Thread Lab

Due: Friday, Dec 2nd, 2022, 11:55 PM PST

Notes

- This is an individual assignment – all code must be your own – no code sharing.
- Use “make submit” to see your actual performance. The performance on other machines won’t be accurate. Feel free to submit a few times, performance is variable.
- Please save your code often
- Lab download / scoreboard website
- Lab TA: shhh it’s a secret

Introduction

In this lab, you will implement several versions of a code which computes a histogram of a set of random numbers. Histogram is a simple kernel that demonstrates the perils of parallel programming, while also having a few interesting optimizations. Also, this is a key kernel in a variety of algorithms (e.g., radix sort).

You will be optimizing for two test cases, each in their own function:

1. a set of 100 million numbers, and a histogram of 8 buckets
2. a set of 25 million numbers, and a histogram of 16 million buckets

The cache access behavior for each of these is quite different, so optimizations may affect them differently.

Downloading and running the Lab

- **Step-1:** Go to the thread lab website on your browser. Click into the “Request Lab” button on the webpage, insert the requested information to get a tarball

Step-2: Upload the tarball from your local machine to the seas machines (using scp on the command line or any other way). In terminal you can do the following:

```
scp <yourlocalfile> <yourusername>@lnxsrv09.seas.ucla.edu:~/<yourdir>
```

-

Step-3: Log onto the lnxsrv07 server either using a terminal or your favorite text editor (VScode, etc), then go into the directory where the tarball was uploaded. Untar the tarball, and go into the corresponding directory. YOURNUMBER should be replaced with the actual number assigned to your tarball.

```
ssh <yourusername>@lnxsrv07.seas.ucla.edu
tar -xvf target-YOURNUMBER.tar.gz
```

-

Step-4: Now you can start optimizing the code contained in `histo.c`. You should only modify `histo.c`.
Once you’re done optimizing, you can check the correctness and the performance of you code by running the following commands.

```
make
./histo
```

-

You can also run a specific test input with the `-i` flag, this one runs test 2:

```
./histo -i 2
```

-

Step-5: To see how good your performance is, you need to submit your work to the grading server, to do so, simply do:

```
make submit
```

-
    - Then you should be able to see immediate feedback from the server, and this new submission will also be reflected on the scoreboard.
    - Please submit your code only when it behaves correctly to avoid overwhelming the server.

A few things to try out

Here are a few things you can try out to speed up your code while ensuring its correctness (be careful of *race conditions*). Some of these approaches will work better than others. We don't want to tell you exactly which ones are the best, so that you suffer through at least a little experimentation. Some approaches may work better for certain test cases. In no particular order:

- Use Atomic Builtins.
    - The X86 ISA (and others) have a special instruction to perform a single increment (e.g., the atomic increment function called `__sync_fetch_and_add( &integer_var_to_increment, increment_amount )`).
- Acquire a global lock when a thread needs to update the histogram.
    - Okay, this is obviously bad, because of lock contention.
- Acquire a per-bucket lock when a thread needs to update the histogram (per-bucket means less lock contention).
    - You can try using one lock for a range of buckets for the case where you have too many buckets.
- Use a private histogram for each thread, then sum together all these private histograms.
    - This reduces communication between threads, at the cost of increased cache contention.
    - Think about the cost of summing up all the local copies – which test case would that hurt more?
    - IMPORTANT: For case 2, if you keep your private histograms on the stack, it's possible you overrun the size of the stack (max 8MB).
      Instead, you can make a global array to store all the private histograms instead, like: `int private_hists[NTHREADS][T2B];`
- Have all threads read all the data, but each thread is responsible only for a specific range of buckets.
    - This technique requires no thread communication at all, but all threads have to read all the data. You could try keeping the threads synchronized to avoid re-reading data into LLC.

As you may expect, this list only captures some optimization techniques that are helpful in general, but it is by no means complete: there are other optimizations for you to discover that may work well for particular test case(s)!

Grading Policy

- Your grade is determined by the geometric mean of the speedups of the 2 test inputs. The speedups are relative to the baseline sequential algorithm running on the grading server's machine, so lnxsrv7 will not be that accurate – don't trust the grade there.
- Your score will increase linearly with your geomean speed up. The geomean speed up of 3x will give you 100% on this lab. If you find that test case 1 can achieve a higher speed up than test case 2, that's totally fine. The score only depends on the geomean.

- The grade based on the scoreboard is your final grade, and we will use your best score. Feel free to submit a few times to make sure you get a fast run of your code, but please do not overwhelm the grading server!

Extra Credit

For this lab, you will get 10% extra credit if your code achieves a geomean speed up of 4x. Have fun!!

No competition this time, the scoreboard is only for fun (and likely many solutions should approximately tie, since the design space of possible algorithms/implementations is not that high).