C Programming and Performance Engineering

Lab Work - OpenMP Parallel Programming (with TAU and PAPI)

November 28, 2017

Spyridoula Chrysikopoulou-Soldatou and Jeremy Williams

Alteration of the code-OpenMP

As a first attempt, a pragma omp parallel for was inserted inside the laplace_step function, in order to parallelize the "for loop". After compiling and running the program, no important improvement in the execution time was observed. This was a result of the creation of the threads in the beginning of the loop, and the destruction of them in the end of the loop. Because this creation and destruction happens multiple times, affects negatively the execution time of the program.

In order to solve this problem, a "pragma omp parallel" was put inside the main function, before the while loop. With this manipulation, the threads have already been created before the laplace_step function. The "pragma omp parallel for" in the laplace_step was replaced by a "pragma omp for".

To be noted that the initiation stage function which contains a "for loop" was not parallelized, as its execution time is not important comparing to the laplace_step one.

Profiling

After the installation of TAU for OpenMp and OpenMP+PAPI, we generated the profiles for 2, 4 and 8 threads. We visualized the text summary with the command pprof and the graphic information about the execution of the application with the command paraprof.

Following are presented the results provided by pprof and paraprof:

2 Threads

Figure 1: Results of the paraprof command for 2 threads

Each color represents an event executing on one or more threads.

The blue color represents the time executing the body of the “for loop”, while the red color the time executing the barrier enter/exit of the “for loop”. The black colour is the memory addresses: addr=<0x419380> [{(unknown)} {0,0}] [THROTTLED] and the addr=<0x4193f0> [{(unknown)} {0,0}] [THROTTLED].

As we can observe, the most time in both threads was spent in the execution of the “for loop”. This result was expected, as the laplace_step nested loop is where the most work takes place. The barrier enter/exit of the “for loop”, refers to the time required to enter the “for construct” and to the time required to leave the construct.

4 Threads

Figure 2: Results of the paraprof command for 4 threads

When compiling the program with 4 threads, the percentage time spent in the body of the “for loop” is decreasing, while the time spent in thebarrier enter/exit of the “for loop” is increasing. That is logical, as with more threads, the work to be done (the “for body”) is distributed between them, and each one has less “for loops” to execute. On the other hand, the barrier enter/exit time of the “for loop” is also smaller, but occupies a bigger percentage of the total time.

8 Threads

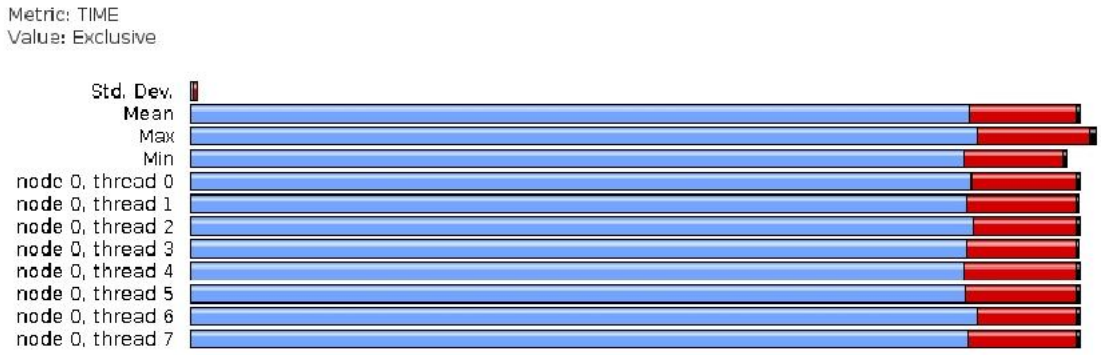

Figure 3: Results of the paraprof command for 8 threads

While compiling the program with 8 threads, the percentage time spent in the body of the “for loop” of each thread is decreasing more, while the time spent in the barrier enter/exit of the “for loop” is also decreasing, as each thread needs to be inserted into the loop less times. The explanation for the above figure is the same as

before. As the work to be done is distributed between the threads, the percentage time spent in the execution of the “for loop” is decreasing.

Tracing

For a more detailed analysis of parallel program behavior, the TAU tracing was activated, in order to analyze the results searching for the behavioral patterns. Tracing shows when the events take place in each process along a timeline.

The application was once again compiled and executed for 2, 4 and 8 threads. In order to visualize the generated trace files, jumpshot was used. The results are being presented bellow.

