C Programming and Performance Engineering

Lab Work- Hybrid MPI + OpenMP Parallel Programming (with TAU)

December, 2017

Spyridoula Chrysikopoulou-Soldatou and Jeremy Williams

1. Basic Parallel Solution

In order to solve the 2D Laplace problem in a distributed way, each process has to take a specific portion of the matrix and perform computations. For this reason the matrix is being divided in portions, where each portion is being distributed to the processors.

As a result, each process has only a portion of the matrix to compute. Because of the nature of the problem, each process generally needs the last row of the above process and the first row of the next process. This is where the processes need to communicate with each other and send/receive the necessary information.

The most important additions and/or changes in the initial code in order to solve the 2D Laplace equation using Jacobi Iteration method with MPI are the following [1]:

1. First of all, the MPI execution environment was initialized inside the main function. From this point on, every process executed a separate copy of this program:
   MPI_Init(&argc,&argv)

2. Afterwards it was necessary to determine the number of processes participating in the computation:
   MPI_Comm_size (MPI_COMM_WORLD, &nprocs)

   And to determine the rank, or the ID number of the calling process within the communicator:
   MPI_Comm_rank (MPI_COMM_WORLD, &rank)

3. It was necessary to return an elapsed wall clock time in seconds (double precision) on the calling processor:
   MPI_Wtime()

4. Afterwards, it was necessary to determine the number of rows and the size of each portion. The number of rows of each portion (my_nrows) can be found by the total number of rows divided by the number of processors (my_nrows = n/nprocs). In this case is not needed to add the two more rows that the portion will receive later, because my_nrows is going to be used in the communication.

The size of each portion is going to be used in the allocation of the matrix my_A and my_temp. For this reason the whole size of the matrix must be calculated, including the two additional rows that will be added after the communication (my_size = n*(my_nrows+2)).

5. Every process has a portion of the matrix, which are named my_A and my_temp (in order to swap them later). Both are allocated similarly to the matrices A and temp:
   my_A = (float*) malloc(my_size*sizeof(float)))
   my_temp = (float*) malloc(my_size*sizeof(float))

6. Regarding the distribution of the rows of A and temp in the processes MPI Scatter was used; which distributes different data to each process in the communicator in rank order. This order is important for the program, as it is needed to know the order that the data are distributed among the processes to perform the communication later. The root process acts as a master and sends the portions to each child process:
   MPI_Scatter(A,my_nrows*n,MPI_FLOAT,my_A+n,my_nrows*n,MPI_FLOAT,0,MPI_COMM_WORLD);

7. After the distribution of the portions into the processes, the portions need to be copied also into the my_temp matrix of each process. Regarding the A and temp matrices, this happened with the use of initialization function laplace_init for both, but this cannot happen for my_A and my_temp.

8. Regarding the communications that had to occur, it was needed to differentiate the code for every case of process. In general, each of the middle processes receive two additional rows, as they need them to compute the new values. The exception is the first and last process; which receive only one additional row.

   All the processes apart from the root need a previous row. Each one of them sends its first row to the previous process and receives the last from the previous process:
   MPI_Send(my_A+n, n, MPI_FLOAT, rank-1, tag, MPI_COMM_WORLD)
   MPI_Recv(my_A, n, MPI_FLOAT, rank-1, tag, MPI_COMM_WORLD, &status)

   On the other hand, the last process does not need a last row. For this reason all the rest processes including the root one receive the first row of the next process and send their last row to the next process:
   MPI_Recv((my_A+n*(my_nrows+1)),n,MPI_FLOAT,rank+1,tag,MPI_COMM_WORLD, &status)
   MPI_Send((my_A+ n*(my_nrows)), n, MPI_FLOAT, rank+1, tag, MPI_COMM_WORLD)

9. Regarding the laplace step function, it is needed for the processes to update only the internal part of the A. For this reason it is necessary again to differentiate the first and last processor compared to the rest. Two new values ri and rf are introduced, where ri is the first row and rf is the last row of each portion, for which the error will be computed in the laplace_MPI_step. These two values are different for each type of processes (first, middle and last).

10. The maximum of all the errors computed from the different processes is saved in the root process:
    MPI_Reduce(&my_error,&error,1,MPI_FLOAT,MPI_MAX,MASTER,MPI_COMM_WORLD)

11. In order for the while loop to exit and the program to end, the error needs to be higher than the tolerance level, or the maximum number of iterations to be reached. As a result, every process has to know the maximum error at every time. For this reason, the error from the root process is broadcasted to all the rest processes:
    MPI_Bcast(&error, 1, MPI_FLOAT, 0, MPI_COMM_WORLD)

12. After the tolerance level is reached, with the MPI_Gather the root processor gathers all the portions of the matrix A stored in each my_A.
    MPI_Gather(my_A+n, my_nrows*n, MPI_FLOAT, A, my_nrows*n, MPI_FLOAT, 0, MPI_COMM_WORLD)

13. And after computing the final error and receiving the time details, the MPI environment is finalized:
    MPI_Finalize()

The commands in order to compile and execute the program are the following:

- module load gcc/6.1.0 openmpi/1.8.1
- mpicc -g -Wall -lm -Ofast -fopenmp -o lapMPI8 lapFusionMPI8.c
- mpirun -np N lapMPI8 n

2. Hybrid Solution

The performance of the basic solution, that has been introduced previously, has as a limiting factor for the overhead introduced by a communication process. In order to reduce this overhead, numerous mechanisms exist.

Firstly, a possibility is to **overlap communication and computation**, where the program instead of sitting idle, while waiting for the remote data to become available, does asynchronous communication calls and compute first the inner points of the matrix portion assigned to each process and then the border points. Nevertheless, the asynchronous communication was not chosen, because errors in the computer clocks can be accumulated and eventually cause errors in the data transfer. Also, the data transfer rate is slower and has usually more overhead than synchronous communication.[2,3]

The second existing mechanism in order to improve the program is the **block partitioning**, where instead of distributing the matrix by rows, a block distribution is being done. In this way the amount of communication between processes is reduced when the number of processes increases. Again this mechanism was not selected, as in order for the matrices to be distributed as blocks, 4,9,16,25 etc processes are needed, and also the matrix has to be divided evenly in each case, and the input can take only specific values.

Regarding the use of **block communication**, in which instead of interchanging the first and last rows for each process in each iteration, interchange the first and last k rows, not only the number of elements computed in each process is increased, but also the functions probably are going to block for more time than with the basic code.

The chosen mechanism in order to improve the MPI program is the **hybrid solution**. For this solution, OpenMP is being used, as the process of each node is parallelized over the available cores. The advantage of the hybrid solution is lower memory latency, data movement within the node as the memory traffic is reduced, with its performance able to be improved.

In MPI-only programming, each MPI process has a single program counter. In MPI and OpenMP hybrid programming there can be multiple threads executing simultaneously. All threads share all MPI objects (communicators, requests). In the Laplace problem, the function that takes the most computation time is the Laplace step, and supposedly if the work is distributed more into threads, the performance will be improved. On the other hand, the disadvantage of hybrid MPI and OpenMP programming is that the code can be slower, because of overhead thread creation.[4]

The most important additions and/or changes in the basic MPI code are the following:

1. Firstly, the header file omp.h was inserted in the code.

2. TheMPI execution environment had to be initialized. In the hybrid case, MPI_Init_thread was used (in the place of MPI_Init) in order to select MPI's thread level of support (thread safety).There are four types of thread support: single, funneled, serialized and multiple. Single means there is no multi-threading. Funneled means only the master thread (the one that called MPI_Init_thread) will make MPI calls (calls are "funneled" to main thread). Serialized means multiple threads can call MPI, but only one call can be in progress at a time (serialized). Multiple means that multiple threads may call MPI with no restriction. From the four above options, the MPI_THREAD_FUNNELED was chosen, as all MPI calls are made by the master thread outside the OpenMP parallel regions:[4]
   MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &provided)

3. The pragmas for the OpenMP have to be inserted inside the main function, and specifically inside the while loop. The "pragma omp parallel" was inserted after the beginning of the while loop. Nevertheless, the MPI_Send and MPI_Recv create problems in the execution of the program. For this reason, all the regions inside the program that contain MPI Send and Receive have been included inside "pragma omp barrier". The omp barrier directive identifies a synchronization point at which threads in a parallel region will not execute beyond the omp barrier until all other threads in the team complete all explicit tasks in the region.[4]

4. At last, in the function laplace_MPI_step the "pragma omp for" was inserted before the for loop.

The commands in order to compile and execute the program are the following:

- module load gcc/6.1.0 openmpi/1.8.1
- export OMP_NUM_THREADS=4
- export I_MPI_PIN_DOMAIN=omp
- mpicc -g -Wall -lm -Ofast -fopenmp -o lapMPI9_HY LapFusionMPI9_hybrid.c
- mpirun -np N lapMPI9_HY n

3. Assessing the Solutions

3.1 Basic Parallel Solution

Firstly the basic parallel solution was executed with a 1000 x 1000 and a 2000 x 2000 mesh , maximum of 1000 iterations, without the -Ofast flag.

