The Clause-Diffusion methodology for distributed deduction

Maria Paola Bonacina [*]
Department of Computer Science
University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu

Jieh Hsiang [†]
Department of Computer Science
National Taiwan University
Taipei, Taiwan
hsiang@csie.ntu.edu.tw

Abstract

This paper describes a methodology for parallel theorem proving in a distributed environment, called *deduction by Clause-Diffusion.* This methodology utilizes *parallelism at the search level*, by having *concurrent, asynchronous* deductive processes searching in parallel the search space of the problem. The search space is partitioned among the processes by distributing the clauses and by subdividing certain classes of inferences. The processes communicate by exchanging data. Policies for distributing the clauses and for scheduling inference and communication steps complete the picture. A *distributed derivation* is made of the collection of the derivations computed by the concurrent deductive processes and it halts successfully as soon as one of them does.

While the Clause-Diffusion methodology applies to theorem proving in general, it has been designed to provide solutions to the problems in the parallelization of *contraction-based* strategies, such as rewriting-based methods. We identify *backward contraction*, i.e. the task of maintaining clauses reduced in a dynamically changing data base, as the main obstacle in parallel theorem proving with contraction. In parallel implementations of contraction-based strategies in shared memory, this difficulty appears as a write-bottleneck, which we have termed the *backward contraction bottleneck.* The Clause-Diffusion approach avoids this problem by adopting a mostly distributed memory and *distributed global contraction schemes.*

We conclude by reporting some of our results with an implementation of Clause-Diffusion.

1 Introduction

The subject of this paper is a methodology for distributed theorem proving, called *distributed deduction by Clause-Diffusion* and appeared first in [7].

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 *expansion inference rules*, such as resolution and paramodulation, generate new clauses from existing ones and add them to the data base. The *contraction inference rules* delete clauses, e.g. by subsumption,

[*]Supported in part by the GE Foundation Faculty Fellowship to the University of Iowa.

[†]Supported in part by grant NSC 83-0408-E-002-012T of the National Science Council of the Republic of China.

or replace them by smaller ones, e.g. by simplification (term rewriting). 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, a complete strategy often fails 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. One way to tackle this problem is to employ contraction inference rules, such as rewriting and subsumption, which keep the data base, and thus the search space, as reduced as possible. Contraction inferences can be classified into *forward contraction* and *backward contraction*. Forward contraction contracts newly generated clauses, before they are fully introduced in the data base and made available for expansion steps. Backward contraction contracts existing clauses, that have been already stored in the data base, used in expansion steps and possibly as simplifiers. The application of backward contraction, and thus the possibility of contracting any clause, is a distinguishing feature of *contraction-based strategies*. These strategies, as implemented for instance in the provers Otter [25], RRL [22] and SBR3 [1], have obtained impressive 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, e.g. [10, 15, 27]. 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 can be used in turn 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, asynchronous deductive processes, which search in parallel for a solution. As soon as one of them succeeds, the whole distributed derivation succeeds. In order to subdivide the search task, we notice that the search space is determined by the given clauses and the inference rules. The Clause-Diffusion method partitions the clauses among the processes and the inferences are partitioned consequently in a *data-driven* fashion. The deductive processes are largely independent: each process has its own local data base, constructs its own derivation and interacts with the others by exchanging clauses. Because of the coarse granularity of parallelism at the search level, the Clause-Diffusion methodology targets a distributed environment, e.g. a high-speed network of computers or a loosely coupled asynchronous multiprocessor. In such a system, each of the theorem proving processes is executed at a different processor node, and communication of clauses is implemented via *message-passing.* Clause-Diffusion may also use a mixed shared-distributed environment, e.g. with separate memories at the nodes and a shared memory component: a subset of the clauses may be shared, while the remaining communication is achieved through messages.

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.
- It realizes *parallelism at the search level* as a form of coarse grain parallelism in deduction. This is a fairly new approach, since most of the existing methods introduce parallelism in the inference mechanism, thus at a finer granularity.
- 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, e.g. [24, 33].
- 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.

The remaining sections of this paper are organized as follows. First, we illustrate how a given strategy $\mathcal{C}$ is executed according to the Clause-Diffusion methodology in a distributed system and we define the distributed derivations generated by Clause-Diffusion strategies. Second, we address the parallelization of contraction in Clause-Diffusion. We describe the *backward contraction bottleneck*, which may arise especially in parallel implementations of term-rewriting based methods in shared memory. Our solution of the backward contraction bottleneck problem are a few *schemes for distributed global contraction*, that is, contraction with respect to the distributed global data base (the union of the clauses of all the concurrent processes). Since the backward contraction bottleneck is observed primarily in equational problems, we discuss in detail how to perform "simplification of the simplifiers", i.e. how to update with respect to simplification the copies, or "images", of the equations used as simplifiers. The following section is devoted to the policies for *distributed allocation of clauses*, i.e. the criteria to assign clauses to processes. These policies serve two very important purposes: partitioning the search space of the theorem proving problem among the deductive processes and balancing their work-load, so that it does not happen that some are idle while others are overcharged. We conclude the detailed description of Clause-Diffusion with some guidelines for the scheduling of inferences and communication steps performed by the concurrent deductive processes. The last technical section is devoted to some experiments with the Clause-Diffusion prover *Peers*[1].

[1]Peers was written jointly by the first author and Dr. Bill McCune of the Argonne National Laboratory.

Related papers are [4, 5, 8, 9]: [4] and [5] are dedicated to specific problems, distributed fairness and distributed subsumption respectively; [8] and [9] describe the specific strategies and the implementation details of the Clause-Diffusion prototypes *Aquarius* and *Peers*. This paper is a comprehensive description of the Clause-Diffusion methodology after [7], which we refer to for the few components or variants of Clause-Diffusion strategies that are not covered here.

2 The Clause-Diffusion methodology

Contraction-based strategies are theorem proving methods developed from the studies in term rewriting theory. Intuitively, a contraction-based strategy assumes the existence of a well-founded ordering on either the set of formulas or on the proof structure (depending on the logic and the inference system). Based on this notion of ordering, contraction-based strategies are characterized by three basic properties. First, the inference system includes contraction rules such as simplification via term rewriting, conditional or constrained simplification, clausal and functional subsumption, which replace *redundant* data by smaller ones. Second, the search plan generally gives priority to contraction rules over expansion rules, so that the data set can be kept to a minimal. Third, the expansion rules may be refined to resolve upon only the maximal elements in the data involved. Knuth-Bendix completion and its extensions are typical examples of contraction-based strategies. We refer to [18] for an introductory survey. Examples of contraction-based strategies for first order logic and beyond can be found in [3, 26]. A formal treatise of contraction-based strategies is given in [6], where also a larger set of references to works on contraction-based strategies is provided.

The Clause-Diffusion methodology was designed with contraction-based strategies in mind, although it applies to other types of strategies as well. We assume then to have a complete theorem proving strategy $\mathcal{C} =< I; \Sigma >$ and we describe how $\mathcal{C}$ is parallelized by the Clause-Diffusion methodology. The strategy will be executed by multiple concurrent deductive processes $p_1, \ldots p_n$. We envision a distributed environment, such as a network or a loosely coupled, asynchronous multiprocessor, where each deductive process runs on a node of the system. Since there is a one-one correspondence between the deductive processes and the nodes, we shall use $p_1 \ldots p_n$ to denote ambiguously both the deductive processes and the nodes. The basic principle of the Clause-Diffusion methodology is *to partition the search space* among the concurrent deductive processes $p_1, \ldots p_n$. In order to partition the search space, the input and the generated clauses are distributed among the processes. After a clause ψ is fully simplified (contracted), it is assigned to a process p_i and becomes a **resident** of p_i. For this purpose, a Clause-Diffusion strategy features an **allocation algorithm**, which decides where to allocate a clause. In this way, each process 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.* Process p_i is responsible for applying the inference rules in I to the clauses in S^i, according to the search plan Σ.

Because 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 process sends the inference messages for its own residents and uses the received inference messages to perform inferences with its 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. The local data base at each node contains at any stage of the derivation both resident clauses and visiting clauses, which came in as inference messages. Thus, one should keep in mind that the physical subdivision of the data set, i.e. whether a clause is stored at a node, is different from the logical subdivision, i.e. whether a clause belongs to a process.

The communication of inference messages may be realized in different ways. In a purely distributed system, inference messages are implemented as messages, which may be routed or broadcast. If it is desirable to have all the nodes receiving the inference messages as soon as possible, then the inference messages should be broadcast. Depending on the broadcasting algorithm, a node may forward copies of the inference message to different nodes, while still retaining a copy for its own inferences. Thus, there may be several inference messages, all carrying the same clause, active at different nodes. In a distributed system with a shared memory component, the exchange of clauses may be implemented also through the shared memory. A process p_i "sends" its resident ψ by storing a copy of ψ in the shared memory. All the other processes "receive" ψ by reading it from the shared memory.

The separation of residents and inference messages is also used to partition the search space at the inference level. Using paramodulation as an example of expansion inference rule, we can require 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 fit naturally in this pattern (see Section 5.3).