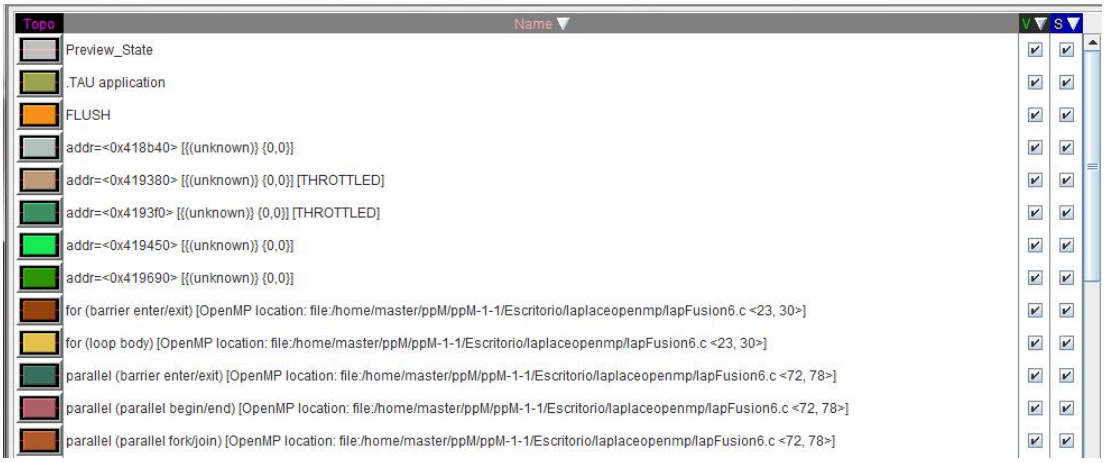

Figure 4: Names and respective colors of the events

2 Threads

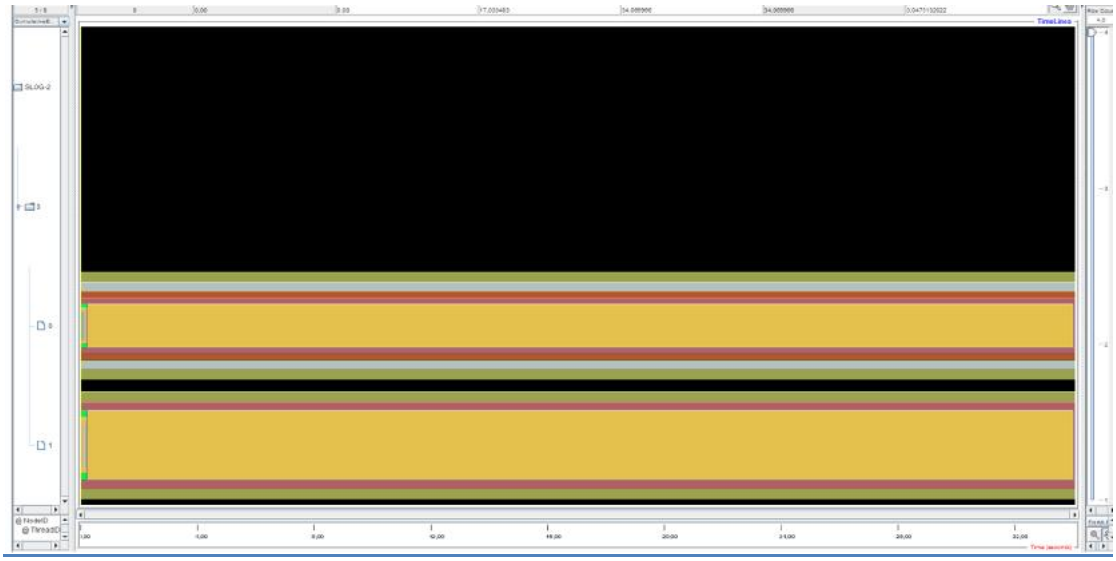

Figure 5: Result of the jumpshot command for 2 threads

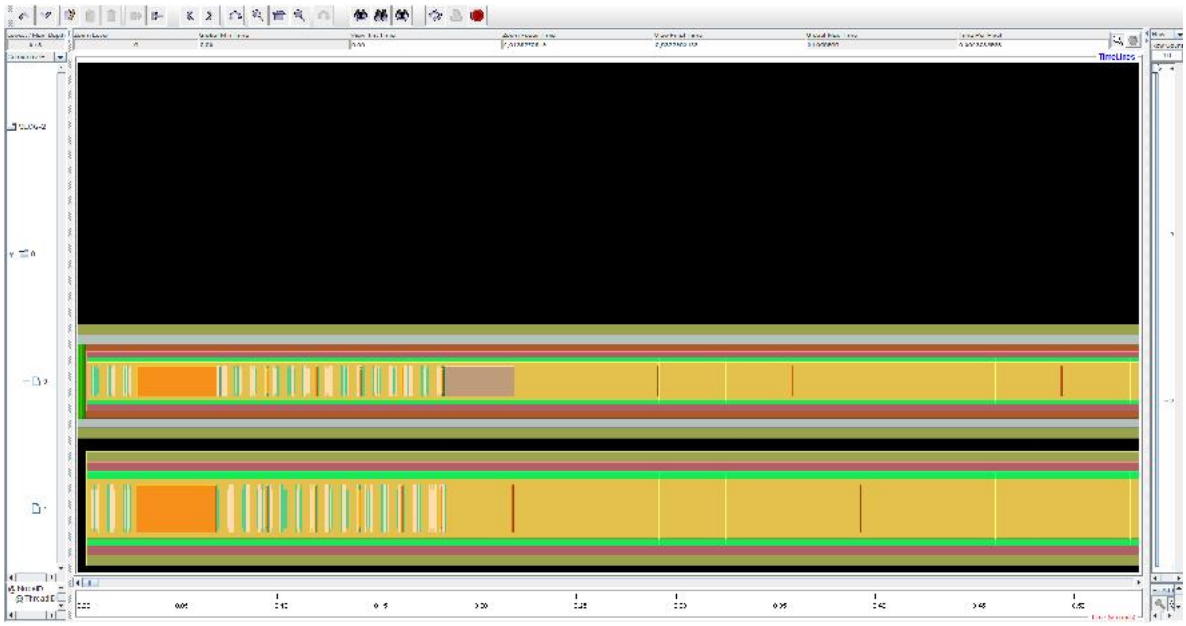

Figure 6: Result of the jumpshot command for 2 threads in the first 0.50 seconds

As we can see, with the beginning of the execution, the program access a specific memory address. The second thread was created approximately after 0,005 sec from when the program started. After the creation of the thread, the access to the memory addr=<0x419450>[{(unknown)} {0,0}] and the "parallel begin/end" ran continuously until the end. This is logical, as we have parallelized our program.

