Lattice Theory for Parallel Programming Project Report

Elia Vaglietti **023279125A**

December 5, 2024

1 First exercise

The first exercise can be found in the folder `first_exe.cpp`.

The file `sequential` contains a simple sequential algorithm to find the minimum.

The first parallel version is in the `gen_parallel_algo.cu` file. Here, I developed a parallel version of the algorithm in which the many blocks compute the minimum per block via block reduction, and then the reduction between blocks results are done sequentially.

The fix-point version can be found in the `fixed_point.cu` file. Here, every thread of every block checks if the number of his index is less than the minimum. By calling the kernel until the minimum is not updated is the fix-point iteration.

2 Second exercise

2.1 NQueens

The solution of the NQueen problem is in the file `nqueens.cpp`. Here, we have the version in which all the solutions are counted and shown. The extension of this problem to any constraint (check an arbitrary list of inequalities) is directly in the PCO solution as a solution for that problem.

2.2 Parallel Combinatorial Optimization

2.2.1 Serial version

I split this exercise and the NQueens because, even if they are both backtracking algorithms, I find them very different and hard to adapt the PCO to the NQueens.

To solve the PCO I have created a fixpoint iteration, where, starting from a **root** node, all the solutions within the domains that respect the constraints are found.

The serial version is found in the `fixpoint_sequential.cpp` file. The algorithm can be summarized as follows:

1. Select a node from the stack.
2. Enter the fix-point loop
3. Loop over the variables. For every variable count how many values of the domain are valid.
   (a) If only one is found, then it is a singleton. Loop over all the other variables: If the constraint is satisfied, update the domain by removing the value from the other variables. Then check if the number of singletons has increased. If it is the case simply do the fixpoint iteration again.
   (b) If no possible value in the domain of the variable has been found, then that node is not a solution and we can exit the fixpoint iteration.
   (c) If more than a variable is found we need to branch, so just go after the fix-point iteration.
4. Check if the node is a solution, i.e. it has as many singletons as number of variables.

5. Branch by selecting the first variable with domain not singleton and branch it in every possible value. These children nodes will go in the stack and will be processed as in the first point.

2.2.2 Parallel version

Since the request was to parallelize only the part of the program that updates the domain, I proceeded to parallelize the `for loop` described in point 3a. The file with the implementation is called `fixpoint_parallel.cu`.

Here, instead of the loop, a kernel is instantiated. Every variable to be analyzed is given to a thread of the kernel. Then, every thread checks if the value has to be changed (constraint matrix) and if the value is within the domain of the analyzed variable. To optimize this code, the constraint matrix, appropriately linearized to be used on GPU, is initialized and instantiated on the device only once at the beginning of the program.

Despite this little optimization, the overhead caused by the movement of the updated domain for many data in every fix-point iteration from host to device and vice versa causes a greater execution time than the serial version.

3 Performances

Here is a table for the performances sequential, parallel with the 3 and 4 size problem.

Program	Size of the problem	Time (ms)
parallel_fixpoint	4	196096
sequential_fixpoint	4	9211
parallel_fixpoint	3	1518
sequential_fixpoint	3	50

Table 1: Execution times for parallel and sequential fixpoint programs with different input sizes.