DAGSTER

- DRAFT IN PROGRESS -
A Parallel Structured SAT Solver
Progress Report Against Project Activities

MARK BURGESS, CHARLES GRETTON,
JOSH MILTHORPE, MARSHALL CLIFFTON, LUKE CROAK

November 16, 2022

ABSTRACT

In this report we update stakeholders on developments since our midterm report, by providing a new record of progress towards goals of the project. In particular we summarise new functionalities/features that have been implemented since our last report, and we outlay several demonstration example runs of Dagster to indicate performance and function.

All Authors: *School of Computing, Australian National University, Canberra, Australia.*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This report outlines some evaluations of Dagster, a SAT solver we have built to solve difficult and/or large SAT problems using HPC infrastructure. Dagster takes as input a sequence of disjunctive clauses and a graphical structure that determines the problem at hand, its compositional structure, and thereby information about how to solve that problem in a distributed computing environment. Dagster proceeds by solving independent subproblems in parallel in an HPC (High Performance Computing) environment, so that eventually the satisfiability of the problem of interest is determined. In the case of satisfiable instances of the SAT problem, the Dagster tool can also be used to count the number of satisfying valuations efficiently using a parallel computing environment. The formula of interest and its compositional structure is described according to a labelled Directed Acyclic Graph (DAG). Labelled vertices index subproblems—i.e. subsets of disjunctive clauses—and directed edges are labelled with small sets of problem variables, parameterising the sets of cubes—i.e. each a conjunctive clause—that define subproblems. There are two opportunities for parallel search here:

1. For each outgoing arc from a source vertex, each satisfying valuation of the source problem poses one subproblem at each destination vertex.

2. In case a destination has multiple incoming arcs, for each logically consistent combination of incoming valuations we have one subproblem at that destination.

In summary, the way the search is distributed across multiple processes in described by the DAG structure, with partial valuations and sets of logically consistent partial valuations determining what search tasks need to be scheduled. This method of solving has several features:

- Search is immediately parallel, as different parts of the problem are resolved across computing cores with minimal overhead.
- Dagsteris designed to take advantage of known substructure within the SAT problem, with different logical components of the problem solved in parallel.
- Large problems do not need to be handled in-memory at-once, allowing for solutions to problems too big to fit in RAM.

There are also several weaknesses of the approach to be aware of:

- The performance of the search is highly sensitive to the quality of the provided problem decomposition. Given an arbitrary formula finding a 'good' decomposition is not necessarily easy.
- The parallel infrastructure that the tool provides has some computational overhead, such that the tool shows worse performance compared to serial baseline solves on small and/or easy problems, where you might expect to solve the problem routinely on a laptop.

In this report we provide an update on the records provided in our midterm report, wherein we reported on some of the features of the Dagster system. We have previously highlighted the following features of Dagster:

- It treats a SAT problem according to a DAG decomposition, taking in a set of disjunctive clauses and a DAG file object which described the problem to be resolved in parallel using the MPI messaging system.
- Features an in-memory and filesystem subproblem breakdown, allowing Dagster to solve CNF problems which are larger than the system has memory.

- An optional hybridisation solving mode between CDCL—particularly the search implemented in the TiniSAT system—and SLS—particularly derived from the gNovelty+ system—allowing the Dagster tool accelerated solving on problems both more suitable for CDCL and also SLS procedures.
- An optimal parallel clause strengthening module, actively simplifying learnt clauses within the CDCL routine.
- A scheduling algorithm that moderates the resolving of subproblem components in an MPI environment, with optional modes tailored to the enumerating all solutions of a SAT problem, and and also tailored to racing to a first solution.
- An optional Binary Decision Diagram (BDD) module storing and compiling knowledge about the solutions to subproblems. This can minimise computational and communication overhead where there might otherwise be combinatorial explosion in memory.

Within these subsystems the Dagster tool has the potential to solve problems in parallel, and a testing suite was designed for verification and validation of the correct execution of this function. In this report we provide some verified use cases and computation experiments about the performance of Dagster, and update stakeholders on developments since the midterm report.

1.1 changelog

Noteworthy software changes since the midterm report include:

- Smaller in-memory storage of DAG information using *RangeSets* for storing indices of subproblem clauses and variables – i.e. to minimise memory bloat for very large DAG instances. This feature was required for the below reported Pentominoes case study, as well as for a range of other problems we have been testing on.
- The addition of an optional, more fragile but quicker, mechanism that skips testing logical compatibility for multiple incoming valuations.
- The addition of solution counting interrupt to workers which allows them to be reallocated to different problems more easily.
- The addition of the ability of Dagster to handle disjoint DAGs, and an additional mode of operation which race to a first solution on each disjoint subgraph. This feature is important for the Property Directed Reachability case study being examined by PhD candidate Marshall Clifton.
- An additional solution simplifying loop inside CDCL processes to minimise the number of variables in reported (sub-)problem solutions. This feature is important for the symbolic execution workflows being examined by honours student Luke Croak.
- Added an fast process of parsing and splitting a CNF into subproblems without having to load it entirely into RAM - allows processing of large CNF files that would otherwise exhaust the RAM to load.
- Very many performance improvements were implemented, and a few critical bugs were discovered and resolved.

2 MODEL COUNTING IN PARALLEL – HARD SATISFIABLE RANDOM FORMULAE

Here, we examine the performance of the tool at counting models in a hard synthetic example that exhibits a single solution. One simple case of verifying the parallel performance of DAGSTER is to test the performance solving small-hard SAT problems which are entirely disjoint. Particularly it is expected that as the number of small-hard problems increases then the time DAGSTER should take in solving these should be a function of the time it takes to solve any single one of the subproblems, and the ratio of the number of multiprocessing cores to the number of subproblems which need to be solved.

