Universitat Autònoma de Barcelona

Parallel Programming: OpenMP

Assignment 2

Alfredo Hernández Alejandro Jiménez

28th November

Contents

1 Introduction

In this assignment we aim to improve the execution time of the now-common Laplace 2D code and measure the impact of using of OpenMP.

In order to make the biggest difference on the code, we chose the most expensive loop (in terms of computational cost) and created multiple threads using OpenMP, which would split the iterations into different, parallel, more efficient processes.

We will measure the impact of the parallelisation of the code using `perf stat` and more sophisticated analytical tools such as *TAU*. Using *TAU* we will try to identify and analyse the different internal operations of the code and how their performance varies when using a different number of threads.

2 Analysis

2.1 Setting up the system before working

Before doing anything, we need to load the `modules` of the compiler and libraries we have to use:

```
module load gcc/6.1.0
module load papi/5.4.3
module load openmpi/1.8.1
```

After installing TAU and compiling the different `Makefile`s, we need to make sure we include the installation directory on the `$PATH`, this can be done on each session (or by modifying the `.bash_profile`:

```
export PATH=/home/master/ppM/ppM-1-10/my_TAU/x86_64/bin:$PATH
```

2.2 Parallelisation with OpenMP

One of the main variables to consider when improving the performance of a code using OpenMP is the number of threads that we use. Specifying the exact number of threads to be used can be done easily using this command on our source C code:

```
#pragma omp parallel for num_threads(NUM_THREADS)
```

This can improve greatly the performance of the code when used on sequential portions of the program, such as loops where each step is independent from the others.

In snippet 1 we can see how we've implemented the **`#pragma`** to select the amount of threads we want to use in our program:

Snippet 1: Example of parallelisation on nested loops using OpenMP

```
float laplace_step(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  #pragma omp parallel for num_threads(NUM_THREADS)
  for ( j=1; j < n-1; j++ )
    #pragma omp simd reduction(max:error)
    for ( i=1; i < n-1; i++ )
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i],
          in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  return error;
}
```

Where `NUM_THREADS` is an integer that depends on the architecture of the machine that we are using. In our case, we used 2, 4, and 8 threads.

To compile the code we just use vanilla `gcc` with the appropriate flags to enable OpenMP:

```
gcc -lm -fopenmp -o test_Np lapFusion.c
```

where `N` is 2, 4, or 8.

For this first measure of performance, we have used the widely known tool `perf stat` on a code that computes the Laplace 2D equation on a 512×512 matrix with 500 iterations:

```
perf stat ./test_Np 512 500
```

The results shown in table 1 reflect an improvement of the performance when using an increasing amount of threads, as expected.

Num. Threads	**Run Time**
No parallel	4.841 s
2 Threads	2.521 s
4 Threads	1.685 s
8 Threads	0.948 s

Table 1: Run time obtained on multiple threads

It is crucial to notice that the more threads we use the more efficient the computation becomes, and by a large difference; this fact reflects the great importance of parallelisation and its implications on high-performance computation.

2.3 Analysis with TAU

Up to this point we have seen a superficial picture of how the performance increases using OpenMP as a tool to parallelise the code. Now we are going to use *TAU*, a toolkit for analysis performance that allows us to study in detail the performance of many parts of the code that we have parallelised.

As a comparison, we have used the same code as before with the same number of iterations and problem size.

2.3.1 Using profile files (`paraprof`)

The toolkit `paraprof` (bundled with TAU) allows us to parallel profile analysis. This is similar to using `perf stat`, but now we have data for each of the threads.

For this part, we need to set the following environment variables:

```
export TAU_MAKEFILE=$HOME/my_TAU/x86_64/lib/Makefile.tau-openmp-opari
export TAU_OPTIONS=-optCompInst
export TAU_PROFILE=1
```

Then we have to compile our source code using TAU, and then run it as usual

```
tau_cc.sh -lm -o openmp.test_Np_trace lapFusionN.c
./openmp.test_Np_trace 512 500
```

where `N` can be 2, 4, or 8.

This will give us `profile.0.0.*` profile files; one for each of the threads used. To read and analyse these files we can use the `paraprof` tool.

Even though we have run the same code, the first thing we notice here is an increase on the *Run Time*. This happens because now we are asking for an on-time measurement of the performance of the code on internal steps; this is more computationally demanding than a simple `perf stat` and thus it takes more time to compute it.

In figure 1 we can see the typical 3D Visualisation of the `TIME` metric using `paraprof`. The height of the bars represents the metric, whilst on the Y axis we have the different functions of the compiled code. We will focus on the time metric of the `parallelfor` functions.

