Distributed Deduction by Clause-Diffusion: the Aquarius Prover *

Maria Paola Bonacina and Jieh Hsiang

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
bonacina@loria.fr hsiang@sbcs.sunysb.edu

Abstract. Aquarius is a distributed theorem prover for first order logic with equality, developed for a network of workstations. Given in input a theorem proving problem and the 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 specific variant of a general methodology for distributed deduction, which we have called *deduction by Clause-Diffusion* and described in full in [6]. 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 an outline of the Clause-Diffusion methodology. Next, we consider in more detail the problem of *distributed global contraction*, e.g. normalization with respect to a distributed data base. The Clause-Diffusion methodology comprises a number of schemes for performing distributed global contraction, which avoid the *backward contraction bottleneck* of purely shared memory approaches to parallel deduction. Then, we describe Aquarius, its features and we analyze some of the experiments conducted so far. We conclude with some comparison and discussion.

1 Introduction

In this paper we describe the Clause-Diffusion methodology for distributed theorem proving, its implementation in the theorem prover Aquarius and we analyze the performances of Aquarius on some experiments.

A theorem proving problem consists in deciding, given a set of clauses S and a clause φ, whether φ is a theorem of S. A theorem proving strategy $\mathcal{C}$ is specified by a set of *inference rules* I and a *search plan* Σ. The inference rules

* Research supported in part by grant CCR-8901322, funded by the National Science Foundation. The first author is also supported by a fellowship of Università degli Studi di Milano, Italy. First author's current address: INRIA-Lorraine, 615 Rue du Jardin Botanique, B.P. 101, 54602 Villers-les-Nancy, France.

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 chooses the inference rule and the premises for each step, so that the repeated application of Σ and I generates a derivation. A derivation is successful if it reaches a solution of the input problem. If the strategy is *complete*, the derivation is guaranteed to succeed whenever the input goal is indeed a theorem. In practice, however, a derivation by a complete strategy may fail to prove a theorem, because it generates so many clauses that it exhausts the available memory before succeeding. In other words, it generates too large a portion of the *search space* of the problem. *Contraction-based strategies* try to reduce the incidence of such failures by applying eagerly powerful contraction rules to keep the data base, and thus the search space, as reduced as possible. These strategies, as implemented for instance in the provers Otter [20], RRL [16] and SBR3 [1], have obtained very encouraging results.

In this paper we present a methodology for parallelizing contraction-based deduction strategies. The main feature of contraction-based strategies is that existing data may be deleted or replaced by others through contraction. For instance, an equation may be reduced to another equation via rewriting. Although such a behaviour is the main reason why contraction-based strategies are effective, it is also the major source of difficulty in parallelization. To illustrate this point, we consider the parallelization of Prolog technology theorem proving (PTTP) methods. In goal-reduction methods such as PTTP, the set of axioms remains static during the course of the derivation. Thus, it is possible to pre-process all the axioms into elaborate data structures before the derivation starts. Such structures are used to exploit parallelism of different granularities. The cost of building them is limited to the pre-processing phase. Contraction-based strategies, on the other hand, are not likely to take advantage of such approaches, because axioms will be added and deleted during the derivation, so that pre-processing is not sufficient.

The basic idea of the Clause-Diffusion methodology, which we present here, is to parallelize 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*.

The Clause-Diffusion approach has a few features which we consider as new:

- It is a general methodology intended for implementing contraction-based strategies in distributed environments.
- The problem of keeping data inter-contracted is dealt with through a notion of *image set* – an approximation of the global data base. This avoids the difficulty of the *backward contraction bottleneck* which often occurs in shared memory implementation of contraction-based strategies [18, 27].

– It is a general methodology not confined to a specific architecture, topology or inference system. Depending on the parameters chosen, our method can be easily adopted in different environments.

Aquarius, a prototype built on top of the sequential theorem prover Otter, is completed. 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. In addition to all those of Otter, new parameters related to distributed execution are added, for a total of 121 options. This flexibility allows the user to tailor the prover to different classes of theorems, to use it to "simulate" to some extent other methods, e.g. the team-work method of [3, 12], and to apply it to other problems, such as Knuth-Bendix completion. Third, Aquarius is highly portable, since it has been written in C and PCN [13], under the Unix operating system, for a network of Sun workstations. In such an environment, each deductive process runs on a different node of the network. We have run Aquarius on many problems and we report a selection of results, including both positive and negative ones. 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.