Unlike expansion steps, contraction steps are not subdivided based on ownership of clauses. Since the motivation behind contraction is to keep the data base always at the minimal, we allow each node to use all available simplifiers, e.g. both residents and received inference messages, to perform as much contraction as possible. The contraction task comprises both *forward and backward contraction.* Forward contraction is contraction of the clauses newly generated from expansion steps. We call such clauses **raw clauses**. After being fully contracted, a raw clause becomes a **new settler** and is entitled to become a resident at a processor. The allocation algorithm is used to assign a new settler 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.

Backward contraction is contraction of the clauses already established in the data base and being used as premises of expansion steps and as simplifiers. Backward contraction is essential in contraction-based strategies, because such strategies should only perform an expansion step when the premises are fully reduced. In Clause-Diffusion, backward contraction is contraction of the residents: each deductive process keeps its residents as reduced as possible. In addition, a process may reduce the inference messages it receives, before using them as simplifiers and as premises for expansion. Contraction of received inference messages is also backward contraction, because an inference message received by one process is a resident at some other node. Both forward and backward contraction need to be done with respect to the global data base. Thus, our methodology features a number of **distributed global contraction schemes** to enable a node to perform contraction with respect to a distributed set of clauses.

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.

The above elements are summarized in the following notion of *distributed derivation*: every deductive process p_k, $1 \leq k \leq n$, computes a derivation

$$(S;M;CP;NS)^k_0 \underset{\mathcal{C}}{\vdash} (S;M;CP;NS)^k_1 \underset{\mathcal{C}}{\vdash} \ldots (S;M;CP;NS)^k_i \underset{\mathcal{C}}{\vdash} \ldots$$

where

- S^k_i is the set of *residents* at p_k at stage i,
- M^k_i is the set of *inference messages* at p_k at stage i,
- CP^k_i is the set of *raw clauses* at p_k at stage i and
- NS^k_i is the set of *new settlers* at p_k at stage i.

A distributed derivation is given by the family of the asynchronous derivations computed by the deductive process. The tuple $(S;M;CP;NS)^k_i$ represents the *state* of the derivations at processor p_k and stage i. Depending on the specific strategy, more components may be needed. The distributed derivation succeeds as soon as a derivation at one node finds a proof. Each 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 in the path.

3 Distributed global contraction

This section treats the parallelization of contraction in Clause-Diffusion. The distinction between *forward contraction* and *backward contraction* is critical in understanding the difficulty of parallelizing strategies with contraction:

1. Forward contraction is the normalization of raw clauses before they are inserted in the data base: if a raw clauses has a non-trivial normal form, it is added to the data base, otherwise it is deleted. Thus, a forward contraction step does not induce other contraction steps. On the other hand, backward contraction is the inter-reduction of the clauses in the data base: if a clause already in the data base is simplified by a newly added clause, its reduced form should be tested for further contraction and normalized with respect to *all* the other clauses in the data base. If the new normal form is not trivial, it should be applied in turn to try to reduce every other clause in the data base. Therefore, a backward-contraction step may trigger many backward-contraction steps, each of which in turn may induce more.

2. In forward contraction there is a clear distinction between the clauses to be normalized – the raw clauses – and the clauses to be used as simplifiers – the clauses already in the data base. On the contrary, in backward contraction, there is no such distinction, because all the clauses in the data base are at the same time simplifiers and subject to being contracted.

3. Concurrent inference steps may incur *conflicts* in trying to access common premises. Concurrent forward contraction steps may cause *write-write conflicts* if they try to reduce concurrently the same term. This problem has been studied already in parallel term rewriting (see [7] for a survey of various approaches and references). In addition to write-write conflicts, concurrent backward contraction may generate *read-write conflicts* between contraction steps (one step tries to read a clause to use it as simplifier, while another step tries to rewrite it) and between expansion and contraction steps (the expansion step tries to read a clause to use it as a parent, while the contraction step tries to rewrite it). Forward contraction does not induce these conflicts, because the clauses being rewritten, the raw clauses, are not yet being used as simplifiers or as premises for expansion. The read-write conflict between expansion and backward contraction steps is the worst type of conflict from the point of view of the design of parallel contraction-first strategies: parallelism would require to let expansion and contraction proceed in parallel, whereas priority to contraction would require to sequentialize contraction and expansion in such a way that expansion is attempted only if contraction has completed.

These elements show that the efficient parallelization of backward contraction is much harder than forward contraction. Indeed, some parallel theorem provers, e.g. [16, 21], did not implement backward contraction; others, e.g. [24, 33], found in backward contraction a major obstacle for implementations of parallel theorem proving in shared memory. In a related area, parallel implementations of the *Buchberger algorithm* [31, 28, 20] also suffered from this problem (see [7] for survey and comparison of these methods).

The above characteristics of backward contraction cause the following dilemma. Assume that concurrent backward contraction and concurrent expansion are to be implemented in shared memory. If unrestricted concurrent inferences are allowed, conflicts arise. Thus, it is necessary to use special control structures, e.g. locks and critical regions, to control write-access and avoid the conflicts. One such approach is described in [33]. This type of solution involves a high amount of fine-grain control on instructions and data structures and is resemblant of the techniques for parallel term rewriting of one given term. In theorem proving, millions of terms need to be rewritten and the "rippling" effect of backward contraction (each step inducing many) may cause an avalanche growth of contraction steps. For instance, this happens if a simplifier that reduces most of the clauses in the data base is generated. In these conditions, backward contraction causes a *write-bottleneck*, because all the backward contraction processes ask write-access to the shared memory. Since write-access is controlled in order to prevent conflicts, a sequentialization is imposed. This is an instance of *backward contraction bottleneck*.

In order to avoid this scenario, one may forbid concurrent backward contraction: multiple parallel processes are allowed to perform expansion in parallel accessing the main data base of clauses in shared memory, whereas only one process does backward contraction working on a separate data structure accessed only by this process. Whenever a clause in the data base needs to be tested for normalization, it is moved from the data base to the separate data structure. This is the approach of [24]. The problem of this solution is that backward contraction causes the number of clauses in need of backward contraction to surge, so that the single backward contraction process is swamped, the main data base is depleted and the expansion processes are forced to be idle. In other words, the single backward contraction process becomes a bottleneck, another instance of *backward contraction bottleneck*.

We use **backward contraction bottleneck** to indicate this problem, including the two scenarios described above as instances. In order to avoid this bottleneck we chose to work with coarse grain parallelism in distributed memory. The clauses are distributed among the nodes, each process works in its own local memory and therefore there are no conflicts. In this context, the issue becomes how to perform *global contraction*, that is, how to enable each process to normalize a clause not only with respect to its own local data base, but with respect to the global distributed data base, without incurring excessive costs of duplication and communication. In the rest of this section, we describe several schemes for achieving global contraction in distributed memory.

3.1 Distributed global contraction schemes

Global contraction by travelling and **global contraction at the source** are the two types of mechanisms for distributed global contraction in Clause-Diffusion. In global contraction by travelling, no node has access to the global data base: global contraction uses messages. In global contraction at the source, every node has access to an "approximation" of the global data base and uses it to contract its clauses. The selection of the appropriate global contraction scheme depends on the available resources: global contraction at the source requires either sufficiently large local memories or a shared memory component to provide access to the global data base, whereas global contraction by travelling needs very fast communication. In this paper we focuse on global contraction at the source, referring to [7] for global contraction by travelling.

In **global contraction at the source**, each node p_i has access to an approximated version of the global data base, called an *image set.* The name "image set" says that such set contains "images", i.e. copies, of residents in the system. We describe **global contraction at the source** in two scenarios, depending on the availability of a shared memory.

- **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's are called *localized image sets.* Each node p_i uses its localized image set SH^i 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.* 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. The sets SH^i can also be used only to contain those intended to be used as simplifiers.

- **Global contraction at the source by global image set in shared memory**: we assume that a shared memory is available and a single approximated version SH of the global data base $\bigcup_{i=1}^{n} S^i$ is stored in the shared memory. The set SH is called *global image set.* Each process accesses SH to get the simplifiers to perform, at each node, global contraction of residents, raw clauses and incoming messages. The global image set SH can be formed as follows: whenever a new settler ψ settles down at a node p_i as a resident, node p_i also takes care of adding ψ to SH in shared memory. Similar to the previous case, the identity $SH = \bigcup_{i=1}^{n} S^i$ is not strictly true, so that the global image set is just an approximation of the global data base. However, SH and $\bigcup_{i=1}^{n} S^i$ are logically equivalent, if an image of every persistent resident is included in SH.

The backward contraction bottleneck does not appear in our schemes, because the clauses being rewritten by contraction are stored 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. Even when our schemes employ a shared memory, the latter is only used to store clauses used as simplifiers, while the clauses subject to contraction are kept in the individual nodes. Therefore, concurrent contractions can always be done at the nodes without creating any global bottleneck, as long as the architecture supports concurrent reads on the shared memory. An additional advantage of using image sets, either in shared memory or with a copy per node, is that such large sets of simplifiers can be implemented as *discrimination nets* [14, 29] for the purpose of fast simplification. A drawback of using (localized) image sets is that the image sets will typically grow to be almost as large as $\bigcup_{i=1}^{n} S^i$, especially in purely equational problems where all equations may act as simplifiers. We remark that even if the SH^i's are very similar, the derivations generated by the deductive processes remain different, because of the partition of expansion inference steps based on the ownership of clauses.

