Parallelizing the Closure Computation in Automated Deduction

John K. Slaney
Automated Reasoning Project
Australian National University
jks@arp.anu.oz.au

Ewing L. Lusk *
Mathematics and Computer Science Division
Argonne National Laboratory
lusk@mcs.anl.gov

Abstract

In this paper we present a parallel algorithm for computing the closure of a set under an operation. This particular type of computation appears in a variety of disguises, and has been used in automated theorem proving, abstract algebra, and formal logic. The algorithm we give here is particularly suited for shared-memory parallel computers, where it makes possible economies of space. Implementations of the algorithm in two application contexts are described and experimental results given.

1 Introduction

There are many contexts in which one wants to compute the closure of a set of objects under an operation. In resolution-based theorem proving, for example, the search for a proof of unsatisfiability for a set of clauses may be viewed as a computation of the closure of the original set of clauses under a resolution operation, where one terminates the computation once the closure has been found to contain the null clause. The Knuth-Bendix completion procedure is a closure computation in which the operation is a restricted form of paramodulation followed by demodulation, and one may not know in advance whether the computation will complete. The closure computation is also strongly related to the

*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.

computation of least fixed points. Details of these and other occurrences of the closure computation paradigm are given in Section 2 below.

Algorithms for carrying out this computation at first glance appear serial, because the results of earlier computations are used in later ones. A small amount of reflection reveals substantial parallelism in the computation of new elements, however, except in pathological cases. Further study reveals a serialization bottleneck in the subsumption phase of the algorithm. The main point of this paper is the presentation of a parallel algorithm for the closure computation that avoids this bottleneck in a way different from other attempts to solve this problem. In Section 3 we describe the new algorithm, prove its correctness, and describe how it differs from other algorithms designed to attack the same problem. Finally, in Section 4 we describe the implementation of this algorithm in two different programs to carry out computations in formal logic. We give results demonstrating that significant speedups can be achieved. Finally, we describe directions for future exploitation of the algorithm in other contexts.

2 The Closure Computation and its Significance

The closure operation is a fundamental one. Suppose we are given a set T and an operation $f : T \times T \cdots \times T \rightarrow T$. Then if S is a subset of T, we define the *closure* $Cl(S,f)$ as:

$$Cl(S,f) = \bigcap \{ R \mid S \subset R \text{ and } f(R \times R \cdots \times R) \subset R \}$$

In other words, the closure of S is the smallest subset of T that contains S and everything in T that can be obtained from S by repeatedly applying the operation f. Depending on the choice of T, S, and f, computing $Cl(S,f)$ is equivalent to a variety of computations which on the surface appear to be quite different. We give here several examples.

2.1 Unsatisfiability of a Set of Clauses

One of the most familiar applications is that of proving a set of clauses unsatisfiable. In this case T is the set of all clauses, S is the set we wish to show unsatisfiable, and f is an inference rule such as binary resolution or paramodulation. (Hyperresolution gives an example in which the arity of f is greater than two.)

In this case, $Cl(S,f)$ is quite likely to be infinite, and so one expects to terminate the computation as soon as the empty clause $\square$ is found to be a member of $Cl(S,f)$. It is the potentially infinite size of $Cl(S,f)$ that makes resolution only a semi-decision procedure for first-order logic. Whether or not $Cl(S,f)$ is finite, it represents the set of all clauses deducible from S under the inference rules being used.

2.2 The Knuth-Bendix Completion Procedure

The computation of a complete set of rewrite rules is similar, except in this case the computation will not terminate if the closure being computed does not turn out to be finite. One starts with an initial set S of rewrite rules, and tries to find a set P of rewrite rules containing S and having the unique termination property. Knuth and Bendix showed that $P = Cl(S, f)$, where f is defined as paramodulation between left sides of rewrite rules, followed by rewriting of the resulting paramodulant. (To compute a practical complete set, one also applies back subsumption and back demodulation to the set being constructed, so that the set being computed is not exactly a closure of S, since it does not contain S itself. This aspect of the "closure" computation will be treated in a later paper.)

2.3 Computing the Size of Certain Finite Free Semigroups

Algebraic objects are frequently defined in terms of generators, operations, and relations. It may be obvious that the resulting object is finite, but there may be no clue to its size, even in the case of a free object (no relations). In this case the object is exactly the closure of the set of generators under the operation, and so the closure computation can be used to compute the size of the free object.

