A Deferred Communication in a Parallel Distributed Expert System Shell

Wided Lejouad

INRIA/CERMICS
2004, Route des Lucioles, BP 93
06902 Sophia Antipolis Cedex
lejouad@sophia.inria.fr

Abstract

In distributed knowledge based systems, in particular production systems, the high cost of inter-process communications may increase the execution time. To speed them up, researchers have proposed parallel implementations. We describe a prototype called DUESS that executes many rules simultaneously, runs in a deferred mode, and in which components are distributed on many processes at many sites. DUESS is novel because it reduces bottlenecks over synchronous parallel production systems, provides more time control to asynchronous communication where transmission delays are random, make available several object oriented databases, and improves concurrency over processes performing the *firing* step in parallel. The two problems of serialization and maintenance of consistent distributed databases are addressed. Estimates of the effectiveness of this approach are given.

1 Introduction

We start by giving our terminology of object oriented production systems. The structures and functionalities used are the following:

- **Rule Base:** a set of rules.
- **Object Base:** a set of objects.
- **Object:** an element of the object base.
- **Declarations:** contain the types of objects or classes.
- **Conditions:** the Left-Hand Side of a rule, determines when it matches with regard to a sequence of objects.

- **Conclusions:** the Right-Hand Side of a rule, determines what to do when the rule is executed (add an object, remove an object, or update its fields values).
- **Match:** for each rule determine the set of objects (instantiation) that match its Conditions.
- **Execute:** perform the conclusions of the rule for the selected instantiation.

Let's take an example of a production rule in the expert system shell SMECI [4] on which our study has been applied.

```
defrule color-office
   let *o an Office
   if  color of *o = ()
   then
       assign color of *o with LightGrey
end-rule
```

The serial execution model of object oriented production systems is presented in figure 1. These systems are often slow and require more speed improvements. Many researchers have studied parallel implementations focusing on the MATCH step in OPS5. They have studied systems that execute many rules simultaneously called *multiple rule execution systems*. These systems have two important problems: the difficulty to control the order of rule execution, and the inconsistency of the final results under any serial execution.

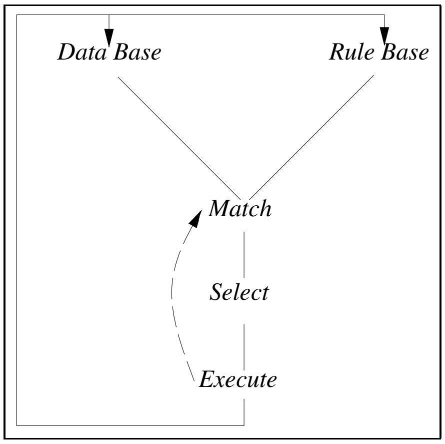

Figure 1: Serial Execution Model for Production Systems

Morgan [11] offers a way to control partially the order of rule execution but does not guarantee serializable results. Ishida [7] proposes a synchronization analysis algorithm which determines the necessity of synchronization between rule firings, and a decomposition algorithm which determines a mapping of rules on multiple processors. All of these multiple rule execution systems operate *synchronously*.

Schmolze and Goel [13] proposed a multiple rule execution system running in a highly *asynchronous* mode. This eliminates the time wasted in synchronous systems where some processors must wait for other processors before starting the next step. This approach makes possible the execution of all steps in parallel, and reduces bottlenecks. Unfortunately, it requires more control in distributing rules over many processors, and executing them in the working memory: conflicts may occur leading to inconsistencies. Moreover, with the asynchronous mode, the dependent events between processes are not well-controlled because of the variation of the data transfer time.

The object of this paper is to show how the *deferred* mode can supply more control and concurrency in a distributed environment including *dependent* or *not-dependent* processes [10]. This approach has been applied on a distributed expert system shell called SMECI. We start by a description of the execution model, and a brief definition of dependent and not-dependent processes with regard to our production system. Then, we describe our inter-process communication protocol and the way to resolve conflicts in the Objet Base. Finally, we give some evaluations of our parallel distributed application running in a deferred mode and we summarize weaknesses of our approach, pointing out directions for further work.

