Uppsala University

High Performance Programing

1TD062 62013 VT2023

Seminar 2 -report

Author:
Csongor Horváth

February 7, 2023

During Lab 8 and 9 I used compiler without any optimization if it is not stated in the description specifically or it isn't the case in the originally given makefile. This way I can get better view of the the advantage coming purely from the usage of threading.

Lab 8

Task 1

Let's start with the explanation what I think is happening. When calling pthread the pthread process starts in a new thread or core. Starting a new process makes some time. So it is possible that the main function reach the join statement earlier than the thread is finished. Than we wait till the thread finishes the process.

To illustrate the running structure first we show code and output with sleep statements. To make it easier we present a program which starts n process, where n is a given integer and in each process do sleep(s) and printing out the thread number in a loop length of 4. The main process doing the same, but writing out main as output.

The c codes can be found in the end of this pdf as appendix. The commandline outputs will be shown with the title of the corresponding codes.

```
Output from Task1-1.c:
 This is the main() function starting.
 the main() function now calling pthread_create().
 This is the main() function after pthread_create()
 main
 main
 Thread: 1
 Thread: 2
 main
 Thread: 1
 Thread: 2
 main
 Thread: 1
 Thread: 2
 the main() function now calling pthread_join().
 Thread: 1
 Thread: 2
```

As we can see the main process already done one extra iteration until the thread processes started. Also the main process finished and calling join earlier than the two thread stops running.

We can also notice that the threads are running paralel, this is shown by the fact that the print statements are happening in mixed version and note the same outpute next to each other.

If we change the number of iteration inside the threads to a smaller number we can see that they finishes earlier. e.g: with for cycle of length of 2:

```
Output:
 This is the main() function starting.
 the main() function now calling pthread_create().
 This is the main() function after pthread_create()
 main
 main
 Thread: 2
 Thread: 1
 main
 Thread: 2
 Thread: 1
 main
 the main() function now calling pthread_join().
```

As we see, now the threads are finished earlier than the main process.

Before using the top, we modify the code, because due to the sleep process our current code only good for illustrating the parallel running, but it is not make the CPU work hard due to sleeping all the time.

So let's run the code shown in *Task1-2.c* where the processes are the one given in the description. And see the output of the top command now. We get greater than 200% CPU usage.

This concludes our work with task 1, because we already used 3 threads and show that they run parallel.

Task 2

Note: I already presented these techniques in Task 1 with giving integers as n argument to be able to out put the number of the thread where the print statements coming from

Now we present the results after casting the void pointer into double and printing out the arguments. This way we can aces data inside a thread which comes from the main(). I already represent the results after adding the second thread. I gave only negative values to the second thread to be able to see which print comes from which thread.

```
Out Task2.c:
 This is the main() function starting.
 the main() function now calling pthread_create().
 Data from thread: 5.700000
 Data from thread: 9.200000
 Data from thread: 1.600000
 Data from thread: -1.000000
 Data from thread: -3.141500
 Data from thread: -0.600000
 This is the main() function after pthread_create()
 the main() function now calling pthread_join().
```

From this we can see that the threads are started after each other and it is takes time each time to start a new thread. Therefore, we can see now that thread 1 is finished sooner, than thread 2 makes the first print.

Task 3

In this task I added the timing to the code as can be seen in *Task3.c* in the appendix. And after changes N1 and N2 values so that the sum stays 800 000 000. And I represent the measured run times in the table below. I run the code with no optimization flag, therefor we can see the pure run times of the code.

N1 (10^6)	N2 (10^6)	overall runtime	thread time	main time
800	0	1.698784	0	1.698562
700	100	1.532832	0.258706	1.532622
600	200	1.272026	0.500200	1.271758
500	300	1.147072	0.731680	1.146820
400	400	0.962239	0.952223	0.962040
300	500	1.204806	1.204492	0.766922
200	600	1.321373	1.321061	0.487110
100	700	1.491471	1.491067	0.250247
0	800	1.735986	1.735727	0

As we see from the run times, we get the best performance when the threads do the same amount of work. The main thread and the extra thread do the works in the same time. We can also notice that the overall runtime of the program is only a bit longer, than the maximum runtime of the two threads. This is because the extra runtime comes from the staring and ending of the thread and some other extra process such as print statements.

The optimal runtime when we split the task equally between the two thread results almost two time faster run time as we can expect it for computer with at least two core.

So this concludes Task 3 with the conclusion that splitting a suitable task to n threads can results up to n time faster runtime if our computer has n core and the task takes significant time compared to the necessary system procedures.

Task 4

First let's measure the run times of the naive serialized implementation which can be found in the appendix under *Task4-1.c.* For time measurement in this task I used the time command.

M	time(s)
10000	0.047 s
100000	0.916 s
200000	3.459 s

Now we will make our code run on two threads as asked in the description. For this purpose I will divide the for cycle into two parts at 70% (I suppose $0.7 \cdot M$ is integer furthermore M dividable by 100), because the later outer iteration have longer inside loop. It would be possible to use mathematical tools to calculate where we should split the iteration into two to get same number of inside iterations, but now I won't do this.

The modified code can be found in appendix under *Task4-2.c.* We present run time measurement below.

M	time(s)
10000	0.38 s
100000	0.584 s
200000	2.168 s

As we can see even without searching the optimal split point we can reach a significant improvement in the time of the program with using one more thread. The improvement results almost 2 time faster run times, or at least better than 1.5 faster run time.

Task 5

For this task I can use the same code as my original code for task 1. I can even delete the for cycles and sleeps and one print statement is enough in the function. The exact code for this task is under the name *Task5.c* in the appendix. I present an output with running 10 thread. Note: since computer has finite number of cores and resources it is not efficient to use too much thread. It can make the run time longer if we use each thread for too small portion of a task, hence the time of system operation for starting new thread will become significant in this case.

```
This is the main() function starting.
the main() function now calling pthread_create().
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Thread: 5
Thread: 6
Thread: 7
Thread: 8
Thread: 9
This is the main() function after pthread_create()
main
the main() function now calling pthread_join().
Thread: 10
```

As we see the order of in which the threads start are the same as we call them. This aspect is different from e.g: python's process pool system, because there we can't be sure in which order the task will be executed.

Task 6

I don't expect to reach such improvement in the run time, because it would be a hard problem to determinate how to split the outer loop, such that the work done by the different threads would be the same. Therefore as a naive approach let's split the outer loop into same size partitions. Note: this way the last thread will be running the longest, so that is the run time what will be matter. And it can be close to the last version if there the longer runtime came from the last 30%, but if the first 70% gave the most of runtime, than we can still see significant improvement.

During the implementation I supposed that M/N is an even integer. Otherwise it would be a bit harder to determinate the split points. And it would require some if statement, and maybe an other implementation of the creation of the split points matrix. So for simplicity we supposed the above in this implementation.

The code can be found in appendix under name *Task-6.c*. Now we represent some run time measurements with $N = 4$. Here we also use the time function and we show the real times in the table.

M	time (s)
10000	0.016 s
100000	0.539s
200000	2.003s

We gained significant improvement only in the case of 10000. There the run times are effected by the starting process due to the small run times. So here we see improvement most likely, because I started running with the longest thread and than started the shorter ones. And this is the only case where the thread started first ain't the last to finish.