1000 x 1000	2000 x 2000
With 2 processes: 17.092315 secs	With 2 processes: 58.154199 secs
With 4 processes: 8.560498 secs	With 4 processes: 28.783829 secs

Afterwards, the -Ofast flag was added while compiling the program, to check the differences without it.

1000 x 1000	2000 x 2000
With 2 processes: 0.579609 secs	With 2 processes: 4.112238 secs
With 4 processes: 0.530142 secs	With 4 processes: 4.072224 secs

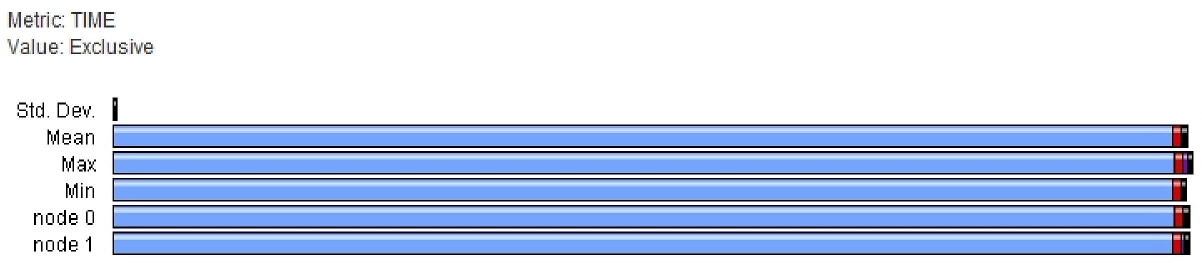
3000 x 3000	4000 x 4000
With 2 processes: 9.185817 secs	With 2 processes: 15.936447 secs
With 4 processes: 9.158911 secs	With 4 processes: 16.428872 secs

When the program is executed without the -Ofast flag, it is obvious that the use of multiple processes causes a significant decrease of the execution time. On the other hand, when the Ofast flag is activated, the execution time is almost similar with two and four processors, for all four mesh sizes.

3.1.1 Profiling

2 processes

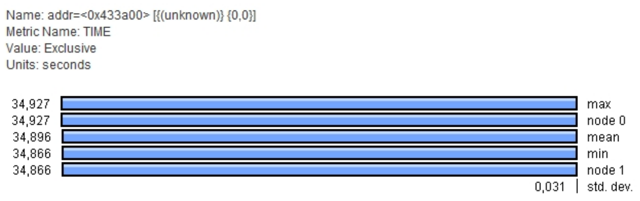
After the installation of TAU for MPI, the profiles for the 2 processors were generated. The text summary was visualized with the command pprof and the graphic information about the execution of the application with the command paraprof.

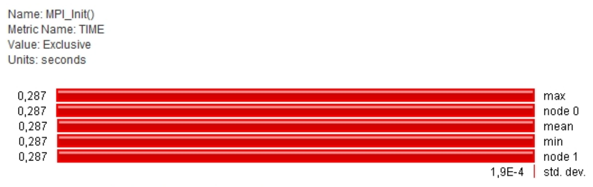


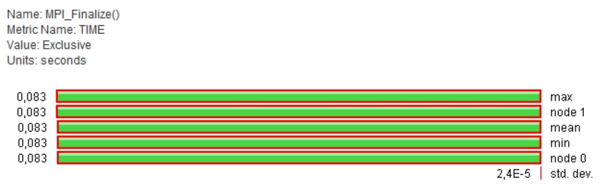
Each color represents an event executing on one or more processes.

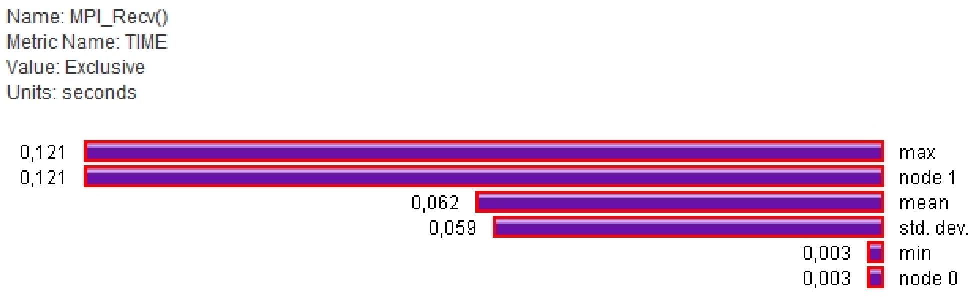
The blue color represents the access to the memory: addr=<0x433a00> [{(unknown)} {0,0}]. The red color represents the time needed for the initialization of the MPI: MPI_Init(), the green the time needed for the finalization of the MPI: MPI_Finalize() and the purple the time spend for the nodes to receive the information: MPI_Recv().

If the different events were seen separately, more information can be obtained.



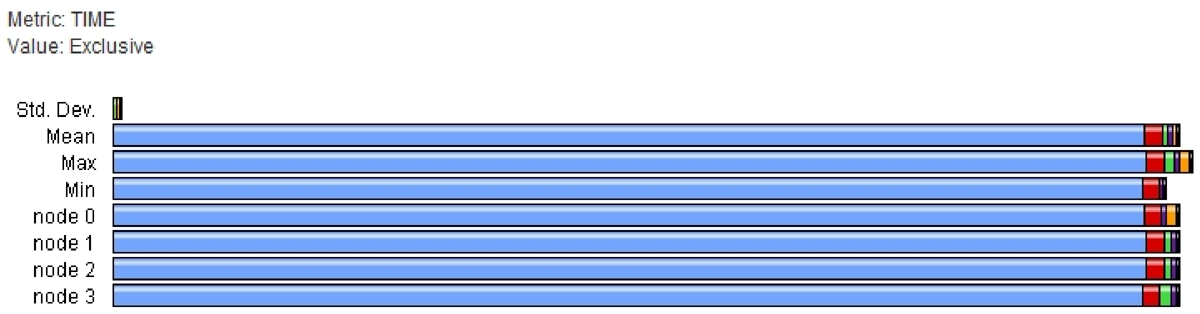




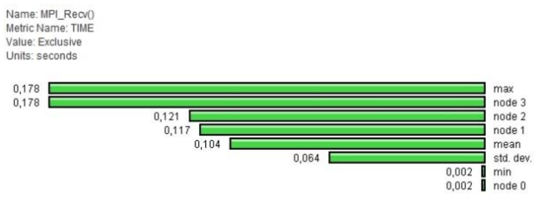


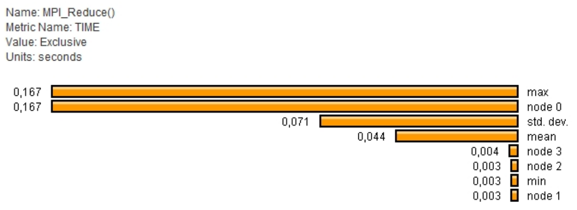
In both nodes the time spent in the memory access, the time for MPI_Init and MPI_Finalize is the same. On the other hand, the time spend in the MPI_Recv is larger in the second node, while in the root node is almost zero.

4 processes



In the case of four processes, the color orange is now appearing, which was not present previously. It is the MPI_Reduce, and it is present only in the zero node as can be seen bellow. This is expected, as the root processor is the one that stores the error value.





The time spent in MPI_Recv is almost zero in the zero node, and increases as the nodes have higher ranks. This could be explained by the fact that the communications take place in an order from the root processor to the rest ones.

