Parallel DPLL Implementation in Chapel

Tayfun Elmas

University of Washington
Computer Science & Engineering
telmas@cs.washington.edu

Abstract. This paper talks about the author's experience with the Chapel parallel programming language in implementing a parallel implementation of the DPLL algorithm for solving the satisfiability (SAT) problem. After we briefly introduce the SAT problem and the DPLL algorithm, along with its parallelization, we indicate the pros and cons of the Chapel languages that we encountered during the implementation, compare it with MPI and give recommendations for the language specification.

1 Introduction

The boolean satisfiability (SAT) problem is one of the most common problems in many disciplines including computer science and industrial engineering. The reason behind this is that lots of problem instances in various research areas can be encoded as SAT instances. This paper talks about experiments the author had while implementing a parallel version of the DPLL algorithm, a well known algorithm for propositional SAT solving, in the Chapel programming language. We first introduce the SAT problem and the DPLL algorithm, along with its parallelization, and then indicate the pros and cons of the Chapel language that were encountered during the implementation.

The goal of a SAT solver is to find a satisfying assignment to a boolean formula, i.e. to make formula evaluate to boolean `TRUE`. Section 2 formally states the SAT problem. Although there are generic algorithms that work on formulas in arbitrary forms, most SAT solvers work on the conjunctive normal form (CNF) of the formula. In the CNF format, the formula is expressed as a set of clauses connected by the `AND` operator. Each clause consists of a set of literals, connected by the `OR` operator. A literal is a boolean formula with only a positive or negated form of a variable. The CNF form makes the SAT problem a special case of constraint satisfaction problems (CSP) and the generic algorithms developed for CSP problems can be applied to SAT instances.

The *complete solution* to a CNF formula is a *complete assignment*, which assigns a boolean value to all the variables and satisfies all the clauses in the formula. The goal of SAT solving is to declare a formula satisfiable and give a complete solution, or to declare it as unsatisfiable.

Although, the SAT problem is one of the well-known NP-complete problems [2], there are very efficient solvers in the literature, some of them are widely-used in industrial applications. The essential algorithm is the DPLL algorithm, developed by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland in

1962 [3], which is a refinement to the Davis-Putnam algorithm [4]. The DPLL algorithm contains some heuristics and optimizations to increase the runtime efficiency of the algorithm. Section 3 explains these features of DPLL. There are also further optimizations to the core algorithm [5].

This paper is organized as follows. We first give a formal introduction to the SAT problem in Section 2. Section 3 explains the DPLL algorithm and its parallelization as implemented by the author. We talk about our implementation in Chapel in Section 4, as well as comments about the specification that we found necessary to mention.

2 The Satisfiability Problem

This section formally explains the SAT problem. The following section defines the problem in general case, while Section 2.2 redefines the problem for the CNF form of formulas. Finally, Section 3 gives information about the DPLL algorithm for solving the problem exactly.

2.1 Boolean Formula Satisfiability

In general case, a SAT problem is defined with a set of boolean variables X and an arbitrary formula F in first-order propositional logic on X. The domain of variables in X is $\{0,1\}$ or $\{FALSE, TRUE\}$[1]. The formula F is constructed with the boolean connectives including $\wedge$ (AND), $\vee$ (OR), $\implies$ (IMPLIES), and the negation operator $\neg$.

An *assignment* gives boolean values to variables in X. An assignment in which all the variables in X are assigned is called a *complete assignment*. Given a complete assignment A, the truth of the formula is determined by the semantics of first-order propositional logic with respect to the valuations of the variables in A. The aim of solving the SAT problem is thus finding a complete assignment A that will evaluate the formula F to 1.

2.2 Conjunctive Normal Form (CNF)

The SAT solvers in the literature handle the formulas in a special form called the conjunctive normal form (CNF), rather than handling them in their arbitrary structure. The CNF form of a formula F consists of conjunction of a set of clauses $C = c_1 \wedge c_2 \wedge ... \wedge c_{|C|}$. Each clause c_i is disjunction of one or more literals: $c_i = (l_1 \vee l_2 \vee ... \vee l_{|c_i|})$. A literal l is an atomic formula that consists of either a variable $x \in X$ or its negation $\neg x$.

