mts: a framework for parallel tree search

David Avis[1] and Charles Jordan[2]

1 School of Informatics, Kyoto University, Kyoto, Japan and School of Computer Science, McGill University, Montréal, Québec, Canada
avis@cs.mcgill.ca

2 Graduate School of Information Science and Technology, Hokkaido University, Japan
skip@ist.hokudai.ac.jp

Abstract

We describe mts, a generic framework for parallelizing certain types of tree search programs, that provides a single common wrapper containing all parallelization, and minimizes the changes needed to the existing single processor legacy code. The tree search properties required for the use of mts are satisfied by any reverse search algorithm and other tree search methods such as backtracking, branch and bound, and satisfiability testing. As examples we parallelize two simple existing reverse search codes, generating topological sorts and generating spanning trees of a graph, and one code for satisfiability testing. We give experimental results comparing the parallel codes with other codes for the same problems.
Keywords: reverse search, parallel processing, topological sorts, spanning trees
Mathematics Subject Classification (2000) 90C05

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Distributed computing, reverse search, satisfiability

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Parallel programming is a vast area and there is a great amount of literature on it (see, e.g., Mattson et al. [20]). Topics include architecture, communication, data sharing, interrupts, deadlocks, load balancing, and the distinction between shared memory and distributed computing. This is all essential for building an efficient parallel algorithm from scratch.

Our starting point was different. We had a large complex code, lrs, developed over about 20 years and tested extensively, which solved vertex/facet enumeration problems. These problems are notoriously hard and running times often take weeks or longer. The underlying algorithm, reverse search, was clearly suitable for parallelization. Nevertheless, the mathematical intricacy of the underlying problem rendered the algorithmic engineering of direct parallelization daunting. This led us to consider building all of the parallelization into a wrapper, making only minor changes to the underlying lrs code. There followed a series of implementations resulting ultimately in the authors' mplrs code [6]. The key features of mplrs are: (a) there is no parallel code inside lrs, (b) parallel threads execute lrs on non-overlapping subproblems, (c) there is no communication between threads except at the beginning and end of a subproblem execution, and (d) the computation can be distributed over a cluster of computers. Most of the topics in parallel computation mentioned above are not major issues in this restricted framework. The exception is load balancing which is the efficient distribution of work among a number of processors and is a well-studied area of parallel computation, see e.g., Shirazi et al. [22]. We experimented with a variety of load balancing methods implementable within our framework, with various degrees of success. Finally a particularly simple method, budgeting, was found to give extremely good load balancing up to several hundred cores [6].

It seemed likely that similar results could be obtained for other algorithms based on reverse search[1] or similar easily parallelizable tree search methods. Many such sequential codes exist, so designing custom wrappers for each is not desirable. Our goal was to build a single generic wrapper that could be used, with little if any modification, to do the required parallelization while maintaining features (a)-(d) described above. This resulted in mts, presented here. The current implementation[2] uses MPI and works on clusters of machines.

We describe our general approach in Section 2 and apply it to reverse search in Section 3. We give concrete examples for two simple enumeration problems: generating topological sorts and spanning trees of a graph. There were several reasons for choosing these problems. They are easily solved by reverse search, have existing codes, and provide simple examples of how to apply mts. They have been extensively studied resulting in extremely sophisticated sequential codes that are much faster than the reverse search codes but which seem to be difficult to parallelize. Knuth [19] devotes considerable attention to these codes and gives experimental results that we use as case studies in Section 5. We will see that, with sufficient parallel hardware, using mts the reverse search codes become competitive with the state of the art codes for topological sorts and faster for spanning tree enumeration. We describe only the general process of adapting a code to mts; see [7] for details on the modifications.

Tree search has wide uses, of which enumeration is just one example. In fact it is a very specific example as all nodes in the enumeration tree are visited. Doing this in parallel does not create any major experimental design issues since the same tree must be searched in all cases. Other uses include backtracking, game tree search, branch and bound, and satisfiability problems. For these the idea is normally *not* to search the entire tree but to prune subtrees when possible. The tree generated in these cases will normally differ depending on the choices made at early stages and luck is involved. Designing and interpreting experiments to evaluate the effect of parallelization is now a challenge: even if the work has been efficiently spread between processes bad luck may cause the whole computation to be slow. The mts framework can be applied to these types of problems. However, unlike the enumeration problems, we may need to share some information between processes. As an example we present a parallelization of the sequential satisfiability code Minisat in Section 4.1.

2 The mts framework

The goal of mts is to parallelize existing tree search codes with minimal internal modification of these codes. The tree search codes should satisfy certain conditions, specified below. The mts implementation starts a user-specified number of processes on a cluster of computers. One process becomes the *master*, another becomes the *consumer*, and the remaining are *workers* which essentially run the original tree search code on specified subtrees. Communication is limited; workers are not interrupted and do not communicate between themselves.

The master sends the input data and parametrized subproblems to workers, informs the other processes to exit when appropriate, and handles checkpointing. The consumer receives and synchronizes output. Workers get budgeted subproblems from the master, run the legacy

[1] In 2008, John White made a list of 130 different applications and implementations, see link at [5].
[2] Version used here available at `https://www-alg.ist.hokudai.ac.jp/~skip/mts/`

code, send output to the consumer, and return unfinished subproblems to the master. This general approach is similar to but simpler than the well-known work-stealing approach [12].

