Universitat Autònoma de Barcelona

Parallel Programming: MPI

Assignment 3

Alfredo Hernández Alejandro Jiménez

9th January

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 MPI.

In order to make the biggest difference on the code and fully appreciate the power of parallel computing, we chose the most computational demanding loop and created many threads using OpenMP, which would split the iterations into different, parallel, more efficient processes. Nextly, we would add MPI orders in as many loops as we did found useful.

Eventually, We will measure the impact of the parallelisation of the code using 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 Implementation of MPI

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 openmpi/1.8.1
```

We need to compile the `Makefile` to use TAU with MPI:

```
cd tau-2.26
./configure -prefix=/home/master/ppM/ppM-1-10/my_TAU -mpi
export PATH=/home/master/ppM/ppM-1-10/my_TAU/x86_64/bin:$PATH
make install
```

We need to export the Makefile to the environment variable to be able to use TAU with MPI whilst compiling and running our C program, as well as the `optCompInst` option:

```
export TAU_MAKEFILE=/home/master/ppM/ppM-1-10/my_TAU/x86_64/lib/
    Makefile.tau-mpi
export TAU_OPTIONS=-optCompInst
```

2.2 Parallelisation with MPI

To compile an MPI program we would usually compile the C code using `mpicc`:

```
mpicc -lm laplaceMPI.c -o compiledfile
```

However, to use TAU, we need to compile it using

```
tau_cc.sh -lm laplaceMPI.c -o compiledfile
```

One of the main variables to consider when improving the performance of a code using MPI is the number of threads that we use. Specifying the exact number of threads to be used can be done easily when running the compiled file.

In the following snippet it is showed how to run the compiled code over a 1000×1000 matrix using 4 threads.

```
mpirun -np 4 compiledfile 1000
```

An important difference between the usage of OpenMP and MPI is that we specify the number of threads within the very code in the former, whereas we set the number of threads when executing the compiled file in the latter.

Using MPI we 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 the proceeding section, we will explain our implementation of MPI to parallelise the code to solve 2D Laplace equation using Jacobi Iteration method.

2.3 Basic parallel solution

Suppose that we are going to use N processes to solve the problem in a distributed way. Accordingly with the characteristics of the problem, each of these processes will have to carry on the computation over a portion of the matrix A, taking care for interchanging the necessary data and synchronising with other processes. The specific processes that will have to communicate will depend on how the data is partitioned among them.

The simplest partition of A is shown in figure 1. In this case, each process P_i takes care of the computation of m/N rows of A and, in the general case ($0 < i < N - 1$), it needs the last row of P_{i-1} and the first of P_{i+1}.

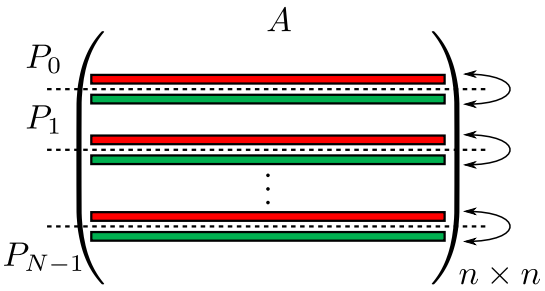

Figure 1: Diagram of the partition of the matrix A

First of all, we need to define some variables in our C program that are needed for the MPI implementation, like `rank`, `size`, `m`, `ri`, `rf`; and to initialize the MPI parallel region:

```
52  // Variables needed for MPI
53 int rank, size;
```

We define the initial and final row of the subpartitions of the matrix as

$$\begin{aligned} r_i[k] &= \frac{n}{N} \times ()k \\ r_f[k] &= \left(\frac{n}{N} \times (k+1)\right) - 1 \end{aligned} \tag{1}$$

where n is the dimension of the matrix, N (or `size`) is the amount of threads used, and $k = 0, \ldots, N-1$ (or `rank`) is the number of the working thread. Notice that Thread 0 will be used to make calculations as well as being the master thread to coordinate all computations. Notice that n/N must be an integer.

