parent ;
RULES
    r1 IS
        IF Parent (x, desc) (desc MOD 2 = 0) THEN arcs (x) ;
    r2 IS
        IF Parent (x, desc) (desc MOD 2 = 1) THEN arcs (y) ;
    r3 IS
        IF Arc (x) THEN anArc (x) ;
CONTROL
    SQL (OPR SQL (r1, BLOCK (x[0]), SQL (RBlocks (y[1]), r3[0]),
        r3[1]));
END MODULE
```

We will compare the performances of the system varying the load and the number of processors available. The load will vary according to the number of tuples in the Parent relation. Measurements are done with different loads. These loads are generated as a tree of ancestors, which has h levels (Figure 7.1). h is also the number of firings of the recursive rule (i.e. the depth of the induced join loop in our implementation). There are no indexes on the relations. The number of tuples vary in each set by having the number of descendants at each node vary by one. In the first set of tuples each node has three descendants; in the second set, each node has four; and in the last set, each node has seven descendants. With h=4, we obtain the five following loads: 363 tuples, 1364 tuples, 3905 tuples, 9330 tuples, 19607 tuples. The number of processors varies from one to five.

7.3. Performance Results

To obtain performance results, we use some system time routines provided by the UNIX System. Eight time measurements are investigated, that are:

1. `Sql_connection`. The time measured includes starting up a DBMS process on each remote station. This time is a priori proportional to the number of stations;
2. Validation of the schema of base relations at run-time;
3. Synchronization. Using the NFS, passing relations consists in loading and dumping relations in files; running parallel processes are synchronized by waiting for each one to have finished running the assigned set of rules. This time measures transfer and synchronization tasks (i.e. the parallelism overhead);
4. Left Hand Side (LHS) of rule processing. This time measures select/join processing time;
5. `Sql_get_schema`. The schema is obtained at run-time;
6. Validation of the schema of the deduced relations at the run-time;
7. Right Hand Side (RHS) of rule processing. This measures the computation time of the union;
8. `Sql_disconnect` measures the time to disconnect the applications from the DBMS.

The first results have shown that only three of these measurements are significant within the overall time: synchronization, LHS and RHS rule processing. The sum of the other five components represents always less than 3% of the overall time (except for a few tuple -363 tuples- and a single processor where the overhead time -connection, deconnection, validation of schema- reaches to 5%). For this reason, the main point of discussion will be the three main time-consuming actions.

![Figure 7.1: Tuples generation tree](image-url)
7.3.1. Sequential Time

The first measurements are the times with a single workstation. The aim was to reveal the behavior of the rule program and the relative amount of execution time spent in each part of the program. Particularly, the time spent to evaluate the condition part of the rules (the LHS processing corresponds to a selection and a join) and the time spent to process the action part (the RHS processing corresponds to an union). Right and Left parts of rules compose the computation time. An external time, obtained as the sum of the connections, disconnections, validations of schema, is also calculated. Table 1 gives the measurements of the three main components of the elapsed time.

<table>
<thead>
<tr>
<th>Table 1: Time spent in part for sequential processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection, disconnection and schema control</td>
</tr>
<tr>
<td>Left Hand Side (Select/Join)</td>
</tr>
<tr>
<td>Right Hand Side (Union)</td>
</tr>
<tr>
<td>363 tuples</td>
</tr>
<tr>
<td>3s</td>
</tr>
<tr>
<td>12s</td>
</tr>
<tr>
<td>41s</td>
</tr>
</tbody>
</table>

Table 1 shows the importance of union processing time. Moreover, these measurements make clear that the external time is very limited and can be neglected when processing large sets of tuples (from 5% of the overall time for a load of 323 tuples to less than 1% for a load of 19,607 tuples). The overhead time due to rule program connection and controls, is very small; the computational power is almost completely used to process the rules.

