GEM5 Assignment Phase 2

Submitted By -

Apurv Ashish **s167503**

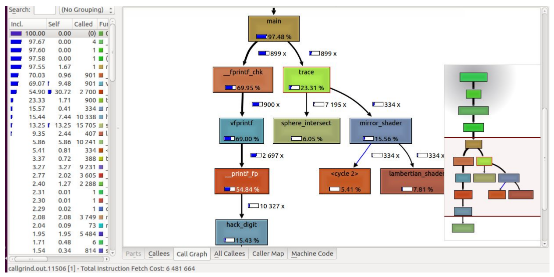
Advait Shashi Kumar **s160416**

Introduction

Parallelism can be implemented by threads, in shared multiprocessor architectures, where a thread is an independent stream of instructions and can be run by the processor. In this assignment, we had to port ray tracer on two multicore system. In summary, we could reduce computation time, by porting the binary of run tracer on multiple processors. In order to do so, we had to utilize Pthread library. Pthread is short for portable operating system interface (POSIX), which allows us to spawn a new concurrent process flow. It can most benefit us when it is on multi-processor or multi-core systems, in which we can schedule for a particular thread to run on a particular processor.

Valgrind Profiler

In the initial phase of this assignment, we first used the the valgrind profiler, to identify the process intensive sections of the code, the profiler was run on the binary of ray tracer and it gave us valuable statistics about the program.



```
Search:        (No Grouping)
Incl.     Self    Called  Fur
100.00    0.00       (0)
 97.67    0.00         4
 97.60    0.00         1
 97.58    0.00         1
 97.55    1.67         1
 70.03    0.96       901
 69.07    9.48       901
 54.90   30.72     2 700
 23.33    1.71       900
 15.57    0.41       334
 15.44    7.44    10 338
 13.25   13.25    15 705
  9.35    2.44       407
  5.86    5.86    10 241
  5.41    0.81       334
  3.37    0.72       388
  3.27    3.27     9 231
  2.77    2.02     3 605
  2.40    1.27     2 288
  2.31    0.01         1
  2.30    0.01         1
  2.29    0.02         1
  2.08    2.08     3 749
  2.04    0.09        73
  1.95    1.95     5 484
  1.71    0.48         6
  1.54    0.34       814
callgrind.out.11506 [1] - Total Instruction Fetch Cost: 6 481 664
```

By analyzing the result we were able to identify that main was subdivided into two processes __fprintf_chk and trace. In the first iteration of the porting, we ported the __fprintf_chk on multiple threads, but unlike expected, the execution time increased. It was because of the fact that we were copying the "xyz.ppm" file from the individual threads and then appending it to the main res.ppm file. This whole process was entirely based on the hard-disk I/O capability. Due to single header, the execution time was increasing becauses of the relative slowness of reading and writing the data back to the file in the hard-disk. Hence, porting __fpritntf_chk turned out to be redundant as it used the same memory space and by doing so further increased the computation time. In the next iteration, we ported the trace function which was running around approximately 23% of the main function. The four for-loops are the bottleneck of the function, hence we broke the first four loop counters in fragments and ported it to different threads.

Code Modifications

Trace block in the main.c file consisted of 4 nested for-loops, which did all the computation for making the ray coming out of the camera, consequently the calculated values were stored in res0, res1 and res2 and later transformed into a ppm file as output.

The header file "#include <pthread.h>" has to be included to enable the multi-threading function. The pthread_t is used to declare the threads. The next step is to create threads using Pthread_create which creates and execute a thread. It takes in as parameter the thread id, thread attribute, the thread function to execute and the argument to be passed to the thread function.

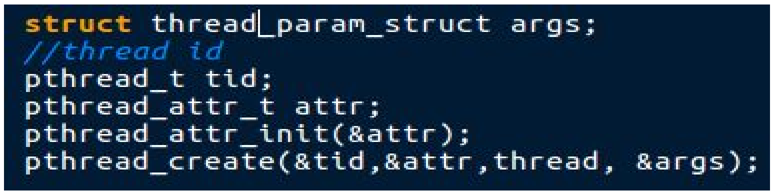
int pthread_create(pthread_t ****thread*, const pthread_attr_t ****attr***,**

void *(**start_routine***) (void *), void ****arg***)**

After the thread is created, further the thread attribute is initialized by using the function -

int pthread_attr_init(pthread_attr_t ****attr*)**

This function initializes the attribute object pointed to by the *attr. After this pthread_join(thread id) is used if it is required that the calling function should wait for the thread to complete its execution.

