Stacks

1 Stacks

This exercise is about handling concurrent access to multiple instances of a stack data structure. In our case, these stacks are used to store scalar integer data and, thus, a stack is defined by the following simple data structure

```
typedef struct stackstruct{
  int cnt;
  int *elems;
} stack_t;
```

which contains an integer `cnt`, that defines the number of elements in the stack, and an array of integers `elems` that contains the elements in the stack. The size of the `elems` array is set to some maximum, predefined value which we assume big enough for our purpose.

We assume that we have `n` stacks and that we have a code that randomly selects one among them and pushes a element on top of it:

```
  for(;;){

    /* Get the stack number s */
    s = get_random_stack();

    if(s==-1) break;

    /* Push some value on stack s */
    stacks[s].elems[stacks[s].cnt++] = process();

  }
```

The execution is halted when the `get_random_stack` routine returns a `-1` value. The objective of this exercise is to parallelize this code using different ways to handle the concurrent access to the stacks.

2 Package content

In the `stacks` directory you will find the following files:

- `main.c`: this file contains the main program that first calls the `stack_seq` routine containing the simple code presented above and then calls three routines `stacks__par_critical`, `stacks_par_atomic` and `stack_par_locks` that have to be developed and are meant to contain three different parallel implementations of the loop above as described below. **Only this file has to be modified for this exercise**.
- `aux.c, aux.h`: these two files contain auxiliary routines and **must not be modified**.

The code can be compiled with the `make` command: just type `make` inside the `stacks` directory; this will generate a `main` program that can be run like this:

```
$ ./main n
```

where `n` is the number of stacks to be used.

3 Assignment

- At the beginning, `stacks_par_critical`, `stacks_par_atomic` and `stacks_par_locks` are a copy of `stacks_seq` routine. Modify these routine in order to parallelize the loop presented above; make sure that potential data access conflicts are avoided: in the first routine use the OpenMP `critical` construct, in the second use the OpenMP `atomic` construct and in the third use OpenMP locks to achieve this. The code will rely on the following **important** assumptions:
  - All the threads must enter the `for(;;)` loop;
  - At each iteration of the loop, the executing thread must push the element returned by the `process` routine onto the stack defined by the `get_random_stack` routine; this is the only constraint to respect for the correctness of the result;
  - It is safe to make concurrent (i.e., simultaneous) calls to the `get_random_stack` and `process` routines.

  Make sure that the result computed by the three parallel routines is consistently (that is, at every execution of the parallel code) the same as the sequential code.
- Report in the `responses.txt` file execution times for the sequential code and the three parallel routines using 1, 2 and 4 threads. Which parallle version is fastest? Can you explain these results? Report your comments and answers in the `responses.txt`.

Advice

- Different `atomic` operations are available in OpenMP: think about which type is correct for the operation you have to protect.
- OpenMP locks are data structures that have to initialized before being used like this

  ```
  /* Declare the lock */
  omp_lock_t lock;

  /* Initialize the lock */
  omp_init_lock(lock);

  /* Set the lock */
  omp_set_lock(lock);

  /* Unset the lock */
  omp_unset_lock(locks);
  ```

  In case you need multiple locks, you may use an array of them:

  ```
  /* Declare the array of locks */
  omp_lock_t *locks;

  /* Allocate the array of locks */
  locks = (omp_lock_t*)malloc(n*sizeof(omp_lock_t));

  ...
  ```