

Lab 4 — High Performance Programming with Multicore and GPUs

pthread Tutorial

Task1

Codes modified in test_generic.c.

Task2

All the output are different.

Task3

Codes modified in test_create.c.

Task4

When sleep function is added to work function, there is nothing printed in the work function since it causes the thread of sleeping for 3 seconds, which makes the main function to finish before the threads are completed.

Task5

After the sleep function is moved to the main function, there will be a 3 second delay for printing the statement.

Task6

After adding the sleep(), the output changes as the “After creating the thread..” statement gets printed first, and then the work function print statements get printed out. This is different from task 4 where the print statements in the work function are skipped entirely.

The statements get printed out in this case because the pthread_join() function ensures each thread completes, before proceeding to the next one. And the sleep() makes the work() function to stop for a while then the print in the main() first goes out.

Task7

When changing the data type into signed char, when compile, the codes report an error; and it does not print the thread ID, but rather just a the value of the ID multiply by -1.

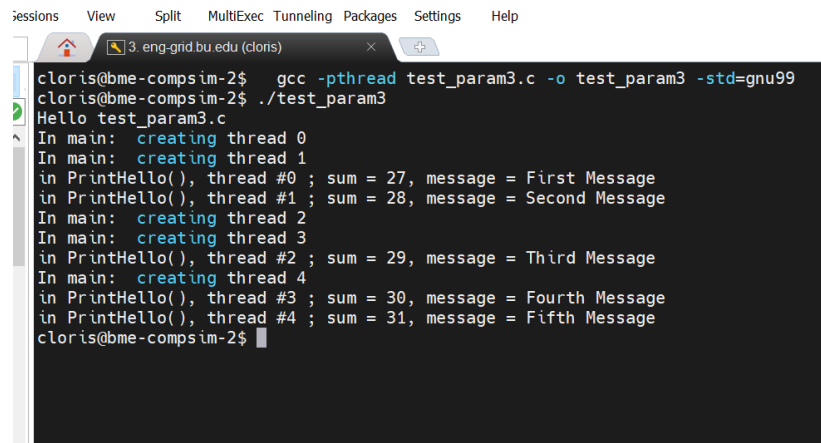
Task8

Before the changing of f and g, f changes frequently, and g stays constant. After I changing the value of f and *g, I found that both change the output while the only difference of changing f and *g is that the last row indicating the t value.

When f changes, the t value does not change; however, when *g is changed, the t value also changes since the location where the variable is stored changed.

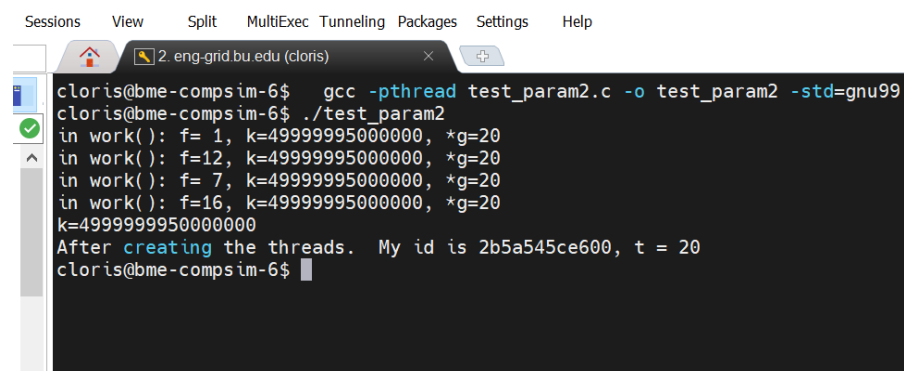
Task9

What the problem asks is to make fewer threads to be created. And as the main() function passes the value of t into the work() function, we can try modify the value of the g in the work() function. As in the main() function it defines that when the t is increased into 10, the threads will stop creating; so we can enlarge the value of *g in the work() function, so that it takes less time to achieve 10. In practice, I tried to modify the *g into 15, and it turns out to be:



```
cloris@bme-compsim-2$ gcc -pthread test_param3.c -o test_param3 -std=gnu99
cloris@bme-compsim-2$ ./test_param3
Hello test_param3.c
In main: creating thread 0
In main: creating thread 1
in PrintHello(), thread #0 ; sum = 27, message = First Message
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main: creating thread 2
In main: creating thread 3
in PrintHello(), thread #2 ; sum = 29, message = Third Message
In main: creating thread 4
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
cloris@bme-compsim-2$
```

I also tried *g += 4, and it also works.



