# mts: a light framework for parallelizing tree search codes

David Avis[a] and Charles Jordan[b]

[a]School of Informatics, Kyoto University, Kyoto, Japan and School of Computer Science, McGill University, Montréal, Québec, Canada; [b] Graduate School of Information Science and Technology, Hokkaido University, Japan

**ARTICLE HISTORY**
Compiled October 29, 2019

**ABSTRACT**
We describe mts, a generic framework for parallelizing certain types of tree search programs including reverse search, backtracking, branch and bound and satisfiability testing. It abstracts and generalizes the ideas used in parallelizing lrs, a reverse search code for vertex enumeration. mts supports sharing information between processes which is important for applications such as satisfiability testing and branch-and-bound. No parallelization is implemented in the legacy single processor programs minimizing the changes needed and simplifying debugging. mts is written in C, uses MPI for parallelization and can be used on a network of computers. We give four examples of reverse search codes parallelized by using mts: topological sorts, spanning trees, triangulations and Galton-Watson trees. We also give a parallelization of two codes 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; triangulations; satisfiability testing

**AMS CLASSIFICATION**
90C05

## 1. Introduction

Parallel programming is a vast area and there is a great amount of literature on it (see, e.g., Mattson et al. [36]). 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

CONTACT C. Jordan. Email: skip@ist.hokudai.ac.jp

underlying lrs code. There followed a series of implementations resulting ultimately in the authors' mplrs code [8]. 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, (d) the computation can be distributed over a cluster of computers, and (e) the wrapper is directly inserted into the lrs library. Most of the topics in parallel computation mentioned above are not major issues in this restricted framework. The exception is load balancing for which we use a particularly simple method which consists of budgeting the number of nodes evaluated in a subproblem.

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)–(e) described above. This resulted in mts, presented here. The current implementation[2] uses MPI and works on clusters of machines. The mts framework is more general than mplrs in that it allows the sharing of data obtained by subproblems, but still maintains the absence of communication between threads. This widens its application to more general tree search problems such as satisfiablility testing and branch and bound.

In Section 2 we survey the literature on parallelizing reverse search codes. We then describe our general approach in Section 3 and apply it to reverse search in Section 4. We give four examples: generating topological sorts, spanning trees of a graph, high dimensional triangulations and Galton-Watson trees. This latter problem was chosen as the trees generated are unbalanced, which is a challenge for parallelization. They allow an analysis of the major source of overhead in mts and its dependence on the crucial budgeting parameter which will be described in Section 3.

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. Two other important uses of tree search are satisfiability testing and branch and bound. Here the goal is *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 the sharing of information learned during the computation. The mts framework includes support for sharing data between processes and can be applied to these types of problems. As an example we present a parallelization for satisfiability testing in Section 5, demonstrating how little of the original code needs to be changed.

In Section 6 we give computational results for parallelized reverse search and satisfiability codes. This is followed in Section 7 by a discussion of how to evaluate the experimental results, the situation being quite different for enumeration problems and for those problems where pruning is used. For the enumeration problems we get near linear speedup using several hundred cores. For the satisfiability problem we show a substantial improvement in the number of SAT instances that can be solved in a given fixed time period. Finally we give some conclusions and areas for future research in Section 8.

---

[1] In 2008, John White made a list of 130 different applications and implementations, see link at [6].

[2] Version used here available at `https://www-alg.ist.hokudai.ac.jp/~skip/mts/`

## 2. Survey of previous work

The reverse search method, initially developed for vertex enumeration, was extended to a wide variety of enumeration problems [5]. From the outset it was realized that it was eminently suitable for parallelization. In 1998, Marzetta announced his ZRAM parallelization platform [14, 35] which can be used for reverse search, backtracking and branch and bound codes. He successfully used it to parallelize several reverse search and branch and bound codes, including lrs from which he derived the prs code. Load balancing is performed using a variant of what is now known as job stealing. Application codes, such as lrs, were embedded into ZRAM itself leading to problems of maintenance as the underlying codes evolved. Although prs is no longer distributed and was based on a now obsolete version of lrs, it clearly showed the potential for large speedups of reverse search algorithms.

The reverse search framework in ZRAM was also used to implement a parallel code for certain quadratic maximization problems [21]. In a separate project, Weibel [45] developed a parallel reverse search code to compute Minkowski sums. This C++ implementation runs on shared memory machines and he obtains linear speedups with up to 8 processors, the largest number reported.

ZRAM is a general-purpose framework that is able to handle a number of other applications, such as branch and bound and backtracking, for which there are by now a large number of competing frameworks. This is a very large area and an extensive survey of those methods relevant to the present study was given in [8]. We give a brief overview here. Recent papers by Crainic et al. [17], McCreesh et al. [37] and Herrera et al. [25] describe over a dozen such systems. While branch and bound may seem similar to reverse search enumeration, there are fundamental differences. In enumeration it is required to explore the entire tree whereas in branch and bound the goal is to explore as little of the tree as possible until a desired node is found. The bounding step removes subtrees from consideration and this step depends critically on what has already been discovered. Hence the order of traversal is crucial and the number of nodes evaluated varies dramatically depending on this order. Sharing of information is critical to the success of parallelization. These issues do not occur in reverse search enumeration, and so a much lighter wrapper is possible.