2 The Execution Model

We assume a set of processors $\mathcal{P}$ interconnected by a network. Any processor can communicate with any other and is able to store rules and objects. A Distributed Knowledge Based System (DKBS) is defined by the triple $(\mathcal{DKB}, \Gamma, \mathcal{T})$:

- $\mathcal{DKB}$ is a Distributed Knowledge Base including the Object Base and the Rule Base.
- $\Gamma = (\Upsilon, \Delta)$ is a pair of matching and executing programs residing at one site (on the same processor).
- $\mathcal{T} = \{\tau_1, ..., \tau_n\}$ is a set of transactions.

A distributed knowledge base consists in a set of objects Ω organized in a class hierarchy, and a set of production rules $\mathcal{R}$. The two sets Ω and $\mathcal{R}$ are disjointed and are considered as two distinct entities seperated physically. At present, most applications include a large object base which needs a large memory space. This is made possible in a distributed environment where multiple object bases are distributed over several processors (see figure 2). For simplicity, we consider that the DKBS contains only one Rule Base.

2.1 The Matching Program

In a distributed environment, the matching program Υ executes remote operations on the knowledge base. Let r be an element of the set of rules $\mathcal{R}$, $\mathcal{V}_r$ the set of variables in r, and ω a subset of the object base Ω. The operations performed by Υ are as follows:

$$\begin{array}{ll} receive & (r, \mathcal{R}) \\ read & (conditions(r)) \\ ask & (\omega, \Omega) \\ match & (\mathcal{V}_r(conditions), \omega) \end{array}$$

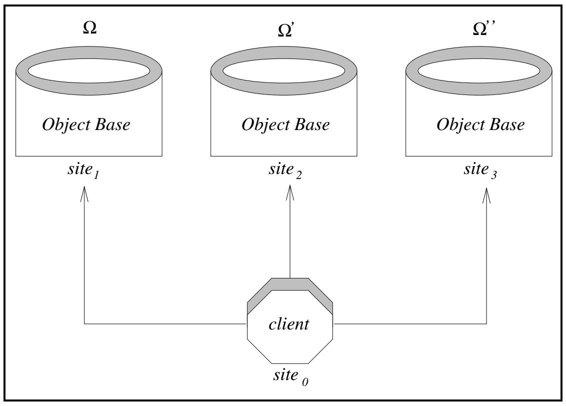

Figure 2: Distributed Object Bases

The program Υ receives one rule from the set $\mathcal{R}$ and consults its conditions. The conditions of a production rule are a set of predicates in which all variable occurences have to be instantiated. Υ performs a read access to the set of objects Ω. The order of operations performed by the matching program is of great importance for each set of conditions received (see figure 3). The read accesses can be seen as queries. A query q sent by Υ asks for the set of n-uplets of objects ω that would satisfy the rule predicates. The conditions of a rule are an input data for the Υ program which after consulting the set of objects Ω and checking the validity of the conditions, provides the substitution list in the following form:

$$\sigma = \{(\alpha_1, ..., \alpha_n) \mid \alpha_i \text{ instance of } v \text{ and } (\alpha_1, ..., \alpha_n) \\ \text{satisfy } Conditions(r) \ and \ v \in \mathcal{V}_r\}$$

If the matching process fails, σ will be empty and the rule r will not be selected. If it succeeds, r will be considered as an applicable rule which conclusions will influence the evolution of the object base.

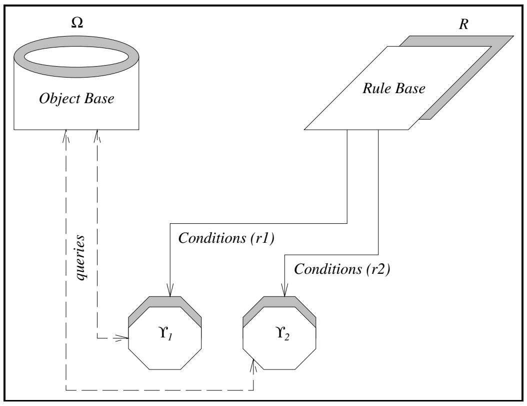

Figure 3: Multiple matching processes

In our distributed knowledge based system we assume that all the rules for which the matching step succeeds are selected and are considered as candidates to the resolution process. So, the Select step is the result of the Match step.

