

Homework 4: Parallel Constraint-Satisfaction Search

Instructor: Vivek Sarkar

Assigned March 7, 2012, due by 11:55pm on Wednesday, March 21, 2012

(1 programming problem totaling 100 points)

All homeworks should be submitted in a directory named “hw_4” using the turn-in script. It is important that you start early on this homework to meet the deadline. Sections 1 and 2 contain background information. Your assignment is in Section 3.

This revised handout includes four clarification notes in italics below.

Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone else’s work as your own. If you use any material from external sources, you must provide proper attribution.

1 Constraint Satisfaction Search algorithms

Constraint-satisfaction problems arise frequently in several applications areas including puzzle-solving and engineering design. These problems are computationally intensive and well suited for speedup through parallel processing. This assignment explores parallelization of *constraint-satisfaction search* algorithms that use *forward checking*. It is well known that some form of look-ahead, as in forward checking, reduces the sequential execution time for the application thereby making it a more credible candidate for parallelization than simple backtracking. This assignment will focus on the use of constraint-satisfaction search in puzzle-solving, with n-queens and Sudoku puzzles as two use cases.

In general, an intermediate state of a constraint-satisfaction search is characterized by a *partial Problem State* in which some variables have a single assigned value, and a *Feasible Value Table* (FVT), that provides a set of possible values for the remaining *free* variables. If the set becomes empty for any variable, then it implies that no feasible solution can be derived from the given intermediate state. If an FVT has exactly one value per variable, then it can be combined with the *partial Problem State* to obtain a *complete Problem State*.

In the sequential code given to you, you can find the constraint-satisfaction search code in method `search()` of `ConstraintSatisfaction.hj`, which is also shown in Listing 1 below.

```
1  public void search(ProblemState state, int curVar, FVT fvt) {
2      if (curVar == fvt.getNumVars())
3          problem.addSolution(state); // feasible solution found
4      else {
5          Iterator<Integer> itr = fvt.getValues(curVar).iterator();
6          while(itr.hasNext()) {
7              Integer v = itr.next();
8              ProblemState newState = state.copy();
9              newState.setValue(curVar, v);
10             FVT newFvt = forwardCheck(curVar, v, fvt);
11             if (newFvt != null) search(newState, curVar+1, newFvt);
12         } // while
13     } // if
14 } // search()
```

Listing 1: `search()` method in `ConstraintSatisfaction.hj`

In this method, parameter `state` (of type `ProblemState`) contains the partial problem statement on entry, parameter `curVar` identifies the current variable to be explored in the search, and parameter `fvt` contains the FVT on entry. Line 2 checks if we are examining the last variable, in which case no further forward checking is needed; instead, `state` and `fvt` together identify one or more solutions that can be added to the set of feasible solutions. The loop body in lines 7–11 is then repeated for each feasible value `v` of `curVar` in `fvt`. This can be seen in line 9 which constructs a new state in which `curVar = v`. The call to `forwardCheck()` in line 10 prunes `fvt` to obtain a reduced `newFvt` that removes values of variables that are not feasible in conjunction with `curVar = v`. If `newFvt` is `null` it means that no feasible solution is possible for `curVar = v`. Otherwise, the recursive call to `search()` in line 11 explores feasible values for later variables.

```
1 public FVT forwardCheck(int curVar, Integer curVal, FVT fvt) {
2     FVT newFvt = new FVT(fvt.getNumVars());
3     for(int freeVar = curVar+1; freeVar < fvt.getNumVars(); freeVar++) {
4         Iterator<Integer> itr = fvt.getValues(freeVar).iterator();
5         while(itr.hasNext()) {
6             Integer v = itr.next();
7             if (problem.isConsistent(curVar, curVal, freeVar, v))
8                 newFvt.addValue(freeVar, v);
9         } // while
10        if (newFvt.getValues(freeVar).size()==0) return null;
11    } // for
12    return newFvt;
13 } // forwardCheck()
```

Listing 2: `forwardCheck()` method in `ConstraintSatisfaction.hj`

Listing 2 contains the sequential code for method `forwardCheck()` of `ConstraintSatisfaction.hj`, which was called in line 10 of Listing 1. Line 3 iterates through the remaining free variables starting with `curVar+1`, and line 5 iterates through the values `v` that can be taken by `freeVar`. The essence of `forwardCheck()` is captured by the call to `problem.isConsistent(curVar, curVal, freeVar, v)` in line 7, which checks if the assignment of `v` to `freeVar` is consistent with the assignment of `curVal` to `curVar`. If so, the assignment of `v` to `freeVar` is added to `newFvt` (otherwise, it is not added). Line 10 checks if the set of feasible values for `freeVar` is empty. If so, a `null` value is returned instead of `newFvt` since no feasible solution exists at this point.

2 The Sequential Constraint-Satisfaction Solver

We have provided you an implementation of a sequential constraint-satisfaction search algorithm in a zip file containing the following:

1. `IConstraintSystem.hj` — this interface defines the methods that a game/puzzle should implement in order for it to be solvable by our solver.
2. `ProblemState.hj` — this class represents the state of the game.
3. `ConstraintSatisfaction.hj` — this file contains a sequential implementation of the constraint-satisfaction search algorithm as described above.
4. `NQueens.hj` — client solver for the NQueens problem.
5. `Sudoku.hj` — client solver for Sudoku, with a `Reader` to read input problems from the disk.
6. `NQueensMain.hj` — main program that starts and validates the results of the NQueens solver.
7. `SudokuMain.hj` — main program that starts and validates the results of the Sudoku solver.