Generating subproblems can be done in many ways. One way would be to report nodes at some initial fixed depth. This works well for balanced trees but many trees encountered in practice are highly unbalanced and the vast majority of subtrees contain few nodes. Increasing the initial search depth does not solve this problem. Ideally we would only break up the large subtrees and in the development of mplrs we tried various ways to estimate the size of a given subtree. Experimentally this did not work well due to the high variance of the estimator and the wasted cost of doing many estimates.

The idea that worked best, and is implemented in mts, was also the simplest: a heuristic to determine large subtrees called *budgeting*. When assigning work the master specifies that a worker should terminate after completing a certain amount of work, called a *budget*, and then return a list of unexplored subtrees. The precise budget may depend on the application. For enumeration problems it could be the number of nodes visited by the worker. Some advantages of budgeting are (a) small subtrees are explored without being broken up, (b) large subtrees will be broken up repeatedly, (c) each worker returns periodically for reassignment, can give information to be passed on to other workers and receive such information and (d) it is implemented on-the-fly and avoids the duplication of work done in estimation. Implementing budgeting does not require interrupting workers or communication between workers. The master uses dynamic budgets to control the job list: small budgets break up more subtrees and lengthen the joblist while large budgets have the reverse effect.

Additional features of mts include checkpointing and restarts, allowing the user to move jobs or free computing resources without losing work. mts can produce various histograms to help tune performance. Histograms and their uses are described in Section 6.

2.1 Sequential tree search code

To be suitable for parallelization with mts the underlying tree search code, which we will call search, must satisfy a few properties. First, when given a positive budget, search should either finish the given job or return a list of unexplored nodes. Any unexplored node should represent a smaller portion of the unfinished work, i.e. running search (with positive budgets) on the unexplored nodes and any resulting unexplored nodes will eventually result in finishing the original job. The code should also interpret the budget in some suitable way where larger budgets correspond to doing more work than smaller budgets. This may require some modification of the legacy code. Our applications usually interpret the budget as *number of traversed nodes* and *depth*, but this is not required (see conflict budgeting in Section 4.1).

Any given worker must be able to work on any given unexplored node that mts has seen. It is helpful for the unexplored nodes to represent non-overlapping jobs. mts supports sharing data between workers, but it is helpful for shared data to be small. Implementing a shared memory version of mts could help performance when large amounts of data are shared. Shared data is not used in our enumeration applications. It is used for satisfiability and similar applications to prune the search tree.

2.2 Master process

The master process begins with initialization, including obtaining an application-provided initial *start_vertex*. It chooses an initial worker and sends it the initial subproblem. We cannot yet proceed in parallel, so the master uses a user-specified (or very small default) initial budget (*max_depth* and/or *max_nodes*) to ensure that this worker will return (hopefully many) unfinished subproblems quickly to create a job list L. The master then executes its main loop which assigns budgeted subproblems to workers, collects unfinished subproblems to add to L, and collects/sends updated *shared_data* from/to the workers. Assigning *shared_data* updates to the master is not essential: it simplifies checkpointing but can increase load on the master and interconnect. Each worker either finishes its subproblem or reaches its budget limitation (*max_depth* and *max_nodes*) and returns unfinished subproblems to the master for insertion into L. This continues until no workers are running and the master has no unfinished subproblems. Once the main loop ends, the master informs all processes to finish. The main loop performs the following tasks:

- subproblems and relevant *shared_data* updates are sent to free workers when available;
- check if any workers are done, mark them as free and receive their unfinished subproblems;
- check and receive *shared_data* updates.

The pseudocode is given as Algorithm 1 in the Appendix. Communication is non-blocking and work proceeds when required information is available.

Using reasonable parameters is critical to performance. This is done dynamically by observing $|L|$. We use parameters *lmin*, *lmax* and *scale* which depend on the type of tree search problem being handled. The following default values are used in this paper. Initially, to create a reasonable size list L, we set $max\_depth = 2$ and $max\_nodes = 5000$. Therefore the initial worker will generate subtrees at depth 2 until 5000 nodes have been visited and then terminates sending roots of unvisited subtrees back to the master. Additional workers are given the same aggressive parameters until $|L|$ grows larger than *lmin* times the number of processors, at which point *max_depth* is removed. Once $|L|$ is larger than *lmax* times the number of processors, we multiply the budget by *scale*. With $scale = 40$ workers will not generate any new subproblems unless their tree has at least 200,000 nodes. If $|L|$ drops below these bounds we return to these smaller budgets. The default is $lmin = 1, lmax = 3$. In Section 6 we show an example of how $|L|$ typically behaves with these settings.

2.3 Workers

The worker processes are simpler – they receive the problem at startup, and then repeat their main loop: receive a parametrized subproblem and possible *shared_data* updates from the master, work on the subproblem subject to the parameters, send the output to the consumer, and send updated *shared_data* and unfinished subproblems to the master if the budget is exhausted. The pseudocode is given as Algorithm 2 in the Appendix.

2.4 Consumer process

The consumer process in mts is the simplest. The workers send output to the consumer in exactly the format it should be output (i.e., this formatting is done in parallel). The consumer simply outputs it. By synchronizing output to a single destination, the consumer delivers a continuous output stream to the user in the same way as search does. The pseudocode is given as Algorithm 3 in the Appendix.