So the measurement results are what I expected. I can't see much improvement compare to the 2 thread, most likely, because there the last 30% of the outer loop takes up more or the same amount of time as the first 70%. And here the run times comes from the last 25% of the outer loop, because here the inner loops are much longer. So the results we see makes sens. If we would like to optimize the performance, than we could make the splittings another way. Either calculate where to split so that all threads have the same run time, or we could use other guessing which would results in improvement (e.g: first split point at 50% than each new split points divide the last part into half).

Now to test what we can reach with more thread let's use $M = 200000$ and measure runtime with different number of threads. The used N number are chosen to satisfy the above said property such that M/N an even integer.

N	time (s)
2	2.612 s
4	1.980 s
8	1.706 s
10	1.589 s
20	1.575 s
40	1.555 s
50	1.580 s
100	1.580 s
200	1.582 s
400	1.559 s
1000	1.560 s
5000	1.767 s

Note: my computer has 2 core in it and both has 2 core in it.
Also my implementation of the problem is not suitable to determinate how long will spiting a problem make it more efficient, since now far the longest process is the thread running the last segment. Therefore as the length of the segments are getting smaller as I increase the N value it will decrease the runtime of the last process therefore it ain't matter that I don't have so much core, because the pthread framework will split

the tasks more equally between the threads even if due to the calls and prints the overall work is greater.

So to test real improvement in the respect to the cores I made a much worse implementation of the problem, where all inner loops have equal length (M-2) and in the outer loop we iterate one by one. This code can be found in the appendix with name *Task6-2.c*. This way the different threads will have the same amount of work and thus the above runtime comparison makes sense. For the run time comparison lets use M=20000.

N	time (s)
1	0.833 s
2	0.472 s
3	0.456 s
4	0.394 s
5	0.376 s
6	0.403 s
7	0.377 s
8	0.389 s
9	0.376 s
10	0.374 s
12	0.392 s
15	0.376 s
20	0.479 s
40	0.482 s
5000	0.634 s
10000	0.758 s

As we can see in the beginning the more core reduce the runtime drastically, but using unnecessary number of core don't have an impact for a while, because there the management of the threads don't require too much time, so it can even result in better use of resources. (e.g: if some thread becomes slow for some reason, than other cores can run more work).
Also with 4 thread in my computer it would be expected to be optimal, but adding more thread results in better performance which I would explain with the above reasoning.

Task 7

The code for this task can be found under the name *Task7.c* in the appendix. This is a straightforward implementation with some printing. Every thread also write out who called it with what subprocess number.

An example output can be seen here:

```
Out:
 the main() function now calling pthread_create().
 Thread 1 started from main start running.
 Thread 2 started from main start running.
 This is the main() function after pthread_create()
 the main() function now calling pthread_join().
 Thread 2 finished creating subprocess.
 Calling subthread started from thread 1 with subnumber 1
 Calling subthread started from thread 2 with subnumber 2
 Thread 1 finished creating subprocess.
 Calling subthread started from thread 1 with subnumber 2
 Thread 1 finished running.
 Calling subthread started from thread 2 with subnumber 1
 Thread 2 finished running.
```

As we can see the sub-threads really working simultaneously. And the threads subprocess are running without a problem.

Task 8

First when we ran the program without any modification for a fixed number of core (more than 1) we can see different results due to the fact that different threads modified the same memory block. And without usage of mutex the threads can mess up each others access to the memory location. This is causing the changing and incorrect results.

Now we can add a mutex to the process. Mutex is used to create matual exclusion of some context. This process can make sure that an object is only accessed by one thread at a time.

If my understanding is correct we can create a *pthread_mutex_t* object outside the main and in the begining of the main using the *pthread_mutex_init* function, which has a return value, which can be used to make sure the initialization of the mutex object really happened.

Now in each thread we can use the *pthread_mutex_lock* and *pthread_mutex_unlock* functions to make sure only one thread writes at a time to the memory. And in the end we can use the *pthread_mutex_destroy* command in the end of the main to destroy the created object.

After adding this parts to our code the result will be the same for fixed N and fixed number of threads. The modified code can be found in the appendix as *Task8.c* with FAST=0 in the 5th line.

Since lock only grant access to one thread to the resources, so it can be useful regarding performance to lock as little part of our code as possible. Only where shared access can be confusing.

To demonstrate this I changed the N to 10^8 so the runtime will be significant. Which is in case I run it for 10 thread results in runtime 1.798s (both real and user). Now I modified the code such that adding a tmp variable which counts the sum in a thread and only after the for cycle locking the mutex object and adding the tmp to the sum. This way due to the extra variables the user time increased to 2.553s, but due to I made the threads run really parallel again the real time decreased to 0.663s. So I made the program overall run time faster even though I increased the overall used resources.

The code for this can be run by setting the FAST to 1 in the 5th line of the code in *Task8.c*.

This concludes the work with task 8.

Lab 9

Task 1

I did as the task says. The thread function is fill up a list of integers with length of a predefined value of N. So I declared a list in the thread and give the elements value. And I give it's pointer back to the main function as explained, through the join function. And than write the values out from the main. I got the values given in the thread. And in the end I freed the location to prevent memory leaks.

The code for the task can be found in the appendix under *Task9_1.c*. An output with N=10 can be found here.

```
Out:
 This is the main() function starting.
 the main() function now calling pthread_create().
 This is the main() function after pthread_create()
 the main() function now calling pthread_join().
 result in array[0]: 0
 result in array[1]: 1
 result in array[2]: 2
 result in array[3]: 3
 result in array[4]: 4
 result in array[5]: 5
 result in array[6]: 6
 result in array[7]: 7
 result in array[8]: 8
 result in array[9]: 9
```

So now we saw how to pass pointers to the threads through the create method and how to retrieve pointers from the threads through the join function.

This concludes the exercises for task 1.

Task 2

When I removed the *DoItNow=1* from the main(), than the process in main still finishes in order, but since the thread function never get the *DoItNow=1* value, so it will run in an infinite loop, so our programs never stop running and don't do anything after the writing out that the main is calling join. Therefore our program is broken, at least it ain't doing what we want it to do.

This way we achieved to wait in a thread until some process is finished. This is a possible way to do it, but it is wasting our resources, since we are running an infinite loop, which results in an almost 200% CPU usage during the wait phase (the highest value I saw with top was 197,4%).

Therefore we have a good reason to use the method of condition variable, which will result a same working scheme, but it is less heavy on the CPU.

After adding the conditional variable to the code as described in the task we see that it is working the same way as before respect to the wait process. If I run the top command now to see how much CPU the process uses this way, I can't see higher than 100% result. So now our wait our process of waiting don't use unnecessary resources till it is waiting and don't do anything useful.

If I removed the *pthread_cond_signal* call, I end up the same as when I removed the *DoItNow=1* in the original version. So the program won't stop running. The only advantage is that I don't use any computer resources for this process. The reason is that the thread won't do anything until it receives a call to continue the process. But we deleted this call, so it will never revive it, so it will never stops running. As it is asleep it ain't use too much resources, but it can't stop running, because the main is waiting to the thread to stop running in the join method, but the thread will never return.