The paper is organized as follows. First, we describe briefly the Clause-Diffusion methodology and we define the distributed derivations generated by a Clause-Diffusion strategy. While our methodology applies to theorem proving strategies in general, we designed it keeping contraction-based strategies in mind. Thus, we discuss some of the problems related to contraction in a distributed data base and the solutions adopted in Aquarius. The remaining sections are devoted to Aquarius and the experimental results. A full treatment of the issues in parallel theorem proving and a formal presentation of the Clause-Diffusion methodology are beyond the scope of this paper and therefore we refer to [6] for a complete description.

2 The Clause-Diffusion methodology

Given a complete theorem proving strategy $\mathcal{C} =< I; \Sigma >$, we describe how $\mathcal{C}$ 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*, are variable.

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 ambiguously 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 need 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. In this way each node p_i is allotted a subset S^i of the global data base. The union of all the S^i's, which are not necessarily disjoint, forms the current *global data base*. Each processor is responsible for applying the inference rules in I to its residents, according to the search plan Σ. Since the global data base is partitioned among the nodes, no node is guaranteed to find a proof using only its own residents. To assure that a solution will be found when one exists, the nodes need to exchange information, by sending each other their residents in form of messages, called *inference messages*. Each node uses the received inference messages to perform inferences with its own residents. The inference messages issued by a process p_i 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 [6]. While subdividing the expansion steps serves its purpose, it is not productive to subdivide the contraction steps, since the motivation behind contraction is to keep the data base always at the minimal. In a contraction-based strategy, an expansion step should be performed only if all the premises are fully reduced, at least with respect to the local data base. To ensure this, we require that each processor keep both its residents and received inference messages fully contracted.

We call *raw clause* a clause newly generated from an expansion step. Input clauses are also considered as raw clauses. In the presence of contraction rules, a raw clause should not become a resident until it has been fully contracted. Thus, our methods also feature a number of *distributed global contraction schemes* to reduce a raw clause with respect to the global data base. We shall describe these schemes in Section 2.1. After contraction, a raw clause becomes a *new settler*. New settlers are given to the allocation algorithm to be assigned to some 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: local contraction and local expansion inferences at the nodes among residents and inference messages, distributed global contraction, 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

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

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. A distributed derivation is the collection of the asynchronous derivations computed by the nodes. The *state* of the derivations 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 distributed derivation succeeds as soon as the derivation at one node finds a proof. A step in a distributed derivation can be either an *expansion* step or a *contraction* step 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 $(S^k \cup \{\psi\}, M^j) \vdash (S^k \cup \{\psi\}, M^j \cup \{\psi\})$. Settling a new settler at node p_k can be written as $(S^k, NS^k \cup \{\psi\}) \vdash (S^k \cup \{\psi\}, NS^k)$. This representation assumes that communication between any two adjacent nodes is instantaneous. It does *not* assume, however, that communication between *any* two nodes is instantaneous. If an inference message sent by p_i reaches p_j through $p_{x_1} \ldots p_{x_m}$, it appears first in M^{x_1}, then in M^{x_2} and so on. The time elapsed in going from the source to the destination is captured in our description, by showing the message stored, at successive stages, in the appropriate component of all the nodes on the path.

2.1 Distributed global contraction

In distributed theorem proving, we term *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 [6], we have proposed several schemes for *distributed global contraction*. The distributed approach which is common to these schemes provides a solution to an important implementation problem of parallel theorem proving in shared memory, which we term the *backward contraction bottleneck*. In this section, we describe first this problem and then the schemes for distributed global contraction implemented in Aquarius. More details can be found in [6].

What is the backward contraction bottleneck Operationally, contraction steps can be separated into *forward contraction* and *backward contraction*. Informally speaking, forward contraction uses existing data to contract new data (raw clauses and incoming inference messages, for instance), while backward contraction uses new data to contract existing ones (the residents). Designing an effective and efficient method for parallel backward contraction is a much more complicated task than for parallel forward contraction. Indeed, backward contraction has turned out to be a critical problem for shared memory implementations [18, 27] of parallel theorem proving with contraction, while some other implementations simply do not implement backward contraction (e.g. DARES [11] and PARROT [15]). In a related area, parallel implementations of the *Buchberger algorithm* [14, 22, 25] have also suffered from this problem.