3 Applying mts to reverse search

Reverse search is a technique for generating large relatively unstructured sets of discrete objects [4]. In its most basic form, reverse search can be viewed as the traversal of a spanning tree, called the reverse search tree T, of a graph $G = (V, E)$ whose nodes are the objects to be generated. Edges in the graph are specified by an adjacency oracle, and the subset of edges of the reverse search tree are determined by an auxiliary function, which can be thought of as a local search function f for an optimization problem defined on the set of objects to be generated. One vertex, v^*, is designated as the *target* vertex. For every other vertex $v \in V$ repeated application of f must generate a path in G from v to v^*. The set of these paths defines the reverse search tree T, which has root v^*.

A reverse search is initiated at v^*, and only edges of the reverse search tree are traversed. When a node is visited, the corresponding object is output. Since there is no possibility of visiting a node by different paths, the visited nodes do not need to be stored. Backtracking can be performed in the standard way using a stack, but this is not required as the local search function can be used for this purpose.

In the basic setting described here a few properties are required. Firstly, the underlying graph G must be connected and an upper bound on the maximum vertex degree, Δ, must be known. The performance of the method depends on G having Δ as low as possible. An adjacency oracle $\mathrm{Adj}(v, j)$, similar to that described in Section 2.1, must be capable of generating the adjacent vertices of any given vertex v in G. For each vertex $v \neq v^*$ the local search function $f(v)$ returns the tuple (u, j) where $v = \mathrm{Adj}(u, j)$ which defines the parent u of v in T. One difference with the discussion in Section 2.1 is that the adjacency oracle in reverse search will also provide the parent of each node of T, except the root. This parent node can easily be determined by use of f. Pseudocode is given in Algorithm 4 in the Appendix and is invoked by setting $start\_vertex = v^*$. C implementations for several simple enumeration problems are given at [5]. For convenience later, we do not output the $start\_vertex$ in the pseudocode shown. Note that the vertices are output as a continuous stream. Also note that Algorithm 4 does not require the parameter $start\_vertex$ to be the root v^* of the entire search tree. If an arbitrary node in the tree is given, the algorithm reports the subtree rooted at this node and terminates.

We need to implement budgeting in order to parallelize Algorithm 4 with mts. We do this in two ways that may be combined. Firstly we introduce the parameter max_depth which terminates the tree search at that depth returning any unvisited subtrees. Secondly we introduce a parameter max_nodes which terminates the tree search after this many nodes have been visited and again returns the roots of all unvisited subtrees. This entails backtracking to the root and returning the unvisited siblings of each node in the backtrack path. These modifications are straight forward and given in Algorithm 5 in the Appendix.

To output all nodes in the subtree of T rooted at v we set $start\_vertex = v$, $max\_nodes = +\infty$ and $max\_depth = +\infty$. This reduces to Algorithm 4 if $v = v^*$. To break up T into subtrees we have two options that can be combined. Firstly we can set the max_depth parameter resulting in all nodes at that depth to be flagged as unexplored. Secondly we can set the budget parameter max_nodes. In this case, once this many nodes have been explored the current node and all unexplored siblings on the backtrack path to the root are output and flagged as unexplored.

3.1 Example 1: Topological sorts

In the tutorial [5] a C implementation (*per.c*) is given for the reverse search algorithm for generating permutations. A small modification of this code generates all topological sorts of a partially ordered set that is given by a directed acyclic graph (DAG). Such topological sorts are also called linear extensions or topological orderings. The code modification is given as Exercise 5.1 and a solution to the exercise (*topsorts.c*) is at [5]. Here we describe how to modify this code to allow parallelization via the mts interface to produce the program mtopsorts. The details and code are available at [5].

It is convenient to describe the procedure as two phases. Phase 1 implements budgeting and organizes the internal data in a suitable way. This involves modifying an implementation of Algorithm 4 to an implementation of Algorithm 5 that can be independently tested. We need to prepare a global data structure bts_data which contains problem data obtained from the input. In Phase 2 we build a node structure for use by the mts wrapper and add necessary routines to allow initialization and I/O in a parallel setting. In practice this involves sharing a common header file with mts. The resulting program *btopsorts.c* can be compiled as a sequential code or with mts as a parallel code with no change in the source files.

In the second phase we add the 'hooks' that allow communication with mts. This involves defining a Node structure which holds all necessary information about a node in the search tree. The roots of unexplored subtrees are maintained by mts for parallel processing. Therefore whenever a search terminates due to the *max_nodes* or *max_depth* restrictions, the Node structure of each unexplored tree node is returned to mts. As we do not wish to customize mts for each application, we use a very generic node structure. The user should pack and unpack the necessary data into this structure as required. The Node structure is defined in the mts header.

The efficiency of mts depends on keeping the job list non-empty until the end of the computation, without letting it get too large. Depending on the application, there may be a substantial restart cost for each unexplored subtree. Surely there is no need to return a leaf as an unexplored node, and the *prune=0* option checks for this. Further, if an unexplored node has only one child it may be advantageous to explore further, terminating either at a leaf or at a node with two or more children, which is returned as *unexplored.* The *prune=1* option handles this condition, meaning that no isolated nodes or paths are returned as unexplored. Note that pruning is not a built-in mts option; it is an example of options that applications may wish to include and was implemented in mtopsorts.

3.2 Example 2: Spanning trees

In the tutorial [5] a C implementation (*tree.c*) is given for the reverse search algorithm for all spanning trees of the complete graph. An extension of this to generate all spanning trees of a given graph is stated as Exercise 6.3. Applying Phase 1 and 2 as described above results in the code *btree.c*. Again this may be compiled as a sequential code or with the mts wrapper to provide the parallel implementation mtree. All of these codes are given at the URL [5].