2.2 The Executing Program

The matching process Υ consists in filtering the expertise. Only rules that may give a solution to the problem are maintained. For simplicity, we are interested in a monotonic knowledge based system: for a selected rule r, the program Δ executes the rule conclusions which consist in creating an object in the object base Ω and/or assigning a value to an empty object field. In a distributed environment, the set of operations performed by the program Δ are the following:

$$\begin{array}{ll} receive & (Conclusions(r)) \\ receive & (\sigma) \\ update & (\Omega) \end{array}$$

If the matching process succeeds, the program Δ sends to the object base Ω some update commands which are executed locally. These write operations performed in the object base and associated with one rule can be seen as a transaction. Many conclusions of many rules can be executed at one time by providing multiple executing processes (see figure 4).

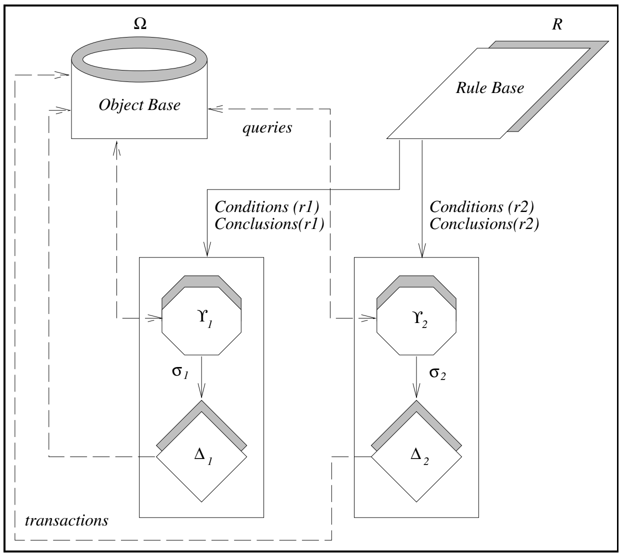

Figure 4: Multiple executing processes

2.3 The Transaction Model in a DKBS

Two principal operations can be performed by the executing program in the object base Ω: to create an object and to assign a value to an object field.

$$update(\Omega) \equiv \begin{cases} create(x, \Omega) \\ assign(x, v) : x \in \Omega \end{cases}$$

In this way, the set of operations in a transaction τ are restricted to two basic ones: the read operation performed by the matching process and the write operation performed by the executing process. After the brief description of a transaction in the DKBS, the operations performed by the executing process can be presented as follows:

$$\begin{array}{ll} receive & (Conclusions(r), \mathcal{R}) \\ receive & (\sigma, \Upsilon) \\ substitute & (\mathcal{V}_r(Conclusions)) \\ apply & (\tau, \Omega) \end{array}$$

In such distributed environment, parallelism is important to increase the system efficiency. We assume that given rules can be fired in parallel and are independent. Therefore, we propose briefly in the next section the dependency analysis between rules.

3 Rule Dependencies

Ishida and Stolfo first discussed the problem of maintaining consistency within a parallel rule-based system [7]. Their solution consists in detecting references by rules to classes of working memory elements (objects in our terminology). As illustrated in the figure 5, synchronization between rule A and rule B is required if there exists un set of objects Ω of class $\mathcal{C}$:

1. changed by rule A and referenced by rule B;
2. changed by rule B and referenced by rule A;
3. changed by rule A and changed by rule B.

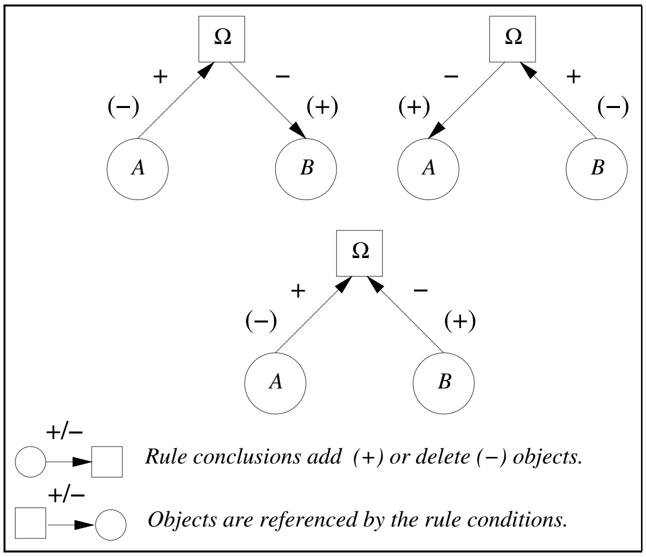

Figure 5: Synchronization Analysis between Production Rules

Their approach consists in constructing a Data Dependency Graph from which they determine the synchronization sets of rules. This approach has certain limitations: it is based on a static analysis and thus does not consider the case where class names are represented by variables. Also, in several cases the analyzer erroneously reports the necessity to synchronize a set of rules that could be fired in parallel. Moreover, this approach restricts parallelism: only one instance of each rule could fire simultaneously.