Relevant to the heaviness of the wrapper and amount of programming effort required, a comparison of three frameworks is given in [25]. The first, Bob++ [19], is a high level abstract framework, similar in nature to ZRAM, on top of which the application sits. This framework provides parallelization with relatively little programming effort on the application side and can run on a distributed network. The second, Threading Building Blocks (TBB) [42], is a lower level interface providing more control but also considerably more programming effort. It runs on a shared memory machine. The third framework is the Pthread model [15] in which parallelization is deep in the application layer and migration of threads is done by the operating system. It also runs on a shared memory machine. All of these methods use job stealing for load balancing [13]. In [25] these three approaches are applied to a global optimization algorithm. They are compared on a rather small setup of 16 processors, perhaps due to the shared memory limitation of the last two approaches. The authors found that Bob++ achieved a disappointing speedup of about 3 times, considerably slower than the other two approaches which achieved near linear speedup.

A more sophisticated framework for parallelizing application codes over large net-

works of computers is MW that works with the distributed environment of HTCondor[3]. MW is a set of C++ abstract base classes that allow parallelization of existing applications based on the master-worker paradigm [23]. We employ the same paradigm in mts although our load balancing methods are different. MW has been used successfully to parallelize combinatorial optimization problems such as the Quadratic Assignment Problem, see the MW home page for references. Although MW could be used to parallelize reverse search algorithms, we are not aware of any such applications.

## 3. 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 subproblems from the master, run the legacy code, send output to the consumer, and return unfinished subproblems to the master.

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*. This general approach is similar to but simpler than the well-known work-stealing approach [13]. 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:

- small subtrees are explored without being broken up
- large subtrees will be broken up repeatedly
- each worker returns periodically for reassignment, can give information to be passed on to other workers and receive such information
- it is implemented on-the-fly and avoids the duplication of work done in estimation
- it can be varied dynamically during execution to control the job list size
- when used statically and without pruning, the overall job list produced is deterministic and independent of the number of workers

This last item is useful for debugging purposes and also enables a theoretical analysis

[3]Available at `https://research.cs.wisc.edu/htcondor/mw/`

of the job list size under certain random tree models, see [4]. In particular, methods that limit work based on time (such as "begetting" in MW) do not have this property.

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 7.

### *3.1. 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 5.1).

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.

### *3.2. Master process*

The master process begins with initialization, including obtaining an application-provided initial *start_vertex*. It places this initial subproblem in a (new) job list $L$, and then enters the main loop. In this main loop, the master 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.

Pseudocode is given as Algorithm 3 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 the smaller budgets. The default is $lmin = 1, lmax = 3$. In Section 7 we show an example of how $|L|$ typically behaves with these settings.

### *3.3. 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. Pseudocode is given as Algorithm 4 in the Appendix.

### *3.4. 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. Pseudocode is given as Algorithm 5 in the Appendix.

## 4. Applying mts to reverse search

Reverse search is a technique for generating large relatively unstructured sets of discrete objects [5]. 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, $\Delta$, must be known. The performance of the method depends on $G$ having $\Delta$ as low as possible. An adjacency oracle $\mathrm{Adj}(v, j)$ 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 $T$. Pseudocode is given in Algorithm 1 and is invoked by setting *start_vertex* $= v^*$. C implementations for several simple enumeration problems are given at [6]. 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 1 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 1 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 straightforward and given in Algorithm 2, which reduces to Algorithm 1 by deleting the items in red.

To output all nodes in the subtree of $T$ rooted at $v$ we set *start_vertex* $= v$, *max_nodes* $= +\infty$ and *max_depth* $= +\infty$. 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.

### *4.1. Example 1: Topological sorts*

A C implementation (*per.c*) of the reverse search algorithm for generating permutations is given in the tutorial [6]. 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 [6]. 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 [6].

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 1 to an implementation of Algorithm 2 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 using a header file from 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.