As usual, you can compile this code and its parallel versions by typing “`hjc Main.hj`” and run it with “`hj -places 1:8 Main`” (for an execution with 8 workers), where `Main` refers to the appropriate `Main` class that you plan to use.

NOTE #1: The file, `ConstraintSatisfaction.hj`, contains two public methods, `solve()` and `parallelSolve()`. `solve()` contains a sequential implementation of the constraint-satisfaction search algorithm. `parallelSolve()` calls `parallelSearch()` which is currently empty, and will need to be filled in by you in parts 1 and 2 below. The main programs also perform a comparison test on the results returned by `solve()` and `parallelSolve()`, which should pass after you’ve completed the assignments (but which fails in the version provided).

Many puzzles can be represented by a set of rules that, applied on the current state of the puzzle, decide what are the possible actions that can be performed, which lead to a new puzzle state (with an assignment of values to a subset of free variables), thereby making them amenable to constraint-satisfaction search. This homework focuses on NQueens and Sudoku as two examples of such puzzles.

Alternative approaches to solving the NQueens puzzle have already been studied in class (Lab 4). The default size used by `NQueensMain` for this problem is $n = 12$.

Sudoku is a popular puzzle game that requires players to fill in missing numbers from 0 to $N-1$ on a square $N \times N$ board, taking into account the following constraints:

- No square contains more than a number
- Every number appears only once on each column of the board.
- Every number appears only once on each row of the board.
- Every number appears only once in each individual region of the board. Regions are usually rectangular areas of size $\sqrt{N} \times \sqrt{N}$ size.

Although Sudoku games are usually 9×9 with 3×3 regions, as in the `9x9.txt` file, there are also variations that take larger board sizes as input, such as `16x16.txt` with 4×4 regions. If 9×9 boards use the digits 1..9 to fill the board, larger sizes use 1..9, A, B, C, etc for the same purpose. Furthermore, some variations of Sudoku allow for multiple solutions, and the solver provided indeed finds all the possible solutions. If a cost function is specified, the solver must find the “cheapest” solution. The supplied serial solver already does that for you; however, it is your responsibility to keep the solver working in all case. The default input used by `SudokuMain.hj` is `9x9-2-multisol.txt`.

3 Your Assignment: Parallel Constraint-Satisfaction Search

Your assignment is to design and implement a parallel algorithm for constraint-satisfaction search, using the provided sequential implementation as a starting point. Your homework deliverables are as follows.

1. [Computation of all solutions (30 points)]

Create a new parallel version of `ConstraintSatisfaction.hj` that is designed to achieve the *smallest execution time* using 8 cores on a *dedicated Sugar compute node* to return *all possible solutions* to the problem. (To obtain a dedicated compute node, use the `qsub` command discussed in the lab exercises since Lab 4). You will be graded on the real speedup achieved relative to the sequential version. You can focus your attention on parallelizing the `search()` method. Keep in mind that the call to `problem.addSolution()` in line 3 of Listing 1 is not thread-safe i.e., it can lead to interference and data races if it is called multiple times (with different solutions) in parallel.

Your solution should work for any constraint-satisfaction problem, but you should test it with `NQueensMain.hj`, both for correctness and for achieving the best performance that you can relative to the sequential version. The `addSolution()` method in `NQueensCS.hj` accumulates all solutions into a single array. (It does not attempt to find the best solution according to some cost function.)

Please place all files related to this solution in a sub directory named “hw_4/part1”.

NOTE #2: You are allowed to add or modify methods in any class, so long as your implementation still works for any constraint-satisfaction problem, and the `parallelSearch()` method returns the same results as `search()`.

2. [Computation of lowest-cost solution (40 points)]

Create a new parallel version of `ConstraintSatisfaction.hj` that is designed to achieve the *smallest execution time* using 8 cores on a *dedicated Sugar compute node* to return a *single solution with lowest cost* (so long as at least one feasible solution exists). While one approach is to simply reuse the solution from Part 1 above and return the solution with lowest cost, you can be smarter and reduce the work done by pruning the exploration of partial solutions that are guaranteed to never lead to a solution lower than the current best solution. You will be graded on the real speedup achieved by both your sequential and parallel versions relative to this brute-force approach. (The speedup will come from a combination of algorithmic improvements and from parallel computing.)

Your solution should work for any constraint-satisfaction problem with a cost function, but you should test it with `SudokuMain.hj`, both for correctness and for achieving the best performance that you can. The `addSolution()` method in `SudokuCS.hj` stores the best solution found (if any) using a (artificial) cost function that corresponds to the single `BigInteger` value obtained by scanning the digits from left-to-right and top-down in the solution. `accumulates` all solutions into a single array.

NOTE #3: As written, `getCost()` only provides a meaningful cost if the solution is a final Sudoku solution. You are welcome to extend to provide useful bounds for partial solutions, if you so choose.

Please place all files related to this solution in a subdirectory named “hw_4/part2”.

NOTE #4: You are allowed to add or modify method definitions in any class, so long as your implementation still works for any constraint-satisfaction problem with a cost function, and the `parallelSearch()` method returns the same result as `search()`.

3. [Homework report (30 points)]

You should submit a brief report summarizing the design of your parallel algorithms in Parts 1 and 2 above, explaining why you believe that each implementation is correct and data-race-free.

Your report should also include the following measurements for both parts 1 and 2:

- (a) Performance of the sequential version with the default input
- (b) Performance of the parallel version with the default input, executed with the “`-places 1:1`”, “`-places 1:2`”, “`-places 1:4`” and “`-places 1:8`” options on a Sugar compute node to run with 1, 2, 4 and 8 workers.

Please place the report file(s) in the top-level `hw_4` directory.