COSC441 - Assignment 1

James Boocock

July 20, 2014

1 Extracting and Compiling

My version of assignment one uses linux pthreads.

First extract the files and run using the following command.

```
tar xzf asgn1.tar.gz
```

Navigate into the asgn1 directory and run make, this will generate 5 exectuables q01-q05 representing each part of the assignment.

2 Questions

1. The first example uses a shared non-volatile global variable, smart compilers may attempt to register optimise the variable, this can be problematic because its value can be changed by another thread. As we increase the number of threads we expect the sum to be $threads * \sum_{i=1}^{n} i$. With small values of n the expected sum is the same or similar to the calculated sum, this occurs because the entire loop is finished before the next thread is even created. As I increased n I found that the difference between the expected sum and calculated sum increased, presumably because longer run-time leads to a subsequent increase in the number of interrupts at critical locations. Thread number also had a similar effect, presumably because of the relationship between thread number and interrupt frequency. The runtime also increases approximately linearly as thread number is increased.

2. Replacing the variable with a volatile keyword guarantees the compiler will not register optimize the global variable. Although operations on volatile variables are not atomic, so the calculations suffer from the same problems as earlier. The run-time also increased slightly because the compiler cannot perform these optimizations I mentioned.

3. Finally we have a sensible program that calculates correctly, my program used the gcc function __sync_add_and_fetch() to make the $global\_sum+=$ operation atomic, guaranteeing the operation is completed before another thread is allowed to read the global value. The program now takes significantly longer to run approximately 3 times as long, presumably because of the overhead associated with guaranteeing the atomicity of each $global\_sum+=i$ statement.

4. We replaced the GCC function call here with a pthread_mutex_lock and pthread_mutex_unlock call. Using the mutex to wrap the $global\_sum+ = i$ ensures the program gives the correct result. Compared to the GCC atomic operation the program now takes even longer, this is likely because the atomic GCC function requires only one function call, as oppossed to pthread's mutex operation which requires two, one for acquiring and one for releasing the lock.

5. In the final part of the assignment each thread calculated partial sums, and then used a pthread_mutex to add these partial sums to the global variable. Here I observed that the runtime is slightly less than in our first example and works correctly. The program is fast because locks are only acquired when absolutely necessary, once per thread, so in total the number of locks acquired and released is equal to the number of threads. I also speculate that slight improvements in speed compared to question 1 may be due to compiler optimisations on the partial sum calculations.