The parallel fork/join ran also continuously, but only for the initial loop, as this is the one that creates the parallelism.

The TAU application ran also through all the program's duration for both threads.
Until the 0,22 seconds, the following pattern is observed if we zoom to the part with the multiple lines.

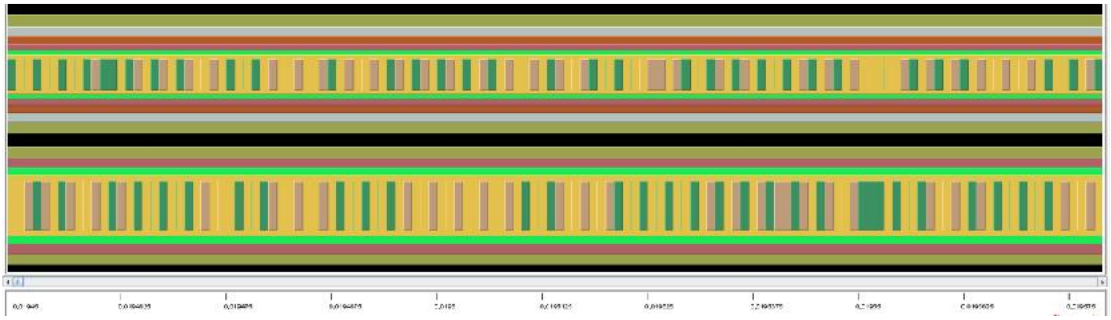

Figure 7: Zoomed result of the jumpshot command for 2 threads

These colors correspond to events with names addr=<0x4193f0>[{(unknown)} {0,0}] [THROTTLED] and addr=0x419380>[{(unknown)} {0,0}] [THROTTLED]. This perhaps signifies that the program access the specific memory addresses.
After the 0,22 seconds, the following pattern continues until the end:

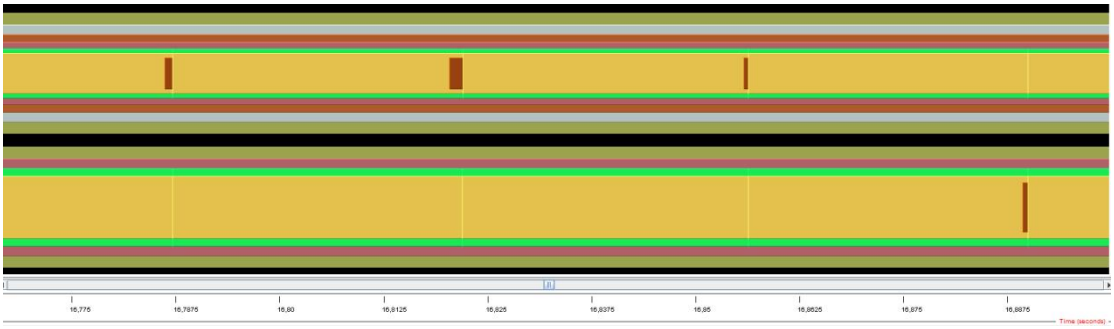

Figure 8: Zoomed result of the jumpshot command for 2 threads

As we saw, both threads execute the “for loop”, alternating with the "for barrier enter/exit". This is logical, as after computing the “for loop” once, the loop ends and a new one starts.

