[4005-735-01] Parallel Computing I
Team Project Report

Team Satisfaction
Christopher Wood, Eitan Romanoff, Ankur Bajoria
caw4567@rit.edu, ear7631@rit.edu, arb2572@rit.edu

5/1/13

Table of Contents

1. The 3-SAT Problem

For our project we choose 3-SAT (or, more formally, 3-CNF-SAT), which is an NP-complete decision problem [1]. 3-SAT takes as input a 3-CNF Boolean formula and returns YES if the formula is satisfiable, and NO otherwise. A 3-CNF formula φ_n on *n* variables is expressed as the conjunction (Boolean AND) of arbitrarily many clauses, where each clause is the disjunction (Boolean OR) of exactly three literals (a literal is a Boolean variable or its negation). A formula φ_n is said to be satisfiable if and only if there exists an assignment of truth values to the *n* variables such that substituting them into the literals of φ will cause it to evaluate to true. Expressed as a formal language, we have 3-$SAT = \{(\varphi_n) : \varphi_n$ is a satisfiable 3-CNF formula$\}$ (i.e. the set of all 3-CNF formula strings that are accepted by the 3-SAT language if they are satisfiable).

2. An Exhaustive Search Algorithm

An exhaustive search algorithm for solving the 3-SAT problem with input φ_n iterates over all 2^n configurations of variable truth values, and for each configuration, assigns the truth value of each variable to the appropriate literal in φ_n, and then evaluates φ_n to determine if it is true (a visual depiction of this evaluation procedure is shown in Figure 1). If for every possible variable configuration φ_n does not evaluate to true, then the exhaustive 3-SAT algorithm returns NO. Otherwise, some satisfiable truth value configuration must exist, and so the exhaustive 3-SAT algorithm returns YES.

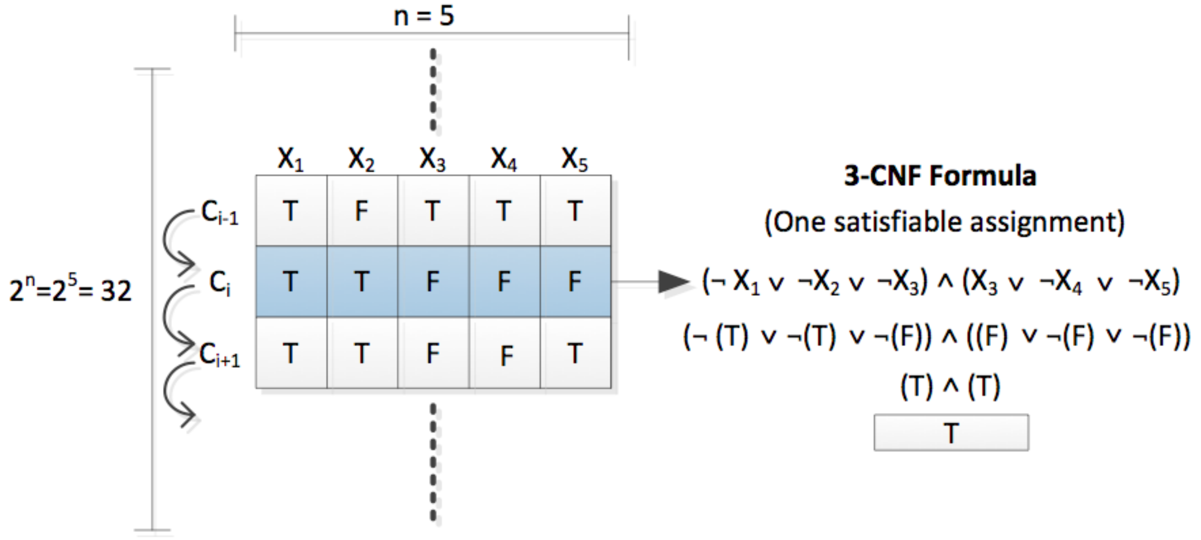

n = 5
2^n=2^5= 32
X1 X2 X3 X4 X5
Ci-1 T F T T T
Ci T T F F F
Ci+1 T T F F T
3-CNF Formula
(One satisfiable assignment)
(¬ X1 ∨ ¬X2 ∨ ¬X3) ∧ (X3 ∨ ¬X4 ∨ ¬X5)
(¬ (T) ∨ ¬(T) ∨ ¬(F)) ∧ ((F) ∨ ¬(F) ∨ ¬(F))
(T) ∧ (T)
T

Figure 1. Visual depiction of the evaluation step in the exhaustive search algorithm.

3. Advanced Algorithms

Most modern SAT solvers derive their base algorithm from what is commonly known as the Davis-Putnam method. The Davis-Putnam method improves on a general search by effectively pruning useless configurations that would lead to a knowingly FALSE configuration. This can be done by identifying a variable to evaluate as both true and false, and removing clauses whose disjunction is immediately known to be false. The formula then reduces itself to exclude the term, and is used for subsequent literal evaluation. Pseudocode for a generalization of the Davis-Putnam method is shown below in Listing 1.

```
DP_Method( Formula F ):
      // Perform unit propagation to force (imply) truth values.
      for each unit clause c in S: // clauses with 1 literal
            if literal l in clause c is negated:
                  F <- Propagate(F, not l)
            else:
                  F <- Propagate(F, l)

      // Check for completion - an empty formula means all clauses
      // have been satisfied from unit propagation.
      if F is empty
            return True
      if a clause in F is empty:
            return False

      // Choose a new literal, and split into two search paths.
      // This choice often relies on trained heuristics...
      Choose a new literal, x’
      if DP_Method(  Formula F, where x’ is True )
            return True
      if DP_Method( Formula F, where x’ is False )
            return True
```

Listing 1. A simplified version of the DPLL algorithm, adapted from [4]. Pure literal elimination, which forces variable truth values for literals with uniform polarity across the entire formula, is not shown in this version of the DPLL algorithm. Also, the propagation procedure works by replacing literals in the formula with the “implied” truth value for the underlying variable (i.e. the value of the variable that would make the unit literal evaluate to true).

3.1. Paper Review #1: 3-SAT on CUDA: Towards a Massively Parallel SAT Solver

This paper [5] presents a modified divide-and-conquer algorithm for exhaustively searching the entire state space of assignments to variables in the formula φ_n. At the core of the algorithm is the solve function, which takes as input a new or partially assigned formula φ_n. The solve function treats φ_n as a stack of clauses, and as such, only looks at the top of the stack when it is called. Internally, the function is divided into three separate cases corresponding to the number of literals in the clause. For example, if the top clause in φ_n has been partially assigned and now only contains one literal then the function enters the first case, if φ_n contains two literals then the function enters the second case, and so on.

In the event that the clause only has one literal, then the function assigns the appropriate truth value to the variable so that the literal evaluates to true, propagates this truth value to the remainder of the φ_n, yielding φ_n^1, and then recursively invokes the solve function for φ_n^1. If this assignment leads to a contradiction, then the base formula is not satisfiable. Similarly, in the event that the clause has two literals, the first literal is assigned to true and propagated throughout φ_n, yielding φ_n^1, which is then recursively called with the solve function. However, in the event that this leads to a contradiction, it is clear that the first literal in φ_n must be false, and the second literal must be true in order to satisfy the clause. Therefore, the first literal is set to false and the second literal is set to true in the formula φ_n, yielding φ_n^2. Now, the result of solve is the disjunction of solve(φ_n^1) and solve(φ_n^2). Following similar logic, one can see that in the third case where the clause has three literals, there the solve function is called recursively with the new formulas φ_n^1, φ_n^2, and φ_n^3. In the worst case scenario, the size of the search tree triples every time solve() is invoked.

The actual implementation of this algorithm is tailored for the Nvidia CUDA GPU architecture. In particular, there are two main parts of the implementation: preprocessing and solving. Given that the solve function returns when contradictions are found, the authors implemented a preprocessing sorting routine in the CUDA kernel that orders clauses in descending order based on how frequently their literals occur in φ_n. In other words, clauses on the top of the φ_n ”stack” will contain literals that occur most frequently in φ_n. This heuristic is helpful in obtaining contradictions early by achieving maximal fan-out during the unit propagation stage. The second part of the algorithm is the solver, which iteratively processes a formula φ_n by searching for contradictions (which results in discarding the formula), removing true literals and null clauses, and then recursively executing the solve function on the resulting formula φ_n'. At the core of the kernel pipeline is the SAT kernel, which is responsible for actually invoking the solve function to perform variable subsumption, resolution, and conflict detection to generate a new set of smaller and valid formulas for the next level of recursion.

Since CUDA devices have limited memory, the entire formula state space is maintained by the host, which swaps subsets of formulas to the solver so as to conserve memory. Therefore, this algorithm is implemented using the master-worker parallel pattern, where the master is the host and the workers are the CUDA kernels processing formulas. A visual depiction of this algorithm is given in Figure 2. A formula is deemed satisfiable if and only if a SAT kernel receives an empty formula (as a result of unit propagation, variable subsumption, and resolution). Conversely, a formula is deemed unsatisfiable if no such empty formula is generated and the master has exhausted its entire state space.

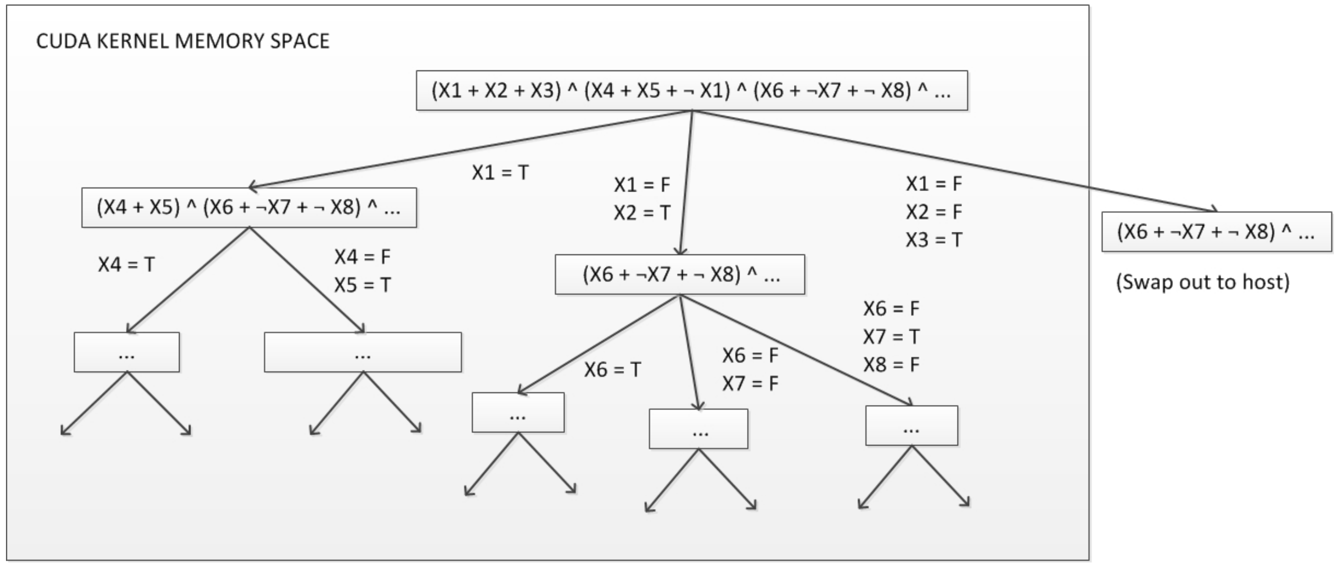


Figure 2. A visual depiction of the divide-and-conquer algorithm implemented to divide the 3-SAT search space. Since the memory on the target device (GPU) cannot store the entire formula state space, formula configurations are swapped to and from the host at runtime using a master-worker pattern.