The fact that this is a closure computation means that a theorem prover can be used to carry it out. This is exactly what was done in [6] to discover the size of the semigroup $F2B2$. It has been used since then to discover the sizes of $F3B2$, $F2B21$, and $F3B21$. All of these were new results. The computation of $F3B21$ was done on a parallel machine because it had the required 120M of memory. Parallelism was not exploited; the run took more than seven hours on a Sequent Symmetry.

2.4 Computation of Ackerman Constants

One case in which it was *not* obvious whether a particular freely generated algebra was finite or infinite was that of the free De Morgan monoid generated by the monoid identity. De Morgan monoids are the natural algebraic counterparts of the system R of relevant logic (see [3] for a definition and explanation). An important step towards understanding constant De Morgan monoids—i.e. those generated from the identity alone—was the computation of the subdirect products thus generated of certain sets of small constant algebras. As reported in [9] and [11], one of these was eventually discovered to be characteristic and to have 3088 elements. The closure computation used to discover this was painfully slow in its serial form but parallelizes well by our method as shown below.

2.5 Formulas of $E_{\rightarrow}$

Another, related, closure problem arising in the theory of relevant logic is that of determining the structure of the one-variable fragment of the sentential logic $E_{\rightarrow}$ of Anderson and Belnap. It is known that with negation as well as implication infinitely many non-equivalent formulae can be defined in E, but very little is known about the pure implication fragment. A recent computational result is that the direct product of all 4-element $E_{\rightarrow}$ algebras can distinguish 7516 such formulae. In Section 4 we present speedup results for the parallelization of this computation.

3 Algorithms for the Closure Computation

3.1 The Sequential Algorithm

Here we describe a straightforward, efficient algorithm for computing the closure. It has been used in countless theorem provers.

Let $S = \{x_1, \ldots, x_k\}$ be a finite subset of a set T. Let $f : T \times \cdots \times T \rightarrow T$ be an n-ary operation on T. The algorithm shown in Figure 1 makes use of a list L and two indices into L, p and q. We say that an element x of T is *new* if it is not subsumed by an element of L.

```
Initialize $L[i] \leftarrow x_i$, for $i = 1, \ldots, k$
Initialize $p \leftarrow 1$, $q \leftarrow k + 1$
While $p < q$ do
    Foreach $(i_1, \ldots, i_n)$ such that $\forall j,\ i_j \leq p$ and $\exists j,\ i_j = p$
        Compute $f(L[i_1], \ldots, L[i_n])$            (generate phase)
        If $f(L[i_1], \ldots, L[i_n])$ is new then      (subsumption phase)
            $L[q] \leftarrow f(L[i_1], \ldots, L[i_n])$         (update phase)
            $q \leftarrow q + 1$
        end if
    end for
    $p \leftarrow p + 1$
end while
```

Figure 1: Sequential Closure Algorithm

There are several observations to be made about this algorithm:

- The pointer p chases the pointer q; the computation completes when it catches up. There is no guarantee that it will do so. This is not a severe drawback in most situations, since either 1) the closure is known in advance to be finite, or 2) one is looking for a particular element of the closure, and so will abort the search when it is found.
- If the algorithm does terminate, then at completion L is an ordering of $Cl(S, f)$. By induction on the number of times through the "While" loop, the generate phase can only compute elements of $Cl(S, f)$, and as long as L is not all of $Cl(S, f)$, the

computation will continue, since all tuples of elements of L are eventually considered in the "Foreach" statement. Finally, the subsumption phase ensures that L contains no duplicates.

- The restriction in the "Foreach" condition ensures that each tuple will only be considered once. This is critical for efficiency, since it prevents redundant calculations. Each time through the main loop, we consider only those elements of L up to and including the pth one, not all of the ones we have produced.
- Anyone reading this as a theorem-proving algorithm would ask about back subsumption and back demodulation, which use new elements of L to modify or delete existing ones. Note that these operations are not admissible here by our definition of closure.

3.2 Variations on the Sequential Algorithm

We take this opportunity to make some observations about the closure algorithm as it stands. These observations will also be true of the parallel version.

The set-of-support strategy In the context of theorem proving, it represents an implementation of the set-of-support strategy[12] used in various Argonne theorem provers over the years. In many cases it is possible to initialize with the pointer p set to an index greater than 1. This is typically done when one knows that S is an unsatisfiable set of clauses but that a subset A of S is satisfiable. Then if we assign the elements of A to $L[1], \ldots, L[n]$ and start with $p \leftarrow n+1$, under certain conditions on the inference rules being used we will still be computing that part of $Cl(S, < inference\ rules >)$ that contains the empty clause, although not the complete closure. The set A typically contains the axioms for a domain, together with parts of the hypotheses of the theorem being proved.