If the number of literals in each clause is limited to at most k literals, it is called the $k-CNF$ SAT problem. While $2-CNF$ SAT is polynomially solvable, $k-CNF$ SAT for $k \neq 3$ is NP-complete. Throughout this paper, we refer by the SAT problem to $k-CNF$ where $|c_i| \geq 3$ for all $c_i \in C$.

[1] In this paper, we will use the notation $\{0,1\}$ for the domain of the variables (1 refers to $TRUE$ and 0 refers to $FALSE$).

In the CNF form, the SAT problem is in fact a special kind of the Constraint Satisfaction Problem(CSP): a SAT instance is a CSP instance where the literals are constraint variables, the common domain is $\{0, 1\}$ and the clauses are constraints along with the additional constraints $\forall i.\ (p_i \neq \bar{p}_i)$. Thus any kind of algorithms and heuristics developed for for CSP problems can be applied to the SAT solving problem in a straightforward manner [6]. However, how to exploit the heuristics for CSP problems for solving the SAT problem instances is an open question and problem-specific SAT solving algorithms, including DPLL, are more preferable in this sense.

Since most of the metaheuristics may not detect unsatisfiability, or it can be high-costly to decide unsatisfiability, some users tend to satisfy as many as possible clauses rather than satisfying the entire formula. This kind of problem is known as the MAX-SAT problem [7], where each clause c_i is associated with some weight w_i and the problem is to maximize the sum $\sum_{\forall c_j \in C} eval(c_j, S)w_j$ ($eval(c_i, S)$ returns 1 if S satisfies c_j, 0 otherwise).

3 The DPLL Algorithm

3.1 The Sequential Algorithm

```
procedure DPLL(F, A)
   if F evaluates to 1 then
       return SATISFIABLE;
   if F has an empty clause then
       return UNSATISFIABLE;
   if there is a unit clause (l) in F then
       return DPLL(assign(l,F), A and l);
   if there is a pure literal l in F
       then return DPLL(assign(l,F), A');
   l = choose an unassigned literal from F;
   return DPLL(assign(l,F), A') OR DPLL(assign(NOT(l),F), A');
```

Fig. 1. The DPLL algorithm

The DPLL algorithm has been used widely as the core algorithm for many SAT solvers. In addition to the fundamental optimizations, many other improvements have been used to increase the efficiency of the algorithm. For example, the Chaff algorithm [5] contains many of the extensions to the core algorithm. MiniSAT and zChaff tools are one of the most widely-used SAT solvers.

Figure 1 shows an outline of the DPLL procedure. The algorithm is based on dividing the SAT solving problem into smaller problems by *splitting* the formula depending on the truth values of the clauses. Splitting assigns values to variable that generate partial assignments and propagates these partial assignments along with the formula to the same procedure in a recursive manner. It also does backtracking so that the possible assignments are enumerated until the formula is determined to be satisfiable or unsatisfiable: Satisfiability is determined when all the clauses are satisfied by a complete assignment. Unsatisfiability can only be determined after the exhaustive search ends.

The algorithm uses a simplification mechanism. The procedure `assign(l,F)` assigns a value to the variable that appear in the literal `l` a value that makes `l` evaluate to 1. Then it simplifies the formula `F` according to the current partial assignment. As a result, the recursive procedure enumerates the possible assignments by assigning a value to a variable at a time and checking the resulting simplified formula in a recursive call to the same procedure.

DPLL uses two optimizations called *unit propagation* and *pure literal elimination.* In the former, if a clause c_i contains only one literal, the corresponding variable of the literal is assigned so that c_i is satisfied. In the latter case, if a variable v_i appears in the formula only in its positive or negated form, v_i is assigned that so that its corresponding literals evaluate to 1.

In addition to the unit propagation and the pure literal elimination, most of the SAT solvers implement a conflict detection and backtracking scheme [1]. In this scheme, during the search for a satisfying assignment, when a conflict is detected, i.e. when a clause becomes unsatisfiable, the solver generates a clause called conflict-induced clause that justifies the cause of the conflict. In this case, the solver backtracks to a higher branch in the search tree without the conflict and adds the conflict-induced clause to the existing set of clauses in order avoid having assignments that would cause the same conflict.

