Introduction to Parallel and Distributed Computing Group Project (June 27)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- a PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address (for all members of the group);
  - the source code of the sequential program(s),
  - the demonstration of a sample solution of the program(s),
  - the source code of the parallel program(s),
  - the demonstration of a sample solution of the program(s),
  - benchmarks of the sequential and of the parallel program(s) in the style of Exercise 1.
- the source (`.c`/`.java`) file(s) of the programs.

The Satisfying Assignments of a Propositional Formula

The goal of this project is to develop parallel programs for determining the number of satisfying assignments of a propositional formula in conjunctive normal form.

Preliminaries

A *propositional formula* F in *conjunctive normal form* is a conjunction of clauses where each *clause* C is a disjunction of literals and each *literal* L is a positive or negative occurrence of a propositional *variable* V. F is thus formed according to the following extended BNF grammar:

$$\begin{array}{lcl} F & ::= & [\ C\ (\wedge\ C)^*\ ] \\ C & ::= & [\ L\ (\vee\ L)^*\ ] \\ L & ::= & V\ \mid\ \neg V \end{array}$$

An example of such a formula is

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y) \wedge (y \vee \neg z).$$

which can be also represented by the set

$$\{\{x, \neg y, z\}, \{\neg x, y\}, \{y, \neg z\}\}.$$

If a clause has both a positive occurrence V and a negative occurrence $\neg V$ of the same variable V, it is equivalent to "true" and can be discarded; we thus may assume that no clause has such conflicting occurrences.

An *assignment* A is a set of n literals such that every variable V occurs in A either in positive or in negative form: $V \in A$ indicates that variable V is assigned the truth value "true" while $\neg V \in A$ indicates that it is assigned the truth value "false". An assignment A *satisfies* a formula F, if the formula becomes true when all the variables are replaced by the truth values indicated by the assignment. In more detail, assignment A satisfies clause C, if there exists a literal L in A that also occurs in C; A satisfies F, if it satisfies every clause in C. For instance, above formula is satisfied by the assignment $A = \{x, y, z\}$.

In order to determine whether there exists any satisfying assignment for a formula F with n variables, the DPLL algorithm[1] depicted in Algorithm 1 may be applied: if F is empty, it represents the formula "true" and is thus satisfiable; if F has an empty clause, this clause represents the formula "false", and F is not satisfiable. Otherwise, we choose some literal L that occurs in F and apply the algorithm recursively, first to formula $F \wedge L$ and, if necessary, also to formula $F \wedge \neg L$. Each of these formulas has $n - 1$ variables, because the computation of $F \wedge L$ removes every negative occurrence of L from F and removes every clause C from F that has a positive occurrence of L (dually for the computation of $F \wedge \neg L$). Thus the recursive call DPPL($F \wedge L$) determines whether there exists any satisfying assignment for F which sets L to "true"; the recursive call DPPL($F \wedge \neg L$) determines whether there exists any satisfying assignment for F which sets L to "false". The execution of the algorithm can be described by a binary tree with 2^n leaves that represent all possible assignments; it thus represents an exhaustive search for a satisfying assignment in the space of all possible assignments.

[1] https://en.wikipedia.org/wiki/DPLL_algorithm

Algorithm 1 DPLL Algorithm (simplified)

function DPLL(F, n) ▷ Formula F has n variables
  if F is empty **then**
    return true
  else if F has an empty clause **then**
    return false
  else
    choose literal L in F
    return DPPL($F \wedge L, n-1$) **or** DPPL($F \wedge \neg L, n-1$)
  end if
end function

The algorithm can be easily extended to return the number of satisfying assignments by the following changes:

1. In first base case, it returns 2^n (if there are still n unassigned variables, there are 2^n possible assignments to these variables).

2. In the second base case, it returns 0 (since there is no assignment).

3. In the recursive case, it returns the sum of the values returned by the two recursive calls.

In the actual implementation of the algorithm by a computer program, a formula F with c clauses in v variables can be represented by a $c \times v$ matrix of elements $\{-1, 0, +1\}$ where $F[i, j]$ has value $+1$, if clause i has a positive occurrence of variable j, value -1, if it has a negative occurrence, and value 0, if variable j does not occur in clause i. Above formula can be thus represented by the matrix

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \end{bmatrix}.$$

Rather than constructing a new formula matrix F for every recursive invocation of the algorithm, we may pass to every invocation just a triple c', l, a where

- c' is the number of clauses that are still in F,
- l is an array of length c where $l[i]$ is the number of literals in clause i of F (the special value $l[i] = -1$ indicates that clause i has been deleted from F; there are $c - c'$ such lines);
- a represents the *partial assignment* computed so far: it is a vector of length v of values $\{-1, 0, +1\}$ (initially 0 everywhere): $a[i] = +1$ indicates that variable i has been assigned the truth value "true", $a[i] = -1$ indicates that it has been assigned the truth value "false"; $a[i] = 0$ indicates that it has not been assigned a truth value yet.