4 Threads

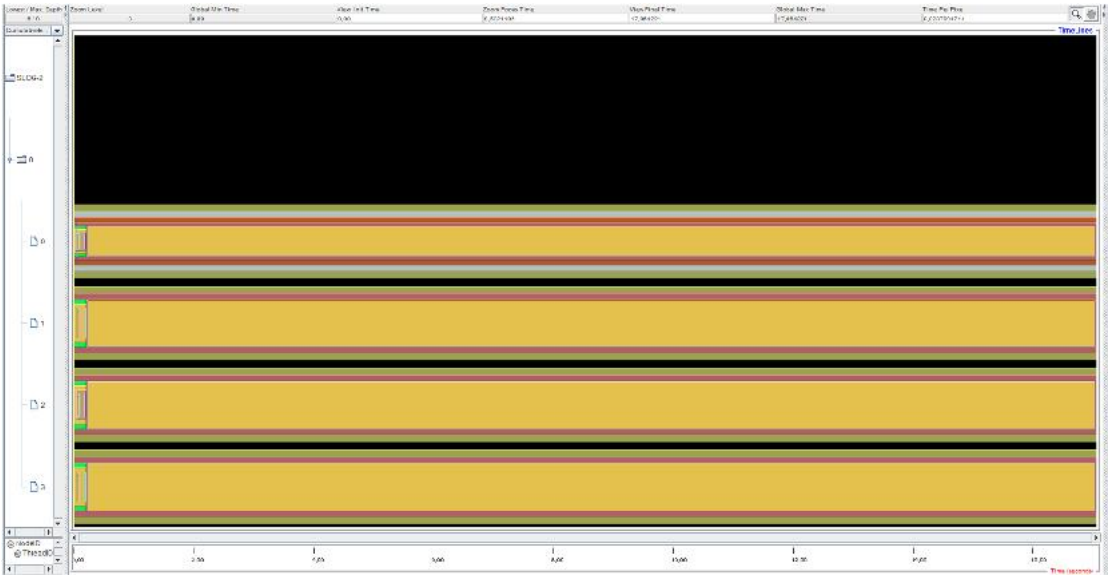

Figure 9: Results of the jumpshot command for 4 threads

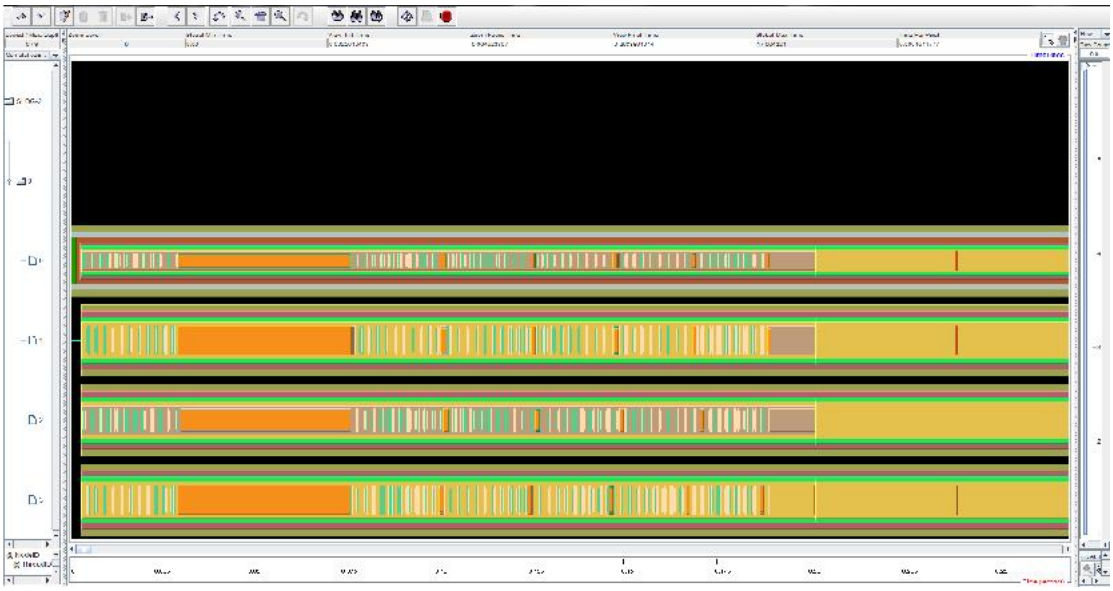

Figure 10: Results of the jumpshot command for 4 threads in the first 0,25 seconds

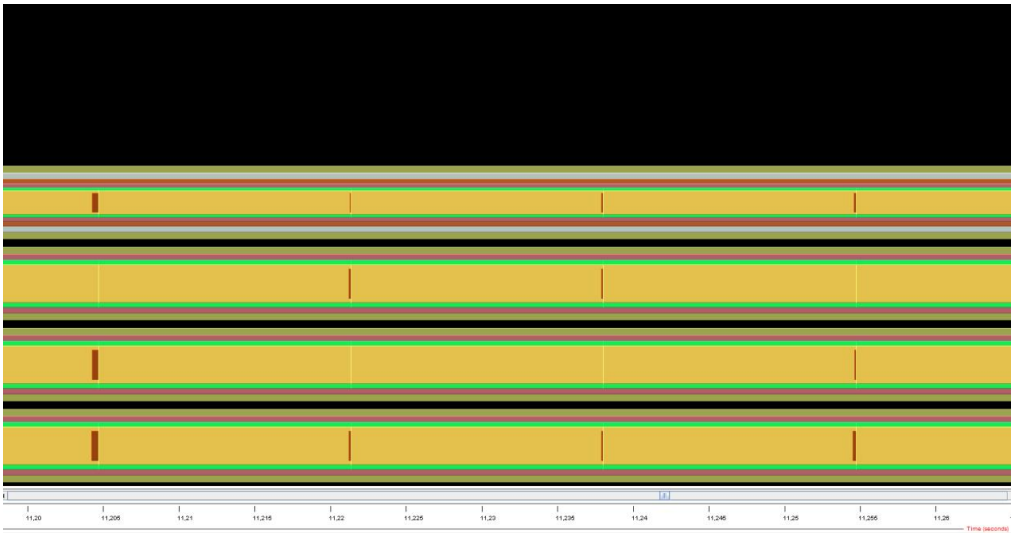

Figure 11: Zoomed result of the jumpshot command for 4 threads after 0,25 seconds