In spite of the optimizations such as unit propagation and pure literal elimination, in the worst case the algorithm enumerates all possible assignments before reaching a conclusion. This is a fundamental motivation for developing a parallelization for the algorithm. The following section explains how the parallel version of the algorithm searches the solution space.

3.2 The Parallel Solver

In the sequential version of the algorithm, a single processor traverses the entire search tree for a satisfying assignment. We first developed the sequential algorithm before proceeding with the parallelization.

In the parallel version of the algorithm, we distribute the task of searching different branches of the search tree to processors in the system. In this way, when a processor reaches a point where it branches on that point of the tree by performing a decision on the value of a variable, say v, it tries to have a new processor search the other branch. If the maximum number of allocated processors has not been reached, it allocates a new processor with the task of searching the other branch of the tree with the opposite value of v. The new processor inherits the assignments done by the parent up to that point but keeps its further assignments separately in another object.

When a processor detects a conflict, it computes the conflict-induced clause as regular and adds it to the global set of clauses. This allows other processors to be aware of the conflict and thus to avoid doing the same assignments. If that processor determines that it needs to backtrack to a depth that is higher than the one it was allocated, it terminates its search at that point. The purpose of this decision is that the processor is only assigned the branch that starts from the starting depth and it has no information about how to proceed from the upper depths. Note that because the main processor starts from depth 0, there is at least one processor that keeps doing the search.

4 The Implementation in Chapel

In this section, we comment about the advantages of Chapel in implementing the parallel DPLL algorithm and list recommendation about the language specification.
Used features Our implementation used the following features of the Chapel language:

- *Arrays, domains, sub-domains:* We had domains for clauses and variables. We stored the clauses and variable assignments in arrays created using these domains. Each clause had a sub-domain of the the variables domain.
- *Object-oriented programming:* We created Clause, Assignment, IGNode and Statistics classes to represent a clause, a set of assignment to variables, an individual assignment to a variable, and runtime measurements, respectively.
- *Tuples:* We used tuples to return more than one value from global functions.
- *Task-parallelism:* We created parallel tasks to search different branches of the assignment space using the `begin` construct.
- *File I/O:* We read the propositional formulas from files stored in the standard CNF file format.
- *Time module:* We used the Timer class in the Time module to measure the elapsed time.
- *Constants:* We used compile-time constants DEBUG and STAT to control debugging and collecting statistics, as well as runtime constants FILE and MAX-PROCS to allow the user to determine the file name to read the formula from and the maximum number of procedures to allocate.
- *Synchronization variables:* We use synchronization variables to allow atomic allocation of search branches to processors (see `alloc_proc`) and addition of new learned clauses to the global set of clauses (see `atomic_add_clause`).

Global-view programming: Chapel's view of parallelization helped a lot while devising the parallel search scheme of our implementation. In fact, we started parallelization from a sequential program. However, transferring a sequential code to a parallel version in MPI is not easy. The reason behind is the transparent access to global variables in Chapel, whereas MPI require management of explicit distribution and storage of data across processes. When you convert a sequential procedure to one that is run by different parallel processing units, you do not need to change the accesses to the global data except accesses you want to make more efficient by considering the distribution of the data. In contrast, while you convert a sequential process to a parallel one, you need to implement the communication mechanism (either by send/receive primitives or explicitly setting up some shared memory).

In addition, flexility of MPI for different number of locales may require extra work to adjust the system configuration most of the time, as it affects data distribution and communication policies. However, Chapel is flexible to any number of locales; unless the programmer explicitly specifies the data and task distribution, the compiler automatically make assignments of data and tasks to locales. In our implementation, we defined our arrays for clauses and assignments as block-distributed arrays but did not explicitly specify the locales for data distribution and parallel search tasks.

Data parallelism: We used data parallelism while applying the same computation to all clauses in the formula in an iteration over the clauses. In this way, we checked satisfiability of the formula by checking satisfiability of each clause, or searched for conflicts or unit clauses by distributing the search on each clause in a `forall` construct.