**Algorithm 1** Reverse Search

```
1:  procedure RS(start_vertex)
2:      v ← start_vertex
3:      j ← 0   depth ← 0
4:      repeat
5:          while j < Δ do
6:              j ← j + 1
7:              if f(Adj(v, j)) = v
    then
8:                  v ← Adj(v, j)
9:                  j ← 0
10:                 depth ← depth + 1
11:                 output (v)
12:             end if
13:         end while
14:         if depth > 0 then
15:             (v, j) ← f(v)
16:             depth ← depth − 1
17:         end if
18:     until depth = 0 and j = Δ
19: end procedure
```

**Algorithm 2** Budgeted Reverse Search

```
1:  procedure BRS(start_vertex, max_depth,
    max_nodes)
2:      v ← start_vertex
3:      j ← 0   depth ← 0   count ← 0
4:      repeat
5:          unexplored ← false
6:          while j < Δ and
7:                unexplored = false  do
8:              j ← j + 1
9:              if f(Adj(v, j)) = v then
10:                 v ← Adj(v, j)
11:                 j ← 0
12:                 count ← count + 1
13:                 depth ← depth + 1
14:                 if count ≥ max_nodes or
                       depth = max_depth then
15:                     unexplored ← true
16:                 end if
17:                 output (v, unexplored)
18:             end if
19:         end while
20:         if depth > 0 then      ▷ backtrack step
21:             (v, j) ← f(v)
22:             depth ← depth − 1
23:         end if
24:     until depth = 0 and j = Δ
25: end procedure
```

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.

### *4.2. Example 2: Spanning trees*

In the tutorial [6] 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 [6].

### *4.3. Example 3: Triangulations*

The previous examples are from the tutorial on reverse search; a more substantial application of mts is the parallel enumeration of triangulations implemented in mptopcom [29]. This was the first parallel code for the problem, and has already found applications [30, 43]. Here we give a brief summary, see [29] for details of mptopcom and [18] for background on triangulations.

Enumerating triangulations in the plane is one of the early applications of reverse search [5]. Imai et al. [28] later gave a reverse search algorithm for enumerating triangulations in general dimensions, including techniques for restricting to regular triangulations and enumerating up to symmetry. However, the standard tool used for this is TOPCOM [41], which essentially performs a breadth-first search of the flip graph.

mptopcom is an implementation of the algorithm by Imai et al. [28] with some improvements, parallelized using mts. It uses TOPCOM for triangulations and flips and polymake [22] for basic data types. The single-threaded version is competitive with other codes for this problem, while the parallel version allows one to enumerate triangulations that are (in practice) beyond reach of the other codes; see [29] for detailed experimental results.

### *4.4. Example 4: Galton Watson trees*

The previous examples give reverse search trees that are rather well balanced. By that we mean that the height of the tree is either logarithmic or polylogarithmic in its size, i.e. the number of nodes in the tree. Balanced trees are especially suited to parallelization and this is indeed the case for mts as we will see in the experimental results in Section 6. To see how well budgeting works for non-balanced trees Avis and Devroye [4] studied its behaviour on a family of random trees with depth roughly the square root of their size. They also studied how the job list size $L$ depends on the budget parameter $b$. We review the main result.

A Galton-Watson (or Galton-Watson-Bienaymé) tree is a rooted random ordered tree that is a classical statistical model of a birth and death process. Each node independently generates a random number of children drawn from a fixed offspring distribution $\xi$. The distribution of $\xi$ defines the distribution of $T$, a random Galton-Watson tree. They consider critical Galton-Watson trees which are those having $\mathbb{E}\{\xi = 1\}$, and $\mathbb{P}\{\xi = 1\} < 1$. In addition, they assume that the variance of $\xi$ is finite (and hence, nonzero).

Moon [39] and Meir and Moon [38] defined the *simply generated trees* as ordered labelled trees of size $n$ that are all equally likely given a certain pattern of labeling for each node of a given degree. The most important examples include the Catalan trees (equiprobable binary trees), the equiprobable $k$-ary trees, equiprobable unary-binary trees (ordered trees with up to two children), random Motzkin trees, ran-

dom planted plane trees (equiprobable ordered trees of unlimited degrees) and Cayley trees (equiprobable unordered rooted trees). It turns out that all these trees can be represented as critical Galton-Watson trees conditional on their size, $n$, a fact first pointed out by Kennedy [32] and further developed by others. For example, when $\xi$ is 0 or 2 with probability 1/4, and 1 with probability 1/2, we obtain the uniform binary (Catalan) tree. Uniformly random full binary trees are obtained by setting $\mathbb{P}\{\xi = 0\} = \mathbb{P}\{\xi = 2\} = 1/2$. A uniformly random $k$-ary tree has its offspring distributed as a binomial $(k, 1/k)$ random variable. A uniform planted plane tree is obtained for the geometric law $\mathbb{P}\{\xi = i\} = 1/2^{i+1}$, $i \geq 0$. When $\xi$ is Poisson of parameter 1, one obtains (the shape of) a random rooted labeled (or Cayley) tree. For $\xi$ uniform on $\{0, 1, 2, \ldots, k\}$, $T_n$ is like a uniform ordered tree with maximal degree of $k$. All such trees can be dealt with at once in the Galton-Watson framework.

