# MPI Exercises

Vicent Santamarta Martinez

## 1. hello_world.c

When running the code, without specifying the number of cores, we can see how we can read in the terminal as many “Hello World” as cores we have in our processor. In this case, as in the previous task, the threads don’t finish in any specific order.

```
(py38) (base) vicentamen@ventagram:~/Documents/Intelligent Systems/SKJ012-HighPerformanceML/2_3_MPI_Exercises$ mpirun ./hello_world.out
Hello World!!, from rank 3 out of 12
Hello World!!, from rank 8 out of 12
Hello World!!, from rank 9 out of 12
Hello World!!, from rank 10 out of 12
Hello World!!, from rank 11 out of 12
Hello World!!, from rank 1 out of 12
Hello World!!, from rank 2 out of 12
Hello World!!, from rank 4 out of 12
Hello World!!, from rank 7 out of 12
Hello World!!, from rank 0 out of 12
Hello World!!, from rank 5 out of 12
Hello World!!, from rank 6 out of 12
(py38) (base) vicentamen@ventagram:~/Documents/Intelligent Systems/SKJ012-HighPerformanceML/2_3_MPI_Exercises$
```

## 2. pi_par_p2p.c

| Processes | Rectangles | Sp | Ep |
|---|---|---|---|
| 1 | 1.00E+07 | 0.927684297 | 0.927684297 |
| 1 | 5.00E+07 | 1.005513043 | 1.005513043 |
| 1 | 1.00E+08 | 0.998311377 | 0.998311377 |
| 1 | 5.00E+08 | 1.000563772 | 1.000563772 |
| 2 | 1.00E+07 | 1.996237734 | 0.998118867 |
| 2 | 5.00E+07 | 1.81928006 | 0.909640029 |
| 2 | 1.00E+08 | 1.976783479 | 0.988391739 |
| 2 | 5.00E+08 | 2.007644124 | 1.003822062 |
| 4 | 1.00E+07 | 4.009198818 | 1.002299704 |
| 4 | 5.00E+07 | 3.819441158 | 0.954860289 |
| 4 | 1.00E+08 | 2.880652181 | 0.720163045 |
| 4 | 5.00E+08 | 3.903675267 | 0.975918817 |
| 8 | 1.00E+07 | 7.898333067 | 0.987291633 |
| 8 | 5.00E+07 | 6.505968947 | 0.813246118 |
| 8 | 1.00E+08 | 3.832501602 | 0.4790627 |
| 8 | 5.00E+08 | 8.004809574 | 1.000601197 |

This table shows the results for each run of the program pi_par_p2p.c in my machine. From the results we can see for each new thread added the speed ups value increases at the same rate, which means that for each new thread added the time taken to complete the task is greatly decreased, keeping an efficiency close to 1 through almost all executions. And this results are worse than they should be, since each thread is processing all of the rectangles

## 3. pi_par_col.c

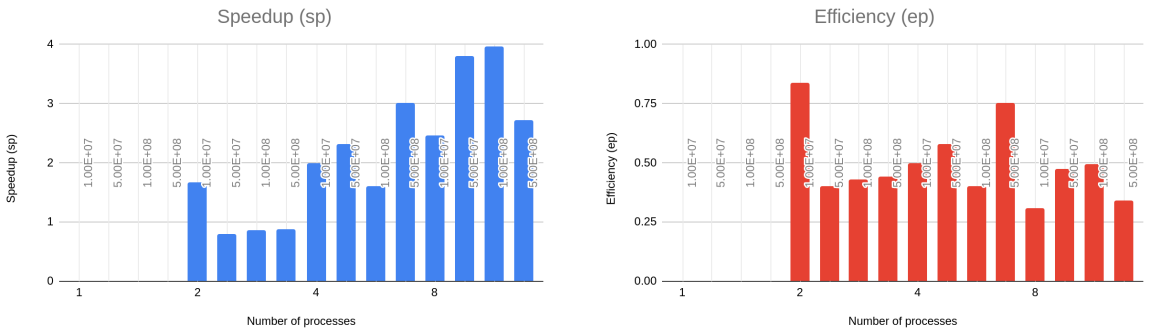
| Processes | Rectangles | Speedup | Efficiency |
|---|---|---|---|
| 1 | 1.00E+07 | 0.98902253 | 0.98902253 |
| 1 | 5.00E+07 | 0.99834641 | 0.99834641 |
| 1 | 1.00E+08 | 0.99741162 | 0.99741162 |
| 1 | 5.00E+08 | 0.99500038 | 0.99500038 |
| 2 | 1.00E+07 | 1.80635695 | 0.90317847 |
| 2 | 5.00E+07 | 1.87897658 | 0.93948829 |
| 2 | 1.00E+08 | 1.98005382 | 0.99002691 |
| 2 | 5.00E+08 | 2.00297211 | 1.00148606 |
| 4 | 1.00E+07 | 3.99931968 | 0.99982992 |
| 4 | 5.00E+07 | 3.88895179 | 0.97223795 |
| 4 | 1.00E+08 | 3.8468762 | 0.96171905 |
| 4 | 5.00E+08 | 3.82534244 | 0.95633561 |
| 8 | 1.00E+07 | 4.08301977 | 0.51037747 |
| 8 | 5.00E+07 | 7.96891348 | 0.99611418 |
| 8 | 1.00E+08 | 7.91262145 | 0.98907768 |
| 8 | 5.00E+08 | 4.11724871 | 0.51465609 |