3.2 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 (or in SH) should be allowed. The main question is whether the advantage of maintaining the SH^i's (SH) fully reduced is worth the cost of rewriting them. In the following, we propose and compare different policies.

Maintenance by direct contraction

A simple policy, with no sophisticated record-keeping, is to keep the members in SH^i fully and inter-contracted using SH^i and S^i within p_i. We call this policy "maintenance by direct contraction". A negative side is the redundancy of contraction steps, since the contraction of a datum ψ may be performed at all nodes which have a copy of ψ. A more serious problem is that in the case of SH in shared memory, maintenance by direct contraction may re-introduce the backward contraction bottleneck, since the members of the global image set SH need to be contracted in the shared memory. This problem can be avoided by updating SH in shared memory with respect to contraction at the nodes, without resorting to apply contraction to the shared memory itself, as we shall see later.

No contraction of image sets

The "no contraction" policy forbids contraction on the SH^i's (on SH) and only allows insertion of new clauses. The rationale for this policy is that if $\psi \in SH^j - S^j$ is reducible, it may be reduced at its home node p_i (such that $\psi \in S^i$) and a reduced form of ψ will be added to SH^j eventually. The "no contraction" policy is especially indicated if the image sets are used only as a data base of simplifiers. In such case, the presence of both ψ and a reduced form ψ' does not represent a serious redundancy. In fact, if the image sets are implemented as discrimination nets, frequent updates of the elements in the net may not be cost-effective. On the other hand, if the clauses in the image sets are used for expansion steps, the presence of redundant clauses would induce the generation of more redundant clauses. Therefore, we need to design other mechanisms to update the image sets with respect to contraction.

Maintenance by direct update

In Clause-Diffusion, every resident of a node has a unique *global identifier* and a *birth-time.* At each node p_i, every resident ψ of p_i is given an identifier a, so that a is the unique identifier of ψ within the local data base at p_i. Then, $< p_i, a >$ is the unique *global identifier* of ψ. The *birth-time* of ψ is the time at p_i's clock when ψ was stored as a resident at 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 SH. The global image set SH may be implemented as a *hash table*, with the global identifier as key. When a resident $< \psi, a, x > \in S^i$ is deleted or replaced by $< \psi', a, y >$, where $y > x$ is the current time at p_i's clock, node p_i also retrieves index $< p_i, a >$ in SH and deletes ψ or replaces it by ψ' in SH as well. This technique is called "maintenance by direct update". Although not as severe as in "maintenance by direct contraction", a potential risk of this approach is still that it may turn out to mimic too closely the direct application of contraction to the shared memory. The replacement of ψ by ψ' still requires a write-access and if a very high number of such accesses is generated, the conditions for a backward contraction bottleneck in shared memory might develop.

Maintenance by delayed update with garbage collection

To prevent the phenomenon mentioned above, we may adopt a mechanism of "delayed update with garbage collection". When $< \psi, a, x >$ is replaced by $< \psi', a, y >$, $y > x$, in S^i, node p_i does not retrieve and delete ψ in SH, but simply add ψ' to the bucket with key $< p_i, a >$ in the hash table SH. Therefore there is only minimal write-conflict. In general, although not necessarily, the element which was inserted most recently in a bucket is the most reduced one. Thus, each bucket may be organized as a last-in-first-out list, so that the most recently added element is easily retrieved at the top to be used as simplifier. Note that different processes may use as simplifiers different elements from the same bucket, without incurring in a read-write conflict, because all the data with the same key ($< i, a >$) are logically equivalent. Periodically, a garbage collection process visits all the buckets of the hash table and delete all the elements in each bucket except the top-most one. It may be determined empirically how often garbage collection should be executed, keeping into account the amount of shared memory available. The major potential drawback is that the garbage collection process may block the access to the shared memory. This should not be a serious problem since the top element of each bucket, the only datum which needs to be accessed by the processors, will be preserved by the garbage collection process anyway.

Update by inference messages

Identifiers and birth-times of residents also help in case of localized image sets. Assume that an inference message carries a clause together with its global identifier and birth-time, then 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 $<\psi, a, x>$ at p_i is reduced to $<\psi', a, y>$, at time $y > x$, then a new inference message $<\psi', p_i, a, y>$ will be broadcast eventually. The localized image sets can also be implemented as hash tables with the global identifier of the clauses as key. 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 smaller in SH^j. If the two clauses are not comparable, the one with more recent birth-time is saved. We call this scheme "update by inference messages". Such an immediate replacement is more reasonable in the distributed memory configuration than in the one with shared memory, because an access to SH^j is just an access to the local memory of p_j.

Update by inference messages does not help if $<\psi, a, x> \in S^i$ is deleted, rather than rewritten, by a contraction step, because no more messages with identifier $<p_i, a>$ will be issued. Therefore, localized image sets may never be updated. However, inference messages may still be useful: 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. If the performance is damaged by not updating the localized image sets with respect to deletions, one may consider broadcasting special deletion messages: a deletion message with identifier $<p_i, a>$ informs all the nodes that the resident at $<p_i, a>$ was deleted.

Different policies may be integrated in order to combine their positive features. Both our implementations of Clause-Diffusion (see [8] and Section 6) apply first update by inference messages and then direct contraction. If direct contraction is preceeded by update by inference messages, fewer direct contraction steps will be performed in general and deletion messages are not needed.

4 Policies for the distribution of new settlers

After a raw clause ψ generated at p_i has undergone the global forward contraction phase, it needs to be allocated at a node as a resident. Node p_i executes the *allocation algorithm* to determine a *final destination* p_q and it creates the *new settler* $<\psi, p_q>$. The allocation algorithm has the double objective of partitioning the search space and balancing the work-loads of the processors. In this section, we present and compare a few allocation policies.

4.1 Best-fit allocation of clauses

A *best-fit* allocation algorithm identifies a node p_q where the work-load is minimum and sends the new settler to p_q. This requires to execute a *distributed selection* algorithm to select a node p_q where the work-load is minimum. The work-load of p_i may be measured by the number of residents at p_i or by the global number of clauses at p_i. These measures may be refined to take into account also the size of the clauses. Without loss of generality, we assume that the work-load is measured by the number of residents. Then, each processor p_i maintains a counter w^i for the number of its residents. Whenever a resident is deleted from S^i or inserted in S^i, w^i is decremented or incremented. If global contraction at the source in shared memory is adopted, we may require that the work-loads w^i of all nodes are stored in the shared memory, so that they are accessible to all the processes. Each p_i is responsible for periodically updating the value of w^i in shared memory. If there is no shared memory, each node p_j may compute and keep in memory an estimate of the work-load w^i of any other node p_i, by counting the inference messages it receives from p_i or by saving the greatest value among the identifiers of the inference messages received from p_i. When a processor p_j needs to determine the final destination of a new settler, it computes the minimum w^q among all w^i's and generates the new settler $< \psi, p_q >$.

4.2 Alternate-fit allocation of clauses

Under an *alternate-fit* policy, each node p_i saves the most recently used destination, i.e. the identifier q of the node p_q, where p_i sent its most recent new settler. When p_i needs to determine the destination for the next new settler, p_i picks $(q+1)$ *mod* n, where n is the total number of processors. A variant of alternate-fit, which we called *half-alternate-fit*, works as follows: each node p_i saves the two most recently used destinations q_1 and q_2, where q_1 is the most recent one. If q_1 is p_i itself, p_i chooses $(q_2 + 1)$ *mod* p. Otherwise, i.e. $q_1 \neq p_i$, the new settler is allocated to p_i itself. Then q_2 and q_1 are updated. In other words, half-alternate-fit consists in applying alternate-fit to every other new settler and keep the remaining ones. Switching from alternate-fit to half-alternate-fit may be useful when doing experiments, if one observes that the nodes would profit from keeping for themselves more of their output. If we push this idea to the extreme, we obtain a *first-fit* allocation policy, according to which the raw clauses become residents at their birth-places. We remark that if first-fit allocation were applied to the input clauses, all input clauses and all their descendants, i.e. *all* the clauses generated during the derivation, would become residents at the node which read the input, so that the distributed derivation collapses onto a sequential derivation. The alternate-fit and first-fit policies may be combined in the *alternate-first-fit* policy, where the input clauses are distributed according to the alternate-fit policy and all successively generated clauses are assigned to the nodes where they are generated. More generally, the nodes may switch back and forth between alternate-fit and first-fit: use alternate-fit m times, then first-fit m times and so on, where m is a threshold parameter of the allocation strategy.

