

Excercise Sheet 5

(To be processed until 13.01.)

Lecture Parallel Processing Winter Term 2025/26

Exercise 1: Understanding a solver for the Sokoban game

In the archive [u05eFiles.zip](http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/pv/u05eFiles.zip)¹ on the lecture's web page, you will find the code of a sequential (brute-force) solver for the game Sokoban ("warehouse manager"). In this game, a player has to push objects (often boxes) to predetermined target positions (storage locations). The boxes can only be pushed, but not pulled, and only one box can be pushed at a time. Background information on the game can be found on Wikipedia:

- <http://en.wikipedia.org/wiki/Sokoban> (English)
- <http://de.wikipedia.org/wiki/Sokoban> (German)

The code is structured into the following classes:

- `Playfield` (in `playfield.h` and `playfield.cpp`):
This class represents a given level of Sokoban, i.e., the playing field (board) with the initial positions of the boxes and the player as well as the positions of the targets.
Internally, the playing field is not represented as a two-dimensional array, but in the form of numbered fields. In addition, the left, upper, right, and lower neighbor fields are stored in an array for each field (if any, i.e., only if the field is not a wall).
This class also contains the code for printing the playing field, where the output is colored for the sake of clarity. If this makes problems, the statement `#define COLOR` can be commented out in `playfield.cpp`.
- `Config` (in `config.h` and `config.cpp`):
This class represents a configuration in the Sokoban game. This includes the playing field (shape, number of fields, positions of the targets, ...), the positions of the boxes and the position of the player. Each configuration may also be represented by a configuration number (i.e., an integer). This is determined by a configuration number for the box positions and the position of the player. By encoding the configuration as a "small" integer, a bit set can be used during the search to store the set of all configurations already visited.
- `Converter` (in `converter.h` and `converter.cpp`):
This class (with only static attributes and methods) converts a configuration of boxes, i.e., an array containing the positions of the boxes, into an integer (the configuration number), and vice versa.
- `BFSQueue` (in `bfsqueue.h` and `bfsqueue.cpp`):
This class is used to support breadth first searches. It essentially implements a queue for configurations connected to a set (implemented as a bit set) that stores the already examined configurations.
The most important method is `lookup_and_add()`. It checks whether a given configuration is already contained in the bit set. If not, the configuration is entered into the bit set and the configuration, the index of the previous configuration, and the number of the moved box are added to the queue.
- `DFSStack` (in `dfsstack.h` and `dfsstack.cpp`):
This class implements a stack of configurations for depth first search. It keeps track of the path from the initial configuration to the currently examined configuration.
- `DFSDepthMap` (in `dfsdepthmap.h` and `dfsdepthmap.cpp`):
For depth first search, this class performs a mapping from a configuration number to the lowest depth found so far for the corresponding configuration (i.e., the minimum number of moves from the initial configuration to the given configuration, which has been found until now).
The most important method is `lookup_and_set()`. It checks to see if there is an entry for the given configuration number with a depth less than or equal to the given (new) depth. If so, `false` is returned. If not, the depth of the configuration in the map is set to the new depth, and `true` is returned.

¹<http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/pv/u05eFiles.zip>

The main program in `sokoban.cpp` contains two fundamental routines:

- `doBreadthFirstSearch()` executes a breadth first search in order to find the solution,
- `recDepthFirstSearch()` executes a depth first search.

Acquaint yourself with the relevant parts of the code and try to understand its operation! The structure of these routines as well as their working principles are explained in section 3.7 of the lecture slides and an [additional screen cast](#).

Compulsory Exercise 2: Sokoban: Parallelization of Breadth First Search

Submit until Tuesday, January 13th, 10:00 via moodle

- a) Parallelize the function `doBreadthFirstSearch()` of the Sokoban solver (see Exercise 1) using OpenMP. Note that all configurations of the depth `depth-1` can be examined in parallel. Remember that you need mutual exclusion when executing the `lookup_and_add()` operation of the `BFSQueue` object. In this exercise, you can just execute the *calls* to `lookup_and_add()` under mutual exclusion, so there is no need to modify the class `BFSQueue`.