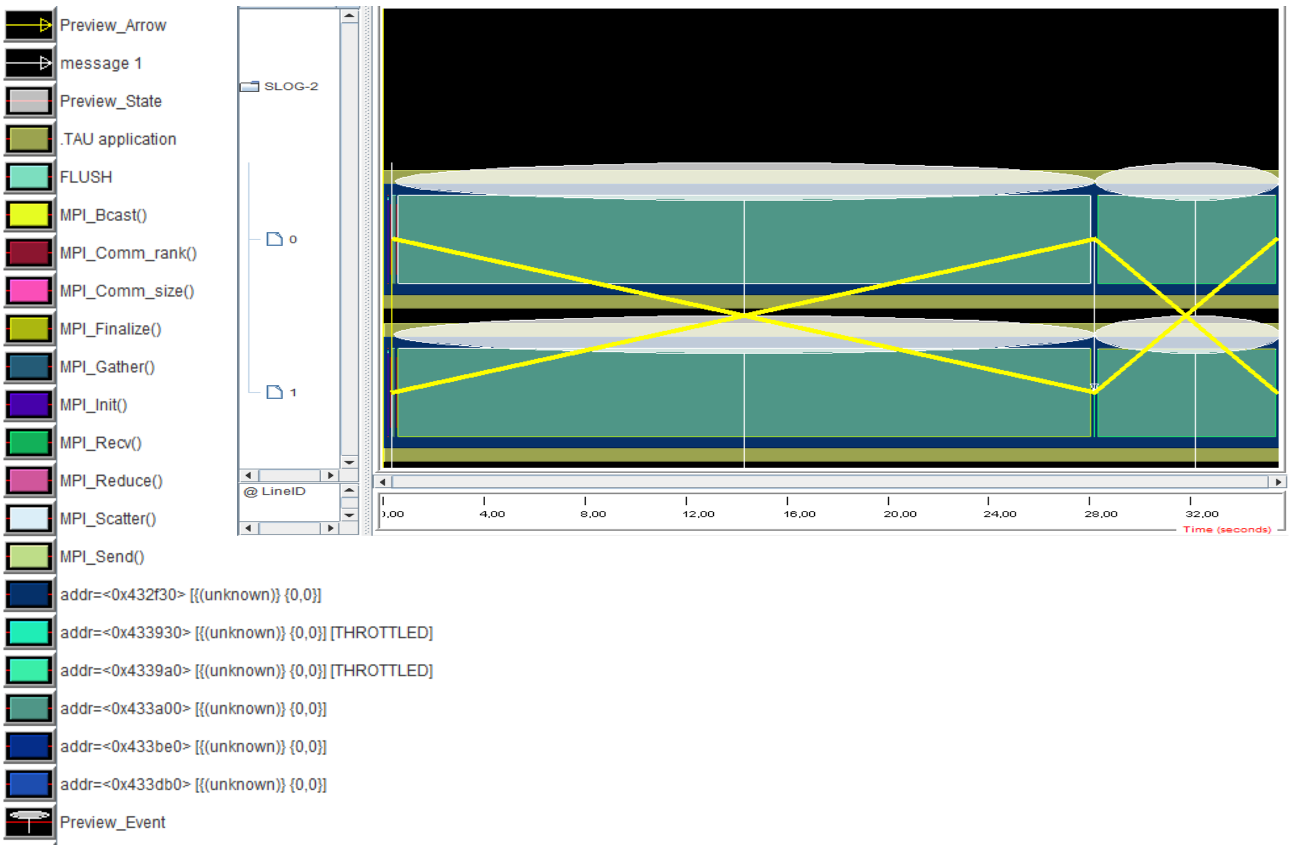
3.1.2 Tracing

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

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

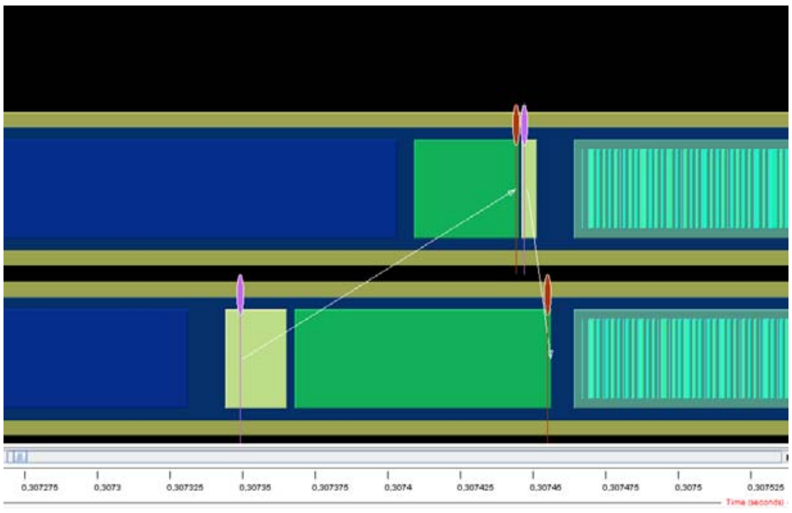
2 processes

First of all, the TAU application runs through the entire program on both nodes. Zooming in the first second of the execution can be seen that the MPI Initialization occurs at the first 0.00005 seconds until the end.

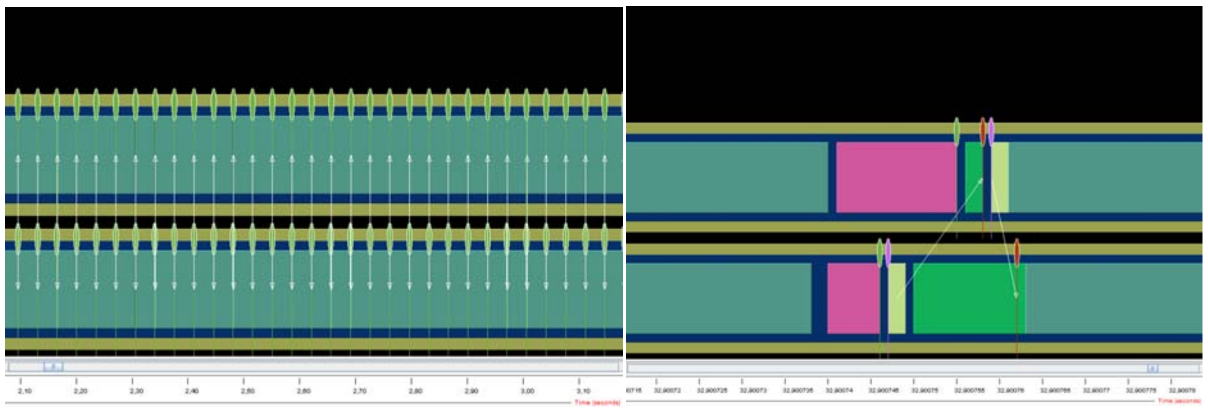


Preview_Arrow
message 1
Preview_State
.TAU application
FLUSH
MPI_Bcast()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Finalize()
MPI_Gather()
MPI_Init()
MPI_Recv()
MPI_Reduce()
MPI_Scatter()
MPI_Send()
addr=<0x432f30> [{(unknown)} {0,0}]
addr=<0x433930> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x4339a0> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x433a00> [{(unknown)} {0,0}]
addr=<0x433be0> [{(unknown)} {0,0}]
addr=<0x433db0> [{(unknown)} {0,0}]
Preview_Event

In approximately the 0,305 seconds, the MPI Scatter occurs, and afterwards the MPI Receive.

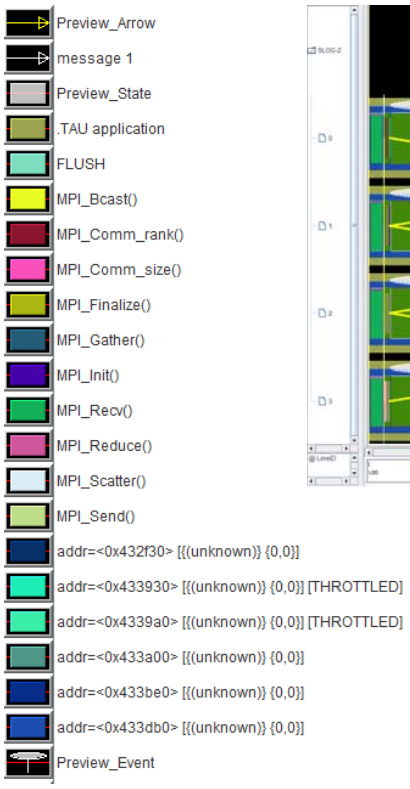


After this point and until the 0,4 seconds, both processes access the memory multiple times. Afterwards, a pattern like the following (zoomed out and in the communication points) continues until the end, where the MPI is Finalized:

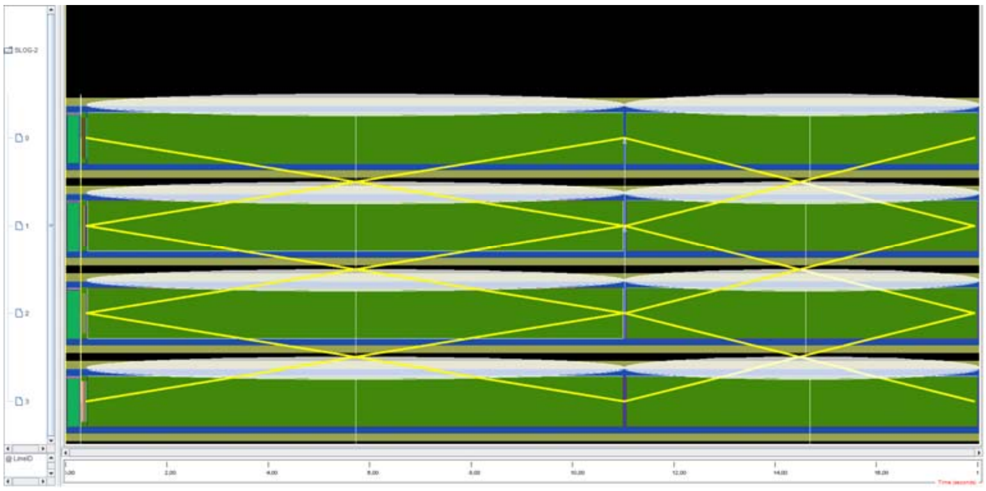


Where in the 2 nodes firstly the MPI Reduce occurs, and then they Send and Receive information.

