Homework 3: Parallel and Distributed Computing

DUE: See exact time on blackboard

ICSI 520

In this homework you will implement a thread safe Stack (LIFO) with the following characteristics:

- Use a data structure of your choice to represent a stack of **integers**. (Linked list or dynamically allocated array). This document will consider you are using a linked list.
- One can push or pop an integer from the stack. Thus you would need two functions to accomplish these:
    - void Push(int value) : This function adds an integer to the Stack (linked list or array)
    - int Pop(): This function removes the last element from the Stack and returns it through the function
    - The push and pop can happen simultaneously
- Another function that you need to implement is int GetStackCount(). This function returns the number of elements currently in the stack. See more below.

Implementation

Given

You have been given the following files

- GetNumbers.c
- Numbers.txt
- stackHw.h (header file)

Numbers.txt

This is the input file **required in the same directory.** This file is used to specify numbers to push into the stack. The structure is as follows: First line indicates the size (# of values) to push. The second line specifies the numbers. The code to read this file is already given in GetNumbers.c

GetNumbers.c

This provides functionality to read a file specified in the above format and creates a numbers array (dynamically allocated) based on the size. This is pretty straightforward. **Do not change anything in this file. You will use the values in this array to insert into your stack.**

You will do a serial and a parallel implementation of the code.

Functionality

Once you are able to create a stack data structure, you will implement functionality to push all the numbers that you read (using the code given) into the stack and also pop all the numbers out.

Serial implementation

Push all the numbers *n* times. Example – You read the numbers array, have a for loop that pushes everything to the stack *n* times

Pop all the numbers. Example – Use a for/while loop to pop all the numbers. You need to keep a count see hint below.

Parallel implementation

Push all the numbers *n times* but by dividing the pushes among threads.

Pop everything from the stack using threads – can divide the pops. The push and pop actions should not wait for each other. **A thread should be able to push while other is popping/pushing. This is an important functionality to handle.**

HINT: There will always be *n x m* pushes and *n x m* pops. Where *n* is the number of times you loop and *m* is the size of the numbers array that you read from the input. For parallel implementation, you do not need to wait for all *n x m* pushes to complete to start doing the pops (be time efficient).

Several Considerations

You will gain or lose points based on your decisions/considerations

1. Data structure – Which data structure will give you best performance
2. Synchronization – Where and how many times you use mutexes
3. Code organization – Organize your implementation properly as noted in above files
4. I can change the input file and have more numbers in there. Your code should still work

Results

Table 1: Varying threads, fixed pushes

NUM_THREADS (*p*)	ITERATIONS (*n*)	Execution time serial	Execution time parallel	Speedup
5	100			
10	100			
20	100			
40	100			
50	100			
100	100			

Table 2: Fixed threads varying pushes

NUM_THREADS (*p*)	ITERATIONS (*n*)	Execution time serial	Execution time parallel	Speedup
25	10			
25	25			
25	50			
25	100			
25	150			
25	200			

Table 3: Varying number of threads and varying pushes

NUM_THREADS (*p*)	ITERATIONS (*n*)	Execution time serial	Execution time parallel	Speedup
10	20			
20	40			
30	60			
40	100			
50	150			
100	200			

Grading

Serial Implementation	5%
Parallel Implementation	30%
Correct execution Parallel	35%
Correct serial execution	5%
Timing competition (as explained in previous assignments)	10%
Speedup of greater than 1	10%
Code clarity	5%

Code clarity includes

- Use of variable names
- Function naming
- File names
- Formatting
- Comments where needed

Late submissions are subject to at least 20% grade deduction.

Reminder: If one or any of your binaries don't execute on student.rit.albany.edu, you get 0 points for this and any other homework where that happens.

Submission:

Your submission should include the following:

1. **Makefile** that builds all required binaries:
    a. Part_N_type.out where N is the homework part and type can be serial/parallel
2. Serial code source file(s)
3. Parallel code source file(s)
4. Binary for both serial and parallel implementations

5. Document (word or excel) detailing the tabular results

All of the above should be zipped up into one file and uploaded on blackboard

If you are doing the bonus, add a comment in your code about it. (if applicable)

Naming convention of your zipped file – If you do not follow this, you get zero points:

FirstName_LastName_Homework_<n>.zip

Where n is the homework number.