OPENMP/TAU - Performance Analysis

Contents

1 Introduction

1.1 Modification of code for parallelization

We use the following `OPENMP` commands to parallelize our code. These especially refers to the double for-loop performing the `laplace_step`, that is, the most time-consuming part of all the program.

```
...

float laplace_step(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
    #pragma omp for nowait
    for ( j=1; j < n-1; j++ )
      for ( i=1; i < n-1; i++ )
      {
        ...
      }
  return error;
}
...

int main(int argc, char** argv)
{
  ...
  omp_set_num_threads(2);//4,8
  int iter = 0;
  #pragma omp parallel default(none) firstprivate(iter,temp,A,error)
     shared(n, iter_max)
  {
    while (error > tol*tol && iter < iter_max )
    {
      ...
    }

    #pragma omp master
    {
      error = sqrtf( error );
      printf("Total Iterations: %5d, ERROR: %0.6f, ", iter, error);
      printf("A[%d][%d]= %0.6f\n", n/128, n/128, A[(n/128)*n+n/128]);
    }
  }
  ...
}
```

In the `int main`, we introduce a OMP section with declaring private the parameters `iter`, `temp`, `A` and `error`, since each thread handles own copies of these variables. On the other hand, the constant global variables `n` and `iter_max` can be shared among all the threads. In the `laplace_step` function, we use the `pragma_omp_for_nowait` command, which branches the process to perform the double for-loop.

2 Results

2.1 Time consumption

2.1.1 Serial

We use the original code for the Laplace application provided and get the results of the `perf stat` for serial execution with the `gcc` compiler and with a mesh of $n = 1000$:

```
Total Iterations:  1000, ERROR: 0.015535, A[7][7]= 0.016615

 Performance counter stats for './lap_2 1000':

      31250.063048      task-clock (msec)         #    0.998 CPUs utilized
   104,364,169,322      cycles                    #    3.340 GHz
   115,660,649,729      instructions              #    1.11  insn per cycle

      31.308894954 seconds time elapsed
```

For all the following, we execute the code with the `tau_cc.sh` compiler and a mesh of $n = 1000$. The results are obtained from the `perf` tool and the TAU `pprof` command.

2.1.2 2 threads

```
Jacobi relaxation Calculation: 1000 x 1000 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.015535, A[7][7]= 0.016615

 Performance counter stats for './lap_2_2 1000':

     112963,788825      task-clock (msec)         #    1,987 CPUs utilized
   371.475.376.651      cycles                    #    3,288 GHz
   706.115.703.767      instructions              #    1,90  insns per cycle

      56,845403345 seconds time elapsed


FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           7       56,826           1           1   56826868 .TAU application
100.0       0.798       56,817           1      1001.5   56817807 parallel (parallel begin/end)
 99.7           1       56,684        1000        1000      56684 laplace_step
 99.7      56,612       56,683        1000      199944      56683 for (loop body)
 50.0       0.286       28,410         0.5         1.5   56821454 main
 50.0       0.159       28,409         0.5         0.5   56818177 parallel (parallel fork/join)
  0.2         132          132           1           0     132790 parallel (barrier enter/exit)
  0.1          47           47       99973           0          0 max_error [THROTTLED]
  0.0          23           23     99971.5           0          0 stencil [THROTTLED]
  0.0           1            1           1           0       1352 laplace_init
  0.0      0.0205       0.0205         0.5           0         41 master (master begin/end)
```

The execution times increases when we use the `tau_cc.sh` compiler. Remarkable is the fact, that it can be seen a strong increase of cycles and instructions (approx. by the factor 7).