All necessary operations of the algorithm may be expressed in terms of these three variables.

Project

The project consists of several tasks.

Java-based Solution Implement a sequential and parallel solution of the problem in Java.

The sequential solution shall take command line parameters `-w -p` c v d s where c is the number of clauses, v is the number of variables, d is a density value between 0 and 1, and s is an integer that represents the seed for the random number generator. The program shall generate a random formula matrix of dimension $c \times v$ of which a fraction d is filled with random literals using seed s for the random number generator (to save space, use type `byte` for the elements of the matrix). The program computes the number of satisfying assignments and prints this number to the standard output. If the option `-w` is given, the wall clock time of the core computation (in ms) is printed to the standard output. If the option `-p` is given, also the formula matrix and the satisfying assignments are printed. Demonstrate the correctness of your solution by showing the result for some (small) formula and benchmark the solution for three different inputs of reasonable size.

The parallel solution shall also accept a parameter `-t` T that states that T concurrent threads shall be applied. Use for this solution the classes `ForkJoinTask` and `ForkJoinPool`[2]. Benchmark the program in the style of the other exercises for $T = 1, 2, 4, 8, 16, 32, 64$ threads and the same inputs as for the sequential program.

Cilk-based Solution Implement a sequential and a parallel solution of the problem in C/C++.

The sequential solution proceeds analogously to the Java-solution (use type `signed char` for the element values). Please note that in C/C++ it is not necessary to allocate the arrays l and a on the heap, you may just allocate them on the stack by local declarations of the corresponding variables (which is much faster and saves the effort of explicitly deallocating the arrays again).

The parallel solution shall be based on Cilk Plus in the most natural (recursive) style.

OpenMP-based Solution Implement a sequential and a parallel solution of the problem in C/C++. The sequential C/C++ solution is the same as described above.

The parallel solution shall be based on OpenMP which creates T threads; this may be achieved by the OpenMP pragma

```
#pragma omp parallel ... num_threads(T)
```

Each thread maintains a stack of partial assignments; an array s of length T holds these stacks (thread i pushes and pops elements from $s[i]$); likewise, an array r of length T holds the number of assignments computed by the threads (thread i updates $r[i]$).

In more detail, each thread i runs in a doubly nested loop:

1. in the outer loop, the thread pops a partial assignment a from its stack and determines from F, n, a the values l and c with which it starts the execution of the inner loop;

[2] See https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html and the documentation of these classes

2. in the inner loop, the thread proceeds as follows:
   a) the thread performs the checks for the "base cases" and only proceeds with the loop, if they are not successful;
   b) for one of the "recursive" cases, it allocates on the heap a new partial assignment a' which it pushes on the stack;
   c) the other "recursive" case with assignment a'' is processed by the thread itself in the next iteration of the loop;
3. the thread leaves the inner loop when it reaches the "base case" where it can determine the number of total assignments arising from the current partial assignment a''; the thread correspondingly increments the value of $r[i]$.

Initially, a single empty assignment is created and pushed on the stack of thread 0 before all threads are activated. Now, whenever a thread encounters an empty stack, it (repeatedly) scans the stacks of the other threads to "steal" partial assignments from them; each access to a stack has therefore to be protected by a lock variable of type `omp_lock_t` with operations `omp_set_lock` and `omp_unset_lock` (an array of T such lock variables is thus needed; variable i is locked, before an element is pushed/popped to/from stack i).

A single shared variable is set to indicate the number of threads that are waiting for a partial assignment arising from another thread (access to this variable has to be protected by another lock variable or simply by using the OpenMP pragma `#pragma omp critical`). If this variable indicates that all threads are waiting, the threads may terminate and the total number of partial assignments may be computed as the sum of all values $r[i]$.

Remark: most probably the OpenMP-based solution will perform significantly worse than the Cilk-based solution; it is the comparison of Cilk/OpenMP that is of interest for this task.

Project Results

You may perform the project in groups of n persons with $1 \leq n \leq 3$. If $n = 1$, only one (e.g., the Java-based) solution has to be elaborated. If $n = 2$, then two solutions (e.g., the Java- and the Cilk-based one) have to be elaborated. If $n = 3$, all three solutions are required. However, all members of a group will receive the same grade for the project, thus all members should feel responsible for the quality of all solutions of the group.

Each group has to give a short (about 10 min) presentation to demonstrate its results (by one or more of its members) in the last lecture on

Monday, June 27, 2016.

This date also represents the deadline for the submission of the results in Moodle (by one member of the group; the submission must indicate all members of the group).

Alternative Deadline: upon common request, we may also shift the deadline of the assignment and the presentations (of all groups) to a date in July or October.