The SGEN1 script distributed with DAGSTER was used to generate small-hard subproblems comprising 95 variables and featuring one unique solution. Such problems take our baseline CDCL procedure, TINISAT, about half a second to solve and also prove that no further solutions exists. Our repository includes a custom script for conjoining subproblems of that type into one super-problem described by a accompanying DAG. Figure 1 gives a depiction of this type of super-problem. The performance of DAGSTER at solving this arrangement of subproblems was measured. We examined the wall-time performance of the tool as we scale the number of cores DAGSTER was operating with as a fraction of the time it took a vanilla CDCL procedure in TINISAT to solve the problem without having to also prove that no further solution exists.

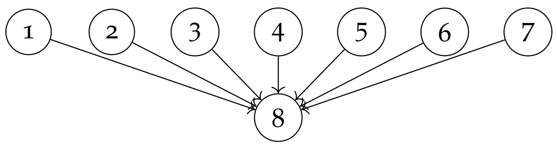


Figure 1: An example DAG for conjoined small-hard problems, where disjoint subproblems are processed in parallel and their solutions joined together at the final node

The run-time of search on super-problems with varying numbers of subproblems, in different computational environments where we vary the number of available cores, is shown in Figures 2 and 3. From Figure 2 we can see that as the number of subproblems increases the time it takes DAGSTER to solve the problem as a fraction of the time it takes the baseline CDCL procedure, TINISAT, to solve the problem decreases – i.e. DAGSTER is significantly faster. Particularly, it is possible to note that the trend-line is approximately hyperbolic, indicating that the computational overhead of using DAGSTER goes to zero as the number of subproblems increases. Also, the speed of DAGSTER is pseudo-linearly increasing for large numbers of subproblems from figure 3 which is expected. The reader should also not that here, DAGSTER is solving a more difficult problem compared to the CDCL baseline. It has to find the one satisfying valuable, and then further prove that no further solutions exists.

We also notice that increasing the number of cores available to DAGSTER solving the problem increases performance, but only upto a certain point. Particularly it was noticed that 64 cores performed significantly worse than 32 cores. This measurement of degrading performance is something we shall investigate in due course, and is likely due to the experimental host being oversubscribed at the time we took the measurement.

The invocation to run the above experiment is to be found by executed the following command within the code repository:

```
python3 ~/summer1819/Benchmarks/cnf_concatenator/run.py
```

SGEN1: http://www.cs.qub.ac.uk/~i.spence/sgen/

Experimenting with with small-hard problems in parallel we were able to verify that Dagster offers significant parallel advantage, as is expected by its design.

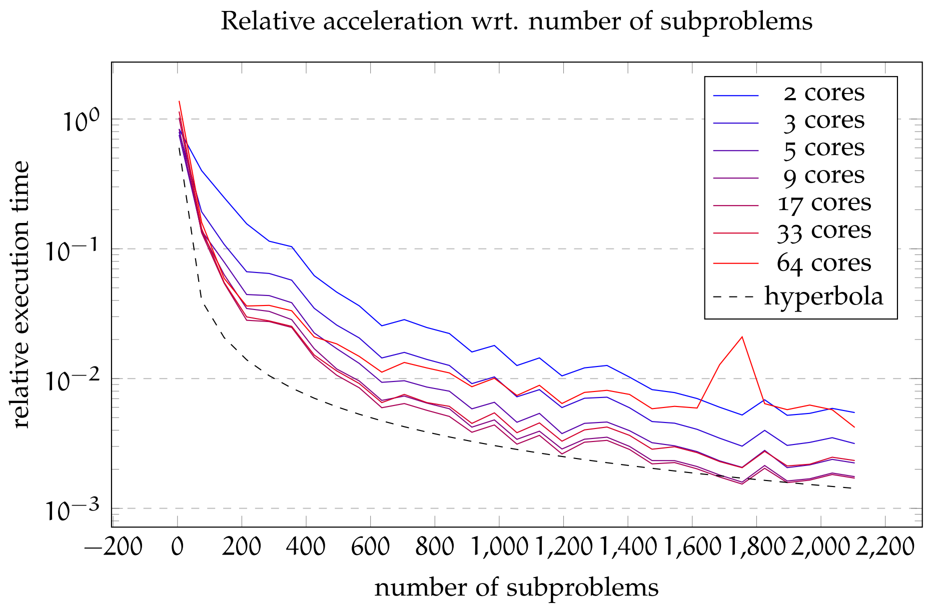


Figure 2: Runtime relative acceleration on paralell small-hard subproblems, relative to Tinisat time, with hyperbolic trendline, for different numbers of cores and subproblems