2.1.3 4 threads

```
Jacobi relaxation Calculation: 1000 x 1000 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.012964, A[7][7]= 0.016615

 Performance counter stats for './lap_2_tau 1000':

     116704,631422      task-clock (msec)         #    3,957 CPUs utilized
   372.373.716.617      cycles                    #    3,191 GHz
   706.643.925.605      instructions              #    1,90  insns per cycle


      29,494571057 seconds time elapsed
```

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           3       29,366           1           1   29366954 .TAU application
100.0        0.49       29,362           1     1001.25   29362354 parallel (parallel begin/end)
 99.4       0.812       29,187        1000        1000      29188 laplace_step
 99.4      29,133       29,186        1000      197776      29187 for (loop body)
 25.0       0.172        7,341        0.25        0.75   29366667 main
 25.0       0.118        7,340        0.25        0.25   29362843 parallel (parallel fork/join)
  0.6         174          174           1           0     174124 parallel (barrier enter/exit)
  0.1          31           31       98889           0          0 max_error [THROTTLED]
  0.1          22           22     98887.2           0          0 stencil [THROTTLED]
  0.0       0.784        0.784         0.5           0       1568 laplace_init
  0.0      0.0138       0.0138        0.25           0         55 master (master begin/end)
```

Here we see a reduction of the time by the half. The major part of $74,9\%$ is taken by the double for-loop inside the `laplace_step` function. The cycles and instructions stay at the same level as before.

2.1.4 8 threads

For the case of 8 threads, we will take a closer view at the behaviour under change of problem size. As before, we start with a mesh size of $n = 1000$.
We compiled and executed the program with 8 threads via the remote access. In difference to the execution in the lab, we only have the option to use 4 CPUs. So it is reasonable that we obtain similar results (~ 29 sec) as in the case of 4 threads.
We expect that the execution time on 8 CPUs would approximately decrease by half.

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           2       29,188           1           1   29188116 .TAU application
100.0       0.428       29,184           1     1001.12   29184637 parallel (parallel begin/end)
 99.8       0.659       29,119        1000        1000      29119 laplace_step
 99.8      29,063       29,118        1000     96512.1      29118 for (loop body)
 12.5       0.078        3,649       0.125       0.375   29194622 main
 12.5      0.0839        3,648       0.125       0.125   29191336 parallel (parallel fork/join)
  0.2          65           65           1           0      65145 parallel (barrier enter/exit)
```

```
  0.1          29          29     48255.8          0          1 stencil [THROTTLED]
  0.1          25          25     48256.4          0          1 max_error [THROTTLED]
  0.0       0.333       0.333        0.25          0       1331 laplace_init
  0.0     0.00762     0.00762       0.125          0         61 master (master begin/end)
```

2.2 Problem size

The effect of the change of problem size on the performance of the program was monitored and the results are presented below. The program was compiled with 8 threads for this section.

```
Jacobi relaxation Calculation: 1000 x 1000 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.010200, A[7][7]= 0.016615

 Performance counter stats:

     116659,711407      task-clock (msec)         #    3,975 CPUs utilized
   372.143.041.826      cycles                    #    3,190 GHz
   706.639.778.898      instructions              #    1,90  insns per cycle

      29,351555976 seconds time elapsed


Jacobi relaxation Calculation: 500 x 500 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.009603, A[3][3]= 0.016854

 Performance counter stats:

      29402,502412      task-clock (msec)         #    3,712 CPUs utilized
    93.508.818.896      cycles                    #    3,180 GHz
   176.784.479.052      instructions              #    1,89  insns per cycle

       7,921265885 seconds time elapsed

Jacobi relaxation Calculation: 100 x 100 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.013125, A[0][0]= 1.000000

 Performance counter stats:

       1488,872256      task-clock (msec)         #    2,258 CPUs utilized
     4.360.606.365      cycles                    #    2,929 GHz
     7.885.220.503      instructions              #    1,81  insns per cycle

       0,659373830 seconds time elapsed
```

As expected, the number of cycles, instructions as well as the execution time decrease approximately with the square of the change of the problem size: Comparing the stats of $n = 1000$ and $n = 500$, all the parameters are 4 times as small in the latter case. Comparing $n = 500$ and $n = 100$, we obtain the same behaviour, except the time, which is more than expected. We suppose that reason for this is the overall short execution times below 1 second, which noises the time result. Especially there are processes like the establishment of the parallel region, that need to be done regardless which problem size is used.

