ECE454, Fall 2016
Homework 4: Pthreads and Synchronization
Assigned: Nov 5th, Due: Nov 19th, 11:59PM

The TAs for this assignment are Timur Malgazhdarov and Ali Jokar
(timur.malgazhdarov@mail.utoronto.ca or alijokar+ece454@gmail.com)

1 Introduction

OptsRus has been contracted to parallelize a customer's utility function that they use to characterize the quality of random numbers generated by the `C` library `rand_r()` function. In particular, you will parallelize it using pthreads. The code makes use of a hash table, which can be non-trivial to parallelize. Besides conventional locking, you will take this opportunity to try the newly-released software transactional-memory library.

Be sure to answer all of the numbered questions that are embedded throughout this lab handout in your "report.txt" file.

2 Setup

Start by copying the `hw4.tar` file from UG shared directory
`/cad2/ece454f/hw4/hw4.tar`
into a protected directory within your **UG** home directory.

Then run the command:
`tar xvf hw4.tar`
This will cause a number of files to be unpacked into the directory.

Looking at the file `randtrack.cc`, you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**

```
team_t team = {
    "group1",           /* Team name */
    "AAA BBB",      /* First member full name */
    "99999999",  /* First member student number */
    "",                   /* Second member full name (leave blank if none) */
    ""                    /* Second member student number (leave blank if none) */
};
```

2.1 Understanding `randtrack.cc`

The randtrack program is reasonably straightforward. After instantiating the hash table, it processes several seed streams (streams of random numbers initiated by a certain seed value), in this case four seed streams. It then goes about collecting a large number of samples (in this case 10,000,000 samples), but skips `samples_to_skip` samples, so it actually only counts one sample out of every `samples_to_skip` samples. The sample to be counted is made to fit in the range 0..999,999 by taking a modulo (`% 100000`). Next we check the hash table to see if there already exists an element for the sample, and create and insert one if not; then we increment the counter for that element/sample. Finally, once all samples have been tracked, we print out all of the sample values and their frequencies (counts).

`randtrack.cc` has the following command-line parameters:

- `num_threads`: the number of parallel threads to create (not used in the initial version we give you);
- `samples_to_skip`: the number of samples to skip

You should carefully read and understand `randtrack.cc` and how it works. Note that while this is a somewhat "toy" program for the purposes of this assignment, hash tables are a very common and important data structure and this assignment considers parallelization in their presence.

2.2 Understanding the Hash Table

The hash table implementation is a one-dimensional array of pointers to lists. Each key value each maps (hashes) to a certain array location based on the key-value modulo the hash-array size. Any collision (two keys that map to the same hash-array location) is handled by inserting a new element at the head of list for that hash-array location.

Note that the hash table and list structures are implemented using the C++ template construction because they become inlined—-this is unfortunately the only way (at least the most straight-forward way) it will work with the current version of the transactional memory library. When this template version of the hash table is instantiated, the types of the hash element and key value are specified, in this case "class sample" and "unsigned" respectively.

You should read and understand `hash.cc` and `list.cc`—other than the template syntax they are fairly straightforward.

Note that for the sake of fair comparisons, please do not modify the size of the hash table array for any experiment.

2.3 Compiling

In this assignment you will be making and measuring several different versions of `randtrack`, and you want to be able to build and run any of them at any time. It is up to you if you want to accomplish this by having different versions of the `randtrack.cc` file, or instead using `#ifdef`'s on one copy of the file. Whichever way you choose, be sure that you modify the Makefile to take care of the details. In other words, running "make randtrack_global_lock", "make randtrack_list_lock", and "make randtrack_tm", etc, should all build the appropriate version of randtrack with those names. If you are going to use `#ifdef`'s and have a single file, then note that any added methods or data structures added for a lock implementation should only be built for that lock implementation that is using them. Ie., if you add two methods and an array for the global lock implementation, those items shouldn't be built for the TM version if it is not using them—they should be `#ifdef`'ed out.

Q1. Why is it important to `#ifdef` out methods and datastructures that aren't used for different versions of `randtrack`?

More compilation details for this assignment:

- You want to compile using `g++-4.7`, a recent version with support for transactional memory. To enable that version you must first execute: `source /cad2/ece454f/gcc/sourceme.csh`
- To compile with support for pthreads you must add the flag `-lpthread` to the gcc compile command line.
- To additionally enable support for transactional memory, add the flag `-fgnu-tm` to the gcc compile command line.

2.4 Debugging

To debug, `gdb` works fine and supports pthreads (although debugging parallel code is a challenge). Adjust the `Makefile` to compile with the -g option. However, for your timing runs, you should use `-O3` optimization to get the best performance.

3 Parallelizing with Pthreads

You should start by parallizing the work using pthreads, following the example code given in class. You should create the number of threads specified by the command line argument `num_threads`, which you can assume will be set to either 1, 2, or 4. For dividing work among threads, for 1 thread the 1 thread should process all of the four seed streams, for 2 threads they should each process two seed streams, and for four threads each thread processes its own seed stream.

It is important to always verify the accuracy of the output by comparing with the output of the original program. If you save the output in a file, eg., by running "randtrack 1 50 > rt1.out", you can decide the correctness of a new version by doing "diff rt1.out rt2.out (which should print no difference). One catch is that, for parallel versions,

elements might be inserted into the hash table lists in a different order than the original (but the counts for each element should be identical). An easy way to account for this difference in the outputs is to sort the outputs before comparing them, i.e., do "sort -n rt1.out > rt1.outs", "sort -n rt2.out > rt2.outs", then compare by doing "diff rt1.outs rt2.outs" which should show no difference.