In [4] an analysis is performed on the expected size of the job list as a function of the budget parameter $b$. They prove that if $T_n$ is a Galton-Watson tree of size $n$ determined by $\xi$, where $\mathbb{E}\{\xi\} = 1$ variance $0 < \sigma^2 < \infty$, then

$$\frac{L_n}{n} \to \sqrt{\frac{\pi\sigma^2}{8b}} \tag{1}$$

in probability as $n \to \infty$, where $b$ is the budget and $L_n$ is the job list size. For example, for the Catalan trees we have $\sigma^2 = 3/2$ and so the constant on the right hand side of (1) is $\sqrt{3\pi/16b}$. With a budget $b = 5000$ this indicates that about 1% of the nodes are returned unexplored to the job list. Jobs returned to $L_n$ are the main source of overhead in mts, so a value of about 1% is very satisfactory in practice.

Experimental results are given in [4] for various Galton-Watson trees to show that the estimate in (1) is quite accurate.

## 5. Applying mts to satisfiability

Boolean satisfiability (SAT) asks us to determine the existence of (or find) satisfying assignments for propositional formulas, see [12] 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 [10], there has long been interest in parallel SAT solvers for hard instances. Despite the many challenges [24, 31] in parallel SAT, there are recent successes [26].

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., [1] for a recent example) or one uses the portfolio approach and runs many sequential solvers on the original problem (e.g., plingeling [11]). In either case, a major issue is determining which learnt clauses[4] 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., [1] and treengeling [11]) work on these subproblems subject to some budget, and hard subproblems can be split again. Cube-and-conquer [27] is another

[4]CDCL solvers learn clauses during the search, pruning the search space. See, e.g., Chapter 4 of [12].

recent approach that uses look-ahead solvers to divide the search space for CDCL solvers.

### *5.1.* 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 [20] 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 [11]. We avoided more sophisticated approaches to sharing clauses [1, 3], 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 [1, 11] 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 4 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 [12]) 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 [24].

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, and so we also parallelize Glucose 3.0. One could easily support workers using a mix of solvers, a hybrid of the divide-and-conquer and portfolio approaches to parallel SAT.

## 6. 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 [6].

### *6.1. Topological sorts:* mtopsorts

The tests were performed using the following codes:

- VR: obtained from [16], generates topological sorts in lexicographic order via the Varol-Rotem algorithm [44] (Algorithm V in Section 7.2.1.2 of [34]);
- Genle: also obtained from [16], generates topological sorts in Gray code order using the algorithm of Pruesse and Rotem [40];
- btopsorts: derived from the reverse search code *topsorts.c* [6] as described in Section 4.1;
- 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[5]:

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

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 [34]) and the third is a complete bipartite graph.

Table 1.: Topological sorts: mai32, times in secs

| Graph | m nodes | n edges | No. of perms | VR | Genle | btopsorts | mtopsorts | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 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 |

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.

### *6.2. Spanning trees:* mtree

The tests were performed using the following codes:

[5] Performance for budgeted reverse search seems quite stable for a range of reasonable parameters. See Section 7 below, or Section 6.4 of [8], for experimental results and [4] for analysis.

- grayspan: Knuth's implementation [33] 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." [33];
- btree: derived from the reverse search code *tree.c* [6] as described in Section 4.2;
- mtree: mts parallelization of btree.

Both grayspan and grayspspan are described in detail in Knuth [34]. Again all codes were used in count-only mode and with the default parameters (2). 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 [34]: $P_5C_5$ appears therein and the other graphs are larger versions of examples in that table.

Table 2.: Spanning tree generation: mai32, times in secs

| Graph | m nodes | n edges | No. of trees | grayspan | grayspspan | btree | mtree | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 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 |

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.

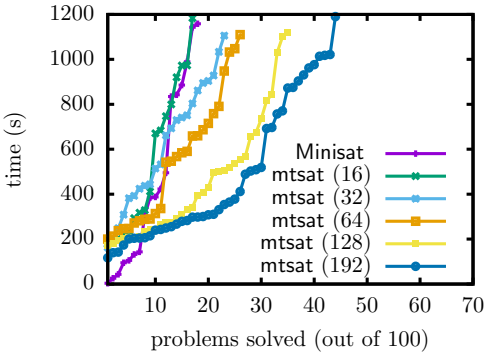
### *6.3. Satisfiability*

The tests were performed using the following codes:

- Minisat: version 2.2.0, classic sequential solver [20];
- Glucose: version 3.0, sequential solver [2] derived from Minisat;
- mtsat: parallel solver using mts and Minisat 2.2.0;
- mtsat-glucose: parallel solver using mts and Glucose 3.0;
- Glucose-Syrup: version 4.0, parallel (shared memory) solver;
- lingeling, treengeling: version bbc, sequential and (shared memory) parallel solvers [11].

Benchmarking parallel SAT solvers is challenging [24] 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 [1], i.e. the 100 instances in the parallel track

of SAT Race 2015 [9] 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 [1]. As noted by [10], 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.