2.3 Level 3 Cache Misses

For Level 3 (L3) Cache misses, we measure how many times an access call was made to the L3 cache and the data requested was not found, meaning it had to be fetched from the main memory, normally resulting in an increase latency. With `MULTI__PAPI_L3_TCM` as the activated `PAPI_METRIC`, we can obtain measurements of the Level 3 cache misses. Again we compare results in the case of 2,4 and 8 threads with a mesh size of $n = 1000$.

2.3.1 2 threads

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
              counts total counts
---------------------------------------------------------------------------------------
100.0        5455        8102           1           1       8102 .TAU application
 32.7        1190        2647           1           3       2647 main
 16.4         622        1328           1           1       1328 parallel (parallel fork/join)
  8.7         118         706           1        1002        706 parallel (parallel begin/end)
  4.3          42         352        1000        1000          0 laplace_step
  3.8         280         310        1000      200002          0 for (loop body)
  2.9         234         234           1           0        234 master (master begin/end)
  1.6         129         129           2           0         64 laplace_init
  0.2          19          19      100001           0          0 stencil [THROTTLED]
  0.1          11          11      100001           0          0 max_error [THROTTLED]
  0.0           2           2           1           0          2 parallel (barrier enter/exit)

NODE 0;CONTEXT 0;THREAD 1:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
              counts total counts
---------------------------------------------------------------------------------------
100.0         464         890           1        1001        890 parallel (parallel begin/end)
 95.8         -37         853           1           1        853 .TAU application
 25.8         230         230           1           0        230 parallel (barrier enter/exit)
 22.0          30         196        1000        1000          0 laplace_step
 18.7         150         166        1000      198046          0 for (loop body)
  1.0           9           9       99022           0          0 stencil [THROTTLED]
  0.8           7           7       99024           0          0 max_error [THROTTLED]


FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
              counts total counts
---------------------------------------------------------------------------------------
 99.6        2709        4478           1           1       4478 .TAU application
 29.4         595        1324         0.5         1.5       2647 main
 17.7         291         798           1      1001.5        798 parallel (parallel begin/end)
 14.8         311         664         0.5         0.5       1328 parallel (parallel fork/join)
  6.1          36         274        1000        1000          0 laplace_step
  5.3         215         238        1000      199024          0 for (loop body)
  2.6         117         117         0.5           0        234 master (master begin/end)
  2.6         116         116           1           0        116 parallel (barrier enter/exit)
  1.4        64.5        64.5           1           0         64 laplace_init
  0.3          14          14     99511.5           0          0 stencil [THROTTLED]
```

```
 0.2           9           9     99512.5           0           0 max_error [THROTTLED]
```

Figure 1: `paraprof` output for the Level 3 total cache misses count

Figure 2: `paraprof` legend of colours representing the different program parts

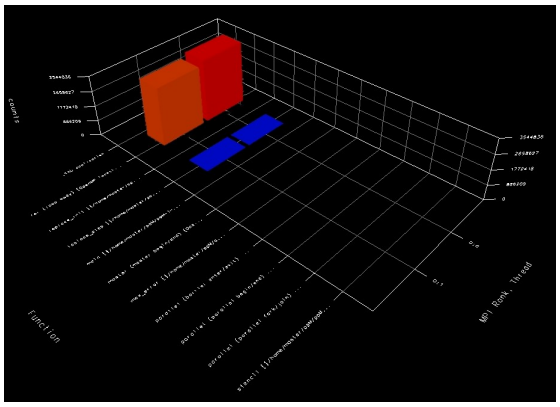


Figure 3: The `paraprof` output shows that the majority of cache misses happen during the for-loop, a small amount at the rest of the `laplace_step`

For the number of L3 cache misses we expected to obtain the highest numbers during the double loop in the `laplace_step`. However, viewing the `pprof` results, we can see that in both threads, the TAU application itself consumes the majority of cache misses (67.3% in thread 0, 70% in thread 1). In contrast to this, the for-loop, which was assumed to be most time consumptive, is only responsible for a small fraction of cache misses (1.1% and 17.7%). Seeing the output of `paraprof`, our expectations are confirmed.

2.3.2 4 threads

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
            counts total counts
---------------------------------------------------------------------------------------
100.0        1387   1.302E+06           1           1    1301916 .TAU application
 99.8        7316     1.3E+06           1     1001.25    1299769 parallel (parallel begin/end)
```

