J. Symbolic Computation (1995) **19**, 245–267

Distributed Deduction by Clause-Diffusion: Distributed Contraction and the Aquarius Prover

MARIA PAOLA BONACINA[††] AND JIEH HSIANG[‡‡]

[†] *Department of Computer Science, University of Iowa, USA*
[‡] *Department of Computer Science, National Taiwan University, Taiwan*

(Received 15 February 1994)

Aquarius is a distributed theorem prover for first order logic with equality, developed for a network of workstations. Given as input a theorem proving problem and a number n of active nodes, Aquarius creates n deductive processes, one on each workstation, which work cooperatively toward the solution of the problem. Aquarius realizes a number of variants of a general methodology for distributed deduction, called *deduction by Clause-Diffusion*, which appeared first in (Bonacina, 1992). The subdivision of the work among the processes, their activities and their cooperation are defined by the Clause-Diffusion method. Aquarius incorporates the sequential theorem prover Otter, in such a way that Aquarius implements the parallelization, according to the Clause-Diffusion methodology, of all the strategies provided in Otter.

In this paper, we give first a brief outline of the Clause-Diffusion methodology, with emphasis on the problem of *distributed global contraction*, e.g. normalization with respect to a distributed data base. We describe the schemes for performing distributed global contraction implemented in Aquarius, which avoid the *backward contraction bottleneck* of purely shared memory approaches to parallel deduction. Then, we describe Aquarius, its design, its features and user interface. We present a set of experiments conducted with Aquarius and we analyze the results. We conclude with some comparison and discussion.

1. Introduction

In this paper we describe the distributed theorem prover *Aquarius*, which implements a methodology for distributed automated deduction called *Clause-Diffusion*. Among the features of Aquarius, we illustrate in detail its mechanisms to perform *distributed global contraction*, which avoids the problem of the *backward contraction bottleneck*.

A theorem proving strategy C is given by a set of *inference rules* I and a *search plan* Σ. The inference rules can be further separated into two classes. The *expansion inference rules*, such as resolution and paramodulation, derive new clauses from existing ones and add them to the data base. The *contraction inference rules*, such as simplification and subsumption, delete clauses or replace them by smaller ones. The search plan Σ controls

[†] Supported in part by the GE Foundation Faculty Fellowship to the University of Iowa and by the National Science Foundation with grant CCR-94-08667. E-mail address: bonacina@cs.uiowa.edu.

[‡] Supported in part by grant NSC 83-0408-E-002-012T of the National Science Council of the Republic of China. E-mail address: hsiang@csie.ntu.edu.tw.

0747–7171/95/010245+23 $08.00/0

the selection of the inference rule and the premises at each step. The repeated application of Σ and I generates a *derivation*. A derivation is successful if it reaches a solution of the input problem. The main difficulty encountered in designing a theorem proving strategy is the question of controlling the search. Even if a strategy is complete in theory, if too large a portion of the search space has to be traversed before finding a proof, then it is still incomplete in practice.

A promising approach emerged in recent years is to employ powerful contraction inference rules. These inference rules are used either to delete redundant clauses or replace clauses by simpler ones, thus keeping the search space at a manageable level. *Contraction-based strategies* are particularly effective in the presence of the equality predicate. A typical example are the Knuth-Bendix type deduction methods (e.g., (Knuth & Bendix, 1970; Hsiang 1985; Hsiang & Rusinowitch 1987; Rusinowitch 1991; Bachmair et al. 1989; Bachmair & Ganzinger, 1990). We refer to (Dershowitz & Jouannaud, 1989; Bonacina, 1992) for more references). The effectiveness of these methods has been amply demonstrated in a number of successful provers, e.g. Otter (McCune, 1990), RRL (Kapur & Zhang, 1988) and SbReve (Anantharaman & Hsiang, 1990), all of which have obtained very impressive experimental results.

Although the ability to delete/replace data is the main reason why contraction-based strategies are effective, it is also the major source of difficulty in parallelization. First, because data are added and deleted, parallelization cannot take advantage of pre-processing techniques, which have been used effectively in Prolog-style deduction systems. We analyzed this and other issues in parallel deduction in (Bonacina & Hsiang, 1994b), and we reached the conclusion that coarse-grain parallelism is the most suitable for contraction-based strategies.

The second problem is that if a datum is contracted into a simpler one, the new datum may be used to contract other existing data. A typical example is when a rewrite rule is simplified to another rewrite rule, and the new rule is used to simplify all existing rules and equations. In order to keep the data base fully inter-reduced, if any equation/rule is simplified during the process, it is used again to simplify other equations/rules. This often results in an avalanche of contraction steps. In a shared-memory environment of parallelization, this may cause a serious bottleneck in the contraction process, which we call the *backward contraction bottleneck*. In our approach to distributed deduction, we avoid this problem by introducing a notion of *image set* – an approximation of the global data base. A number of ways of effectively using image sets to perform *global contraction* are implemented in Aquarius, which we describe in this paper.

The foundation of Aquarius, the *Clause-Diffusion methodology*, aims at parallelizing a strategy *at the search level*, by partitioning the search space among many concurrent deductive processes, which search in parallel for a solution. As soon as one of them succeeds, the whole distributed derivation succeeds. The deductive processes are asynchronous and work in a largely independent fashion: each process has its own local data base, constructs its own derivation and interacts with the others through *message-passing*. We refer to (Bonacina & Hsiang, 1995) for a full account of the Clause-Diffusion approach to distributed automated deduction.

Aquarius is built on top of the sequential theorem prover Otter. Aquarius implements a few of the variants of the Clause-Diffusion methodology for all the theorem proving strategies offered by Otter. Thus, Aquarius inherits most of Otter's valuable features. First, it exploits the high efficiency of basic operations and data structures, for which Otter is well-known. Second, Aquarius maintains the philosophy of Otter of providing the

user with a wealth of options to experiment with. New parameters related to distributed execution are added to those of Otter. This flexibility allows the user to tailor the prover to different classes of theorems, to use it to "simulate", to some extent, other approaches to distributed deduction such as the Team-Work method of (Avenhaus & Denzinger, 1993), and to apply it to other strategies, such as Knuth-Bendix completion. Third, Aquarius is highly portable, since it is written in C and PCN (Chandy & Taylor, 1991), under the Unix operating system, for a network of workstations. In such an environment, each deductive process runs on a different node of the network. We ran Aquarius on a number problems, which are reported in this paper. The experimental results are both positive and negative. For the latter, we analyze the possible causes, especially in terms of performance of communication, duplication of clauses and ways of partitioning the data base. We feel that negative experimental results are important, because they highlight the difficulties which remain to be solved and may contribute to further work.

In order to make the paper sufficiently self-contained, we decided to provide the reader with some knowledge on the motivations behind the design of Aquarius. This is done in Sections 2 and 3. Section 2 is a terse overview of the Clause-Diffusion methodology, which serves as the theoretical basis for Aquarius. Section 3 discusses some of the problems related to contraction in a distributed data base and the solutions adopted in Aquarius. Therefore, these two sections have some overlap with part of (Bonacina & Hsiang, 1995), which is the complete description of Clause-Diffusion. The remaining four sections, and thus the core of this paper, are devoted to Aquarius: the subtle points in its design and implementation, the experimental results and their analysis. Related papers on different topics are (Bonacina & Hsiang, 1993), on distributed fairness, (Bonacina & Hsiang, 1994a), on distributed subsumption, (Bonacina & McCune, 1994), on another Clause-Diffusion theorem prover, and (Bonacina & Hsiang, 1994b), which is a survey of existing approaches to parallel deduction.

ACKNOWLEDGEMENTS

The bulk of this work was done while both authors were with the Department of Computer Science of the State University of New York at Stony Brook. In addition to the support from the Department and the University, we acknowledge support from the National Science Foundation with grant CCR-8901322 given to SUNY Stony Brook. The first author was also supported by a fellowship of Università degli Studi di Milano, Italy. This paper is a revised and extended version of "Distributed Deduction by Clause-Diffusion: the Aquarius Prover", Miola, A. (ed.), *Proc. of the Third Int. Symp. on Design and Implementation of Symbolic Computation Systems*, Springer Verlag, Lecture Notes in Computer Science 722, 272–287.

2. Distributed theorem proving by Clause-Diffusion

The Clause-Diffusion methodology is designed mainly for parallelizing contraction-based strategies, although it also applies to other deduction strategies. In this section we describe how a complete theorem proving strategy $C =< I; \Sigma >$ is executed according to the Clause-Diffusion methodology. We consider a network of computers or a loosely coupled, asynchronous multiprocessor with distributed memory. The latter may be endowed with a shared memory component. Our methodology does not depend on a specific architecture; it can be realized on different ones. Parameters such as the amount of *memory*

at each processor, the availability of *shared memory* and the *topology* of interconnection of the processors or *nodes*, can vary.