```
75 // Initialisation of MPI parallel region
76 MPI_Init(&argc, &argv);
77 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
78 MPI_Comm_size(MPI_COMM_WORLD, &size);
79 MPI_Status status;
80
81 // Define the partition of the matrix
82 int m = n/size;
83 int ri = rank * m,
84 int rf = (rank+1) * m-1;
```

In the main loop of the program, we implement the basic parallel solution as described above. In the snippets below we can see excerpts of the script where MPI has been used. The rest of the program is almost identical to the standard implementation without MPI.

We can send and recieve messages from the different threads:

```
96  // Interchange of messages between consecutive rows
97  if ( rank > 0 ) {
98    MPI_Send(&A[ri*n], n, MPI_FLOAT, rank-1, 1, MPI_COMM_WORLD);
99    MPI_Recv(&A[(ri-1)*n], n, MPI_FLOAT, rank-1, 2, MPI_COMM_WORLD,
          &status);
100 }
101 if ( rank < size-1 ) {
102   MPI_Send(&A[rf*n], n, MPI_FLOAT, rank+1, 2, MPI_COMM_WORLD);
103   MPI_Recv(&A[(rf+1)*n], n, MPI_FLOAT, rank+1, 1, MPI_COMM_WORLD,
          &status);
104 }
```

The main part of the program, performing the Laplace step function using the implemented partition of the matrix A:

```
106 // Computation of the Laplace Step using MPI
107 if ( rank == 0 ) {
108   for ( j = ri+1; j <= rf; j++ ) {
109     for ( i = 1; i < n-1; i++ ) {
110       temp[j*n+i] = stencil(A[j*n+i+1], A[j*n+i-1], A[(j-1)*n+i],
              A[(j+1)*n+i]);
111       error = max_error( error, temp[j*n+i], A[j*n+i] );
112     }
113   }
114 }
115 if ( rank == (size-1) ) {
116   for ( j = ri; j < rf; j++ ) {
117     for ( i=1; i < n-1; i++ ) {
118       temp[j*n+i] = stencil(A[j*n+i+1], A[j*n+i-1], A[(j-1)*n+i],
              A[(j+1)*n+i]);
119       error = max_error( error, temp[j*n+i], A[j*n+i] );
120     }
121   }
122 }
123 if ( rank != 0 && rank != (size-1) ) {
124   for ( j = ri; j <= rf; j++ ) {
125     for ( i = 1; i < n-1; i++ ) {
126       temp[j*n+i] = stencil(A[j*n+i+1], A[j*n+i-1], A[(j-1)*n+i],
              A[(j+1)*n+i]);
127       error = max_error( error, temp[j*n+i], A[j*n+i] );
128     }
129   }
130 }
```

We send the partial errors of each thread to the master:

```
132 // Sending partial errors to the master and computing the maximum to
        find the global error
```

```
133 if ( rank != 0 ) {
134     MPI_Send(&error, 1, MPI_FLOAT, 0, 3, MPI_COMM_WORLD);
135 } else {
136   errors[0] =error;
137   for (i = 1; i < size; i++ ) {
138     MPI_Recv(&errors[i], 1, MPI_FLOAT, i, 3, MPI_COMM_WORLD,
          &status);
139   }
140 }
141 global_error = maximum(errors, size);
```

The master calcualtes the global error using the partial errors previously sent:

```
143 // Sending back the resulting error
144 if ( rank == 0 ) {
145   for(i = 1; i<size; i++ ) MPI_Send(&global_error, 1, MPI_FLOAT, i,
        4, MPI_COMM_WORLD);
146 } else {
147   MPI_Recv(&global_error, 1, MPI_FLOAT, 0, 4, MPI_COMM_WORLD,
        &status);
148 }
```

As usual, once we've finished the computations, we need to *close* MPI.

```
156 MPI_Finalize();
```

As it can be seen, it is almost the same algorithm implemented in the provided `lapFusion.c` code, just distributing the work and adding the data interchange among processes.

3 Assessment of the solution

3.1 Analysis with TAU

Up to this point we are well aware of how the performance increases using OpenMP as a tool to parallelise the code. So now we want to explore further parallelisation, but using MPI this time. Once again 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.

3.2 Influence of the number of threads

3.2.1 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 variable:

```
1 export TAU_PROFILE=1
```

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