4.3 Discussion

Since the assignment of clauses to processors determines the work-load at the processors, one would like the allocation policy to distribute the clauses as evenly as possible, so that the work-load is well-balanced. On the other hand, deciding the allocation of clauses to nodes is part of the overhead of working in a distributed environment. An allocation algorithm which diffuses clauses evenly but is very time-consuming may not be reasonable, because the advantage of maintaining the work-load balanced may be negated by the cost of computing a complex allocation algorithm for each and every new settler. The best-fit algorithm has the advantage of being *adaptive*, as it takes into account how the work-loads at the nodes evolve during the computation. In this way, it ensures a well-balanced work-load among the processors. However, simple policies such as alternate-fit may fare better in practice than more refined allocation algorithms, which guarantee an even distribution of clauses at the cost of an higher overhead. Complicated allocation algorithms may be worthwhile in applications where it is possible that some processors are idle for a long time while others are overwhelmed with work. In theorem proving work is anything but scarce. According to the observation of our experiments and others', e.g. [15], it seems unlikely that processors be idle in distributed theorem proving. Thus, simple allocation algorithms may be sufficient. Our choice has been to implement more than one allocation policy, so that it is possible to experiment with them on different problems.

5 Inferences on residents and inference messages

When a new settler $<\psi, p_i>$ has reached its final destination p_i, it settles down as a resident at p_i. Each resident is assigned two attributes: an *identifier* and a *birth-time*. A clause ψ is given an identifier a never used before at p_i, so that a is the unique identifier of ψ within the local data base at p_i and $<p_i, a>$ is the unique identifier of ψ within the global data base. The birth-time of ψ is *the current time at p_i's clock, when ψ settles as resident at p_i*. The format of a resident is then $<\psi, a, x>$, where a is the identifier and x is the birth-time. Identifier and birth-time are employed to keep track of the modifications of residents, as we shall see shortly. If $<\psi, a, x>$ is not simplified later at p_i, p_i will eventually send a message $<\psi, p_i, a, x>$ informing other nodes of the existence of ψ as a resident at p_i.

5.1 Contraction

Contraction has been already largely covered by the description of the schemes for distributed global contraction. In addition, a Clause-Diffusion strategy may feature local contraction tasks, which involve only the local data base at each node. The distinction between global and local contraction depends in fact on the global contraction scheme the strategy uses. In case of global contraction at the source with localized image sets, there is no difference between global and local contraction steps, because any contraction task performed at a node may be done with respect to an approximation of the global data base. In case of global contraction at the source in shared memory, instead, there is a distinction, because global contraction requires to access the shared memory, whereas local contraction does not. Thus, local contraction is less expensive. Typically, a strategy would first try to reduce a clause by local contraction and access the shared memory for global contraction only if the clause was not deleted by local contraction.

The amount of local contraction in a strategy depends on the effectiveness of the global contraction scheme. The more effective the global contraction scheme is, the less stringent is the need of local contraction. In a strategy that features global contraction at the source it may be decided not to normalize received inference messages and incoming new settlers, because it can be assumed that such clauses have been normalized with respect to an approximation of the global data base at the sender.

If a resident is simplifed by a backward contraction step, it should be tested again for global contraction. This can be implemented simply by treating a simplified resident as a raw clause. A better alternative, however, is to reset the birth-time to the current clock. That is, if a resident $< \varphi, a, x >$ is reduced to φ' at time y at node p_i, then $< \varphi', a, y >$ replaces $< \varphi, a, x >$ in S^i. In this way the simplified resident φ' is regarded as a new settler at p_i, and undergoes global contraction. This scheme not only saves the cost of allocating the new clause φ', but also let φ' retain the identifier of φ. The latter is sound since φ', a contracted form of φ, is logically equivalent to φ. If a resident is deleted, e.g. by subsumption, both its identifier and birth-time are erased. In the special case of rewrite rules, i.e. oriented equations in the form $l \rightarrow r$, it is possible to stipulate that the birth-time is updated only when the left hand side is modified, since a reduction on the right hand side does not induce new inference steps.

5.2 Communication

A resident $< \psi, a, x >$ at p_i needs to be emitted eventually as an inference message $< \psi, p_i, a, x >$. If a resident $< \psi, a, x >$ is replaced by $< \psi', a, y >$, where $y > x$, through contraction, the resident with identifier a should be re-scheduled for broadcasting eventually. Intuitively, ψ' is a different clause and thus it deserves to be sent again. This is part of the policies which may be adopted to fulfill the requirement of *fairness* of a distributed derivation (see Appendix A and [4, 7]). It follows that more than one inference message may be issued for the resident with global identifier $< p_i, a >$ during a derivation. Updating the birth-times of reduced residents, as we have seen in the previous section, allows us to know for which residents the inference messages need to be repeated.

Nodes receiving inference messages forward them according to the message-passing algorithm used. The presence of contraction, however, poses the problem of how to handle contraction of messages. There are two basic possibilities, depending on whether reduced messages are forwarded:

- **No communication of contracted messages**: in this scheme the message is deleted if the clause carried by an inference message is reduced. For instance, if node p_j receives $< \psi, p_i, a, x >$ and reduces ψ to ψ', p_j simply deletes $<$

$\psi, p_i, a, x >$. This decision is taken on the ground that, since ψ is reducible, it will be reduced eventually at p_i to some ψ'', possibly $\psi'' = \psi'$, and p_i will emit $< \psi'', p_i, a, y >$. Thus, the message $< \psi', p_i, a, x >$ is redundant. The choice of not to forward reduced messages will therefore help to contain redundancy. There is, however, another issue. If ψ' happens to be a very useful clause, not forwarding it may delay success considerably. The consideration that reduced clauses should be diffused as soon as possible is the rationale for the opposite policy.

- **Communication of contracted messages**: if the clause carried by an inference message is reduced, the message is forwarded with the reduced form of the clause. More precisely, if node p_j receives $< \psi, p_i, a, x >$ and reduces ψ to ψ', p_j forwards $< \psi', p_i, a, x >$. The choice of forwarding an inference message even if it has been reduced has the advantage of diffusing a reduced clause as soon as possible. The disadvantage is the potential higher level of redundancy represented by the circulation of inference messages carrying different reduced forms of the same clause.

Because of the combination of contraction and communication, distinct inference messages, e.g. $m =< \psi, p_i, a, x >$ and $m' =< \psi', p_i, a, y >$, with the same global identifier $< p_i, a >$ but different clauses and/or different time-stamps (birth-times of the carried clauses) may be circulating. This phenomenon may be the result of one of the two scenarios: one possibility is that the message-passing algorithm forwards distinct copies of the same inference message along different paths. Then the copies which originally carry identical clauses are reduced to different forms by contraction on different paths. Another possibility is that a resident $< \psi, a, x >$ at p_i issues an inference message $< \psi, p_i, a, x >$, then the resident is reduced to $< \psi', a, y >$ and it emits another message $< \psi', p_i, a, y >$. Regardless of their origin, we call inference messages which have the same global identifier, but differ in the other components, *generalized duplicates*. If a node p_j receives two generalized duplicates, one of them is redundant: the two messages carry clauses which are logically equivalent and thus p_j should not use them both for inferences. Generalized duplicates may be deleted by contraction of the clauses they carry. In addition, we gave in [4, 7] specific inference rules to detect and delete generalized duplicates based on the fields in the messages.

The choice between forwarding or deleting contracted messages rests ultimately on experimental practice. In our prototypes, inference messages are broadcast in one hop, without forwarding at intermediate nodes. Received and possibly reduced inference messages are saved in the image sets. In case of a copy-and-forward broadcasting algorithm, it may be advisable to experiment with both the options given above.

5.3 Expansion

A node p_i executes two types of expansion inferences: expansion steps between two residents and expansion steps where a clause from an inference message paramodulates into a resident. An inference message paramodulates into a resident, but not vice versa. This restriction, which distributes expansion inferences among the nodes and prevents duplications of expansion steps, can be applied to expansion inference rules other than paramodulation as follows. For binary resolution, we say that the clause which provides the positive literal resolved upon "paramodulates into" the clause which provides the negative literal resolved upon. For hyperresolution, negative hyperresolution and unit-resulting resolution, we say that the satellites "paramodulate into" the nucleus. Thus, negative literals of inference messages are not resolved upon and inference messages serve as satellites, not as nuclei. For expansion inference rules with just one premise, such as factoring, one may establish that each node applies the rule to its residents only.

This distribution of expansion inferences, however, causes a problem with respect to the requirement that all expansion steps between persistent residents are considered (this property is part of the *uniform fairness* of a derivation, see Appendix A). A node p_j sends an inference message for every one of its residents, unless the resident is contracted beforehand. If a resident φ is reduced to some φ' at p_j, after having been sent, a new inference message for φ' will be generated eventually. Let φ be a *persistent* resident: φ becomes a resident with identifier a and birth-time y at p_j, and it is never reduced afterwards. As part of its treatment as a resident, φ is broadcast as an inference message $< \varphi, p_j, a, y >$. Let ψ be an equation which becomes a resident at some other node p_i , *after* p_i has received, processed and discarded the inference message with φ. It is guaranteed that paramodulation of ψ into φ will be tried, when ψ is sent as inference message by p_i and reaches p_j. However, φ will not have a chance to paramodulate into ψ, because a message for φ has been already generated and consumed. Since φ is persistent, it will not be simplified and thus it will never be the case that an inference message for a reduced form of φ is issued.

