UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Exercise Sheet 3

(Deadline: 16.01.2024)

Parallel Processing
Winter Term 2023/24

Preparation: Download the program codes for this exercise sheet from the Internet:
`https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u3Files.zip`

Exercise 1 (Compulsory Exercise for 5 CP and 6 CP): Parallelization of a solver for the Sokoban game

In the downloaded ZIP file, 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` und `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` und `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` und `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.

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.

The structure of these routines as well as their working principles are explained in section 5.3 of the lecture slides.

a) Parallelization of breadth first search: Parallelize the function `doBreadthFirstSearch()`. Note that all configurations of the depth `depth-1` can be examined in parallel. Remember to realize mutual exclusion where necessary.

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 an 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`.

Measure the speedup achieved, perform a performance analysis (you may use Scalasca, if possible) and try to optimize your code.

b) Lock-free implementation of `BFSQueue::lookup_and_add()`: Try to create a lock-free implementation of the method `BFSQueue::lookup_and_add()`. Because OpenMP only very recently supports the corresponding atomic operations (and thus, your compiler may not yet include this support), use the built-in gcc functions:

- `bool __sync_bool_compare_and_swap(type *ptr, type oldval, type newval)`
- `type __sync_fetch_and_or(type *ptr, type value)`
- `type __sync_fetch_and_add (type *ptr, type value)`

Here, `type` can be an arbitrary integer type or a pointer type.
See `http://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html` for more details.

The gcc compiler also supports newer builtin fuctions that support the C++-11 memory model. These functions are more flexible, but also a bit more complex to use. You can find a documentation of these functions at `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`. If you like, you can use the functions `__atomic_compare_exchange_n()`, `__atomic_fetch_or()`, and `__atomic_fetch_and()` instead of the functions shown above.

c) Parallelization of depth first search: Parallelize the `recDepthFirstSearch()` function 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`. Also consider where mutual exclusion is required.

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 neccessary, 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?), perform a performance analysis (you may use Scalasca, if possible) and try to optimize your code.

d) Lock-free implementation of `DFSDepthMap::lookup_and_set()`:

Try to implement the method `DFSDepthMap::lookup_and_set()` lock-free.