Received by OCTI

FEB 11 1992

ROO—a Parallel Theorem Prover

by

Ewing L. Lusk, William W. McCune, and John K. Slaney

November 1991

**MATHEMATICS AND
COMPUTER SCIENCE
DIVISION**

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Reproduced from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62

Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL/MCS-TM--149

DE92 003232

ANL/MCS-TM-149

ROO—a Parallel Theorem Prover

by

*Ewing L. Lusk, William W. McCune, and John K. Slaney**

Mathematics and Computer Science Division
Technical Memorandum No. 149

November 1991

*Permanent address: Automated Reasoning Project, Australian National University, GPO Box 4,
Canberra 2601, Australia

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy
Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

MASTER
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Contents

Abstract	1
1 Introduction	1
2 The Parallel Closure Algorithm in an Automated Theorem-Proving Context	2
2.1 A Sequential Theorem-Proving Algorithm without Deletion	2
2.2 Roo without Deletion	3
2.3 Portable Implementation of Parallelism	5
2.4 Results	6
3 Roo with Deletion	9
3.1 OTTER's Full Sequential Algorithm	9
3.2 Results	10
3.3 Balancing Memory Usage	10
4 Conclusions and Future Work	11
References	11

ROO—a Parallel Theorem Prover

Ewing L. Lusk
William W. McCune
John K. Slaney

Abstract

We describe a parallel theorem prover based on the Argonne theorem-proving system OTTER. The parallel system, called ROO, runs on shared-memory multiprocessors such as the Sequent Symmetry. We explain the parallel algorithm used and give performance results that demonstrate near-linear speedups on large problems.

1 Introduction

In this paper we introduce a parallel theorem-proving system based on the Argonne family of theorem provers, and in particular on the well-known sequential system OTTER[5]. The parallel version is based on a parallel algorithm for computing the closure of a set under an operation, described in [7]. Theorem proving can be viewed as the computation of the closure of a set of clauses under a set of inference rules. In particular, we have applied the parallel closure algorithm to OTTER, currently the fastest sequential theorem-proving system for large problems. The result is ROO (Radical OTTER Optimization, with Australian origins), a parallel theorem prover compatible with OTTER and capable of near-linear speedup over OTTER for many problems on shared-memory multiprocessors.

OTTER's collection of theorem-proving tools includes both back subsumption and back demodulation, which are crucial for Knuth-Pendix completion runs. Each of these in turn involves deletion from the central database, an operation not contemplated in [7]. We describe here the problems engendered by deletion and our solution to these problems, which not only allows the back subsumption and back demodulation operations, but allows them to proceed in parallel with the other operations of ROO.

We present results on a variety of difficult theorem-proving problems, comparing ROO running on multiple processors with ROO on one processor and also with OTTER. We demonstrate some newly developed graphical tools to provide insight into the performance of ROO.

The paper is organized as follows. In Section 2 we treat the case without deletion. This is a straightforward application to OTTER of the parallel closure algorithm described in [7]. We present a simplified version of OTTER's theorem-proving algorithm and present the parallel closure algorithm in the theorem-proving context. This yields a simplified version of ROO. We then present some results obtained with this version.

In Section 3 we describe the problems introduced by deletion. We then present some results using the full-featured ROO.

In the final section we draw conclusions about ROO's effectiveness and discuss problems remaining in the area of parallel automated deduction.

2 The Parallel Closure Algorithm in an Automated Theorem-Proving Context

In this section we describe the application of the parallel closure algorithm given in [7] to automated theorem proving. Rather than repeat the fairly abstract presentation of [7], we specialize the presentation to the theorem-proving case.

2.1 A Sequential Theorem-Proving Algorithm without Deletion

We present here a simplified form of the theorem-proving algorithm that has been used in Argonne systems over the years. For an introduction to the approach, see [9]. We assume that the goal is to prove a set of clauses unsatisfiable. We assume further that the input clauses are divided into two sets, which we call the "axioms" and the "set of support," in such a way that the axioms are satisfiable. (If the input clauses make this difficult to do, then the axioms may be chosen to be the empty set.) Thus we use the set of support strategy first described in [8].

```

While (the null clause has not been produced and the set of support is not empty)
  Choose a clause from the set of support, call it the given clause, and move it to
    the axioms
  Generate all clauses that can be deduced from the given clauses and
    other clauses in the axiom set
  For each new generated clause
    Process it (rewrite to canonical form, merge literals, etc.)
    Test whether it is subsumed by any existing clause (in either axioms or s.o.s.)
    If the new clause survives
      Add it to the set of support
    end if
  end for
end while

```

Figure 1: Sequential Theorem-Proving Algorithm without Deletion (Algorithm 1)

The algorithm given in Figure 1 can complete with the empty clause found (giving a proof that the input clause set is unsatisfiable), complete without the empty clause being found (no proof), or fail to terminate within the constraints of time and memory. If the filtering mechanism applied is appropriately chosen, then this scheme is complete: that is, if the algorithm terminates without finding the empty clause, then the original set of clauses is satisfiable.