4 Process Dependencies

In this section, we describe process dependencies in a distributed environment and the way to preserve them without loosing efficiency. Our study has been applied on the expert system shell SMECI. The traditional cycle of inference can be summarized in four steps:

1. Decomposition of a problem into sub-problems called *tasks*.
2. Choice of a rule base to activate depending on the selected task.
3. Matching rules which are associated with the activated rule base.
4. Execution of the rule conclusions to deduce new facts.

We limited our study to the three last steps. Our distributed artificial intelligence system is based on a *client-server* model. From now on we call the object base Ω an *object server*, the rule base $\mathcal{R}$ an *expertise server*, the matching and executing programs an *inference engine* which plays the role of client/server in the same time: it asks for the set of objects, and it performs the matching and deduction services. The synchronous mode of interaction seems to be the most natural one. The scenario for a rule r would be the following:

- ◇ The expertise server sends the rule r to the matching process and *waits* for the final result (execution or not of r).
- ◇ From the rule declarations, the matching process determines the set of object classes $\mathcal{C}$ needed and sends a query to the object server asking for the list of objects that belong to the set $\mathcal{C}$. The matching process is then blocked waiting for the reception of objects. The object transfer protocol is out of scope of this paper, but we can affirm that it constitutes a complex task in which the semantic links between objects have to be considered.
- ◇ After the reception of the set of objects, the matching process tries to find the instantiation that matches the conditions. It returns the set σ. This set is empty if the matching process fails.
- ◇ If the matching process succeeds, the executing process sends the update commands to the object server which executes them locally.
- ◇ The expertise server is then released to send the next rule.

The synchronous mode makes the resolution slow by blocking some processes. The expertise server must wait for the end of processing of each rule before considering the next one, even when many inference engines are available. Moreover, when the matching process consults the object servers, it must send its query to the first server, wait for the result, and repeat this treatment as many times as there are object servers.

Therefore, the use of the asynchronous mode means that the matching process receives at the same time a number of rules even when it is busy. Then it has to manage the queue of rules and to deal with them following their arriving order: it comes to synchronize the rule processing. The expertise server can also send rules to some busy inference engines while others are free. Furthermore, with the asynchronous mode, the response sent by the object servers to the matching process is not controlled: it must be equal to the sum of the results of all consulted object servers.

Information exchanged between the matching process and the object servers are *objects*. These objects represent complex structures. Semantic links between them constitute a graph hardly transferable on a network. Our work was to find a network representation of an object (see figure 6), and to linearize the graph.

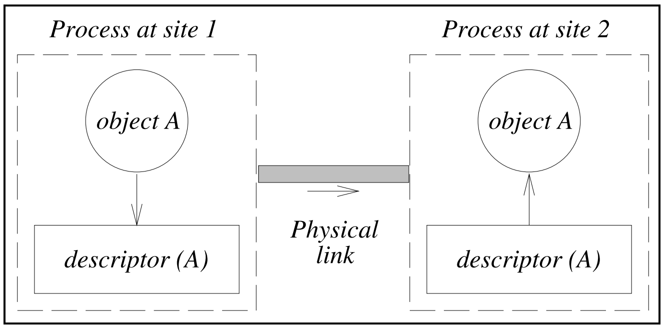

Figure 6: Object encoding and decoding

After describing the communication modes that are the most frequently used in distributed systems and their limitations, we propose a solution which consists to find a trade-off between an event synchronization and a parallel resolution. This is possible if we provide an interprocess communication in a deferred mode.

5 A Parallel Deferred Rule Matching

The production rule matching is the basic operation performed during a distributed expert system shell session. We propose here the way to match simultaneously a collection of rules defined in the expertise server. The number of inference engines and the names of processors on which they will be located are given by the user to the expertise server which creates them remotely. The expertise server is considered as the main process.