The working version of the modified code with conditional variable is in the appendix with name *Task9_2.c.*

Task 3

First let's present the result of simply using make and running the program:

```
Out:
 Main: creating thread 0
 Main: creating thread 1
 Thread 0 starting...
 Main: creating thread 2
 Thread 1 starting...
 Main: creating thread 3
 Thread 2 starting...
 Thread 3 starting...
 Thread 0 done. Result = 4.931540e+06
 Thread 1 done. Result = 4.931540e+06
 Thread 3 done. Result = 4.931540e+06
 Thread 2 done. Result = 4.931540e+06
 Main: completed join with thread 0 having a status of 0
 Main: completed join with thread 1 having a status of 1
 Main: completed join with thread 2 having a status of 2
 Main: completed join with thread 3 having a status of 3
 Main: program completed. Exiting.
```

Now I set the type from *PTHREAD_CREATE_JOINABLE* to *PTHREAD_CREATE_DETACHED*. Now we got error message from the join calls, it makes sense, since the threads now are not joinable, so it was expected that the join call would give us error.

```
Out:
 Main: creating thread 0
 Main: creating thread 1
 Thread 0 starting...
 Thread 1 starting...
 Main: creating thread 2
 Main: creating thread 3
 Thread 2 starting...
 ERROR; return code from pthread_join() is 22
```

Therefore I remove the join call from the main, since as we see it is not working, and unnecessary. With the join calls I also removed the whole for cycle around it, because it doesn't make sense to me to try and print out reference if we don't call the join function, where the reference would be determinated. The output now:

```
Out:
 Main: creating thread 0
 Main: creating thread 1
 Thread 0 starting...
 Main: creating thread 2
 Thread 1 starting...
 Main: creating thread 3
 Main: program completed. Exiting.
 Thread 2 starting...
 Thread 3 starting...
 Thread 3 done. Result = 4.931540e+06
 Thread 0 done. Result = 4.931540e+06
 Thread 2 done. Result = 4.931540e+06
 Thread 1 done. Result = 4.931540e+06
```

Now if I delete the exit in the end of the main, I get the following output:

```
Out:
 Main: creating thread 0
 Main: creating thread 1
 Thread 0 starting...
 Main: creating thread 2
 Thread 1 starting...
 Main: creating thread 3
 Main: program completed. Exiting.
```

As we can see the program now stops running when it reaches the end of the main, and don't wait for the threads to finish their work first. So in this case the exit command works a bit similarly as the join when we used joinable threads. So it waits for the threads to be executed. At least seemingly that what's happening here.

Note: for this part I only conclude the original program code in the document, because the others are come from removing certain, well explained parts of the original code.

Task 4

If I understand correctly what the barrier function does, it makes wait the threads for each other. Precisely it waits until all thread reaches the barrier function in their running and then let every thread continue it's running.

This is in harmony with the output of the program:

```
Out:
 Hello World! 0
 Hello World! 1
 Hello World! 2
 Hello World! 3
 Hello World! 5
 Hello World! 6
 Hello World! 4
 Hello World! 7
 Bye Bye World! 7
 Bye Bye World! 1
 Bye Bye World! 0
 Bye Bye World! 3
 Bye Bye World! 2
 Bye Bye World! 5
 Bye Bye World! 6
 Bye Bye World! 4
```

So the expectation is that if we remove the barrier, than in the output the Hello and Bye Bye prints will be mixed with each other, of course with a given number the hello always be earlier. So now let's test this theory:

```
Out:
 Hello World! 0
 Hello World! 1
 Bye Bye World! 1
 Hello World! 6
 Hello World! 7
 Bye Bye World! 7
 Hello World! 4
 Bye Bye World! 4
 Hello World! 5
 Bye Bye World! 5
 Bye Bye World! 0
 Hello World! 2
 Bye Bye World! 2
 Bye Bye World! 6
 Hello World! 3
 Bye Bye World! 3
```

So I was right what will happen. So we made sure the barrier really is causing a wait between the threads.

When I run the *spinwait.c* without optimization it has the same effects as the first implementation. An example output is:

```
Out:
 Hello World! 0
 Hello World! 2
 Hello World! 1
 Hello World! 5
 Hello World! 4
 Hello World! 6
 Hello World! 7
 Hello World! 3
 Bye Bye World! 7
 Bye Bye World! 2
 Bye Bye World! 3
 Bye Bye World! 0
 Bye Bye World! 1
 Bye Bye World! 6
 Bye Bye World! 4
 Bye Bye World! 5
```

Now the output will be if I compile with **-O3** the following:

```
Out:
 Hello World! 0
 Hello World! 3
 Hello World! 2
 Hello World! 6
 Hello World! 1
 Hello World! 5
 Hello World! 7
 Hello World! 4
 Bye Bye World! 4
```

And the program don't stops running. I think the reason is that the while loop are becoming an infinite loop in the optimized version. Now we can prevent this with the usage of the volatile keyword, which means to the compiler that a variable can be changed even if it ain't in the source code. In this case by another thread. And without this the compiler with -O3 flag assume no change therefore an infinite while loop, which can be made optimal with checking the condition once.

So now we add the volatile key word to the declaration of the *state* variable. And now the program works fine even with -O3 optimization flag.

Note: for synch.c I attach the original code which works fine. For spinwait.c I attached the modified code wich compiles with optimization well.

The extra part: The -O1 flag is enough for see this anomaly without volatile. Now lets see the difference below in the assembly code:

```
With volatile -O1:
.L4:
# spinwait.c:20:   while (mystate==state) ;
        movl    state(%rip), %eax       # state, state.2_4
        cmpl    %ebx, %eax      # mystate, state.2_4
        je      .L4     #,
```

```
Without volatile -O1:
# spinwait.c:20:   while (mystate==state) ;
        movl    state(%rip), %eax       # state, state.2_4
.L4:
# spinwait.c:20:   while (mystate==state) ;
        cmpl    %ebx, %eax      # mystate, state.2_4
        je      .L4     #,
```

As we can see without the key word the compiler declares the state variable outside the loop, because it thinks it can't change in the loop, but with volatile it is getting the value of state in every iteration in the cycle, so this is the difference in the assembly code level. And whats happening is mostly what I guessed earlier.

Task 5

First I run the code in the original version after using make command. The original code can be found in the appendix as *Task9-5-1.c.* The result of the running was the following: *PI is approx 3.1415926535899894.* The runtime was in this case *real 0m0,592s*

Now I modify the code to the most obvious parallel version. So each thread does equal parts of the sum. Actually in the implementation the threads do most of the work and the main only calling the threads and do a small amount of work, only the iterations which remains from the iteration after divided into equal parts through the threads.

In the parallel version I tested two option, one where I use mutex to write into a common sum. In this case with the default iteration value I got *PI is approx. 3.1415926743284515* with runtime *real 0m0,592s.* So this way I ain't made the process faster.

The other version was with temporary variables and only adding that to sum with mutex. Now results is *PI is approx. 3.1415926743281033* and the runtime *real 0m0,351s.*

Both case was with usage of 3 threads.

As we see the results are a bit different in each case. It is due to the storage of the double numbers, it always has numerical errors and depending on how we working with it it has different size of error as we saw it on previous labs. And it is also depends on the optimizations and weather we makes it use normalized floating point numbers or not. So now we understand what causes the difference in the results.

Just for fun and to see the differences better I run the code with multiplying iteration with 10. For this purpose I set the variables to long.

version	results	time
original	3.1415926535898167	5,959s
parallel	3.1415926588186847	5,801s
with tmp	3.1415926588122001	3,125s

As we see we can reach a 2 time improvement. Now I used 8 threads. The difference of the results are still remaining.

