Distributed Theorem Proving by *Peers*

Maria Paola Bonacina[1⋆] and William W. McCune[2]

[1] Department of Computer Science
University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu
[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4801, USA
mccune@mcs.anl.gov

1 Overview

Peers is a prototype for parallel theorem proving in a distributed environment. It implements a number of strategies for refutational, contraction-based theorem proving in equational theories, whose signatures may include associative-commutative (AC) function symbols. "Contraction-based" strategies is a more general term for "simplification-based", "rewriting-based" or "(Knuth-Bendix) completion-based" strategies.

Such strategies are parallelized in Peers according to the *Clause-Diffusion* methodology for distributed deduction [2, 4]. This is a *coarse-grain* approach to parallel theorem proving, where *concurrent, asynchronous deduction processes* work cooperatively to solve the given theorem proving problem. In Clause-Diffusion, the deduction processes are rather independent and *loosely-coupled*: each process executes a theorem proving strategy, has its own data base of clauses and develops its own derivation. The processes are all "peers" (hence the name of the prototype) in the sense that there is no master-slave type of organization between them and there are no specialized processes, e.g. a process that performs only normalization of terms. In Clause-Diffusion, the processes may execute the same strategy or different strategies. In the current version of Peers, all processes execute the same strategy.

The work is subdivided among the processes by dynamically partitioning the data base of clauses. Whenever a clause is generated, a *distributed allocation algorithm* is used to decide which process is its owner. Each process executes the allocation algorithm for every clause it generates: all allocation decisions are taken locally. A partition of the clauses induces a subdivision of the inferences by stipulating that each process is responsible for the inference steps on its clauses, called its *residents*. This "data-driven" distribution of the theorem proving task serves two purposes. The first one is to subdivide the inferences, in such a way that each parallel process has less work than a single sequential process would. The second one is to balance the work-load of the parallel processes. As a consequence of this partition, each process has direct access to just a subset of the data base. Since finding a proof requires in general global knowledge, the processes need to communicate clauses to each other by *message-passing*. Thus, a derivation by a process includes both inference steps and communication steps and the collection of these derivations forms the *distributed derivation*. As soon as one of the processes finds a proof, the distributed derivation halts successfully.

The Clause-Diffusion methodology does not assume a specific parallel architecture. Given the coarse-grain type of parallelism in Clause-Diffusion, a purely distributed environment, such as a network of computers or a multiprocessor with distributed memory, is a natural choice. However, Clause-Diffusion

⋆ Supported in part by the GE Foundation Faculty Fellowship to the University of Iowa.

may also take advantage of a shared memory, by implementing send/receive of messages as read/write in a shared part of the memory. In Peers, the creation of parallel processes and the communication are realized by using the p4 library of functions for parallel programming [6]. Since p4 works on a variety of parallel machines, neither the method (Clause-Diffusion) nor the software (p4 and C) pose significant architecture-related restrictions. Indeed, we developed Peers on a network of workstations, but we also ported it to a shared-memory Sequent. Networks of workstations remain our basic target environment, because they are widely available. In such clusters, each workstation hosts a deductive process and the deductive processes communicate via message-passing over the network.

Peers is one of the few coarse-grain, contraction-based, parallel theorem provers written so far (two other such provers are e.g. [1, 5]). The current version is for equational theories only, but Peers can be extended to larger logics without modifying its basic architecture. To our knowledge, Peers is the first parallel prover to feature built-in treatment of AC operators. Peers is small, portable and easy to use: it is not interactive, but the user can drive the behaviour of the prover by setting *flags* (boolean options) and *parameters* (integer options) in the input file. We refer to [2, 4] for more details on Clause-Diffusion. In the following, we describe the design of Peers, some of its strategies, and we discuss a selection of experiments.

2 The Structure of Peers

The Peers program expects an input file, which contains the input clauses and options. One of the processes, *peer_0*, reads the input file, pre-processes the input clauses according to the strategy and broadcasts the input clauses, the symbol table for the input signature, the options and the same initialization information to all the processes. After this initialization phase, all "peers" become active and the distributed derivation starts.

During the distributed computation, each deduction process may be in one of three states: answering a message from another process, performing local work, i.e. inferences in its local data base, or being idle. Each peer executes a basic loop, called the *work-loop*, where it tests for pending messages, selects the pending message of highest priority and serves it by executing the appropriate action. If no message is pending, the process executes a unit of local work. If no such work is available, the process becomes idle and remains idle until it receives a message. After processing a message or a unit of local work, the peer repeats the selection. It is necessary to break the local work into units, because a theorem proving process may generate an infinite derivation by performing local inferences only, and thus it would indefinitely postpone handling the messages. Also, all messages from the outside have higher priority than local work, on the ground that communication is often the bottleneck in distributed computation and thus should have the highest priority.