8 Threads

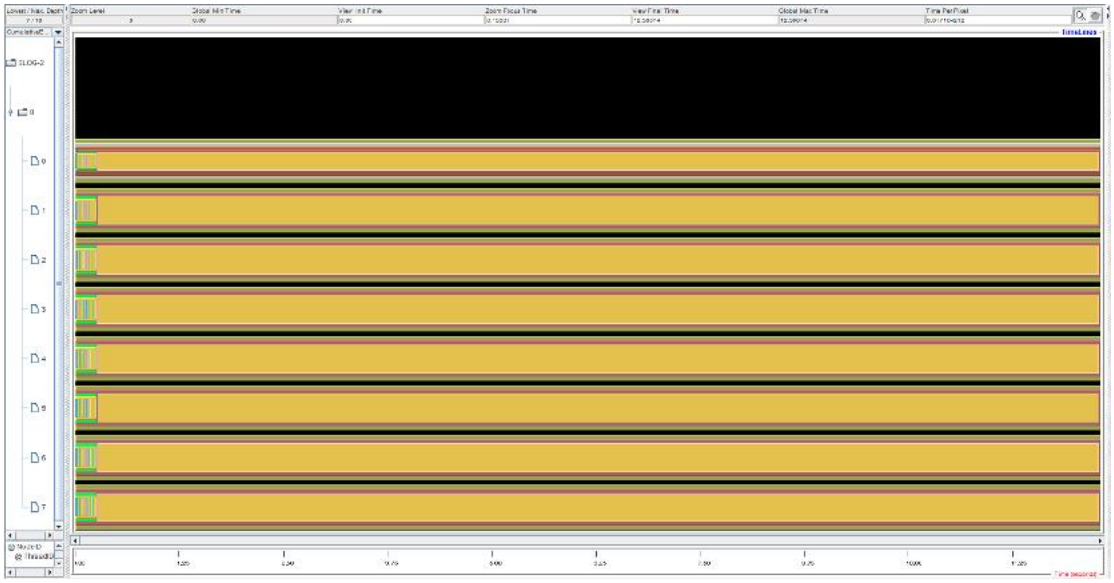

Figure 12: Results of the jumpshot command for 8 threads

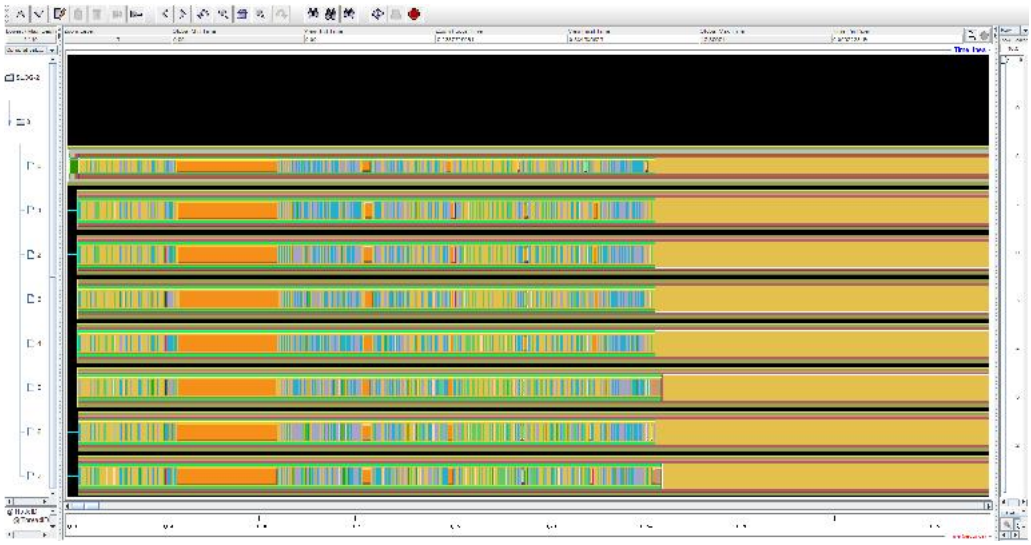

Figure 13: Results of the jumpshot command for 8 threads in the first 0,36 seconds

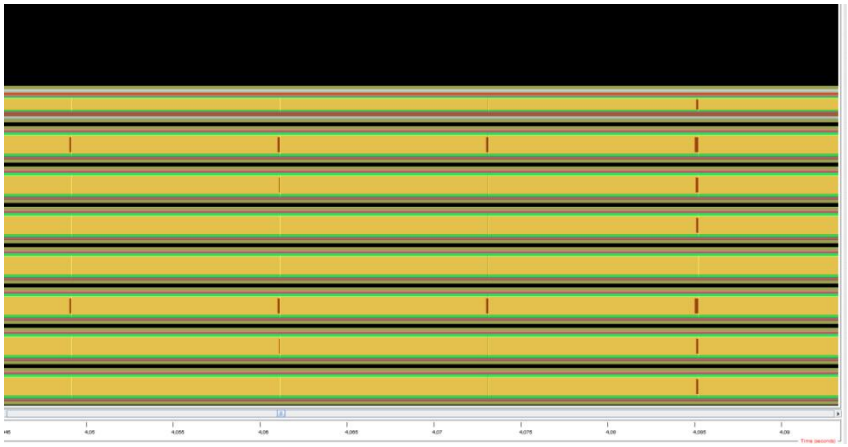

Figure 14: Zoomed results of the jumpshot command for 8 threads after the 0,25 seconds

As we saw in general, the patterns when running the application with 4 and 8 threads are similar when running it with 2 threads.