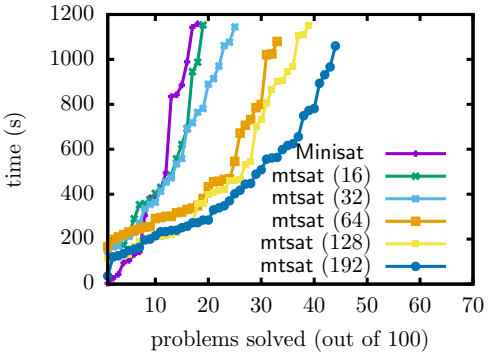

(a) Instances solved vs time

| 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 (2))

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 [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 |

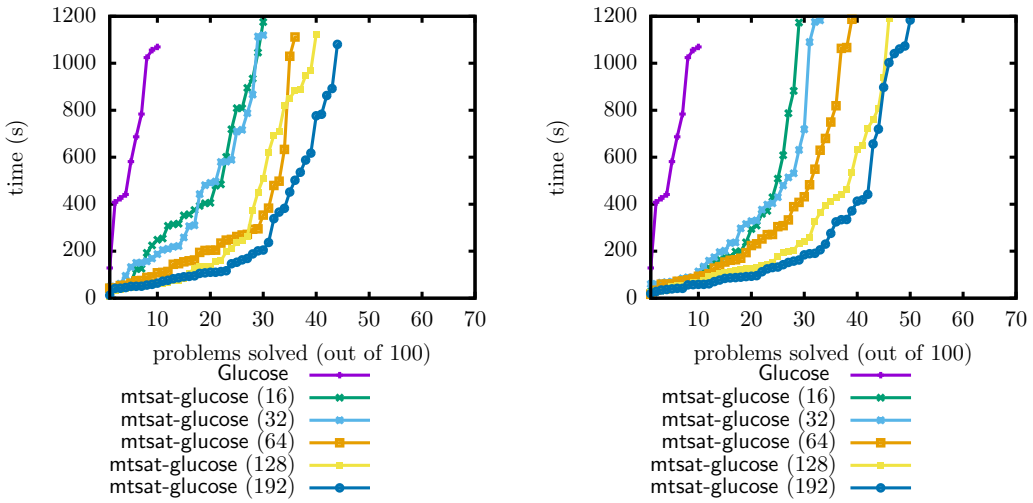
(b) Instances solved within 1200s

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

All non-timeout outputs are correct, and the 32-core run with conflict budgeting solves problem `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.

Along these lines, we also report results applying mts to parallelize Glucose. Figures 3 and 4 show results using the default decision budgeting and also conflict budgeting. Note that like in mtsat, only unit clauses are shared in mtsat-glucose. A more sophisticated approach to sharing learnt clauses, e.g. sharing glue clauses, would likely help

performance.




(a) Decision budgeting with default parameters (2)

(b) Conflict budgeting with $max\_nodes = 10000$, $scale = 10$

Figure 3.: mtsat-glucose performance: Instances solved vs time

| Solver | SAT | UNSAT | Total |
|---|---|---|---|
| Glucose | 6 | 5 | 11 |
| mtsat-glucose (16) | 26 | 5 | 31 |
| mtsat-glucose (32) | 26 | 5 | 31 |
| mtsat-glucose (64) | 31 | 6 | 37 |
| mtsat-glucose (128) | 32 | 9 | 41 |
| mtsat-glucose (192) | 35 | 10 | 45 |
| Glucose-Syrup (32) | 32 | 18 | 50 |
| Glucose-Syrup (64) | 31 | 18 | 49 |

(a) Decision budgeting with default parameters (2)

| Solver | SAT | UNSAT | Total |
|---|---|---|---|
| Glucose | 6 | 5 | 11 |
| mtsat-glucose (16) | 23 | 7 | 30 |
| mtsat-glucose (32) | 27 | 7 | 34 |
| mtsat-glucose (64) | 30 | 10 | 40 |
| mtsat-glucose (128) | 34 | 13 | 47 |
| mtsat-glucose (192) | 38 | 13 | 51 |

(b) Conflict budgeting with $max\_nodes = 10000$, $scale = 10$

Figure 4.: mtsat-glucose performance: Instances solved within 1200s

In all cases, we see that additional cores allow one to solve more problems given a fixed amount of time. While it is likely that performance can be improved by tuning and a better approach to sharing clauses, these results suffice for our purpose: to show that one can easily parallelize a legacy code with mts.