99.3	2.422E+04	1.292E+06	1000	1000	1292	laplace_step
97.4	1.268E+06	1.268E+06	1000	192937	1268	for (loop body)
25.1	460.2	3.263E+05	0.25	0.75	1305293	main
25.0	191.5	3.258E+05	0.25	0.25	1303019	parallel (parallel fork/join)
0.0	108.2	108.2	0.5	0	216	laplace_init
0.0	100	100	1	0	100	parallel (barrier enter/exit)
0.0	98.25	98.25	0.25	0	393	master (master begin/end)
0.0	7	7	96468	0	0	stencil [THROTTLED]
0.0	6.5	6.5	96469.2	0	0	max_error [THROTTLED]

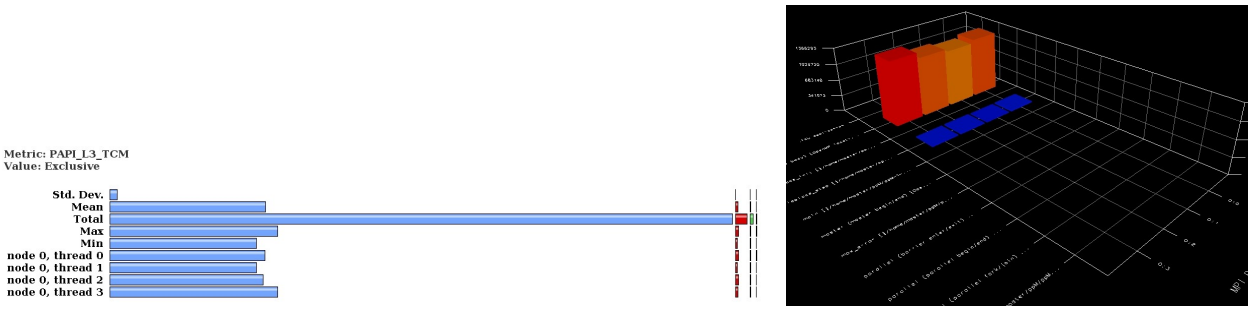


Figure 4: exclusive level 3 cache misses count

2.3.3 8 threads

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
              counts total counts
---------------------------------------------------------------------------------------
100.0       752.2   6.267E+05           1           1     626672 .TAU application
 99.8        4846   6.255E+05           1     1001.12     625518 parallel (parallel begin/end)
 99.0   1.614E+04   6.206E+05        1000        1000        621 laplace_step
 96.4   6.042E+05   6.044E+05        1000      199216        604 for (loop body)
 13.7       231.2   8.608E+04       0.125       0.375     688645 main
 13.7       111.6   8.579E+04       0.125       0.125     686330 parallel (parallel fork/join)
  0.0       191.9       191.9     99608.5           0          0 max_error [THROTTLED]
  0.0       67.75       67.75           1           0         68 parallel (barrier enter/exit)
  0.0       58.12       58.12        0.25           0        232 laplace_init
  0.0       45.12       45.12       0.125           0        361 master (master begin/end)
  0.0        4.75        4.75     99607.5           0          0 stencil [THROTTLED]
```

Executing with 8 threads, we observe that in mean, the total inclusive counts increases from 4500 to 62000. The `parallel fork/join` consumes 13.7% of all cache misses, the double for-loop is responsible for 83.7% of them.

Figure 5: The other parts of the program play an insignificant role compared to the double for-loop, whilst there is some variability between the threads

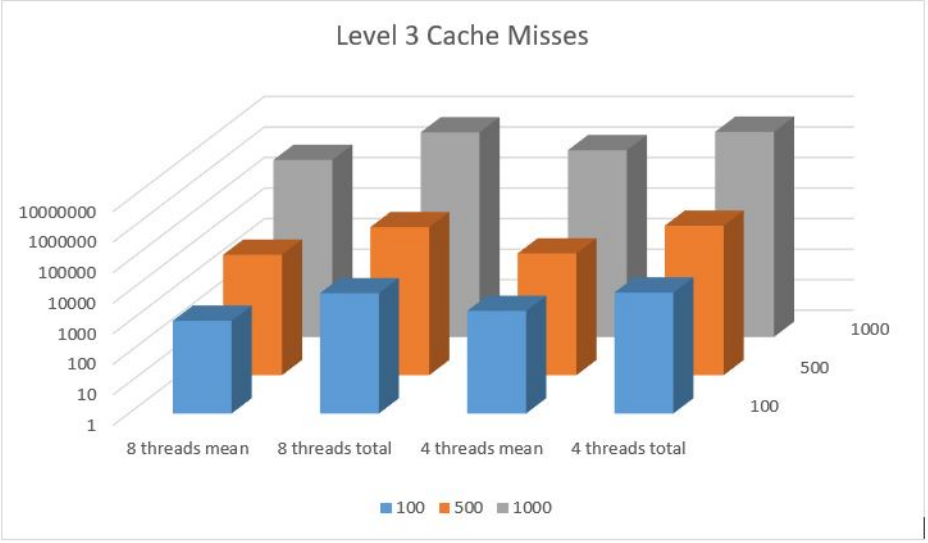


Figure 6: comparison of 4 and 8 threads cache misses for different problem sizes on a logarithmic scale

2.4 Total Instructions

2.4.1 2 threads

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
           counts total counts
---------------------------------------------------------------------------------------
100.0    1.196E+07    3.757E+09           1           1 3756922236 .TAU application
 99.7    8.471E+04    3.745E+09           1           3 3744966401 main
 99.7    2.261E+05    3.745E+09           1           1 3744841689 parallel (parallel fork/join)