3.2. Paper Review #2: ManySAT: A Parallel SAT Solver

The ManySAT paper [6] describes a portfolio based parallel SAT solver, which consists of several sequential SAT solvers with different heuristic strategies running in parallel and combining their results (through reduction) into a single YES or NO decision. Modern SAT solvers typically employ a plethora of heuristics such as unit propagation, literal selection based on their occurrence in formula clauses (or activity), and clause learning strategies to focus the computations on constrained parts of a formula. ManySAT introduces novel strategies that are tailored for parallel implementations, including dynamic restart policies to periodically refresh the literal assignments that will may lead to a satisfiable solution, variable polarity in the activity-based literal selection strategy, and finally, advanced clause learning. We describe each of these contributions in further detail below.

Typical restart policies are based on the rate at which literal assignments are “flipped” after a conflict (i.e. contradiction between literal assignments) arises in the formula. Generally, a higher rate of change means that new paths in the search tree are being discovered more frequently, and vise versa. Therefore, restarts often occur when there is a low rate of change among literals, indicating that some particular branch in the state space tree has been investigated for too long. ManySAT provides an alternative metric that influences when restarts occur. In particular, ManySAT will use the average length of the state space backtracking paths, or backjumps (i.e. how many levels of recursion are removed to change the value of a literal assignment when using clause-based learning), to determine when restarts should commence. If the change in the average length of these backtracking paths is high, then restarts will occur more frequently, and vise versa. In the ManySAT implementation, one core uses this new dynamic restart policy, and the other sequential solver instances use the classic Luby policy [8] and a geometric policy in which restart threshold increases exponentially for each decision level.

The literal selection heuristic is based on a modification of the VSIDS branching heuristic. This strategy works by choosing literals based on the frequency with which they appear in formulas, which is done by maintaining a simple counter for each variable. This frequency (sum) is then reduced, or divided, by a fixed constant at regular intervals during the solver’s execution. Since there are typically four physical cores working on a given formula, each core is configured to use a different variation of the literal selection strategy. Core 0 uses a polarity-based literal selection strategy which assigns true to a literal if its non-negated form occurs more frequently than its negated form, and vise versa. Cores 1 and 3 use a progress saving heuristic in which truth values are recorded at each decision level, and in the event that the same literal is encountered again at some other branch of the formula, the previously chosen truth value is preferred. Finally, core 2 defaults to choosing false for all newly encountered literals.

Finally, the ManySAT solver uses conflict-driven clause learning with a modification of the traditional implication graph used to store partial variable assignments and the implications that are made by these assignments (see Figure 3 for an example of an implication graph). In particular, ManySAT proposes the use of an “inverse arc” to increase the length of backtracking paths when conflicts arise. The main idea of the inverse arc is to reintroduce satisfied clauses in the implication graph, and then use these new clauses to “eliminate” literals in a partial assignment that were found at the deepest decision level, which enables the backtracking to recurse farther up the tree. As an example, if we have a partial assignment $\rho = (\neg x^1 \vee y^3 \vee z^5)$ that creates a conflict at decision level 5, where the $\neg x^1$ means that x received a value of false at decision level 1, the inverse arc technique introduces previously satisfied clauses that causes the backtracking to skip the assignment of y and revisit x instead. More specific details of this technique are discussed in the paper.

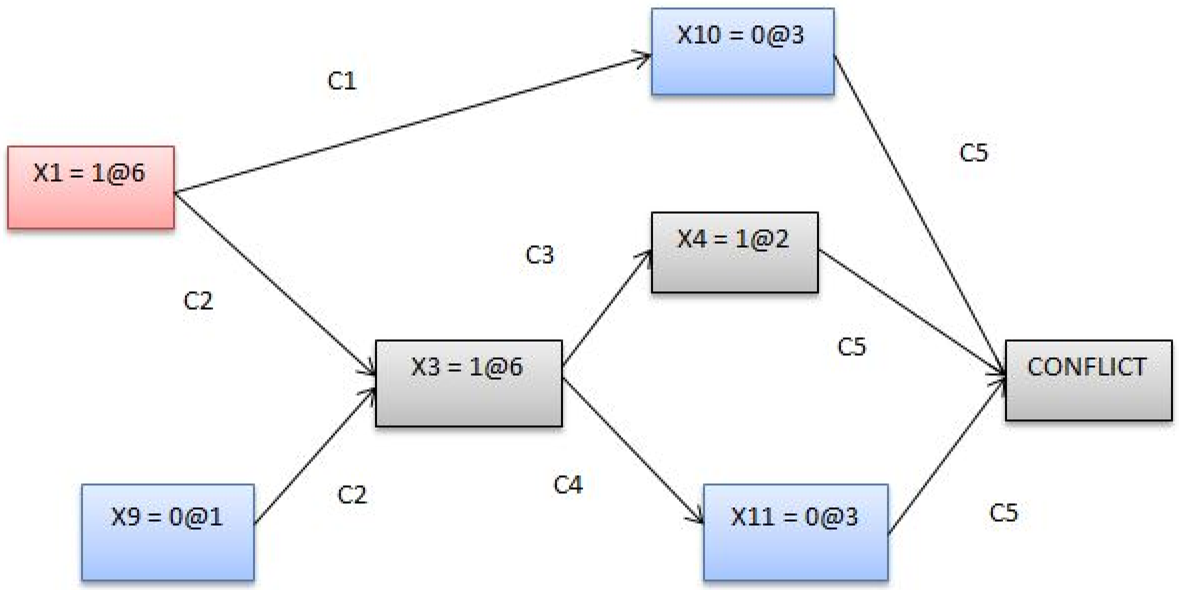


Figure 3. An implication graph which shows partially assigned values for the variables in the form truth_assignment@decision level. In this graph, the current truth assignment is {X9 = 0@1. X10 = 0@3, X11 = 0@3} (marked in blue) and the current decision assignment is {X1 = 1@3} (marked in red). The graph shows the values for the variables that were assigned at earlier decision levels and the value for the variable X1 that is being assigned. Also, note that a clause Ci forms a directed edge between two variables if the value of Ci implied the truth value of the tail vertex on the edge. In this case, C1 = (¬X1 V X2), C2 = (¬X1 V X9 V X3), C3 = (¬X3 V X4), C4 = (¬X3 V X11), and C5 = (X4 V X11 V X10). Finally, note that we have a conflict from the implied clause (¬X1 V X9 V X11 V X10), since ¬X1, X9, X11, X10 all yield false given the current partial assignment.

The last contribution of the ManySAT algorithm is the concept of clause sharing, which serves to share learnt clauses among the separate cores in order to reduce redundant computations. The distribution of these clauses occurs through lockless queues in shared memory.

3.3. Paper Review #3: PSATO: A Distributed Propositional Prover and its Application to Quasigroup Problems

The third paper [7] describes an implementation of a parallel SAT solver based on a distributed master/slave workflow. PSATO is a parallel version of the SATO solver, a solver derived from the standard Davis-Putnam approach. SATO introduces several optimizations to the general Davis-Putnam approach that are derived from other optimized SAT solver approaches. In particular, SATO is a trie-based implementation of the Davis-Putnam approach.

A SAT configuration can be conceptualized in the following manner: all variables in a particular configuration have a unique index, and the negation of that variable is the negation of the index. A clause is then a list of these indices. The relationships between clauses is then stored as a trie - a tree structure where the edges between nodes have values which denote the index of a literal's value. A clause is represented by a path from the root of the trie, to a non-null leaf, where all the edge values represent the disjunction. The trie representation is borrowed from the DDPP implementation of the Davis-Putnam approach. Using a trie representation in modeling a propositional satisfiability problem provides several inherent benefits, such as the removal of duplicate clauses, which greatly reduces the amount of required memory. Furthermore, given how the trie is ordered, propagating a variable's value can be done in a single pass of the trie, which is a sublinear operation.

The standard trie approach is slow, however, due to the necessary trie-merge operations performed on the structure performed when there is a split on a variable (recursively propagating a variable as true, and as false). To further optimize this approach, SATO borrows the usage of configuration state from the LDPP approach to eliminate the need of trie-merge. SATO accomplishes this by using two lists: a head list which contains the occurrences of the first literal of each clause, and a tail list which does the same for the last literal in each clause. Furthermore, each node in the trie has a reference to its parent. Simple modification operations can be done to these lists to avoid the need for trie merging entirely.

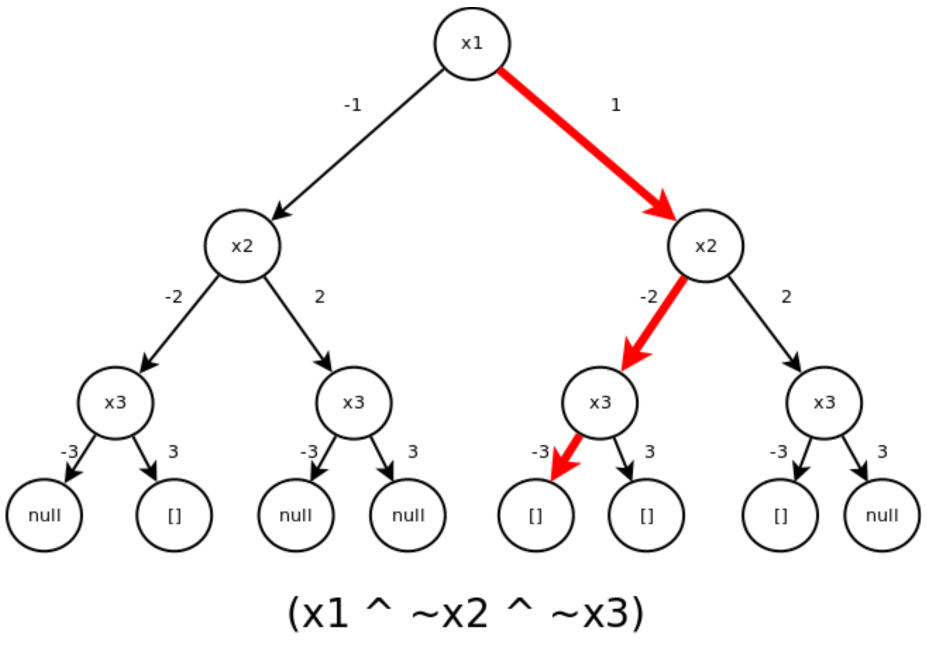


Figure 4. Path representation in a trie for a particular clause.