The basic idea in our approach is to have a deductive process running at each node and *to partition the search space* among these processes. We use $p_1 \ldots p_n$ to denote both the deductive processes and the nodes. The search space is determined by the input clauses and the inference rules. At the clauses level, the input and the generated clauses are distributed among the nodes. For this purpose we use an *allocation algorithm*, which decides where to allocate a clause. Once a clause ψ is assigned to processor p_i, ψ becomes a *resident* of p_i. We also say that ψ belongs to p_i or p_i owns ψ. We denote by S^i the set of residents of p_i and the union of all the S^i's forms the current *global data base*. Each processor develops its own derivation by applying the inference rules in I to its residents, according to the search plan Σ.

The sets S^i's may not be disjoint, because a clause ψ may be generated in many ways and different copies of ψ may appear in different sets of residents. Thus, the subdivision into sets of residents is not a mutually exclusive partition. ¿From the point of view of automated theorem proving, however, no two clauses generated during a derivation are identical. First, each clause has its own set of variables and thus copies of clauses are not copies but variants. Second, a theorem prover typically needs to associate an identifier to a clause and if the same clause is generated more than once during a derivation, it gets each time a different identifier. If we keep these considerations into account, the S^i's form a partition of the global data base, that we call the *logical partition*.

Since the global data base is partitioned among the nodes, no node is guaranteed to find a proof using only its own residents. To ensure that a solution will be found when one exists, the nodes exchange information, by sending each other their residents in form of messages, termed *inference messages*. Each node uses the received inference messages to perform inferences with its own residents. The derivation generated by a process p_i is made of all the inferences performed by p_i on both residents and received inference messages. We remark that the data base at node p_i contains both the residents of p_i and the inference messages received by p_i. In other words, the *physical "partition"* of the data base is different from the *logical partition*, because a node physically stores clauses that do not belong to it. Clearly, the physical "partition" is not a partition in the mathematical sense of the word.

The purpose of the inference messages issued by a process p_i is to let the other processes know which clauses belong to p_i, so that they can use them for inferences. In a purely distributed system, inference messages are implemented as messages, which may be routed or broadcast. Depending on the broadcasting algorithm, there may be several inference messages, all carrying the same clause, active at different nodes. In a system with a shared memory component, inference messages may be communicated through the shared memory.

The separation of residents and inference messages is also used to partition the search space at the inference level. Using the paramodulation inference rule as an example of expansion step, we establish that the inference messages are paramodulated *into* the residents, but not vice versa. This restriction has two purposes. First, it distributes the expansion inference steps among the nodes. Second, it prevents a systematic duplication of steps: if this restriction were not in place, then each paramodulation step between two residents ψ_1 of p_1 and ψ_2 of p_2 would be performed twice, once when ψ_1 visits p_2 and once when ψ_2 visits p_1. Other expansion inference rules can be treated in a similar way (see Section 5.2).

In a contraction-based strategy, an expansion step should be performed only if all the premises are fully reduced. The contraction inferences to satisfy this key requirement can be classified into *forward contraction* inferences and *backward contraction* inferences. We call *raw clause* a clause newly generated from an expansion step. Input clauses are also considered as raw clauses. Forward contraction is the reduction of raw clauses. If a raw clause is deleted during forward contraction, it is not even added to the data base. Backward contraction is the reduction of all the other clauses, i.e. clauses that are already in the data base.

In a distributed contraction-based strategy, a first issue is whether contraction should be restricted based on the ownership of clauses. In Clause-Diffusion as we developed it so far, and in Aquarius, there is no subdivision of contraction steps based on ownership. The rationale for this choice is that the motivation behind contraction is to keep the data base always at the minimal and therefore it is better to allow each process to reduce as much as possible. A second, even more important issue, is that contraction needs to be done with respect to the global data base. Thus, Clause-Diffusion features a number of *distributed global contraction schemes* to enable a node to perform contraction with respect to a distributed set of clauses. We shall describe in part these schemes in Section 3.2 and we refer to (Bonacina, 1992) for a full account.

The distributed global contraction schemes are applied to both forward and backward contraction. In the distributed setting, contraction of residents and inference messages is backward contraction, while contraction of raw clauses is forward contraction. Only after forward contraction, a raw clause is entitled to become a resident at a processor. A clause that needs to be allocated is called a *new settler*. An allocation algorithm is used to assign a new settler to a node. Every process executes the allocation algorithm for its new settlers: it may decide either to retain a new settler or to send it to another node. The purpose of the allocation algorithm is to partition the search space and keep the work-load balanced as much as possible.

This is the basic working of the Clause-Diffusion methodology: expansion inferences among residents and inference messages, distributed global contraction (both forward and backward), which includes also the "local" contraction steps, e.g. between residents at the same node, allocation of new settlers and mechanisms for passing inference messages. By specifying the inference mechanism I, the search plan Σ to schedule inference steps and communication steps, the allocation algorithm, the distributed contraction scheme and the mechanisms for the communication of messages, one obtains a specific strategy. These elements are summarized in the following notion of *distributed derivation*: every processor p_k, $1 \leq k \leq n$, computes a derivation

$$(S; M; CP; NS)_0^k \underset{C}{\vdash} (S; M; CP; NS)_1^k \underset{C}{\vdash} \ldots (S; M; CP; NS)_i^k \underset{C}{\vdash} \ldots$$

where S_i^k is the set of *residents*, M_i^k is the set of *inference messages*, CP_i^k is the set of *raw clauses* and NS_i^k is the set of *new settlers* at p_k at stage i. The *state* of the derivation at processor p_k and stage i is represented by the tuple $(S; M; CP; NS)_i^k$. More components may be added if indicated by a specific strategy. A step in a derivation at a processor p_k can be either an inference step, *expansion* or *contraction*, or a *communication* step. For instance, sending an inference message for $\psi \in S^k$ from node p_k to an adjacent node p_j can be written as $(S^k \cup \{\psi\}, M^j) \vdash (S^k \cup \{\psi\}, M^j \cup \{\psi\})$. A distributed derivation is the collection of the asynchronous derivations computed by the nodes and it succeeds as soon as the derivation at one node finds a proof.

3. Contraction in a distributed data base

In distributed theorem proving, we call *global contraction* the task of reducing a clause with respect to the global data base, i.e. the union of the sets of residents of the parallel deductive processes. In Clause-Diffusion, the global data base is distributed and global contraction is done by the schemes for *distributed global contraction* introduced in (Bonacina, 1992). By allowing global contraction in distributed memory, these techniques offer a solution to the problem of the *backward contraction bottleneck*, which appears in shared memory implementations. In this section, we describe first this problem and then the mechanisms for distributed global contraction implemented in Aquarius. More details can be found in (Bonacina & Hsiang 1995).

3.1. THE BACKWARD CONTRACTION BOTTLENECK

We anticipated in the previous section how contraction steps can be separated into *forward contraction* and *backward contraction*. Designing an effective and efficient method for parallelizing a strategy which features backward contraction is a much more complicated task than for a strategy which employs only forward contraction. Indeed, backward contraction has turned out to be a critical problem for shared memory implementations (Lusk & McCune 1992; Yelick & Garland 1992) of parallel theorem proving with contraction, while some other implementations simply do not implement backward contraction (e.g. DARES (Conry et al., 1990) and PARROT (Jindal et al., 1992)). This problem emerges, although with less dramatic consequences than in theorem proving, when designing parallel implementations of the *Buchberger algorithm* (Chakrabarti & Yelick, 1993a; Chakrabarti & Yelick, 1993b; Hawley, 1991; Siegl, 1990; Vidal, 1990).

Forward contraction amounts to the normalization of a raw clause with respect to the *static* data base of all the clauses existing when the raw clause is generated. Thus, the task can be done once and for all when the raw clause is derived. Backward contraction involves the normalization of any clause with respect to all the clauses which may be generated afterwards. The normalization tasks need to be repeated as new clauses are generated. It follows that the data base is *highly dynamic* and there is *no read-only data*, i.e. all the items in the data base need be accessible not only for reading but also for writing. In turn, this implies that the clauses cannot be pre-processed into fast, specialized data structures, such as those used in approaches to parallel rewriting in equational programs, e.g. (Kirchner & Viry, 1992).