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. [17].

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 [18, 27], 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 residents reside. 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.

Distributed global contraction schemes In [6], we have given 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 $\bigcup_{i=1}^{p} S^i$ 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 born (its "source"). By an approximation of the global data base, we mean a set of copies of the residents in the systems, which may

be used as simplifiers. We call such a set an *image set*. An image set is an approximation, because it is not guaranteed to be identical to the global data base $\bigcup_{i=1}^{p} S^i$ at any stage of the execution.

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 a *localized image set* SH^i of $\bigcup_{i=1}^{p} S^i$. Each node uses its localized image set as set of simplifiers to perform global contraction of residents, raw clauses and incoming messages. The localized image sets can be built by utilizing the inference messages: *whenever a node p_i receives an inference message, it stores the clause carried by the message in SH^i*. 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 just an approximation of the global data base. However, each of the SH^i's is logically equivalent to the global data base $\bigcup_{i=1}^{p} S^i$, if all the persistent residents, i.e. those not deleted by contraction, are broadcast as inference messages. In *global contraction at the source by global image set in shared memory*, a single, global image set is held in shared memory. The choice of the appropriate global contraction scheme is related to 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. In this paper we consider global contraction at the source by localized image sets, because it is the scheme implemented in Aquarius, while we refer to [6] for the others.

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* [10, 23] for the purpose of fast simplification.

Maintenance of the image sets A fundamental issue in global contraction at the source is whether contraction of the simplifiers in the SH^i's should be allowed. The question is whether the advantage of maintaining the SH^i's fully reduced is worth the cost of updating them. In [6], we have proposed several different approaches. One possibility is to have each p_i performing contraction on SH^i just like on S^i ("maintenance by direct contraction"). Each node contracts its own raw clauses, but all nodes execute independently contraction of all residents: if ψ is a resident at p_i which is also stored in the SH^j components at the other nodes, contraction of ψ is performed at all nodes which have a copy of ψ. If contraction inferences are sufficiently fast, this may be a reasonable choice.

At the other extreme, one may forbid contraction on the SH^i's and allow only insertion of new elements ("no contraction" policy). 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. If SH^j is used only as a data base of simplifiers, the presence of both ψ and a reduced form ψ' should 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. However, if no element is ever deleted from the SH^j's, their sizes may grow up to compromise their performances. Furthermore, if the elements in SH^j are used for expansion steps, redundant clauses in SH^j would induce the generation of more redundant clauses. In order to avoid such phenomena, we may design mechanisms to update the SH^i's without resorting to the direct application of contraction inferences.

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 at p_i and $< p_i, a >$ is the unique *global identifier* of ψ. 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. For instance, an image set may be implemented as a *hash table*, with the global identifier as key. We require that inference messages carry 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 $< \psi, a, x >$ is reduced to $< \psi', a, y >$, where $y > x$, at p_i, 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 ordering on clauses and saves the smallest in SH^j. If the two clauses are not comparable, the one with most recent birth-time is saved. We call this solution "update by inference messages".

In case of update by inference messages, the situation is less favorable if $< \psi, a, x > \in S^i$ is deleted, rather than replaced, by a contraction step. In such case, 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: for instance, the strategies implemented in Aquarius apply first update by inference messages and then direct contraction. Thus, deletion messages are not necessary. Also, fewer direct contraction steps will be performed if direct contraction is preceded by update by inference messages.

2.2 Clause-Diffusion strategies

In the above 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 [6, 7], 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 [6, 7], we gave a set of formal properties that the algorithms and policies handling messages in a distributed strategy need to satisfy. We proved that these properties imply fairness and we showed that the specific policies of the Clause-Diffusion method satisfy those properties.

In the following we describe a specific class of Clause-Diffusion strategies that we have implemented.

3 The Aquarius theorem prover

Aquarius implements a version of the Clause-Diffusion methodology with global contraction at the source by localized image sets. Each of the concurrent deduction processes executes a modified version, called *Penguin* [6], of the code of the theorem prover *Otter* (version 2.2) [20]. Otter is a resolution-based theorem

prover for first logic with equality. Its basic mechanism works as follows: select a *given_clause* from the *Sos* (Set of Support), execute all the expansion inferences with the *given_clause*, pre-process (forward contraction) and post-process (backward contraction) all the generated raw clauses and then iterate. Different strategies may be obtained by setting several options. Penguin is structured into a *communication layer* and a *deduction layer*. The communication layer, written in PCN [13], implements the message-passing part. The deduction layer, written in C, incorporates the code of Otter, so that each Penguin process executes the basic cycle on the *given_clause*. In addition, the deduction layer implements the features required by the Clause-Diffusion methodology, e.g. the partitioning of the expansion steps based on the ownership of clauses, the distributed allocation of clauses and so on.