 99.7    1.279E+06    3.745E+09           1        1002 3744615637 parallel (parallel begin/end)
 99.6    1.802E+06    3.743E+09        1000        1000    3743327 laplace_step
 99.6    3.583E+09    3.742E+09        1000      200002    3741525 for (loop body)
  2.1     7.95E+07     7.95E+07      100001           0        795 max_error [THROTTLED]
  2.1     7.92E+07     7.92E+07      100001           0        792 stencil [THROTTLED]
  0.0        4E+04        4E+04           2           0      20001 laplace_init
  0.0         8133         8133           1           0       8133 master (master begin/end)
  0.0          815          815           1           0        815 parallel (barrier enter/exit)

NODE 0;CONTEXT 0;THREAD 1:
---------------------------------------------------------------------------------------
```

```
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
           counts total counts
---------------------------------------------------------------------------------------
100.0    5.672E+04    3.754E+09           1           1 3753769193 .TAU application
100.0    1.315E+06    3.754E+09           1        1001 3753712468 parallel (parallel begin/end)
 99.9    1.843E+06     3.75E+09        1000        1000    3750294 laplace_step
 99.9     3.59E+09    3.748E+09        1000      198046    3748451 for (loop body)
  2.1    8.001E+07    8.001E+07       99024           0        808 max_error [THROTTLED]
  2.1    7.843E+07    7.843E+07       99022           0        792 stencil [THROTTLED]
  0.1    2.103E+06    2.103E+06           1           0    2102895 parallel (barrier enter/exit)


FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
           counts total counts
---------------------------------------------------------------------------------------
100.0    6.006E+06    3.755E+09           1           1 3755345714 .TAU application
 99.8    1.297E+06    3.749E+09           1      1001.5 3749164052 parallel (parallel begin/end)
 99.8    1.823E+06    3.747E+09        1000        1000    3746811 laplace_step
 99.7    3.586E+09    3.745E+09        1000      199024    3744988 for (loop body)
 49.9    4.236E+04    1.872E+09         0.5         1.5 3744966401 main
 49.9     1.13E+05    1.872E+09         0.5         0.5 3744841689 parallel (parallel fork/join)
  2.1    7.976E+07    7.976E+07     99512.5           0        801 max_error [THROTTLED]
  2.1    7.881E+07    7.881E+07     99511.5           0        792 stencil [THROTTLED]
  0.0    1.052E+06    1.052E+06           1           0    1051855 parallel (barrier enter/exit)
  0.0        2E+04        2E+04           1           0      20001 laplace_init
  0.0         4066         4066         0.5           0       8133 master (master begin/end)
```