After taking a look at the results we can see that the original serialized method is the most precise. The explanation can be that in this way the we always use the same floating point precision, resulting in the best results. But the parallel methods also produces better and better results as we increase the number of iterations.

This conclude task 5. We saw that using threads we can make the process faster and we also saw that working with floting point numbers can results in different results if we sum the values together differently.

Task 6

First I run the original program (*Task9-6-1.c*) with the usage of make. The result is : *Time: 3.506381 NUM_THREADS: 5.*

Now I modified the code, so that instead of lots of for loop in the main and little task on one thread at a time I now only calling threads once and in the i-th thread I sort to the correct space the first i-th part of the list. This implementation can be found in the appendix as *Task9-6-2.c.* Now the runtime data is : *Time: 0.945144 NUM_THREADS: 5*

As we can see we reached a significant time improvement with this modification. The computational complexity of the enumeration-sort algorithm is $\mathcal{O}(n^2)$, because for each element we iterate through the whole list. This is n^2 operation.

The merge sort has a runtime of $\mathcal{O}(nlogn)$. So it is a better algorithm in theoretical complexity. Although with the usage of the parallel computation we made an efficient implementation of this algorithm. And the merge sort is very hard to make parallel, so in certain cases this may be a faster algorithm. Mostly because for small arrays the merge sort algorithm is not too efficient.

So for comparison here are the runtime data from Lab6 Task1 of the merge sort with $-O3$. Same as the enum_sort:

sort	N	time(s)
merge	10000	0.004
merge	100000	0.037
merge	1000000	0.111
merge	10000000	1.114
merge	100000000	12.991
enum	10000	0.016
enum	100000	0.927
enum	1000000	182.957391

We can see that for large lists the enum method getting slower much faster, than with merge sort. So this is enough proof for that in large lists the merge sort is much better, than the enumerate sort.

Task 7

First I ran the original code with make and take a look at the runtime with $n = 1000$. The results was: *3.186866 wall seconds* and a correct result. The code for this is *Task9-7-1.c*.

Now I made my first parallel version. From the three for cycle I run the inner two in the threads, and I call THREAD_NUM amount of thread at a time inside the outer for loop, if it doesn't become greater, than the limit. The runtime for this implementation is : *Elapsed time: 2.180577 wall seconds.* This code can be found as *Task9-7-2.c*.

To decide weather or not this have the same problem as the last task, so it is running too small process in threads. So I make an implementation where I make more computation in each thread, so I make a same change as in Task 6, and I only call threads once and make iteration inside them.

Now I make time measurements with $n = 1000$ and using THREAD_NUM of 2, because my laptop little resources this will result in the best runtime. Now my second implementation runs faster *Elapsed time: 1.481405 wall seconds* and I could only fast up the code a bit with the change to run time *Elapsed time: 1.445460 wall seconds.* And these results are 2 time faster than the original, which is around the improvement I could reach in my computer.

We will find out for which n-s are the parallelization worth it. For this purpose I will use my last implementation as the parallel one.

sort	N	time(s)
serial	1000	3.199540
serial	500	0.143459
serial	400	0.079965
serial	300	0.054702
serial	200	0.013072
par	1000	1.448582
par	500	0.085197
par	400	0.050932
par	300	0.036233
par	200	0.027863

As we can see between 300 and 200 will the serialized version become faster.

With this we finished Task 9.

Appendix -the C codes of the tasks

Codes of Lab8 Task 1

Code: *Task1-1.c*

```c
#include <stdio.h>
#include <pthread.h>
#include<unistd.h>

static int s= 1;
static int n=2;
static int l= 4;

void* the_thread_func(void* arg) {
  int a =*(int *)arg;
  for(int i=0; i<l;i++){
    sleep(s);
    printf("Thread: %d\n",a);}
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  /* Start thread. */
  pthread_t thread[n];
  printf("the main() function now calling pthread_create().\n
     ");
  void* arg[n];
  int a[n];

  for(int i=0; i<n;i++){
  a[i]=i+1;
  arg[i] = &a[i];
  pthread_create(&thread[i], NULL, the_thread_func, arg[i]);
  }

  printf("This is the main() function after pthread_create()\
     n");

  /* Do something here? */
  for (int i = 0; i < l; i++)
  {printf("main\n");
  sleep(s);
  }

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
     ;
  for(int i=0; i<n;i++)
  pthread_join(thread[i], NULL);

  return 0;
}
```

Code: ***Task1-2.c***

```
#include <stdio.h>
#include <pthread.h>
#include<unistd.h>

static int s= 1;
static int n=2;
static int l= 4;

void* the_thread_func(void* arg) {
  int a =*(int *)arg;
  for(int i=0; i<l;i++){
    sleep(s);
    printf("Thread: %d\n",a);}
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  /* Start thread. */
  pthread_t thread[n];
  printf("the main() function now calling pthread_create().\n
     ");
  void* arg[n];
  int a[n];

  for(int i=0; i<n;i++){
  a[i]=i+1;
  arg[i] = &a[i];
  pthread_create(&thread[i], NULL, the_thread_func, arg[i]);
  }

  printf("This is the main() function after pthread_create()\
     n");

  /* Do something here? */
  for (int i = 0; i < l; i++)
  {printf("main\n");
```

```c
  sleep(s);
  }

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
     ;
  for(int i=0; i<n;i++)
  pthread_join(thread[i], NULL);

  return 0;
}
```

Codes of Lab8 Task 2

Code: *Task2.c*

```
#include <stdio.h>
#include <pthread.h>

void* the_thread_func(void* arg) {
  double * data=(double *)arg;
  for(int i=0; i<3;i++)
    printf("Data from thread: %lf\n", data[i]);
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  double data_for_thread[3];
  data_for_thread[0] = 5.7;
  data_for_thread[1] = 9.2;
  data_for_thread[2] = 1.6;

  double data_for_thread_B[3];
  data_for_thread_B[0] = -1;
  data_for_thread_B[1] = -3.1415;
  data_for_thread_B[2] = -0.6;

  /* Start thread. */
  pthread_t thread_1;
  pthread_t thread_2;
  printf("the main() function now calling pthread_create().\n
     ");
  pthread_create(&thread_1, NULL, the_thread_func,
     data_for_thread);
  pthread_create(&thread_2, NULL, the_thread_func,
     data_for_thread_B);

  printf("This is the main() function after pthread_create()\
     n");

```

```c
  /* Do something here? */

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
    ;
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);

  return 0;
}
```

Codes of Lab8 Task 3

Code: ***Task3.c***

```c
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

const long int N1 = 400000000;
const long int N2 = 400000000;

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

void* the_thread_func(void* arg) {
  double start = get_wall_seconds();
  long int i;
  long int sum = 0;
  for(i = 0; i < N2; i++)
    sum += 7;
  double end = get_wall_seconds();
  /* OK, now we have computed sum. Now copy the result to the
      location given by arg. */
  long int * resultPtr;
  resultPtr = (long int *)arg;
  *resultPtr = sum;

  printf("Runtime thread - computations runetime: %lf\n", end
     -start);
  return NULL;
}

int main() {
  double start_tima_all=get_wall_seconds();
  printf("This is the main() function starting.\n");

  long int thread_result_value = 0;

  /* Start thread. */
  pthread_t thread;
  printf("the main() function now calling pthread_create().\n
    ");
  pthread_create(&thread, NULL, the_thread_func, &
     thread_result_value);

  printf("This is the main() function after pthread_create()\
    n");

  double start=get_wall_seconds();
  long int i;
  long int sum = 0;
  for(i = 0; i < N1; i++)
    sum += 7;
  double end=get_wall_seconds();

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
    ;
  pthread_join(thread, NULL);

  printf("sum computed by main() : %ld\n", sum);
  printf("sum computed by thread : %ld\n",
     thread_result_value);
  long int totalSum = sum + thread_result_value;
  printf("totalSum : %ld\n", totalSum);

  double end_time_all=get_wall_seconds();
  printf("Total runetime: %lf\n", end_time_all-start_tima_all
    );
  printf("Runtime main - computations runetime: %lf\n", end-
     start);
  return 0;
}
```