The paper describes the search space for a boolean satisfiability problem as a binary tree, where each node is a call to the general Davis-Putnam approach, and each call branches based on the selection of a literal to evaluate (negated and not negated). Paths can be derived by linking subsequent calls to their parent, creating a list of function calls in the search space. In the case of an early termination (due to reasons other than finding a solution), the path generated to that terminated call is referred to as a guiding path, which can be analyzes on subsequent runs with the same input to avoid re-computing branches of the search space on large problems. A path is said to be open, or closed, depending on if the node was yet considered in the depth-first search for a solution. Guided paths are applied to create a modification to the Davis-Putnam approach, where if a guided path is supplied, rather than using a heuristic to choose a new literal to split the search space on, it checks against the guiding path to skip to literals that have not been fully explored. Furthermore, when using guiding paths, large SAT problems can be solved in multiple time steps, rather than in one execution.

PSATO distributes the computation of a SAT problem by generating 2^k guiding paths for a particular configuration to solve, where k is the number of literals in the configuration. The paths are generated such that they are on completely disjoint portions of search-space tree. For any particular guided path, the first node found on the path that is unclosed can be split into two subpaths. PSATO runs on a master/slave system, where the master delegates guided paths to the various slave nodes, maintaining a list of guided paths sorted by length. If the number of available paths drops below a certain threshold, more paths can be generated by simply splitting at the first open node. Slave nodes search along a particular guided path until either finding a result, or after spending a sufficient amount of time. If the slave does not find a result, it returns a guided path which can be delegated to subsequent slave machines.

4. Program Design

Due to the nature of the 3-SAT problem, it can be deceiving to analyze the performance of the decision variant. This is because the time required to prove the satisfiability of a set of formulas may differ depending on the literal placement in each of the respective clauses. In other words, some formulas with a large number of variables and clauses may be proved satisfiable with the default variable assignment configuration (i.e. all variables are set to false), whereas other formulas with the same number of formulas and clauses may only be satisfiable with the last configuration checked (i.e. all variables are set to true). Therefore, we implemented two variants of the 3-SAT problem: a decision program that returns an answer for satisfiability as soon as one is obtained, and an exhaustive program that computes the total number of satisfying variable configurations and prints the first satisfying configuration encountered. The minor differences in these two variants are discussed in the following sections, but, generally speaking, they have very similar designs.

4.1. High-Level Design of Sequential Program

Based on the prior description of the exhaustive search algorithm, our sequential program required data structures for the CNF formula and current variable assignment. To store the CNF formula we created a separate helper class, `Literal`, which stores both a literal identifier, which is just an integer ID, and a negation flag, which indicates whether or not the literal is negated in the formula. Then, the formula is represented as a two-dimensional array of Literal objects which are written once and then read many times. While we could have stored the variable ID and negated flags in two separate arrays, doing so would not make good use of the locality of reference principle. With the use of `Literal` objects we keep both pieces of data close to each other in memory and thus also close in the cache. To store the variable configurations, we simply store a single-dimensional array where the indices of the array correspond to the unique identifiers for the variables. That is, index 0 corresponds to variable 0. There is no other additional data that needs to be stored by the program.

The program itself is divided into two parts: the initialization and solution tasks. The initialization task is responsible for parsing the command line parameters and populating the CNF formula data structure. Currently, both the decision and exhaustive sequential programs support the ability to initialize the formula from a file using the format specified in the preceding section, or read in the number of variables, clauses, and a random seed used to populate the formula. Once complete, the solution task then initializes the variable configuration array and performs the steps outlined in the exhaustive search algorithm. For the decision program, the traversal over all 2^n variable configurations stops as soon as the formula evaluates to true and the program outputs “YES” or “NO” depending on whether the formula is satisfiable. For the exhaustive search program, the total number of satisfying configurations and the first satisfying configuration, if one exists, are recorded over all 2^n configurations, and then this information is printed upon completion. A high-level pseudocode description of this design for the exhaustive program run using a specified number of variables, clauses, and seed is given in Algorithm 1

```
procedure initialize():
   Read numVars, numClauses, and seed from the command line
   formula = Literal[numClauses][3]
   variables = boolean[numVars]
   for i = 0 -> numClauses - 1:
      formula[i][0] = new Literal(PRNG.nextInt(), PRNG.nextBoolean())
      formula[i][1] = new Literal(PRNG.nextInt(), PRNG.nextBoolean())
      formula[i][2] = new Literal(PRNG.nextInt(), PRNG.nextBoolean())

procedure solve():
   numSat = 0
   for i = 0 -> numVars - 1:
      variables[i] = False
   for i = 0 -> 2^numVars - 1:
      formulaSat = True
      for c = 0 -> numClauses - 1:
         clauseSat = False
         for l = 0 -> 2:
            id = formula[c][l].id, negated = formula[c][l].negated
            if (negated == True && variables[id] == False)
                or negated == False && variables[id] == True):
                clauseSat = True
         if (clauseSat == False):
            formulaSat = False
      if (formulaSat == True):
         numSat++
   return numSat

procedure main():
   initialize()
   numSat = solve()
   print(numSat)
```

Algorithm 1. Pseudocode design of the sequential program. Some particular details are omitted for brevity. Please see the commented source code for a complete description.

4.2. High-Level Design of Parallel Program

The design of the exhaustive and decision parallel 3-SAT programs are very similar, and heavily based on the sequential program design. Each variant is composed of two major tasks: an initialization and solution task, where each task performs the same behavior as in its sequential counterpart. Generally speaking, their parallelization occurs by transforming the outermost for loop in these tasks into a parallel for loop. We discuss the finer intricacies of this design in the following sections, in which we describe the important concepts of our parallel program design.

4.2.1. Parallel Design Patterns

The exhaustive and decision 3-SAT parallel programs used two different parallel design patterns. Specifically, the decision variant uses agenda parallelism to determine if there exists at least one satisfying solution for the 3-CNF formula. Given the nature of the decision problem, we are only concerned with answering the question of whether or not a particular variable configuration satisfies the formula. Therefore, this is clearly agenda parallelism.

Quite oppositely, the exhaustive parallel program is trying to find and count all possible satisfying solutions to the 3-CNF formula. For correctness, the program must therefore determine if every possible configuration satisfies the formula, so this is clearly result parallelism. However, in order to match the output of its sequential counterpart, this program must display the first satisfying variable configuration as output. To do so, each thread maintains its first encountered solution and then, when all threads finish with their given slice, these results are reduced together and the first solution is returned to the main thread to be displayed to the user. The details of this reduction step are discussed in the following section.

4.2.2. Data Structure Selection

Based on the description of our sequential programs, the two-dimensional array of Literal objects and one-dimensional array of primitive boolean elements are the main data structures in the parallel programs. For both the exhaustive and decision programs, the Literal data structure is populated in parallel during the initialization task. Since each thread is given a unique slice of clauses to populate and it is only ever read during the solution task, there was no need to synchronize access to this data structure across multiple threads. Also, given the computation partition scheme outlined in the following section, each thread only needs to maintain its own variable configuration array in both the exhaustive and decision programs. In both programs, these thread-local data structures are modified as each thread iterates over its configuration slice. Therefore, no synchronization was needed to manage access to this data structure for any thread.

Aside from these two main data structures, the exhaustive program also maintains a thread-local counter for the number of satisfying solutions and the actual index for the first satisfying solution that is encountered. Each thread returns these two values in an object derived from a helper class called SolutionWrapper. Then, the per-thread count for satisfying solutions is reduced into a SharedLong variable for the entire solution count. Similarly, the satisfying indices are reduced into a SharedLong using the LongOp.MINIMUM operator. These two SharedLong instances handle the synchronization across all threads so we do not have to write our own.

With regards to the decision program, there is an additional shared storage container of type long that holds the total number of satisfying solutions across all threads. In order to determine if any thread has found a satisfying solution and prematurely stop searching, each thread checks the value of this shared counter prior to checking a single variable configuration against the 3-CNF formula. This does not create any performance issues because reading is not synchronized, and furthermore, it does not need to be synchronized. Since this variable is only ever set when a satisfying solution is found and we are not concerned with which configuration actually yielded the solution, we safely assume that whenever the counter goes above a value of zero that *some* thread found a solution, and so the threads all terminate, as expected.

4.2.3. Computation Partitioning Strategy

The computations required for the initialization and solution tasks were evenly distributed among the available threads using two different techniques. During the initialization task, the clauses in the formula are evenly partitioned among the set of available threads and then initialized independently from the rest. Without this division of labor, the formula initialization task would contribute a significant amount of time to the program's overall sequential fraction. Also, since each literal in a given clause is initialized using a PRNG, we needed to create a thread-local PRNG for each clause chunk and skip ahead to the appropriate position in the PRNG sequence. Since there are always three literals per clause, the skip value was equivalent to `3*first.`

In a similar fashion, the work of the solution task is evenly partitioned among the set of available threads by evenly splitting the variable configuration search space. This partitioning scheme enables us to distribute work at the outermost loop in out exhaustive search algorithm, thus yielding the optimal performance gains from multiple threads. Under this partition scheme, each thread is assigned a slice of the total variable configuration range (i.e. some contiguous range between 0 and 2^n - 1, inclusive). As discussed in the previous section, each thread needs to access the entire two-dimensional formula array to check if a particular variable configuration satisfies the formula. This partitioning scheme is illustrated in Figure 6.