5.1 Matching Processes Behavior

Rules are sent in an asynchronous mode from the expertise server to the available matching processes. These processes become busy and the expertise server waits for the first process to be free. To announce that a matching process is free, this one sends its identification to the mailbox associated with the expertise server $\mathcal{R}$. After reading its mailbox, the expertise server knows the released process to which it can send the next rule. As we are in a distributed environment, the matching processes can either pass to a free state at the same time or have some time differences between them. This protocol is described in the figure 7. The principle of sending queries with

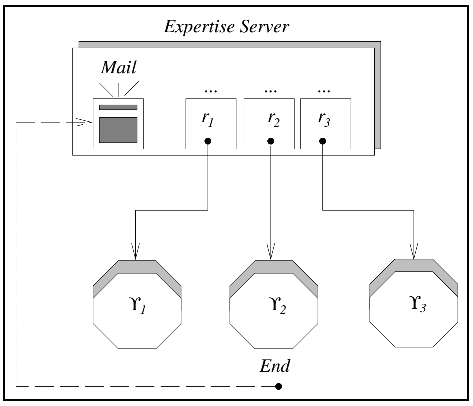

Figure 7: Deferred communication between the expertise server and the matching processes

returned results without locking the sender defines the *deferred mode*. This mode is implemented in the distributed environment manager called CHOOE [9] used in our application.

5.2 Object Servers Behavior

The presence of several object servers is authorized in our application. These servers are distributed on remote processors. Each matching process sends a query to all of these servers to get the set of objects needed to perform the rule matching. We have seen that with the synchronous mode, the matching process sends a query to the first server and has to wait for the result to ask the second server... With the deferred mode each matching process broadcasts its query to the object servers (or some of them) and waits for the responses. The responses are a network representation of a set of objects. After their reception, each matching process converts them to their initial representation and lets the executing process to work. This protocol is described in the figure 8. The object servers capable of giving a solution to the problem are grouped under a service name **S**. The query sent by every matching process has to include this service name, this allows all the object servers in **S** to receive the query.

6 A Parallel Asynchronous Rule Execution

Following each successful rule matching, the execution process Δ gets the set of substitutions σ and sends asynchronously to the eventual object servers the update commands which consist

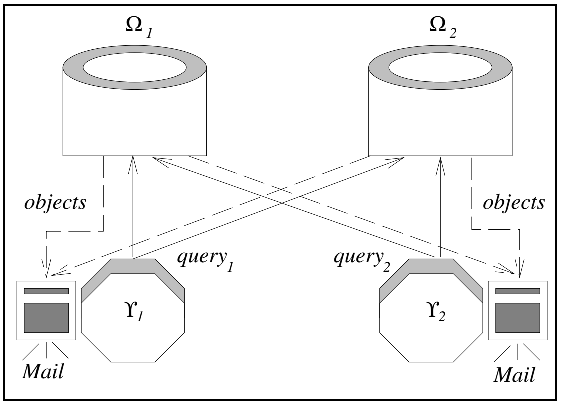

Figure 8: Deferred communication between matching processes and object servers

in applying the rule conclusions. Grouping the matching and executing processes at one site simplify widely the resolution: no more data transfer is needed.

6.1 Object Base Inconsistencies

Although we have limited our study to monotonic expert system shell, we analyze the case of interference between rules and the inconsistencies which occur when several distributed transactions update one or all object servers at a time. Let's take an example with the class hierarchy presented in the figure 9. Let Υ be a matching process on the processor $\mathcal{P}_1$ and Δ an executing

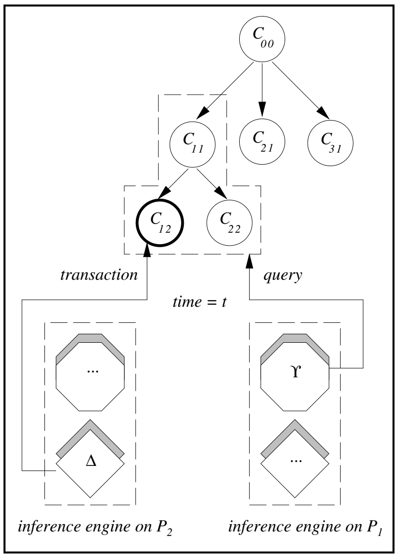

Figure 9: Concurrent access to an object server

process on the procesor $\mathcal{P}_2$. Υ sends a query and asks for the set of objects of the class C_{11} and so of all its subclasses C_{12} and C_{22}. The object server receives the query at a moment t. At the same moment, the process Δ executes a transaction, which consists in updating an object of the class C_{12}, on the same object server. As it is shown in the figure 9, the object to update is included in the domain of objects invoked by the matching process Υ: it is a *conflict*. The priority cannot be given to the read operation performed by Υ because the object will be invalid after the transaction execution.