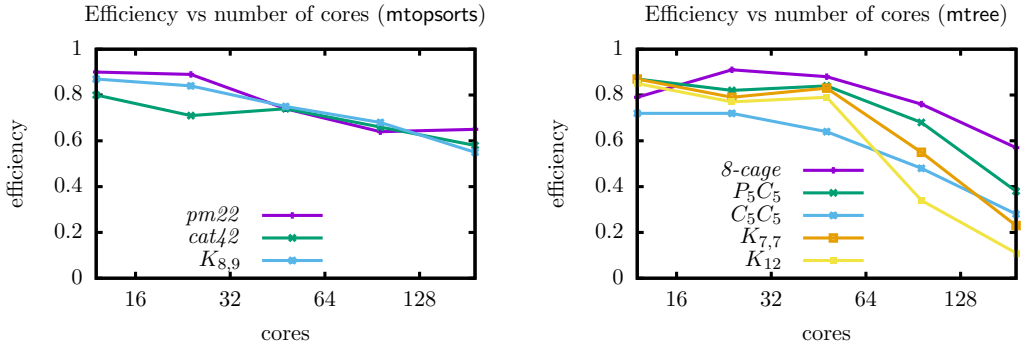
As mentioned earlier, conflict budgeting is most common in this kind of parallel solver. This is because generating a conflict clause guarantees at least some progress has been made and instances can have very different and enormous numbers of variables. Conflict budgeting usually slightly outperformed decision budgeting in the runs here, however this was not universal. Given the minimal tuning for both budget types, our results do not show a clear difference regarding how to budget.

## 7. Evaluating and improving performance

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

$$\text{efficiency} \;=\; \frac{\text{single core running time}}{\text{number of cores} * \text{multicore running time}} \tag{3}$$

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 5 we plot the efficiencies obtained by mtopsorts and mtree for the runs shown in Tables 1 and 2 respectively.




(a) Efficiency with mtopsorts

(b) Efficiency with mtree

Figure 5.: Efficiency vs number of cores (data from Tables 1 and 2)

The amount of work contained in a subproblem can vary dramatically. mts can produce histograms to help understand and tune performance. We discuss three here: processor usage, job list size and distribution of subproblem sizes. Figure 6 shows the first two histograms for the mtopsorts run on $K_{8,9}$ with default parameters (2).

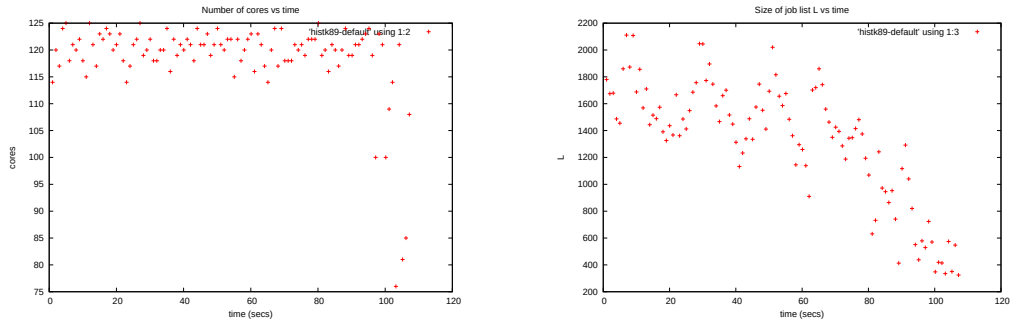



Figure 6.: Histograms for mtopsorts on $K_{8,9}$: busy workers (left) job list size (right)

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

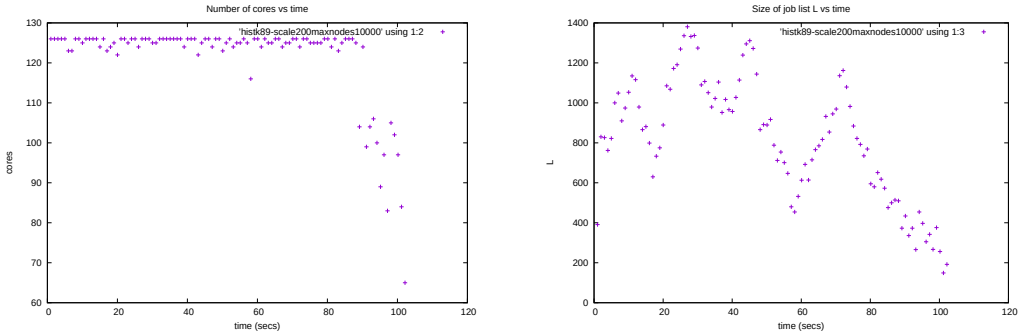



Figure 7.: Histograms with *-scale* 200 *-maxnodes* 10000 on $K_{8,9}$: busy workers (l), joblist size (r)

Figure 7 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 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.