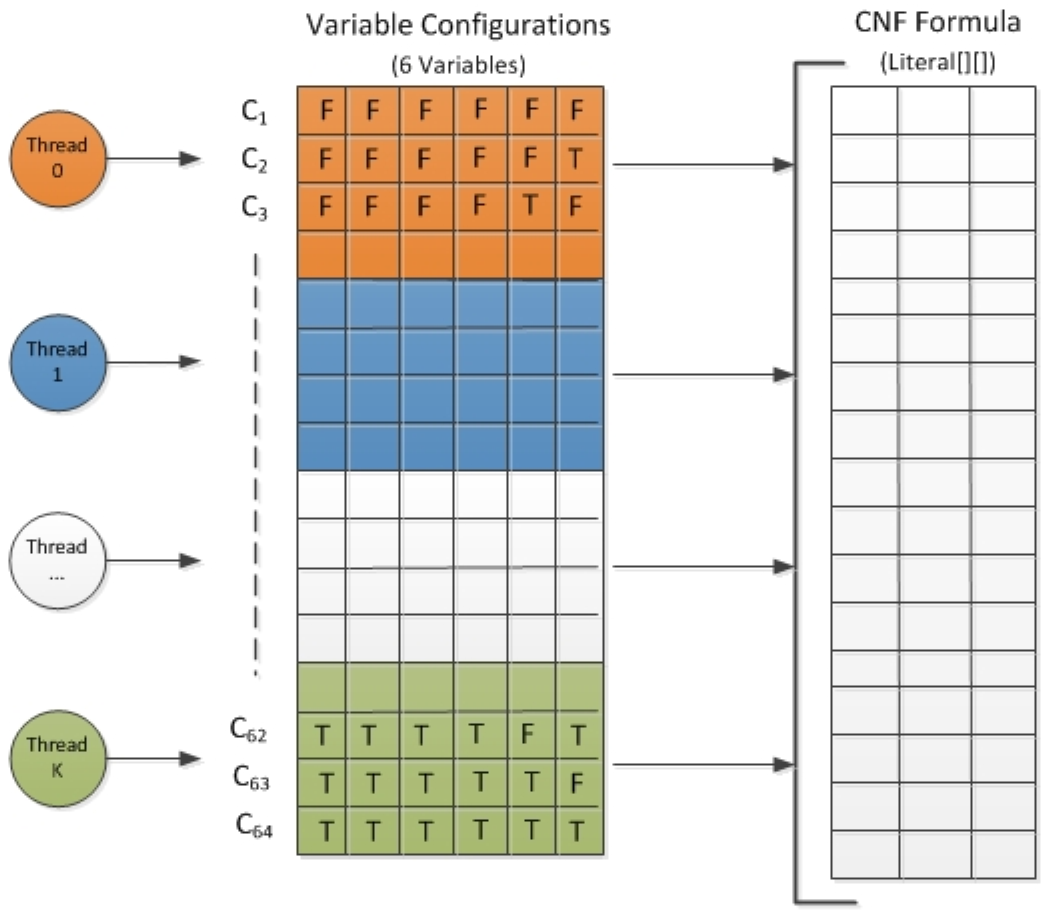


Figure 6. Configuration-based computation partition for the parallel program. Notice that each thread still requires access to the entire CNF formula to perform its computation. In this figure, each C_i represents a unique variable configuration.

In the decision parallel program, the solution task returns when a single thread identifies a satisfying configuration and writes a value to a shared variable indicating such result. However, since the exhaustive parallel program needs to record and display more information to the user, there is a reduction step that needs to occur at the end of the solution task. During this reduction step, the per-thread solution counters are added together using the addAndGet method of a shared SharedLong variable, and the first satisfying solution index is recorded by performing a global reduction on all per-thread solution indices with the LongOperation.MINIMUM operator.

4.2.4. Thread Synchronization

The only sequential dependency in both the exhaustive and decision parallel programs is the population of the formula with a valid set of literals to attempt to solve. The solution task cannot proceed without having this information beforehand. Therefore, in order to synchronize the threads and ensure the program's correctness, we make use of the implicit barrier at the end of the IntegerForLoop that iterates over the set of clauses during the initialization task. If any thread should finish before the others, it will be blocked at this barrier before proceeding to the solution task.

Synchronization in the solution task differs between the exhaustive and decision programs. In the latter, the only form of synchronization required is a single flag that states if a solution has been found. As previously discussed, this flag is represented as a shared long counter that is written to by every thread. Since we are not concerned about the actual value of this counter, we can neglect any read-write conflicts that occur on this shared variable. Therefore, in a sense, formal synchronization does not occur in the decision program. This is not the case in the exhaustive program, which needs to record the index of the first satisfying configuration as well as the total number of satisfying configurations. To synchronize the threads to capture this information, SharedLong objects from the Parallel Java library are utilized as storage containers for these shared pieces of information to enable us to perform synchronized updates and reductions.

A visual depiction of the synchronization logic used in the initialization and solution tasks is shown in Figure 7.

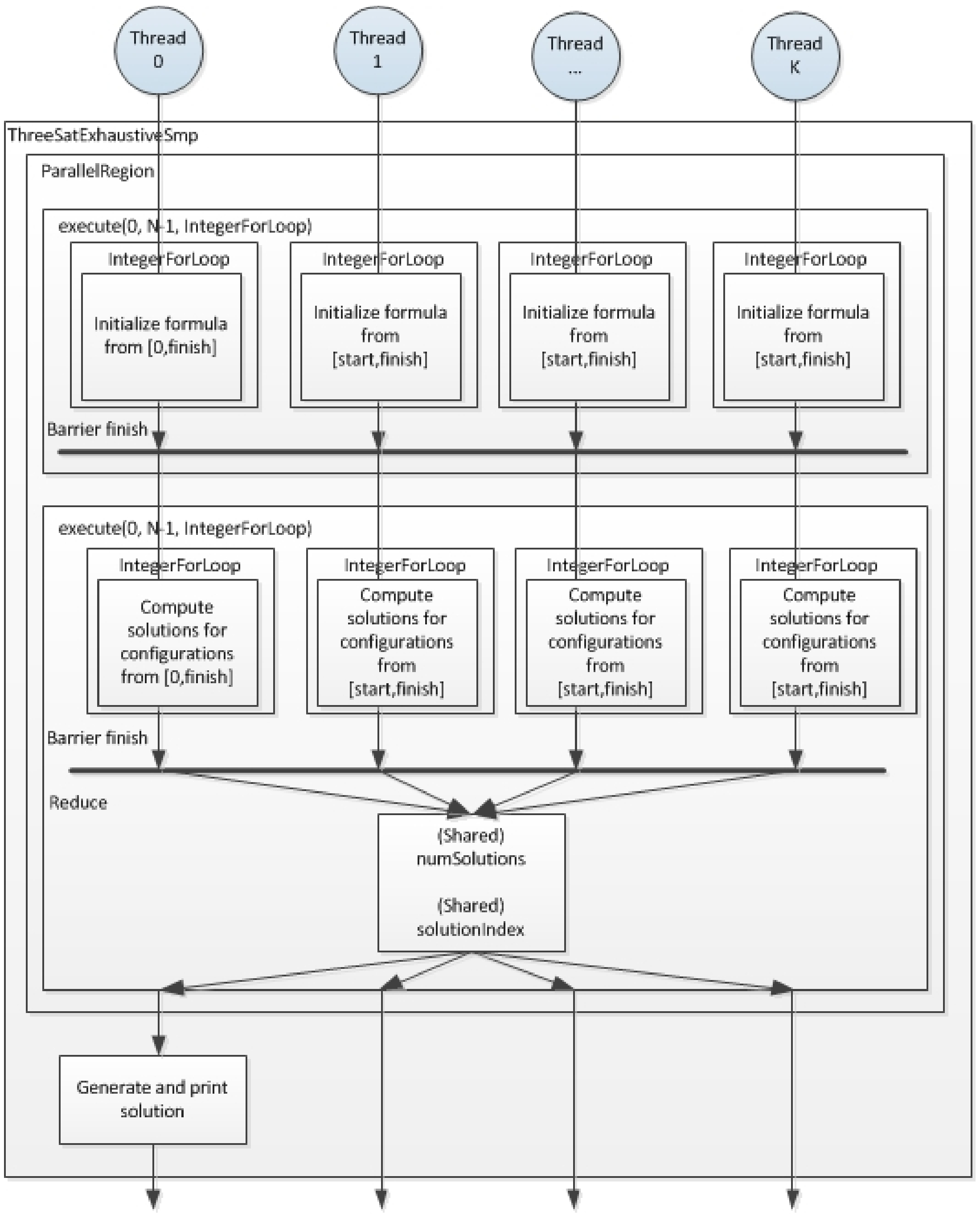

Thread 0
Thread 1
Thread ...
Thread K
ThreeSatExhaustiveSmp
ParallelRegion
execute(0, N-1, IntegerForLoop)
IntegerForLoop
Initialize formula from [0,finish]
Initialize formula from [start,finish]
Barrier finish
execute(0, N-1, IntegerForLoop)
IntegerForLoop
Compute solutions for configurations from [0,finish]
Compute solutions for configurations from [start,finish]
Barrier finish
Reduce
(Shared) numSolutions
(Shared) solutionIndex
Generate and print solution

Figure 7. High-level control flow of the parallel exhaustive 3-SAT program.

4.2.5. Load Balancing

Load balancing was a critical step in optimizing the performance of both parallel programs. Given the nature of the 3-SAT problem, certain variable configurations may immediately yield false formulas after checking the first clause, a slightly different configuration may only fail at the last clause in the formula. For problems with a large amount of clauses, such situations would result in a highly unbalanced load. Therefore, we experimented with all three different types of schedules (fixed, dynamic, and guided) to determine which one yielded the best results. To do this, we left the default runtime schedule for all IntegerForLoops used in the initialization and solution tasks, and then simply specified the type of schedule we wanted as a JVM flag at runtime.

Based on our empirical observations, which are discussed in section N, we found that using a guided and fixed schedule resulted in the best performance, respectively. Quite surprisingly, using a dynamic schedule severely deteriorated the speedups and efficiencies. Therefore, we settled on a final design with uses a guided schedule to achieve an optimally balanced load for random 3-SAT problem instances.

5. Developer Manual

The TeamSatisfactionDeliverables.jar archive will include four different versions of the ThreeSat solver programs in the 'src' directory (two sequential versions and two parallel versions). The difference in versions is specified by the names of the programs. The programs with Exhaustive in their names imply that these programs will list all the solutions that it will find that satisfy the Boolean formula. The programs that do not have the keyword Exhaustive in their names are the ones that terminate on finding the first solution to the Boolean formula. There is an additional helper class, Literal, that acts as an object container to hold the variable ID for a literal along with its negation flag (i.e. whether or not it is negated in the formula). To compile the programs they must be extracted into their a separate directory from the JAR file. This can done using the following command after navigating to the directory that the JAR file resides in:

```
jar xf TeamSatisfactionDeliverables.jar
```

In order to be able to compile the programs correctly the Parallel Java Library is needed. If the Parallel Java Library is not already set in your CLASSPATH then it can be set by using the following command for the executable version.

```
export CLASSPATH=.:/extracted_directory/pj20120620.jar
```

To compile the programs independently the commands would be,

```
javac –source 1.5 –target 1.5 ThreeSatSeq.java
javac –source 1.5 –target 1.5 ThreeSatSmp.java
javac –source 1.5 –target 1.5 ThreeSatExhaustiveSeq.java
javac –source 1.5 –target 1.5 ThreeSatExhaustiveSmp.java
```

The programs can also be compiled together using the command,

```
javac –source 1.5 –target 1.5 *.java
```

6. User Manual

All the programs are present in the ‘src’ directory in the provided JAR file. When the JAR file is extracted into the desired location and the programs have been complied they can be tested by executing them. The programs need the Parallel Java Library to run. If this is not already set in the CLASSPATH then it can be done using the following command:

```
export CLASSPATH=.:/extracted_directory/pj20120620.jar
```

The CLASSPATH can also be set at runtime by using the following arguments in the command line,

```
java -cp pj20120620.jar:. filename <arguments>
```

The programs accept input either in the form of a .cnf file or by generating a random Boolean formula specified by the command line arguments <num_literals> <num_clauses> <seed>. The arguments are used to specify the number of literals or variables, the number of clauses in the formula and the random seed respectively. The input file uses a modified form of the DIMACS CNF format. DIMACS CNF is an accepted standard for specifying input in CNF format for Boolean satisfiability problems. The file normally has the following structure:

```
c simple_v3_c2.cnf
c this is a comment...
p cnf 3 2
1 -3 2 0
2 3 -1 0
```

The file starts with a comment specified by ‘c’ in the beginning. The comment lines are followed by the problem line specified by using ‘p cnf’. This line specifies the number of variables and the number of clauses that will follow in the later lines. All the remaining lines are used to specify one clause each for the formula. A negative ‘-‘ symbol at the starting of a literal implies a negation. The ‘0’ at the end of each clause specifies that the clause has ended.