Furthermore, in contraction-based strategies, raw clauses are not used for expansion steps. Therefore, forward contraction does not enter in conflict with expansion. But backward contraction does, because it affects clauses that are already being used as parents of expansion steps. Finally, a clause which is reduced by a backward contraction step, should be tested for further contraction with respect to all the other clauses. Thus, *a single backward-contraction step may induce many*. In shared memory implementations such as (Lusk & McCune 1992; Yelick & Garland 1992), this avalanche growth of contraction steps causes a write-bottleneck, the *backward contraction bottleneck*, since all the backward contraction processes ask write-access to the shared memory, where all the clauses are stored. Not all of them may be served and an otherwise unnecessary sequentialization is imposed. The clauses which are supposed to be subject to backward contraction may not be made available for other tasks, e.g. expansion steps, so that these are delayed as well.

3.2. MECHANISMS FOR DISTRIBUTED GLOBAL CONTRACTION

In (Bonacina, 1992), we gave two classes of schemes for distributed global contraction: *global contraction by travelling* and *global contraction at the source*. In the first, we assume that no node has access to the global data base $\bigcup_{i=1}^{n} S^i$ and thus global contraction employs messages. In global contraction at the source, we assume that every node has access to an "approximation" of the global data base, so that a raw clause can be contracted at the node where it was generated. The choice of the appropriate global contraction scheme depends on the available resources: global contraction by travelling requires very fast communication, while global contraction at the source requires either sufficiently large local memories or a shared memory component to store the global data base. In this paper we describe a scheme called *global contraction at the source by localized image sets*, which is implemented in Aquarius.

In *global contraction at the source by localized image sets*, we assume that the local memory of each node p_i is large enough to hold an approximated version SH^i of the global data base $\bigcup_{i=1}^{n} S^i$. The SH^i are called *localized image sets*. The name "image set" says that such set contains "images", i.e. copies, of the residents in the systems. The attribute "localized" specifies that the image sets are in the local memories of the nodes. Each process uses its localized image set as set of simplifiers to perform global contraction of residents, raw clauses and incoming messages. (In *global contraction at the source by global image set in shared memory*, a single image set of the global data base is held in shared memory.)

The localized image sets can be built by utilizing the inference messages, that are already in place to allow non-local expansion inferences. It is sufficient to establish that *whenever a node p_i receives an inference message, it stores the clause carried by the message in SH^i*. Depending on the specific strategy, only those inference messages intended to be used as simplifiers need to be saved in the sets SH^i's. The identities $SH^j = \bigcup_{i=1}^{n} S^i$ for all j, $1 \leq j \leq n$, do not hold in general, because the sets of residents S^i's keep evolving. Thus, a localized image set is an approximation of the global data base. However, each of the SH^i's is logically equivalent to the global data base $\bigcup_{i=1}^{n} S^i$, if all the persistent residents, i.e. those not deleted by contraction, are broadcast as inference messages.

Our global contraction schemes do not suffer from the backward contraction bottleneck, because *the clauses being rewritten by contraction are held in the local memories of the nodes*. Therefore, concurrent contractions are done independently in the local memories at the nodes, with no need to wait to get write-access to a shared memory. An additional advantage of image sets is that such large sets of simplifiers can be implemented as *discrimination nets* (Christian, 1989; Stickel, 1989) for the purpose of fast simplification.

3.3. UPDATING THE IMAGE SETS WITH RESPECT TO CONTRACTION

Global contraction by image sets poses the fundamental problem of whether and how the simplifiers in the image sets should subject themselves to contraction. The question is whether the advantage of keeping the SH^i's fully reduced is worth the cost of updating them. In (Bonacina, 1992), we proposed several different approaches. In the following, we apply them to global contraction at the source by localized image sets.

1 *Maintenance by direct contraction*:

An obvious policy is to keep the elements in SH^i fully and inter-contracted by using SH^i and S^i within node p_i. This solution is conceptually simple and does not involve sophisticated record keeping. The main disadvantage is that there is redundancy of contraction steps, since the contraction of a clause ψ may be performed at all nodes which have a copy of ψ. The impact of such redundancy on performances will depend on the amount and complexity of the contraction steps prescribed by the strategy and on the efficiency of the implementation of contraction.

2 *No contraction of image sets*:
If the image sets are used only as a data base of simplifiers, one may choose to forbid contraction on the SH^i's. Only insertion of new elements is allowed. If $\psi \in SH^j - S^j$ is reducible, it may be reduced at the node p_i, such that $\psi \in S^i$, and a reduced form of ψ will be added to SH^j eventually. The rationale for this policy is that if SH^j is used only as a data base of simplifiers, the presence of both ψ and a reduced form ψ' does not represent serious redundancy. In fact, especially if the SH^j's are implemented as discrimination nets, frequent updates of the elements in the net may not be cost-effective. On the other hand, if the elements in SH^j are used for expansion steps, redundant clauses in the image sets would induce the generation of more redundant clauses. The next policy provides a mechanism to update the SH^i's with respect to contraction without resorting to the direct application of contraction inferences.

3 *Update by inference messages*:
We associate to every resident of a node a unique *identifier*: for every node p_i and for every resident ψ of p_i, ψ receives an identifier a, so that a is the unique identifier of ψ within the local data base S^i at p_i. It follows that $< p_i, a >$ is the unique *global identifier* of ψ over the network. We also establish that a resident ψ at p_i has another attribute, the *birth-time*, i.e. the time at p_i's clock when ψ was recorded as a resident of p_i. Overall the format of a resident is $< \psi, a, x > \in S^i$, where a is the identifier and x is the birth-time. The global identifiers of the residents can be used to index the clauses in the image sets. An image set may be implemented as a *hash table*, with the global identifier as key.
According to this labelling of residents, we assume that an inference message carries a clause together with its global identifier and birth-time. An inference message for a resident $< \psi, a, x > \in S^i$ has the form $< \psi, p_i, a, x >$. These additional fields allow a node to recognize that an inference message is carrying a reduced form of a previously received clause. If a resident $< \psi, a, x >$ of p_i is reduced at p_i to $< \psi', a, y >$, where x and y $(y > x)$ are times at p_i's clock, then a new inference message $< \psi', p_i, a, y >$ will be broadcast eventually. Whenever a node p_j receives an inference message, e.g. $< \psi', p_i, a, y >$, it checks whether an element ψ with the same global identifier $< p_i, a >$ is stored in SH^j. If this is the case, node p_j compares ψ and ψ' according to the strategy's ordering on clauses and saves the smaller in SH^j. (Contraction-based strategies feature a well-founded ordering on clauses used in contraction rules). If the two clauses are not comparable, the one with most recent birth-time is saved.

Update by inference messages does not apply if $< \psi, a, x > \in S^i$ is deleted, rather than replaced, by a contraction step. No more messages with identifier $< p_i, a >$ will be issued and therefore, localized image sets may never be updated. However, inference messages may still help: whenever an inference message $< \psi, p_i, a, x >$ is deleted at a node p_k, it is

possible to check whether SH^k contains any clause with identifier $< p_i, a >$ and delete it. This is not sufficient in general to update all the localized image sets, because clause ψ may not be deleted at p_k. Then, if performance is hindered by not updating the localized data bases with respect to deletions, one may consider broadcasting a special deletion message with identifier $< p_i, a >$ to inform all the nodes that the resident at $< p_i, a >$ has been deleted.

It is also possible to integrate different policies, in order to combine their positive features. The strategies implemented in Aquarius apply first update by inference messages and then direct contraction. Fewer direct contraction steps will be performed in general, if direct contraction is preceeded by update by inference messages. Also, deletion messages are not needed.

4. Clause-Diffusion strategies

In the previous sections, we briefly presented the Clause-Diffusion methodology by describing its objectives, essential operations and various unique features. We give a summary of operations performed by a strategy designed according to our methodology:

local expansion inferences between *residents* and between residents and *inference messages* (resulting in the generation of *raw clauses*),
local contraction of residents and inference messages,
global forward contraction of raw clauses,
global backward contraction of residents,
allocation of *new settlers*,
communication of messages.

For most of the operations we outlined a number of possibilities. A specific Clause-Diffusion theorem proving strategy can be formed by making specific choices from the various options described. In other words, a *Clause-Diffusion strategy* is specified by choosing

a set of inference rules,
a search plan that specifies the order of performing expansion, contraction and communication steps at each process,
the algorithm to allocate new settlers,
the scheme for global contraction,
the mechanism for message-passing.