If a peer finds a proof or raises an exception, e.g. running out of memory, it broadcasts a *halt message* and halts. All other peers will halt upon receiving a *halt message*. The Clause-Diffusion methodology is fault-tolerant: if a deductive process runs out of memory and halts, the other processes may continue without hindrance for the soundness and completeness of the derivation [2, 5]. The choice of halting all peers when one raises an exception was motivated by the intent of keeping this prototype simple. In addition to successful termination and exceptional termination, Peers terminates if all the processes are idle and all messages ever sent have been received. In order to detect this condition, Peers implements the *Dijkstra-Pnueli global termination detection algorithm*: this is a distributed algorithm that circulates a token, carrying status information, among the processes, and establishes termination if the token has circulated at least twice without detecting any activity (see e.g. [9], pages 48–49 and 182–185, for a definition and proof of correctness of this algorithm).

Given this organization, specific Clause-Diffusion strategies can be implemented by determining the types of work (messages and units of local work), with their priorities and corresponding actions.

3 The "Types of Work"

In the current version of Peers, the main types of work (e.g. excluding the messages used only during the initialization phase and the messages for termination) are two types of messages, *new settlers* and *inference messages*, and a type for the unit of local work, called *expansion work* from *expansion inference rules*, e.g. paramodulation and resolution.

New settlers and *inference messages* are the two basic types of messages in the Clause-Diffusion methodology. The purpose of *new settlers* is to subdivide the work among the processes by dynamically distributing the clauses. Whenever a new clause c is generated by a process, say $peer_i$, c is subject first to *forward contraction*, that is contraction, e.g. simplification and subsumption, with respect to the existing clauses in the data base of $peer_i$. If c is not deleted, $peer_i$ executes the allocation algorithm to decide which process c should be a resident of. If the result is that c belongs to $peer_i$, c is used for *backward contraction*, that is, c is applied to contract the existing clauses in the data base of $peer_i$. Then it is stored in the data base as a resident of $peer_i$. If the allocation algorithm decides that c should be a resident of another process $peer_j$, $peer_i$ sends c to $peer_j$ as a new settler message. Upon reception of the message, $peer_j$ performs forward and backward contraction on c and then stores it as one of its residents.

Each process performs the expansion inferences between its residents. If each process executes only steps between its residents, the strategy would not be complete. The purpose of *inference messages* is to preserve completeness by making inferences between residents at different nodes possible. We refer to [2, 3] for a treatment of completeness of Clause-Diffusion. When a process $peer_i$ selects its resident c for *expansion work*, it also broadcasts c as an inference message. Appropriate book-keeping is in place to avoid sending a clause as an inference message more than once. Any other process $peer_j$ will receive c and perform expansion between c and its residents and (backward) contraction on c.

Peers features two mechanisms for organizing expansion inferences, one of which can be selected by setting appropriate flags. In the first mechanism, similar to that of Otter [8], a unit of local work consists in selecting a *given clause* c and performing all expansion inferences between c and the other clauses in the local data base. In the second mechanism, a unit of local work consists in selecting a pair of clauses (c, d) and performing all expansion inferences between c and d. The second mechanism was designed for AC theories: if paramodulation is done modulo AC, there are generally so many AC-paramodulants that generating all the AC-paramodulants between the given equation and all the other equations is a too large amount of work to be a single unit of expansion work. In both mechanisms, the inferences are subdivided based on the ownership of clauses: for instance, for paramodulation, each process executes only paramodulations into its residents, i.e. it paramodulates either a resident or a received inference message into a resident. Therefore, distinct peers perform different selections of steps in different orders, so that their computations are different.

4 Some Strategies in Peers

Peers has 23 flags and 13 parameters and different settings of some of these options define different strategies. An important parameter is the one that controls the choice of the allocation algorithm. Peers currently has three allocation algorithms. In the *"rotate"* allocation algorithm, each peer keeps track of the most recently used destination q, including itself, and simply picks $q + 1 \ mod \ n$, where n is the number of processes. The *"syntax"* allocation algorithm maps a clause to a process based on the syntax of the clause. Each symbol in the signature is associated to an integer code, its key in the symbol table. A clause is assigned to the process whose number is the sum of the codes of all the symbols in the clause modulo the number of processes. This algorithm has the nice property that it sends all identical copies of a clause to the same destination, where all but one can be deleted by subsumption. The *"select-min"* allocation algorithm

chooses the peer which is estimated to have minimum work-load. For the purpose of this allocation criterion, each process needs to have some information on the work-load of all the other processes. Currently, the number of residents at a process is used as measure of its work-load. A process $peer_i$ may estimate the number of residents of another process $peer_j$ based on the inference messages that $peer_i$ receives from $peer_j$. Intuitively, the higher are the identifiers of inference messages from $peer_j$, the higher is the number of residents at $peer_j$. Thus, process $peer_i$ saves the identifier of the most recently received inference message from $peer_j$. This number is regarded by $peer_i$ as an esteem of the work-load of $peer_j$.

