Assignment 2

Introduction to Parallel Programming

due **Thursday 5 October 2017, 18:00** (hard deadline)

Instructions

As in the first assignment, you will need to register in a group on Studentportalen. The group can be different than your previous one, but, once again, we suggest that you find another student to work together and form a group. In case of trouble, please contact Stephan (`stephan.brandauer@it.uu.se`).

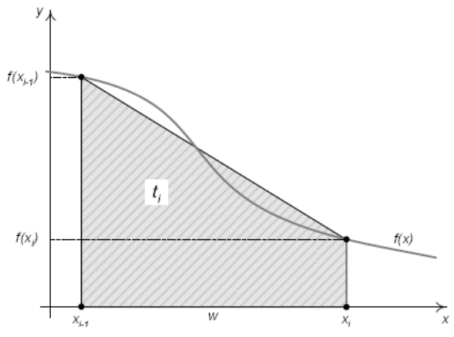
Submission checklist:

- Submissions must clearly show your name(s).
- Submit a **single** PDF report, as well as all source code related to the exercises in Studentportalen.
- Solutions must be in C11 or C++11, as specified in the exercises.
- Use appropriate synchronization mechanisms from the C/C++ standard libraries. (For the C++ one refer to `http://en.cppreference.com/w/cpp/thread`.) Do **not** rely on `semaphore.h`, `sem_t`, etc.
- All source code must compile and run on the **Linux lab machines running Ubuntu**. For the list of the Linux servers that you can use, refer to `https://www.it.uu.se/datordrift/maskinpark/linux`.
- **Provide instructions for compilation and running, preferably by including Makefile(s).**
- No source code modifications must be required for reproducing your results.
- Your report must describe the theoretical concepts used as well as all relevant details of your solution.

In case you do not reach a working solution, describe the main challenges and proposals to address them. Please keep your answers short and concise, but clear and complete.
Also, note that some of the exercises ask you to do benchmarking on the Lab machines and this requires 1) using the machine at a time when it is lightly loaded, and 2) taking multiple measurements to see if there is any variation. Thus, it's not a good idea to leave this part of the assignment for too close to the deadline!

Exercise 1: Numerical Integration (2 points)

Numerical integration is a technique for numerical (approximate) calculation of a definite integral of a function f. The algorithm is defined as follows: The interval of the integral is partitioned into many small parts of size w. For each of those small intervals the area under the curve is approximated by the area of the trapeze defined by x_{j-1}, x_j, $f(x_{j-1})$ and $f(x_j)$. Then the approximation of the whole integral can be computed by summing up all trapeze areas. The total accuracy of the calculation can be scaled arbitrarily by increasing the total number of trapezes. As the calculation of one trapeze area is independent from the other areas, this parallel numerical integration algorithm can be parallelized quite easily.



a) Implement a parallel program in C11 or C++11 for numerical integration of the following function:

$$\int_0^1 \frac{4}{1+x^2} dx$$

Your program should accept the total number of threads and trapezes with appropriate command-line arguments. Also, the command line argument `-h` should print a help message and exit.

b) Test your program and measure the runtime (of the computation, without setup time) for various configurations (number of threads, number of trapezes, each starting at 1).

Document your results in tabular form and analyze. By the way, did you notice something concerning the value of the computed integral?

c) Evaluate and document different methods of distributing the work among threads.

Hint: you may consider using different numbers of threads and trapezes.

You need to provide: 1) your code and 2) your report. Each is worth 1 point.

Exercise 2: Sieve of Eratosthenes (3 points)

For this exercise, you will use Posix threads to implement a parallel version of the Sieve of Eratosthenes,[1] an algorithm to find prime numbers. For some maximum positive integer Max, the Sieve of Eratosthenes works as follows:

1. Create a list of natural numbers: $1, 2, 3, \ldots, Max$.
2. Set k to 2, the first unmarked number in the list.
3. repeat

   Mark all multiples of k between k^2 and Max.

   Find the smallest number greater than k that is still unmarked.

   Set k to this new value.

   until k^2 is greater than Max.
4. The unmarked numbers are all prime.

To parallelize this algorithm,

1. First sequentially compute primes up to $\sqrt{Max}$.
2. Given p cores, build p chunks of roughly equal length covering the range from $\sqrt{Max} + 1$ to Max, and allocate a thread for each chunk.
3. Each thread uses the sequentially computed "seeds" to mark the numbers in its chunk.
4. The master waits for all threads to finish and collects the unmarked numbers.

Besides your code, you need to provide a brief report that explains your solution (e.g., did you use synchronization between threads and why, how did you distribute the work to threads, how did you minimize communication, how did you achieve load balance, etc.) and reports the speedup curve you get as the number of cores is increased. You should also try to vary the size of Max to gauge the impact of program size on parallel performance. Make sure to pick large numbers for Max (e.g., in the millions to ensure there's enough work for threads to do).