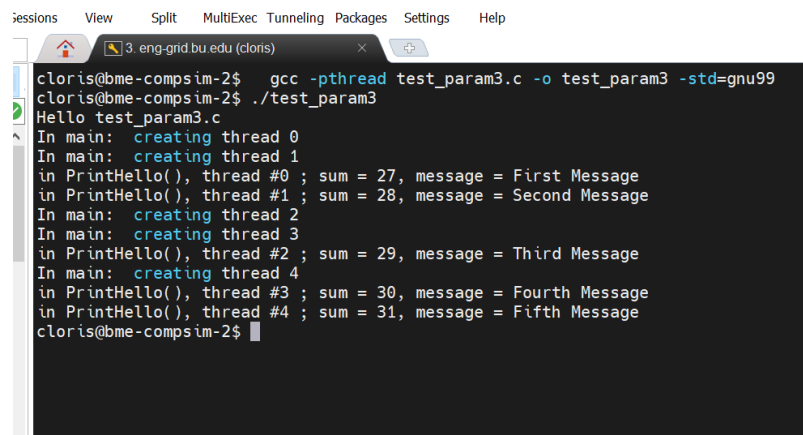
```
cloris@bme-compsim-6$ gcc -pthread test_param2.c -o test_param2 -std=gnu99
cloris@bme-compsim-6$ ./test_param2
in work(): f= 1, k=49999995000000, *g=20
in work(): f=12, k=49999995000000, *g=20
in work(): f= 7, k=49999995000000, *g=20
in work(): f=16, k=49999995000000, *g=20
k=4999999950000000
After creating the threads. My id is 2b5a545ce600, t = 20
cloris@bme-compsim-6$
```

Task10

From my perspective, the second way seems better. Added a global int variable- global which was set to 0 and incremented by 1 for each thread, so each thread got a unique value from the array.

Task11

The output of the test_param3.c is:



```
cloris@bme-compsim-2$ gcc -pthread test_param3.c -o test_param3 -std=gnu99
cloris@bme-compsim-2$ ./test_param3
Hello test_param3.c
In main: creating thread 0
In main: creating thread 1
in PrintHello(), thread #0 ; sum = 27, message = First Message
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main: creating thread 2
In main: creating thread 3
in PrintHello(), thread #2 ; sum = 29, message = Third Message
In main: creating thread 4
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
cloris@bme-compsim-2$
```

Task12

The thread completes before mutex is unlocked.

Task13

It no longer produces an output – syncing multiple threads with one mutex lock causes issues.

Task14

After adding the sleep function, the prints before barriers still occur before the prints after barriers. Hence the barrier still works.

However, the adding of the sleep function makes is that all the threads get created before any printing occurs. I tried to change the sleep function with variables, and it turns out still working since it prints after barrier statements.

Task15

The reason why the loop being executed after threads is that in the main() function, every thread is locked until when the user type in a letter and press enter, it will be unlocked. So, in the printHello() function, during the switch case, first there is something to be printed out, and then every thread is locked and unlocked.

And, only if one thread is unlocked can it be locked again, so that the thread is just stuck in the switch, until it is unlocked in the main() function, it can be locked and unlocked again so that the “I’m unlocking” can be printed out.

Task16

Shown in modified codes.

Task17

I tried several times, for most of the cases, the result turns out to be 1000, but in very limited cases it prints out 999 or 1001. The balance is almost always correct, but when I add the sleep function to it, the balance ends up with 999.

When I increase the number of threads to 10000, the changes every time without the sleep function and the printed "qr_total" varies from time to time.

Task18

Shown in modified codes.

Task19

When using 2500000 iterations, it takes about 2.32 second.

Task20

When changing the USE_BARRIERS to 1, we can see that it takes less iterations and less time to achieve the goal. In my estimate, it takes around 15000 iterations and about 0.67 second. So we can conclude that when the number of barriers is set to 1, it is slower but takes less iterations.

The reason that there are less iterations when using the barriers is that the barriers ensure that all the threads complete for each iteration, i.e. each element in the array is updated in each iteration before moving on to the next iteration. So that the elements in each iteration are consistently the same. And when not using barriers, the neighboring elements are used to update the current element, and may not be aligned. So that the time also decreases due to less of iteration.