7.3.2. Parallel Time

Neglecting the connection time, the components of the total time are the join computation, the union computation and the parallelism overhead including the dumps, the loads and the waiting. We measured the overall and the partial times for five sets of basic tuples: the first one (363 tuples) produced 1641 ancestors, the second one (1,364 tuples) produced 6,372 ancestors, the third one (3,905 tuples) produced 18,555 ancestors, the fourth one (9,330) produced 44,790 ancestors and the fifth one (19,607) produced 94,773 ancestors. The number of workstations ranged from one to five.

Figure 7.2 presents a summary of these results. The times spent within the union and the select/join processing are represented by the grey areas. The black areas concern the time used to connect and disconnect the application and the DBMS processes. They can be neglected for a large amount of tuples. Last, the white areas summarize the overhead introduced by the parallelism, i.e. the time spent loading and dumping relations and files, and synchronizing. These times are cumulated to illustrate the overall response time.

7.3.3. Discussion

Figure 7.2 shows the benefits and limitations of parallelism in our experiment. Two points have to be discussed.

The first one concerns the application. The curves show that the join and the union processing times are not divided by the number of processors. For LHS processing, the partition of the processing does not divide the work into the number of processor. The complete Parent relation has indeed to be read at each iteration on each server. The corresponding amount of time increases with the number of processor. The benefit is concentrated in the parallel generation of new tuples, without redundant production, which allows the response time to decrease with an increase in the number of processors. The RHS or the cost of the UNION operation is expensive. Intuitively, we would have expected better performances since no duplicate computation is done on individual nodes. Rebuilding the full ancestor relation from the individual partitions is responsible for these results. In fact, the UNION operation is used in the last rule, to merge an individual result to those already merged in the ancestor relation. In our example, this is done by rule r3. Better results should be obtained...
by using a relational MERGE operation instead of the UNION operation to obtain the final result. This is possible knowing that there are no duplicates among nodes.

The second remark concerns the overhead induced by the parallelism. This overhead is represented on Figure 7.2 by the white area. Our experiments show that its augmentation as a function of the number of processors, is limited. The parallelism is thus a good solution in our case. However, it is necessary to note that the transfers in our program are reduced to computation of the final result. In the case of more sophisticated algorithms, transfers are necessary during the ancestors generation; the overhead due to these transfers would increase, especially in our configuration (a network of workstations) which is not an ideal hardware to reach high performances of parallelism.

Efficiency of the parallelism with a grain of a rule, is largely influenced by the user's choices in his RDL/C program. There is a traditional tradeoff between a good partition of computation tasks and a limitation of the transfers.

We have also tried to evaluate the influence of the number of tuples on the benefit provided by using multiple workstations. Figure 7.3 draws the overall time as function of the number of basic tuples. The maximum benefit is achieved for three or four stations for any number of tuples.

![Figure 7.3: Overall time as a function of the number of basic tuples](image)

During the measurements, the database was duplicated on each station's disk. We have also experimented the storage of all the databases on a single disk; because a large main memory was available within each station, the results were rather identical.

8. Conclusion and Future Work

This paper presented a database rule language compiler which supports parallelism among rules. The type of parallelism supported by the language is user-specified, as opposed to user-transparent parallelism. The user partitions his application into independent tasks, which are scheduled to run in parallel. The compiler checks that the sets of rules to be processed on one DBMS processes which are on different workstations, each with its own database. Parallelism is achieved by having the application process fork into N processes, each one assigned a set of rules to be processed on one DBMS. Relations are transferred among the different DBMS by dumping relations onto files and loading relations from files using a common file sub-system managed by the network.

The main contribution of the paper are (i) the definition of a sub-language of control to specify a parallel execution of a rule program and (ii) the support of parallelism without specialized hardware and with standard relational technology. A prototype of RDL/C is operational over a set of UNIX workstations. Future works will study an automatic generation of the control structure which specifies the parallelism and the use of the RDL/C system to prototype parallel database algorithms that have been developed.

9. References