4 Applying mts to satisfiability

Boolean satisfiability (SAT) asks us to determine the existence of (or find) satisfying assignments for propositional formulas, see [11] for more background. SAT solvers have made tremendous progress over the years, and are now widely used as general NP solvers. While most application problems seem to result in easy SAT instances [9], there has long been interest in parallel SAT solvers for hard instances. Despite the many challenges [14, 17] in parallel SAT, there are recent successes [15].

There are two major approaches to parallel SAT solvers. Either one somehow partitions the space of possible assignments and uses divide-and-conquer (e.g., [2] for a recent example) or one uses the portfolio approach and runs many sequential solvers on the original problem (e.g., plingeling [10]). In either case, a major issue is determining which learnt clauses[3]

[3] CDCL solvers learn clauses during the search, pruning the search space. See, e.g., Chapter 4 of [11].

to share between workers [3]. While sharing these clauses helps prune the search space, additional clauses slow the solver and enormous numbers of clauses are learned.

Another question for divide-and-conquer solvers is the question of how to divide the search space. Many approaches have been tried, often setting initial variables and using a common feature of sequential solvers to "solve under assumptions". Some recent solvers (e.g., [2] and treengeling [10]) work on these subproblems subject to some budget, and hard subproblems can be split again. Cube-and-conquer [16] is another recent approach that uses look-ahead solvers to divide the search space for CDCL solvers.

4.1 mtsat: parallelizing Minisat with mts

We used mts to implement a divide-and-conquer solver mtsat, using Minisat 2.2.0 as sequential solver. Our goal was to demonstrate the use of *shared_data* and show that mts can be used in settings other than enumeration. mtsat is still experimental and much work remains to reach the level of state-of-the-art dedicated parallel SAT solvers, but it allows for experimentation with, e.g., budgeting and restart strategies in parallel SAT.

Minisat [13] supports solving under assumptions, i.e. solving subject to some partial assignment. It also supports solving subject to a budget, given in propagations or conflicts, returning *unknown* if the given subproblem could not be solved within the budget.

The major modification required is to report unexplored partial assignments when the budget is exhausted. At any point in the search, SAT solvers distinguish between decision variables and propagated variables. Decision variables are those where the solver chose an assignment, while propagated variables are those where the solver was able to determine (because of a unit clause) that only one option need be explored. It suffices to return unexplored nodes corresponding to the current partial assignment and to those formed by taking the unexplored options for decision variables (including the last one) along the backtrack path.

Regarding learnt clauses, we implemented a simple scheme sharing only learnt unit clauses. The idea is that short clauses cut the search tree more than longer clauses; an early version of plingeling also shared only units [10]. We avoided more sophisticated approaches to sharing clauses [3, 2], using conflicts to prune the job list L and similar ideas for simplicity.

mtsat includes additional options. For example, while the parallel solvers most similar to our approach [2, 10] budget using conflicts – we added the option to budget using *decisions*. Conflict budgets correspond to hitting a leaf in the search tree, while decision budgets correspond to nodes in the search space (omitting propagated variables since those are forced). Conflict budgets are attractive, but decision budgets correspond more closely to the budgets used in Section 3 and allow us to experiment with different budgeting techniques.

Modern solvers generally perform random restarts, abandoning the current search to start over (cf. Chapter 4 of [11]) and hopefully avoid getting stuck in hard parts of the search space. We split problems along the backtrack path and schedule these abandoned portions of the search space for later exploration – possibly resulting in much duplicated work. We therefore added an option to disable restarts, in order to experiment with their impact on performance in mtsat, and formula preprocessing, to experiment with the idea that avoiding preprocessing can be beneficial to divide-and-conquer parallel SAT solvers [14].

The total is 50 lines of changes to legacy Minisat (including support to parse inputs from strings) of the original 4803 lines, plus a few hundred lines of generic code interfacing the Minisat API and mts that can be re-used. Essentially identical changes suffice to parallelize Glucose (since it is based on Minisat) and others. One could easily support workers using a mix of solvers, a hybrid of the divide-and-conquer and portfolio approaches to parallel SAT.

5 Experimental results

The tests were performed at Kyoto University on mai32, a cluster of 5 nodes with a total of 192 identical processor cores, consisting of:

- mai32abcd: 4 nodes, each containing: 2x Opteron 6376 (16-core 2.3GHz), 32GB memory, 500GB hard drive (128 cores in total);
- mai32ef: 4x Opteron 6376 (16-core 2.3GHz), 64 cores, 256GB memory, 4TB hard drive.

A complete description of the problems solved below is given in [7] and the input files are available by following the link to tutorial2 at [5].

5.1 Topological sorts: mtopsorts

The tests were performed using the following codes:

- VR: obtained from [1], generates topological sorts in lexicographic order via the Varol-Rotem algorithm [23] (Algorithm V in Section 7.2.1.2 of [19]);
- Genle: also obtained from [1], generates topological sorts in Gray code order using the algorithm of Pruesse and Rotem [21];
- btopsorts: derived from the reverse search code *topsorts.c* [5] as described in Section 3.1;
- mtopsorts: mts parallelization of btopsorts.