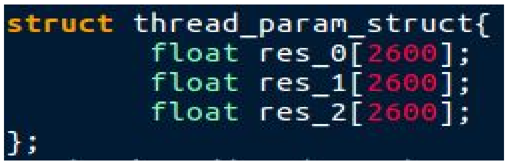


```
struct thread_param_struct args;
//thread id
pthread_t tid;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_create(&tid,&attr,thread, &args);
```

Once the thread is created and executed pthread_exit is used to pass any values from the thread function to the main function. Also, pthread _join is used to halt the main file execution until the thread is executed completely.The first step taken was to include the pthread library later a thread was created which included all the executables for the tracing algorithm.

Global Structure definition

In order to pass the values of constructed image among various threads and to preserve image structure, we define a global structure containing the arrays of res_0,res_1 and res_2. This structure is passed into all the threads, the executing thread modify the values of this array and exits the threads. Therefore all the variables initialised in the thread are popped as soon as the function ends, although the structure values are preserved.

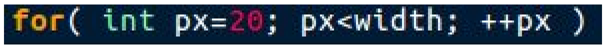


```
struct thread_param_struct{
        float res_0[2600];
        float res_1[2600];
        float res_2[2600];
};
```

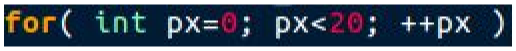
<u>A15-A9 Architecture Configuration :</u>

In the first iteration for A15-A9, we made a single parallel thread and ported all of the tracing algorithm in that thread. Then we run make clean and make for the X86 processor, to confirm that the program is running after modification and then make -f Makefile.arm for the ARM executable. Following are the results for trace loop splitting and relevant ticks data.

The above graph represent the splitting of loops of the tracing code. The tracing code has in total of 30 outer loop iteration with the counter running from “px =0 ” to “px<width”.

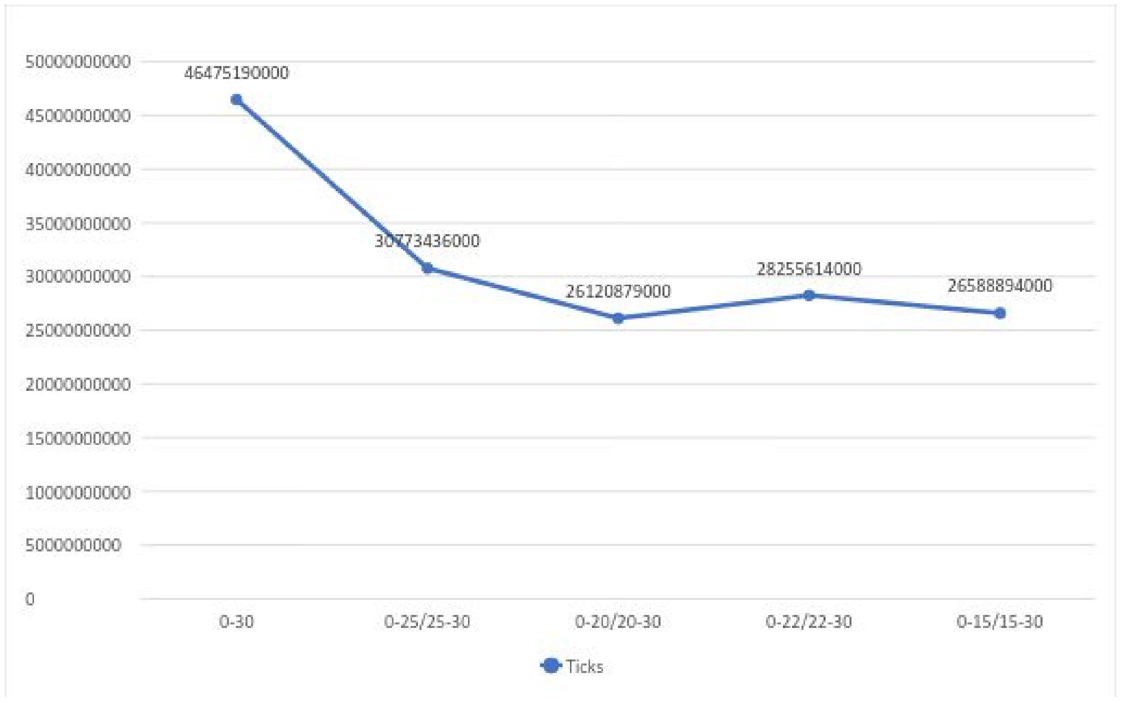


```
for( int px=20; px<width; ++px )
```



```
for( int px=0; px<20; ++px )
```