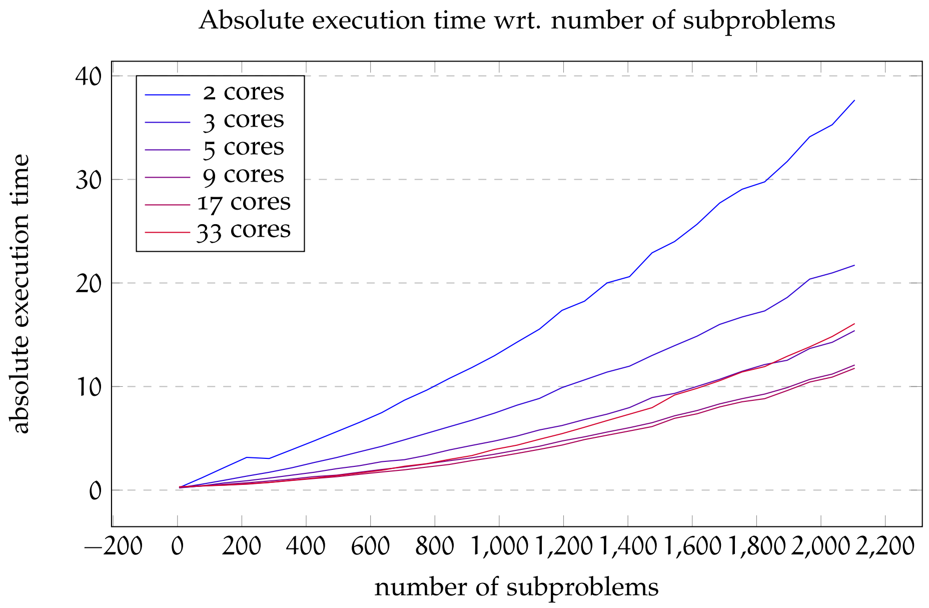


Figure 3: Absolute execution time wrt. small-hard subproblems

3 A LARGE SATISFIABLE PROBLEM – PENTOMINOES

A tiling problem was extended as a benchmark for Dagster from a recreational puzzle solving youtube channel, as shown in Figure 4, where the challenge is to fill a grid with Pentominoes (5 connected blocks) such that no pentomino crosses a boldened black line, and that no pentominoes of the same shape (counting reflections/rotations) touch each other.

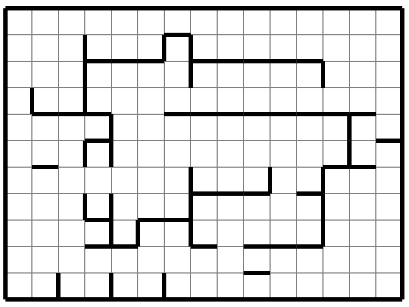


Figure 4: A Pentomino puzzle featured on youtube channel 'Cracking the Cryptic'

A generator of these kind of problems was coded to randomly generate hard 15x15 pentomino problems, involving a process of:

1. Randomly filling a 15x15 grid with pentominoes
2. Bolden the outline of those pentominoes, and removing them
3. Iteratively remove a random boldened line segments while the puzzle is uniquely solvable, until no more such removals are possible.

This process was shown to become prohibitively slow to generate problems larger than 25x25, and so to generate bigger pentomino problems these 15x15 pentomino problems were cascaded side-by-side together in a grid pattern, such as shown in Figure 5. In this way, the grid of pentomino problems constitutes a larger problem which has logically distinct parts and where each subproblem is logically related only to its immediate neighbours: above, below, left and right; and because every pentomino subproblem is uniquely solvable then also this larger pentomino problem is also uniquely solvable.

For these large pentomino problems a DAG would be generated embodying a solution process of solving from the top left diagonally through to the bottom right, as shown in Figure 6. In these particular problems the size of the grid of 15x15 pentomino subproblems would determine the maximum branching of the parallel process and thus the efficiency of the solving in parallel.

For these problems we tested the performance of Dagster for different numbers of processor cores, against the TiniSAT and Lingeling CDCL baselines for different sized problems, the results are shown in figure 7. In this figure, we see that Dagster is upto an order of magnitude faster (in the median) for these problems than serial sat solvers, but that increasing the number of cores does not necessarily improve performance. Specifically we suspect that this is because the DAGs of these problems (such as per instance in figure 6) do not support sufficiently many parallel processing streams to take advantage of higher parallel processing cores.

These Pentomino problems verified the functioning of Dagster in providing speedup due to parallelisation in solving larger structured problems with coupled subproblems.

The invocation to run these experiments is to be found by executed the following command within the code repository:

```
~/summer1819/Benchmarks/Pentomino/runme.sh
```

Cracking the Cryptic, youtube video: youtube.com/watch?v=S2aN-s3hG6Y

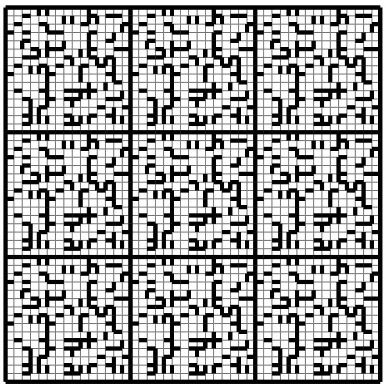


Figure 5: An example 3x3 pentomino superproblem

$$\begin{tikzcd}
1 \arrow[r] \arrow[d] & 2 \arrow[r] \arrow[d] & 4 \arrow[r] \arrow[d] & 7 \arrow[d] \\
3 \arrow[r] \arrow[d] & 5 \arrow[r] \arrow[d] & 8 \arrow[r] \arrow[d] & 11 \arrow[d] \\
6 \arrow[r] \arrow[d] & 9 \arrow[r] \arrow[d] & 12 \arrow[r] \arrow[d] & 14 \arrow[d] \\
10 \arrow[r] & 13 \arrow[r] & 15 \arrow[r] & 16
\end{tikzcd}$$