For the tests all codes were used in count-only mode due to the enormous output that would otherwise be generated. All codes were used with default parameters:

$$max\_depth = 2 \quad max\_nodes = 5000 \quad scale = 40 \quad lmin = 1 \quad lmax = 3 \tag{1}$$

The following graphs were chosen, listed in order of increasing edge density: *pm22*, *cat42*, $K_{8,9}$. The constructions for the first two partial orders are well known (see, e.g., Section 7.2.1.2 of [19]) and the third is a complete bipartite graph.

Graph	m	n	No. of perms	VR	Genle	btopsorts	mtopsorts				
	nodes	edges					12	24	48	96	192
pm22	22	21	13,749,310,575	179	14	12723	1172	595	360	206	125
cat42	42	61	24,466,267,020	654	171	45674	4731	2699	1293	724	408
$K_{8,9}$	17	72	14,631,321,600	159	5	8957	859	445	249	137	85

Table 1 Topological sorts: mai32, times in secs

Results are in Table 1. The reverse search code btopsorts is very slow, over 900 times slower than Genle and over 70 times slower than VR on *pm22*. However the parallel mts code obtains excellent speedups and is faster than VR on all problems when 192 cores are used.

5.2 Spanning trees: mtree

The tests were performed using the following codes:

- grayspan: Knuth's implementation [18] of an algorithm that generates all spanning trees of a given graph, changing only one edge at a time, as described in Malcolm Smith's M.S. thesis, *Generating spanning trees* (University of Victoria, 1997);
- grayspspan: Knuth's improved implementation of grayspan: "This program combines the ideas of grayspan and spspan, resulting in a glorious routine that generates all spanning trees of a given graph, changing only one edge at a time, with 'guaranteed efficiency'—in the sense that the total running time is $O(m + n + t)$ when there are m edges, n vertices, and t spanning trees." [18];

- btree: derived from the reverse search code *tree.c* [5] as described in Section 3.2;
- mtree: mts parallelization of btree.

Both grayspan and grayspspan are described in detail in Knuth [19]. Again all codes were used in count-only mode and with the default parameters (1). The problems chosen were the following graphs which are listed in order of increasing edge density: *8-cage*, P_5C_5, C_5C_5, $K_{7,7}$, K_{12}. The latter 4 graphs were motivated by Table 5 in [19]: P_5C_5 appears therein and the other graphs are larger versions of examples in that table.

Graph	m	n	No. of trees	grayspan	grayspspan	btree	mtree				
	nodes	edges					12	24	48	96	192
8-cage	30	45	23,066,015,625	3166	730	10008	1061	459	238	137	92
P_5C_5	25	45	38,720,000,000	3962	1212	8918	851	455	221	137	122
C_5C_5	25	50	1,562,500,000,000	131092	41568	230077	26790	13280	7459	4960	4244
$K_{7,7}$	14	49	13,841,287,201	699	460	2708	259	142	68	51	61
K_{12}	12	66	61,917,364,224	2394	1978	3179	310	172	84	97	148

Table 2 Spanning tree generation: mai32, times in secs

The computational results are given in Table 2. This time the reverse search code is a bit more competitive: about 3 times slower than grayspan and about 14 times slower than grayspspan on *8-cage* for example. The parallel mts code runs about as fast as grayspspan on all problems when 12 cores are used and is significantly faster after that. Near linear speedups are obtained up to 48-cores but then tail off. For the two dense graphs $K_{7,7}$ and K_{12} the performance of mts is actually worse with 192 cores than with 96.

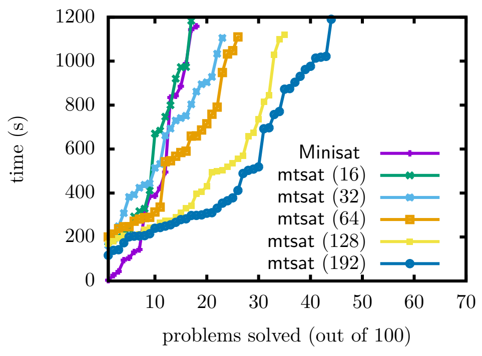
5.3 Satisfiability

The tests were performed using the following codes:

- Minisat: version 2.2.0, classic sequential solver [13];
- mtsat: parallel solver using mts and Minisat 2.2.0;
- lingeling, treengeling: version bbc, sequential and (shared memory) parallel solvers [10].

Benchmarking parallel SAT solvers is challenging [14] and any particular instance may give superlinear speedups or timeouts. We use a standard set of hard instances from applications, and count the number of problems that each solver can solve within a given time. We re-use the setup of [2], i.e. the 100 instances in the parallel track of SAT Race 2015 [8] and a timeout of 20 minutes. Results are in Figure 1. Due to different computers used, our results are not directly comparable to those in [2]. As noted by [9], solvers like mtsat can use substantial memory on very large instances, limiting the number of processes that can execute in a given amount of memory. The computers we used had sufficient memory for the instances used.

The results in Figure 1 show improvement from additional cores using default parameters and decision budgeting with no attempt at tuning. Performance with conflict budgeting is shown in Figure 2, using an initial budget of 10000 conflicts (i.e. the corresponding value in [2]). All non-timeout outputs are correct, and the 32-core run with conflict budgeting solves `62bits_10.dimacs.cnf` (reported as unsolved in the SAT Race 2015 results) giving a correct satisfying assignment. It is likely that experimenting with parameter values can improve performance, and using a newer sequential solver on the workers may be another source of improvement given the performance treengeling achieves starting from the higher baseline performance of lingeling.

