Distributed Cube and Conquer with Paracooba

Maximilian Heisinger, Mathias Fleury , and Armin Biere

Johannes Kepler University Linz, Austria
{maximilian.heisinger,mathias.fleury,armin.biere}@jku.at

Abstract. Cube and conquer is currently the most effective approach to solve hard combinatorial problems in parallel. It organizes the search in two phases. First, a look-ahead solver splits the problem into many sub-problems, called cubes, which are then solved in parallel by incremental CDCL solvers. In this tool paper we present the first fully integrated and automatic distributed cube-and-conquer solver Paracooba targeting cluster and cloud computing. Previous work was limited to multi-core parallelism or relied on manual orchestration of the solving process. Our approach uses one master per problem to initialize the solving process and automatically discovers and releases compute nodes through elastic resource usage. Multiple problems can be solved in parallel on shared compute nodes, controlled by a custom peer-to-peer based load-balancing protocol. Experiments show the scalability of our approach.

1 Introduction

SAT solvers have been successfully applied in many practical domains, including cryptanalysis, hardware and software verification but also with increasing interest have been used to solve hard mathematical problems [17,21,26]. Sequential state-of-the-art SAT solving combines the well-known conflict-driven-clause-learning procedure (CDCL) [33,34] with sophisticated preprocessing techniques [10,23] and other efficient heuristics for variable selection [6,28,30], restarts [2,7,32], and clause database reduction [32]. While some authors argue that there was "no major performance breakthrough in close to two decades" [29], at the same time computers have become more and more powerful thanks to the ubiquitous availability of multi-core processors and the increasing usage of computers in the cloud. Thus improving the efficiency of parallel SAT solving remains an important topic. Accordingly, beside the traditional parallel track, the SAT Competition 2020 [19] features for the first time also a cloud track.

One approach to solve large problems in parallel consists in splitting the problem into smaller, more manageable instances, for example, using cube and conquer [16,20]. All these sub-problems are subsequently solved independently in parallel. This method was used by Heule to settle some long-standing mathematical conjectures [17,21]. Splitting the problems was done automatically by a tool, but then required to manually distribute instances for parallel solving.

In this paper, we present Paracooba [15]. After splitting a problem with the look-ahead solver March, Paracooba transfers the sub-problems in an

efficient way to many nodes (including over network). It detects when new instances become online and balances the work across all available nodes.

Other attempts for automatic and efficient distribution of problems exist, but use divide and conquer: Problems are dynamically split when nodes are underused (Sect. 2). In contrast, PARACOOBA assumes the problem is already split. Each node runs at least one instance of the SAT solver CADICAL [5]. The sub-problems are solved incrementally to reuse information from the previous solving. PARACOOBA relies on a custom protocol to automatically detect nodes that are underused and balances work across all nodes, including newly joining ones. It also supports disconnecting nodes by rebalancing the jobs (Sect. 3).

In the experiments, we focus on a single CNF `cruxmiter`, a miter for 32-bit adder trees [25], which is considered a challenge for resolution-based solvers and exemplary for the difficulties that arise in the verification of arithmetic circuits (see also [24]). Such benchmarks were also used in the SAT Race 2019. Already in the original work on cube and conquer similar multiplier equivalence checking problems were shown to benefit from the cube-and-conquer approach. Our results in Sect. 4 show that we get linear scaling with respect to the number of threads.

2   Preliminaries and Related Work

We use standard notations and refer the reader to the *Handbook of Satisfiability* for an introduction to SAT [8] as well as to the chapter on parallel SAT solving [4] in the *Handbook of Parallel Constraint Reasoning* [14].

One idea to improve solving of large instances is to distribute the work across different machines, via either a *diversification* of the search or *splitting* of the search space. In the first approach, several solvers are used as portfolio. By changing some parameters used by SAT solvers, they heuristically search on different parts of the search space and share some of the clauses they learned. ManySAT [13] pioneered the approach, which is now used in various tools like CRYPTOMINISAT [35], HORDESAT [3], PLINGELING [5], and SYRUP [1]. As soon as any instance derives SAT or UNSAT, then the problem is solved.