Currently, the most effective implementation of this algorithm is OTTER[5]. OTTER

has a wide variety of inference rules and control parameters for adapting this algorithm to a particular problem and sophisticated indexing methods to make each of the operations quite fast, even when the set of kept clauses has grown to one hundred thousand clauses or more.

This particular approach contrasts with some more recent “Prolog technology” theorem provers[6], which use the compilation techniques from WAM-based Prolog implementations to achieve extremely high inference rates, at a cost of possibly redundant computations. The “closure” approach taken here is based on the expectation that a clause deduced and processed is worth keeping as a filter for preventing the deduction and subsequent use of duplicate or weaker clauses. Experience is on the side of the closure approach; although certain small problems can be done very quickly with the Prolog-technology approach, many large problems done years ago with the closure approach remain out of reach of even the best Prolog-technology systems.

2.2 ROO without Deletion

In this section we present the parallel version of Algorithm 1 and its implementation based on OTTER.

An early attempt to parallelize Algorithm 1 focused on the inner “for” loop. This approach is relatively straightforward, since the rewriting, subsumption, and filtering of one new clause is independent from that of another, as long as one finishes up the loop by checking for subsumption among members of the batch. The problem with this approach is that even when there are large numbers of clauses in each batch, the barrier at the end of the loop means that many processes become temporarily idle.

The key idea here is to parallelize the outer “while” loop instead. That is, we consider multiple “given” clauses simultaneously. This approach both increases the grain size of the parallel computation and removes any barriers. The difficulty, of course, is that without some care, two copies of the same clause might enter the permanent clause space, each deduced and postprocessed by a different process. We also need to solve technical problems associated with adding clauses to the clause space while it is in use by other clauses.

Our approach is to use an intermediate holding area, which we call (arbitrarily) K . New clauses are first put in K , from which they are removed by a single process which repeats the postprocess of the clause before adding it to the clause database. Thus we break down the work to be done into two tasks: A and B (See Figures 2–4). At any moment multiple processes are executing task A, but at most one process is executing task B. Each instance of Task A is associated with a given clause. Task B is executed only when the set K is non-empty.

Note that there is not a separate process dedicated to Task B. Rather, we create a uniform pool of processes, each of which performs the loop shown in Figure 5.

The loop continues until some process detects unit conflict or until the set of support is empty and all processes are waiting for a clause to appear there.

Figure 2: Flow of Data in Simplified ROO

Task A(given clause):
 generate new clauses
 For each new clause
 rewrite it
 subsumption test
 filter
 If it survives
 lock K
 put new clause in K
 unlock K
 end if
 end for

Figure 3: Task A

Task B:
 While K is not empty
 lock K
 choose a clause from K
 unlock K
 redo rewrite, subsumption test, filter
 If it survives
 integrate new clause into database
 put new clause in K
 end if
 end while

Figure 4: Task B

```

While it is not time to stop
  If K is nonempty and no process is already doing it
    do Task B
  else
    If the Set of support is not empty
      choose a new given clause from set of support
      do Task A(given clause)
    end if
  end if
end while

```

Figure 5: Main Loop Performed by All Processes

2.3 Portable Implementation of Parallelism

The parallel algorithm described here does depend on a shared-memory model of computation, since multiple instances of Task A, together with at most one instance of Task B will be accessing the data structures of the permanent database simultaneously, as well as the list K of intermediate clauses. On the other hand, there is no reason to specify the program for a particular shared-memory multiprocessor.

To achieve both portability and ease of implementation, we use the approach described in [1] and the **p4** system[2], which is the current version of the system described in [1]. **p4** provides a set of C macros and subroutines not only for the portable implementation of low-level primitives such as locks, but also for the structured use of locks through monitors, and it includes a library of useful monitors. One particular **p4** monitor, the **askfor** monitor, is in fact a dispatcher. It implements the model of a pool of processes operating on a shared pool of work units, which is ROO's basic design. The **askfor** monitor is the core of the main loop shown in Figure 5. In our case the work units are of two types:

- generate resolvents or paramodulants for a given clause, demodulate them, and perform initial subsumption test (Task A);
- empty list K by performing final rewriting and subsumption test (Task B).