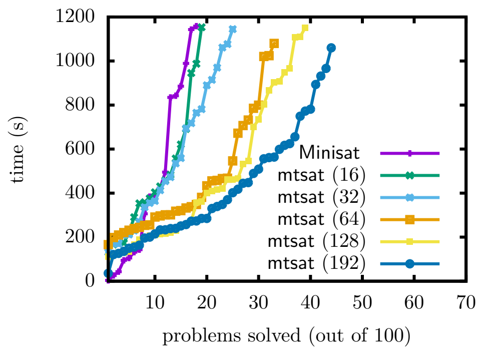


(a) Instances solved vs time

Solver	SAT	UNSAT	Total
Minisat	18	1	19
mtsat (16)	17	1	18
mtsat (32)	22	2	24
mtsat (64)	23	4	27
mtsat (128)	29	7	36
mtsat (192)	35	10	45
lingeling	17	10	27
treengeling (32)	38	21	59

(b) Instances solved within 1200s

Figure 1 mtsat performance (decision budgeting, default parameters (1))



(a) Instances solved vs time

Solver	SAT	UNSAT	Total
Minisat	18	1	19
mtsat (16)	18	2	20
mtsat (32)	23	3	26
mtsat (64)	27	7	34
mtsat (128)	30	10	40
mtsat (192)	34	11	45

(b) Instances solved within 1200s

Figure 2 mtsat performance (conflict budgeting, $max\_nodes = 10000$, $scale = 10$)

6 Evaluating and improving performance

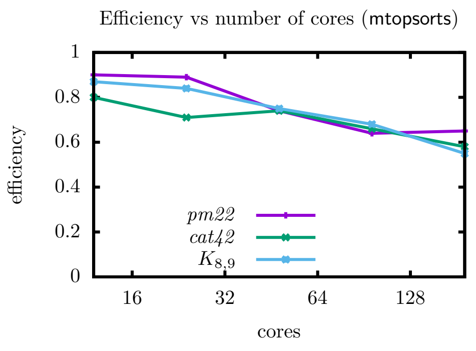
Our main measures of performance for the enumeration problems are the elapsed time taken and the *efficiency* defined as:

$$\text{efficiency} = \frac{\text{single core running time}}{\text{number of cores} * \text{multicore running time}} \quad (2)$$

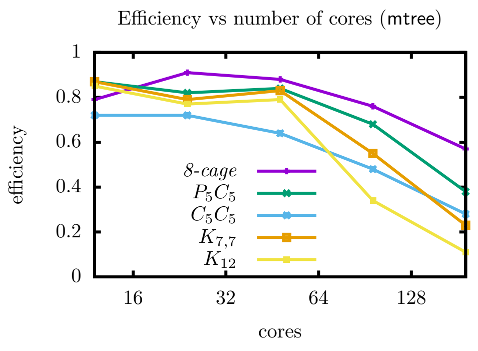
Multiplying efficiency by the number of cores gives the speedup. Speedups that scale linearly with the number of cores give constant efficiency. External factors can affect performance as the load on the machine increases. One example is dynamic overclocking, where the speed of working cores may be increased by 25%–30% when other cores are idle. This limits the maximum efficiency achievable when all cores are used, since the single core running times are measured on otherwise idle machines. In Figure 3 we plot the efficiencies obtained by mtopsorts and mtree for the runs shown in Tables 1 and 2 respectively.

The amount of work contained in a subproblem can vary dramatically. mts can produce histograms to help understand and tune its performance. We discuss three of these here: processor usage, job list size and distribution of subproblem sizes. Figure 4 shows the first two histograms for the mtopsorts run on $K_{8,9}$ with default parameters (1).

We see the master struggling to keep workers busy despite having jobs available. This suggests that we can improve performance with better parameters. Here, a larger *-scale* or



(a) Efficiency with mtopsorts



(b) Efficiency with mtree

Figure 3 Efficiency vs number of cores (data from Tables 1 and 2)

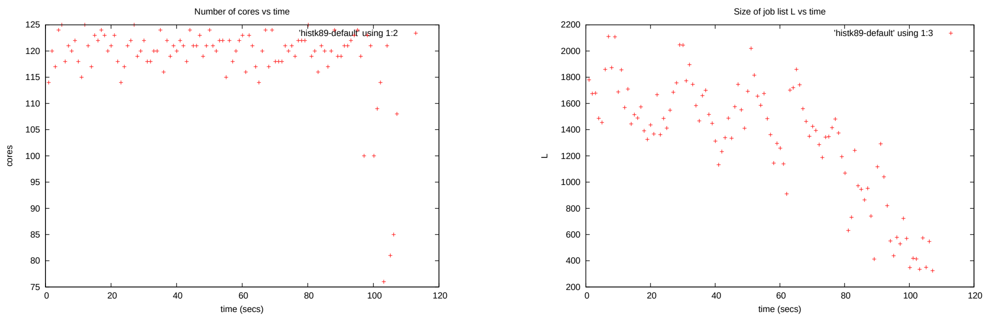


Figure 4 Histograms for mtopsorts on $K_{8,9}$: busy workers (left) job list size (right)

-*maxnodes* value may help, since it will allow workers to do more work (assuming a sufficiently large subproblem) before contacting the master.