We use another approach that divides the search space explicitly as pioneered in [9,22,36] and refined in [16,20]. Solving the formula φ is equivalent to splitting it into the two formulas $\varphi \wedge x$ and $\varphi \wedge \neg x$ and solving them. Unlike diversification, the overall problem is only considered to be UNSAT if all sub-problems are. Still, if any sub-problem is SAT, the overall problem is SAT, too. Splitting can be done dynamically during solving whenever a problem is deemed too hard. This is used for instance by PAINLESS [27] or MAPLEAMPHAROS [31]. These tools also share clauses to get some of the benefits of portfolio solvers.

Splitting can also be done upfront by look-ahead. By splitting the formula recursively, we obtain a formula of the form $\varphi \wedge c_1, ..., \varphi \wedge c_n$ where the conjunctions c_i are called *cubes*. We use MARCH [18] to split the problem: It produces cubes, e.g., of the form $L_1 L_2 L_3, L_1 L_2 \neg L_3, ..., \neg L_1 \neg L_2 \neg L_3$. The cubes can be represented as a binary tree, the *cube tree*, where cubes are a path to a leaf: At each node, either the left (positive) or the right path (negative) is taken.

3   Architecture

PARACOOBA distinguishes between the masters that initiated work and workers that do the actual solving. Each node can either explore the cube tree deeper by sending work further (see Sect. 3.1) or solve the problem itself if a leaf node of the cube tree has been reached (Sect. 3.2). Nodes are also responsible for sending the result SAT or UNSAT back. PARACOOBA supports joining of new nodes dynamically, and the leaf nodes are able to wait for new tasks without consuming resources or shut down automatically, which is important if PARACOOBA is run in the cloud (Sect. 3.3). Fig. 1 gives an overview of the solving process.

3.1   Static Organization

To combine fast local solving with automatic distribution to networked compute nodes, PARACOOBA sees tasks as *paths* in the cube tree. It distinguishes between *assigned tasks* (path to leafs) that are waiting for an available local worker and *unassigned tasks*. Only unassigned tasks are distributed further. A compute node is mapped to one PARACOOBA process which contains a fixed-size thread pool of local workers. Beside maintaining information on available nodes, every compute node has a unique 64-bit ID.

Connections between compute nodes are established at any time either by an integrated auto-discovery protocol or by providing a known peer at startup. Once connected, each compute node receives the full formula sent by the master. Then it announces that it is ready to receive tasks. Each compute node has a *solving context* for every master with the problem and the cubes to solve, a queue for unassigned tasks, and one for assigned tasks. Only paths in cube trees are exchanged during solving and similar assigned tasks are solved by the same solver. New contexts are created whenever a new master becomes online, and old ones are deleted if its master becomes offline. By using low-level socket functionality (UDP/TCP), PARACOOBA can be run without setting up a specialized environment (as needed for MPI [12]).

New (unassigned) tasks received by a compute node are inserted into the queue. When a compute node becomes idle, tasks with paths to leafs are instantiated into assigned tasks to be solved locally, whereas shorter paths are split (by going deeper in the cube tree) into unassigned tasks that are distributed further. The overall strategy is to solve tasks with longer paths locally (as we are closer to leafs), while other tasks are distributed to further known compute nodes.
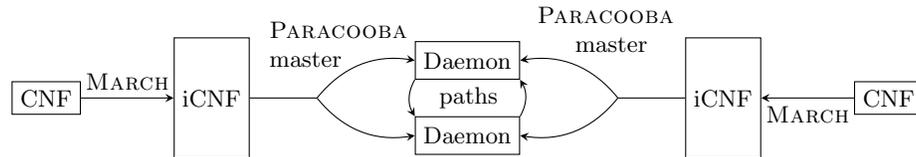

Fig. 1. Workflow of PARACOOBA with two different daemons

The SAT solver CaDiCaL [5] solves the assigned tasks incrementally [11]. It makes use of efficient preprocessing, including variable elimination and relying on efficiently restoring preprocessing steps if necessary [11]. This also provides a motivation for solving long paths locally: the cubes after a long shared path will be similar, making it possible to reuse more information compared to solving diverse cubes, where most of the preprocessing will have to be undone. If hard sub-problems are clustered on a single compute node, some can be offloaded.