<u>4 processes</u>

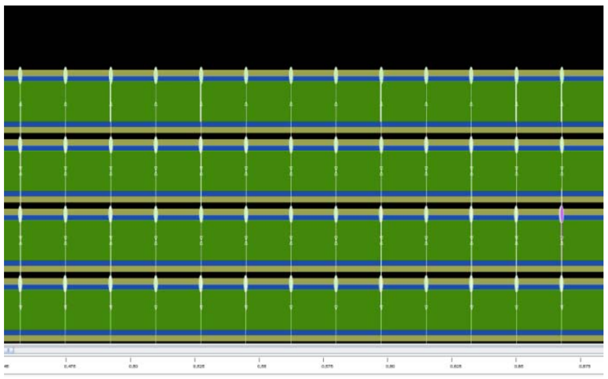
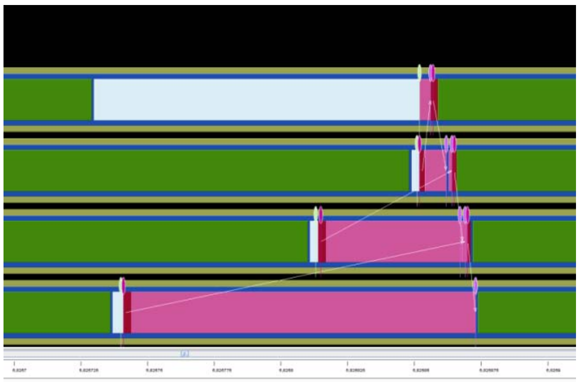


⬛ Preview_Arrow
⬛ message 1
⬜ Preview_State
🟫 .TAU application
🟩 FLUSH
🟨 MPI_Bcast()
🟥 MPI_Comm_rank()
🟪 MPI_Comm_size()
🟨 MPI_Finalize()
🟦 MPI_Gather()
🟪 MPI_Init()
🟩 MPI_Recv()
🟪 MPI_Reduce()
⬜ MPI_Scatter()
🟩 MPI_Send()
🟦 addr=<0x432f30> [{(unknown)} {0,0}]
🟩 addr=<0x433930> [{(unknown)} {0,0}] [THROTTLED]
🟩 addr=<0x4339a0> [{(unknown)} {0,0}] [THROTTLED]
🟩 addr=<0x433a00> [{(unknown)} {0,0}]
🟦 addr=<0x433be0> [{(unknown)} {0,0}]
🟦 addr=<0x433db0> [{(unknown)} {0,0}]
⬛ Preview_Event



When the program is executed with four processors, similar patterns appear as before. In the beginning the MPI Initialization occurs, followed by the MPI Scatter and Receive.

In the majority of the execution time, the following pattern appears, and after zooming in, it is shown that Reduce occurs mainly in the root process, and afterwards the processes Send and Receive information.

The above result is similar to the paraprof' s, where the time of MPI_Recv increases as the nodes have higher ranks.

3.2 Hybrid Solution

The hybrid parallel solution was executed with the -Ofast flag, and 4 threads.

<u>1000 x 1000</u>
With 2 processes: 0.687867 secs
With 4 processes: 0.517982 secs

<u>2000 x 2000</u>
With 2 processes: 4.576035 secs
With 4 processes: 4.105426 secs

<u>3000 x 3000</u>
With 2 processes: 9.281551 secs
With 4 processes: 9.694333 secs

<u>6000 x 6000</u>
With 2 processes: 16.144432 secs
With 4 processes: 16.213485 secs

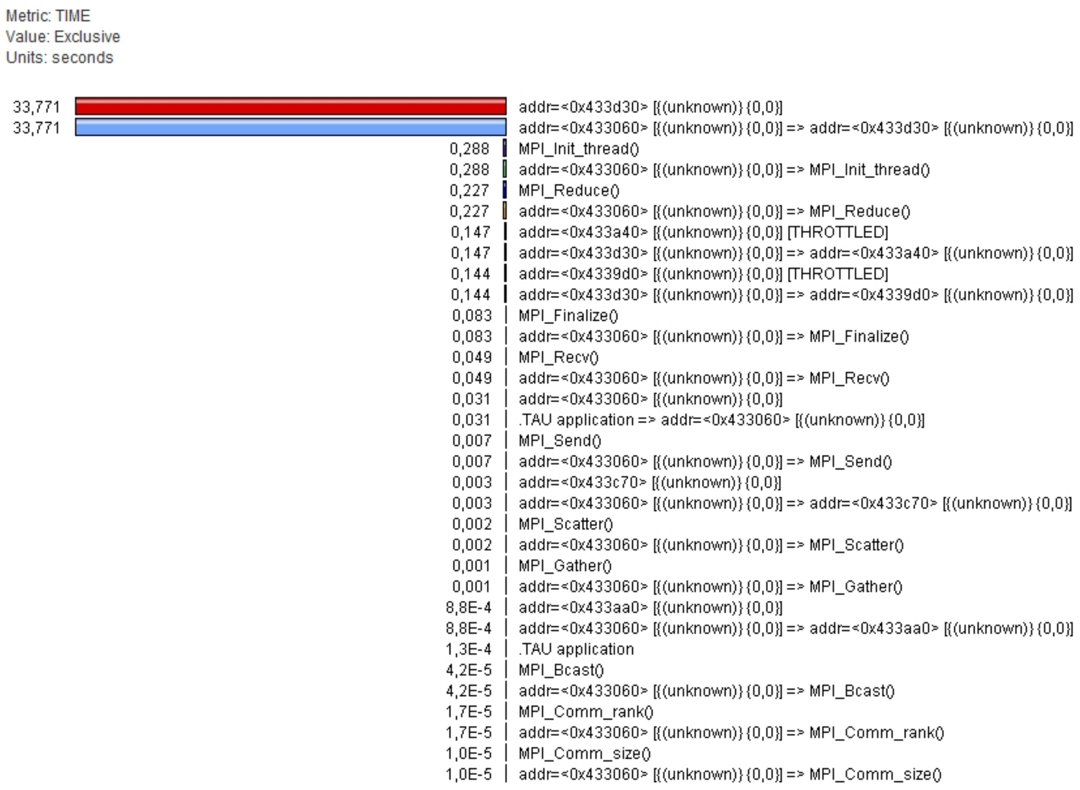
With small matrices, the increasing of the number of processes lowers the execution time, while for bigger matrices the time is increasing. Comparing the results with the basic solution, it can be said that the times are almost identical. As a first comment, can be assumed that the time for the creation and destruction of the threads is the reason for the results not to be faster.

During the installation of TAU, the TAU for hybrid MPI and OpenMP applications had not been configured. For this reason, the following TAU analysis will present each process, but not differently the threads of each process. As a result, the analysis will lack the events related to the parallelism of threads (parallel enter/exit, fork/join, begin/end, parallel for loop etc).

3.2.1 Profiling

<u>2 processes</u>

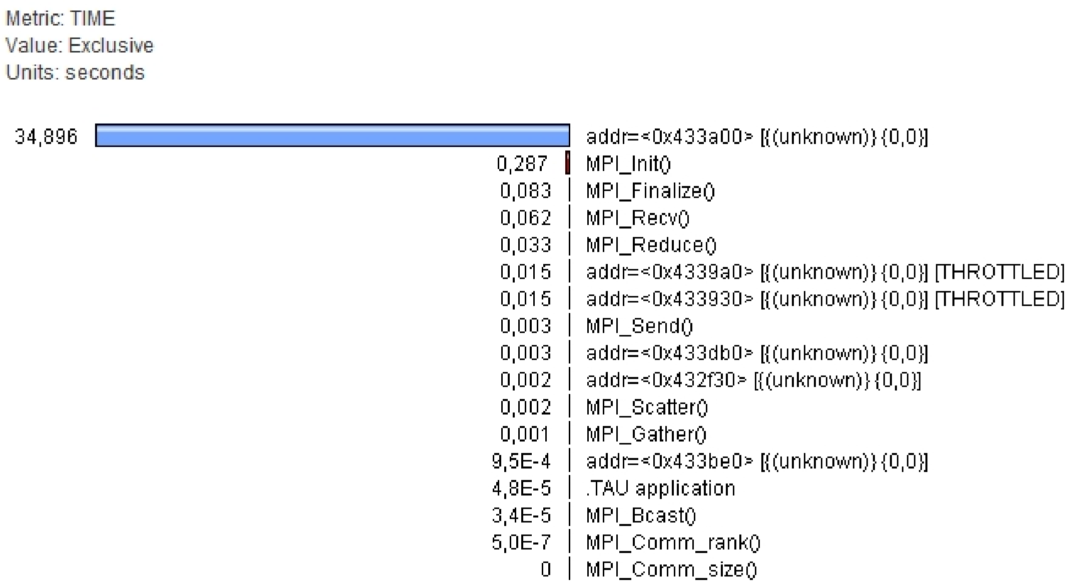
For the hybrid optimized solution, the profiles for the 2 processors were generated, and the graphic information about the execution of the application is presented below:



```
Metric: TIME
Value: Exclusive
Units: seconds

33,771  addr=<0x433d30> [{(unknown)} {0,0}]
33,771  addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433d30> [{(unknown)} {0,0}]
 0,288  MPI_Init_thread()
 0,288  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Init_thread()
 0,227  MPI_Reduce()
 0,227  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Reduce()
 0,147  addr=<0x433a40> [{(unknown)} {0,0}] [THROTTLED]
 0,147  addr=<0x433d30> [{(unknown)} {0,0}] => addr=<0x433a40> [{(unknown)} {0,0}]
 0,144  addr=<0x4339d0> [{(unknown)} {0,0}] [THROTTLED]
 0,144  addr=<0x433d30> [{(unknown)} {0,0}] => addr=<0x4339d0> [{(unknown)} {0,0}]
 0,083  MPI_Finalize()
 0,083  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Finalize()
 0,049  MPI_Recv()
 0,049  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Recv()
 0,031  addr=<0x433060> [{(unknown)} {0,0}]
 0,031  .TAU application => addr=<0x433060> [{(unknown)} {0,0}]
 0,007  MPI_Send()
 0,007  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Send()
 0,003  addr=<0x433c70> [{(unknown)} {0,0}]
 0,003  addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433c70> [{(unknown)} {0,0}]
 0,002  MPI_Scatter()
 0,002  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Scatter()
 0,001  MPI_Gather()
 0,001  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Gather()
8,8E-4  addr=<0x433aa0> [{(unknown)} {0,0}]
8,8E-4  addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433aa0> [{(unknown)} {0,0}]
1,3E-4  .TAU application
4,2E-5  MPI_Bcast()
4,2E-5  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Bcast()
1,7E-5  MPI_Comm_rank()
1,7E-5  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Comm_rank()
1,0E-5  MPI_Comm_size()
1,0E-5  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Comm_size()
```