Codes of Lab8 Task 4

Code: *Task4-1.c*

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

int main(int argc, char**args){

int M =(atoi(args[1]));
if(M==0 || M<0){printf("Error: n needs to be an integer
   bigger to or equal to 1\n");}

int counter=1;
for(int i=3; i<M; i=i+2){
     int prime=1;
     for (int j = 2; j < i; j++)
     {
          if(i%j==0){
               prime=0;
               break;
          }
     }
     if (prime)
          counter++;
}
printf("The number of primes from 1 to %d is %d\n" , M,
   counter);
return 0;
}
```

Code: ***Task4-2.c***

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* the_thread_fun(void* arg){
int * data= (int *)arg;
int M=*data;
int counter=0;
for(int i=0.7*M+1; i<M; i=i+2){
    int prime=1;
    for (int j = 2; j < i; j++)
    {
        if(i%j==0){
            prime=0;
            break;
        }
    }
    if (prime)
        counter++;
}
*data = counter;
}


int main(int argc, char**args){

int M =(atoi(args[1]));
if(M==0 || M<0){printf("Error: n needs to be an integer
   bigger to or equal to 1\n");}

int counter=1;


int arg =M;
pthread_t thread;
pthread_create(&thread, NULL, the_thread_fun, &arg);

//main
for(int i=3; i<0.7*M; i=i+2){
    int prime=1;
    for (int j = 2; j < i; j++)
    {
        if(i%j==0){
            prime=0;
            break;
        }
    }
    if (prime)
        counter++;
}
pthread_join(thread, NULL);

counter+=arg;
printf("The number of primes from 1 to %d is %d\n" , M,
   counter);
return 0;
}
```

Codes of Lab8 Task 5

Code: *Task5.c*

```c
#include <stdio.h>
#include <pthread.h>

static int n=10;

void* the_thread_func(void* arg) {
  int a =*(int *)arg;
    printf("Thread: %d\n",a);
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  /* Start thread. */
  pthread_t thread[n];
  printf("the main() function now calling pthread_create().\n
     ");
  void* arg[n];
  int a[n];

  for(int i=0; i<n;i++){
  a[i]=i+1;
  arg[i] = &a[i];
  pthread_create(&thread[i], NULL, the_thread_func, arg[i]);
  }

  printf("This is the main() function after pthread_create()\
     n");

  /* Do something here? */
    printf("main\n");

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
     ;
  for(int i=0; i<n;i++)
  pthread_join(thread[i], NULL);

  return 0;
}
```

Codes of Lab8 Task 6

Code: ***Task-6.c***

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* the_thread_fun(void* arg){
int * data= (int *)arg;
int L=*data;
data ++;
int U=*data;
int counter=0;
for(int i=L; i<U; i=i+2){
     int prime=1;
     for (int j = 2; j < i; j++)
     {
          if(i%j==0){
               prime=0;
               break;
          }
     }
     if (prime)
          counter++;
}
printf("Thread bounds and count: U=%d, L=%d, count=%d\n", U,L
    ,counter);
*data = counter;
}


int main(int argc, char**args){

int M =(atoi(args[1]));
if(M==0 || M<0){printf("Error: M needs to be an integer
    bigger to or equal to 1\n");}

int N =(atoi(args[2]));
if(N==0 || N<0){printf("Error: N needs to be an integer
    bigger to or equal to 1\n");}

int counter=1;


int arg[N-1][2];
pthread_t thread[N-1];


for(int i=0; i<N-1; i++){
arg[i][0]=M-(i+1)*(M/N)+1;
arg[i][1]=M-i*(M/N);
pthread_create(&thread[i], NULL, the_thread_fun, &arg[i]);
}

//main
for(int i=3; i<M/N; i=i+2){
      int prime=1;
      for (int j = 2; j < i; j++)
      {
            if(i%j==0){
                  prime=0;
                  break;
            }
      }
      if (prime)
            counter++;
}
for (int i = 0; i < N-1; i++)
{
pthread_join(thread[i], NULL);
counter+=arg[i][1];
}


printf("The number of primes from 1 to %d is %d\n" , M,
    counter);
return 0;
}
```

Code: ***Task6-2.c***

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* the_thread_fun(void* arg){
int * data= (int *)arg;
int M=*data;
data ++;
int L=*data;
data ++;
int U=*data;
int counter=0;
for(int i=L; i<U;i++){
     int prime=1;
     for (int j = 2; j < M; j++)
     {
          if(i%j==0 && j!=i){
               prime=0;
          }
     }
     if (prime)
          counter++;
}
printf("Thread bounds and count:M=%d, U=%d, L=%d, count=%d\n"
    , M,U,L,counter);
*data = counter;
}


int main(int argc, char**args){

int M =(atoi(args[1]));
if(M==0 || M<0){printf("Error: M needs to be an integer
    bigger to or equal to 1\n");}

int N =(atoi(args[2]));
if(N==0 || N<0){printf("Error: N needs to be an integer
    bigger to or equal to 1\n");}

int counter=0;


int arg[N-1][3];
pthread_t thread[N-1];


for(int i=0; i<N-1; i++){
arg[i][0]=M;
arg[i][1]=(i+1)*(M/N);
arg[i][2]=(i+2)*(M/N);
pthread_create(&thread[i], NULL, the_thread_fun, &arg[i]);
}

//main
for(int i=2; i<M/N;i++){
    int prime=1;
    for (int j = 2; j < M; j++)
    {
        if(i%j==0 && i!=j){
            prime=0;
        }
    }
    if (prime)
        counter++;
}
for (int i = 0; i < N-1; i++)
{
pthread_join(thread[i], NULL);
counter+=arg[i][2];
}


printf("The number of primes from 1 to %d is %d\n" , M,
   counter);
return 0;
}
```

Codes of Lab8 Task 7

Code: ***Task7.c***

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

static int n=2;

void* the_thread_func_sub(void* arg) {
    int * data= (int *)arg;
    int a =*data;
    data++;
    int b =*data;
    printf("Calling subthread started from thread %d with
       subnumber %d\n",b,a);

    return NULL;
}

void* the_thread_func_main(void* arg) {
  int a =*(int *)arg;
  printf("Thread %d started from main start running.\n",a);
  pthread_t threadi[n];
  int argi[2*n];
  int ai[n];

  for(int i=0; i<n;i++){
  argi[2*i] = i+1;
  argi[2*i+1] = a;
  pthread_create(&threadi[i], NULL, the_thread_func_sub, &
     argi[2*i]);
  }

printf("Thread %d finished creating subprocess.\n",a);
  for(int i=0; i<n; i++){
    pthread_join(threadi[i],NULL);
  }
printf("Thread %d finished running.\n",a);
```