3.2   Solving

We use the look-ahead solver March [18] to generate cubes. Paracooba takes the output file containing the formula and the cubes as argument. This Paracooba instance is the master node. All compute nodes parse both formula and cubes (reusing CaDiCaL's parser). After parsing, the initial task consisting of the empty path is created on the master compute node which will then branch on the first variable of the cube tree and create new unassigned tasks. These are either solved directly on the master or distributed to other compute nodes.

Paths in the cube tree are often transmitted across the network and should, therefore, have a compact representation. We represent them as 64-bit unsigned integers, where the first 58 bits describe the path in the binary tree and the last 6 bits specify the length of the path. This representation entails a maximum tree height of 58, which limits the number of different tasks to 2^{58}. This constraint is not an issue, since it is 11 orders of magnitude larger than the one million cubes used for Heule's proof for Pythagorean Triples [21] that already created a 200 TB proof. Communication between compute nodes is done using a custom protocol, which defines messages sent over UDP and TCP. The former is unreliable (packages can be dropped) and is used for non-critical messages, like auto-discovery, while the (reliable) latter is used for transmission of formulas, tasks, results, and status updates. Once a new compute node becomes known, all other nodes establish a TCP connection to it, which is used for all remaining transfers in order to circumvent UDP reliability issues in larger environments.

A sample interaction between a master and two daemon compute nodes is given in Fig. 2. First, the master starts with a problem to solve. It broadcasts an announcement request to all devices on the network. The daemons 1 and 2 answer the request and receive the formula in iCNF and a job initiator message. After that, solving starts and a path is sent from master to daemon 1. Work is rebalanced from daemon 1 to daemon 2. Once the problem is solved, the status is bubbled up to master each node is responsible for collecting the results of offloaded jobs. Finally, master can conclude (UN)SAT.

Every daemon and every master sends a status message at every "tick", i.e., in configurable intervals with default 100 ms, to all compute nodes it knows. These messages describe the current queue sizes and are used by the distribution algorithm to decide whether and where tasks should be offloaded.

Paracooba allows an "m to n" relation between masters and daemons, where daemons are used by different masters at the same time. Jobs are scheduled based on path length, not on the identity of the master.
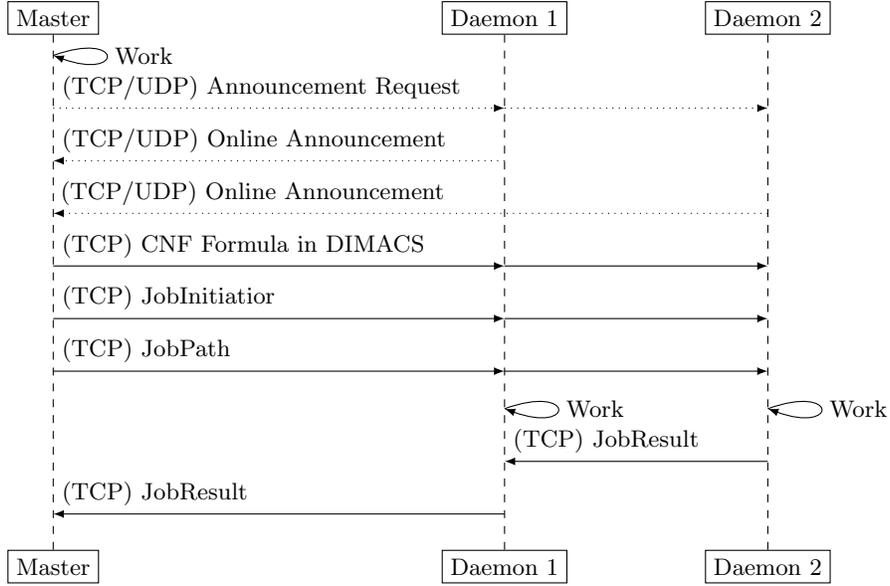
Fig. 2. Interaction between master and two daemons, without status messages