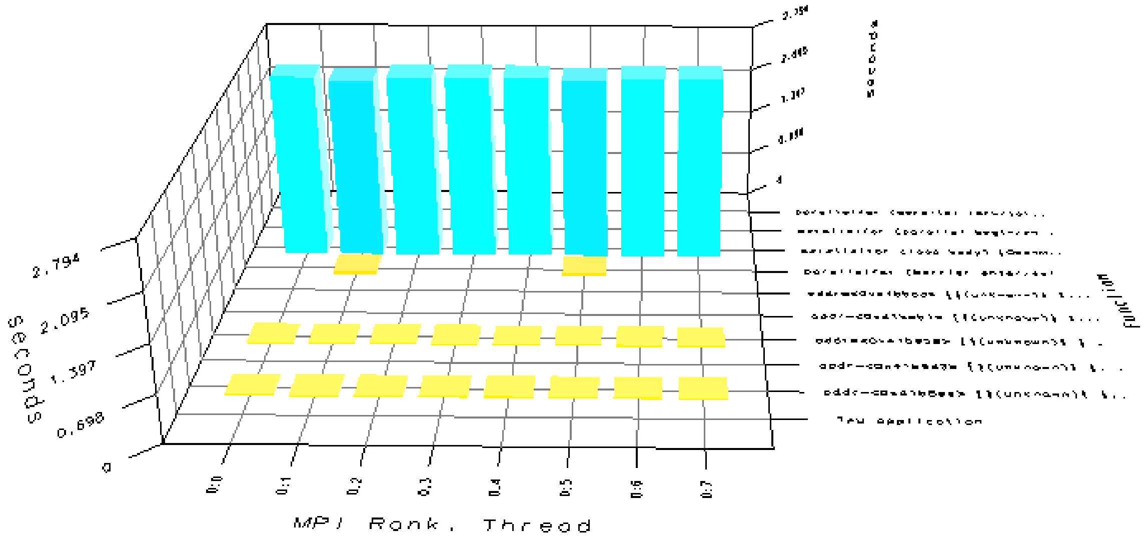

Figure 1: 3D Visualisation of the run time on 8 threads

Function	**2 Threads**	**4 Threads**	**8 Threads**
`parallelfor (barrier enter/exit)`	0.002 s	0.007 s	0.016 s
`parallelfor (loop body)`	6.684 s	5.65 s	2.788 s
`parallelfor (parallel begin/end)`	0.000 22 s	0.000 33 s	0.000 42 s
`parallelfor (parallel fork/join)`	0.001 s	0.033 s	0.002 s

Table 2: Run time obtained on multiple threads

This table shows interesting results. On one hand we can see how the *Run Time* of execution of some internal operations increases; such as `barrier enter/exit`, `parallel begin/end`, and `parallel fork/join`. On the other hand we have an outstanding improvement of the performance of the main loop, the `loop body` part; and this is important, for this is the more demanding part of the code and its performance improvement largely compensates the loss of speed on the others.

2.3.2 Using traces (jumpshot)

Now that we have seen some of the standard measurement tools for parallelised code, we can explore even more detailed analytical tools.

For this part, we need to set the following environment variables:

```
export TAU_MAKEFILE=$HOME/my_TAU/x86_64/lib/Makefile.tau-openmp-opari
export TAU_OPTIONS=-optCompInst
export TAU_TRACE=1
```

Then we have to compile our source code using TAU, and then run it as usual

```
tau_cc.sh -lm -o openmp.test_Np_trace lapFusionN.c
./openmp.test_Np_trace 512 500
```

where `N` can be 2, 4, or 8.

After doing this, we have to merge the obtained `tautrace.0.0.*.trc` trace files into a readable `.slog2` log file using the following commands:

```
tau_treemerge.pl
tau2slog2 tau.trc tau.edf -o tauNthreads.slog2
```

To visualise the data of the logfile, we simply use the `jumpshot` tool (also bundled with TAU):

```
jumpshot tauNthreads.slog2
```

In tables 3, 4, and 5 we can see the results of the `Category count`, `Inclusive ratio`, and `Exclusive ratio` obtained from the `.slog2` log files for 2, 4, and 8 threads respectively.

We have selected to show only `FLUSH` and the parallel operations of the program, to ease the analysis of the data. The `FLUSH` operation deals with the different buffers of the program (temporary memory areas in which data is stored while it is being processed or transferred).