Figure 6: An example DAG for a 4x4 pentomino superproblem

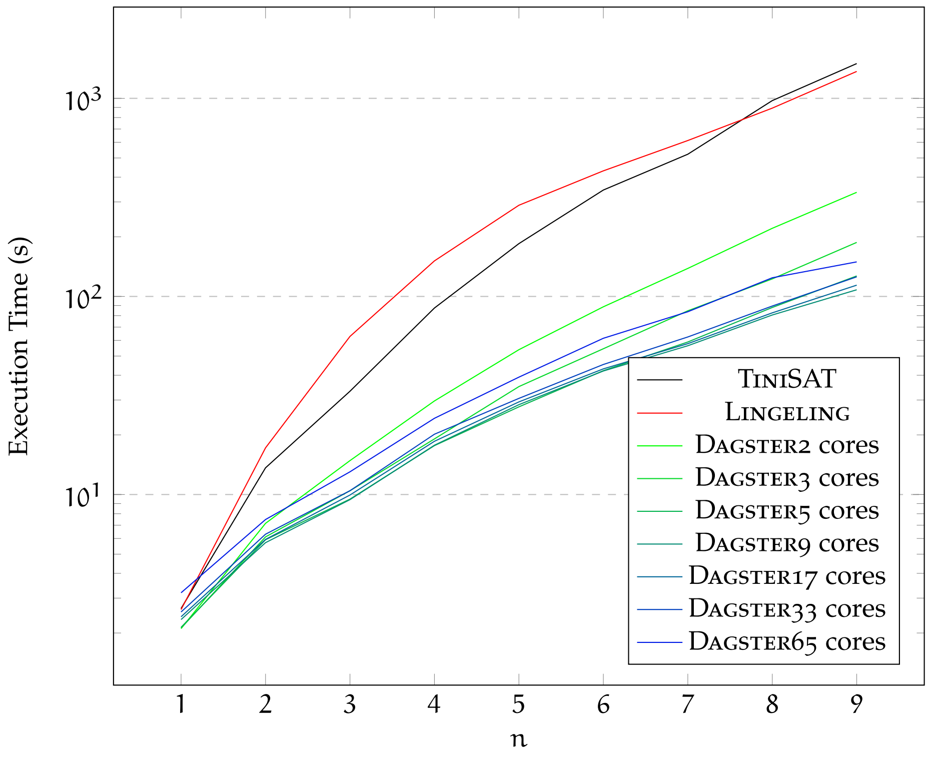


Figure 7: Runtime performance (specifically medians) of DAGSTER against TINISAT and LINGELING solvers for different numbers of cores, across n, for $n \times n$ grid of 15×15 pentomino problems.

4 COUNTING MODELS IN PARALLEL – COSTAS ARRAYS

A Costas array is a set of n points in an nxn array such that each column and row contains exactly one point and each of the $n(n-1)/2$ displacement vectors between the the points are distinct; Costas arrays are well known and have various applications, and the process of using search techniques to solve for them is known to be challenging. Specifically there is an open question about whether any Costas arrays exist notably for sizes 32x32 and 33x33. Searching processes have revealed that there are none of specific sub-classes of Costas arrays those sizes [1], which has invited some to predict that Costas arrays of those sizes exist [2]. Notwithstanding, searching processes has been conducted at least upto size 29x29 [3].

As an example, consider the following 6x6 Costas array in table 1:

		■			
				■	
					■
■					
			■		
	■				

Table 1: An example 6x6 Costas array

from this example Costas array we can see that there is exactly one filled-in cell for every column and row; additionally we can see that the vector displacement between any of the filled-in cells is unique and that there are no two sets of cells that have the same spacing between them. Particularly if we we tabulate all of the $n(n-1)/2$ displacements between pairs of nodes in table 2 we can see that all the displacements values along each of the rows are unique.

	1	2	3	4	5
1	+2	-5	+4	-3	+1
2	-3	-1	+1	-2	
3	+1	-4	+2		
4	-2	-3			
5	-1				

Table 2: The displacements of the example 6x6 Costas array 1, organised by horisontal distance in rows, by horisontal offset in columns

Costas arrays are known to exist for many sizes, and for every Costas array, there are potentially 8 symmetry mappings of the same array that are also Costas arrays (by rotations and flip, ie. the dihedral group), consider numbers of Costas arrays by size given by online encyclopedia of integer sequences (OEIS): `https://oeis.org/A008404` and `https://oeis.org/A001441`.

The investigative question therefore is: if we can encode the Costas problem into SAT, and then decompose the resulting SAT problem for accelerated solving using Dagster. Particularly the SAT encoding of the Costas problem was produced with optional symmetry breaking constraints to break dihedral mappings, particularly with lex-leader symmetry breaking, and a more simplified and less total symmetry breaking as found in [2]. From this SAT problem we considered primarily two different ways of decomposing the SAT problem.

1. Inspired by the construction of such tables as table 2, we considered a decomposition of iteratively constructing Costas arrays from the bottom of the table up. This decomposition proved to be poor performing as a combinatorial number of intermediate solutions between depths were created
2. Taking a much simpler approach of decomposing the problem into two parts, the first being placing the first two columns of the Costas array, and the second

part of the problem being that the rest of the Costas array would be filled in. This proved to be more effective decomposition of the problem. See Figure 8.

$$\begin{tikzcd} \boxed{0} \arrow[r] & \boxed{1} \end{tikzcd}$$

Figure 8: A simplified DAG structure for the Costas decomposition

The performance of Dagster at solving Costas problems for different sized Costas arrays is given in Figure 9. In this figure the time taken to count all the Costas arrays of a given size is plotted against the size of the problem for different numbers of cores against the time taken to do the same thing running the serial baseline CDCL procedure implemented in TiniSAT (shown in black). From this figure we can see that for larger sized Costas arrays that Dagster shows significantly improved performance in solving the Costas model counting problem, where the performance increases with the number of cores. However it is also noticed that for smaller and easier Costas problems (of size $n x n$ where $n < 10$) that the parallel overhead of using Dagster is the primary determinant of the solution time, where the greater the number of cores the larger the overhead and the slower the process.