The locking approaches used in Distributed Databases cannot resolve the inconsistency problem met in a DKBS. Theses approaches do not consider the case where the read operation preceeds the write one for the same data. The solution we propose consists in implementing *the algorithm of forward chaining to saturation*:

1. $C_\sigma \leftarrow$ Compute for each rule in $\mathcal{R}$ the set σ

2. repeat
  $C_{\sigma'} \leftarrow$ Compute for each rule in $\mathcal{R}$ the set σ
 until $C_{\sigma'} = C_\sigma$

This approach seems costly in running time. The rule base has to be read at least two times.

7 Performance Measuring

A distributed implementation based on the expert system shell SMECI exists in LE-LISP [3] using network tools like RPC-LINK [5] and CHOOE. We start by the evaluation of the distributed environment with its three basic processes: the expertise server, the object server and the inference engine [1]. After, we vary the number of inference engines as well as the number of the object servers in ordre to see the impact of the parallelism on the system efficiency.

The running times are those of a Sun SS10 processors with 64MB and 32 MB, and a Sun 4.50 processor with 32 MB. Each object server contains from 50 to 250 SMECI objects. Running times are given in seconds.

7.1 Impact of Parallel Rule Execution

The figure 10 gives the execution time of 5 to 100 rules in sequential mode (curve A), in parallel mode using two inference engines (curve B), and in parallel mode using three inference engines (curve C).

The use of multiple inference engines raises the system efficiency on an average of 20%. The object transfer phase through a network seems costly in our application. This can explain the gain diminution as the number of rules progresses: the more the number of rules increases, the more the object server is enriched and the more the object transfer becomes costly. Furthermore, when the number of inference engines increases (clients), each object server has to manage concurrent accesses and consequently to *serialize* queries and transactions set off by clients. Hence, the insertion of a new inference engine remains an intersting approach when the number of rules to fire becomes more important.

7.2 Impact of Parallel Access to Object Servers

The system performance is reduced in the case of parallel rule-firing with parallel access to several object servers. Our evaluation results show only a 15% of efficiency. As shown in figure 11, we