In the beginning of the execution, the programs access a specific memory address (addr=<0x419690). Afterwards, the threads were created and a continuous pattern, which contains alternative access to two memory addresses occurred. Later, from a point and afterwards we observe a different pattern; where the execution of the “for loop” is alternating with the "for enter/exit". In all the cases (2, 4 and 8 threads) this begins at approximately 0,25 seconds.

From these, we understand that in the initialization part many events took place, even though the duration of it was very small.

Correctly, we have also had in all thread cases, the "parallel begin/end" event ran continuously from when the parallelization occurred until the end. Then the parallel fork/join also ran continuously, but only for the initial loop, as this is the one that creates the parallelism.

Finally, it was interesting for us to see, that the threads were not being created simultaneously. However, they were being destroyed simultaneously.

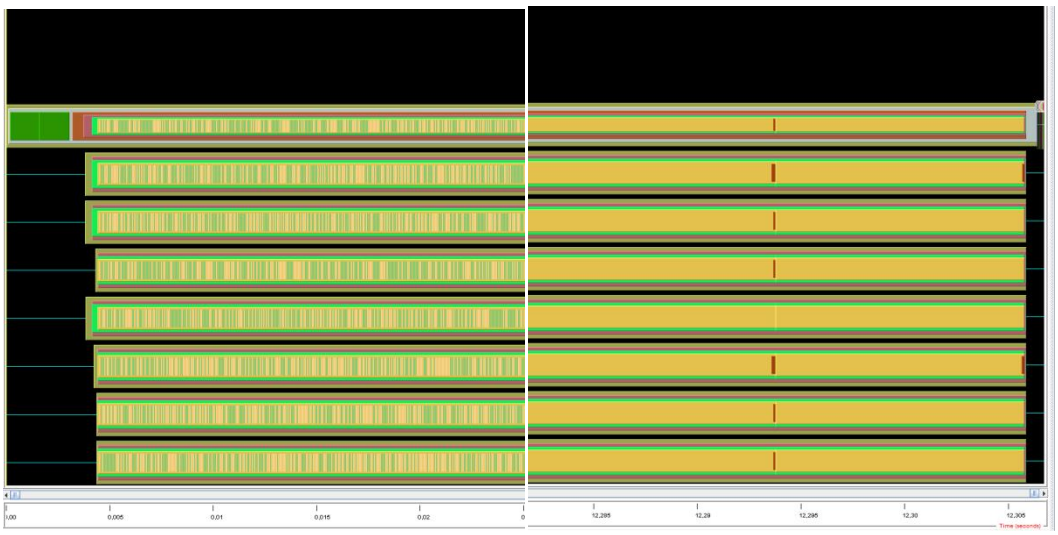

Figure 15: Beginning and ending of program (8 threads)

Profiling with PAPI counters

Next, we activated the profiling with some counters. We chose to execute the application with 8 threads. The results are being presented below:

PAPI_L3_TCM

The event measured is the Level 3 total cache misses.

With the paraprof command, we have the following results:

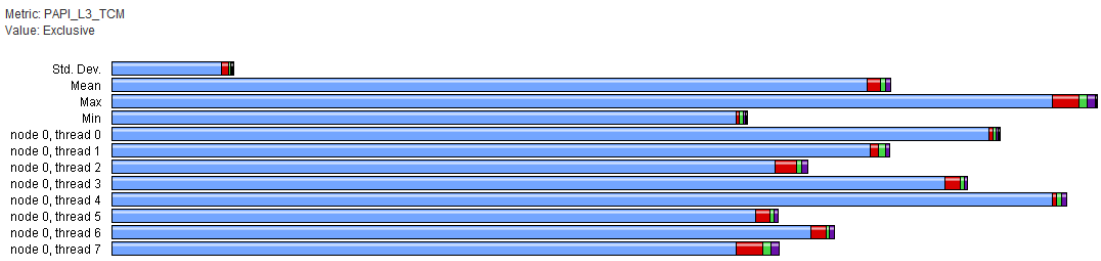

Figure 16: Level 3 total cache misses for each thread

Where each color represents:

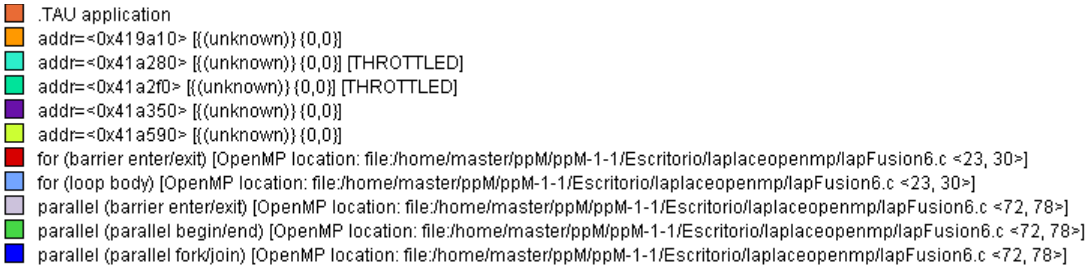

Figure 17: Color of each event

Below we see the counts of total level 3 cache misses for the threads 0 and 7:

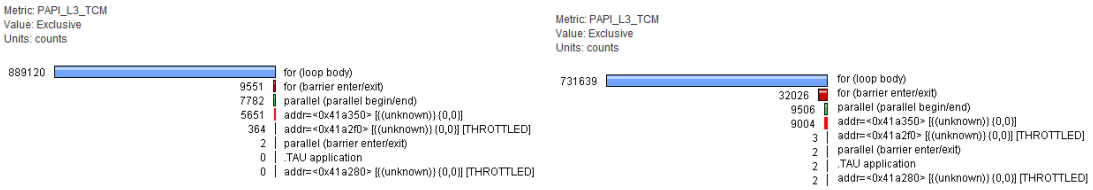

Figure 18: Counts of L3 total cache misses for threads 0 and 7

As we see, Level 3 total cache misses occur to a great extent on the body of the “for loop”. The next biggest percentages are in the “for barrier” enter/exit and in the parallel begin/end.

PAPI_TOT_CYC

The event measured is the total cycles.

With the paraprof command, we have the following results:

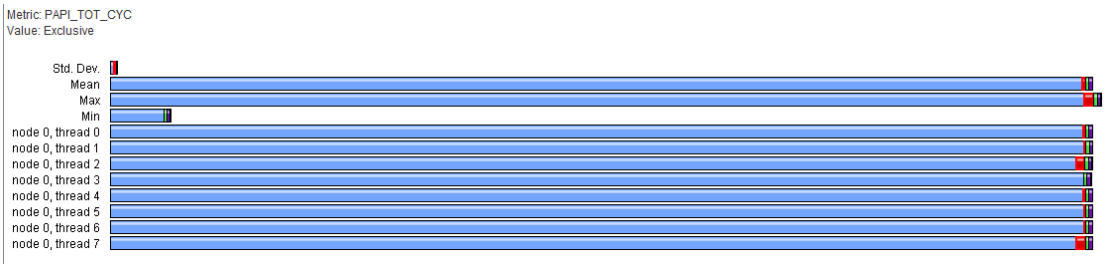

Figure 19: Total circle for each thread

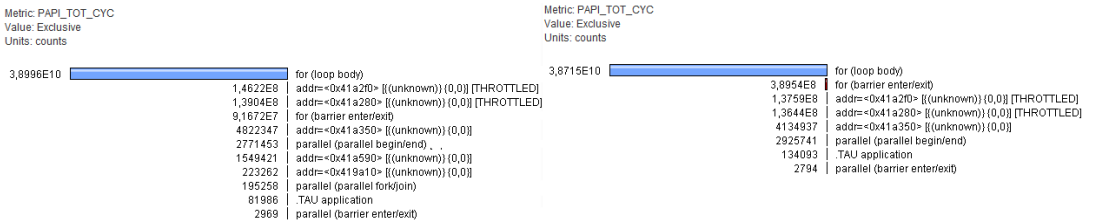

Figure 20: Counts of total circles for threads 0 and 7

Once again the majority of circles occur in the “for loop” body.

References

1. OpenMP Shared Memory Parallel Programming: International Workshop, IWOMP 2005 and IWOMP 2006, Matthias S. Müller, Barbara Chapman, Bronis R. de Supinski, Allen D. Malony, Michael Voss
2. https://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01s04.html