With this simulation we can verify that Dagster can be used to speedup solving performance on the types of problems which are geometrically difficult and resemble research-interesting questions.

The Costas simulation that produced the data shown in Figure 9 can be invoked from the Dagster repository by executing:

```
/summer1819/Benchmarks/run.sh
```

Where the script compiles and invokes the 'generate-costas-N' program from inside that same directory.

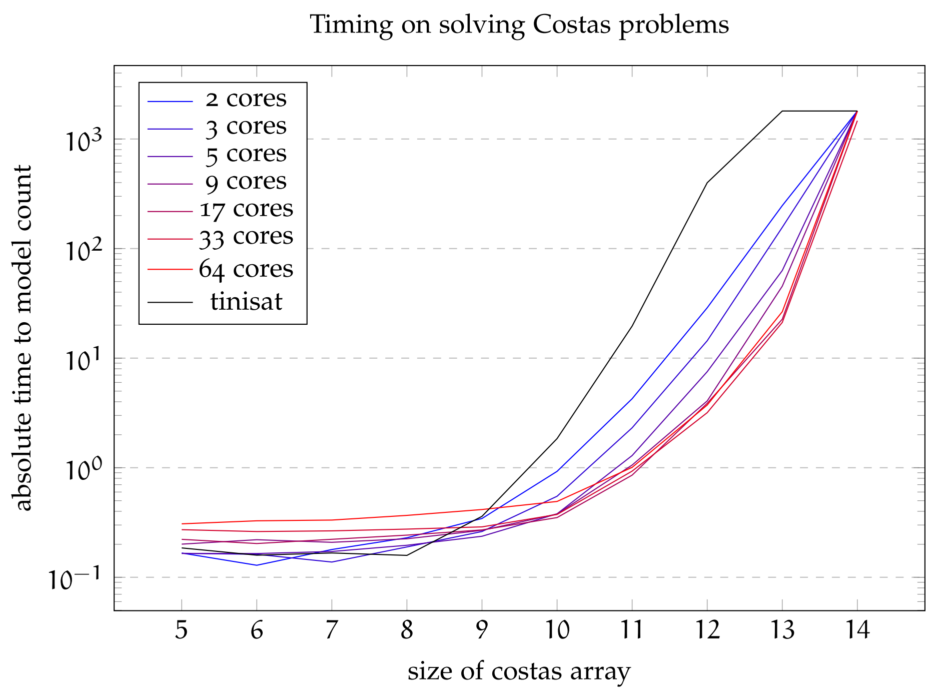


Figure 9: Runtime to solve Costas arrays, by core count and costas array size w/ 1800s timeout

5 ACCELERATING UNSATISFIABILITY PROOFS VIA PARALLELISATION

Empirical studies of the Boolean satisfiability problem late last century identified and studied a notion of empirically *hard* problems. The earliest works studied formulae occurring in conjunctive normal form with all disjunctive clauses having a fixed length k. A range of studies of this "k-*Satisfiability*" problem have been undertaken treating different values of k, and other more flexible concepts of structural invariants. Taking $k = 3$ and studying sets of pseudo-random problems researchers found that so-called "hard" problems occur when the ratio of the count of clauses to the count of problem variables is approximately 4.26 [4]. Here we present a small study of pseudo-random 3-Satisfiability using the problem distributions associated with Dubois et al. (2000)[5].

Somewhat trivially—i.e. primarily due to a portfolio effect—the hybrid search implemented in Dagster outperforms CDCL baselines in hard satisfiable random instances. For satisfiable 3-Satisfiability the best solution procedure is a local search [6]. Our hybrid configuration of Dagster allows any number of such searches to be scheduled to run more-or-less independently in parallel on your cluster. Thus, not only is this parallel system better than CDCL here, courtesy of implementing a local search, it is also faster than running a serial local search procedure, because the expected walltime to a solution being emitted is at the lower end of the runtime distribution of that stochastic procedure. Here, we focus on the more interesting setting, accelerating search in a family of unsatisfiable 3-Satisfiability problems.

In our first result, we accelerate the walltime performance of the default implementation of CDCL in TiniSAT, by leveraging the hybrid search mode, using the local search to generate a complete set of relatively easy UNSAT subproblems that can be solved independently in parallel. A comparison of runtime of Dagster as compared with the serial CDCL procedure in TiniSAT is given in Figure 10. Specifically, we can decompose the problem using a 2 node DAG in which the "source" node indexes 94% of clauses, and this communicates solutions to a "sink" node that includes all the clauses from the concrete problem at hand. What is communicated is a small prefix of the total subproblem valuation. The problem at the source node, with only 94% of clauses, is easily satisfiable using a local search, and indeed the hybrid procedure can quickly enumerate all solutions to that subproblem and subsequently the CDCL procedure can prove that no further solutions exists. The problem of proving that the concrete problem in unsatisfiable, for each of the valuations associated with the source node subproblem, is extremely easy.

In conclusion, our hybrid solver can, by virtue of local search, easily enumerate the solutions to an underconstrained subproblem, and by virtual of parallelism quickly eliminate those solution candidates using systematic search.