When distributing tasks to other compute nodes, the ID of the original master, of the distribution target, and of the sender are always referenced, making all Paracooba instances aware of senders and receivers of each task. For the same reason, status messages of daemon compute nodes also contain a list of all current contexts to announce the formulas for which they can solve tasks.

3.3   System Management

By automatically discovering compute nodes in the same network, Paracooba can manage its overall resources automatically. Every daemon that is newly discovered by a master gets the formula and the cubes and, once ready, can receive tasks from all other connected compute nodes. Whenever a master node goes offline, it sends an offline announcement, which removes its solving context from all connected daemons, including all results and solver instances.

Compute nodes maintain a moving average of time between status messages for all other connected nodes. If a remote compute node does not send a status update early enough, it gets removed from the list of known nodes and all tasks sent to that node get re-added to the local unassigned-task queue (and can, for example, be offloaded again).

To save compute resources, an auto-shutdown timer can be enabled to measure the time a compute node has been idle without active tasks to shut down the compute node, if no new tasks are added before the timer runs out. Because tasks get distributed to inactive nodes quickly, the timeout can be set to low values (e.g., 3 s) to reduce cost, making Paracooba suited for cloud scenarios.

Table 1. Time to solve the `cruxmiter` depending on the number of threads.

Threads t	Nodes n	Wall-Clock Time T_t^n	Speedup T_t^n / T_1^1	Network Speedup T_t^n / T_{16}^1
1	1	23 h 27 min 50 s	1.00	0.05
2	1	9 h 19 min 40 s	2.52	0.14
4	1	4 h 57 min 59 s	4.72	0.25
8	1	2 h 33 min 47 s	9.15	0.49
16	1	1 h 15 min 38 s	18.61	1.00
32	2	31 min 51 s	44.20	2.37
64	4	14 min 18 s	98.45	5.29
128	8	7 min 58 s	176.72	9.49
256	16	5 min 10 s	272.48	14.64
512	32	3 min 22 s	418.17	22.47

4   Experiments

As motivated in the introduction, we tested our tool PARACOOBA on a 32-bit `cruxmiter` problem [25], for which MARCH takes less than 10 seconds to split the initial problem into 52 520 cubes. We then run PARACOOBA on our compute cluster of 32 nodes connected through cheap commodity 1 Gbit ethernet cards. Each node contains two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB main memory. Thus every node has 16 cores.

Table 1 shows the performance with respect to the number of threads. The run-time distribution for solving cubes is heavily skewed. Most tasks need only a few seconds, but some take more than a minute, limiting the performance improvement that can be achieved by using more threads, as, following Amdahl's law, the possible speedup is limited by the time required to solve the slowest cube. After 2 minutes, 5 instances of CADICAL are still running and it takes another minute to solve those. PARACOOBA outperforms static scheduling done by splitting the cubes upfront over 512 threads and solving each resulting iCNF for each group of cubes incrementally by CADICAL (4 min 17 s wall-clock time). We experimented with resplitting cubes, but could not improve solving time.

5   Conclusion

PARACOOBA is the first distributed cube-and-conquer solver. It relies on the state-of-the-art look-ahead solver MARCH to split the problem and then efficiently distributes the cubes over as many nodes as available. Our experiments reveal that the speedup is larger than the number of cores until saturation is reached.

As future work, it would be interesting to support proof generation in the nodes and store them in the master node.