```
  return NULL;
}

int main(){

pthread_t thread[n];
  printf("the main() function now calling pthread_create().\n
     ");
  void * arg[n];
  int a[n];

  for(int i=0; i<n;i++){
  a[i]=i+1;
  arg[i] = &a[i];
  pthread_create(&thread[i], NULL, the_thread_func_main, arg[
     i]);
  }

  printf("This is the main() function after pthread_create()\
     n");
  printf("the main() function now calling pthread_join().\n")
     ;
  for(int i=0; i<n;i++)
  pthread_join(thread[i], NULL);

  return 0;
}
```

Codes of Lab8 Task 8

Code: ***Task8.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define FAST=0

long int sum = 0;
int N = 100000000;
pthread_mutex_t lock;


void* the_thread_func(void* arg) {
  #if FAST
  long int tmp=0;
  for(int i = 1; i <= N; ++i)
        tmp += 1;

  pthread_mutex_lock(&lock);
  sum+=tmp;
  pthread_mutex_unlock(&lock);
  return NULL;
  #else
  pthread_mutex_lock(&lock);
  for(int i = 1; i <= N; ++i)
        sum += 1;
  pthread_mutex_unlock(&lock);
  return NULL;
  #endif
}

int main(int argc, char **argv) {

  if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

  if(argc != 2) {printf("Usage: %s N\n", argv[0]); return -1; }

  printf("This is the main() function starting.\n");

  int N = atoi(argv[1]);

  /* Start thread. */
  printf("the main() function now calling pthread_create().\n");
  pthread_t threads[N];
  for(int i = 0; i < N; i++)
    pthread_create(&threads[i], NULL, the_thread_func, NULL);

  printf("This is the main() function after pthread_create()\n");


  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n");
  for(int i = 0; i < N; i++)
    pthread_join(threads[i], NULL);

  printf("sum = %ld\n", sum);

  pthread_mutex_destroy(&lock);
  return 0;
}
```

Codes of Lab9 Task 1

Code: *Task9_1.c*

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

int N=10;

void* the_thread_func(void* arg) {
  /* Do something here? */
  int *p=(int *) malloc(N*sizeof(int));
  for(int i=0; i<N; i++){
   p[i]=i;
  }
  return p;
}

int main() {
  printf("This is the main() function starting.\n");

  /* Start thread. */
  pthread_t thread;
  printf("the main() function now calling pthread_create().\n
    ");
  if(pthread_create(&thread, NULL, the_thread_func, NULL) !=
    0) {
    printf("ERROR: pthread_create failed.\n");
    return -1;
  }

  printf("This is the main() function after pthread_create()\
    n");

  /* Do something here? */
  long int sum=0;
  for(int i=0;i<100000000;i++)
    sum+=i;
  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n")
    ;

  int * res;
  if(pthread_join(thread, (void**)(&res)) != 0) {
    printf("ERROR: pthread_join failed.\n");
    return -1;
  }
  for(int i=0; i<N; i++)
    printf("result in array[%d]: %d\n",i,res[i]);

  free(res);
  return 0;
}
```

Codes of Lab9 Task 2

Code: *Task9_2.c*

```c
#include <stdio.h>
#include <pthread.h>

int DoItNow = 0;


pthread_mutex_t m;
pthread_cond_t cond;

void* thread_func(void* arg) {
  printf("This is thread_func() starting, now entering loop
     to wait until DoWorkNow is set...\n");
  while(1) {
    /* Check if DoItNow has been set to 1. */
    int shouldBreakLoop = 0;
    pthread_mutex_lock(&m);
    if(DoItNow == 1)
      shouldBreakLoop = 1;
    else
      pthread_cond_wait(&cond, &m); //between lock and unlock
    pthread_mutex_unlock(&m);
    if(shouldBreakLoop == 1)
      break;
  }
  printf("This is thread_func() after the loop.\n");
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  pthread_cond_init(&cond,NULL);
  pthread_mutex_init(&m, NULL);

  /* Start thread. */
  pthread_t thread;
  printf("the main() function now calling pthread_create().\n");
  pthread_create(&thread, NULL, thread_func, NULL);
  printf("This is the main() function after pthread_create()\n");

  /* Here we let the main thread do some work. */
  long int k;
  double x = 1;
  for(k = 0; k < 2000000000; k++)
    x *= 1.00000000001;
  printf("main thread did some work, x = %f\n", x);

  pthread_mutex_lock(&m);
  DoItNow = 1;
  pthread_cond_signal(&cond); // Waking up the process in the thread
  pthread_mutex_unlock(&m);

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n");
  pthread_join(thread, NULL);
  printf("This is the main() function after calling pthread_join().\n");

  pthread_cond_destroy(&cond);
  pthread_mutex_destroy(&m);

  return 0;
}
```

Codes of Lab9 Task 3

Code: ***Task9_3.c***

```c
/*
   *************************************************************************
 * FILE: join.c
 *
 ***************************************************************************
     */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS     4

void *BusyWork(void *t)
{
  int i;
  long tid;
  double result=0.0;
  tid = (long)t;
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<20000000; i++)
    {
      result = result + sin(i) * tan(i);
    }
  printf("Thread %ld done. Result = %e\n",tid, result);
  pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)
     ;

  for(t=0; t<NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)
       t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\
         n", rc);
      exit(-1);
    }
  }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n"
         , rc);
      exit(-1);
    }
    printf("Main: completed join with thread %ld having a
       status of %ld\n",t,(long)status);
  }

  printf("Main: program completed. Exiting.\n");
  pthread_exit(NULL);
}
```

Codes of Lab9 Task 4

Code: *synch.c*

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

pthread_mutex_t lock;
pthread_cond_t mysignal;
int waiting = 0;
int state = 0;

void barrier() {
  int mystate;
  pthread_mutex_lock (&lock);
  mystate=state;
  waiting++;
  if (waiting == NUM_THREADS) {
    waiting = 0;
    state = 1 - mystate;
    pthread_cond_broadcast(&mysignal);
  }
  while (mystate == state) {
    pthread_cond_wait(&mysignal, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void* HelloWorld(void* arg) {
  long id=(long)arg;
  printf("Hello World! %ld\n", id);
  barrier();
  printf("Bye Bye World! %ld\n", id);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  long t;
  // Initialize things
  pthread_cond_init(&mysignal, NULL);
  pthread_mutex_init(&lock, NULL);
  // Create threads
  for(t=0; t<NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, HelloWorld, (void*)t);
  // Wait for threads to finish
  for(t=0; t<NUM_THREADS; t++)
    pthread_join(threads[t], NULL);
  // Cleanup
  pthread_cond_destroy(&mysignal);
  pthread_mutex_destroy(&lock);
  // Done!
  return 0;
}
```

Code: ***spinwait.c***

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

pthread_mutex_t lock;
int waiting = 0;
volatile int state = 0;

void barrier() {
  int mystate;
  pthread_mutex_lock (&lock);
  mystate=state;
  waiting++;
  if (waiting==NUM_THREADS) {
    waiting=0; state=1-mystate;
  }
  pthread_mutex_unlock (&lock);
  while (mystate==state) ;
}

void* HelloWorld(void* arg) {
  long id=(long)arg;
  printf("Hello World! %ld\n", id);
  barrier();
  printf("Bye Bye World! %ld\n", id);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  long t;
  // Initializa things
  pthread_mutex_init(&lock,NULL);
  // Create threads
  for(t=0 ; t<NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, HelloWorld, (void*)t);
  // Wait for threads to finish
```