Here, the `pprof` output draws a clear picture: $\sim 97\%$ of the execution time and $\sim 96\%$ of the instructions are consumed during the nested for-loop, marginalizing the instructions for TAU and `parallel`.

Figure 7: The `main` (orange) as well as `laplace_init` (blue) only happens in the main thread, which is reflected in this normalized `paraprof` output

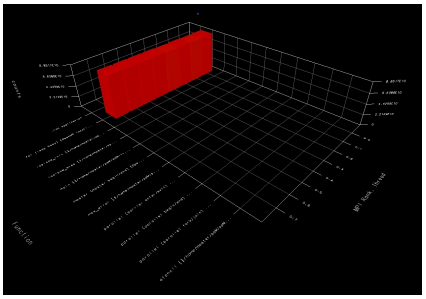
2.4.2 8 threads

Problem size $n = 1000$

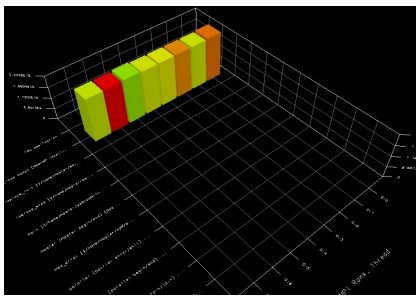
```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
           counts total counts
---------------------------------------------------------------------------------------
100.0    1.663E+06     8.85E+10           1           1 88503563932 .TAU application
100.0    1.301E+06     8.85E+10           1     1001.12 88501648242 parallel (parallel begin/end)
100.0    1.958E+06     8.85E+10        1000        1000   88498507 laplace_step
100.0    8.833E+10     8.85E+10        1000      199216   88496549 for (loop body)
 12.5    1.051E+04    1.108E+10       0.125       0.375 88672931312 main
```

```
 12.5   9.826E+04   1.108E+10       0.125       0.125 88671696272 parallel (parallel fork/join)
  0.1   9.027E+07   9.027E+07     99608.5           0        906 max_error [THROTTLED]
  0.1   7.889E+07   7.889E+07     99607.5           0        792 stencil [THROTTLED]
  0.0   1.839E+06   1.839E+06           1           0    1838586 parallel (barrier enter/exit)
  0.0   1.439E+05   1.439E+05        0.25           0     575460 laplace_init
  0.0        1052        1052       0.125           0       8418 master (master begin/end)
```

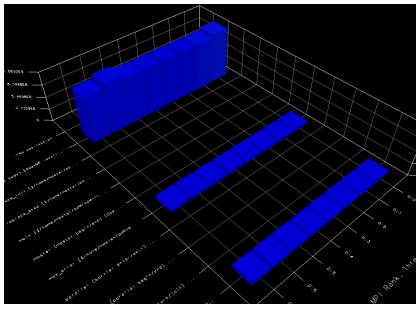
Figure 8: Compared to the Cache misses, the number of instructions are nearly constant between the 8 threads, which leads to a very small standard deviation