In (Bonacina & Hsiang, 1993), we proved that the Clause-Diffusion methodology is correct: if $\mathcal{C} =< I; \Sigma >$ is a complete sequential strategy, its parallelization by Clause-Diffusion yields complete distributed strategies. Since in Clause-Diffusion all the concurrent processes have the given inference system I, parallelization does not affect the completeness of the inference system. Therefore, our correctness result consisted in proving that parallelization by Clause-Diffusion preserves the completeness of the search plan, i.e. its *fairness*. In the following, we describe the Aquarius theorem prover, an implementation of a specific class of Clause-Diffusion strategies.

5. The Aquarius theorem prover

Aquarius is a distributed contraction-based theorem prover, designed according to the Clause-Diffusion methodology. Aquarius has been developed on a network of workstations, so that each deductive process runs on a node of the network. Each process executes a modified version, called *Penguin* (Bonacina, 1992), of the code of the theorem prover *Otter* (version 2.2) (McCune, 1990). Therefore, Aquarius inherits the logic (first order logic with equality), the theorem proving approach (refutational, resolution-based theorem proving) and the strategies of Otter. The Penguin program is structured into a *communication layer* and a *deduction layer*. The communication layer, written in PCN (Foster & Tuecke, 1991), implements the message-passing part. The deduction layer, written in C, incorporates the code of Otter and implements the components required by Clause-Diffusion, such as the partition of the expansion steps, the distributed contraction scheme, the allocation algorithm, et cetera. The presentation of Aquarius in this section is organized in three parts: the communication layer, the deduction layer and the user interface.

5.1. THE COMMUNICATION LAYER

The communication layer is written in the language PCN (2.0) (Chandy & Taylor, 1991; Foster & Tuecke, 1991). PCN is a high-level language with parallel statements, guarded commands, recursion and primitives for communication. PCN allows to invoke C functions from within PCN functions. This makes it natural to use PCN for the communication interface and C for the computationally intensive work, e.g. deduction. We describe first the data structures and then the control for the communication layer.

Communication among the deductive processes is realized by using *streams*, a data structure provided by PCN. A stream is a data structure that permits communication of messages from a producer to one or more consumers. In Aquarius, streams are used to form a *fully connected virtual topology*: for any two processes p_i and p_j there is a stream with producer p_i and consumer p_j and a stream with producer p_j and consumer p_i. This is implemented as an $n \times n$ matrix C of streams, where $C[i,j]$ is the stream from node p_i to node p_j.

Streams are connected by means of two PCN primitives: the *merger* and the *distributor* (Foster & Tuecke, 1991). A merger implements many-to-one communication by merging many input streams into one output stream. A distributor implements one-to-many communication by placing the contents of one input stream onto many output streams. Each deductive process is equipped with a merger and a distributor: the process reads messages from the output stream of the merger and writes messages on the input stream of the distributor. Then, for node p_i, all the streams $C[k,i]$, $0 \leq k \leq n-1$, are connected as input streams to the merger of p_i, so that p_i receives all the messages from the streams $C[k,i]$. All the streams $C[i,k]$, $0 \leq k \leq n-1$, are connected as output streams to the distributor of p_i: a message emitted by p_i with destination p_j is sent on the stream $C[i,j]$, while a message intended for broadcasting is placed on all the streams $C[i,k]$.

Aquarius features both data messages, e.g. inference messages and new settlers, and control messages, e.g. termination messages. The latter should have higher priority. However, PCN streams are first-in-first-out queues, so that it is not possibile to differentiate the priority of messages placed on a stream. A simple solution is to define a second matrix D and reserve $D[i,j]$, $0 \leq i,j \leq n-1$ to control messages from node p_i to node p_j.

Then, the streams in D are serviced with higher priority than the streams in C, thereby ensuring that control messages have higher priority than data messages.

The communication layer is the outermost level of the Penguin program. It is structured into two main procedures, called *receive* and *main_infer*, invoked by a parallel statement:

$$receive() \parallel main\_infer.$$

Parallel statements in PCN are executed by interleaving. Thus, each process in Aquarius consists of some interleaving of receiving messages from the other nodes (the activity of *receive*), performing deductions and sending messages (the activities of *main_infer*). The *receive* procedure acts as a consumer on the input stream of the deductive process. It receives both control messages and data messages and stores the clauses contained in the received data messages in the *Inbound_messages* list. The *main_infer* procedure invokes the deduction layer, which may return one of the following:

1 a request of sending messages: *main_infer* resumes control and forward the messages generated by the deduction layer on the appropriate streams. In case of data messages, the clauses are extracted from the *Outbound_messages* list.
2 an exit code meaning that the deduction layer suspended, because its *Sos* (Set of Support: see next subsection) became empty: *main_infer* suspends too. It will resume, and re-start the deduction layer, as soon as the *receive* procedure receives data messages, carrying clauses that will be inserted in the *Sos*.
3 an exit code meaning that the deduction layer found a proof: *main_infer* broadcasts a message $< \texttt{halt}, i >$ and halts.

Upon receiving $< \texttt{halt}, i >$, all the other processes will halt as well. Synchronization of termination is also ensured by using streams: each deductive processes has a status stream for this purpose. Whenever a deductive process sends or receives a message of the type $< \texttt{halt}, i >$, it closes its status stream. When all the status streams are closed, Aquarius halts.

5.2. THE DEDUCTION LAYER

The deduction layer of the Penguin program inherits the inference system, the data base organization and the basic search plan of Otter. The expansion inference rules include binary resolution, factoring, hyperresolution (both positive and negative), unit-resulting resolution and paramodulation. The contraction inference rules include subsumption and simplification. For most of these rules a few restrictions and variations are implemented. Different subsets of inference rules may be selected by setting appropriate options.

The data base of clauses is divided into two main components, the *Set of Support* (*Sos*) and the set of *Usable* clauses. According to the Set of Support Strategy (Wos et al., 1965), each expansion inference step uses at least one parent from the *Sos*. The *Demodulators* list contains the equations to be used as simplifiers and the *Passive* list contains clauses to be used for forward subsumption and unit-conflict only. (A unit-conflict step is a binary resolution step which generates the empty clause.) In addition to these lists, that are the same as in Otter, there are the lists *Inbound_messages* and *Outbound_messages*, that we mentioned in the previous section.

The Aquarius program is invoked with the input file and the number of requested deductive processes as parameters. The user also specifies on which workstations in the network the deductive processes should be run. An input file for Aquarius has the same format as for Otter. A typical input file contains up to four list of clauses, i.e. the initial contents of the lists *Usable*, *Sos*, *Demodulators* and *Passive*, and the commands to set the options. In the input phase, one deductive process, e.g. p_0, reads the clauses from the input file, inter-reduces them, executes the allocation algorithm for each input clause and broadcasts all the input clauses to all the other processes. We remark that while all the input clauses are physically allocated at all the nodes, they are logically partitioned by the allocation algorithm: each clause belongs to a specific process.

The basic search plan prescribes the execution of a loop. At each iteration, a clause, termed *given clause*, is selected from the *Sos*. Before the selection of the given clause, all the clauses in *Inbound_messages* are moved to the *Sos*. In this way, the clauses received as inference messages from other deductive processes enter the *Sos* and take part in the deduction: the given clause can be either a resident or an inference message. Let $i_1, \ldots, i_n$ be the set of active expansion inference rules. For each rule i_k, $1 \leq k \leq n$, the deductive process, e.g. p_j, executes two phases:

1 It generates all the clauses, $\psi_1 \ldots \psi_n$, that can be derived by rule i_k from the given clause and any clause in the *Usable* list. Each newly generated clause, or raw clause, is forward contracted, ("pre-processed" in the terminology of Otter), right after having been generated. For instance, ψ_m, $1 \leq m \leq n-1$, is forward contracted before ψ_{m+1} is generated. If ψ_m is not deleted by forward contraction, the allocation algorithm is executed to determine its destination. If ψ_m, or possibly its reduced form, is allocated to p_j itself, ψ_m is appended to *Sos*. Otherwise, it is appended to *Outbound_messages* to be sent as new settler to its destination.
2 The clauses which have been just appended to *Sos* are applied to contract pre-existing clauses (backward contraction, or "post-processing" in Otter terminology) †.

These two phases are performed for all expansion rules in $i_1, \ldots, i_n$. If the given clause is a resident, a copy of it is appended to *Outbound_messages*, so that it will be broadcast as inference message‡. Each deductive process broadcasts only its own residents. Care is taken to prevent repeated broadcasting of the same clause: for instance, input clauses are not broadcast when selected as given clauses. Then the given clause is appended to *Usable* and the execution proceeds with the next iteration of the loop body.