The Aquarius program is invoked with two main parameters, the name of the input problem and the number of requested Penguin processes. Each Penguin process reads its own input file and creates its own output and error files. The format of the files is the same as in Otter. The user may set a very high number of options: Aquarius has 121 options, 99 flags (boolean-valued options) and 22 parameters (integer-valued options), including all those of Otter. These options determine the components of the executed strategy: the inference mechanism, e.g. by selecting which inference rules are active, the search plan at each node, e.g. by sorting the lists of clauses and messages, the allocation of clauses as residents and the interleaving of inference steps and communication steps at each node. Since each Penguin process reads its own input file, the user may set different options patterns, and thus different strategies, at different Penguins. This flexibility allows the user to set 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 [3, 12], albeit without the "referee processes" and the periodical reconstruction of a common data base that are characterizing parts of the team-work method. The KNUTH-BENDIX option (inherited from Otter), allows to perform Knuth-Bendix completion, so that Aquarius executes Knuth-Bendix completion in parallel. The STAND-ALONE option induces each Penguin to work 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 Penguin a different input and have the nodes working in parallel on different problems. For instance, one may want to give to each Penguin a different lemma from a large problem and have the lemmas proved independently. While it provides the user with many input options, Aquarius is not interactive: like for Otter, the emphasis is on obtaining fully automated proofs, with no human intervention during the run. In the following, we analyze some experiments with Aquarius, while we refer to [6] for a complete description of the prover.

3.1 Experiments with Aquarius

In the following table, we give the performances of Aquarius on some problems. Aquarius-n is Aquarius with n nodes, where each node is a Sun Sparcstation.

So far we have been able to experiment with up to 3 of them. They communicate over the departmental Ethernet at Stony Brook. The sparcstations 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 (in seconds) represent the performances under realistic working conditions. 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 Penguin to succeed, which includes both inference time and communication time. However, it includes neither the initialization time spent to set up the Penguin 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 Penguins 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.

Problem	*Aquarius-1*	*Aquarius-2*	*Aquarius-3*
andrews [8, 21]	18.00	25.40	24.39
apabhp [18]	11.86	18.11	14.18
bledsoe [4, 18]	12.29	21.53	23.00
cd12 [18]	104.18	50.98	47.56
cd13 [18]	98.79	45.32	51.07
cd90 [18]	3.10	0.63	11.87
cn [18]	5.04	8.63	14.50
ec	3.03	1.96	1.77
imp1 [18]	6.63	2.64	3.54
imp2 [18]	7.25	3.31	7.43
imp3 [18]	32.05	17.92	38.89
luka5 [5]	844.20	299.24	1079.45
pigeon (ph4) [21]	8.21	7.66	8.14
robbins2 [18]	21.62	22.91	24.12
s7 [2]	630.62	208.37	192.54
salt	3.89	4.45	5.49
sam's lemma	6.35	5.40	3.90
subgroup [26]	15.55	9.36	17.40
w-sk [19]	3.50	3.52	3.34

3.2 Analysis of the experiments

The significance of these experiments is limited by having only up to 3 nodes. Also, the problems which can be solved sequentially in a few seconds are probably too easy for the parallelization to pay off. Furthermore, this problems set may not be the most ideal to test Clause-Diffusion. Most of the above problems are taken from the input sets for Otter and ROO 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 is most dramatic. Aquarius-1 is generally slower than Otter, which indicates that the overhead induced merely by having linked the PCN part with the C part is not irrelevant. As can be seen, the experimental results are quite unstable. There may be many factors for such mixed results. One reason is of course the prototype nature of Aquarius, which was developed in a short 5 months period. In the following, we try to analyze the performances of Aquarius in terms of *communication*, *duplication* and *distribution of clauses*.