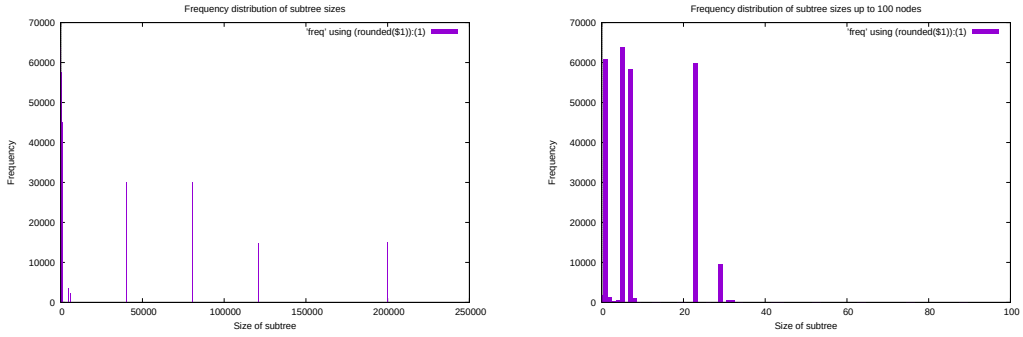



Figure 8.: Subproblem sizes for $K_{8,9}$: all (left) small subproblems only (right)

Figure 8 shows the distribution of subproblem sizes that was produced in a run of mtopsorts on $K_{8,9}$ with default parameters (2). $L$ 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 $K_{8,9}$.

## 8. Conclusions

We have presented a generic framework for parallelizing certain tree search codes requiring only minimal changes to the legacy codes. 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. There is no separate library to install and just a few routines need to be inserted in a user's existing library. Applying mts to four reverse search codes we obtained comparable results to that previously obtained by the customized mplrs wrapper applied to the the lrs code [8]. 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. Other ongoing work involves using mts to parallelize existing integer programming solvers that use the branch-and-bound approach.

## Funding

This work was partially supported by JSPS Kakenhi Grants 16H02785, 18K18027, 23700019 and 15H00847, Grant-in-Aid for Scientific Research on Innovative Areas, 'Exploring the Limits of Computation (ELC)'.

## References

[1] G. Audemard, J.M. Lagniez, N. Szczepanski, and S. Tabary, *An Adaptive Parallel SAT Solver*, in *CP*, LNCS Vol. 9892. 2016, pp. 30–48.

[2] G. Audemard and L. Simon, *Predicting Learnt Clauses Quality in Modern SAT Solvers*, in *IJCAI*. 2009, pp. 399–404.

[3] G. Audemard and L. Simon, *Lazy Clause Exchange Policy for Parallel SAT Solvers*, in *SAT*, LNCS Vol. 8561. 2014, pp. 197–205.

[4] D. Avis and L. Devroye, *An analysis of budgeted parallel search on conditional Galton-Watson trees*, arXiv:1703.10731 (2017).

[5] D. Avis and K. Fukuda, *Reverse search for enumeration*, Discrete Applied Mathematics 65 (1993), pp. 21–46.

[6] D. Avis and C. Jordan, *Reverse search: tutorials* (2000,2016). `http://cgm.cs.mcgill.ca/~avis/doc/tutorial`.

[7] D. Avis and C. Jordan, *A parallel framework for reverse search using mts*, arXiv:1610.07735 (2016).

[8] D. Avis and C. Jordan, *mplrs: A scalable parallel vertex/facet enumeration code*, Math. Program. Comput. 10 (2018), pp. 267–302.

[9] T. Balyo, A. Biere, M. Iser, and C. Sinz, *SAT Race 2015*, Artificial Intelligence 241 (2016), pp. 45–65.

[10] T. Balyo, M.J. Heule, and M. Järvisalo, *SAT Competition 2016: Recent Developments*, in *AAAI*. 2017.