```
1 tau_cc.sh -lm laplaceMPI.c -o mpiruntau
2 mpirun -np N ./mpiruntau 512 500
```

where `N` can be 2 or 4.

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.

The result of the execution of the program using 2 and 4 threads can be seen in the tables 1, 2 and figures 2, 3 below. They show quite interesting results. On one hand we can see how the *Run Time* of execution of the internal MPI operations are pretty much the same. The only main difference is the execution of the whole program itself; this is the calculation of the Laplace step.

An interesting conclusion that can be extracted from these graphs is that all threads are performing more or less the same. This is reflected in the fact we can not appreciate a big difference in the time needed for every thread; they all took the same time to finish the same work. This would be completely different if we have assigned a specific thread to act as a master only, instead of the master doing calculations as well.

Function	Thread 0	Thread 1
`MPI_Init()`	283	282
`MPI_Comm_size()`	0	0
`MPI_Comm_rank()`	0	0.001
`MPI_Send()`	0.877	1
`MPI_Recv()`	47	50
`MPI_Finalize()`	81	81
`.TAU application`	7139	7138

Table 1: Run time (in ms) obtained on 2 threads

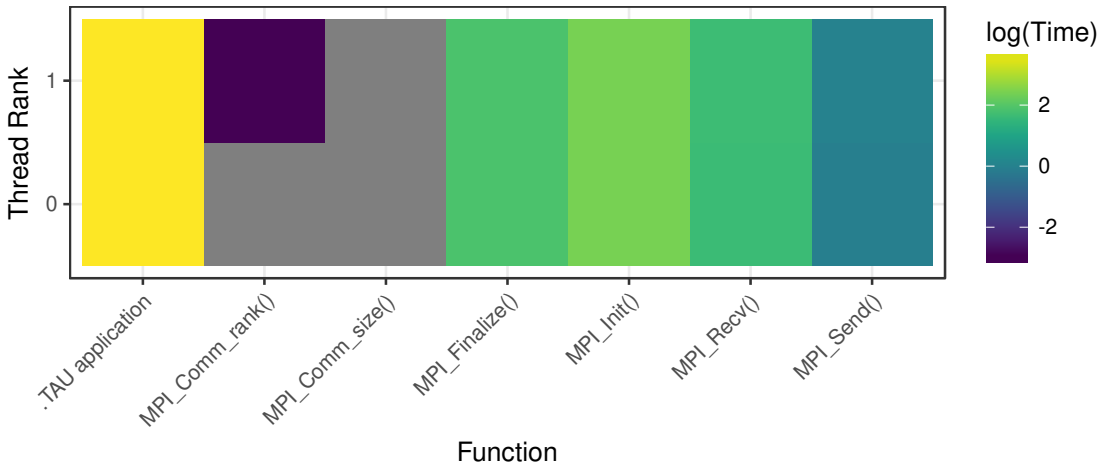