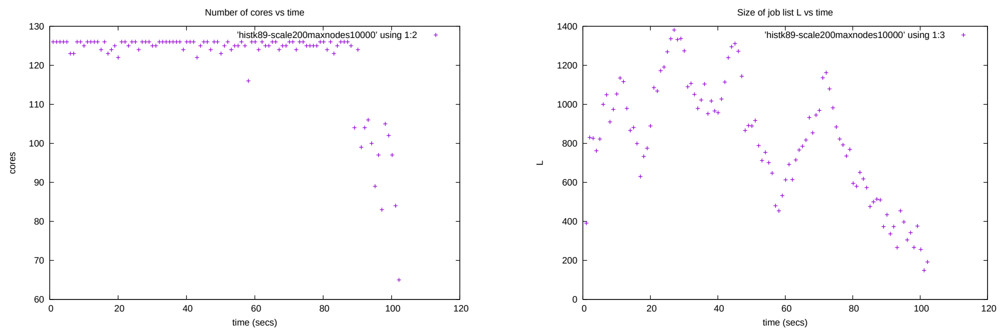


Figure 5 Histograms with -*scale* 200 -*maxnodes* 10000 on $K_{8,9}$: busy workers (l), joblist size (r)

Figure 5 shows the result of using 200 for -*scale* and 10000 for -*maxnodes*. These parameters produce less than half the number of total number of jobs compared to the default parameters, and increase overall performance by about five percent on this input.

In addition to the performance histograms, mts can generate a *frequency* file containing a list of values returned by each worker on the completion of each job. For the enumeration applications this is normally the number of nodes visited by the worker during the job. Such a list provides statistical information about the tree that is helpful when tuning the parameters for better performance. For example, it may be helpful to implement and use pruning if many jobs correspond to leaves. Likewise, increasing the budget will have limited effect if only few jobs use the full budget. Figure 6 shows the distribution of subproblem sizes that was produced in a run of mtopsorts on $K_{8,9}$ with default parameters (1). L is usually large so the scaled budget constraint of 200000 is normally in use. The left figure shows this constraint was invoked about 15000 times. The right figure shows that most subproblems have less than 40 nodes and so are not broken up. The three spikes in the middle of the left figure are interesting and show there are large numbers of subtrees with these specific sizes. This is probably due to the high symmetry of the graph $K_{8,9}$.

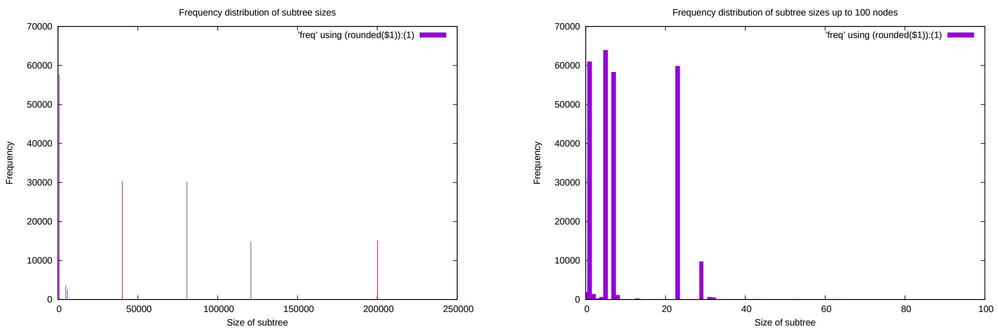


Figure 6 Subproblem sizes for $K_{8,9}$: all (left) small subproblems only (right)

7 Conclusions

We have presented a generic framework for parallelizing reverse search codes requiring only minimal changes to the legacy code. Two features of our approach are that the parallelizing wrapper does not need to be user modified and the modified legacy code can be tested in standalone single processor mode. Applying this framework to two very basic reverse search codes we obtained comparable results to that previously obtained by the customized mplrs wrapper applied to the the complex lrs code [6]. We expect that many other reverse search applications and will obtain similar speedups when parallelized with mts.

The application to SAT demonstrates the use of shared data, and the ease with which a widely-used existing legacy code can be parallelized using mts. While mtsat remains work in progress, it shows some promise and further experimentation can likely improve performance.

Acknowledgements. This work was partially supported by JSPS Kakenhi Grants 16H02785, 23700019 and 15H00847, Grant-in-Aid for Scientific Research on Innovative Areas, 'Exploring the Limits of Computation (ELC)'.

References

1 Combinatorial Object Server : Linear extensions. `http://theory.cs.uvic.ca/inf/pose/LinearExt.html`.

2 Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel SAT solver. In *CP*, volume 9892 of *LNCS*, pages 30–48, 2016.

3 Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *SAT*, volume 8561 of *LNCS*, pages 197–205, 2014.

4 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65:21–46, 1993.

5 David Avis and Charles Jordan. Reverse search: tutorials, 2000,2016. `http://cgm.cs.mcgill.ca/~avis/doc/tutorial`.

6 David Avis and Charles Jordan. mplrs: A scalable parallel vertex/facet enumeration code. *CoRR*, 2015. URL: `http://arxiv.org/abs/1511.06487`.

7 David Avis and Charles Jordan. A parallel framework for reverse search using mts. *CoRR*, 2016. URL: `https://arxiv.org/abs/1610.07735`.

8 Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT Race 2015. *Artificial Intelligence*, 241:45–65, 2016.

9 Tomáš Balyo, Marijn J.H. Heule, and Matti Järvisalo. SAT Competition 2016: Recent developments. In *AAAI*, 2017.

10 Armin Biere. Lingeling and friends entering the SAT Challenge 2012. In *Proc. SAT Challenge 2012*, volume B-2012-2 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 33–34, 2012.