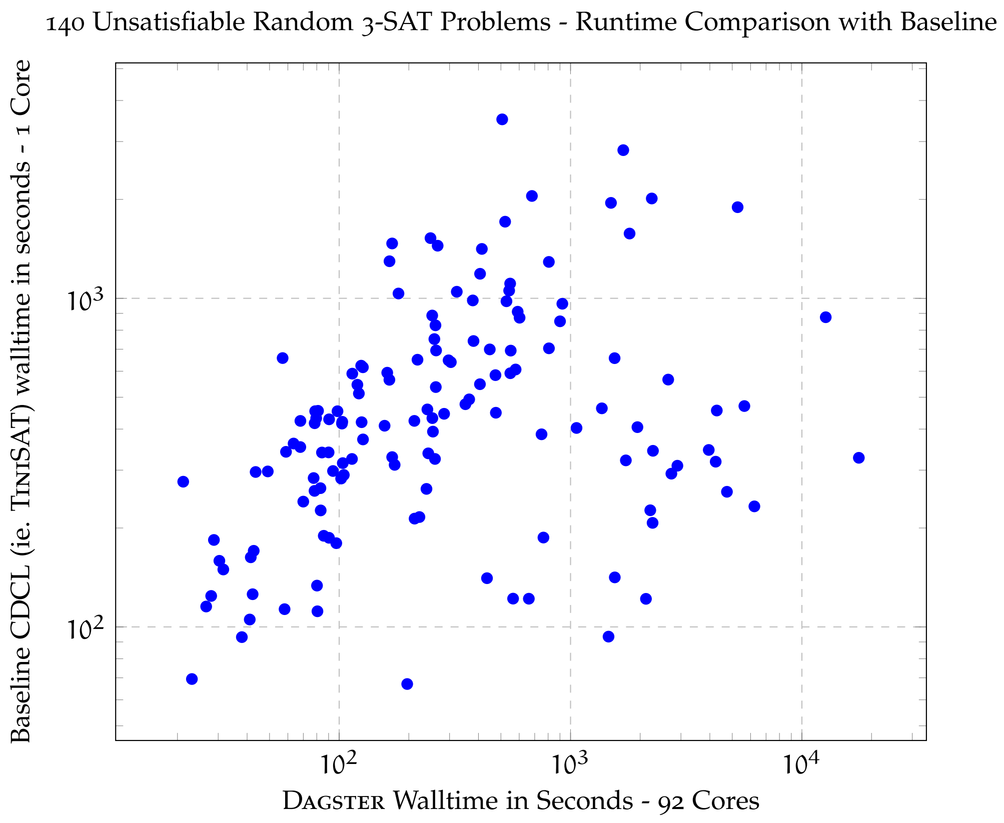


Figure 10: Logscale scatter plot of runtime distributions of TiniSAT (vertical) and Dagster using 92 cores (horizontal) on pseudo-random unsatisfiable 3-Satisfiability problem instances.

6 DECOMPOSING SOFTWARE VERIFICATION PROBLEMS FOR USE IN DAGSTER

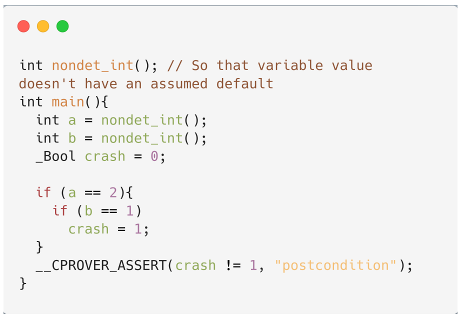
Software verification problems are extremely important and finding a way to decompose them is critical to making proper use of Dagster. We will consider one pipeline that produces SAT problems from C and C++ source code and a preliminary approach which results in a working decomposition for Dagster.

We use the tool CBMC, which is a bounded model checker for C and C++ programs, to generate the SAT problems. CBMC takes a program, which we want to verify certain properties about, and broadly speaking it passes it through 3 stages to result in the SAT formula. We will work through the example code in Figure 11 to illustrate the process.

1. In the first stage the given C/C++ program is converted into a GOTO program, which results in all code that performs a jump of some kind such as if statements, jumps and loops being converted into an equivalent goto statement which optionally has a guard or condition attached to it. The resulting GOTO program will only contain guarded goto statements, assignments, assertions, labels and goto instructions.

2. In the second stage the GOTO program is converted into single static assignment (SSA) form as CBMC performs loop unwinding which is done to a fixed bound. In SSA form variables can only be assigned once and with a special ϕ function are inserted at merge points to determine which of the possible instances of the variable are now being referred to. An example of this conversion can be seen in Figure 12, where ϕ nodes have been expanded into a form that uses the C ternary operator "?:".

3. In the final stage the SSA form is converted into a formula, by turning the assignments into equalities and forming the conjunction of all them as in Figure 13, noting that the property to be checked is negated which in this case means we check to see if crash_5 $= 1$. This formula is then converted into a form amenable for SAT by a process known as bit-blasting, we will not go into the details of that here.



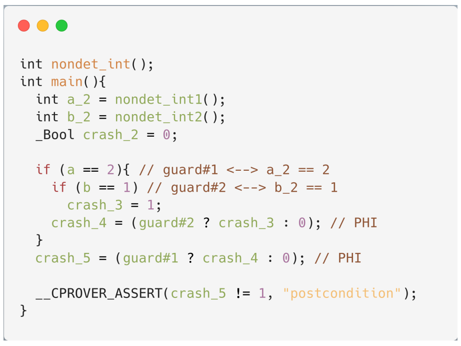
```
int nondet_int(); // So that variable value
doesn't have an assumed default
int main(){
  int a = nondet_int();
  int b = nondet_int();
  _Bool crash = 0;

  if (a == 2){
    if (b == 1)
      crash = 1;
  }
  __CPROVER_ASSERT(crash != 1, "postcondition");
}
```

Figure 11: Example code in C.