(a) Total instructions with 8 threads an $n = 1000$



(b) Total instructions with 8 threads and $n = 500$



(c) Total instructions with 8 threads and $n = 100$

Problem size $n = 500$

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
            counts total counts
---------------------------------------------------------------------------------------
100.0     1.58E+06     2.23E+10           1           1 22302478701 .TAU application
```

```
100.0   1.301E+06    2.23E+10          1     1001.12 22300556447 parallel (parallel begin/end)
100.0   1.952E+06    2.23E+10       1000        1000   22297415 laplace_step
100.0   2.213E+10    2.23E+10       1000      197794   22295463 for (loop body)
 12.6   1.051E+04    2.82E+09      0.125       0.375 22557877852 main
 12.6   2.873E+05    2.82E+09      0.125       0.125 22557434948 parallel (parallel fork/join)
  0.4   8.985E+07   8.985E+07    98898.1           0        909 max_error [THROTTLED]
  0.4   7.833E+07   8.278E+07    98896.9         0.5        837 stencil [THROTTLED]
  0.0   1.839E+06   1.839E+06          1           0    1838587 parallel (barrier enter/exit)
  0.0   4.485E+04   4.485E+04       0.25           0     179404 laplace_init
  0.0        1050        1050      0.125           0       8396 master (master begin/end)
```

Problem size $n = 100$

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
           counts total counts
---------------------------------------------------------------------------------------
100.0   1.593E+06   1.201E+09          1           1 1201193798 .TAU application
 99.8   1.301E+06   1.199E+09          1     1001.12 1199314411 parallel (parallel begin/end)
 99.6     1.9E+06   1.196E+09       1000        1000    1196173 laplace_step
 99.4   1.038E+09   1.194E+09       1000      196502    1194274 for (loop body)
 13.0   1.053E+04   1.556E+08      0.125       0.375 1244577237 main
 13.0   2.704E+05   1.556E+08      0.125       0.125 1244452974 parallel (parallel fork/join)
  6.5   7.831E+07   7.831E+07    98251.8           0        797 max_error [THROTTLED]
  6.5   7.781E+07   7.781E+07    98250.6           0        792 stencil [THROTTLED]
  0.2   1.839E+06   1.839E+06          1           0    1838592 parallel (barrier enter/exit)
  0.0        5000        5000       0.25           0      20001 laplace_init
  0.0        1022        1022      0.125           0       8179 master (master begin/end)
```

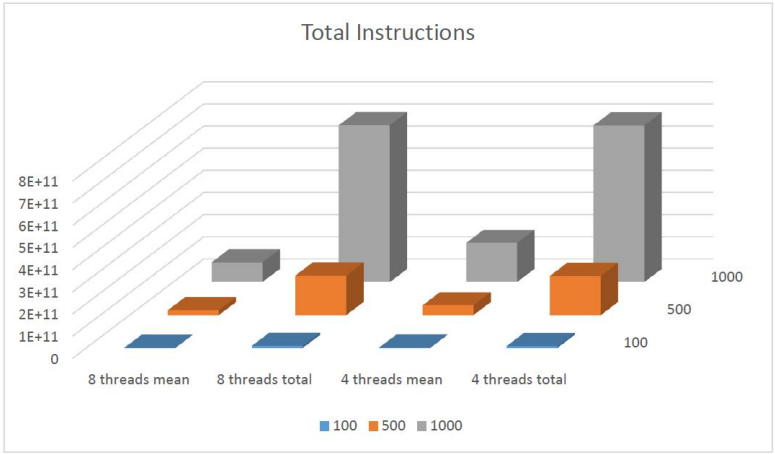


Figure 10: comparison of 4 and 8 threads total instructions for different problem sizes

3 Ofast

Ofast with `gcc`, 8 threads

```
Jacobi relaxation Calculation: 1000 x 1000 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.010200, A[7][7]= 0.016615

 Performance counter stats for './lap_2_open_fast 1000':

       926,100107      task-clock (msec)         #    2,014 CPUs utilized
     2.441.747.945     cycles                    #    2,637 GHz
     4.105.164.512     instructions              #    1,68  insns per cycle

       0,459805784 seconds time elapsed
```