Note that when parallelizing the code in the suggested way, the order in which the successor configurations are entered into the queue changes, so strictly speaking, data dependencies are violated. In this exercise, this is allowed for you as an exception. However, it is important that the number of examined configurations for each tree depth exactly matches that of the sequential program. For better control, this output is sent to *standard error*, so that it can be separated into a file with `./sokoban level.txt 2> output.txt`. The enclosed makefile allows you to check the correctness of this output by calling `make test`.

In the directory `LEVELS` you will find some Sokoban levels as well as the associated outputs for testing. In the `makefile`, you can select the level file to use. Note the complexity of each level, as specified in the file `LEVELS/README.txt`.

Document to be submitted: modified C++ file `'sokoban.cpp'`.

- b) You will notice that the program doesn't show any speedup. Conduct a performance analysis using Scalasca to find out the reason for it, and explain the reason in sufficient detail.

Document to be submitted: PDF file with a screen dump of Scalasca showing the performance problem, and the explanation.

Compulsory Exercise 3: Sokoban: Parallelization of Depth First Search

Submit until Tuesday, January 13th, 10:00 via moodle

Parallelize the `recDepthFirstSearch()` function of the Sokoban solver (see Exercise 1) using OpenMP tasks. To do so, do not directly execute the recursive calls at a certain depth in the tree, but instead create OpenMP tasks. Think carefully which arguments of the call must be copied (`firstprivate`) and which must be shared. Be sure to implement mutual exclusion when executing the `lookup_and_add()` operation of the `DFSDepthMap` object. In this exercise, you can just execute the *calls* to `lookup_and_add()` under mutual exclusion, so there is no need to modify the class `DFSDepthMap`.

Test your program by calling `make DEPTH=depth test`, where `depth` is the maximum depth at which the tree should be examined. It is best to use the values specified in `LEVELS/README.txt`.

Note: If necessary, a copy of the stack (argument `DFSStack *stack`) can be created using the copy constructor:

```
DFSStack *newStack = new DFSStack(*stack);
```

Do not forget to deallocate the copy again, using `delete`.

Measure the achieved speedup (to think about: why is the speedup not meaningful here?), and try to optimize your code.

Document to be submitted: modified C++ file `'sokoban.cpp'`.

Compulsory Exercise 4: Optimizing Mutual Exclusion in the Sokoban Data Structures

Submit until Tuesday, January 13th, 10:00 via moodle

You will have noticed that the performance of the parallel Sokoban solver is not optimal, as the complete execution of the `lookup_and_add()` methods is under mutual exclusion. In this exercise, you will improve the performance by implementing a more fine grained locking: instead of executing the whole method in a critical section, only the critical parts of the method should be executed under mutual exclusion, i.e., those parts where shared data is read and written.

- a) If you examine the code of the two `lookup_and_add()` methods (in `bfsqueue.cpp` and `dfsdepthmap.cpp`), you will several times see the code pattern

```
if (condition) // is it necessary to perform the action?
    action;    // executing the action results in the condition becoming false
```

In order to ensure that the action is executed exactly once, you would naively execute the complete if-statement in a critical section:

```
#pragma omp critical
if (condition)
    action;
```

However, this is not efficient, since just entering the critical section is a noticeable runtime overhead. If the condition is frequently false, the performance can be improved by first testing the condition (in an unsafe way) before entering the critical section:²

```
if (condition) {
    #pragma omp critical
    if (condition)
        action;
}
```

Use this strategy to implement a fine grained locking in two `lookup_and_add()` methods in `bfsqueue.cpp` and `dfsdepthmap.cpp`. There is also an additional explanation at the end of the [screen cast](#) explaining the Sokoban solver.

Documents to be submitted: modified C++ files `'bfsqueue.cpp'` and `'dfsdepthmap.cpp'`.

- b) Compare the performance of your original implementation from Exercise 2 and Exercise 3 with the improved implementation.

Document to be submitted: PDF file with a performance comparison.

²This is similar to the idea behind 'test and test-and-set', see e.g. https://en.wikipedia.org/wiki/Test_and_test-and-set.