[11] A. Biere, *Lingeling and Friends Entering the SAT Challenge 2012*, in *Proc. SAT Challenge 2012*, Department of Computer Science Series of Publications B, University of Helsinki Vol. B-2012-2. 2012, pp. 33–34.
[12] A. Biere, M.J. Heule, H. van Maaren, and T. Walsh (eds.), *Handbook of Satisfiability*, IOS Press, 2009.
[13] R.D. Blumofe and C.E. Leiserson, *Scheduling multithreaded computations by work stealing*, Journal of the ACM 46 (1999), pp. 720–748.
[14] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, *The parallel search bench ZRAM and its applications.*, Ann. Oper. Res. 90 (1999), pp. 45–63.
[15] L.G. Casado, J.A. Martínez, I. García, and E.M.T. Hendrix, *Branch-and-bound interval global optimization on shared memory multiprocessors*, Optimization Methods and Software 23 (2008), pp. 689–701.
[16] *Combinatorial Object Server : Linear extensions.* `http://theory.cs.uvic.ca/inf/pose/LinearExt.html`.
[17] T.G. Crainic, B. Le Cun, and C. Roucairol, *Parallel Branch-and-Bound Algorithms*, in *Parallel Combinatorial Optimization*, John Wiley & Sons, Inc. (2006), pp. 1–28.
[18] J.A. De Loera, J. Rambau, and F. Santos, *Triangulations*, Algorithms and Computation in Mathematics Vol. 25, Springer-Verlag, 2010.
[19] A. Djerrah, B. Le Cun, V.D. Cung, and C. Roucairol, *Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods*, in *2006 15th IEEE International Conference on High Performance Distributed Computing*. 2006, pp. 369–370.
[20] N. Eén and N. Sörensson, *An Extensible SAT-solver*, in *SAT*, LNCS Vol. 2919. 2003, pp. 502–518.
[21] J. Ferrez, K. Fukuda, and T. Liebling, *Solving the fixed rank convex quadratic maximization in binary variables by a parallel zonotope construction algorithm*, European Journal of Operational Research 166 (2005), pp. 35–50.
[22] E. Gawrilow and M. Joswig, `polymake`*: a framework for analyzing convex polytopes*, in *Polytopes—combinatorics and computation (Oberwolfach, 1997)*, DMV Sem. Vol. 29, Birkhäuser, Basel, 2000, pp. 43–73.
[23] J.P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, *Master–worker: An enabling framework for applications on the computational grid*, Cluster Computing 4 (2001), pp. 63–70.
[24] Y. Hamadi and C.M. Wintersteiger, *Seven challenges in parallel SAT solving*, in *AAAI*. 2012, pp. 2120–2125.
[25] J.F.R. Herrera, J.M.G. Salmerón, E.M.T. Hendrix, R. Asenjo, and L.G. Casado, *On parallel branch and bound frameworks for global optimization*, Journal of Global Optimization (2017), pp. 1–14.
[26] M.J. Heule, O. Kullmann, and V.W. Marek, *Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer*, in *SAT*, LNCS Vol. 9710. 2016, pp. 228–245.
[27] M.J. Heule, O. Kullmann, S. Wieringa, and A. Biere, *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*, in *HVC*, LNCS Vol. 7261. 2012.
[28] H. Imai, T. Masada, F. Takeuchi, and K. Imai, *Enumerating triangulations in general dimensions*, International Journal of Computational Geometry & Applications 12 (2002), pp. 455–480.
[29] C. Jordan, M. Joswig, and L. Kastner, *Parallel enumeration of triangulations*, Electronic Journal of Combinatorics 25 (2018), p. P3.6.
[30] M. Joswig and L. Kastner, *New Counts for the Number of Triangulations of Cyclic Polytopes*, in *ICMS*, LNCS Vol. 10931. 2018, pp. 264–271.
[31] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon, *Resolution and Parallelizability: Barriers to the Efficient Parallelization of SAT Solvers*, in *AAAI*. 2013, pp. 481–488.
[32] D.P. Kennedy, *The Galton-Watson process conditioned on the total progeny*, Journal of Applied Probability 12 (1975), pp. 800–806.
[33] D.E. Knuth, *Programs to read.* `http://www-cs-faculty.stanford.edu/~uno/programs.html`.
[34] D.E. Knuth, *The Art of Computer Programming, Volume 4A*, Addison-Wesley Profes-

sional, 2011.
[35] A. Marzetta, *ZRAM: A library of parallel search algorithms and its use in enumeration and combinatorial optimization*, Ph.D. diss., Swiss Federal Institute of Technology Zurich, 1998.
[36] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed., Addison-Wesley Professional, 2004.
[37] C. McCreesh and P. Prosser, *The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound*, ACM Transactions on Parallel Computing 2 (2015), pp. 8:1–8:27.
[38] A. Meir and J.W. Moon, *On the altitude of nodes in random trees*, Canadian Journal of Mathematics 30 (1978), pp. 997–1015.
[39] J. Moon, *Counting Labelled Trees*, Canadian mathematical monographs, 1970.
[40] G. Pruesse and F. Ruskey, *Generating the linear extensions of certain posets by transpositions*, SIAM Journal on Discrete Mathematics 4 (1991), pp. 413–422.
[41] J. Rambau, *TOPCOM: Triangulations of Point Configurations and Oriented Matroids*, in *Mathematical Software—ICMS 2002*. World Scientific, 2002, pp. 330–340. Available at `http://nbn-resolving.de/urn:nbn:de:0297-zib-6849`.
[42] J. Reinders, *Intel Threading Building Blocks*, O'Reilly & Associates, Inc., 2007.
[43] B. Schröter, *Multi-splits and tropical linear spaces from nested matroids*, Discrete & Computational Geometry 61 (2019), pp. 661–685.
[44] Y.L. Varol and D. Rotem, *An algorithm to generate all topological sorting arrangements*, The Computer Journal 24 (1981), pp. 83–84.
[45] C. Weibel, *Implementation and Parallelization of a Reverse-Search Algorithm for Minkowski Sums*, in *2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2010, pp. 34–42.

## Appendix A. Pseudocode

**Algorithm 3** 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 ← 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 · lmin then
10:                maxd ← max_depth
11:            else
12:                maxd ← ∞
13:            end if
14:            if |L| > size · lmax then
15:                node_budget ← scale · max_nodes
16:            else
17:                node_budget ← 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 4** 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 5** 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
```