Task B has priority; whenever K is nonempty, the first process to ask for work is assigned Task B.

The **askfor** monitor as supplied in the **p4** package has facilities for delaying processes when the pool of work becomes empty and stopping all processes when one of them signals that a proof has been found.

Since ROO is implemented in terms of **p4**, it is portable among shared-memory multiprocessors on which **p4** is implemented. Currently these include the Sequent Symmetry and Balance, the Encore Multimax, and the BBN GP1000 and TC2000.

2.4 Results

In this section we discuss the workings of ROO without deletion and present some experimental results.

At first glance one's intuition might suggest that the ability of ROO to deliver the expected performance would be hobbled by the restriction that only one process can be carrying out Task B at a time. If Task B falls behind (the K list grows continually and is never emptied), there will be two bad effects. The first is that the use of an important generated clause may be delayed by a long residence in the K list, thus postponing the proof. The second is that, particularly for large numbers of processes, Task B may not be able to keep the pool of work units filled by adding new, fully processed clauses to the set of support. In this situation, idle processes will await work in the **askfor** monitor, wasting potential processing power.

In practice, however, this situation does not arise as often as one might think. To understand just what does happen as ROO proceeds, it is useful to study its behavior with a general-purpose tool developed at Argonne called **upshot**[3].

Figure 6: Beginning of Sam's Lemma

In Figure 6 we see the situation at the beginning of the run of Sam's Lemma on ROO with sixteen processes on a Sequent Symmetry. But later in the run (3/4 sec. later) the situation has changed, and the rule that only one process can do B at a time is no longer a

restriction. All processes are busy. This situation is seen in Figure 7.

Figure 7: Later in Proof of Sam's Lemma

What has happened is that although the instances of A are generating new candidate clauses, after a while the existing clause space has expanded to the point where most new clauses are subsumed by Task A and never make it into the K list. When K does become nonempty, it quickly emptied by the next process to become available.

We now look at specific speedup results. Table 1 is for Sam's Lemma, the problem shown in Figure 6. This has been a standard theorem-proving benchmark for many years. This is not a large problem, and so speedups begin to level out at around 8 processors. Nevertheless, it is useful for demonstrating that ROO is at least continuing to reduce total execution time even at 24 processes. Note also that ROO with one process is not appreciably slower than OTTER.

Table 2 shows the IMP-4 problem from Lukaciewicz logic, one of the most difficult problems that OTTER can do, despite the small set of input clauses. ROO reduces the execution time from more than 7 1/2 hours to just 20 minutes.

Table 3 refers to a problem that uses ROO to compute the size of the semigroup F2B2, whose size was unknown until it was found by OTTER. A smaller version of this problem is described in [4]. Here, instead of proving that a set of clauses is unsatisfiable, we compute the entire logical closure (the entire semigroup).

	OTTER	Roo1	Roo4	Roo8	Roo16	Roo24
seconds	26.5(1.02)	27.1(1.0)	7.5(3.6)	4.1(6.6)	2.6(10.4)	2.1(12.9)
generated	5924	5981	6139	6077	6199	6278
gen/sec	224(1.02)	220(1.0)	819(3.7)	1482(6.7)	2384(10.8)	2990(13.5)
kept	134	132	131	131	130	131
memory	96K	128K	192K	319K	575K	830K

Table 1: Sam's Lemma

	OTTER	Roo1	Roo4	Roo8	Roo16	Roo24
seconds	27303(1.1)	30458(1.0)	7398(4.1)	3541(8.6)	1876(16.2)	1199(25.4)
generated	6706380	6668046	6369094	6142292	6352314	6251041
gen/sec	246(1.1)	218(1.0)	860(3.9)	1734(7.9)	3386(15.5)	5213(23.9)
kept	20410	20342	20323	20571	18951	15849
memory	7185K	7664K	8590K	10250K	12965K	12039K

Table 2: The IMP-4 Problem of Lukaciewicz Logic

	Roo1	Roo8	Roo16	Roo24
seconds	1120.38	143.94	71.94	50.57
generated	8610	8610	8610	8610
kept	1429	1429	1429	1429
memory (K)	6259	8334	11464	16541
speedups:				
“proof”	1.0	7.8	15.6	22.2

