School of Computing: assessment brief

Module title	Parallel Computation
Module code	XJCO3221
Assignment title	Coursework 1
Assignment type and description	OpenMP Programming Assignment
Rationale	To implement thread-safe routines on a shared memory system, and to identify and handle data dependencies in loop parallelism.
Guidance	See overpage
Weighting	15%
Submission deadline	10am (China), Monday 27$^{\text{th}}$ March
Submission method	Gradescope
Feedback provision	Marks and comments returned *via* Gradescope
Learning outcomes assessed	Apply parallel design paradigms to serial algorithms. Evaluate and select appropriate parallel solutions for real world problems.
Module lead	Peter Jimack

1. **Assignment guidance**

   A student is trying to implement a **stack** in C, that is, a LIFO ('last in first out') data container. To help develop their implementation, the student has started on a simple stack that stores items of type `int`, held in a global array `int *stack` of size `maxStackSize`, with the current size of the stack given by the global variable `stackSize`. Memory for this array is allocated in the routine `allocateStack`, and freed by `finaliseStack`. To help with debugging, there is a routine `displayStack` that prints the current stack.

   In the current version of the code, square numbers n^2 starting from $n = 1$ are added to the stack in serial, using the routine `pushToStack`. The student wishes to implement 3 further routines: one to 'pop' values from the top of the stack; one to invert the stack; and one to rotate the stack down to a given depth (see below). To help with testing, the code currently expects 5 command line arguments:

   ```
   ./cwk1 20 10 5 0 0
   ```

   where in this example:

   1. The first command line argument (`20` in the example above) is the maximum stack size, `maxStackSize` in the code.
   2. The second argument (`10` above) is the number of values to push to the stack initially, `initStackSize` in the code.
   3. The third argument (`5` above) is the number of values to be popped from the initial stack, `numToPop` in the code.
   4. If the fourth argument is `1`, the stack should be inverted (see below). `0`, as in the example above, denotes no inversion. The corresponding variable in the code is `invertYesNo`.
   5. The fifth argument is the depth of rotation (see below); code variable `rotateDepth`. If this is zero, as in the example, there is no rotation.

   After allocating memory for the stack based on `maxStackSize`, operations 2–5 should be performed in the order given above. To help with debugging, the current state of the stack is printed out after the initial addition of values (point 2 in the list above), and again at the very end.

   Inverting the stack means reversing the order of the entire stack, so that the item at the top of the stack changes places with the item at the bottom, the item second from the top changes places with the item second from the bottom, *etc.* **Rotating the stack** is slightly more subtle. If the specified rotation depth is greater than zero, then the stack element placed `rotateDepth` below the top of the stack should be moved to the top, with other items being moved down by one. Stack entries more than `rotateDepth` from the top of the stack are not affected. A serial implementation of this is provided.

2. **Assessment tasks**

   The current version of the code is provided on Minerva as two files, `cwk1.c` and `cwk1_extra.h`, plus a makefile. Download the code, compile using the provided make-

file, and execute as per the example given above. Inspect the code until you are happy you understand how it works.

Your first two tasks are to parallelise the loops that push and pop items to and from the stack, and then make the associated routines **thread-safe**.

1. In the `main` function in `cwk1.c`, there is a loop that calls `pushToStack` a total of `initStackSize` times. Make this loop parallel, and modify the provided `pushToStack` method so that it is thread-safe. It does not matter if the order in which items are added to the stack in parallel is different to the serial code, but each item should appear exactly. You should **not** assume `initStackSize` $\leq$ `maxStackSize`.
2. The following loop calls `popFromStack` a total of `numToPop` times. Parallelise this loop, and implement a thread-safe pop method in `popFromStack`. You **may** assume that `numToPop` $\leq$ `initStackSize`. Note that you do not need to return the value that was at the top of the stack, just remove it.

The routines to invert and rotate the stack require potentially large loops to be implemented. Therefore, your final two tasks are to improve performance by implementing parallel loops for both of these tasks.

3. Implement a parallel loop in `invertStack` that inverts the stack as described above.
4. Implement a parallel `rotateStack` that rotates the stack as described above. A serial version is provided in `cwk1.c` to get you started.

Note that, unlike the first two tasks, the routines `invertStack` and `rotateStack` will **not** be called from a parallel context, and so do not need to be thread-safe.

For all tasks, your parallel implementation should be as efficient as possible using the material up to and including Lecture 6. You do not need to use any material (*i.e.* locks) from Lecture 7 to receive full marks.

The current implementation is provided on Minerva as three files:

`cwk1.c`	:	The current serial implementation.
`cwk1_extras.h`	:	Contains the data structures, and routines to initialise, print and destroy the set. **Do not modify this file**; it will be replaced with a different version for assessment.
`makefile`	:	A simple makefile (usage optional).

You are required to use unaltered versions of the data structures and routines in `cwk1_extras.h`, but are free to add new routines to `cwk1.c`. It is expected that you will only edit this one file.

3. General guidance and study support

If you have any queries about this coursework please raise these during your timetabled lab sessions in the first instance.

4. **Assessment criteria and marking process**

   Your code will be checked using an autograder on Gradescope to test for functionality. Staff will then inspect your code the allocate the marks as per the mark scheme (see below).

5. **Submission requirements**

   Submission is *via* Gradscope.

   You should first test your submission on the **Coursework 1: Check Submission** portal, which will check your submission compiles on the test machine and report back immediately. Only once you pass all of these checks should you submit your solution to **Coursework 1: Final Submission** for assessment.

   Since you should only have edited `cwk1.c`, submit only this file.

   Do not modify `cwk1_extras.h`, **or copy any of the content to another file and then modify**. In particular, you **must** use the data structures as provided, and the routine `displayStack()` **must** be called at the same points in the program as the initial code.

6. **Academic misconduct and plagiarism**

   Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

   By submitting this assignment you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

   Code similarity tools will be used to check for collusion, and online source code sites will be checked.

7. **Assessment/marking criteria**

   There are 15 marks in total.

   5 marks : Correct functionality of all operations.
   4 marks : thread-safe implementation of pop and push routines.
   6 marks : Parallel implementation of `invertStack` and `rotateStack`.