For our project we use a modified form of the DIMACS format. We do not include the comments or problem definitions. An input file to be used with our programs has the following structure:

```
3 2
1 -3 2
2 3 -1
```

The first line is used to specify the number of literals and the number of clauses respectively. The following lines specify one clause each. There is no ‘0’ at the end to specify the end of the clause. Instead each new line is treated as a new clause. Also each clause line has only three literals. It is expected that the user will adhere to this file format rule. Including more clauses than is specified in the first line will cause them to not be read by the program, and quite oppositely, including less clauses than is specified in the first line will cause the program to terminate abnormally. Also, each literal must be some integer between the range [1, numVars], as specified in the first line of this file. Failing to adhere to this rule will cause the program to terminate abnormally.

To prepare the input for the program using provided formulas encoded in these files, one may first generate a CNF file by hand or download sample problems from online resources, such as those provided here [9]. Then, using the convertCnf.py script available in ‘scripts’ directory of the JAR file, one may convert these DIMACS CNF files to the equivalent versions compatible with our program. The convertCnf.py script must be used as follows:

```
python convertCnf.py filelist
```

where filelist is the name of a file that contains the names of other CNF files to be converted. Our provided JAR file contains a plethora of sample CNF files to test for correctness in the “test_files” directory.

Alternatively, our programs may be run by randomly populating the CNF formula. In fact, this is the approach we took for testing the performance of our programs. To use this mode, one simply needs to pass in a specified number of variables, clauses, and a random seed on the command-line. Internally, each program will use these parameters to randomly populate the formula that is to be solved.

The rationale for including these two separate modes of operation was to be able to verify the correctness of our program using known satisfiable or unsatisfiable formulas, which could be checked with existing SAT solvers, and then generate random formulas for the performance analysis.

As previously mentioned, each of the four programs can be run using either the file or the random generated Boolean formula as the command line arguments. To execute the programs the following commands can be used:

```
java -Xmx2000m ThreeSatSeq [<file> | <num_vars> <num_clauses> <seed>]
java -Xmx2000m ThreeSatSmp [<file> | <num_vars> <num_clauses> <seed>]
java -Xmx2000m ThreeSatExhaustiveSeq [<file> | <num_vars> <num_clauses> <seed>]
java -Xmx2000m ThreeSatExhaustiveSmp [<file> | <num_vars> <num_clauses> <seed>]
```

where <file> is the input file for the program (filename.cnf), <num_vars> is the number of variables for the formula, <num_clauses> is the number of clauses for the formula, and <seed> is the seed use to initialize the PRNG that creates the random CNF formula if it's not read in from a file.

The output for the programs can be split into two categories. The output for the decision programs consists of a "Yes" or "No" statement followed by the total execution time. The output for the exhaustive programs consists of a "Satisfiable with N solutions" or "Not satisfiable" statement, along with the first encountered satisfying solution if one was found, followed by the total execution time. One may then use the script FormatOutput.py to collect the output for a large set of runs and prepare it for input into the Speedup and Sizeup programs in the Parallel Java Library. Please see the directories "random_out/clauseFiles" and "random_out/varFiles" for the output files generated after running the test scripts "scripts/ThreeSatLiterals.sh" and "scripts/ThreeSatClauses.sh" on the `paragon.cs.rit.edu` parallel computer.

7. Performance Metrics

In this section we describe the performance metrics we collected for both the exhaustive and decision program variants. Before discussing these metrics, we first note that the 3-SAT problem is unique in that the problem size is not simply a function of a single parameter. Rather, the problem size is determined by both the number of variables and clauses. Let N_v and N_c denote the number of variables and clauses, respectively. Then, the problem size for the 3-SAT programs is on the order of $O(2^{N_v} + N_c)$. Since the problem size is typically dominated by the number of variables, unless N_c is very large, we focus our attention on the performance analysis by fixing the number of clauses and then varying the number of variables. However, for completeness, we also conducted experiments with a fixed number of variables and varied number of clauses. Given our computation partition scheme, varying the number of variables should intuitively yield better speedup and sizeup than varying the number of clauses. As we will show in the following sections, this is precisely what we observed.

7.1. Variable Experiments

In this section we describe present the results from our performance analysis by fixing the number of clauses to 1000 and solving formulas with 23, 24, 25, 26, and 27 variables. We present a complete set of speedup and sizeup metrics for the best implementations of our parallel programs (i.e. those which used a guided schedule to balance the load among each thread). The remaining set of data collected during our performance analysis is presented in the corresponding document entitled "Performance Analysis Results."

7.1.1. Speedup Metrics with a Guided Schedule

7.1.1.2. Exhaustive 3-SAT

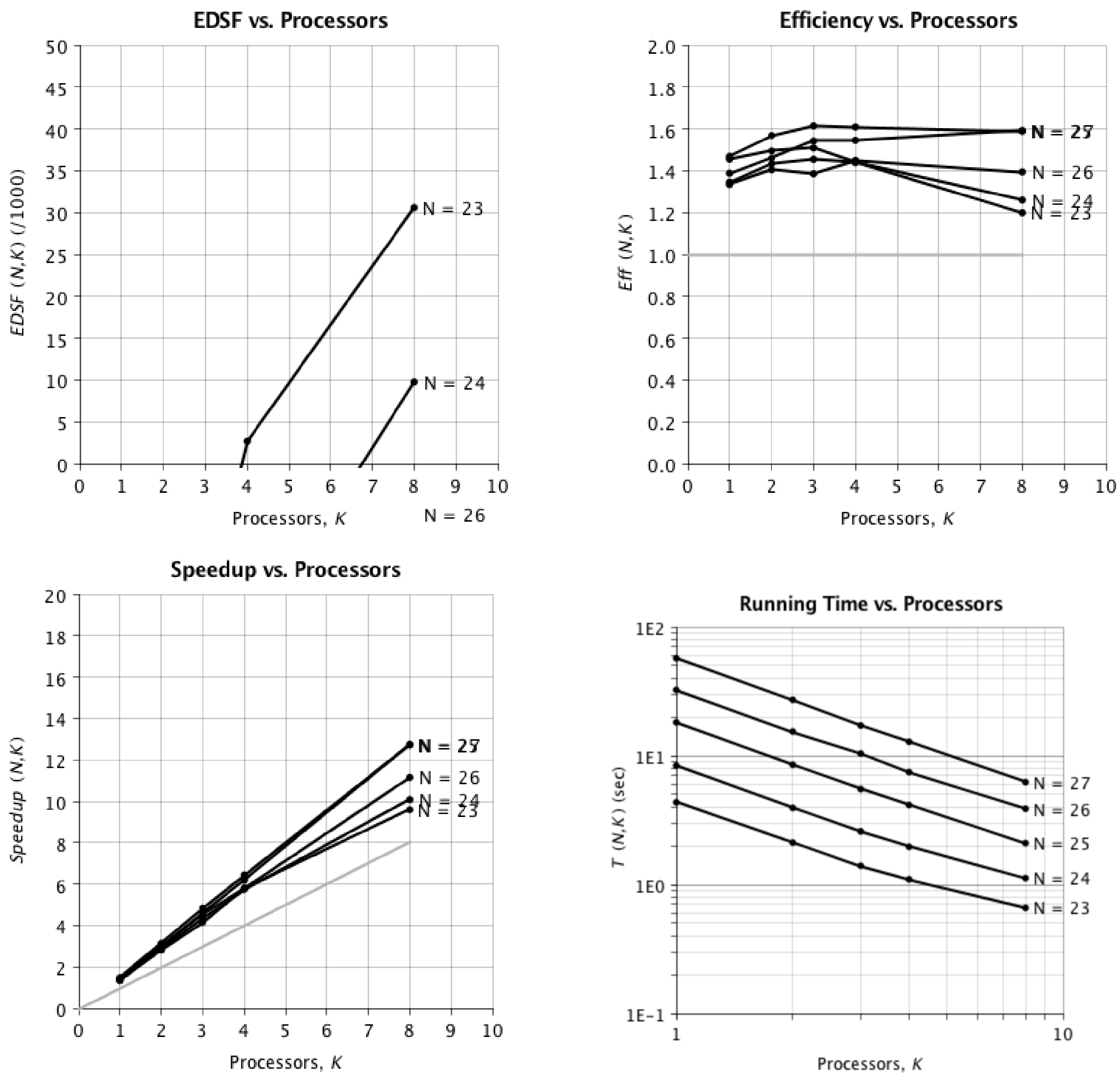


Figure 8. Speedup performance metrics from the exhaustive 3-SAT program run with a guided schedule.

Table 1. Speedup metrics for the exhaustive parallel program with 23 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	6329			
1	4349	1.455	1.455	
2	2118	2.988	1.494	-0.026
3	1398	4.527	1.509	-0.018
4	1096	5.775	1.444	0.003
8	660	9.589	1.199	0.031

Table 2. Speedup metrics for the exhaustive parallel program with 24 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	11283			
1	8387	1.345	1.345	
2	3938	2.865	1.433	-0.061
3	2587	4.361	1.454	-0.037
4	1961	5.754	1.438	-0.022
8	1120	10.074	1.259	0.010

Table 3. Speedup metrics for the exhaustive parallel program with 25 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	26674			
1	18174	1.468	1.468	
2	8504	3.137	1.568	-0.064
3	5506	4.845	1.615	-0.046
4	4143	6.438	1.610	-0.029

8	2098	12.714	1.589	-0.011

Table 4. Speedup metrics for the exhaustive parallel program with 26 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	43213			
1	32384	1.334	1.334	
2	15388	2.808	1.404	-0.050
3	10398	4.156	1.385	-0.018
4	7456	5.796	1.449	-0.026
8	3875	11.152	1.394	-0.006

Table 5. Speedup metrics for the exhaustive parallel program with 27 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	79313			
1	57144	1.388	1.388	
2	27178	2.918	1.459	-0.049
3	17146	4.626	1.542	-0.050
4	12831	6.181	1.545	-0.034
8	6228	12.735	1.592	-0.018