From the speed and Efficiency columns, we can see how this method keeps with an increasing speedup value as the number of threads increases, which means that we are getting better performance on the task than with the sequential method. One particular thing when implementing the reduce for this exercise was that I needed to put a barrier before the reduce, so every process would finish their task before calculating the value of pi, otherwise I would get incorrect values whenever the process 0 was not the last to finish.

## 4. pi_par_m2s

| Processes | Rectangles | Speedup | Efficiency |
|---|---|---|---|
| 1 | 1.00E+07 | 0 | 0 |
| 1 | 5.00E+07 | 0 | 0 |
| 1 | 1.00E+08 | 0 | 0 |
| 1 | 5.00E+08 | 0 | 0 |

| | | | |
|---|---|---|---|
| 2 | 1.00E+07 | 1.67914561 | 0.8395728 |
| 2 | 5.00E+07 | 0.80466472 | 0.40233236 |
| 2 | 1.00E+08 | 0.86034206 | 0.43017103 |
| 2 | 5.00E+08 | 0.88448361 | 0.44224181 |
| 4 | 1.00E+07 | 2.00129097 | 0.50032274 |
| 4 | 5.00E+07 | 2.3147903 | 0.57869758 |
| 4 | 1.00E+08 | 1.60048679 | 0.4001217 |
| 4 | 5.00E+08 | 3.00798917 | 0.75199729 |
| 8 | 1.00E+07 | 2.4629147 | 0.30786434 |
| 8 | 5.00E+07 | 3.80559721 | 0.47569965 |
| 8 | 1.00E+08 | 3.96179097 | 0.49522387 |
| 8 | 5.00E+08 | 2.71891415 | 0.33986427 |




The first thing that we notice from executing the program is that we need at least 2 processes to be able to work with master/worker structure, otherwise as we see when we use only 1 process, the master will stay on waiting since there is no worker that will make the calculations for him. The next thing that we notice is that though the speedup has an incremental tendency, the efficiency stays more or less stable throughout the executions.

## 5. mat_vec_col.c

| Processes | M | N | sp | ep | sp_rec |
|---|---|---|---|---|---|
| 1 | 80 | 80 | 1.019648328 | 1.019648328 | 0.7432756109 |
| 1 | 800 | 800 | - | - | - |
| 1 | 8000 | 8000 | - | - | - |
| 2 | 80 | 80 | 1.208312847 | 0.6041564237 | 0.4036716953 |
| 2 | 800 | 800 | 1.953953659 | 0.9769768296 | 1.103721743 |
| 2 | 8000 | 8000 | - | - | - |
| 4 | 80 | 80 | 1.698660463 | 0.4246651158 | 0.3378937155 |
| 4 | 800 | 800 | 3.3468198 | 0.8367049499 | 1.349166019 |

| | | | | | |
|---|---|---|---|---|---|
| 4 | 8000 | 8000 | - | - | - |
| 8 | 80 | 80 | 0.8006379968 | 0.1000797496 | 0.1744764035 |
| 8 | 800 | 800 | 8.386671362 | 1.04833392 | 4.318263452 |
| 8 | 8000 | 8000 | - | - | - |

From the results for this case we can see that in both cases the speed up for the parallel and all the operations increases as we increase the number of threads, having a decent efficiency value. For the case of considering all the operations, we see that it has a worse performance than just the parallel version.