Observations of communication problems in Aquarius Communication in Aquarius is very slow. An immediate evidence of this is the following. In the current version, only one Penguin, i.e. Penguin0, finds the input clauses in its input file and broadcasts them to the other Penguins. In many cases, Penguin0 is the first one to succeed. Also, it happens that Penguin1 and Penguin2 have shorter run time than Penguin0, since the start of the derivations by Penguin1 and Penguin2 is delayed by the necessity of waiting for the input clauses. This observation suggests that a simple improvement would be to have each Penguin reading the input clauses from its input file. 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. Penguin0 and Penguin1. In most runs, it happens that Penguin0 generates and broadcasts ψ, but Penguin1 derives it on its own, *before* receiving the inference message from Penguin0. 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 happens in Aquarius the purpose of the inference messages is sort of defeated.

Communication among Penguin processes is handled by PCN. PCN [9] is a logic-programming-like language built on top of a sequential language such as C to serve as the communication layer. The performance of Aquarius is affected by PCN in at least two ways:

1. The current implementation of PCN 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 Penguins, 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 processes, i.e., the deduction layer of each Penguin. However, this does not seem to have been sufficient. An alternative approach is to synchronize the communication and deduction layers within each Penguin. 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.

Duplication After having experienced the problem with communication, we resorted to try to reduce the amount of communication by empowering the single nodes. Because communication is so slow, it is better that all nodes are able to work as independently as possible. Some of the reported experiments have been done by setting the flags in such a way that each node owns most of the input clauses. In other experiments, the flags for the allocation of clauses have been set in such a way that each node retains most of its raw clauses as residents. None of the reported results, however, refer to executions under a combination of flags equivalent to the $STAND\_ALONE$ mode. In other words, in all the listed experiments, there is some partitioning of the search space.

While reducing communication, these settings of flags, together with the use of localized image sets, induce a strong increase in duplication. It appears from the trace files of the experiments, that often most of the clauses needed in the proof are 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 Penguin2 as follows: first, it is generated and sent as new settler to Penguin0; second, it is generated again and kept as resident; third, it is received as inference message from Penguin0; fourth, it is generated one more time and sent as new settler to Penguin1; 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 may explain the lack of speed-up in many experiments. The Clause-Diffusion methodology and Aquarius are sufficiently flexible to provide combinations of different degrees of communication and duplication. However, the current version of Aquarius realizes a highly duplication-oriented version of the Clause-Diffusion methodology, 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 Penguin 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 superlinear 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 $P(e(e(x,y),e(x,y)))$, but does not even generate $P(e(x,x))$.

Distribution of clauses The third issue, i.e. the distribution of clauses, is more of a conceptual nature. 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 also be allocated to node 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.

4 Discussion

In the first part of the paper, we outlined our methodology for distributed deduction by Clause-Diffusion. This approach realizes a sort of coarse-grain parallelism, that we have termed *parallelism at the search level* [6]. Our methodology does not exclude the application of techniques for fine-grain parallelism, such as those employed for parallel rewriting languages, e.g. [17]. While the Clause-Diffusion methodology applies to theorem proving strategies in general, we have devoted special attention to contraction-based strategies. 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. We indicated in the *bottleneck of backward contraction* a critical problem in the shared memory approaches to parallel automated deduction. This source of inefficiency had not been identified before. In [6], we proposed as solutions several schemes for distributed global contraction. Here we have focused on *global contraction at the source by localized image sets*, since it is the scheme implemented in Aquarius.

In the second part of the paper, we described Aquarius and analyzed some experiments. Other parallel theorem provers have obtained better experimental results than Aquarius. For instance, ROO [18] shows linear speed-up on most non-equational problems, while its performances on equational problems suffer 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 very different, because by partitioning the search space, we

aim at parallelism at the search level. 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 Penguin does not retain a certain clause and sends it to settle at another node to change dramatically the search space for that Penguin. 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 be trying to generate the sequential proof from a fragmented search space. The irregular results are the consequence of this kind of phenomena.

In summary, at the operational level, the main cause for the mixed results of Aquarius is the inefficiency of communication. At least part of the problem seems to be 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 2.1 and [6]) 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 probes a radically new approach to parallelization, whose success will require a better understanding of the parallelization of search.