The above control is the very basic search plan of the prover. Aquarius has many options, partly inherited from Otter, which allow the user to tailor this basic search plan into different variations. For instance, one may choose among several criteria to sort clauses in the *Sos*, criteria to retain or discard clauses, and orderings to orient equations, just to mention a few. The advantage of this organization, i.e. a basic search plan and many variations implemented as options, is that the user can experiment with a variety of

† Aquarius has an option, called **post-process-new-settlers-before-send**, that enables the new settlers destinated to other nodes to be simplifiers for a round of backward contraction.

‡ An option in Aquarius controls whether the given clause should be appended to *Outbound_messages* before or after the inferences with the given clause are performed. As a default, a copy is appended to *Outbound_messages* before the inferences. The clause will actually be broadcast later by the communication layer.

strategies. For instance, since not all combinations of options yield complete strategies, the user has the opportunity to play with incomplete strategies, which may be very interesting, as they may turn out to be especially efficient on specific problems.

5.2.1. THE EXPANSION INFERENCES

A characterizing feature of the treatment of expansion inferences in a Clause-Diffusion strategy is their subdivision among the deductive processes based on ownership. The subdivision is realized by establishing that only the owner of a clause can perform paramodulation steps into that clause. In other words, each process uses its residents and the received inference messages to paramodulate into its residents, but it does not paramodulate into inference messages.

This restriction can also be applied to expansion inference rules other than paramodulation. For binary resolution, we call positive-literal (negative-literal) parent the clause which provides the positive (negative) literal resolved upon, and we say that the positive-literal parent "paramodulates into" the negative-literal parent. Thus, a process will not resolve upon the negative literals of received inference messages. For hyperresolution, negative hyperresolution and unit-resulting resolution, we say that the satellites "paramodulate into" the nucleus. Accordingly, a clause is considered as a nucleus only if it is resident, while inference messages may serve as satellites. We recall that this partition does not affect the completeness of the strategy. Intuitively, those inferences that a process does not perform on inference messages are performed by the process which owns those clauses. A proof of this result was given in (Bonacina & Hsiang, 1993).

The implementation of the subdivision of expansion inferences in Aquarius utilizes the indexing techniques for term retrieval that Aquarius inherits from Otter. All inferences require some form of term retrieval (atoms and thus literals are regarded as terms for this purpose). Expansion inferences require to retrieve from the data base all the terms which are unifiable with a given term. Forward contraction inferences require to retrieve anti-instances: when one tries to reduce or subsume a given term, one looks for more general terms in the data base. Backward contraction inferences require to retrieve instances: when trying to apply a clause ψ to reduce or subsume other clauses in a data base, we look for instances of the terms of ψ. Otter, and thus Aquarius, employs *path-indexing* (Stickel, 1989) for retrieval of unifiable terms and instances, and *discrimination-net-indexing* (Christian, 1989; Stickel, 1989) for retrieval of anti-instances (McCune, 1988). (This is the default, which the user may modify by setting specific options.)

¿From the point of view of the implementation of inferences, the data base of clauses is regarded as a data base of terms: all clauses are indexed, i.e. their terms are inserted in path-indexes and discrimination nets. For each inference steps, the appropriate indexes are consulted: for instance, when resolving with a positive literal A, the prover consults a path-index to retrieve all the negative literals unifiable with A. Back-pointers from literals to clauses allow one to know which clauses a retrieved literal belongs to.

In this context, ownership-based restrictions to the application of expansion inferences can be implemented naturally by restricting accordingly the operations of clause indexing and term retrieval. For instance, only the terms of residents are inserted in the index of terms to be paramodulated into. For binary resolution, term retrieval will return negative literals only if they belong to residents. Similarly, when the term retrieval procedure is applied to search for the nucleus of an hyperresolution step, it will return only literals of residents.

5.2.2. THE CONTRACTION INFERENCES

Aquarius implements distributed global contraction by localized image sets (Subsection 3.2). The image sets are formed by saving the received inference messages. This is done simply by treating the received inference messages like the residents: a received inference message is stored in *Sos*, indexed, possibly appended to *Demodulators* and then moved from *Sos* to *Usable* after having been extracted as given clause. In this way, we do not add another data structure to implement the localized image set, but we implement it through the lists of clauses and the indexes (path-indexes and discrimination nets) that are already available. This approach keeps the code simple, by handling residents and foreign clauses as uniformly as possible.

The localized image sets are updated by using both *direct contraction* and *update by inference messages* (Section 3.3). The latter mechanism compares an incoming inference message with a clause in the localized image set having the same global identifier. Either one of the two clauses is deleted. The relative order of execution of this type of deletion (called *Discard Messages* (Bonacina & Hsiang, 1993)) and other contraction inference rules is relevant to the monotonicity of the inferences†. For instance, assume that p_i stores in its localized image set the equation $l \simeq r$, received as an inference message $< l \simeq r, p_j, a, t >$ from p_j. At a later stage of the derivation, p_i receives an inference message $< l \simeq r', p_j, a, t' >$, which has the same global identifier of $l \simeq r$ and carries a reduced form $l \simeq r'$ of $l \simeq r$. If Simplification is applied before Discard Messages, $l \simeq r$ may reduce $l \simeq r'$ to $r \simeq r'$ and then $l \simeq r$ would be deleted by Discard Messages because of $r \simeq \dot{r}'$. The result is non-monotonic, since both $l \simeq r$ and its reduced form $l \simeq r'$ are lost. The cause of the problem is that $l \simeq r$ is applied to simplify its reduced form $l \simeq r'$. Such a phenomenon could never happen in a sequential computation, where the unique copy of $l \simeq r$ would have been deleted upon the generation of $l \simeq r'$. This incorrect behaviour can be prevented by applying Discard Messages before the other contraction rules.

A new feature of Aquarius is the implementation of *Distributed Subsumption*: in (Bonacina & Hsiang, 1994a), we observed how the unrestricted application of subsumption may violate the fairness, hence the completeness, and the monotonicity of a distributed derivation. The distributed subsumption inference rules of (Bonacina & Hsiang, 1994a) allows to perform subsumption, including subsumption of variants, without causing these problems. We refer to (Bonacina & Hsiang, 1994a) for a full treatment of subsumption in distributed derivations.

5.2.3. THE DISTRIBUTED ALLOCATION OF CLAUSES

The basic allocation algorithm in Aquarius implements a simple idea of rotation: if p_i is the most recently selected destination, the next destination will be p_j, where $j = (i + 1) \ mod \ n$, for n the total number of nodes. We recall that in Clause-Diffusion there is no centralized decision-making: the allocation algorithm is executed independently by all processes. Thus, each process saves its most recently selected destination and picks the next choice accordingly.

A few options allow the user to variate the basic allocation algorithm, e.g. by increasing

† *Monotonicity is the dual property of soundness: an inference step is monotonic if it preserves the* set of logical consequences of the given set of clauses.

the number of clauses that the processes allocate to themselves. For instance, the options `own-in-usable` (`own-in-sos`) enables each process to keep as residents all the input clauses in *Usable* (*Sos*). These two options proved to be valuable experimentally: basic axioms, such as associativity, commutativity or distributivity, are generally given in the input *Usable* and it is clearly very useful, sometimes necessary to obtain a proof in reasonable time, that such axioms are owned by all the processes. Two more options that apply the same philosophy, although just to special types of raw clauses, are `own-factors` and `own-new-function-rules`. The first one induces each process to allocate to itself the factors of its residents. The second one causes each process to keep as residents the equations generated from its residents by the *New Function Rule*, i.e. "splitting" as defined in the original paper by Knuth and Bendix (Knuth & Bendix, 1970).

5.3. THE USER INTERFACE

Similar to Otter, Aquarius is not interactive during the derivation: given the input, the program runs to completion, generally without any significant interaction with the user. The motivation for this characteristic is that we want to obtain proofs that are fully automated, with no human intervention during the run. Interactivity with the user is concentrated during the preparatory phase, when the user sets the options for the experiment. Accordingly, these provers have a very high number of options. Otter 2.2 has 96 options, which affect several components and features of the prover: the inference mechanism, the search plan, the amount and type of the information recorded in the log file of a run et cetera. Aquarius adds 25 new options, controlling communication, distributed allocation of clauses, priority of communication versus deduction, criteria to extract clauses from the *Sos* and more. As attested by the popularity of Otter, the high number of options does not make these provers difficult to use. The main reason is that given a class of problems, e.g. purely equational problems, the setting of most options is the same for almost all the problems in the class, while the user may still play with the more subtle options on each problem. During a typical experimental session, the user will run the prover with a standard configuration of options, and then, depending on the outcome, proceed to modify the configuration, usually one option at a time, and repeat the run. Thus, even if the options are many, in practice the user needs to concentrate on very few of them at one time. Different settings of the options define different strategies and therefore the result, whether a proof is found and in how much time, is clearly influenced by the selection of the options.