Figure 2: Heatmap of the run time (in ms) obtained on 2 threads

Function	Thread 0	Thread 1	Thread 2	Thread 3
`MPI_Init()`	298	284	281	276
`MPI_Comm_size()`	0	0	0	0
`MPI_Comm_rank()`	0	0	0	0
`MPI_Send()`	1	1	1	1
`MPI_Recv()`	60	28	34	55
`MPI_Finalize()`	81	81	82	82
`.TAU application`	3809	3795	3793	3788

Table 2: Run time (in ms) obtained on 4 threads

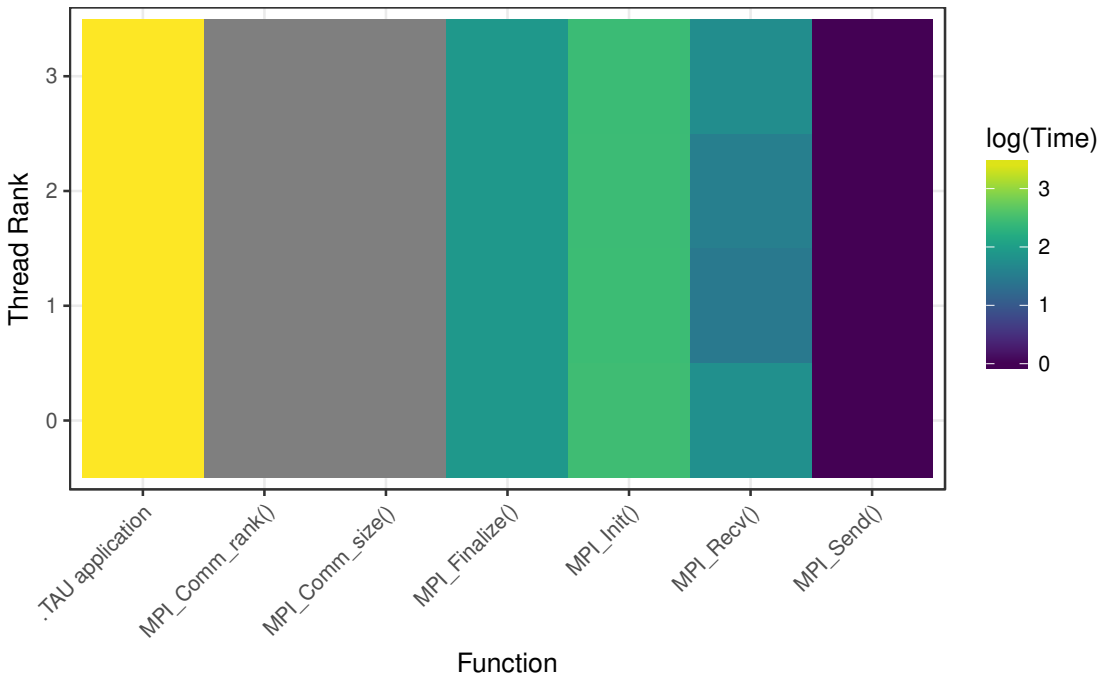

Figure 3: Heatmap of the run time (in ms) obtained on 4 threads

We can see that the time of this distributed version using MPI is $T/N+$ communication-overhead (where T is the time it takes for the sequential algorithm to complete). Using 4 threads instead of 2 does not give us exactly half the compilation time; we get $\approx 3800\,\text{ms}$ instead of $\approx 7100/2\,\text{ms} = 3550\,\text{ms}$.

3.2.2 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.

In particular, we aim to see the relations established between the threads and the information they process. This analysis could lead to a better insight on the strengths and weaknesses of our parallelisation and facilitate further improvement.

In order to do so, we must first to arrange some technical issues.

First of all, we need to set the following environment variable:

```
export TAU_TRACE=1
```

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

```
tau_cc.sh -lm laplaceMPI.c -o mpiruntau_trace
mpirun -np N ./mpiruntau_trace 512 500
```

where `N` can be 2 or 4.

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 figures 4 and 5 we can see an example of the Time Line of internal operations of each one of the used threads. Essentially, it shows us the information exchanged between different threads in chronological order. This helps us to understand how information/messages are sent and recieved by the different threads to perform the parallelisation.

One crucial point of this visualisation is to spot redundancies or any kind of misperformance within the code.

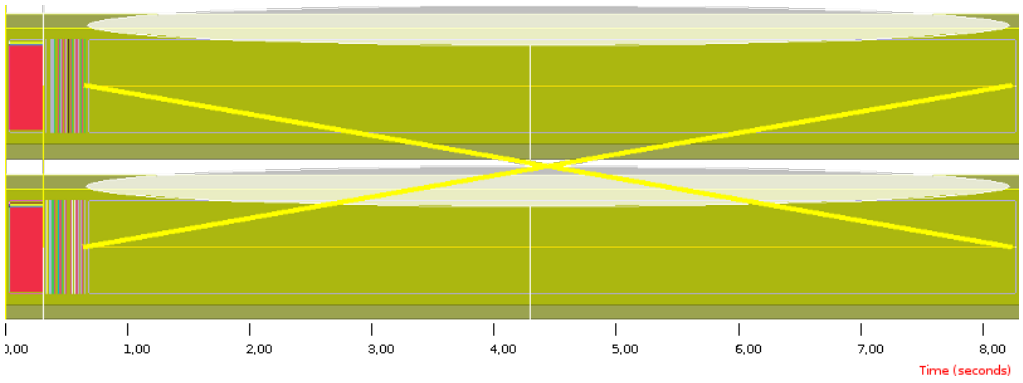

Figure 4: Time Line of the 2 threads using `jumpshot`

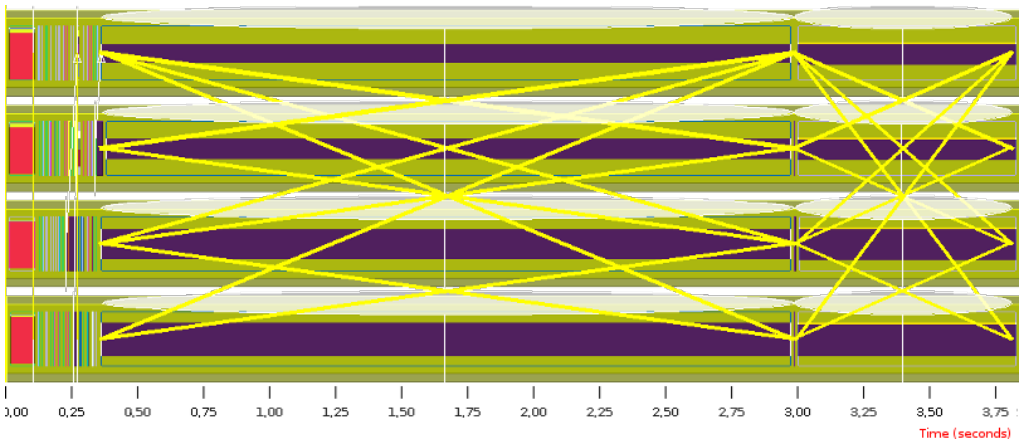

Figure 5: Time Line of the 4 threads using `jumpshot`

It is interesting to notice how the work is distributed and iteratively reorganised along the different threads within the timeline of the computation.

3.3 Influence of the problem size

3.3.1 Using profile files (paraprof)

Up until now, we have learnt how our code behaves when different degrees of parallelisation are applied. Moreover, we now know some of the internal details involved in the performance of our new optimised code.

One question that we now could ask ourselves is how the performance of the code scales. Until now we have studied the increase in the performance for a given magnitude of the matrix, but we have not considered how a change in this magnitude could affect the the computational effort needed to solve it. At first sight, we could guess that the larger the size of the matrix, the longer it will take to finish the problem. But, would all functions within the code increase in the same way? Or some functions would be more affected than others?

One way to address these questions and evaluate the scalability is simply to run the code for different matrix sizes and measure the time needed each time.

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

```
export TAU_PROFILE=1
export TAU_TRACE=0
```

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