7.1.1.2. Decision 3-SAT

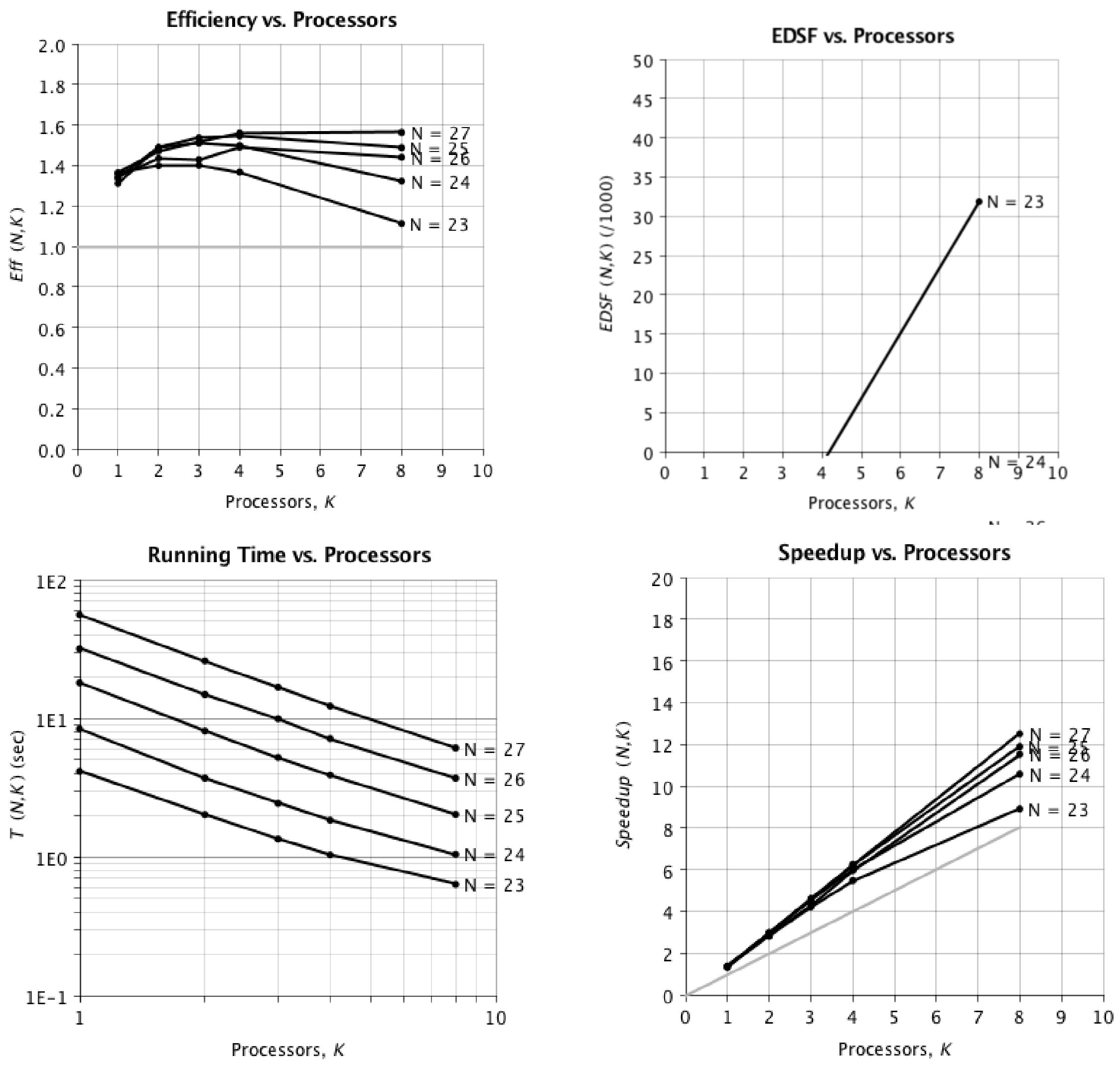


Figure 9. Speedup performance metrics from the decision 3-SAT program run with a guided schedule.

Table 6. Speedup metrics for the decision parallel program with 23 variables and 1000 clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	5645			
1	4147	1.361	1.361	
2	2018	2.797	1.399	-0.027
3	1344	4.200	1.400	-0.014
4	1032	5.470	1.367	-0.002

8	634	8.904	1.113	0.032

Table 7. Speedup metrics for the decision parallel program with 24 variables and 1000 clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	11019			
1	8409	1.310	1.310	
2	3701	2.977	1.489	-0.120
3	2433	4.529	1.510	-0.066
4	1838	5.995	1.499	-0.042
8	1042	10.575	1.322	-0.001

Table 8. Speedup metrics for the decision parallel program with 25 variables and 1000 clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	24097			
1	18055	1.335	1.335	
2	8107	2.972	1.486	-0.102
3	5224	4.613	1.538	-0.066
4	3898	6.182	1.545	-0.045
8	2026	11.894	1.487	-0.015

Table 9. Speedup metrics for the decision parallel program with 26 variables and 1000 clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	42555			
1	31674	1.344	1.344	
2	14833	2.869	1.434	-0.063
3	9928	4.286	1.429	-0.030
4	7140	5.960	1.490	-0.033

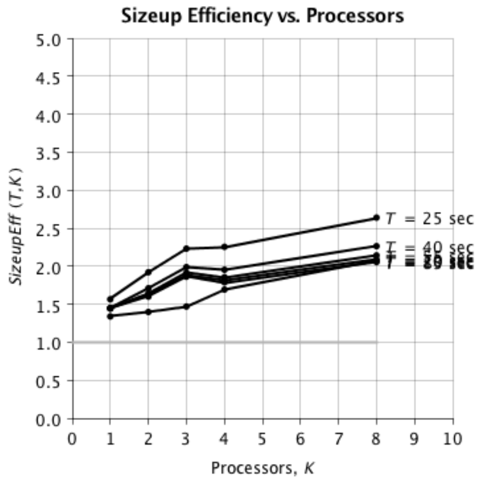
8	3693	11.523	1.440	-0.010

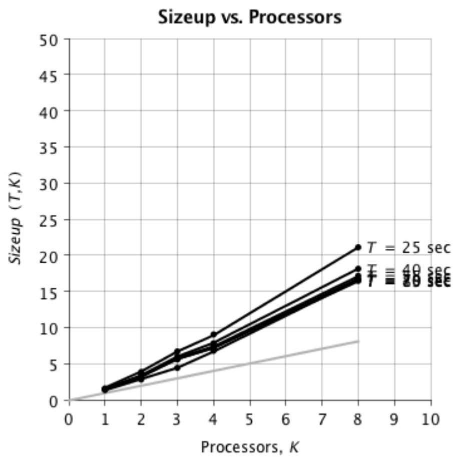
Table 10. Speedup metrics for the decision parallel program with 27 variables and 1000 clauses using a guided schedule.

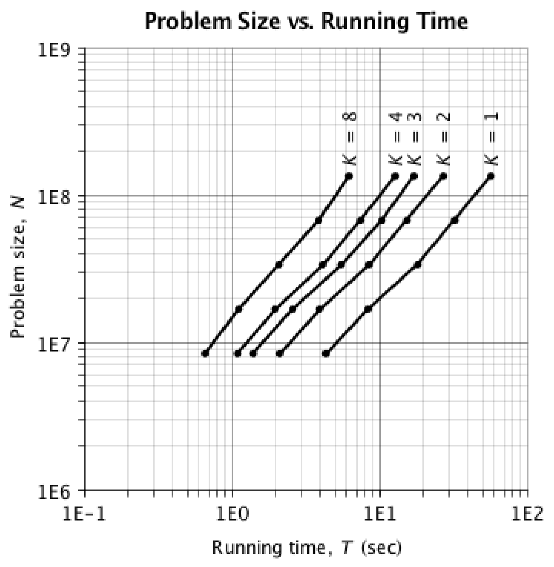
NT	Time (ms)	Speedup	Efficiency	ESDF
seq	76206			
1	55890	1.363	1.363	
2	25919	2.940	1.470	-0.072
3	16767	4.545	1.515	-0.050
4	12213	6.240	1.560	-0.042
8	6096	12.501	1.563	-0.018

7.1.2. Sizeup Metrics with a Guided Schedule

7.1.2.1. Exhaustive 3-SAT







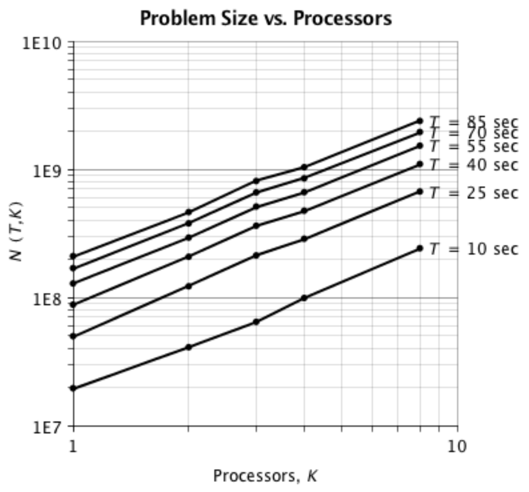


Figure 10. Sizeup performance metrics from the exhaustive 3-SAT program run with a guided schedule.

Table 11. Sizeup metrics for the exhaustive parallel program with 23, 24, 25, 26, and 27 variables and 1000 clauses using a guided schedule.

T	K	N	Sizeup	SizeupEff
10000	seq	14604712		
10000	1	19542277	1.338	1.338
10000	2	40846331	2.797	1.398
10000	3	64378965	4.408	1.469
10000	4	98871645	6.770	1.692
10000	8	241797258	16.556	2.070
25000	seq	31729660		
25000	1	49672838	1.566	1.566
25000	2	121820518	3.839	1.920
25000	3	212325763	6.692	2.231
25000	4	286152196	9.018	2.255
25000	8	669605571	21.103	2.638
40000	seq	60590308		

40000	1	87751074	1.448	1.448
40000	2	207200752	3.420	1.710
40000	3	361500772	5.966	1.989
40000	4	473432747	7.814	1.953
40000	8	1097413884	18.112	2.264
55000	seq	89020559		
55000	1	128406686	1.442	1.442
55000	2	292580986	3.287	1.643
55000	3	510675780	5.737	1.912
55000	4	660713297	7.422	1.856
55000	8	1525222197	17.133	2.142
70000	seq	116905128		
70000	1	169062298	1.446	1.446
70000	2	377961220	3.233	1.617
70000	3	659850789	5.644	1.881
70000	4	847993848	7.254	1.813
70000	8	1953030510	16.706	2.088
85000	seq	144789698		
85000	1	209717910	1.448	1.448
85000	2	463341454	3.200	1.600
85000	3	809025798	5.588	1.863
85000	4	1035274399	7.150	1.788
85000	8	2380838822	16.443	2.055