In our first iteration, we executed the code on two threads(one for main and the other thread). On executing, the code, we got the same number of ticks as for the “naive” code. This was because in this iteration, we were creating the thread using pthread_create and then declaring the pthread_join for the thread.



Hence, the calling function i.e, main, was calling the thread and was waiting for the thread to complete. Therefore, in spite of having two threads, the code was running on single core. We cross-checked our assumption using the single core A15.py architecture file, and the number of ticks was same as it was for A15-A9 configuration. In our next configuration, we moved the pthread_join function after the execution of the entire for-loop. Hence, both the main and thread t1 were executing the entire for loop parallely and after execution, main function waits for the thread to complete its execution before writing the res_0,res_1,res_2 to the “res.ppm” file. Hence, the entire for-loop was ported successfully to the two core. In dividing for loop iteration between the two threads, we ran few simulation to find the optimum load for both the threads. This is because both the threads were running on two cores with different pipeline stages. The iterations and its result in number of ticks is shown in the following table.

S.no.	For Loop structure	Number of Ticks
1.	Entire loop on single thread	46475190000
2.	main(px=0 to 19) , thread(px=20 to 30)	26120879000
3.	main(px=0 to 14) , thread(px=15 to 30)	26588894000
4.	main(px=0 to 24) , thread(px=25 to 30)	30773436000
5.	main(px=0 to 21) , thread(px=22 to 30)	28255614000

We have split those loops into different proportions and implemented on threads, to enforce parallel processing. From the table, we can observe that the optimum value for the loop division was achieved in iteration 2. Hence, the naive code was successfully ported on the two threads with loop division. The loop division ratio for optimum results is justified by considering the pipeline stage ratio for both the cores.

A9-A9-A9 Architecture Configuration :

Code modification for the A9-A9-A9 configuration, included dividing the code into in total of three threads including the main function. Thereby we divided the for loop, from 0-15 for the first thread and 15-30 for the final thread. This enabled us to optimise the code to work in all three cores parallely. The main function was responsible for only collecting all the data from various threads and storing the file in .ppm. This structure resulted in a very convincing result of 27677093000. We also tried dividing the loops by including the main function in trace processing. Thereby we executed 0-10 in main loop itself, 10-20 in thread1 and 20-30 in final thread, but we were disappointed by the result as it resulted in 28329661000 which is an increase of 652,568,000 ticks.

We can infer from this experiment that when we were running the trace algorithm in the thread function and using main function for calling the threads and storing the .ppm file we were optimising the load on all 3 processors therefore we were getting a good execution time, however when we put a part of trace algorithm in the main function, it resulted in an increase in the computation time because of workload was not balanced for threads.

S.no.	For loop Structure	Number of Ticks
1	main(px=0 to 10), thread1(px=10 to 20), thread2(px=20 to 30)	28329661000
2	main, thread1(px=0 to 15), thread2(px=15 to 30)	27677093000

Results

19.07% improvement was achieved in computation for the A15-A9 configuration and 40.99% improvement was achieved for the A9-A9-A9 configuration.

Acknowledgment

1) https://www.youtube.com/watch?v=ynCc-v0K-do&t=406s
2) “Pthreads Programming” by B. Nichols, D.Buttlar and J. P. Farrell.