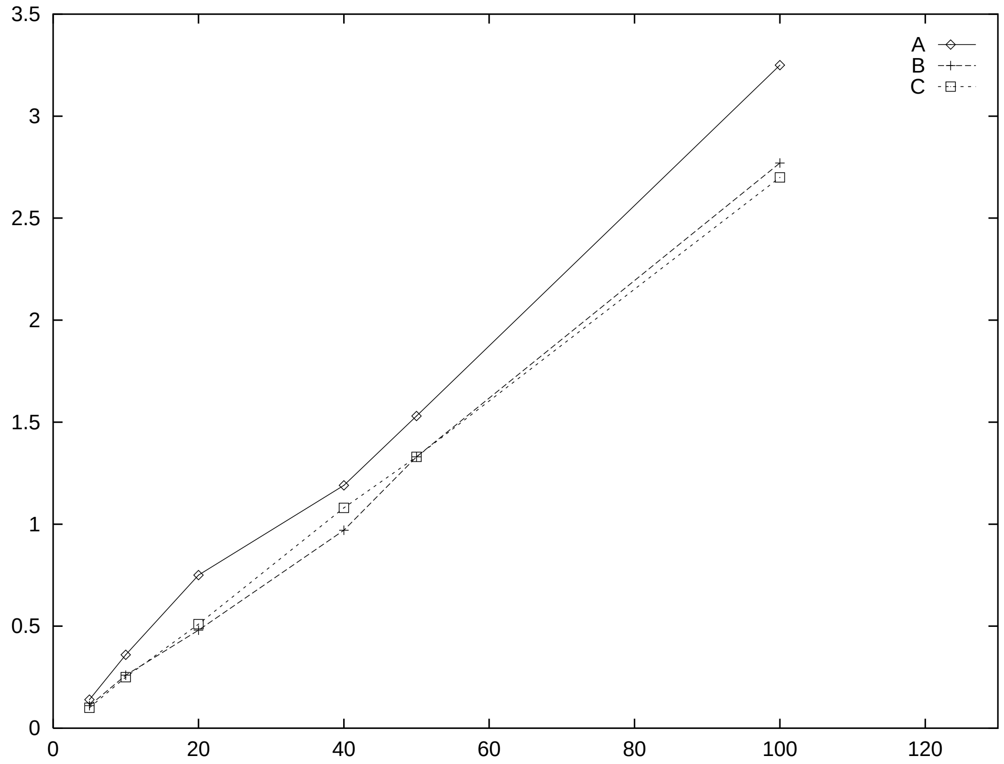

Figure 10: Running time for n rules with changing the number of inference engines

preserved 2 object servers and we varied the number of inference engines from 1 to 3 (respectively curve *1*, curve *2*, and curve *3*). This reduction can be explained by the fact that the more the processes are large, the more their scheduling is complex.

In return, in an environment containing only one object server, the use of three inference engines was not too efficient (see curve *C*). As illustrated in figure 11, the system performance is improved with two object servers and three inference engines (see curve *3*) since two kinds of parallelism are supplied: simultaneous rule-firing and multiple accesses to many object servers at a time.

A preliminary evaluation has been performed on the SMECI expert system shell. This result was contrary to our expectations. After further investigation, we think that the number of processors used is small, the effective parallelism is reduced to *time-sharing*. Processors used are not homogeneous, they have not the same computing power. The set of processes in our distributed environment are too big: since they are based on SMECI, a useless part of the expert system shell code is loaded. Shrobe &al. have shown the impact of the hardware environment in increasing software efficiency [14]. They assumed a system environment consisting of an ensemble of conventional uniprocessors connected by an Ethernet, and a hardware environment consisting of a shared-memory multiprocessor. They concluded that the first allows modest scale parallelism with a very high cost for initiating tasks, while the second allows much geater opportunity for parallelism for smaller grain-sized tasks.

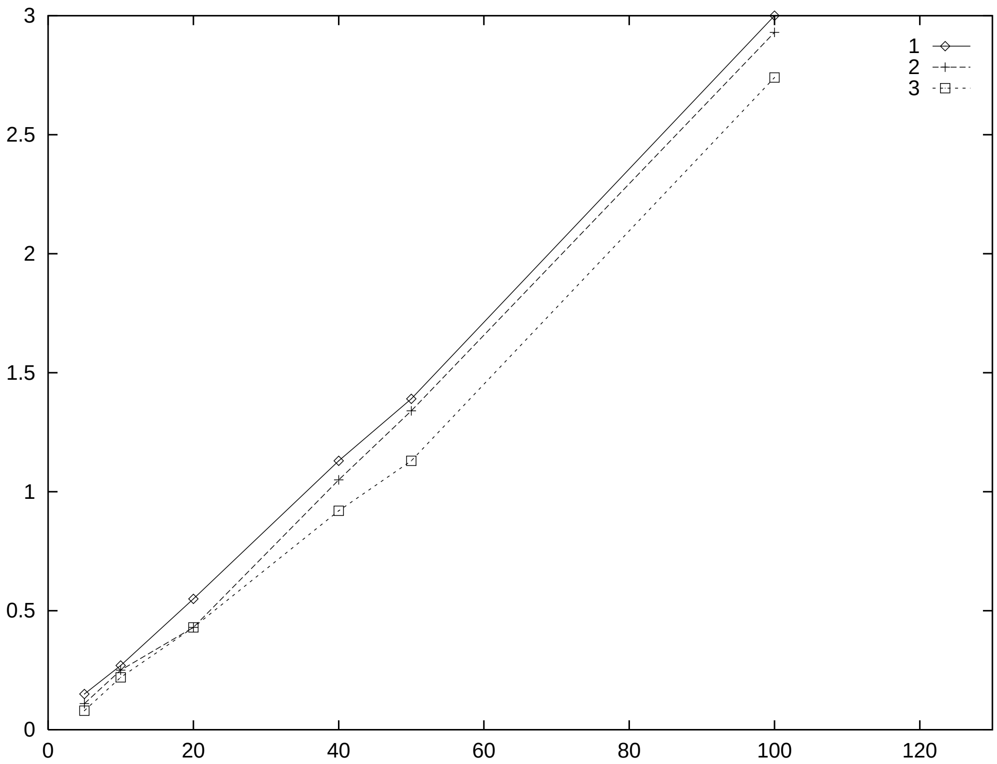

Figure 11: Running time for n rules with 2 object servers, changing the number of inference engines