Ofast with `tau_cc.sh`, 8 threads

```
Jacobi relaxation Calculation: 1000 x 1000 mesh, maximum of 1000 iterations
Total Iterations:  1000, ERROR: 0.010200, A[7][7]= 0.016615

 Performance counter stats:

    100208,549694      task-clock (msec)         #    3,933 CPUs utilized
   319.593.850.400     cycles                    #    3,189 GHz
   602.953.311.419     instructions              #    1,89  insns per cycle

     25,476217679 seconds time elapsed
```

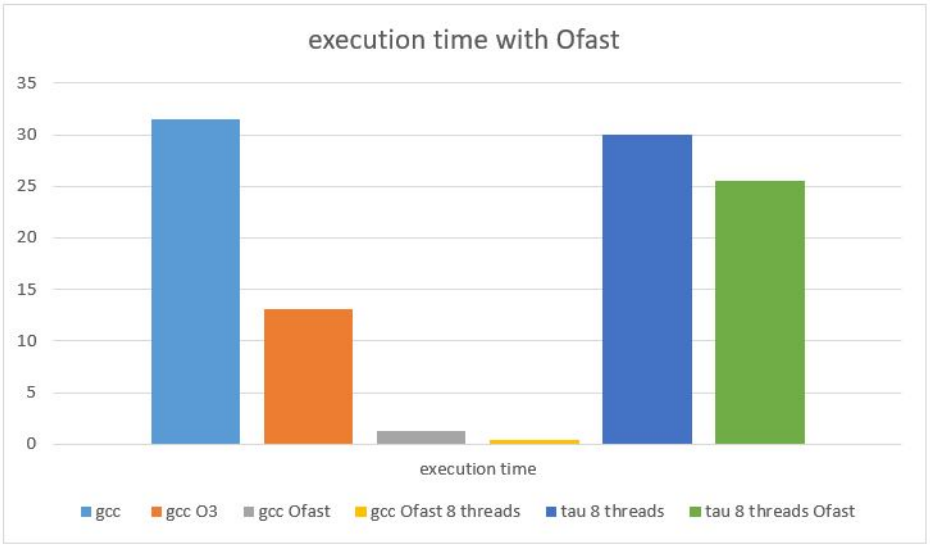


Figure 11: Comparison of execution times with the optimization tool Ofast.

We compiled the program under different scenarios with the compiler flag `-Ofast`. For the serial execution, we obtained a massive time decrease from ~ 31 sec to ~ 1.3 sec. When we added the parallelization with `fopenmp`, execution time was even reduced to 0.45 sec. However, compiling with `tau_cc.sh`, the effect of `Ofast` was only minor. Adding `Ofast` gave a time reduction from ~ 29 sec to ~ 25 sec.

4 Conclusion

As part of this exercise and as presented in this report, we have implemented parallelization of the laplace program using OMP and monitored several metrics and their impact on performance.

Execution time
With respect to the program compiled with `tau` compiler, the execution time decreased as the number of threads was increased. In most cases, this decrease was proportional to the change in number of threads. However, not much difference is noticed between using 4 threads on a 4-core CPU and using 8 threads on the same CPU.

Problem size
The execution time increased quadratically in proportion to the size of the mesh when the mesh size was increased, and this was as expected as the major part of the program has a quadratic complexity ($\mathcal{O}(n^2)$).

Level 3 cache misses
It was noticed that the amount of L3 cache misses increased as the number of threads was increased.

Total instructions
In terms of instructions it was noticed that there was little variation between the number of instructions executed by each thread when the program was parallelized. The total instructions did remain the same even with an increase in the number of threads, however the average did reduce linearly with respect to the number of threads.

Despite the fact that the use of the OFAST compiler flag had a significant impact on the performance of the program when compiled with `gcc`, very little change was noticed when combined with the Tau compiler.