References

1. Audemard, G., Lagniez, J., Szczepanski, N., Tabary, S.: A distributed version of Syrup. In: SAT 2017. LNCS, vol. 10491, pp. 215–232. Springer (2017)
2. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 118–126. Springer (2012). https://doi.org/10.1007/978-3-642-33558-7_11
3. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S.A. (eds.) SAT 2015. LNCS, vol. 9340, pp. 156–172. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_12
4. Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_1
5. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2018-1, pp. 13–14. University of Helsinki (2018)
6. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S.A. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_29
7. Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: Berre, D.L., Järvisalo, M. (eds.) POS 2015 / POS 2018. EPiC Series in Computing, vol. 59, pp. 1–17. EasyChair (2018), http://www.easychair.org/publications/paper/RdBL
8. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
9. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Comput. **29**(7), 969–994 (2003). https://doi.org/10.1016/S0167-8191(03)00068-1
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer (2005). https://doi.org/10.1007/11499107_5
11. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 136–154. Springer (2019)
12. Graham, R.L.: The MPI 2.2 standard and the emerging MPI 3 standard. In: PVM/MPI. LNCS, vol. 5759, p. 2. Springer (2009)
13. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT **6**(4), 245–262 (2009)
14. Hamadi, Y., Sais, L. (eds.): Handbook of Parallel Constraint Reasoning. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3
15. Heisinger, M.: https://github.com/maximaximal/Paracooba.git, accessed February 2020
16. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
17. Heule, M.J.H.: Schur Number Five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) AAAI-18. pp. 6598–6606. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952

18. Heule, M.J.H., Dufour, M., van Zwieten, J., van Maaren, H.: March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: SAT 2004. LNCS, vol. 3542, pp. 345–359. Springer (2004)
19. Heule, M.J.H., Järvisalo, M., Suda, M., Iser, M., Balyo, T.: `https://satcompetition.github.io/2020/track_cloud.html`, accessed February 2020
20. Heule, M.J.H., Kullmann, O., Biere, A.: Cube-and-conquer for satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 31–59. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_2
21. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method. In: Creignou, N., Berre, D.L. (eds.) IJCAI'17. LNCS, vol. 9710, pp. 228–245. IJCAI (Aug 2017). https://doi.org/10.24963/ijcai.2017/683
22. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: A distribution method for solving SAT in grids. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 430–435. Springer (2006). https://doi.org/10.1007/11814948_39
23. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28
24. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: Barrett, C.W., Yang, J. (eds.) FMCAD 2019. pp. 28–36. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894250
25. Kaufmann, D., Kauers, M., Biere, A., Cok, D.: Arithmetic verification problems submitted to the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, p. 49. University of Helsinki (2019)
26. Konev, B., Lisitsa, A.: Computer-aided proof of Erdős discrepancy properties. Artif. Intell. **224**, 103–118 (2015). https://doi.org/10.1016/j.artint.2015.03.004
27. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the Painless framework. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 135–151. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_8
28. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Berre, D.L. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_9
29. Marques-Silva, J.P.: SAT: Disruption, demise & resurgence (2019), `http://www.pragmaticsofsat.org/2019/disruption.pdf`, pOS'2019
30. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
31. Nejati, S., Newsham, Z., Scott, J., Liang, J.H., Gebotys, C.H., Poupart, P., Ganesh, V.: A propagation rate based splitting heuristic for divide-and-conquer solvers. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 251–260. Springer (2017). https://doi.org/10.1007/978-3-319-66263-3_16
32. Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: Heule, M., Weaver, S.A. (eds.) SAT 2015. LNCS, vol. 9340, pp. 307–323. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_23
33. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability,

Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-131

34. Silva, J.P.M., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) ICCAD 1996. pp. 220–227. IEEE Computer Society / ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607

35. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer (2009)

36. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21**(4), 543–560 (1996). https://doi.org/10.1006/jsco.1996.0030