7.1.2.2. Decision 3-SAT

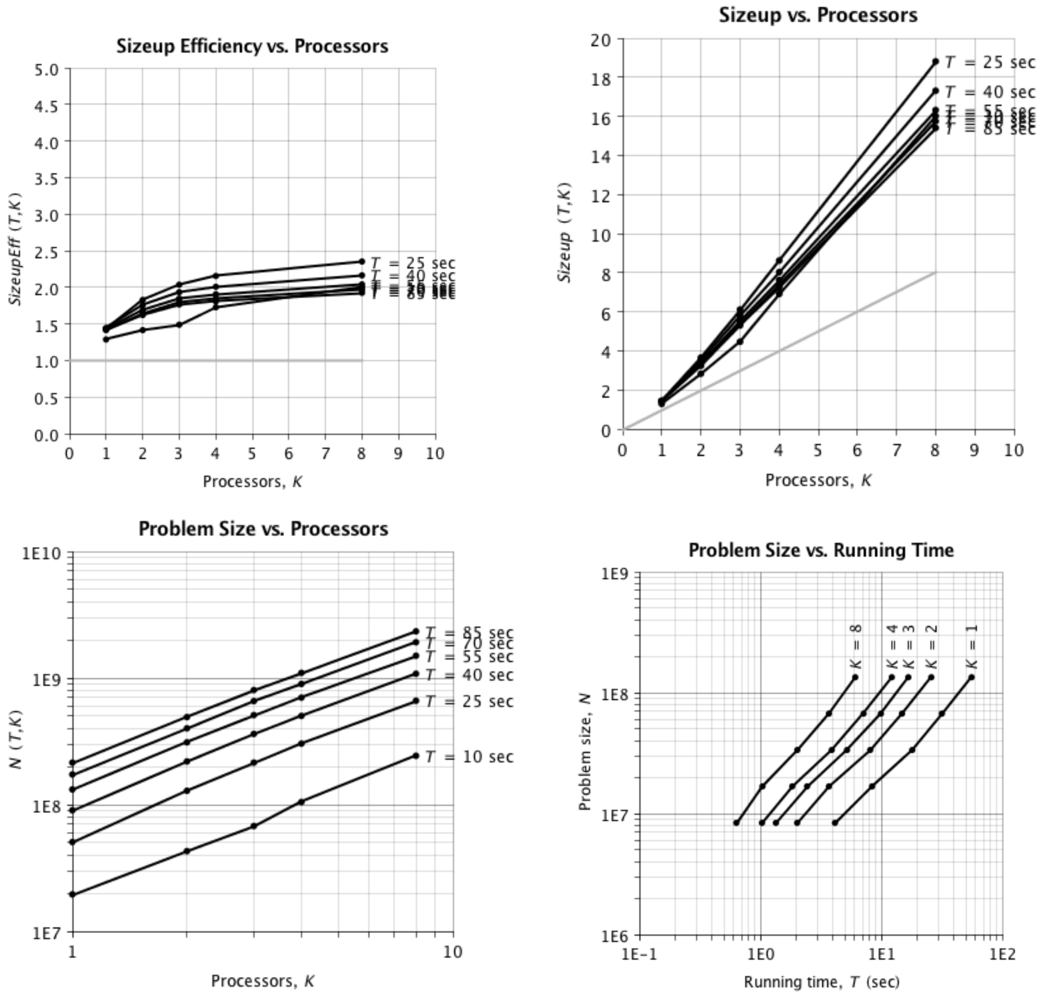


Figure 11. Sizeup performance metrics from the decision 3-SAT program run with a guided schedule.

Table 12. Sizeup metrics for the decision parallel program with 23, 24, 25, 26, and 27 variables and 1000 clauses using a guided schedule.

T	K	N	Sizeup	SizeupEff
10000	seq	15186596		
10000	1	19544430	1.287	1.287
10000	2	42998164	2.831	1.416
10000	3	67815376	4.465	1.488
10000	4	104942759	4.465	1.728
10000	8	243245196	16.017	2.002

25000	seq	35195978		
25000	1	50665492	1.440	1.440
25000	2	128654581	3.655	1.828
25000	3	215005457	6.109	2.036
25000	4	303372280	8.620	2.155
25000	8	662151962	18.813	2.352
40000	seq	62464180		
40000	1	90182386	1.444	1.444
40000	2	219456760	3.513	1.757
40000	3	362195537	5.798	1.933
40000	4	501801801	8.033	2.008
40000	8	1081058729	17.307	2.163
55000	seq	91927437		
55000	1	131751305	1.433	1.433
55000	2	310258940	3.375	1.688
55000	3	509385618	5.541	1.688
55000	4	700231322	7.617	1.904
55000	8	1499965495	16.317	2.040
70000	seq	121841347		
70000	1	173320225	1.423	1.423
70000	2	401061119	3.292	1.646
70000	3	656575698	5.389	1.796
70000	4	898660843	7.376	1.844
70000	8	1918872262	15.749	1.969

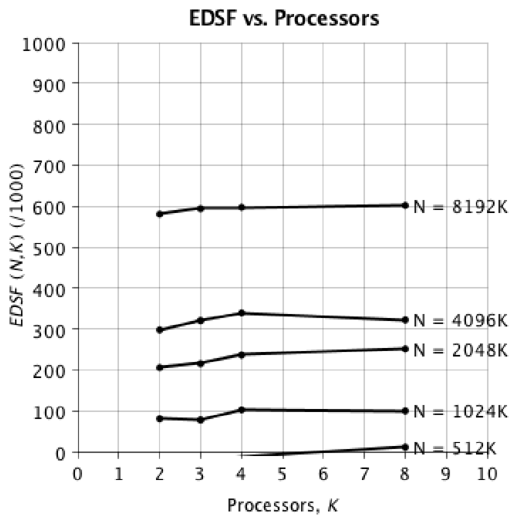
85000	seq	151755256		
85000	1	214889145	1.416	1.416
85000	2	491863298	3.241	1.621
85000	3	803765778	5.296	1.765
85000	4	1097090364	7.229	1.807
85000	8	2337779028	15.405	1.807

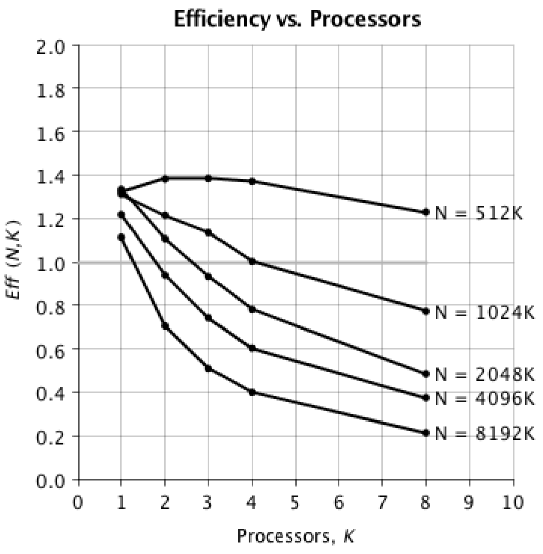
7.2. Clause Experiments

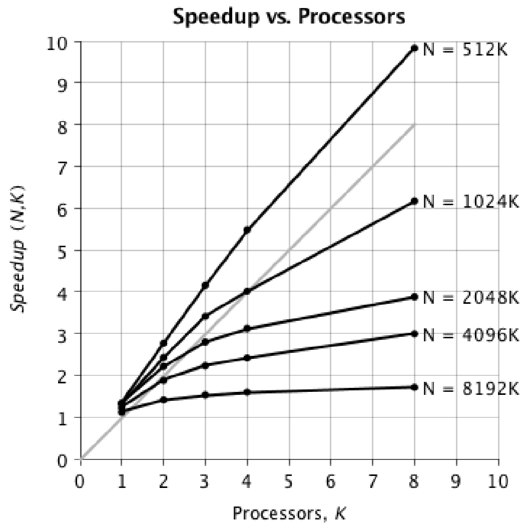
In this section we describe present the results from our performance analysis by fixing the number of variables to 25 and solving formulas with 512K, 1024K, 2048K, 4096K, and 8192K clauses. We present a complete set of speedup metrics for the best implementations of our parallel programs (i.e. those which used a guided schedule to balance the load among each thread).

7.2.1. Speedup Metrics with a Guided Schedule

7.2.1.1. Exhaustive 3-SAT







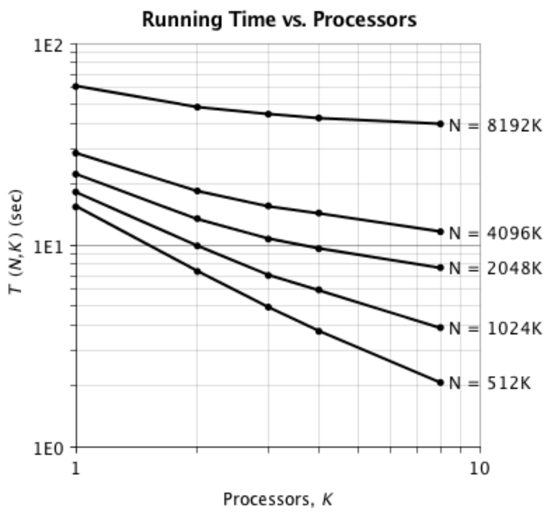


Figure 12. Speedup performance metrics from the exhaustive 3-SAT program run with a guided schedule.

Table 13. Speedup metrics for the exhaustive parallel program with 25 variables and 512K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	20409			
1	15417	1.324	1.324	
2	7383	2.764	1.382	-0.042
3	4913	4.154	1.385	-0.022
4	3722	5.483	1.371	-0.011
8	2076	9.831	1.229	0.011

Table 14. Speedup metrics for the exhaustive parallel program with 25 variables and 1024K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	23986			
1	18293	1.311	1.311	
2	9883	2.427	1.213	0.081
3	7043	3.406	1.135	0.078

4	5980	4.011	1.003	0.103
8	3887	6.171	0.771	0.100

Table 15. Speedup metrics for the exhaustive parallel program with 25 variables and 2048K clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	29890			
1	22405	1.334	1.334	
2	13500	2.214	1.107	0.205
3	10686	2.797	0.932	0.215
4	9572	3.123	0.781	0.236
8	7716	3.874	0.484	0.251

Table 16. Speedup metrics for the exhaustive parallel program with 25 variables and 4096K clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	34727			
1	28528	1.217	1.217	
2	18510	1.876	0.938	0.298
3	15601	2.226	0.742	0.320
4	14384	2.414	0.604	0.339
8	11610	2.991	0.374	0.322

Table 17. Speedup metrics for the exhaustive parallel program with 25 variables and 8192K clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	67998			
1	61070	1.113	1.113	

2	48248	1.409	0.705	0.580
3	44535	1.527	0.509	0.594
4	42617	1.596	0.399	0.597
8	39781	1.709	0.214	0.602