References

1. S.Anantharaman, J.Hsiang, Automated Proofs of the Moufang Identities in Alternative Rings, *JAR*, Vol. 6, No. 1, 76–109, 1990.
2. A.Wasilewska, Personal communication, March 1993.
3. J.Avenhaus and J.Denzinger, Distributing Equational Theorem Proving, in C.Kirchner (ed.), *Proc. of the 5th RTA Conf.*, Springer Verlag, LNCS, to appear.
4. W.Bledsoe, Challenge problems in elementary calculus, *JAR*, Vol. 6, No. 3, 341-359, 1990.
5. M.P.Bonacina, Problems in Lukasiewicz logic, *Newsletter of the AAR*, No. 18, 5-12, Jun. 1991.
6. M.P.Bonacina, Distributed Automated Deduction, Ph.D. Thesis, Dept. of Computer Science, SUNY at Stony Brook, Dec. 1992.
7. M.P.Bonacina and J.Hsiang, On fairness in distributed deduction, in P.Enjalbert, A.Finkel and K.W.Wagner (eds.), *Proc. of the 10th STACS*, Springer Verlag, LNCS 665, 141–152, 1993.

8. D.Champeaux, Sub-problem finder and instance checker: Two cooperating preprocessors for theorem provers, in *Proc. of the 6th IJCAI*, 191–196, 1979.
9. K.M.Chandy, S.Taylor, An Introduction to Parallel Programming, Jones and Bartlett, 1991.
10. J.D.Christian, High-Performance Permutative Completion, Ph.D. Thesis, Univ. of Texas at Austin and MCC Tech. Rep. ACT-AI-303-89, Aug. 1989.
11. S.E.Conry, D.J.MacIntosh and R.A.Meyer, DARES: A Distributed Automated REasoning System, in *Proc. of the 11th AAAI Conf.*, 78–85, 1990.
12. J.Denzinger, Distributed knowledge-based deduction using the team-work method, Tech. Rep. SR-91-12, Univ. of Kaiserslautern, 1991.
13. I.Foster, S.Tuecke, Parallel Programming with PCN, Tech. Rep. ANL-91/32, Argonne Nat. Lab., Dec. 1991.
14. D.J.Hawley, A Buchberger Algorithm for Distributed Memory Multi-Processors, in *Proc. of the International Conference of the Austrian Center for Parallel Computation*, Oct. 1991, Springer Verlag, LNCS, to appear.
15. A.Jindal, R.Overbeek and W.Kabat, Exploitation of parallel processing for implementing high-performance deduction systems, *JAR*, Vol. 8, 23–38, 1992.
16. D.Kapur. H.Zhang, RRL: a Rewrite Rule Laboratory, in E.Lusk, R.Overbeek (eds.), *Proc. of CADE-9*, LNCS 310, 768–770, 1988.
17. C.Kirchner, P.Viry, Implementing Parallel Rewriting, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, LNAI 590, 123–138, 1992.
18. E.L.Lusk, W.W.McCune, Experiments with ROO: a Parallel Automated Deduction System, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, LNAI 590, 139–162, 1992.
19. W.W.McCune, L.Wos, Some Fixed Point Problems in Combinatory Logic, *Newsletter of the AAR*, No. 10, 7–8, Apr. 1988.
20. W.W.McCune, OTTER 2.0 Users Guide, Tech. Rep. ANL-90/9, Argonne Nat. Lab., Mar. 1990.
21. F.J.Pelletier, Seventy-five problems for testing automatic theorem provers, *JAR*, Vol. 2, 191-216, 1986.
22. K.Siegl, Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages, M.S. Thesis and Tech. Rep. 90-54.0, RISC-LINZ, Nov. 1990.
23. M.E.Stickel, The Path-Indexing Method for Indexing Terms, Tech. Note 473, SRI Int., Oct. 1989.
24. S.Tuecke, Personal communications, May 1992 and Dec. 1992.
25. J.-P.Vidal, The Computation of Gröbner Bases on A Shared Memory Multiprocessor, in A.Miola (ed.), *Proc. of DISCO90*, Springer Verlag, LNCS 429, 81–90, Apr. 90 and Tech. Rep. CMU-CS-90-163, Carnegie Mellon Univ., Aug. 1990.
26. L.Wos, *Automated Reasoning: 33 Basic Research Problems*, Prentice Hall, 1988.
27. K.A.Yelick and S.J.Garland, A Parallel Completion Procedure for Term Rewriting Systems, in D.Kapur (ed.), *Proc. of the 11th CADE*, Springer Verlag, LNAI 607, 109–123, 1992.