Associative operations Further optimizations are possible when the function f is a binary operation and is *associative*. In this case the "Foreach" condition can use the stronger restriction that $i_1 \leq k$. That is, in the generate step, the first element of the tuple being considered must belong to the original input set S. This greatly reduces the number of tuples (pairs, in this case) considered. This is illustrated in the case of the semigroup problem, where the operation is associative by the definition of semigroup. If we let $x_1, \ldots, x_k$ be the original set of generators, then a general product computed by the algorithm is of the form $(y_1 \cdot \ldots \cdot y_r)(y_{r+1} \cdot \ldots \cdot y_s)$, where $\forall j, \exists i$ with $y_j = x_i$. Since the operation is associative, this same product will also be computed as $y_1 \cdot (y_2 \cdot \ldots \cdot y_s)$. Hence we may safely restrict ourselves to the case in which the first element of the pair is one of the original generators.

Another, more general, case arises when one is computing the closure of any relation under the "composition" operation, which is associative. An example many are familiar with is the ancestor relation. There are two ways to axiomatize the ancestor relation generated from the parent relation:

(1) If parent(x,y) then ancestor(x,y)
If ancestor(x,y) and ancestor(y,z) then ancestor(x,z)

(2) If parent(x,y) then ancestor(x,y)
If parent(x,y) and ancestor(y,z) then ancestor(x,z)

Axiomatization (1) may compute a specific ancestor relationship in fewer steps, but (2) will compute all ancestors (the closure) much more efficiently. The optimization is possible precisely because this is the computation of the composition operation on ordered pairs, which is associative.

An even further optimization is possible when f is a binary operation which is commutative. In that case we can impose the additional restriction in the "Foreach" condition that $i_1 \leq i_2$.

The infinite case The closure computation can be generalized even more than we have stated. What it turns on is the very elementary fact of recursion theory that the closure of a recursively enumerable set under a recursive function is recursively enumerable. It is not necessary that the set of generators be finite; we might be given not a list of them but just an algorithm for generating them. Nor is it necessary that the closure computation should complete. A completely general form of the algorithm is given in Figure 2.

```
Initialize $L[1] \leftarrow x_1$
Initialize $p \leftarrow 1, q \leftarrow 2, k \leftarrow 2$
While $p < q$ do
    Foreach $(i_1, \ldots, i_n)$ such that $\forall j,\ i_j \leq p$ and $\exists j,\ i_j = p$
        Compute $f(L[i_1], \ldots, L[i_n])$
        If $f(L[i_1], \ldots, L[i_n])$ is new then
            $L[q] \leftarrow f(L[i_1], \ldots, L[i_n])$
            $q \leftarrow q + 1$
        end if
    end for
    If $x_k$ exists then
        $L[q] \leftarrow x_k$
        $k \leftarrow k + 1, q \leftarrow q + 1$
    end if
    $p \leftarrow p + 1$
end while
```

Figure 2: Generalized Sequential Closure Algorithm

What we now claim is that (regardless of whether S is finite or not) L is an enumeration of the closure of S under f. In the case that S is finite we can indeed simplify by putting all the x_i in at the start, as in Figure 1, but this is not generally possible.

3.3 The Parallel Algorithm

The most obvious way to parallelize the algorithm is to parallelize one of the loops. The inner loop is relatively easy to parallelize but leads to a sequential bottleneck because processes must wait for each other to complete and synchronize for the advancement of the pointer p.

A better technique is to parallelize the outer "while" loop; that is, to consider several values of p at once. The generate phase of the computation is not affected, since for $l = p, p+1, \ldots, p+r < q$, the lth iteration of the loop is exactly the same, whether the iterations are done sequentially or in parallel. The subsumption and update phases of the computation, however, must be changed, to protect against simultaneous updating of L and, more importantly, to ensure that a complete subsumption test is carried out before updating. A straightforward way to do this would be for each process to lock L while it performed the subsumption check and added newly generated unique elements to L. However, this would create too much of a serial bottleneck, since the subsumption phase of many closure computations is the most expensive. Our solution separates the tasks to be carried out in order to allow for greater parallelism.