And comparing it with the basic solution:



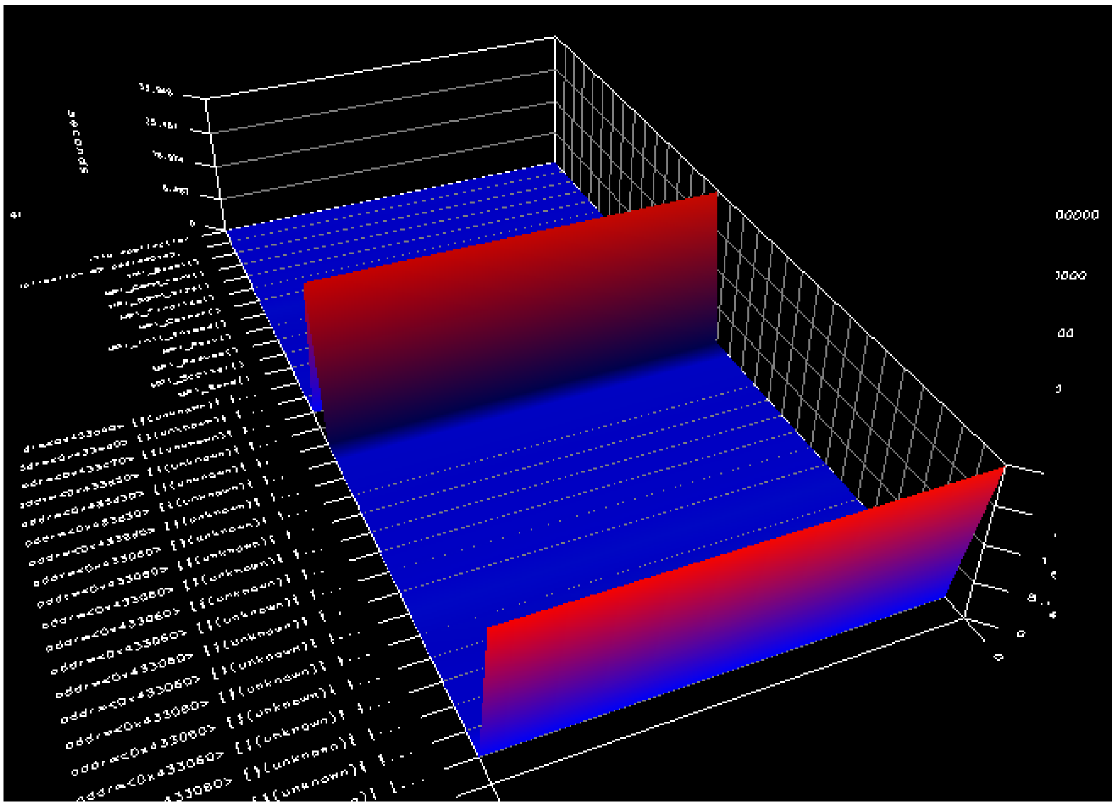
```
Metric: TIME
Value: Exclusive
Units: seconds

34,896  addr=<0x433a00> [{(unknown)} {0,0}]
 0,287  MPI_Init()
 0,083  MPI_Finalize()
 0,062  MPI_Recv()
 0,033  MPI_Reduce()
 0,015  addr=<0x4339a0> [{(unknown)} {0,0}] [THROTTLED]
 0,015  addr=<0x433930> [{(unknown)} {0,0}] [THROTTLED]
 0,003  MPI_Send()
 0,003  addr=<0x433db0> [{(unknown)} {0,0}]
 0,002  addr=<0x432f30> [{(unknown)} {0,0}]
 0,002  MPI_Scatter()
 0,001  MPI_Gather()
9,5E-4  addr=<0x433be0> [{(unknown)} {0,0}]
4,8E-5  .TAU application
3,4E-5  MPI_Bcast()
5,0E-7  MPI_Comm_rank()
     0  MPI_Comm_size()
```

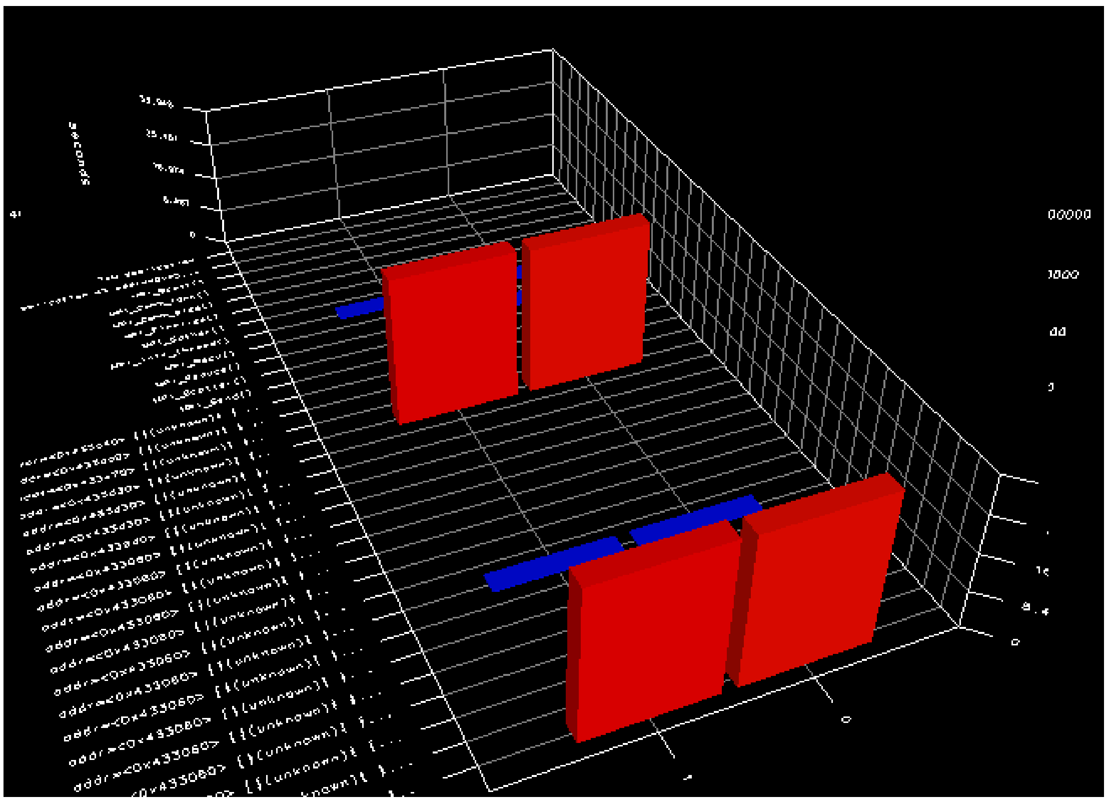
The time spend on the events related to parallelism are almost identical for both versions, and the time spent in the main event is less in the hybrid version.