This problem may be solved in different ways depending on whether the processes have access to some approximation of the global data base. In they do, regardless of whether the nodes can access SH in shared memory or the SH^i's at the nodes themselves, the solution is immediate: the persistent resident φ which is missing in the above scenario is stored eventually in all the SH^i's (or in SH). Thus, it is sufficient to allow the clauses in $SH^i - S^i$ (or $SH - S^i$) to paramodulate into the clauses in S^i at each node p_i. If the processes do not have access to an approximation of the global data base, we may address the problem by introducing a type of control messages, termed *wake-up calls*. The purpose of a wake-up call directed to a node p_j is to induce node p_j to issue again an inference message for one of its residents. The disadvantage of this approach is that it may generate a large amount of communication. Although by carefully analyzing the routing algorithms one can eliminate some messages [7], it may still worsen the computation/communication ratio quite significantly. On the other hand, if the architecture under consideration has very limited amount of local memory for each node and has low latency and high throughput in communication, then this scheme may be considered. Since in this paper we have focused on global contraction at the source, thereby assuming that the nodes have access to an approximation of the global data base, we do not elaborate on wake-up calls and refer to [7] for more details.

5.4 Guidelines for the schedule of the operations at a node

In this section we outline briefly some basic *priorities* between classes of operations at a node, for a contraction-based strategy. The schedule of the operations at a node should first of all be *fair*, i.e. ensure that no needed step is postponed indefinitely (see Appendix A). Provided fairness is preserved, the general philosophy is that the operations which *reduce* the amount of data stored at the node have higher priority than those which *increase* it. The only exception to this rule is represented by the operation of *receiving* a message from the outside, since such an operation is determined by an external event which the node should perform as soon as possible.

Then, *contraction* has higher priority than *expansion* and *communication*. The goal of this rule is to make sure that each process uses as parents in expansion steps and sends/forwards to other processes clauses which have been reduced as much as possible at that node. Furthermore, contraction steps may delete clauses, eliminating the need of communication steps for them. Among *contraction* steps, a good rule is to give forward contraction priority over backward contraction. It is known since [23] that performing forward subsumption before backward subsumption is sufficient to prevent certain violations of fairness, which may result from the uncontrolled application of backward subsumption of variants. This issue becomes significantly more complicated in distributed deduction and we refer to [5] for a detailed study of *distributed subsumption*.

Communication steps have higher priority than *expansion* inferences, in order to enhance parallelism. Otherwise, if a processor does not forward an inference message until after its expansion phase, the broadcasting process may be delayed too severely.

6 Experiments

The Clause-Diffusion methodology has been implemented in two prototypes, the *Aquarius* theorem prover, developed at SUNY Stony Brook, and the *Peers* theorem prover, developed at the Argonne National Laboratory, in cooperation with Dr. Bill McCune. Both provers implement *global contraction by localized image sets in distributed memory*. Aquarius is written in C and PCN [19] for a network of workstations and features strategies for first order logic with equality. Peers is entirely written in C, using the p4 library of routines for communication [11] and focussing on strategies for equational logic. It has been developed on a network of workstations and ported on a shared memory machine (a Sequent Symmetry). On the latter, shared memory is used to simulate distributed memory and message-passing. We refer to [7, 8] for a full description of Aquarius and its experimental results and we give in the following the performances of Peers on a few problems.

The next two tables refer to our experimentation with Peers at the Argonne National Laboratory and at INRIA-Lorraine, which was also reported in [9]. At both locations, Peers ran on Sun Sparc 2 workstations communicating over Ethernet. The workstations used for our experiments were not isolated from the rest of the network and were simultaneously used by other users. In the tables, n-Peers is Peers with n nodes. The run-time of n-Peers is the CPU time expressed in seconds of the first Peer to succeed. The other Peers run till either they receive a halting message or also find a proof, whichever comes first.

Problem	*1-Peers*	*2-Peers*	*4-Peers*	*6-Peers*	*8-Peers*
x3	96.45	50.29	43.28	30.66	7.51
r2	40.04	16.51	18.74	34.97	22.31
sa1	15.99	7.30	16.06	12.96	9.65
sa2	24.28	20.09	12.76	81.05	20.34

Problem $x3$ consists in proving that $x^3 = x$ implies commutativity in ring theory. Problem $r2$ is a problem in Robbins algebra. A Robbins algebra is an algebra that satisfies the axiom $-(-(x + y) + (-(x + (-y)))) = x$, called Robbins' axiom, where the operator $+$ is associative and commutative. It is known that if a Robbins algebra also satisfies Huntington's axiom $-(-x + y) + (-(-x + (-y))) = x$, then it is Boolean [32]. Thus, one may want to investigate which axioms imply Huntington's axiom and thus are sufficient to make a Robbins algebra Boolean. Problem $r2$ consists in proving that the hypothesis that there exists an element C such that $C + C = C$, if added to Robbins' axiom, implies Huntington's axiom. Problems $sa1$ and $sa2$ are "single axioms problems" in group theory. The goal is to prove that a given single axiom is sufficient to axiomatize group theory. Problem $sa1$ consists in proving that the single axiom $f(x, g(f(y, f(f(f(z, g(z)), g(f(u, y))), x)))) = u$ implies $f(f(x, y), z) = f(x, f(y, z))$, i.e. associativity of the product f. Similarly, problem $sa2$ shows that the single axiom $f(x, f(f(e, f(f(x, f(x, f(f(x, f(y, z)), z))), z)), z)) = y$ implies associativity[2]. In some cases, e.g. $x3$, the run-time decreases somewhat regularly, albeit not linearly, as the number of nodes increases. In others, e.g. $sa1$, the run-time first improves and then gets worse with 6 and 8 nodes.

Peers features many options: different configurations of the options define different Clause-Diffusion strategies. The following table shows the run-times for five runs on the problem $x3$, denoted a, b, c, d, e, where each run corresponds to the selection of a different Clause-Diffusion strategy:

Problem	*1-Peers*	*2-Peers*	*4-Peers*	*6-Peers*	*8-Peers*
x3-a	96.28	53.58	46.87	54.04	25.95
x3-b	96.45	50.29	43.28	30.66	7.51
x3-c	96.06	51.37	44.06	43.52	28.06
x3-d	95.86	49.16	44.52	31.65	8.60
x3-e	96.36	87.64	38.34	24.93	31.02

The five strategies differ mainly in the treatment of residents reduced by backward contraction and in the distributed allocation of clauses. In strategies **a** and b, residents reduced by backward contraction are treated as raw clauses, while in strategies **c**, **d** and **e**, their birth-time is updated, but their identifier is not changed and they are not

[2]Problems $r2$, $sa1$ and $sa2$ were provided by Bill McCune.

re-allocated. In strategies **a** and **c**, clauses are allocated according to the alternate-fit policy of Section 4. In strategies **b** and **d**, clauses are allocated based on their syntax: each symbol in the signature is identified by a number in the symbol table of the prover. The destination of a clause is determined as the sum of the numbers associated to its symbols modulo the number of nodes[3]. In strategy **e**, clauses are allocated according to the best-fit policy of Section 4, where each process estimates the work-load of the other processes based on the identifiers of the inference messages it receives from the other nodes.

The third table contains a few results from the experiments conducted more recently at the University of Iowa on a network of HP workstations:

Problem	*1-Peers*	*2-Peers*	*4-Peers*	*6-Peers*	*8-Peers*
kbcomm	5.14	1.62	0.55	0.55	0.58
x3	62.42	49.40	24.09	23.80	14.03
sa1	25.84	10.05	20.32	4.60	5.29
sa2	12.97	20.73	2.24	1.56	2.15

The problem *kbcomm* is the commutator problem in group theory, a well known example for (Knuth-Bendix) completion-based strategies[4]. For these experiments the workstations were not homogeneous: the first node is an HP 715/75 with 64M of memory, the second node is an HP 715/50 with 64M of memory, the third node is an HP 715/33 with 32M of memory and the remaining five nodes are HP 705 with 16M of memory. In this third table the run-times decrease smoothly, except on problem $sa1$ with three nodes and problem $sa2$ with two nodes.

The main problem with our first Clause-Diffusion prover Aquarius was communication [8]. Our current understanding is that implementation of communication in C and better design have reduced the cost of communication in Peers. On the other hand we feel that redundancy, that is excessive overlap among the derivations at the nodes, is still a problem. The analysis of the experimental results and therefore the identification of the weaknesses of the prover (and the method) is made very difficult by the puzzling variability of the run-times: different executions of Peers on the same input clauses may exhibit different performances even for the same configuration of options. Similar unstability was displayed by Aquarius and by other distributed programs for symbolic computation, e.g. the distributed Buchberger algorithm of [12]. Some non-determinism is expected of a distributed program with asynchronous processes: it is sufficient that messages are delivered in a different order to differentiate the derivations. However, the non-determinism of Peers is higher than we had expected and it is an obstacle to improve and even to debug this type of program. Unstability is a general problem in distributed processing, but it is a new issue in theorem proving. We think that future work in distributed theorem proving will need to study non-determinism systematically.

The experimentation with Peers, together with the previous experimentation with

[3]This allocation policy is due to Bill McCune.

[4]Throughout this paper we use "contraction-based" as a general term that subsumes "completion-based", "simplification-based" and "rewriting-based".