To accomplish this, we make use of a separate list K indexed by a pointer r, which holds elements that have been generated and passed an "almost complete" subsumption test, but have not passed a final subsumption test. Carrying out the final check and moving such elements from list K to list L is done separately. More formally, we break the computation down into units of work as shown in Figure 3. It is possible to design the data structures that index L for fast partial subsumption testing so that the generate phase of Task A can go on concurrently with Task B.

Finally, we dispatch processes that are available for work as follows: When a process requests work, we give it Task B if K is not empty and no other process is executing Task B, otherwise we assign it Task A. Thus we typically have many instances of A in progress, and (occasionally) at most one instance of Task B.

There are several observations to be made about this algorithm:

- It also computes the closure of S under f, by the same argument as in the sequential case.
- No duplicates are put in L, because updating of L is only done by Task B, of which there is only one instance at a time.
- The algorithm retains most of the efficiency of the sequential algorithm, but not all. It remains true that no tuples are considered twice, since each instance of Task A uses a different value of s, so the total amount of work done in the generate phase is the same in both cases. The final subsumption test will duplicate work done in the partial subsumption test. Some of this duplication is reduced by the removal of

```
    Initialize $L[i] \leftarrow x_i$, for $i = 1, \ldots, k$
    Initialize $p \leftarrow 1$, $q \leftarrow k+1$
    Initialize $K \leftarrow empty, r \leftarrow 1$

Task A:
    Lock $p$, set $s \leftarrow p$, $p \leftarrow p+1$, Unlock $p$
    Foreach $(i_1, \ldots, i_n)$ such that $\forall j,\ i_j \leq s$ and $\exists j,\ i_j = s$
        Compute $f(L[i_1], \ldots, L[i_n])$              (generate phase)
        If $f(L[i_1], \ldots, L[i_n])$ is new then       (partial subsumption phase)
            Lock $K$
            $K[r] \leftarrow f(L[i_1], \ldots, L[i_n])$  (partial update phase)
            $r \leftarrow r+1$
            Unlock $K$
        end if
    end for

Task B:
    While $r > 0$
        Lock $K$
        $a \leftarrow K[r], r \leftarrow r-1$
        Unlock $K$
        If $a$ is new then                               (complete subsumption)
            $L[q] \leftarrow a$                          (complete update)
            $q \leftarrow q+1$
        end if
    end while
```

Figure 3: Parallel Closure Algorithm

duplicates from K, but some of it is unavoidable. The reason this is not a severe problem is that most subsumed elements are removed by the partial subsumption check, and thus are not retested in Task B.

- The parallelism is of large grain size, since Task A contains not only generation of new elements but also the bulk of the work of subsumption. The grain size can be increased slightly (synchronization reduced) by batching within each process the partial updates to K. We have not shown this in Figure 3 since it complicates the presentation of the algorithm.

- Nearly all attempts to generate a new element fail. If the computation completes with $q = Q + 1$, then Q^n tuples will be considered, but only $Q - k$ new elements will be added. Assuming k negligibly small with respect to Q, we still have only approximately 1 in every Q computations resulting in updates to L. Therefore Q need only be on the order of 100 for the computation to be well worth parallelizing.

  By the computation above the proportion of trials which actually result in any updating of L is rather small. The proportion which result in updating K is usually larger. The question of how much larger is crucial to any account of the efficiency of the parallel algorithm. The serial bottleneck is avoided only if list K is at least close to being emptied for the last time when the computation completes. In the more general case of a closure computation which does not complete, or which is terminated for reasons other than exhaustion, K should grow slowly in comparison with L. Unfortunately, we have no firm theoretical results concerning the size of K, but our experimental results indicate that in typical cases K remains small. Most of the time, in fact, K is empty, and because elements from K do not have to be reconstructed before their second comparison with L it can be emptied again fairly quickly when it does become occupied. We present our speedup results as evidence of the degree to which the serialization problem has been overcome. Although precise results are hard to come by, we may at least note that in general the greater the final value of q the better the algorithm will parallelize, while the more processes there are adding to list K the greater will be the threat of a recurrence of serialization.

- Other parallel algorithms for parallelizing this type of computation have been presented, both of them based on message-passing rather than shared-memory models of parallel computation. The difference is that synchronization among processes is done by the sending and receiving of messages rather than by locks on shared data structures, which are an inherently faster synchronization mechanism. One implementation of the semigroup computation was given in [1]. There speedups of close to 3 were obtained with 4 processes. The message-passing model limited the number of processes that were runnable because extra memory was required for duplicating most of the data structures.