11 Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

12 Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

13 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518, 2003.

14 Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. In *AAAI*, pages 2120–2125, 2012.

15 Marijn J.H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean Pythagorean triples problem via cube-and-conquer. In *SAT*, volume 9710 of *LNCS*, pages 228–245, 2016.

16 Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, volume 7261 of *LNCS*, 2012.

17 George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI*, pages 481–488, 2013.

18 Donald E. Knuth. Programs to read. `http://www-cs-faculty.stanford.edu/~uno/programs.html`.

19 Donald E. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley Professional, 2011.

20 Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

21 Gara Pruesse and Frank Ruskey. Generating the linear extensions of certain posets by transpositions. *SIAM Journal on Discrete Mathematics*, 4(3):413–422, 1991.

22 Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

23 Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.

Appendix

Algorithm 1 Master process

```
1: procedure MASTER($input\_data$, $max\_depth$, $max\_nodes$, $lmin$, $lmax$, $scale$, $num\_workers$)
2:     Send ($input\_data$) to each worker
3:     Create empty table $sdata$
4:     Create empty list $L$
5:     Get $start\_vertex$ from application, add to $L$
6:     $size \leftarrow num\_workers + 2$
7:     while $L$ is not empty or some worker is marked as working do
8:         while $L$ is not empty and some worker not marked as working do
9:             if $|L| < size \cdot lmin$ then
10:                $maxd \leftarrow max\_depth$
11:            else
12:                $maxd \leftarrow \infty$
13:            end if
14:            if $|L| > size \cdot lmax$ then
15:                $node\_budget \leftarrow scale \cdot max\_nodes$
16:            else
17:                $node\_budget \leftarrow max\_nodes$
18:            end if
19:            Remove next element $start$ from $L$
20:            Send ($start$, $maxd$, $node\_budget$) to first free worker $i$
21:            Mark $i$ as working
22:            Send any $shared\_data$ in $sdata$ newer than $i$ has
23:        end while
24:        for each marked worker $i$ do
25:            Check for new message $unfinished$ from $i$
26:            if incoming message $unfinished$ from $i$ then
27:                Join list $unfinished$ to $L$
28:                Receive $shared\_data$ update from $i$
29:                Unmark $i$ as working
30:                if non-empty update then
31:                    Update $i$'s $shared\_data$ in $sdata$
32:                end if
33:            end if
34:        end for
35:    end while
36:    Call application with final set of $shared\_data$
37:    Send terminate to all processes
38: end procedure
```

Algorithm 2 Worker process

```
1: procedure WORKER
2:     Receive ($input\_data$) from master
3:     Create empty $shared\_data$
4:     while true do
5:         Wait for message from master
6:         if message is terminate then
7:             Exit
8:         end if
9:         Receive ($start\_vertex$, $max\_depth$, $max\_nodes$)
10:        Receive $shared\_data$ updates, update local copy
11:        Call search ($start\_vertex$, $max\_depth$, $max\_nodes$, $shared\_data$)
12:        Send list of unfinished vertices to master
13:        Send $shared\_data$ update to master
14:        Send output list to consumer
15:    end while
16: end procedure
```

Algorithm 3 Consumer process

```
1: procedure CONSUMER
2:     while true do
3:         Wait for incoming message
4:         if message is terminate then
5:             Exit
6:         end if
7:         Output this message
8:     end while
9: end procedure
```

Algorithm 4 Generic Reverse Search

```
1: procedure RS($start\_vertex$)
2:     $v \leftarrow v^*$    $j \leftarrow 0$    $depth \leftarrow 0$
3:     repeat
4:         while $j < \Delta$ do
5:             $j \leftarrow j + 1$
6:             if $f(\mathrm{Adj}(v, j)) = v$ then                ▷ forward step
7:                 $v \leftarrow \mathrm{Adj}(v, j)$
8:                 $j \leftarrow 0$
9:                 $depth \leftarrow depth + 1$
10:                output $v$
11:            end if
12:        end while
13:        if $depth > 0$ then                                  ▷ backtrack step
14:            $(v, j) \leftarrow f(v)$
15:            $depth \leftarrow depth - 1$
16:        end if
17:    until $depth = 0$ and $j = \Delta$
18: end procedure
```

Algorithm 5 Budgeted Reverse Search

```
1: procedure BRS($start\_vertex$, $max\_depth$, $max\_nodes$)
2:     $j \leftarrow 0$   $v \leftarrow start\_vertex$   $count \leftarrow 0$   $depth \leftarrow 0$
3:     repeat
4:         $unexplored \leftarrow$ false
5:         while $j < \Delta$ and $unexplored =$ false do
6:             $j \leftarrow j + 1$
7:             if $f(\mathrm{Adj}(v, j)) = v$ then                          ▷ forward step
8:                 $v \leftarrow \mathrm{Adj}(v, j)$
9:                 $j \leftarrow 0$
10:                $count \leftarrow count + 1$
11:                $depth \leftarrow depth + 1$
12:                if $count \geq max\_nodes$ or $depth = max\_depth$ then
13:                    $unexplored \leftarrow$ true                        ▷ budget exhausted
14:                end if
15:                put_output ($v$, $unexplored$)
16:            end if
17:        end while
18:        if $depth > 0$ then                                          ▷ backtrack step
19:            $(v, j) \leftarrow f(v)$
20:            $depth \leftarrow depth - 1$
21:        end if
22:    until $depth = 0$ and $j = \Delta$
23: end procedure
```