Aquarius, encourages us to think that Clause-Diffusion has the potential of achieving significant speed-up's on some non-trivial class of problems. Much more work is needed in order to obtain more regular and more satisfactory performances. It will entail not only the fine-tuning and optimization of the implementation, but also the refinement of the Clause-Diffusion method itself, for instance in terms of better allocation policies, improved global contraction schemes and new search plans.

7 Discussion

We summarize our contributions and we suggest directions for further work.

7.1 The Clause-Diffusion methodology

The main contribution of this paper is the Clause-Diffusion methodology itself. Although our approach to parallelization applies to theorem proving in general, we focused on *contraction-based strategies*. This choice has two reasons. First, these strategies proved to be more powerful than strategies without contraction on significant classes of problems. Second, the presence of contraction rules makes the parallelization more difficult. As it has been often observed, those features which make a sequential algorithm or procedure more efficient than others, are also those which may make its parallelization harder. Thus, the study of strategies without contraction can be regarded as a special case of the study of strategies with contraction.

Our Clause-Diffusion methodology realizes a notion of *parallelism at the search level*: the search space is partitioned among deductive processes, which search concurrently and cooperatively for a solution. The search space is determined by the input problem, i.e. the clauses, and the inference rules. One may partition the set of clauses or the set of rules or a combination of the two. The first approach amounts to *data-driven parallelism*: each concurrent process is given a subset of the data and it is in charge of all the operations on its data. The second one is *operations-driven parallelism*: each concurrent process is given a subset of the operations, e.g. the inference rules, and it is responsible for applying them to all the data. In theorem proving applications, there are many more data items than inference rules and therefore data-driven parallelism is more natural.

Accordingly, the Clause-Diffusion methodology partitions the clauses among the processes, so that each one of them owns a portion (its *residents*) of the data base of clauses. The expansion inferences are subdivided based on the ownership of the clauses. Contraction inferences are not subdivided, so that each process can perform as much contraction as possible. Exactly because each process is assigned just a portion of the search space, the processes need to communicate, in order to find a proof whenever there exists one. Communication is realized via *message-passing*: the processes send their residents to the other processes in form of messages, termed *inference messages*. In a distributed environment each deduction process runs at a different node and they exchange messages over the links interconnecting the nodes.

Specific Clause-Diffusion strategies are obtained by selecting several components, beside the inference rules and the search plan, such as the distributed global contraction scheme, the distributed allocation algorithm and the algorithms for message-passing. Also, the search plan to be executed by each deduction process needs to schedule both inference and communication steps. For most of these components we analyzed the underlying problems and proposed several solutions:

- Schemes for distributed global contraction: global contraction by travelling and global contraction at the source, either by localized image sets or by a global image set in shared memory.
- Policies to maintain the image sets with respect to contraction: direct contraction, no contraction, direct update, delayed update with garbage collection or update by inference messages.
- Criteria for distributed allocation: best-fit, alternate-fit, half-alternate-fit, first-fit or alternate-first-fit.

7.2 Distributed global contraction

We formulated the problem of global contraction in parallel automated deduction, 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 theorem proving. This source of inefficiency was not identified before. We proposed schemes for distributed global contraction, which represent our solution to the backward contraction bottleneck problem.

Global contraction at the source by localized image sets represents a *distributed, duplication-oriented approach*, which requires a sufficiently large amount of memory at the nodes. It is duplication-oriented, because parallel inferences are made possible by forming the localized image sets, i.e. by duplicating the clauses.

Global contraction at the source by global image set represents a *mixed shared-distributed approach*, which relies on a fast shared memory and also sufficient memory at the nodes. This is a hybrid approach. It reduces the amount of communication, because exchange of messages may be replaced in part by access to the shared memory. It still involves duplication, since all the clauses in the shared memory are duplicates of the residents at the nodes. But duplication is reduced with respect to the previous approach, because just one shared global image set, rather than many localized image sets, is maintained.

In all our schemes, the clauses to be rewritten by contraction are held in the local memories of the nodes. This is the key feature which prevents the backward contraction bottleneck. The main costs of our solutions are represented by memory and/or communication requirements as indicated above. In particular in the mixed approach, the backward contraction bottleneck does not appear, even if a shared memory is used, because the clauses in shared memory are used as simplifiers only. They are not subject to contraction themselves, while the residents at the node are. The negative sides are the duplication of clauses and the delay in updating the shared memory with respect to the nodes.

This mixed approach seems to be especially appealing. The general wisdom is that distributed memory may be more convenient for intrinsically independent, asynchronous activities, while shared memory may be more appropriate for concurrent activities which cooperate closely and synchronously. We feel that parallel deduction processes with contraction involve activities of both types. Expansion and local contraction may be considered as belonging to the first category, while global contraction may fall in the second one. Indeed, the repeated contraction of a clause with respect to a given set of simplifiers may be conceived as a single inference step, rather than a combination of steps. Under such view, it becomes apparent that it may be convenient to have all the simplifiers in one place. Our scheme for global contraction at the source with shared memory seems promising because it allows to share the simplifiers, without incurring in the backward contraction bottleneck of other purely shared memory approaches.

7.3 Comparison with related work

Most previous works in parallel deduction do not apply to contraction-based strategies (see [30] for a survey). The pure shared memory approaches of [24] and [33] have been applied to contraction-based strategies, with the backward contraction bottleneck. The transition-based philosophy of [33] has been applied also to an implementation of the Buchberger algorithm in distributed memory [13]. The theorem prover ROO of [24] performs very well and scales regularly with the number of processors on problems solved by hyperresolution, i.e. problems suitable for expansion-oriented strategies, but it has displayed the backward contraction bottleneck on equational problems, where backward contraction plays a key role.

The approach proposed in [2, 17] addresses contraction-based strategies and does not cause the backward contraction bottleneck. This method, called the *Team-Work method*, also features concurrent, deductive processes which develop their derivations in a largely indipendent way. However, there are more differences than similarities between Team-Work and Clause Diffusion. First, in the basic Team-Work method the search space is not partitioned: each deductive process owns all the clauses in its data base and executes all the inferences[5]. The different deductive processes start with identical sets of clauses and inference rules. The difference among the deductive processes is made by having them executing different search plans. Second, the Team-Work method has no message-passing. Third, the deductive processes are not completely asynchronous. Periodically, the processes halt their deductive work and evaluate their current data bases as "referees". One of the processes plays the role of "supervisor": based on these evaluations, it forms a "best" data base and gives it to all the processes, which then may re-start the deduction. Thus, the processes communicate by periodical synchronizations and re-constructions of a common data base, rather than by message-passing between asynchronous processes as in Clause-Diffusion.

[5]There are variants of the Team-Work method where some processes – the "non-fair experts" of [2, 17] – do only some type of inference and do not need to have all the clauses.

The most fundamental difference, however, is in the way the two methods try to obtain speed-up over a sequential strategy. The Team-Work method tries to achieve speed-up by *interleaving of search plans.* Suppose that clause ψ and clause φ yield a proof. Assume that search plan A generates ψ very early and φ much later and search plan B generates φ very early and ψ much later. Thus, if either A or B are executed sequentially, this proof is not obtained till both clauses are generated. In the Team-Work method, A and B are executed in parallel. If the common data base is reconstructed after A has generated ψ and B has generated φ, and if both ψ and φ are saved in the new data base, then the process executing A (or B) will have φ (or ψ) much earlier than it would in a sequential execution, and thus will find that proof much faster. In this sense, the Team-Work method realizes interleaving of search plans. Computations generated by interleaving, however, are not regarded as "truly concurrent", since they can be naturally simulated by sequential executions. Indeed, a sequential strategy, which executes first A for at least as many steps as it is necessary to generate ψ and then B for at least as many steps as it is necessary to generate φ, would yield a derivation very similar to that of the Team-Work method. The Team-Work method has been implemented for equational logic. The experimental results in [2, 17], are not sufficient to establish how the Team-Work method scales with the number of processors, because they refer only to the case of two nodes.

Clause-Diffusion, on the other hand, tries to achieve speed-up by subdividing the problem, i.e. the search space, so that each deductive process faces a smaller search task than the sequential process. The effect of partitioning the search space and of asynchronous communication by message-passing is that the deductive processes generate portions of the search space which may be radically different from those generated sequentially. Since the portions of the search space can be different, a Clause-Diffusion process may find a much faster proof than the sequential one. If this happens, the result is very good. Because of this dramatic changes in the search spaces, the derivations by Clause-Diffusion may be very different from the sequential ones and it is not at all obvious how to simulate a Clause-Diffusion derivation by a sequential method. In this sense, we feel that Clause-Diffusion represents a "truly concurrent" approach, and takes a bolder step in parallelization of theorem proving than other methods. There is, of course, the potential drawback that a Clause-Diffusion strategy may not partition the search space properly. Then the speed-up may be disappointing, as the Clause-Diffusion prover may generate a proof similar to the sequential one by searching different fragments of the search space, rather than the whole space seen by the sequential prover. This type of phenomena explains in part the irregularity of some experiments with both Aquarius, see [8], and Peers.