We are able to produce a decomposition for this program by considering the formula as it is represented in SSA form. The resulting decomposition that we create will have four nodes, see Figure 14. The first node contains clauses that relate to calculating the goal and determining the values that guard#1 and guard#2 should be to reach that goal. These guard values will be communicated to nodes

CBMC can perform other safety checks in relation to memory and array bounds etc.. Our example will only consider an assertion that the user wants to be satisfied.
Which can be thought of, and are represented as, a conditional "if !X THEN GOTO Y".
The unwinding can be determined manually by the user or left to the program to verify when enough unwinding has been done. We will not go into the details of this here.

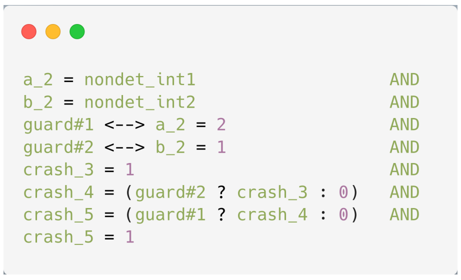


```
int nondet_int();
int main(){
  int a_2 = nondet_int1();
  int b_2 = nondet_int2();
  _Bool crash_2 = 0;

  if (a == 2){ // guard#1 <--> a_2 == 2
    if (b == 1) // guard#2 <--> b_2 == 1
      crash_3 = 1;
    crash_4 = (guard#2 ? crash_3 : 0); // PHI
  }
  crash_5 = (guard#1 ? crash_4 : 0); // PHI

  __CPROVER_ASSERT(crash_5 != 1, "postcondition");
}
```

Figure 12: Conversion of code into SSA form.



```
a_2 = nondet_int1                        AND
b_2 = nondet_int2                        AND
guard#1 <--> a_2 = 2                     AND
guard#2 <--> b_2 = 1                     AND
crash_3 = 1                              AND
crash_4 = (guard#2 ? crash_3 : 0)        AND
crash_5 = (guard#1 ? crash_4 : 0)        AND
crash_5 = 1
```

Figure 13: Resultant formula from conversion.

two and three which contain clauses that calculate the corresponding the values for a and b separately as the variable values are independent of one another. The final node will merge the results of nodes two and three and confirm that they cohere to form a solution.

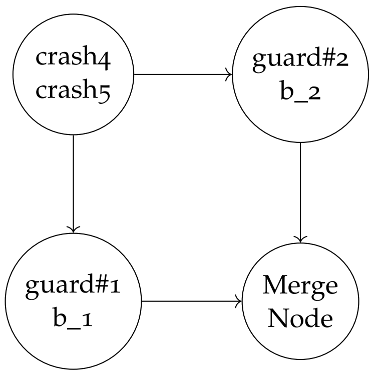
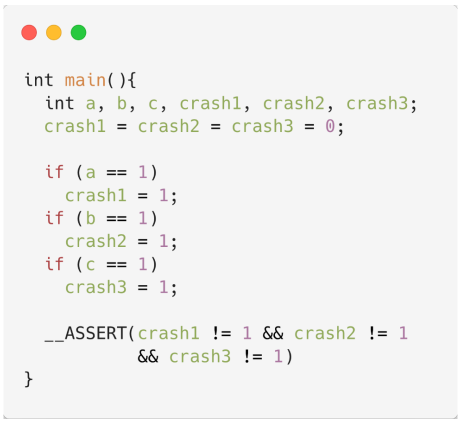


Figure 14: DAG for example code.

Using a similar method we can produce a decomposition for a program with three independent conditionals 15, which will result in three completely independent sub-problems that are embarrassingly parallel 16.

Finally, we are also able to decompose software verification problems which contain loops where variables are independently acted upon 17. This will result in the first node computing the goal as before, and then two child nodes which deal with calculating the resultant values for the independent variables a and b, whose results are then passed onto the merge node cf. 18.

The above decompositions were all produced using the same recipe and were successfully run with DAGSTER. These small examples don't motivate large parallel compute, but we are validating our recipes and running them with DAGSTER which is executing these decompostions in parallel. Research is still ongoing into improving the decomposition method that was used to form the above DAG structures.



```
int main(){
  int a, b, c, crash1, crash2, crash3;
  crash1 = crash2 = crash3 = 0;

  if (a == 1)
    crash1 = 1;
  if (b == 1)
    crash2 = 1;
  if (c == 1)
    crash3 = 1;

  __ASSERT(crash1 != 1 && crash2 != 1
           && crash3 != 1)
}
```

Figure 15: Three independent conditionals.

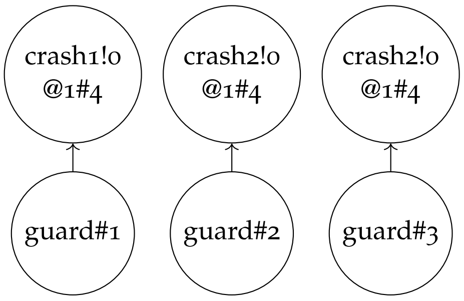
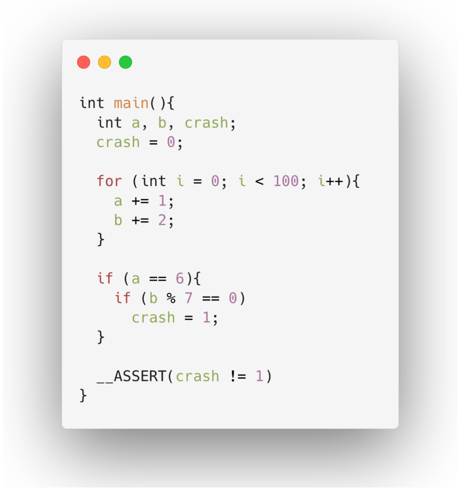


Figure 16: DAG for the problem with three independent conditionals.



```
int main(){
  int a, b, crash;
  crash = 0;

  for (int i = 0; i < 100; i++){
    a += 1;
    b += 2;
  }

  if (a == 6){
    if (b % 7 == 0)
      crash = 1;
  }

  __ASSERT(crash != 1)
}
```

Figure 17: A crash is triggered based on the resulting values of two variables acted upon independently in a loop.