7.2.1.2. Decision 3-SAT

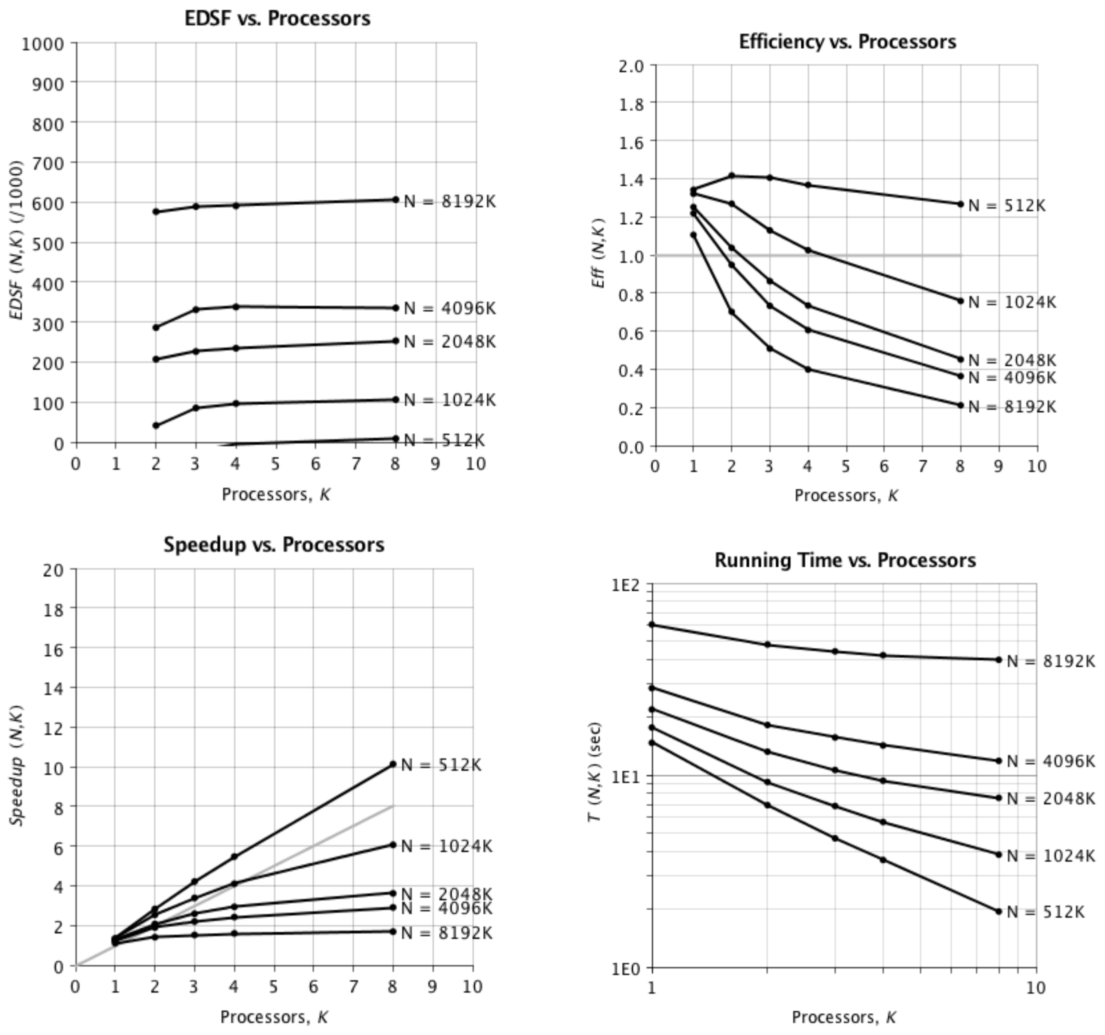


Figure 13. Speedup performance metrics from the decision 3-SAT program run with a guided schedule.

Table 18. Speedup metrics for the decision parallel program with 25 variables and 512K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF

seq	19720			
1	14706	1.341	1.341	
2	6961	2.833	1.416	-0.053
3	4681	4.213	1.404	-0.023
4	3605	5.470	1.368	-0.006
8	1947	10.128	1.266	0.008

Table 19. Speedup metrics for the decision parallel program with 25 variables and 1024K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	23340			
1	17677	1.320	1.320	
2	9202	2.536	1.268	0.041
3	6895	3.385	1.128	0.085
4	5695	4.098	1.025	0.096
8	3850	6.062	0.758	0.106

Table 20. Speedup metrics for the decision parallel program with 25 variables and 2048K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF
seq	27449			
1	21942	1.251	1.251	
2	13240	2.073	1.037	0.451
3	10620	2.585	0.862	0.226
4	9344	2.938	0.734	0.234
8	7600	3.612	0.451	0.253

Table 21. Speedup metrics for the decision parallel program with 25 variables and 4096K clauses using a guided schedule.

NT	Time (ms)	Speedup	Efficiency	ESDF

seq	34501			
1	28332	1.218	1.218	
2	18216	1.894	0.947	0.286
3	15693	2.198	0.733	0.331
4	14254	2.420	0.605	0.337
8	11850	2.911	0.605	0.335

Table 22. Speedup metrics for the decision parallel program with 25 variables and 8192K clauses using a guided schedule.

NT	**Time (ms)**	**Speedup**	**Efficiency**	**ESDF**
seq	67132			
1	60751	1.105	1.105	
2	47867	1.402	0.701	0.576
3	44082	1.523	0.508	0.588
4	42085	1.595	0.399	0.590
8	39800	1.687	0.211	0.606

7.3. Performance Analysis

Using a guided schedule, our parallel programs achieved superlinear speedups and efficiencies above 1.0 for K = 1, 2, 3, 4, and 8 with all data set sizes. The reason for these speedups is clear: the JIT enabled early machine-code compilation to yield faster execution times than the sequential version, and the use of cache memory to store the two-dimensional array of literals greatly reduced the memory access time. Since some formulas may be deemed unsatisfiable with a given variable configuration after checking the only the first clause, the amount of cache thrashing appears to be reduced for random instances of 3-SAT formulas. That is, on average, a thread will only need to examine the first section of the formula to determine if it is satisfiable, and this information can be stored in cache without being flushed very often. Also, notice that the efficiencies decrease as the number of processors increase. This is a direct result of Amdahl's law, which states that the speedups approach the asymptotic limit of 1/F as the number of processors increases. In our programs, the sequential fraction is only composed of the code needed to read in command-line arguments, set up the parallel team, perform the reduction, and then print out the solution. This fraction is so small compared to the parallel portion (as indicated by our negative ESDFs), that our approach to this limit is much slower than expected.

With regards to sizeup on the programs using the guided schedule, the results were extremely superlinear. In particular, in a configuration with 23 literals, and 8 processors, the calculated sizeup achieved was 16.017 - an improvement that exceeds the expected ideal size up of 8 by a factor of two. This can be attributed to the configurations being stored into the processor's cache memory, allowing for extremely quick access to the underlying data structures used our solver algorithm. Furthermore, Java's just-in-time compiler applies optimizations to certain functions in our algorithm, in particular the decide() function, which is called multiple times by each thread, causing the JIT to apply optimization quickly.

8. Lessons Learned

Over the course of this quarter, several things were learned, both in terms of the application of various parallel paradigms towards a parallelizable problem, as well as the general process one should follow when doing runtime analysis on problems as a whole.

As far as scheduling was concerned, both a dynamic scheduler and a guided scheduler were used. The dynamic scheduler ended up being detrimental to runtimes, as the granularity of scheduled chunks was extremely small, resulting in a large amount of scheduling overhead. Guided scheduling, on the other hand, was extremely effective in reducing the computation time for the parallel version of the solution. To take a particular example, for 27 literals (using the not-exhaustive approach) the guided scheduler took 6.1 seconds to complete, but when using the dynamic scheduler, the computation took 148.8 seconds. In fact, the amount of scheduling overhead is so large, that the speedup actually *decreases* as the number of processors increase.

When running tests on parallel programs, it is extremely important for the tests to be structured such that only a single variable gets altered, while the rest stays consistent - a necessary approach that ensures results change, and are thus dependant, on a single factor. In the case of our 3-SAT study, the resulting runtime had several influencing factors: the number of literals present in a particular configuration, the number of clauses present in a particular configuration, and the type of scheduler (if one was used). While the number of variables is clearly a more dominant factor in terms of runtime and complexity, variations in more than a single variable during testing would invalidate the certitude in the results. It was therefore necessary to conduct multiple sets of tests.

The proper way to approach the problem of performing analysis where multiple factors are a concern is to perform multiple sets of tests, and to be exhaustive with all of the possibilities of variables being tested. Furthermore, selecting ranges for these variables is very much a process of trial and refinement. While for some tests, these ranges are obvious, for example in the case of testing a sizeup, one ought to double the problem size for every added processor to test against ideal sizeups, but this is not so clear for setting the other constant factors. For example, if one is testing the impact of clauses on the runtime of a configuration set, there is no secret to selecting a number of literals used to generate the set of configurations. In our case, choosing 24 for the number of literals was the result of many tests, and weeding out values that led to runtimes that were too large, or runtimes that were too small (leading, in turn, to configurations that exhausted the JVM's memory due to the storage of the clauses themselves).

9. Future Work

There are many avenues of future work for our project. First, we could pursue cluster- and GPU-based implementations of the parallel programs. Doing so would require us to implement the master-worker pattern for distributing parts of the SAT search space among the different processors or threads available. We could also consider experimenting with different data structures and measure the performance impact for each possible solution. In this work we made the assumption that our choice of data structure adhered to the principle of locality of reference. However, this assumption was not empirically verified. We could also attempt to implement more advanced heuristics and integrate them into the scheduling logic of the master-worker pattern. Doing so would help tailor our programs to particular types of 3-SAT formula instances. To verify and profile these new implementations we could devise hard 3-SAT formula instances from industry and scientific problems to feed into our programs. Alternatively, we could use provided 3-SAT formulas provided on the SAT competition website [3].

10. Individual Contributions

Christopher
Contributed to the sequential and parallel programs, wrote and prepared part of the report, gathered the speedup and sizeup metrics, generated test files, and helped debug the output formatting scripts.

Eitan
Contributed to sequential and parallel programs, wrote and prepared part of the report, and wrote the output formatting scripts.

Ankur
Contributed to sequential and parallel programs, wrote and prepared part of the report, and generated test files.

References

[1] Gormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms. MIT Press* 44 (1990), 97-138.

[2] Satisfiability: Suggested Format, 1993. Available online at: `http://people.sc.fsu.edu/~jburkardt/pdf/dimacs_cnf.pdf`. Accessed: 3/9/13.

[3] The International SAT Competitions web page. Available online at:
`http://www.satcompetition.org/.` Accessed 4/30/13.

[4] Zhang, Hantao, and Mark Stickel. "Implementing the Davis–Putnam Method."*Journal of Automated Reasoning* 24.1-2 (2000): 277-296.

[5] Meyer, Quirin, et al. "3-SAT on CUDA: Towards a massively parallel SAT solver." *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010.

[6] Hamadi, Youssef, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver." *Journal on Satisfiability, Boolean Modeling and Computation* 6.4 (2009): 245-262.

[7] Zhang, Hantao, Maria Paola Bonacina, and Jieh Hsiang. "PSATO: a distributed propositional prover and its application to quasigroup problems." *Journal of Symbolic Computation* 21.4 (1996): 543-560.

[8] Luby, Michael, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms." *Information Processing Letters* 47.4 (1993): 173-180.

[9] SATLIB Benchmark Problems. Available online at:
`http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`. Accessed 4/30/13.