The user may set different options patterns, and thus different strategies, at different deductive processes. This flexibility allows the user to induce Aquarius to reproduce interesting features of other methods. For instance, by having different strategies at different nodes, Aquarius may "simulate" to some extent the *Team-Work method* of (Avenhaus & Denzinger, 1993), albeit without the "referee processes" and the periodical reconstruction of a common data base that are characteristic of the Team-Work method. The `saturation` option (which is basically the `knuth-bendix` option of Otter) prescribes to perform (unfailing) Knuth-Bendix completion, so that Aquarius executes Knuth-Bendix completion in parallel. The `stand-alone` option induces a mode of execution where each deductive process works by itself as a sequential prover, with no message-passing. One purpose of this option is to try in parallel different strategies on a given problem. Another application is to give to each deductive process a different input and have the nodes work-

ing in parallel on different problems. For instance, one may want to give to each process a different lemma from a large problem and have the lemmas proved independently.

6. Experiments with Aquarius

In this section we report the results of some experiments performed with Aquarius at Stony Brook. The experiments were conducted on three Sun 4 Sparc workstations connected by the Ethernet of the department. At the time when the experiments were conducted, only up to three workstations were available for experiments with Aquarius. The Sparc-stations used for our experiments were not isolated from the rest of the network and were simultaneously used by other users. Therefore the reported run times represent the performances under realistic working conditions.

In Table 1, Aquarius-n is Aquarius with n nodes and run times are expressed in seconds. For Aquarius-1 the run-time is that of the best run found. For $n > 1$, the run-time of Aquarius-n is the run time of the first node to succeed, which includes both inference time and communication time. However, it includes neither the initialization time spent to set up the PCN processes at the nodes nor the time spent to close all the PCN processes upon termination. Thus, the turn-around time observed by a user is usually longer than the run time. The other processes run till either they receive a halting message or also find a proof, whichever comes first. Among the listed problems, two are propositional (*pigeon* and *salt*), four are purely equational (*luka5*, *robbins2*, *s7* (a problem in algebraic logic) and *w-sk*), two are in first order logic with equality (*ec* and *subgroup*) and the remaining ones are in first order logic.

6.1. ANALYSIS OF THE EXPERIMENTS

Table 1 shows mixed results. On problems *cd*12, *cd*13, *cd*90, *ec*, *imp*1, *imp*2, *imp*3, *luka*5, *s*7, *sam lemma* and *subgroup*, that is more than half of the tests, Aquarius shows some speed-up. The speed-up is approximately linear, e.g. Aquarius-2 versus Aquarius-1 on problems *cd*12, *cd*13, *imp*1, *imp*2, *imp*3, and Aquarius-3 versus Aquarius-1 on problem *s*7, or even super-linear, e.g. Aquarius-2 versus Aquarius-1 on problems *luka*5 and *s*7. On the remaining problems the run-time remains approximately the same, partly because those problems which can be solved sequentially in a few seconds are probably too easy for the parallelization to pay off. However, in some cases where Aquarius shows some speed-up, it happens that Aquarius-2 speeds up over Aquarius-1, but Aquarius-3 does not improve or even worsen the run-time. Many factors contribute to these results. First, Aquarius is a prototype, which was developed in a short five months period. Second, Aquarius-1 is generally slower than Otter, which indicates that the overhead induced merely by having embedded the C part in the PCN part is not irrelevant. Third, most of the above problems are taken from the input sets for Otter and ROO (Lusk & McCune, 1992) and therefore problems in first order logic prevail over problems with equality. On the other hand, problems with equality are those where the impact of backward contraction, the backward contraction bottleneck and its avoidance by Clause-Diffusion, is most dramatic. In the following, we analyze in more detail the performances of Aquarius in terms of *communication*, *duplication* and *distribution of clauses*.

Table 1. Experiments with Aquarius

Problem	*Aquarius-1*	*Aquarius-2*	*Aquarius-3*
cd12 (Lusk & McCune, 1992)	104.18	50.98	47.56
cd13 (Lusk & McCune, 1992)	98.79	45.32	51.07
cd90 (Lusk & McCune, 1992)	3.10	0.63	11.87
cn (Lusk & McCune, 1992)	5.04	8.63	14.50
ec	3.03	1.96	1.77
imp1 (Lusk & McCune, 1992)	6.63	2.64	3.54
imp2 (Lusk & McCune, 1992)	7.25	3.31	7.43
imp3 (Lusk & McCune, 1992)	32.05	17.92	38.89
luka5 (Bonacina, 1991)	844.20	299.24	1079.45
pigeon (ph4) (Pelletier, 1986)	8.21	7.66	8.14
robbins2 (Lusk & McCune, 1992)	21.62	22.91	24.12
s7 (Wasilewska, 1993)	630.62	208.37	192.54
salt	3.89	4.45	5.49
sam's lemma	6.35	5.40	3.90
subgroup (Wos, 1988)	15.55	9.36	17.40
w-sk (McCune & Wos, 1988)	3.50	3.52	3.34

6.1.1. OBSERVATIONS OF COMMUNICATION PROBLEMS IN AQUARIUS

Communication in Aquarius appears to be slow, not necessarily because the Clause-Diffusion method generate too many messages for the network, but rather because the implementation of message-passing in PCN appears to be slow in delivering each message. An evidence of this is that some delay may already be observed in the broadcasting of the input clauses, when still very few messages have been generated. Indeed, it happens that the process which reads and broadcasts the input clauses is in many cases the first one to succeed. Also, it happens that p_1 and p_2 have shorter run times than p_0, simply because the start of the derivations by p_1 and p_2 is delayed by the necessity of waiting for the input clauses. Another evidence that communication is hindering the performances is the following. Let ψ be a clause which can be derived independently at two nodes, e.g. p_0 and p_1. In many runs, it happens that p_0 generates and broadcasts ψ, but p_1 derives it on its own, *before* receiving the inference message from p_0. The intuitive idea of inference messages in the Clause-Diffusion methodology is that in general the clause carried by the message is new for the receiver. Therefore, when the above phenomenon appears in Aquarius the purpose of the inference messages is sort of defeated.

The performance of Aquarius is affected by the implementation of communication in PCN in at least two ways:

1 PCN version 2.0, that was used for Aquarius, gives priority to the execution of C code over the execution of PCN code.
2 The communication done through PCN and Unix is hampered by too many levels of software, causing too much copying for each message.

The effect of the first problem is that no PCN message-passing will take place until the C code completes. The producers of messages, i.e. the deduction layers of the processes, are written in C, while the consumers, i.e. the communication layers, are written in PCN. It follows that a consumer may not be scheduled from the active queue to get its pending messages while the C code is being executed at the node. Therefore communication, which is already likely to be the potential bottleneck in a distributed implementation, is at a strong disadvantage with respect to inference. The producers generate messages at a much faster pace than the consumers may consume them. Indeed, we observed executions, where the inference part of the computation halts upon finding a proof and then several pending messages are delivered all together.

We countered this problem by reducing the size of the C subprocesses, i.e. by causing the deduction layer to suspend more frequently. If the deduction layer (which is C code) suspends, the communication layer (PCN code) may resume and consume the pending messages. In a first implementation of Aquarius, the deduction layer would not suspend until it needs to send messages or it found a proof or its *Sos* is empty. This was modified so that the deduction layer would suspend after processing every given clause. This modification brought an improvement in most experiments, but it does not seem to have been sufficient. An alternative approach is to synchronize the communication and deduction layers within each node. Currently, they are largely asynchronous. A possible synchronization is to let the deduction layer proceed only when all the pending messages have been received by the communication layer.

6.1.2. DUPLICATION

After having experienced these problems with communication, we resorted to try to reduce their impact by reducing the amount of communication, that is, by empowering the single nodes. If communication is slow, it is better that all nodes are able to work as independently as possible. Some of the reported experiments were done by setting the options affecting the allocation of clauses in such a way that each node owned most of the input clauses. In other experiments, the chosen allocation options induced each node to retain most of its raw clauses as residents. None of the reported results, however, refer to executions where each node could keep all the clauses: neither the `stand-alone` mode nor any combination of options having the same effect were used. In all the listed experiments, there was some partitioning of the search space.

