Kennesaw State University

Department of Computer Science

CS4491 Programming for HPC

Assignment 3 : Reinforce pthreads Concepts

Michael Hug

hmichae4@students.kennesaw.edu

September 22, 2014

Intial problem statement

4.1 : When we discussed matrix-vector multiplication we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is not the case?

4.3 : Recall that the compiler is unaware that an ordinary C program is multithreaded, and as a consequence, it may make optimizations that can interfere with busy-waiting. (Note that compiler optimizations should not affect mutexes, condition variables, or semaphores.) An alternative to completely turning off compiler optimizations is to identify some shared variables with the C keyword ***volatile****. This tells the compiler that these variables may be updated by mutiple threads and, as a consequence, it shouldnt apply optimizations to statements involving them. As an example, recall our busy-wait solution to the race condition when multiple threads attempt to add a private variable into a shared variable:*

```
/* x and flag are shared, y is private */
/* x and flag are initialized to 0 by main thread */

y = Compute(my rank);
while (flag != my rank);
x = x + y;
flag++;
```

Its impossible to tell by looking at this code that the order of the while statement and the x = x + y *statement is important; if this code were singlethreaded, the order of these two statements wouldnt affect the outcome of the code. But if the compiler determined that it could improve register usage by interchanging the order of these two statements, the resulting code would be erroneous.*

If, instead of defining

```
int flag;
int x;
```

we define

```
int volatile flag;
int volatile x;
```

then the compiler will know that both x *and* flag *can be updated by other threads, so it shouldnt try reordering the statements.*

With the gcc *compiler, the default behavior is no optimization. You can make certain of this by adding the option* -O0 *to the command line. Try running the* π *calculation program that uses busy-waiting* (pth_pi_busy.c) *without optimization. How does the result of the multithreaded calculation compare to the single-threaded calculation? Now try running it with optimization; if youre using* gcc*, replace the* -O0 *option with* -O2 *. If you found an error, how many threads did you use?*

Which variables should be made volatile in the π *calculation? Change these variables so that theyre volatile and rerun the program with and without optimization. How do the results compare to the single-threaded program?*

4.14 : Recall that in C a function that takes a two-dimensional array argument must specify the number of columns in the argument list. Thus it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the Pthreads matrix-vector multiplication so that it uses a one-dimensional array for the matrix and calls a matrix-vector multiplication function. How does this change affect the run-time?

Response to problem 4.1

In class we discussed parrelization of matrix operations. We were able to efficiently optimize matrix operations by using our knowledge of how C stores arrays. C is row major and will store rows contiguiously. We spawned threads who each were assigned their own row. The thread completed the mathematical operations on it's own row without examining any other rows. The thread then wrote to the last column(still in assigned row) the result of it's calculation. Each thread read it's data and never other thread's data. Each thread wrote to a location where no other thread would write to a location. There were no race conditions because we cleverly assigned read and write locations to each thread.

If we consider that the number of rows and columns is not evenly divisible by the number of threads, we will have to make some changes to our code. The ideal situation would be that the number of threads is exactly a multiple of the number of rows. The book has been asuming that all mathematical operations are treated equally. Treating all operations the same leaves us with the knowledge that when two threads start to do math on two rows, they will end at the same time. I think it would be best to think about this recursivly. The base case is when there are no rows left: exit the recursion. If there are rows left and more rows than thread, each thread will get it's own row. If there are rows left, but less rows than threads, we would chunk the rows and give each threads it's own portion of the remaining rows. When we do this, we have to be careful about the race condition that exists when the threads are finished. They will all finish at the same time, so we will have a small botleneck on writing to the last column of the row(the result). The threads have to take turns adding their result to the final column.

Response to problem 4.3

The man page for gcc did not contain information about the -O arguemnt. The info page had sufficient information.

Running the unoptomized unaltered code :
Both single and multithreaded estimations were identical except for the least significant digit. The threaded execution was an only marginally faster.

Running the optomized unaltered code:
This overheated my box!!! I let it run and went to grab a soda to find my machine shutting down. Busy-wait and deadlock is not a good combination

altering lines 40 and 41 from :

```
int flag;
double sum;
```

to :

```
int volatile flag;
double volatile sum;
```

Running the unoptomized altered code :
Same results as the unoptomized unaltered code - no suprise

Running the optomized altered code :
Now we get the same results as the unoptomized unaltered code.The grand total of all optomizations in the code were likely only the flag and sum variable modification orderings.

In conclusion, we accomplished the same goal several different ways. We told the compiler not to optomize our code from within the source code and also as an agrument to the compiler. We also found that busy-waits should have a threshold or something to prevent your system from crashing. And also, I should use the advice that I give grandma, and not run code I find on the internet. I like the idea of the volatile, that the profesional programer can cherry pick the variables that should not be optomized, rather than having either a yes or no on compiler optomization.

Response to problem 4.14

```
see code in .c file
```

The unexpected part was that I was able to get done quicker, even with the overhead of translating the 2d matrix to 1d!