Another option which is relevant to strategies definition, is the parameter that controls the treatment of backward contracted clauses. In many sequential contraction-based theorem provers, the reduced form of a backward-contracted clause is regarded as a new clause. This approach is available in Peers. It has the advantage of being a uniform, simple treatment of all backward-contracted clauses. However, in the distributed case, one may want to restrict the extent to which backward contraction of clauses causes the clauses to be re-allocated. Accordingly, another possibility in Peers is to treat backward-contracted residents as new clauses, but for re-allocation: if c, resident of $peer_i$, is reduced to c' by backward contraction at $peer_i$, c' is regarded as a new settler settling down at $peer_i$. A third possibility is to regard backward-contracted clauses as new clauses, except that they are not re-allocated and they do not get an entirely new identifier (just a new "birth-time"). This mechanims allows to recognize, based on common identifier, different reduced forms of a common ancestor and delete all of them but one (see [2] for details of this mechanism).

5 Experiments

In the following we give the performances of Peers on a few problems. N-Peers is Peers with n nodes. The reported run time (in seconds) of n-Peers is the CPU time of the first Peer to succeed. The other Peers run till either they receive a halting message or also find a proof, whichever happens first. The nodes are workstations Sun Sparc 2, communicating over the Ethernet. The workstations used for our experiments were not isolated from the rest of the network and were possibly simultaneously used by other users.

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 $x3$ is proving that $x^3 = x$ implies commutativity in ring theory. Problem $r2$ is a problem in Robbins algebra. It consists in deriving Huntington's axiom $(-(-x + y) + (-(-x + (-y))) = x)$ from Robbins' axiom $(-(-(x + y) + (-(x + (-y)))) = x)$ and the hypothesis that there exists an element C such that $C + C = C$ [10]. Problems $sa1$ and $sa2$ are "single axioms problems" in group theory, where the goal is to prove that a given single axiom is sufficient to axiomatize group theory. All problems are solved by contraction-based strategies (Knuth-Bendix completion-based strategies), working modulo AC for $x3$ and $r2$. 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. The latter observation may be explained in part by redundancy, e.g. duplication of clauses among the processes, and the non-determinism of the distributed prover.

The following table shows the run-times for five runs on the problem $x3$. In each run a different strategy was selected. Strategies **a** and **c** use the *"rotate"* allocation algorithm, **b** and **d** use *"syntax"* and **e** uses *"select-min"*. Strategies **a** and **b** treat residents reduced by backward contraction as new clauses, while **c**, **d** and **e**, only update their birth-times:

Problem	*1-Peers*	*2-Peers*	*4-Peers*	*6-Peers*	*8-Peers*
x3a	96.28	53.58	46.87	54.04	25.95
x3b	96.45	50.29	43.28	30.66	7.51
x3c	96.06	51.37	44.06	43.52	28.06
x3d	95.86	49.16	44.52	31.65	8.60
x3e	96.36	87.64	38.34	24.93	31.02

Peers displays a considerable unstability of performances. This phenomenon, which appeared also in [5] and in radically different approaches to parallelization, e.g. [7], has not yet been studied systematically. We did not compare Peers with highly optimized sequential provers such as Otter, because Peers is a prototype, a tool to understand the problems in distributed deduction. We feel that Clause-Diffusion has the potential of achieving significant speed-ups on some non-trivial class of problems. Much more work is needed, on both the method and its implementation, in order to obtain more stable and more satisfactory performances.

Acknowledgements

Peers was written when the first author was visiting the Argonne National Laboratory in January/February 1993, supported by the Argonne National Laboratory and the Università degli Studi di Milano. The work on Peers continued when the first author was at INRIA-Lorraine and CRIN in the Spring of 1993, supported by INRIA-Lorraine and CRIN and the Università degli Studi di Milano.

References

1. 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.
2. M.P.Bonacina, Distributed Automated Deduction, Ph.D. Thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.
3. 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.
4. M.P.Bonacina and J.Hsiang, The Clause-Diffusion methodology for distributed deduction, submitted for publication.
5. 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.
6. R.Butler and E.L.Lusk, User's Guide to the p4 Programming System, Technical Report ANL-92/17, Argonne National Laboratory, Argonne, Illinois, October 1992.
7. 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.
8. W.W.McCune, OTTER 2.0 Users Guide, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, March 1990.
9. S.Taylor, *Parallel Logic Programming Techniques*, Prentice Hall.
10. L.Wos, Searching for Open Questions, Newsletter of the Association for Automated Reasoning, No. 15, May 1990.