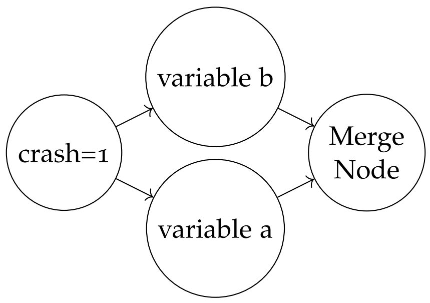


Figure 18: DAG for the problem with a loop.

7 MODEL COUNTING IN PARALLEL – EASY LARGE SATISFIABLE RANDOM FORMULAE

It is also possible to verify that Dagster has steady performance with larger problems, particularly we consider solving large-easy SAT problems which are entirely disjoint and in parallel, similar to the scheme implemented in section 2. Particularly it is verified that as the number of large-easy problems increases then the memory that Dagster should take in solving these should be less than the size of the CNF file of all those problems combined; and in this way it is verifiable that Dagster can solve problems which are larger than the amount of memory on the machine. This is accomplished by splitting the CNF of the larger problem into parts corresponding to subproblems, which are loaded sequentially and solved in turn in accordance with the DAG.

Particularly a 5MB random-7-SAT problem was generated, and multiple copies of the same problem were concatenated to form a parallel DAG - as previously shown in figure 1). In this way the maximum size of any subproblem was 5MB, whereas the CNF of the problem would be many multiples of 5MB.

The memory consumed by Dagster running these problems against the size of the CNF of these problems for different numbers of cores is shown in figures 19. From figure 19 we can see that the memory used is a barely a fraction of the size of the CNF, which doesn't vary considerably with the number of cores. From this experiment it can be verified that Dagster can tackle problems which are large with respect to the available memory.

The invocation to run these experiments is to be found by executed the following command within the code repository:

```
~/summer1819/Benchmarks/cnf_concatenator/runme.sh
```

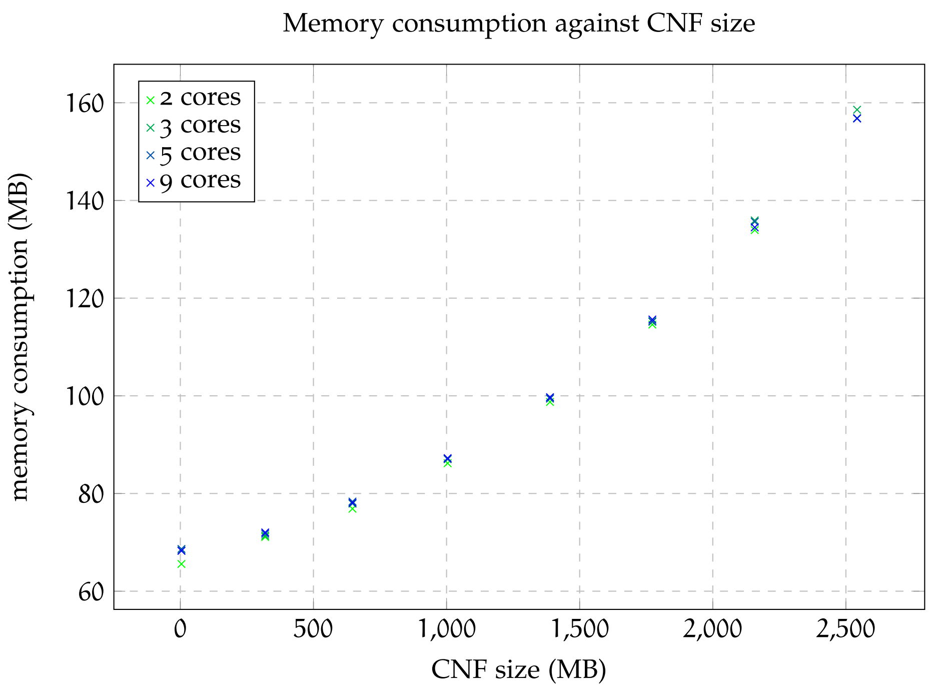


Figure 19: Dagster memory consumption for conjunctions of large-easy problems against core count.

REFERENCES

[1] C.P. Brown, M. Cenkl, R.A. Games, J.J. Rushanan, and O. Moreno. New enumeration results for costas arrays. In *Proceedings. IEEE International Symposium on Information Theory*, pages 405–405, 1993.

[2] Jon Carmelo Russo, Keith G. Erickson, and James K. Beard. Costas array search technique that maximizes backtrack and symmetry exploitation. In *CISS*, pages 1–8. IEEE, 2010.

[3] Konstantinos Drakakis, Francesco Iorio, Scott Rickard, and John Walsh. Results of the enumeration of costas arrays of order 29. *Adv. Math. Commun.*, 5(3):547–553, 2011.

[4] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, page 331–337, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[5] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-sat formulae and the satisfiability threshold. In David B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 126–127. ACM/SIAM, 2000.

[6] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *JSAT*, 4(2-4):149–172, 2008.