We raise one issue at this point. Most of our computations as indicated above consisted of checking a condition, e.g. satisfiability, on each clause. When the condition is not satisfied we required that no other clauses need to be checked for the condition, as only one clause that does not satisfy the condition suffice to decide the overall result. For example, to check satisfiability of a formula, we check each clause to see whether the current assignment satisfies the clause. If all the clauses are satisfied, the entire formula is satisfied. However, if at least one clause is not satisfied, this is sufficient to decide that the formula is not satisfied yet, so other processors checking other clauses can safely be terminated at this point. However, the specification states that commands like `break` or `return` cannot be called inside the `forall` loops. Therefore, we decided to have global boolean flag to indicate whether a final conclusion has been reached, if so each process check this flag using a conditional (`if`) and terminates terminates the iteration. The author suggest a mechanism to support these kinds of computations, in which a processor could reach a final conclusion and have others terminate.

Task parallelism: Task parallelism in MPI requires more than just statement of what to be performed, such as communication and synchronization between the parent and the forked threads. This makes each creation of a parallel task very hard for the programmer. In contrary, Chapel makes it extensively easy to create a task, just write the code that will be executed in parallel and the task will use the global variables as in the sequential case. However, the synchronization issues between the parent and the children threads should be made more clear as unexpected situations, such as data races may happen because of this transparent mechanism of parallelism. As indicated below, for example, the side effects of parallel executions of iterations and data accesses in them must be strictly specified. In MPI, because data accesses across processes require explicit communication (except the new mechanisms for shared memory), the programmer is aware of what is shared and what is not during the execution.

Synchronization inside loops: The specification states that the iterations in `forall` loop can be performed in parallel. In this case one issue is accesses to shared variables inside iterations. For example, if there is a write to a global variable in the `forall` body, two iterations run in parallel will attempt to write to the same variable simultaneously. Does the programmer have to be aware of this situation, and perform synchronization to prevent the data race? The same issue happens when the processes that run iterations of the `forall` loop access local variables of the procedure in which the loop is defined.

Synchronization mechanisms: Synchronization variables allow programmers to devise coarser-level synchronization idioms like mutexes, semaphores, readers/writers locks. We use synchronization variables to obtain mutual exclusive code blocks that update shared data (global array of clauses and current number of processors). However, the author suggests having a standard library of common synchronization idioms such as the one for Java (`java.util.concurrent` package, implemented using the primitive synchronization variable. This will prevent programmers to make mistakes while building these mechanisms with the primitive synchronization variables.
Copy-on-access for global variables: In our implementation we used several global arrays that are accessed by processes running parallel searches simultaneously. Clauses array, for example, does not change while the program is running. Most of the procedures read from this array
Conflicts with the C language: Variable and constant names conflict with the ones in the standard C library. For example, the author could not declare a constant named `FILE` which conflicts with the definition in `stdio.h`.

References

1. J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, 1999.
2. Cook, S. A. The complexity of theorem-proving procedures. In Proceedings of the Third Annual ACM Symposium on theory of Computing (Shaker Heights, Ohio, United States, May 03 - 05, 1971). STOC '71. ACM Press, New York, NY, 151-158.
3. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. Commun. ACM 5, 7 (Jul. 1962), 394-397.
4. Davis, M. and Putnam, H. A Computing Procedure for Quantification Theory. J. ACM 7, 3 (Jul. 1960), 201-215.
5. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: engineering an efficient SAT solver. In Proceedings of the 38th Conference on Design Automation (Las Vegas, Nevada, United States). DAC '01. ACM Press, New York, NY, 530-535.
6. Kumar, V. Algorithms for constraint-satisfaction problems: a survey. AI Mag. 13, 1 (Apr. 1992), 32-44.
7. Roli, A. Design of a New Metaheuristic for MAXSAT Problems. In Proceedings of the 8th international Conference on Principles and Practice of Constraint Programming (September 09 - 13, 2002). P. V. Hentenryck, Ed. Lecture Notes In Computer Science, vol. 2470. Springer-Verlag, London, 767.