Option patterns that reduce communication, combined with the use of localized image sets, induce an increase in duplication. It appeared from the trace files of the experiments, that often many of the clauses needed in the proofs were generated independently at all nodes. For instance, in one run of the problem *cd*90, the clause $P(e(e(x,y),e(x,y)))$ appeared in the trace of the execution at p_2 as follows: first, it is generated and sent as new settler to p_0; second, it is generated again and kept as resident; third, it is received

as inference message from p_0; fourth, it is generated one more time and sent as new settler to p_1; fifth, it is received as new settler. Finally, $P(e(e(x,y),e(x,y)))$ is subsumed by $P(e(x,x))$. This amount of duplication is one of the causes of the lack of speed-up.

The Clause-Diffusion methodology and Aquarius are sufficiently flexible to provide combinations of different degrees of communication and duplication. However, the Aquarius prover and the option patterns used in the experiments reported here realized a duplication-oriented version of Clause-Diffusion, which was not intended to be the main one, since it reduces the significance of partitioning the search space. The basic idea in the Clause-Diffusion methodology is to partition the search space. Indeed, the cases where Aquarius-2 speeds-up significantly over Aquarius-1 are exactly those where partitioning the search space helps. More precisely, in most of the positive results, one of the concurrent processes finds a shorter proof than the one found by the sequential prover, because it does not retain some clauses. An example is *cd*90, where Aquarius-2 has super linear speed-up over Aquarius-1. The latter finds an 8-steps proof, which uses first $P(e(e(x,y),e(x,y)))$ and then $P(e(x,x))$. Aquarius-2 finds a 5-steps proof, which uses $P(e(e(x,y),e(x,y)))$ and does not even generate $P(e(x,x))$.

6.1.3. DISTRIBUTION OF CLAUSES

The criteria for distributed allocation of clauses implemented in Aquarius try to balance the work-load by balancing the number of residents at the nodes. They keep into account neither the contents of a message, i.e. the clause, nor the history of the derivation, in order to decide its destination. The design of more informed allocation policies, e.g. policies which use informations about the clause being allocated and the history of the derivation, may be an important progress. As an example, one may think of heuristics of the form: if more than n clauses with property Q have been allocated to node p_i, then the next clause with property Q will (or will not, depending on Q) be allocated to p_i. Such criteria, however, will be more expensive to compute and it may not be simple to devise them. More generally, the question is how to find better ways to partition the search space of a theorem proving problem.

6.2. COMPARISON WITH RELATED WORK

Few parallel theorem provers for contraction-based strategies have been implemented so far. The closest to Aquarius are probably ROO (Lusk & McCune, 1992) and DISCOUNT, that implements the Team-Work method for equational logic (Avenhaus & Denzinger, 1993). Few experiments were reported in (Avenhaus & Denzinger, 1993). A comparison of Team-Work and Clause-Diffusion is given in (Bonacina & Hsiang, 1995). Therefore, we consider here ROO, which has the same logic as Aquarius, first order logic with equality, and was also conceived as a parallelization of Otter. ROO showed linear speed-up on most non-equational problems, but its performances on equational problems suffered from the backward contraction bottleneck. ROO uses *parallelism at the clause level*, since each concurrent process consists in selecting and processing a given clause. A common data base of clauses is kept in shared memory and thus the search space is not partitioned. Such a purely shared approach to parallel theorem proving, with parallelism at the term/clause level, does not modify the search space (and does not intend to). Thus, the parallel prover works on a search space which is basically the same as in the sequential case and it is

likely to find a similar proof. The parallel prover speeds-up over the sequential one by generating faster the same proof and the results are rather regular.

Our philosophy is quite different. Because of the high degree of dynamicity of the data base induced by backward contraction, we adopted coarse-grain parallelism. In order to avoid the backward contraction bottleneck, we chose to work in distributed memory. In this context, we aim at parallism at the search level by partitioning the search space. Then, the concurrent processes deal with search spaces that may be radically different from that of the sequential prover. For instance, in Aquarius, it is sufficient that a process does not retain a certain clause and sends it to settle at another node to change its search space dramatically. By considering a different portion of the search space, a shorter proof may be found. In such cases, the distributed theorem prover speeds-up considerably. However, if the search space turns out to be partitioned in a way that does not reveal a shorter proof, the distributed prover is at a strong disadvantage, as it may try to generate the sequential proof from a fragmented search space. The irregular results are the consequence of this kind of phenomenon.

7. Discussion

In the first part of the paper, we outlined our methodology for distributed deduction by Clause-Diffusion, which is the foundation of the Aquarius theorem prover. The basic idea of this methodology is to partition the search space of the problem among concurrent, asynchronous deductive processes, which search for a solution in a largely independent fashion. This approach realizes a sort of coarse-grain parallelism, that we termed *parallelism at the search level* (Bonacina, 1992). Our methodology does not exclude the application of techniques for fine-grain parallelism, such as those employed for parallel rewriting languages, e.g. (Kirchner & Viry, 1992). While the Clause-Diffusion methodology applies to theorem proving strategies in general, it is designed especially for contraction-based strategies. In previous work (Bonacina & Hsiang, 1995), we formulated the problem of global contraction with respect to a distributed data base, clarifying the differences between forward global contraction and backward global contraction, and indicating in the *bottleneck of backward contraction* a critical problem in the shared memory approaches to parallel automated deduction. In (Bonacina, 1992), we proposed as solutions several schemes for distributed global contraction. In this paper, we focused on *global contraction at the source by localized image sets*, since it is the scheme implemented in Aquarius.

In the following sections, we described Aquarius and analyzed some experiments. Aquarius showed speed-up on most of the reported problems, especially those involving equations and requiring more than a few seconds to be solved sequentially. On the other hand, we did not obtain speed-up on all problems and on some problems Aquarius-2 improved over Aquarius-1, but Aquarius-3 did not. At the operational level, the main cause for the mixed results of Aquarius is the inefficiency of communication. At least part of the problem is related to the choice of the PCN language, which perhaps was not designed for the parallelization of a large, computation-bound C program, such as Otter. The problem with communication may represent evidence in favor of a less distributed version of the Clause-Diffusion methodology. Because of the use of localized image sets, Aquarius implements a *distributed duplication-oriented approach*. If a shared memory component is available, one may choose global contraction at the source by image set in shared memory (see Section 3.2 and (Bonacina, 1992)) and obtain a *mixed*

shared-distributed approach. This approach reduces the amount of both communication, because exchange of messages may be replaced in part by access to the shared memory, and duplication, because just one image set is maintained. On the other hand, if a single image set in shared memory is used, the search spaces considered by the different concurrent processes may turn out to be less differentiated than in the more distributed approach of Aquarius. Thus, the results might be more regular, but also, in a sense, less challenging than in Aquarius. The latter probed a new thinking of the parallelization of search problems in general, whose success may require a better understanding of the parallelization of search.

Our analysis is based on empirical observations. A theoretical analysis in terms of computational complexity would be desirable in principle, but it is not possible at this early stage of research in parallel theorem proving. In fact, a fine-grained algorithmic complexity analysis is not usually done even for sequential theorem proving strategies. One reason is that a theorem proving strategy embeds several algorithms, including matching, unification, term retrieval, term replacement, term ordering, proof reconstruction, and more, so that a fine-grained analysis should combine the complexities of all these components. Furthermore, a theorem proving strategy is a semidecision procedure, so that the classical concept of worst-case analysis may be applied separately to the inner algorithms, but does not apply naturally to the strategy as a whole. Beside these technical difficulties, the main problem is that the complexity of theorem proving is in essence complexity of search, and little work has been done yet to develop tools for the analysis of search procedures comparable to those available for the analysis of algorithms.

Since the development of Aquarius, we implemented a second prototype for Clause-Diffusion (Bonacina & McCune, 1994) and we oriented our research on parallel search. Partitioning effectively the search space of a theorem proving strategy is generally a difficult problem. For contraction-based strategies, it is especially difficult and largely unexplored. In subgoal-reduction strategies, such as Prolog technology theorem proving methods (e.g., (Astrachan & Loveland, 1991; Bose et al., 1992; Schumann & Letz, 1990; Stickel, 1988)), the search space is well-defined. It is formed by all the subgoals that can be derived from the original goal by using the axioms and the inference rules, e.g. the rules of model-elimination (Loveland, 1969). The standard representation as a tree of subgoals is satisfactory and most subgoal-reduction strategies use search plans that have in common the effective and well-understood idea of depth-first search with iterative deepening (Korf, 1985). In strategies that do not work by reduction of the goal, the definition and representation of the search space is not as well-defined. These strategies have a very high variety of inference rules and search plans. In addition, in contraction-based strategies, the deletion and replacement of clauses by contraction induces a degree of dynamicity in the search space itself. Based on this understanding, one of the directions in our current research aims at studying the definition and representation of search spaces and search plans, both sequential and parallel, in the presence of contraction. One of the goals of this type of research is to contribute toward a theoretical analysis of theorem proving strategies.