```
tau_cc.sh -lm laplaceMPI.c -o mpiruntau
mpirun -np 4 ./mpiruntau n 500
```

where we have chosen `n` to be 128, 256, 384, 512, 640, and 768.

Function	**128 × 128**	**256 × 256**	**384 × 384**	**512 × 512**	**640 × 640**	**768 × 768**
`MPI_Init()`	287	334	284	281	280	281
`MPI_Comm_size()`	0.0005	0	0.000 25	0.0005	0.000 75	0.0005
`MPI_Comm_rank()`	0.000 25	0	0	0	0.000 25	0.000 25
`MPI_Send()`	0.583	1	1	1	1	2
`MPI_Recv()`	7	15	12	13	13	13
`MPI_Finalize()`	82	282	81	82	82	119
`.TAU application`	600	1618	2299	3781	5483	7705

Table 3: Run time (in ms) obtained on 4 threads for different problem sizes

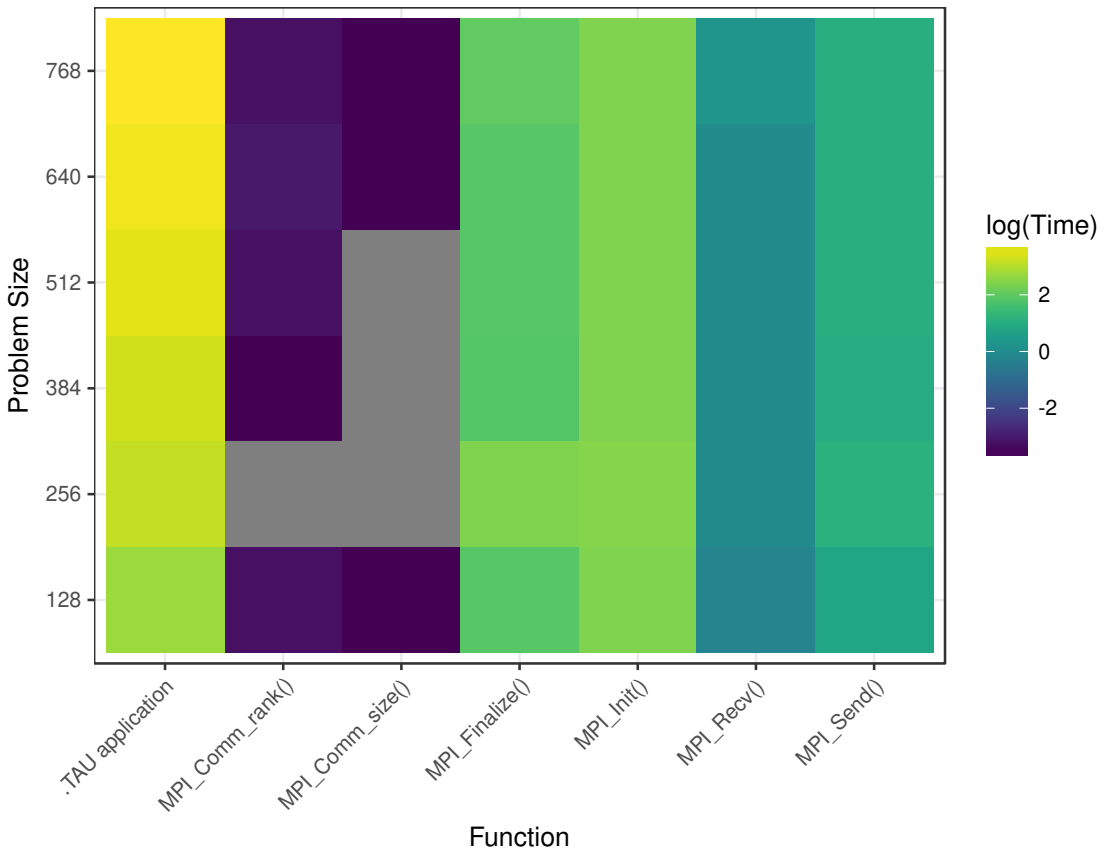

Figure 6: Heatmap of the mean run time (in ms) obtained on 4 threads for different problem sizes

A possible concern involving the usage of MPI could be the loss of efficiency caused by the constant interchange of information between the threads. This loads more work to our code and thus can slow down the final output of the code, even though we aim to increase the performance thanks to making some processes parallel. We have checked that this not happens for a matrix where $n = 512$, because the improvement caused thanks to the parallelisation largely overcomes the possible lack of performance that more lines of orders could cause. Then, a still reasonable concern would be asking whether the parallelisation is also worth it for different dimensions of the problem.

Table 3 shows how the time needed to execute our program increases over the size of the problem. Looking at figure 6, we can visualise and distinguish where this increase in time is happening; and notice that the function `.TAU application` is the one suffering an increase in time execution, whereas all others increase are clearly negligible, if existent. That function is the one responsible for the computations of the matrix, whilst the others are those MPI functions needed to exchange information between the different threads.

Every possible drawback is, then, absolutely dwarfed by the increase in the performance caused by the parallelisation method that we have implemented. And this effect becomes even greater when we increase the size of the problem.

4 Conclusions

During the resolution of this exercise, we have had the opportunity to manually create a parallel code, in contrast with the one we had optimised in the OpenMP assignment, that was itself an improvement from the very first code of the course. This time we have used a different methodology to paralellise the code, the widely employed *MPI*. Moreover, we have got the opportunity to further practice with the analytical tool kits known as *TAU* and the basics to understand and interpret different visualisation techniques for performance measurement, profiling, and tracing on parallelised code.

All of this has let us comprehend the importance of parallel programming when aiming for high performance computation, for we have experienced an increase of the performance in every code we have been improving.

At one point we have considered to combine the power of MPI and OpenMP together, as suggested in the assignment; but we have discarded it. The logic for this decision is that there is a limit in the parallelisation we can perform in the code, settled by the number of threads of our computer; and there is no point in adding more parallelisation than our system can handle.