Following presents the 3D visualizations for 2 processes respectively:

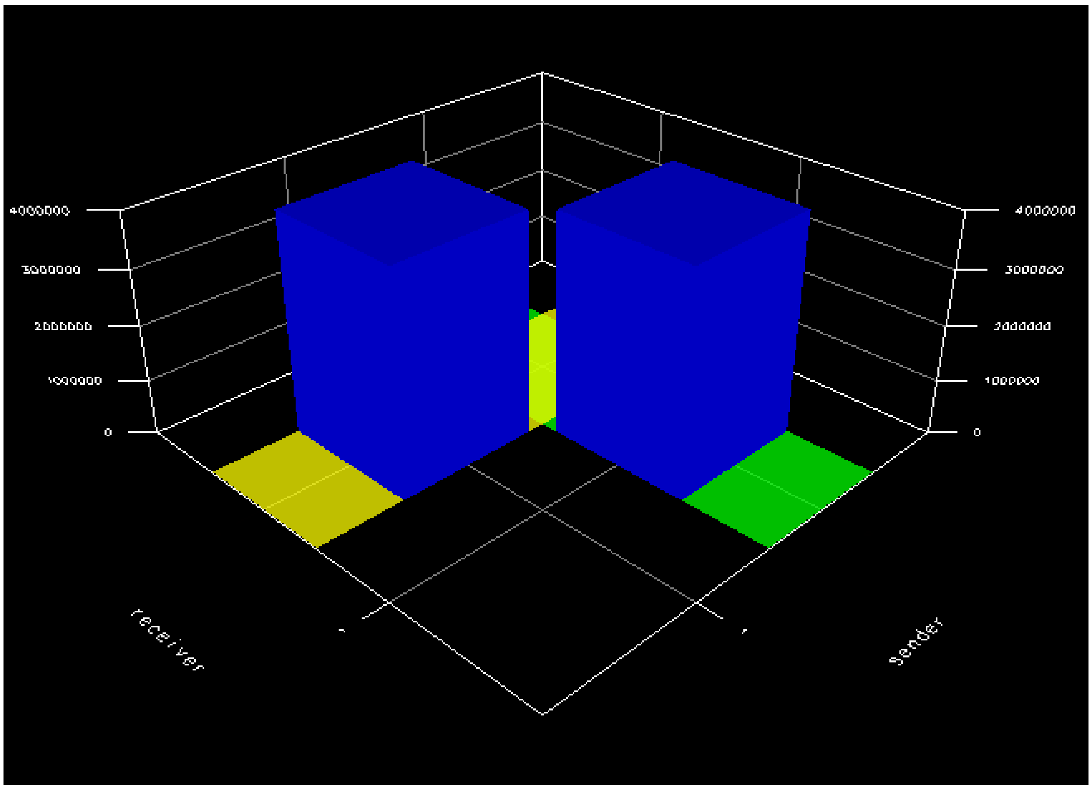
1) Triangle Mesh



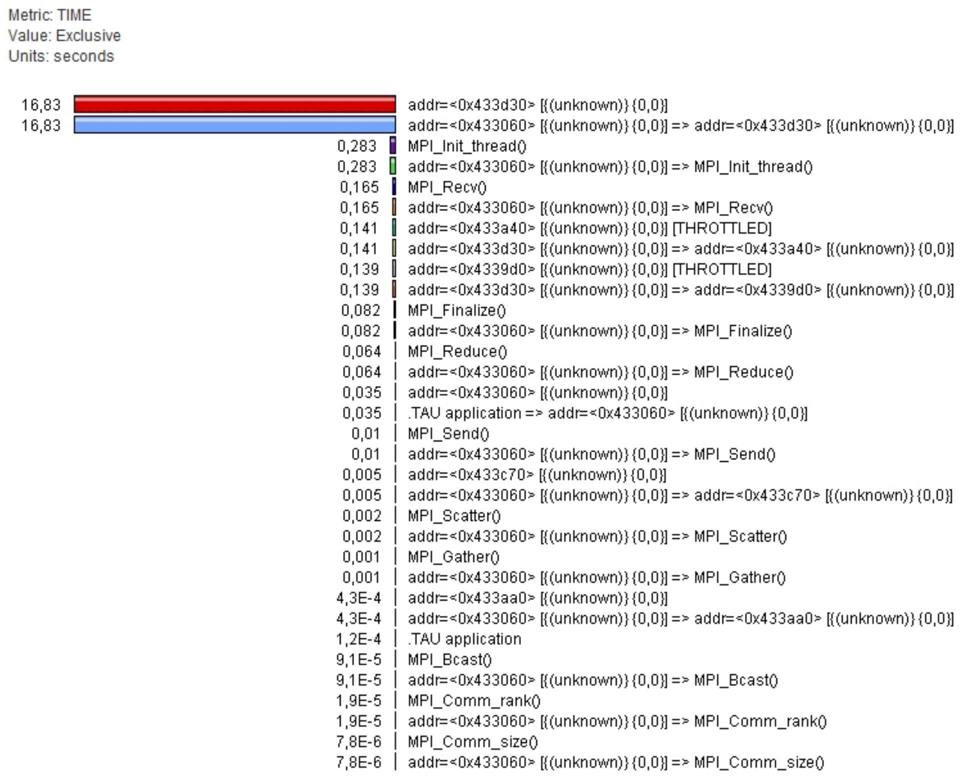
2) Bar Plot



3) 3D Communication Matrix



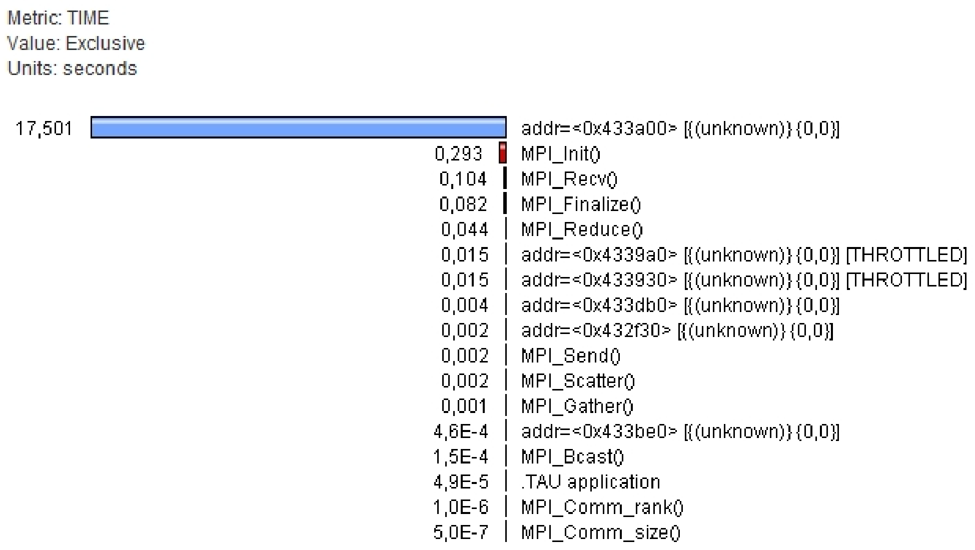
4 processes



```
Metric: TIME
Value: Exclusive
Units: seconds

16,83   addr=<0x433d30> [{(unknown)} {0,0}]
16,83   addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433d30> [{(unknown)} {0,0}]
0,283   MPI_Init_thread()
0,283   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Init_thread()
0,165   MPI_Recv()
0,165   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Recv()
0,141   addr=<0x433a40> [{(unknown)} {0,0}] [THROTTLED]
0,141   addr=<0x433d30> [{(unknown)} {0,0}] => addr=<0x433a40> [{(unknown)} {0,0}]
0,139   addr=<0x4339d0> [{(unknown)} {0,0}] [THROTTLED]
0,139   addr=<0x433d30> [{(unknown)} {0,0}] => addr=<0x4339d0> [{(unknown)} {0,0}]
0,082   MPI_Finalize()
0,082   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Finalize()
0,064   MPI_Reduce()
0,064   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Reduce()
0,035   addr=<0x433060> [{(unknown)} {0,0}]
0,035   .TAU application => addr=<0x433060> [{(unknown)} {0,0}]
0,01    MPI_Send()
0,01    addr=<0x433060> [{(unknown)} {0,0}] => MPI_Send()
0,005   addr=<0x433c70> [{(unknown)} {0,0}]
0,005   addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433c70> [{(unknown)} {0,0}]
0,002   MPI_Scatter()
0,002   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Scatter()
0,001   MPI_Gather()
0,001   addr=<0x433060> [{(unknown)} {0,0}] => MPI_Gather()
4,3E-4  addr=<0x433aa0> [{(unknown)} {0,0}]
4,3E-4  addr=<0x433060> [{(unknown)} {0,0}] => addr=<0x433aa0> [{(unknown)} {0,0}]
1,2E-4  .TAU application
9,1E-5  MPI_Bcast()
9,1E-5  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Bcast()
1,9E-5  MPI_Comm_rank()
1,9E-5  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Comm_rank()
7,8E-6  MPI_Comm_size()
7,8E-6  addr=<0x433060> [{(unknown)} {0,0}] => MPI_Comm_size()
```