References

Anantharaman, S., Hsiang, J. (1990). Automated Proofs of the Moufang Identities in Alternative Rings. *J. of Automated Reasoning* **6**:1, 76–109.

Astrachan, O.L., Loveland, D.W. (1991). METEORs: High performance theorem provers using model elimination. Boyer, R.S. (ed.), *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Kluwer Academic Publisher.

Avenhaus, J., Denzinger, J. (1993). Distributing Equational Theorem Proving. Kirchner, C. (ed.), *Proc. of the Fifth Conf. on Rewriting Techniques and Applications*, Springer Verlag, Lecture Notes in Computer Science 690, 62–76.

Bachmair, L. Dershowitz, N., Plaisted, D.A. (1989). Completion without failure. Aït-Kaci, H., Nivat, M. (eds.), *Resolution of Equations in Algebraic Structures* **II**: Rewriting Techniques, 1–30, New York: Academic Press.

Bachmair, L., Ganzinger, H. (1990). On Restrictions of Ordered Paramodulation with Simplification. Stickel, M.E. (ed.), *Proc. of the Tenth Int. Conf. on Automated Deduction*, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 427–441.

Bonacina, M.P. (1991). Problems in Lukasiewicz logic. *Newsletter of the AAR* **18**, 5-12.

Bonacina, M.P. (1992). Distributed Automated Deduction. Ph.D. Thesis, Department of Computer Science, State University of New York at Stony Brook.

Bonacina, M.P., Hsiang, J. (1993). On fairness in distributed deduction. Enjalbert, P., Finkel, A., Wagner, K.W. (eds.), *Proc. of the Tenth Symp. on Theoretical Aspects of Computer Science*, Springer Verlag, Lecture Notes in Computer Science 665, 141–152.

Bonacina, M.P., Hsiang, J. (1994a). On subsumption in distributed derivations. *J. of Automated Reasoning* **12**, 225–240.

Bonacina, M.P., Hsiang, J. (1994b). Parallelization of deduction strategies: an analytical study. *J. of Automated Reasoning*, **13**, 1–33.

Bonacina, M.P., Hsiang, J. (1995). The Clause-Diffusion methodology for distributed deduction. Plaisted, D.A. (ed.), *Fundamenta Informaticae*, Special Issue on Term Rewriting Systems, in press.

Bonacina, M.P., McCune, W.W. (1994). Distributed theorem proving by *Peers*. Bundy, A. (ed.), *Proc. of the Twelfth Int. Conf. on Automated Deduction*, Springer Verlag, Lecture Notes in Computer Science 814, 841–845.

Bose, S., Clarke, E.M., Long, D.E., Michaylov, S. (1992). Parthenon: A parallel theorem prover for non-Horn clauses, *J. of Automated Reasoning* **8**:2, 153–182.

Chakrabarti, S., Yelick, K.A. (1993). Implementing an Irregular Application on a Distributed Memory Multiprocessor. *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

Chakrabarti, S., Yelick, K.A. (1993). On the Correctness of a Distributed Memory Gröbner Basis Algorithm. Kirchner, C. (ed.), *Proc. of the Fifth Conf. on Rewriting Techniques and Applications*, Springer Verlag, Lecture Notes in Computer Science 690, 77–91.

Chandy, K.M., Taylor, S. (1991). *An Introduction to Parallel Programming*. Jones and Bartlett.

Christian, J.D. (1989). High-Performance Permutative Completion. Ph.D. Thesis, University of Texas at Austin and MCC Tech. Rep. ACT-AI-303-89.

Conry, S.E., MacIntosh, D.J., Meyer, R.A. (1990). DARES: A Distributed Automated REasoning System. *Proc. of the Eleventh Conf. of the American Association for Artificial Intelligence*, 78–85.

Dershowitz, N., Jouannaud, J.-P. (1989). Rewrite Systems. *Handbook of Theoretical Computer Science* **15**:B, Amsterdam: Elsevier.

Foster, I., Tuecke, S. (1991). Parallel Programming with PCN. Tech. Rep. ANL-91/32, Argonne National Laboratory.

Hawley, D.J. (1991). A Buchberger Algorithm for Distributed Memory Multi-Processors. *Proc. of the Int. Conf. of the Austrian Center for Parallel Computation*, Springer Verlag, Lecture Notes in Computer Science 591, 385–390.

Hsiang, J. (1985). Refutational Theorem Proving Using Term Rewriting Systems. *Artificial Intelligence* **25**, 255–300.

Hsiang, J., Rusinowitch, M. (1987). On word problems in equational theories. Ottman, Th. (ed.), *Proc. of the Fourteenth Int. Conf. on Automata, Languages and Programming*, Springer Verlag, Lecture Notes in Computer Science 267, 54–71.

Jindal, A., Overbeek, R., Kabat, W. (1992). Exploitation of parallel processing for implementing high-performance deduction systems. *J. of Automated Reasoning* **8**, 23–38.

Kapur, D., Zhang, H. (1988). RRL: a Rewrite Rule Laboratory. Lusk, E., Overbeek, R. (eds.), *Proc. of the Nineth Int. Conf. on Automated Deduction*, Springer Verlag, Lecture Notes in Computer Science 310, 768–770.

Kirchner, C., Viry, P. (1992). Implementing Parallel Rewriting. Fronhöfer, B., Wrightson, G. (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 123–138.

Knuth, D.E., Bendix, P.B. (1970). Simple Word Problems in Universal Algebras. Leech, J. (ed.), *Proc. of the Conf. on Computational Problems in Abstract Algebras*, 263–298, Oxford: Pergamon Press.

Korf, R.E. (1985). Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence* **27**:1, 97–109.

Loveland, D.W. (1969). A simplified format for the model elimination procedure. *J. of the ACM* **16**:3, 349–363.

Lusk, E.L., McCune, W.W. (1992). Experiments with ROO: a Parallel Automated Deduction System. Fronhöfer B., Wrightson, G. (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence *590*, 139–162.

McCune, W.W. (1988). An indexing mechanism for finding more general formulas. *Newsletter of the AAR* **9**, 7–8.

McCune, W.W., Wos, L. (1988). Some Fixed Point Problems in Combinatory Logic. *Newsletter of the AAR* **10**, 7–8.

McCune, W.W. (1990). OTTER 2.0 Users Guide. Tech. Rep. ANL-90/9, Argonne National Laboratory.

McCune, W.W. (1991). What's New in OTTER 2.2. Tech. Mem. ANL/MCS-TM-153, Argonne National Laboratory.

Pelletier, F.J. (1986). Seventy-five problems for testing automatic theorem provers. *J. of Automated Reasoning* **2**, 191-216.

Rusinowitch, M. (1991). Theorem-proving with Resolution and Superposition. *J. of Symbolic Computation* **11**:1& 2, 21–50.

Schumann, J., Letz, R. (1990). PARTHEO: A High-Performance Parallel Theorem Prover. Stickel, M.E. (ed.), *Proc. of the Tenth Int. Conf. on Automated Deduction*, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 28–39.

Siegl, K. (1990). *Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic* Computation in Logic Programming Languages. Master Thesis and Tech. Rep. 90-54.0, RISC-LINZ.

Stickel, M.E. (1988). A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *J. of Automated Reasoning* **4**, 353-380.

Stickel, M.E. (1989). The Path-Indexing Method for Indexing Terms. Tech. Note 473, SRI Int..

Tuecke, S. (1992). Personal communications.

Vidal, J.-P. (1990). The Computation of Gröbner Bases on A Shared Memory Multiprocessor. Miola, A. (ed.), *Proc. of Int. Symp. on the Design and Implementation of Symbolic Computation Systems*, Springer Verlag, Lecture Notes in Computer Science 429, 81–90. Full version available as Tech. Rep. CMU-CS-90-163, School of Computer Science, Carnegie Mellon University.

Wasilewska, A. (1993). Personal communication.

Wos, L., Carson, D., Robinson, G. (1965). Efficiency and completeness of the set of support strategy in theorem proving. *J. of the ACM* **12**, 536–541.

Wos, L. (1988). *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall.

Yelick, K.A., Garland, S.J. (1992). A Parallel Completion Procedure for Term Rewriting Systems. Kapur, D. (ed.), *Proc. of the Eleventh Int. Conf. on Automated Deduction*, Springer Verlag, Lecture Notes in Artificial Intelligence 607, 109–123.