8 Related Work

Previous studies on multiple rule execution systems have focussed on:

- The *Decomposition Algorithms* to find an optimal partition of a given production system [7, 2].
- The *Synchronization Algorithms* to determine what rules need to synchronize with another [7, 8, 13].
- The *Optimization Algorithms* to determine the best join structure for production system programs [6]. The inappropriate ordering of rule conditions causes a large amount of intermediate data, which increases the cost of subsequent join operations.

All of these studies have used the RETE algorithm in OPS5 and have been implemented in a distributed memory machines. Our research has focussed instead on the problem of *controlling* a parallel rule-firing system: we do not deal with the synchronization problem, we encourage the use of independent rules and we try to provide a solution when interferences occur. We employ a rule-firing policy called *fire-when-ready* as studied by Neiman in [12]. Therefore, we think that the scheme of read/write locks on working memory elements to maintain database consistency used by Neiman is not sufficient. Our algorithm of forward chaining to saturation provides more correct systems.

9 Conclusion

We presented a prototype of a parallel distributed expert system shell called DUESS. Novel about DUESS is that it (1) executes many rules simultaneously, (2) runs mostly in a deferred mode, (3) runs in an object oriented environment, and (4) constitutes a dynamic envionment of distributed processes. It reduces a number of bottlenecks and slowness found in synchronous multiple rule execution systems. The deferred mode used allows to find a trade-off between two important constraints: to insure a high degree of concurrency, and to maintain the order of events if it exists. Finally we will continue to work in a non-monotonic distributed expert system shell, with some improvement in the data transfer protocol. In this way, the speedups estimated for DUESS can be compared to other similar systems implemented on distributed memory machines.

References

[1] A.-M. Dery, W. Lejouad, and C. Djossou. How to Distribute an Existing Modular Expert System Shell? In *Proc. IEEE International Conference on Tools with Artificial Intelligence (TAI)*, Boston, 1993.

[2] S. Gallucci, J. Tan, and K. Hwang. PARTES: A Partitioning Scheme for Parallel Matching. In *Proc. IEEE International Conference on Tools with Artificial Intelligence (TAI)*, pages 374–379, 1993.

[3] ILOG, 2 Av. Galliéni, F-94253 Gentilly. LE_LISP *Version 15.22 : Le manuel de référence*, 1989.

[4] ILOG, 2 Av. Galliéni, F-94253 Gentilly. SMECI *: Le manuel de référence*, 1990.

[5] ILOG, 2 Av. Galliéni, F-94253 Gentilly. RPC LINK *: Manuel de référence*, 1991.

[6] T. Ishida. Optimizing Rules in Production System Programs. In *Proc. AAAI*, pages 699–704, 1988.

[7] T. Ishida and S. Stolfo. Towards the Parallel Execution of Rules in Production System Programs. In *Proc. International Conference on Parallel Processing*, pages 568–575, 1985.

[8] S. Kuo and D. Moldovan. Implementation of Multiple Rule Firing Production Systems on Hypercube. In *Proc. AAAI*, pages 304–309, 1991.

[9] F. Lebastard. CHOOE : un gestionnaire d'environnement distribué. Rapport Technique 93–22, Inria-Cermics, 1993.

[10] W. Lejouad. *Etude et application des techniques de distribution pour un générateur de systèmes à base de connaissances.* PhD thesis, Université de Nice, 1994.

[11] K. Morgan. BLITZ: A Rule-Based System for Massively Parallel Architectures. In *Proc. Conference for Lisp and Functional Programming.* ACM, 1988.

[12] D.E. Neiman. Control Issues in Parallel Rule-Firing Production Systems. In *Proc. AAAI*, pages 310–316, 1991.

[13] J. Schmolze and S. Goel. A Parallel Asynchronous Distributed Production System. In *Proc. AAAI*, pages 65–71, 1990.

[14] H.E. Shrobe, J.G. Aspinall, and N.L. Mayle. Towards a Virtual Parallel Inference Engine. In *Proc. AAAI*, pages 654–659, 1988.