And comparing it with the basic solution:



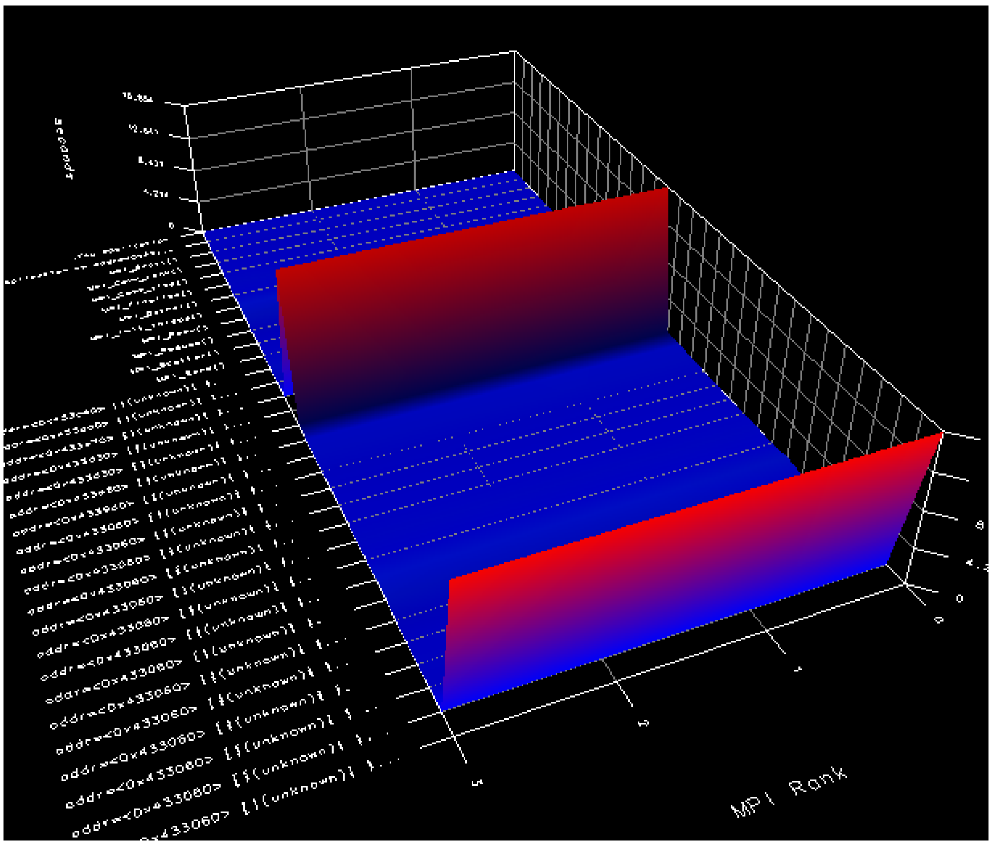
```
Metric: TIME
Value: Exclusive
Units: seconds

17,501   addr=<0x433a00> [{(unknown)} {0,0}]
 0,293   MPI_Init()
 0,104   MPI_Recv()
 0,082   MPI_Finalize()
 0,044   MPI_Reduce()
 0,015   addr=<0x4339a0> [{(unknown)} {0,0}] [THROTTLED]
 0,015   addr=<0x433930> [{(unknown)} {0,0}] [THROTTLED]
 0,004   addr=<0x433db0> [{(unknown)} {0,0}]
 0,002   addr=<0x432f30> [{(unknown)} {0,0}]
 0,002   MPI_Send()
 0,002   MPI_Scatter()
 0,001   MPI_Gather()
4,6E-4   addr=<0x433be0> [{(unknown)} {0,0}]
1,5E-4   MPI_Bcast()
4,9E-5   .TAU application
1,0E-6   MPI_Comm_rank()
5,0E-7   MPI_Comm_size()
```

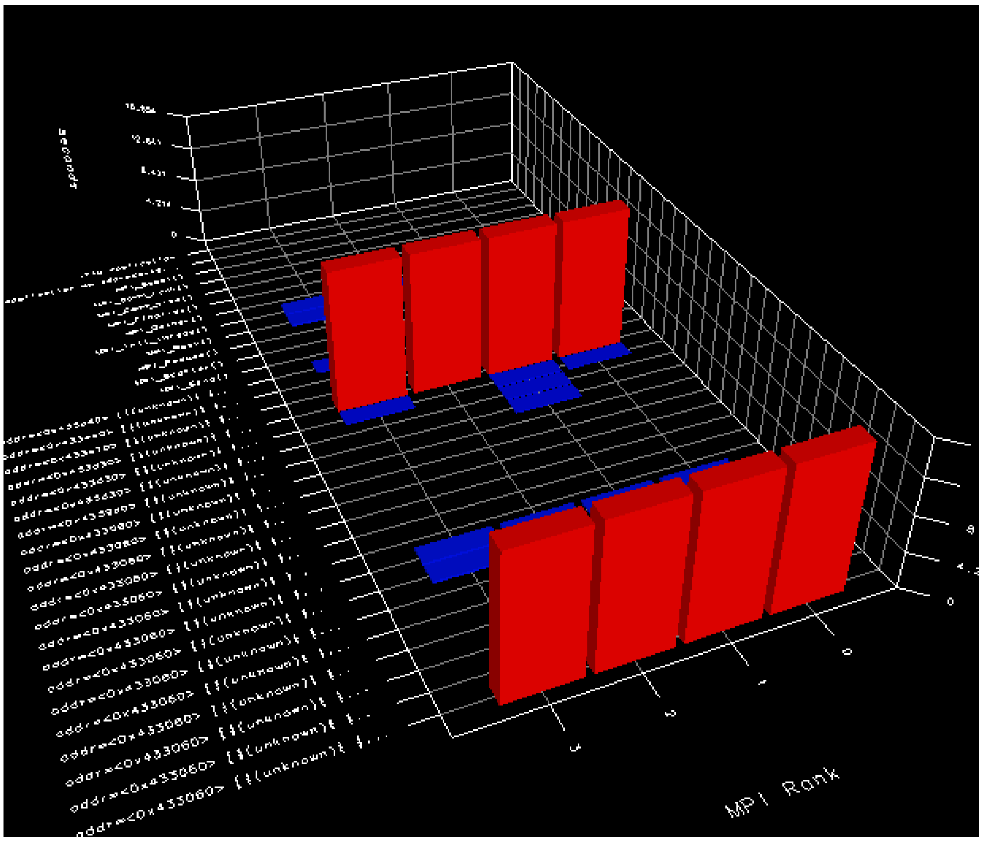
Similarly to the 2 processes, when the program is executed with 4 threads the seconds spend in the main event are less.

Following presents the 3D visualizations for 4 processes respectively:

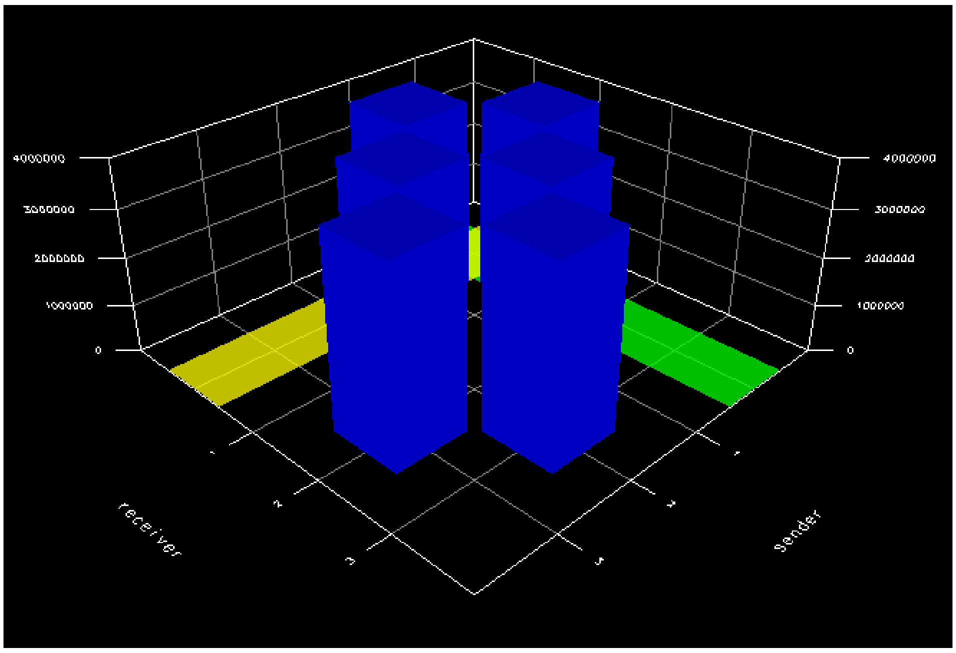
1) Triangle Mesh



2) Bar Plot



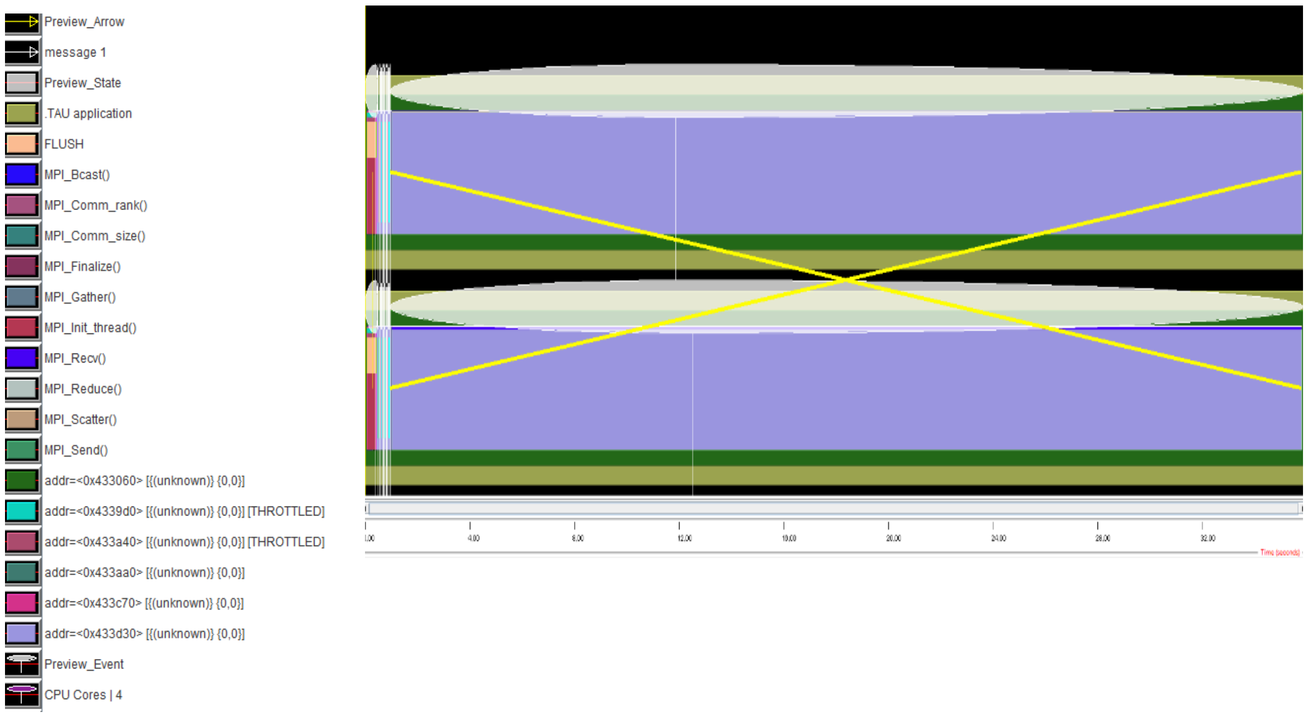
3) 3D Communication Matrix



3.2.2 Tracing

The results from the jumpshot for 2 processes and 4 threads are being presented below:

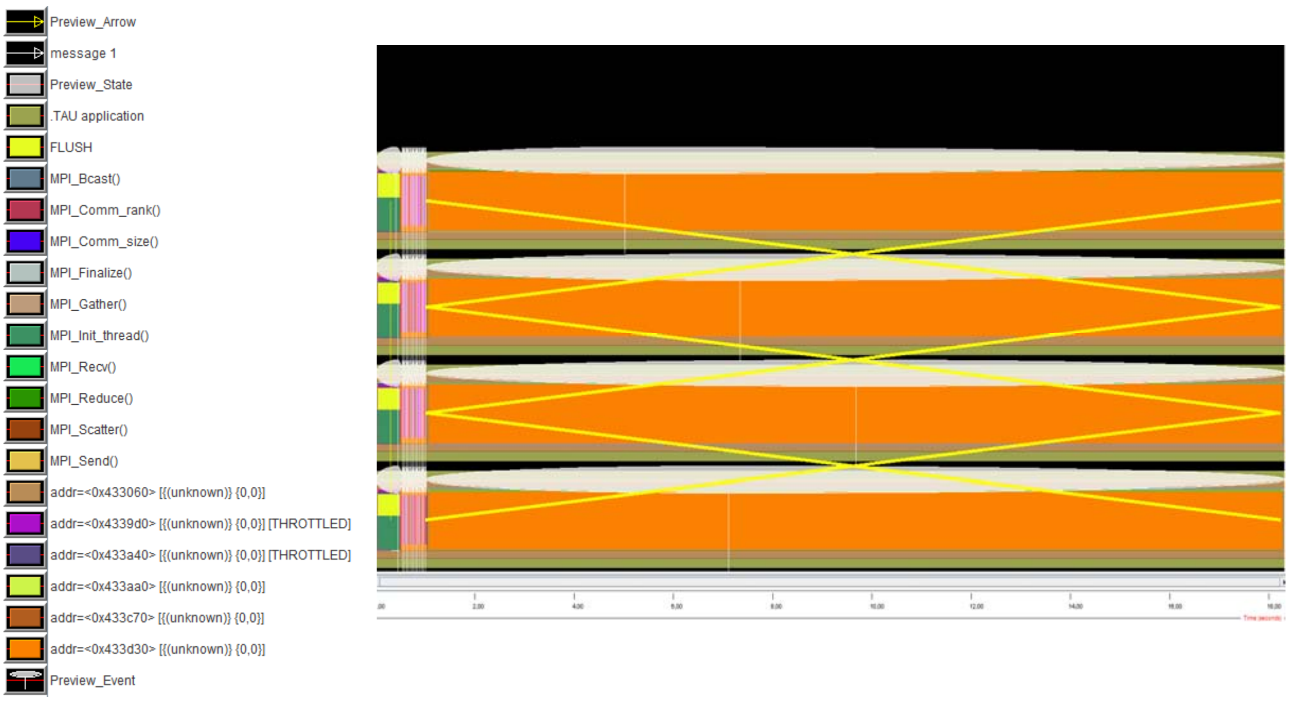
2 processes



Preview_Arrow
message 1
Preview_State
.TAU application
FLUSH
MPI_Bcast()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Finalize()
MPI_Gather()
MPI_Init_thread()
MPI_Recv()
MPI_Reduce()
MPI_Scatter()
MPI_Send()
addr=<0x433060> [{(unknown)} {0,0}]
addr=<0x4339d0> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x433a40> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x433aa0> [{(unknown)} {0,0}]
addr=<0x433c70> [{(unknown)} {0,0}]
addr=<0x433d30> [{(unknown)} {0,0}]
Preview_Event
CPU Cores | 4

Unfortunately, it is not possible to see all different threads in jumpshot. Nevertheless, it can be observed that in the beginning, while the Initialization time of the hybrid MPI and OpenMP had almost the same time in the base version (0,3secs), the program started its monotonous pattern (reduce, send and receive) in approximately 1 sec, opposed to the 0,4 secs of the basic program; that again can be explained by the creation of the threads.

4 processes



Preview_Arrow
message 1
Preview_State
.TAU application
FLUSH
MPI_Bcast()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Finalize()
MPI_Gather()
MPI_Init_thread()
MPI_Recv()
MPI_Reduce()
MPI_Scatter()
MPI_Send()
addr=<0x433060> [{(unknown)} {0,0}]
addr=<0x4339d0> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x433a40> [{(unknown)} {0,0}] [THROTTLED]
addr=<0x433aa0> [{(unknown)} {0,0}]
addr=<0x433c70> [{(unknown)} {0,0}]
addr=<0x433d30> [{(unknown)} {0,0}]
Preview_Event

Similarly as before, the program starts its continuous pattern, which is allegedly the Laplace step function, beginning after 1 second opposed to the 0,3 secs of the basic MPI program.

4. Results

The MPI implementation of the Laplace program was succeeded by distributing the work within numerous processors and adding the necessary communications. Even though the programming complexity is increasing significantly, it is obvious the importance of parallel programming is in the execution time.

To improve the performance of the program, the Hybrid MPI and OpenMP was chosen, in order for the work to be distributed even more among the threads of the processors. This is because of the execution time not decreasing. If the goal is less execution time then it is recommended to use another optimization mechanism, which maybe would be more suitable.

On the other hand, because the work is distributed more, it can be assumed that the cache misses will be less while the tasks per clock will be more, and as a result the performance of the of the Hybrid MPI + OpenMP will be better. The performance could be checked more with the separate analysis of the threads, and most importantly with the use of an MPI performance measuring the program.

5. Bibliography

1. Notes for Parallel Programming with MPI, Modelling for Science and Engineering, Autonomous University of Barcelona
2. https://techdifferences.com/difference-between-synchronous-and-asynchronous-transmission.html
3. www.engr.iupui.edu/~skoskie/ECE362/lecture_notes/LNB25_html/text12.html
4. www.training.prace-ri.eu/uploads/tx_pracetmo/hybridProgramming_04.pdf