[1] https://www.famousscientists.org/eratosthenes/

Exercise 3: Concurrent Data Structure (7 points in total)

Concurrent data structures require proper synchronization so that users of the data structure do not have to think about it themselves. The example given in this exercise is a sorted singly linked list data structure with the following interface:

`insert` inserts a specified value into the linked list at the correct position.

`remove` removes one copy of the specified value if it is in the list.

`count` counts the number of elements with the specified value.

You can find a sequential implementation of this data structure in `sorted_list.hpp`. The driver program in `benchmark_example.cpp` prefills the sorted list with a number of elements and then runs three benchmarks:

- a read-only benchmark, using only `count`s.
- a write-only benchmark, using 50% `insert`s and 50% `remove`s.
- a mixed benchmark, using 3.125% `insert`s, 3.125% `remove`s, and 93.75% `count`s.

You can test the driver program as follows:

```
$ g++ -Wall -std=c++11 -pthread -O3 benchmark_example.cpp -o bench
$ ./bench 1
non-thread-safe read / threads: 1 - thousands of operations per second: 1592.289538
non-thread-safe update / threads: 1 - thousands of operations per second: 76.683851
non-thread-safe mixed / threads: 1 - thousands of operations per second: 182.392604

$ ./bench 2
non-thread-safe read / threads: 2 - thousands of operations per second: 3125.396324
^C
$
```

In the second run, the program stopped progressing and we had to abort it manually. As the program contains data races, it may also randomly crash, delete random files, or spawn demons from your nose.[2] Fortunately, all it does in this case is to hang.
Your task is to implement (in C++) and compare thread-safe versions of the sorted list, using the following synchronization mechanisms:

1. coarse-grained locking (for example, in C++, using the `std::mutex` class from the standard library; see http://en.cppreference.com/w/cpp/thread/mutex);
2. fine-grained locking (again, using the `std::mutex` class from the C++ standard library);
3. coarse-grained locking, using a *test-and-test-and-set (TATAS)* lock implementation;
4. fine-grained locking, using a *test-and-test-and-set (TATAS)* lock implementation;
5. a lock-free sorted list, using *Read-Copy-Update (RCU)*; see below.

Using the standard library's mutex locks should be straightforward. For your own lock implementation, we suggest to use the standard C++ library atomic functions and classes (e.g., `std::atomic<bool>`; see http://en.cppreference.com/w/cpp/atomic/atomic). Finally, for the lock-free implementation, you will have to program your own list data structure.

[2] See http://www.catb.org/jargon/html/N/nasal-demons.html.

Read-Copy-Update is an algorithm for mutual exclusion [1] that does not require explicit ownership of a lock object. Instead, it uses a single pointer that points to the current data. Updates are then applied on a copy of the data structure, and finally the pointer to the current data will be atomically switched to the updated copy using a *compare-and-swap (CAS, compare_exchange)* operation. If the CAS fails, a new copy must be created and the update must be applied again. Using this setup, all threads can safely read the data without synchronization. More information about RCU can be found on the article cited above or in Wikipedia's entry (https://en.wikipedia.org/wiki/Read-copy-update).
However, your implementation will require some kind of garbage collection to safely deal with the old data. For this, the standard library provides reference counted garbage collection through `std::shared_ptr`, see http://en.cppreference.com/w/cpp/memory/shared_ptr. These *smart pointers* function essentially like normal pointers, but automatically destruct the object when no reference to it remains. For atomic access to `std::shared_ptr`s, see http://en.cppreference.com/w/cpp/memory/shared_ptr/atomic. To compile code that uses `std::shared_ptr` with atomic accesses, you can use the following on the lab machines:

```
$ g++ -std=c++11 benchmark.cpp -o benchmark
$ ./benchmark 2
```

If you also want to test this on your own desktop, we note there is a bug in gcc's implementation of the C++ standard library, which is only fixed in version 5 (or newer), so make sure you use a new gcc version.

Evaluate, plot, and explain the results/behaviour of the five versions from the point of view of performance and scalability with respect to:

- the number of threads;
- the performed operations.

Make sure you run your experiments on a machine that allows to run at least 16 concurrent threads (e.g., one that has at least 8 physical cores, each with hyperthreading), and run your experiments using a suitable subset of worker threads in that range, making sure you include all powers of two (1, 2, ..., 2^n) in the range of numbers that your machine allows as concurrent threads.
Discuss the advantages and disadvantages of each form of synchronization in terms of: ease of implementation, lock contention, performance, and scalability.
Identify situations when one version is more suitable than another, e.g. % of read vs. % of update operations. Is the lock-free version especially good or bad in some use cases? If it is bad, reason about possible improvements or other lock-free implementations to avoid the problem(s).
Dedicate a section of your report to explain the reasoning, challenges and solutions in implementing the five versions of the data structure. Describe both the general solution and technical details.
The points of this exercise are split equally into 3.5 points for implementation and 3.5 for the report. The implementation and discussion of the lock-free mechanism (5) carries about 3.5 points in total.

References

[1] Paul E. McKenney. Structured deferral: Synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.

Good luck!