Table 3: Determining the size of F3B2

3 ROO with Deletion

3.1 OTTER's Full Sequential Algorithm

In Section 2 we described a simplified version of OTTER's algorithm, in which no clauses are deleted from the database once they have been added. In the complete version, back subsumption tests are done on newly derived clauses, causing deletions, and new rewrite rules may be derived in the course of the run. These new rewrite rules are immediately applied to existing clauses in the database, causing both deletions and new additions. Coping with deletions complicates ROO since we do not want to interfere with the generation of new clauses when deleting others. It turns out that this problem can be solved by having Task B handle actual deletions. Backward subsumption and backward demodulation processes can run in parallel with all other processes; but instead of actually deleting clauses from the database, they place their identifiers in shared lists where Task B (executed by only one process at a time) can find them and carry out the actual deletions. This algorithm is shown schematically in Figure 3.1.

Figure 8: Flow of Data in ROO

3.2 Results

In Table 4 we exhibit the behavior of the full ROO, with deletion, attacking another difficult problem in Lukaciewicz logic. This is an equality formulation of the problem, in which the inference rule used is paramodulation, and both forward and backward demodulation are required, as well as backward subsumption.

	otter	roo8	roo16	roo24
seconds	3953.73	608.74	316.40	204.87
generated	538331	614588	637223	606136
kept	2493	2484	2419	2414
memory (K)	2554	3353	6003	10378
gen/sec	136	1009	2016	2957
speedups:				
proof	1.0	6.5	12.5	19.4
gen/sec	1.0	7.4	14.8	21.7

Table 4: Equality Formulation of Luka5

3.3 Balancing Memory Usage

Because ROO is written in C, there is no automatic deallocation of unused nodes (garbage collection) as in some implementations of higher-level languages such as Prolog or Lisp. As programmers of ROO, we are responsible for allocating and deallocating nodes used in the construction of clauses and indices. Lists of available nodes are maintained for the various types of node, and “get-node” and “free-node” routines are used by the programmer. If get-node finds that the list is empty, it allocates memory from the operating system.

Since getting and freeing nodes are two of the most frequently used operations, we cannot allow locking of global lists to prevent more than one process from simultaneously updating the lists. Therefore, each process maintains its own lists of available nodes. The nodes are in shared memory, however, and data built with those nodes is passed between various processes. The following problem arises. Suppose process P_a builds more than its share of clauses, and process P_b frees more than its share of clauses. Then P_a must keep allocating memory from the operating system, while P_b ’s lists of available nodes grow.

Our preliminary experiments with ROO showed that this was a severe problem when one of the processes spent most of its time executing Task B: when freeing clauses, that process would collect all of the nodes allocated by the other processes while building inferred clauses in Task A. Our solution is (1) to have Task A stamp all inferred clauses with the ID of the process executing Task A (with the “owner” of the clause); (2) to have Task B, rather than deallocate a clause, place it in an area accessible to the owner of the clause; and (3) to have a process reclaim and deallocate its clauses when it is convenient to do so (for example, at the beginning of Task A). If ROO applications arise in which memory balance is a problem when managing nodes used in the construction of indices, then the same technique can be used there as well.

A second solution is to have each process monitor the sizes of its lists of available nodes and to have it give up any large lists to a global area. We have not experimented with this second solution.

4 Conclusions and Future Work

We have shown how to apply the parallel closure algorithm to OTTER so that it executes effectively in parallel on shared-memory multiprocessors. ROO is completely compatible with OTTER in terms of parameter settings and input formats and is now the very fastest way to attack large problems with automated theorem provers. Our next project is to develop a distributed-memory version of the algorithm in order to take advantage of new machines with very large numbers of processors.

References

- [1] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.
- [2] R. Butler and E. Lusk. Unpublished information, 1991.
- [3] V. Herrarte and E. Lusk. Studying parallel program behavior with *upshot*. Technical Report ANL-91/15, Argonne National Laboratory, Argonne, Illinois, 1991.
- [4] E. Lusk and R. McFadden. Using automated reasoning tools: A Study of the semigroup f2b2. *Semigroup Forum*, 36(1):75–88, 1987.
- [5] William W. McCune. OTTER 2.0 user's guide. Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, 1990.
- [6] J. Schumann and R. Letz. Partheo: A high-performance parallel theorem prover. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 449*, pages 40–56. Springer-Verlag, 1990.
- [7] John K. Slaney and Ewing L. Lusk. Parallelizing the closure operation in automated deduction. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 449*, pages 28–39. Springer-Verlag, 1990.
- [8] L. Wos, D. Carson, and G. Robinson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 14:536–541, 1965.
- [9] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

END

**DATE
FILMED
3 / 10 / 92**