```
    for(t=0 ; t<NUM_THREADS; t++)
        pthread_join(threads[t], NULL);
    // Cleanup
    pthread_mutex_destroy(&lock);
    // Done!
    return 0;
}
```

Codes of Lab9 Task 5

Code: ***Task9-5-1.c***

```c
/*****************************************************************************
 * This program calculates pi using C
 *
 *****************************************************************************/
#include <stdio.h>

int main(int argc, char *argv[]) {

  long int i;
  const long int intervals = 500000000;
  double sum, dx, x;

  dx  = 1.0/intervals;
  sum = 0.0;

  for (i = 1; i <= intervals; i++) {
    x = dx*(i - 0.5);
    sum += dx*4.0/(1.0 + x*x);
  }

  printf("PI is approx. %.16f\n",  sum);

   return 0;
}
```

Code: *Task9-5-2.c*

```c
/*
   ********************************************************************
 * This program calculates pi using C
 *
 ********************************************************************
    */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS      8

pthread_mutex_t lock;
double sum = 0.0;
const long int intervals = 5000000000;
double dx =1.0/intervals;

void * thread_funct(void* arg){
  double x; long int L,U;
  double tmp =0.0;
  long bnd=(long)arg;
  L=bnd*(intervals/NUM_THREADS);
  U=(bnd+1)*(intervals/NUM_THREADS);
  for (long i = L; i <= U; i++) {
    x = dx*(i - 0.5);
    tmp += dx*4.0/(1.0 + x*x);
  }

  pthread_mutex_lock(&lock);
    sum+=tmp;
  pthread_mutex_unlock(&lock);
  return NULL;
}

int main(int argc, char *argv[]) {

  pthread_t threads[NUM_THREADS];
  pthread_mutex_init(&lock, NULL);
  long i;
  double x;

  for(i=0; i<NUM_THREADS; i++)
    pthread_create(&threads[i], NULL, thread_funct, (void*)i)
        ;

  // Wait for threads to finish(intervals/NUM_THREADS)
  for(i=0; i<NUM_THREADS; i++)
    pthread_join(threads[i], NULL);

  pthread_mutex_lock(&lock);
  for (i = NUM_THREADS*(intervals/NUM_THREADS)+1; i <=
     intervals; i++) {
    x = dx*(i - 0.5);
    sum += dx*4.0/(1.0 + x*x);
  }
  pthread_mutex_unlock(&lock);

  printf("PI is approx. %.16f\n", sum);
  pthread_mutex_destroy(&lock);
  return 0;
}
```

Codes of Lab9 Task 6

Code: *Task9-6-1.c*

```c
/*
   ************************************************************************
 * Enumeration sort
 *
 **********************************************************************
    */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

#define NUM_THREADS      5
#define len 100000

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

double indata[len], outdata[len];

void *findrank(void *arg)
{
        int rank,i;
        long j=(long)arg;

        rank=0;
        for (i=0;i<len;i++)
                if (indata[i]<indata[j]) rank++;
        outdata[rank]=indata[j];
        pthread_exit(NULL);
}

```

```

int main(int argc, char *argv[]) {

  pthread_t threads[NUM_THREADS];
  pthread_attr_t attr;
  int i, j, t;
  long el;
  void *status;

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)
      ;

  // Generate random numbers
  for (i=0;i<len;i++){
      indata[i]=drand48();
      outdata[i]=-1.0;
  }

  // Enumeration sort
  double startTime = get_wall_seconds();
  for (j=0;j<len;j+=NUM_THREADS)
    {
              for(t=0; t<NUM_THREADS; t++) {
                    el=j+t;
                  pthread_create(&threads[t], &attr,
                     findrank, (void *)el); }

              for(t=0; t<NUM_THREADS; t++)
                    pthread_join(threads[t], &status);
    }
  double timeTaken = get_wall_seconds() - startTime;
  printf("Time: %f  NUM_THREADS: %d\n", timeTaken,
     NUM_THREADS);

  // Check results, -1 implies data same as the previous
     element
    for (i=0; i<len-1; i++)
      if (outdata[i]>outdata[i+1] && outdata[i+1]>-1)
        printf("ERROR: %f,%f\n", outdata[i],outdata[i+1]);

    return 0;
}
```

Code: ***Task9-6-2.c***

```c
/*
   *****************************************************************
 * Enumeration sort
 *
 *****************************************************************
    */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

#define NUM_THREADS      4
#define len 1000000

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

double indata[len], outdata[len];

void *findrank(void *arg)
{
        int rank,i;
        long bnd=(long)arg;

    //creating bound variables
    long L = bnd*(len/NUM_THREADS);
    long U = (bnd+1)*(len/NUM_THREADS);
    if(bnd+1==NUM_THREADS)
        U=len;

    for(long j=L;j<=U;j++){
        rank=0;
        for (i=0;i<len;i++)
```

```
                  if (indata[i]<indata[j]) rank++;
          outdata[rank]=indata[j];
      }
          pthread_exit(NULL);
}


int main(int argc, char *argv[]) {

   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
   int i;
   void *status;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)
       ;

   // Generate random numbers
   for (i=0;i<len;i++){
        indata[i]=drand48();
        outdata[i]=-1.0;
   }

   // Enumeration sort
   double startTime = get_wall_seconds();
                  for(long t=0; t<NUM_THREADS; t++) {
                        pthread_create(&threads[t], &attr,
                            findrank, (void *)t); }

                  for(long t=0; t<NUM_THREADS; t++)
                              pthread_join(threads[t], &status);


   double timeTaken = get_wall_seconds() - startTime;
   printf("Time: %f  NUM_THREADS: %d\n", timeTaken,
       NUM_THREADS);

   // Check results, -1 implies data same as the previous
       element
```

```
    for (i=0; i<len-1; i++)
      if (outdata[i]>outdata[i+1] && outdata[i+1]>-1)
        printf("ERROR: %f,%f\n", outdata[i],outdata[i+1]);

  return 0;
}
```

Codes of Lab9 Task 7

Code: ***Task9-7-1.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

double **A,**B,**C;
int n;

int main(int argc, char *argv[]) {
  int i, j, k;

  if(argc != 2) {
    printf("Please give one argument: the matrix size n\n");
    return -1;
  }

  n = atoi(argv[1]);

  //Allocate and fill matrices
  A = (double **)malloc(n*sizeof(double *));
  B = (double **)malloc(n*sizeof(double *));
  C = (double **)malloc(n*sizeof(double *));
  for(i=0;i<n;i++){
    A[i] = (double *)malloc(n*sizeof(double));
    B[i] = (double *)malloc(n*sizeof(double));
    C[i] = (double *)malloc(n*sizeof(double));
  }

  for (i = 0; i<n; i++)
    for(j=0;j<n;j++){
      A[i][j] = rand() % 5 + 1;
      B[i][j] = rand() % 5 + 1;
      C[i][j] = 0.0;
    }

  printf("Doing matrix-matrix multiplication...\n");
  double startTime = get_wall_seconds();

  // Multiply C=A*B
  for(i=0; i<n; i++)
    for (j=0; j<n; j++)
      for (k=0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];

  double timeTaken = get_wall_seconds() - startTime;
  printf("Elapsed time: %f wall seconds\n", timeTaken);

  // Correctness check (let this part remain serial)
  printf("Verifying result correctness for a few result matrix elements...\n");
  int nElementsToVerify = 5*n;
  double max_abs_diff = 0;
  for(int el = 0; el < nElementsToVerify; el++) {
    i = rand() % n;
    j = rand() % n;
    double Cij = 0;
    for(k = 0; k < n; k++)
      Cij += A[i][k] * B[k][j];
    double abs_diff = fabs(C[i][j] - Cij);
    if(abs_diff > max_abs_diff)
      max_abs_diff = abs_diff;
  }
  printf("max_abs_diff = %g\n", max_abs_diff);
  if(max_abs_diff > 1e-10) {
    for(i = 0; i < 10; i++)
      printf("ERROR: TOO LARGE DIFF. SOMETHING IS WRONG.\n");
    return -1;
  }
  printf("OK -- result seems correct!\n");

  // Free memory
  for(i=0;i<n;i++){
    free(A[i]);
    free(B[i]);
    free(C[i]);
  }
  free(A);
  free(B);
  free(C);

  return 0;
}
```