Once you have created threads and divided the work among them as above, if you execute and compare with the original you will get incorrect results, since the accesses to the shared hash table are not yet synchronized. This is what you will do using several different methods in the next sections.

The first challenge is to carefully identify which parts of the code need to be synchronized (ie., executed atomically) versus which are safe to do in parallel.

For further details on pthreads, an introductory tutorial is here:

`http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`

3.1 Single Global Lock

Create a version of the program called `randtrack_global_lock` that synchronizes by creating and using a single `pthread_mutex` around the critical shared accesses in the code. Ensure via testing that it produces the same results as the original version as described above.

3.2 Using Transactional Memory

In theory an easier alternative to the more fine-grain locking approaches below is transactional memory (TM) as discussed in class. `gcc` version 4.7 supports a TM API, and implements software TM (hardware TM is not yet available but coming soon). After following the instructions above for parallelizing with pthreads and compiling for pthreads and TM support, create a version of the program called `randtrack_tm` that uses TM to synchronize the critical section. This is done by wrapping the critical section of code with `__transaction_atomic { }` (note that there are two underscores '_' that preceed the word transaction).

After getting this working and tested, answer these questions:

Q2. How difficult was using TM compared to implementing global lock above?

3.3 List-Level Locks

While having a single global lock is easy to implement, it is often better for performance to have multiple locks that protect smaller data structures, i.e., "fine-grain locks".

Create a version of the program called `randtrack_list_lock` that creates a `pthread_mutex` for every location in the hash array (eg., one for each list), and synchronizes using those locks. You will find that this requires a bit of thinking.

Q3. Can you implement this without modifying the `hash` class, or without knowing its internal implementation?

Q4. Can you properly implement this solely by modifying the `hash` class methods `lookup` and `insert`? Explain.

Q5. Can you implement this by adding to the `hash` class a new function `lookup_and_insert_if_absent`? Explain.

Q6. Can you implement it by adding new methods to the `hash` class *lock_list* and *unlock_list*? Explain.

Implement the simplest solution above that works (or a better one if you can think of one).

Q7. How difficult was using TM compared to implementing list locking above?

3.4 Element-Level Locks

An even finer-grain locking approach than list-level could be implemented, i.e., locking individual sample elements. Think carefully about whether your implementation handles both the case where the key already exists in the hash table and also when it does not exist and needs to be inserted. Call this version `randtrack_element_lock`.

3.5 Reduction Version

Rather than sharing and synchronizing on the hash table, another solution could instead give each thread its own private copy of a hash table, each counts its streams into that local hash table, then combine the counts from the multiple hash tables into one at the end before printing. Call this version `randtrack_reduction`.

Q8. What are the pros and cons of this approach?

4 Measuring

For this homework your implementations should be measured from start to finish, using `/usr/bin/time`. In other words, you will include the entire computation in your measurement, including reading and writing files, initialization, and thread creation or other overheads associated with parallelization. Note that there is also a shell command called "time", but we prefer the one above. Report the "Elapsed" time.

4.1 Getting a Clean Timing Measurement

The code already has a timing function, measured at the appropriate points. Since we are measuring timing for parallel execution using multiple processors, it is important that we take our measurements when the machine is not heavily loaded. You can check the load on either machine with the command "`w`". You should be able to get a relatively clean measurement when the load average is less than 1.0. Since each machine has 4 processors, the load average may be as high as 4.0 when all thread slots are busy, or even higher if they are overloaded. Once you are confident in the correctness of your code, you probably want to write a script (perl or other) to perform all of your runs and collect the data.

For every timing measurement always do 5 runs and average them (please only report the final average).

4.2 Experiments to Run

Run the following experiments and report the runtime results in a table. For those that implemented element-lock or reduction versions, report results for them as well.

1. measure the original `randtrack` with samples_to_skip set to 50 (num_threads can be any number, it isn't used)
2. measure the global-lock, list-lock, (element-lock), and TM versions of randtrack with 1, 2, and 4 threads, and with samples_to_skip set to 50.

Q9. For samples_to_skip set to 50, what is the overhead for each parallelization approach? Report this as the runtime of the parallel version with one thread divided by the runtime of the single-threaded version.

Q10. How does each approach perform as the number of threads increases? If performance gets worse for a certain case, explain why that may have happened.

Q11. Repeat the data collection above with samples_to_skip set to 100 and give the table. How does this change impact the results compared with when set to 50? Why?

Q12. Which approach should OptsRus ship? Keep in mind that some customers might be using multicores with more than 4 cores, while others might have only one or two cores.

5 Evaluation

Your grade will be calculated as follows:

- *Global Lock (and basic pthreads): 20 points.*
- *TM: 15 points.*
- *List Lock: 20 points.*
- *Element Lock: 15 points.*
- *Reduction: 10 points.*
- *Answers to questions (report): 20 points.*
- *Total: 100 points.*

6 Submission

Create a report that answers all of the questions above. Be sure to submit all of the files necessary for building your solutions.

Submit your assignment by typing

```
submitece454f 4 *.cc
submitece454f 4 *.h
submitece454f 4 Makefile
submitece454f 4 report.txt
```

on one of the UG machines.