Legend	Function	count	incl	excl
🟪	FLUSH	12	0.014	0.014
🟥	parallelfor (barrier enter/exit)	1000	0.002	0.002
🟦	parallelfor (loop body)	1000	1.999	1.898
🟧	parallelfor (parallel begin/end)	1000	1.999	0
🟩	parallelfor (parallel fork/join)	500	1	0

Table 3: Data obtained from the logfile, using TAU with 2 threads

Legend	Function	count	incl	excl
🟪	FLUSH	24	0.048	0.048
🟥	parallelfor (barrier enter/exit)	2000	0.345	0.345
🟦	parallelfor (loop body)	2000	3.573	2.887
🟧	parallelfor (parallel begin/end)	2000	3.576	0.003
🟩	parallelfor (parallel fork/join)	500	1	0.058

Table 4: Data obtained from the logfile, using TAU with 4 threads

Legend	Function	count	incl	excl
🟪	FLUSH	48	0.97	0.97
🟥	parallelfor (barrier enter/exit)	4000	2.632	2.632
🟦	parallelfor (loop body)	4000	6.328	2.012
🟧	parallelfor (parallel begin/end)	4000	6.339	0.011
🟩	parallelfor (parallel fork/join)	500	1	0.222

Table 5: Data obtained from the logfile, using TAU with 8 threads

In figure 2 we can see an example of the Time Line of internal operations of each one of the used threads. Essentially, it reinforces the conclusions extracted from the previous section where we used `paraprof` but it enables us to visualise the stats in more detail how each of the threads behaves by itself and in relation to the others.

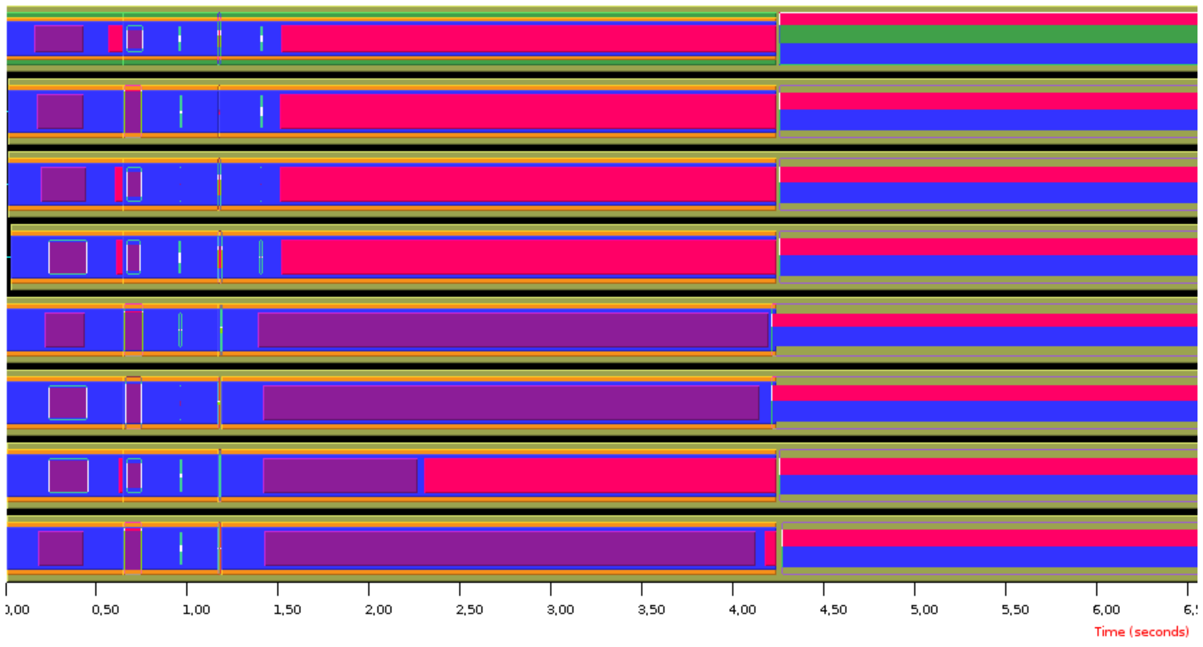

Figure 2: Time Line of the 8 threads visualised using an Identity Map

It is interesting to notice how Thread 0 is the one responsible to join the different threads after the parallel part of the program (initialisation of the matrix; see snippet 1) is finished.

3 Conclusions

During the resolution of this exercise, we have had the opportunity to further improve the code of the first exercise of the course. This time, though, we have used a different methodology to paralellise the code, the widely employed *OpenMP*. Moreover, we have got started in the analytical tool kits known as *TAU* and we have learnt the basics to understand and interpret different visalisation techniques for performance measurement, profiling, and tracing on parallelised code.

All of this has let us understand the importance of parallel programming when aiming for high performance computation, for we have experienced the increase of performance in our code when applying such techniques.