7.4 Directions for future research

Many directions for ongoing and future work can be envisioned. On the practical side, we plan to continue the development and fine-tuning of our prototypes. The experimentation conducted so far has led already to variations and enhancements. For instance, in Aquarius the nodes may execute different search plans and select different subsets of the available inference rules. Indeed, the Clause-Diffusion method does not require to have the same strategy at all sites. We expect that most of the room for improvement is at the level of the search plans, i.e. the criteria to allocate and sort clauses and inference/communication steps. The experimentation done so far has shown that the allocation algorithm plays an especially important role in determining the performances of a Clause-Diffusion prover. Therefore, we shall consider the problem of designing better criteria to decide where to allocate a given clause during the derivation. Such criteria may keep into account statistics about the derivations developed so far at the nodes. Examples of statistics are the number of clauses stored at a node, the number of clauses of a certain type, e.g. simplifiers, the time spent so far in performing a certain type of inference and so on. Another possibility is to design criteria based on the syntax of the clauses, e.g. distributing the clauses according to some partition of the signature. Our second prototype Peers features a simple criterion of this nature.

Aquarius and Peers implement a distributed duplication oriented approach, since global contraction is done by localized image sets. Another direction is to develop in detail and implement the mixed shared-distributed approach, with a global image set in shared memory. This will require an environment where each node, while still having a fairly large local memory, may access a shared memory component.

On the theoretical side, we may work on the design of *parallel search plans*. In the present work, we assumed a sequential search plan for the local inferences at a single node. Such a search plan is simply extended to incorporate communication according to the priorities dictated by the Clause-Diffusion methodology. The next phase is to study how to develop intrinsically parallel search plans, i.e. search plans which keep into account that the strategy is executed in a distributed environment.

Acknowledgements

The bulk of the work on Clause-Diffusion 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. The prototype Peers was jointly written by Dr. Bill McCune of the Argonne National Laboratory and the first author, while she was visiting Argonne. The experimentation with Peers was started at the Argonne National Laboratory and continued at INRIA-Lorraine by the first author. The support provided by both laboratories is much appreciated.

References

[1] S.Anantharaman and J.Hsiang, Automated Proofs of the Moufang Identities in Alternative Rings, *Journal of Automated Reasoning*, Vol. 6, No. 1, 76–109, 1990.

[2] J.Avenhaus and J.Denzinger, Distributing Equational Theorem Proving, in C.Kirchner (ed.), *Proceedings of the Fifth Conference on Rewriting Techniques and Applications*, Montréal, Canada, June 1993, Springer Verlag, Lecture Notes in Computer Science 690, 62–76, 1993.

[3] L.Bachmair and H.Ganzinger, Non-Clausal Resolution and Superposition with Selection and Redundancy Criteria, in A.Voronkov (ed.), *Proceedings of Logic Programming and Automated Reasoning*, Springer Verlag, Lecture Notes in Artificial Intelligence 624, 273–284, 1992.

[4] M.P.Bonacina and J.Hsiang, On fairness in distributed deduction, in P.Enjalbert, A.Finkel and K.W.Wagner (eds.), *Proceedings of the Tenth Symposium on Theoretical Aspects of Computer Science*, Würzburg, Germany, February 1993, Springer Verlag, Lecture Notes in Computer Science 665, 141–152, 1993.

[5] M.P.Bonacina and J.Hsiang, On subsumption in distributed derivations, to appear in the *Journal of Automated Reasoning.*

[6] M.P.Bonacina and J.Hsiang, Towards a Foundation of Completion Procedures as Semidecision Procedures, to appear in *Theoretical Computer Science.*

[7] M.P.Bonacina, Distributed Automated Deduction, Ph.D. Thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.

[8] M.P.Bonacina and J.Hsiang, Distributed Deduction by Clause-Diffusion: the Aquarius Prover, in A.Miola (ed.), *Proceedings of the Third International Symposium on Design and Implementation of Symbolic Computation Systems*, Gmunden, Austria, September 1993, Springer Verlag, Lecture Notes in Computer Science 722, 272–287, 1993.

[9] M.P.Bonacina and W.McCune, Distributed theorem proving by *Peers*, in A. Bundy (ed.), *Proceedings of the Twelfth International Conference on Automated Deduction*, Nancy, France, June 1994, Springer Verlag, Lecture Notes in Computer Science, to appear.

[10] S.Bose, E.M.Clarke, D.E.Long and S.Michaylov, Parthenon: A parallel theorem prover for non-Horn clauses, *Journal of Automated Reasoning*, Vol. 8, N. 2, 153–182, April 1992.

[11] R.Butler and E.Lusk, User's Guide to the p4 Programming System, Technical Report ANL-92/17, October 1992.

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

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

[14] J.D.Christian, High-Performance Permutative Completion, Ph.D. thesis, The University of Texas at Austin, available as MCC Technical Report ACT-AI-303-89, August 1989.

[15] E.M.Clarke, D.E.Long, S.Michaylov, S.A.Schwab, J.-P.Vidal and S.Kimura, Parallel Symbolic Computation Algorithms, Technical Report CMU-CS-90-182, School of Computer Science, Carnegie Mellon University, October 1990.

[16] S.E.Conry, D.J.MacIntosh and R.A.Meyer, DARES: A Distributed Automated REasoning System, in *Proceedings of the Eleventh Conference of the American Association for Artificial Intelligence*, 78–85, 1990.

[17] J.Denzinger, Distributed knowledge-based deduction using the team work method, Technical Report SR-91-12, University of Kaiserslautern, 1991.

[18] N.Dershowitz and J.-P.Jouannaud, Rewrite Systems, Chapter 15, Volume B, *Handbook of Theoretical Computer Science*, North-Holland, 1989.

[19] I.Foster and S.Tuecke, Parallel programming with PCN, Technical Report ANL-91/32, Version 1.2, December 1991.

[20] D.J.Hawley, A Buchberger Algorithm for Distributed Memory Multi-Processors, in *Proceedings of the International Conference of the Austrian Center for Parallel Computation*, Linz, Austria, October 1991, Springer Verlag, Lecture Notes in Computer Science.

[21] A.Jindal, R.Overbeek and W.Kabat, Exploitation of parallel processing for implementing high-performance deduction systems, *Journal of Automated Reasoning*, Vol. 8, 23–38, 1992.

[22] D.Kapur and H.Zhang, RRL: a Rewrite Rule Laboratory, in E.Lusk, R.Overbeek (eds.), *Proceedings of the Ninth International Conference on Automated Deduction*, Argonne, Illinois, May 1988, Springer Verlag, Lecture Notes in Computer Science 310, 768–770, 1988.

[23] D.W.Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam, 1978.

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

[25] W.W.McCune, OTTER 2.0 Users Guide, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, March 1990.

[26] M.Rusinowitch, *Démonstration Automatique: Techniques de Réécriture*, Collection Science Informatique, InterEditions, Paris, 1989.

[27] J.Schumann and R.Letz, PARTHEO: A High-Performance Parallel Theorem Prover, in M.E.Stickel (ed.), *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, Germany, July 1990, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 28–39, 1990.

[28] K.Siegl, Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages, Master thesis and Technical Report N. 90-54.0, RISC-LINZ, November 1990.

[29] M.E.Stickel, The Path-Indexing Method for Indexing Terms, Technical Note 473, SRI International, October 1989.

[30] C.B.Suttner and J.Schumann, Parallel Automated Theorem Proving, in L.Kanal, V.Kumar, H.Kitano and C.B.Suttner (eds.), *Parallel Processing for Artificial Intelligence*, Elsevier, 1994.

[31] J.-P.Vidal, The Computation of Gröbner Bases on A Shared Memory Multiprocessor, in A.Miola (ed.), *Proceedings of International Symposium on the Design and Implementation of Symbolic Computation Systems*, Capri, Italy, April 1990, Springer Verlag, Lecture Notes in Computer Science 429, 81–90, 1990. Full version available as Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, August 1990.

[32] L.Wos, Searching for Open Questions, Newsletter of the Association for Automated Reasoning, No. 15, May 1990.

[33] K.A.Yelick and S.J.Garland, A Parallel Completion Procedure for Term Rewriting Systems, in D.Kapur (ed.), *Proceedings of the Eleventh International Conference on Automated Deduction*, Saratoga Springs, New York, June 1992, Springer Verlag, Lecture Notes in Artificial Intelligence 607, 109–123, 1992.

A Uniform fairness of distributed derivations

In this appendix we extend the definition of uniform fairness for sequential theorem proving derivations to distributed derivations by Clause-Diffusion; we give a characterization of distributed uniform fairness in three conditions and we prove that these conditions are sufficient for the uniform fairness of a distributed derivation. These conditions are more concrete than the general definition of uniform fairness and are therefore useful to establish the fairness of specific Clause-Diffusion strategies[6].