Code: ***Task9-7-3.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>
#include <pthread.h>
#define NUM_THREADS     2

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}


double **A,**B,**C;
int n;

void * thread(void * arg){
    long bnd=(long)arg;

    //creating bound variables
    long L = bnd*(n/NUM_THREADS);
    long U = (bnd+1)*(n/NUM_THREADS);
    if(bnd+1==NUM_THREADS)
        U=n;

    for(long i=L;i<U;i++){
        for (int j=0; j<n; j++)
        for (int k=0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];
    }
    return NULL;
}

int main(int argc, char *argv[]) {
  int i, j, k;
  pthread_t threads[NUM_THREADS];

  if(argc != 2) {
    printf("Please give one argument: the matrix size n\n");
    return -1;
  }

  n = atoi(argv[1]);

  //Allocate and fill matrices
  A = (double **)malloc(n*sizeof(double *));
  B = (double **)malloc(n*sizeof(double *));
  C = (double **)malloc(n*sizeof(double *));
  for(i=0;i<n;i++){
    A[i] = (double *)malloc(n*sizeof(double));
    B[i] = (double *)malloc(n*sizeof(double));
    C[i] = (double *)malloc(n*sizeof(double));
  }

  for (i = 0; i<n; i++)
    for(j=0;j<n;j++){
      A[i][j] = rand() % 5 + 1;
      B[i][j] = rand() % 5 + 1;
      C[i][j] = 0.0;
    }

  printf("Doing matrix-matrix multiplication...\n");
  double startTime = get_wall_seconds();

  // Multiply C=A*B
              for(long t=0; t<NUM_THREADS; t++) {
                  pthread_create(&threads[t], NULL, thread,
                      (void *)t); }

              for(long t=0; t<NUM_THREADS; t++)
          pthread_join(threads[t], NULL);

  double timeTaken = get_wall_seconds() - startTime;
  printf("Elapsed time: %f wall seconds\n", timeTaken);

```

```c
  // Correctness check (let this part remain serial)
  printf("Verifying result correctness for a few result matrix elements...\n");
  int nElementsToVerify = 5*n;
  double max_abs_diff = 0;
  for(int el = 0; el < nElementsToVerify; el++) {
    i = rand() % n;
    j = rand() % n;
    double Cij = 0;
    for(k = 0; k < n; k++)
      Cij += A[i][k] * B[k][j];
    double abs_diff = fabs(C[i][j] - Cij);
    if(abs_diff > max_abs_diff)
      max_abs_diff = abs_diff;
  }
  printf("max_abs_diff = %g\n", max_abs_diff);
  if(max_abs_diff > 1e-10) {
    for(i = 0; i < 10; i++)
      printf("ERROR: TOO LARGE DIFF. SOMETHING IS WRONG.\n");
    return -1;
  }
  printf("OK -- result seems correct!\n");

  // Free memory
  for(i=0;i<n;i++){
    free(A[i]);
    free(B[i]);
    free(C[i]);
  }
  free(A);
  free(B);
  free(C);

  return 0;
}
```

Code: ***Task9-7-3.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>
#include <pthread.h>
#define NUM_THREADS     2

 static double get_wall_seconds() {
   struct timeval tv;
   gettimeofday(&tv, NULL);
   double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
   return seconds;
 }


 double **A,**B,**C;
 int n;

 void * thread(void * arg){
     long bnd=(long)arg;

     //creating bound variables
     long L = bnd*(n/NUM_THREADS);
     long U = (bnd+1)*(n/NUM_THREADS);
     if(bnd+1==NUM_THREADS)
         U=n;

     for(long i=L;i<U;i++){
         for (int j=0; j<n; j++)
         for (int k=0; k<n; k++)
         C[i][j] += A[i][k] * B[k][j];
     }
     return NULL;
 }

 int main(int argc, char *argv[]) {
   int i, j, k;
```

```
  pthread_t threads[NUM_THREADS];

  if(argc != 2) {
    printf("Please give one argument: the matrix size n\n");
    return -1;
  }

  n = atoi(argv[1]);

  //Allocate and fill matrices
  A = (double **)malloc(n*sizeof(double *));
  B = (double **)malloc(n*sizeof(double *));
  C = (double **)malloc(n*sizeof(double *));
  for(i=0;i<n;i++){
    A[i] = (double *)malloc(n*sizeof(double));
    B[i] = (double *)malloc(n*sizeof(double));
    C[i] = (double *)malloc(n*sizeof(double));
  }

  for (i = 0; i<n; i++)
    for(j=0;j<n;j++){
      A[i][j] = rand() % 5 + 1;
      B[i][j] = rand() % 5 + 1;
      C[i][j] = 0.0;
    }

  printf("Doing matrix-matrix multiplication...\n");
  double startTime = get_wall_seconds();

  // Multiply C=A*B
              for(long t=0; t<NUM_THREADS; t++) {
                  pthread_create(&threads[t], NULL, thread,
                      (void *)t); }

              for(long t=0; t<NUM_THREADS; t++)
          pthread_join(threads[t], NULL);

  double timeTaken = get_wall_seconds() - startTime;
  printf("Elapsed time: %f wall seconds\n", timeTaken);

  // Correctness check (let this part remain serial)
  printf("Verifying result correctness for a few result matrix elements...\n");
  int nElementsToVerify = 5*n;
  double max_abs_diff = 0;
  for(int el = 0; el < nElementsToVerify; el++) {
    i = rand() % n;
    j = rand() % n;
    double Cij = 0;
    for(k = 0; k < n; k++)
      Cij += A[i][k] * B[k][j];
    double abs_diff = fabs(C[i][j] - Cij);
    if(abs_diff > max_abs_diff)
      max_abs_diff = abs_diff;
  }
  printf("max_abs_diff = %g\n", max_abs_diff);
  if(max_abs_diff > 1e-10) {
    for(i = 0; i < 10; i++)
      printf("ERROR: TOO LARGE DIFF. SOMETHING IS WRONG.\n");
    return -1;
  }
  printf("OK -- result seems correct!\n");

  // Free memory
  for(i=0;i<n;i++){
    free(A[i]);
    free(B[i]);
    free(C[i]);
  }
  free(A);
  free(B);
  free(C);

  return 0;
}
```