  A more sophisticated message-passing theorem prover is presented in [4]. The technique used there provided speedups up to 23 with 29 processes on a very large problem, but requires complex load-balancing decisions to be made ahead of time, and seems to require a large problem to outweigh the overhead of message-passing. The results we present in Section 4 lead us to hope that when the shared-memory algorithm of Figure 3 is applied to a theorem prover, speedups will be good even on small problems, and the load-balancing will be automatic, because of the shared-memory model.

4 Experimental Results

This algorithm has first been tested in the application areas described in Sections 2.4 and 2.5. The implementation was in C, using the Argonne monitor macro package described in [5] to provide a portable and relatively high-level paradigm for writing the program. The package provides a general dispatcher (the ASKFOR monitor) which was used to dispatch tasks A and (one instance at a time of) task B to however many processes were started. No particular process was dedicated to any particular task. The code was developed on a Sequent Symmetry at the Automated Reasoning Project at the Australian National University, and run on a larger Symmetry at Argonne National Laboratory. Results obtained are shown in Tables 1 and 2.

processes	time(seconds)	speedup
1	1964.43	1.00
5	408.31	4.81
10	222.74	8.82
15	163.65	12.00
20	129.85	15.13

Table 1: Computation of Ackerman Constants

processes	time(seconds)	speedup
1	13597.26	1.00
5	2913.37	4.67
10	1612.07	8.43
15	1185.83	11.47
20	972.39	13.98

Table 2: Computation of Formulas of $E_{\rightarrow}$

In each of these cases, an order of magnitude reduction in computing time was achieved.

5 Future Work

The algorithm has so far been implemented and tested in special purpose programs for computing the closures of particular sets under particular operations. The next step is to implement a general-purpose theorem prover using this algorithm, incorporating back demodulation and back subsumption. Not only will this provide a wider range of applicability but we will be able to compare the speed of the parallel closure computation using general methods with the special-purpose programs described here and with the message-based parallel closure algorithms. We have begun to construct such a theorem prover, using OTTER[7] as a base, since it already contains many desirable features unrelated to parallelism. The goal is to produce a parallel version of OTTER which behaves identically with regard to input and output, but is capable of greatly increased performance on parallel machines.

References

[1] Butler, R. M., and N. T. Karonis, "Exploitation of Parallelism in Prototypical Deduction Problems", *Proceedings of the 9th International Conference on Automated Deduction*, E. L. Lusk and R. A. Overbeek, (Eds.) Springer-Verlag Lecture Notes in Computer Science #310 (1987), pp. 333–343.

[2] Dunn, J. M., The Algebra of Intensional Logics, doctoral dissertation (unpublished), University of Pittsburg, 1966.

[3] Dunn, J. M., "Relevance Logic and Entailment", Gabbay, D, & Günthner, F (eds), *Handbook of Philosophical Logic, Vol. 3*, Dordrecht, Reidel, 1986.

[4] Jindal, A., R. Overbeek, and W. Kabat, "Exploitation of Parallel Processing for Implementing High-Performance Deduction Systems", *Journal of Automated Reasoning*, to appear.

[5] Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R., *Portable Programs for Parallel Processors*, New York, Holt, Rinehart & Winston, 1987.

[6] Lusk, E. and McFadden, R., "Using Automated Reasoning Tools: A Study of the Semigroup F2B2", *Semigroup Forum* **36**, no. 1 (1987), pp. 75–88.

[7] McCune, W. W., "OTTER 1.0 Users' Guide", Report ANL–88–44, Argonne National Laboratory, Argonne, Illinois.

[8] Meyer, R. K., "Sentential Constants in R and $R^{\neg}$", *Studia Logica* 45 (1986), pp. 301–327.

[9] Slaney, J. K., "3088 Varieties: A Solution to the Ackermann Constant Problem", *Journal of Symbolic Logic* 50 (1984), pp. 487–501.

[10] Slaney, J. K., "On the Structure of DeMorgan Monoids, With Corollaries on Relevant Logic and Theories", *Notre Dame Journal of Formal Logic* 30 (1989), pp. 117–129.

[11] Slaney, J. K., "The Ackermann Constant Theorem: A Computer-Assisted Investigation", *Journal of Automated Reasoning*, forthcoming.

[12] Wos, L., D. Carson, and G. Robinson, "Efficiency and completeness of the set-of-support strategy in theorem proving," *Journal of the ACM* **14**, (1965), pp. 536–541.