Given n deductive processes $p_1 \dots p_n$, we define the *local data base* at node p_k at stage i as the union $G_i^k = S_i^k \cup M_i^k \cup CP_i^k \cup NS_i^k$. Then, the *local limit* at processor p_k is $G_\infty^k = \bigcup_{i\geq 0} \bigcap_{j\geq i} G_j^k$. The *global data base* at stage i is the union of the local data bases, i.e. $G_i = \bigcup_{k=1}^n G_i^k$, and the *global limit* is $G_\infty = \bigcup_{k=1}^n G_\infty^k$. Local and global limits may

[6]This material appeared already in [7, 4]. Therefore it is reported here only as an appendix for the sake of completeness of the treatment of Clause-Diffusion in this paper.

be defined similarly for each component of the states in a distributed derivation, e.g. S^k_∞ and S_∞, M^k_∞ and M_∞.

Then, we apply to distributed derivations the definition of uniform fairness in the sequential case. A few (related) definitions of uniform fairness in theorem proving have been given over the years (citations and comparisons can be found in [7]). Here we apply the definition given in [3]. In this definition, the notation $R(S)$ denotes the set of clauses that are *redundant* in S based on the *redundancy criterion* R of the given strategy (see [3] and [7] for details). The notation $I_e(S)$ denotes the set of clauses that can be inferred from premises in S in one expansion step of the given strategy.

Definition A.1 *A distributed derivation*

$(S; M; CP; NS)^k_0 \vdash_{\mathcal{C}} (S; M; CP; NS)^k_1 \vdash_{\mathcal{C}} \ldots (S; M; CP; NS)^k_i \vdash_{\mathcal{C}} \ldots,$

for all k*,* $1 \leq k \leq n$*, is* uniformly fair *if* $I_e(G_\infty - R(G_\infty)) \subseteq \bigcup_{k=1}^n \bigcup_{j\geq 0} G^k_j$.

This means that all the clauses that can be generated from persistent and non-redundant premises are considered eventually. The following three conditions form a more concrete specification of uniform fairness of distributed derivations.

1. All messages should be processed eventually and thus there are *no persistent messages*:

   $\forall k$, $1 \leq k \leq n$, $M^k_\infty = CP^k_\infty = NS^k_\infty = \emptyset$.

2. Given k and $\psi \in S^k_\infty$, we define the *abstract birth-time* of ψ in k to be the smallest index $i \geq 0$ such that $\psi \in \bigcap_{j\geq i} S^k_j$.[7] Then for every node p_k and for every persistent, non-redundant resident φ at p_k, *all persistent, non-redundant residents at the other nodes will eventually appear as inference messages at* p_k*, after the birth of* φ:

   $\forall k$, $1 \leq k \leq n$, $\forall \varphi \in (S^k_\infty - R(S^k_\infty))$, if i is the abstract birth-time of φ, then $\forall h$, $1 \leq h \neq k \leq n$, $\forall \psi \in (S^h_\infty - R(S^h_\infty))$, there exists an $l > i$ such that $\psi \in M^k_l$.

   Notice that i and l are stages of the same derivation, i.e. the derivation at p_k.

3. The derivation is uniformly fair with respect to the local inferences at each node. That is, every clause that can be generated from persistent, non-redundant clauses at p_k will be generated: $\forall k$, $1 \leq k \leq n$, $I_e(G^k_\infty - R(G^k_\infty)) \subseteq \bigcup_{i\geq 0} CP^k_i$.

Conditions 1 and 2 take care of the distributed part of the derivation, by guaranteeing that a clause which can be generated from two persistent non-redundant clauses φ_1 and φ_2 residing at two different nodes, will be considered. Condition 2 ensures that φ_1 and φ_2 will eventually meet each other through inference messages. Condition 1 makes sure that all inference messages be processed ($M_\infty = \emptyset$), all raw clauses (those that, in the presence of contraction rules, remain non-trivial after having been fully contracted) become new settlers ($CP_\infty = \emptyset$) and all new settlers become residents at some place

[7]The adjective *abstract* indicates that i is an index in the abstract view of the derivation, and not a time of any processor's clock.

($NS_\infty = \emptyset$). Condition 3 takes care of fairness of the local derivation at each node. The following theorem shows that these three conditions imply Definition A.1:

Theorem A.1 *If a distributed derivation is such that*

1. *$\forall k$, $1 \leq k \leq n$, $M^k_\infty = CP^k_\infty = NS^k_\infty = \emptyset$,*
2. *$\forall k$, $1 \leq k \leq n$, $\forall \varphi \in (S^k_\infty - R(S^k_\infty))$, if i is the abstract birth-time of φ, then, $\forall h$, $1 \leq h \neq k \leq n$, $\forall \psi \in (S^h_\infty - R(S^h_\infty))$, there exists an $l > i$ such that $\psi \in M^k_l$ and*
3. *$\forall k$, $1 \leq k \leq n$, $I_e(S^k_\infty - R(S^k_\infty)) \subseteq \bigcup_{i\geq 0} CP^k_i$.*

then $I_e(G_\infty - R(G_\infty)) \subseteq \bigcup_{k=1}^n \bigcup_{i\geq 0} G^k_i$, i.e. the distributed derivation is uniformly fair*.*

Proof: let φ be any clause in $I_e(G_\infty - R(G_\infty))$ with parents ψ_1 and ψ_2. Since $M_\infty = CP_\infty = NS_\infty = \emptyset$, $G_\infty = S_\infty$, i.e. $\psi_1, \psi_2 \in S_\infty$. It follows that $\psi_1 \in (S^k_\infty - R(S_\infty))$ and $\psi_2 \in (S^h_\infty - R(S_\infty))$ for some $1 \leq k, h \leq n$.

If $k = h$, then $\varphi \in I_e(S^k_\infty - R(S_\infty))$. Since $S^k_\infty \subseteq S_\infty$, by the monotonicity of the redundancy criterion (see [3]), $R(S^k_\infty) \subseteq R(S_\infty)$ and thus $I_e(S^k_\infty - R(S_\infty)) \subseteq I_e(S^k_\infty - R(S^k_\infty))$. By Condition 3, there exists an i such that $\varphi \in CP^k_i \subseteq G^k_i \subseteq \bigcup_{k=1}^n \bigcup_{i\geq 0} G^k_i$.

If $k \neq h$, let i_1 and i_2 be the abstract birth-times of ψ_1 at p_k and ψ_2 at p_h respectively. By Condition 2, we have $\psi_1 \in M^h_{l_1}$ for some $l_1 > i_2$ and $\psi_2 \in M^k_{l_2}$ for some $l_2 > i_1$. Since $M_\infty = \emptyset$ by Condition 1, we know that the inference message ψ_1 does not persist at p_h and the inference message ψ_2 does not persist at p_k. An inference message may be deleted before performing expansion steps, by a contraction step. Since ψ_1 and ψ_2 are in $G_\infty - R(G_\infty)$, i.e. they are globally persistent and non-redundant, this is impossible. It follows that the inference messages $\psi_1 \in M^h_{l_1}$ and $\psi_2 \in M^k_{l_2}$ are deleted only after having been processed. Thus, paramodulation of ψ_1 into ψ_2 is tried at p_h and paramodulation of ψ_2 into ψ_1 is tried at p_k. Either one of these two steps generates φ, i.e. either $\varphi \in CP^h_i$ or $\varphi \in CP^k_i$ at some stage i, i.e. $\varphi \in \bigcup_{k=1}^n \bigcup_{i\geq 0} G^k_i$. $\square$

By this theorem, the abstract definition of uniform fairness is reduced to more concrete requirements. Given a complete sequential strategy $\mathcal{C} =< I; \Sigma >$ (the inference system I is refutationally complete and the search plan Σ is uniformly fair), its parallelization by Clause-Diffusion is also complete, provided that the above conditions are satisfied:

Corollary A.1 *Let $\mathcal{C} =< I; \Sigma >$ be a complete sequential theorem proving strategy and $\mathcal{D}$ be a Clause-Diffusion strategy with inference system I and search plan Σ at each node. If the algorithms and policies handling messages satisfy Conditions 1 and 2, then $\mathcal{D}$ is a complete distributed theorem proving strategy.*

We review briefly how the Clause-Diffusion strategies described in this paper satisfy Condition 2 (see [7, 4] for more details). Condition 2 requires that for every non-redundant, persistent resident ψ_1 at any node p_k, any other non-redundant, persistent resident ψ_2 appear at p_k as inference messages, after the abstract birth-time of ψ_1. We assume that ψ_1 is born "before" ψ_2, i.e. $i_1 \leq i_2$ for i_1 and i_2 the birth-times of ψ_1 and

ψ_2 respectively. The case where $i_2 \leq i_1$ is clearly symmetrical. In order to make sure that the "younger" clause ψ_2 paramodulates into the "older" clause ψ_1, it is sufficient that ψ_2 is broadcast and reaches all nodes. For this purpose, we have required that any new settler ψ, which settles down at a node, be emitted as inference message eventually, unless it is deleted before hand. However, it is not sufficient to consider new settlers, because residents are subject to backward contraction. Thus we have introduced the policy that whenever a resident is reduced, its birth-time is updated and thus will be re-scheduled for emission as inference message (see Section 5.2). Of course this means that in practice also non-persistent residents are broadcast, since it is not possible to predict which clauses will be persistent. In order to ensure that the "younger" equation ψ_2 is paramodulated into by the "older" clause ψ_1, we may use either the image sets or the wake-up calls, as explained in Section 5.3.