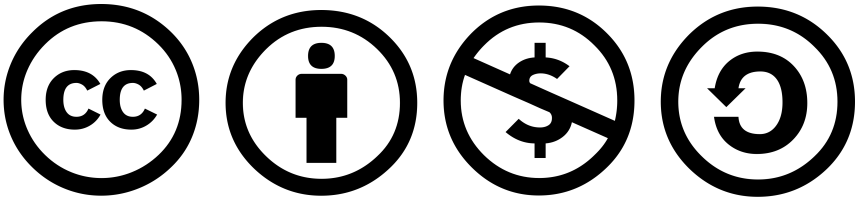


Report Common Assignment MPI01

Matrix dot Matrix

Lecturer: Francesco Moscato - fmoscato@unisa.it

Group:

- **Capitani Giuseppe 0622701085 g.capitani@studenti.unisa.it**
- **Falanga Armando 0622701140 a.falanga13@studenti.unisa.it**
- **Terrone Luigi 0622701071 l.terrone2@studenti.unisa.it**

Index

Problem description

Parallelize and Evaluate Performances of "Matrix dot Matrix" Algorithm , by using MPI.

Matrix multiplication to use is the "row - column" multiplication, where elements or result matrix is obtained by scalar multiplication of one row of the first Matrix for one column of the second Matrix.

Use MPI DataTypes to model row and column types.

Evaluate performances when reading files with different MPI I/O approaches.

Solution

We implemented an algorithm that evaluates different MPI I/O approaches, referred as Versions, according to what was asked. In this report, only version 1 and 4 are shown since no relevant changes across other versions were observed.

Version 1

Matrix A is read from file in an ordered manner (i.e. using *MPI_File_read_ordered*).
Matrix B is read from file too, but its transposed has been previously written to the corresponding file. This means that the program only has to read matrix B contiguously.

Version 4

Matrix A is read from file with a file view and a displacement computed using processes ranks and *MPI_File_read.*
Matrix B is read as in Version 1.

Version 2

Matrix A is read from file using *MPI_File_seek* and *MPI_File_read_all* using a displacement computed using processes ranks.
Matrix B is read as in Version 1

Version 3

Matrix A is read from file with a file view and *MPI_File_read_all* using a displacement computed using processes ranks.
Matrix B is read as in Version 1

Version 5

Matrix A is read from file with a file view and a displacement computed using processes ranks.
Matrix B is read from a file using an MPI Datatype to read the matrix columns. The matrix B file does not contain Matrix B transposed.

Version 6

Matrix A is read from file in an ordered manner (i.e. using *MPI_File_read_ordered*).
Matrix B is read while computing the matrix product, each process reads a column and computes the partial product.

Experimental setup

Hardware

CPU

```
processors       : 8
vendor_id        : GenuineIntel
cpu family       : 6
model            : 142
model name       : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
stepping         : 10
microcode        : 0xe0
cpu MHz          : 800.097
cache size       : 8192 KB
physical id      : 0
siblings         : 8
core id          : 0
cpu cores        : 4
apicid           : 0
initial apicid   : 0
fpu              : yes
fpu_exception    : yes
cpuid level      : 22
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon
pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16
xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb
stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase
tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap
clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat
pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d
vmx flags  : vnmi preemption_timer invvpid ept_x_only ept_ad
ept_1gb flexpriority tsc_offset vtpr mtf vapic ept vpid
unrestricted_guest ple pml ept_mode_based_exec
```

```
bugs            : cpu_meltdown spectre_v1 spectre_v2
spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips        : 3999.93
clflush size    : 64
cache alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
```

RAM

```
Memory Device
        Array Handle: 0x0008
        Error Information Handle: Not Provided
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 8192 MB
        Form Factor: SODIMM
        Set: None
        Locator: ChannelA-DIMM0
        Bank Locator: BANK 0
        Type: DDR4
        Type Detail: Synchronous Unbuffered (Unregistered)
        Speed: 2400 MT/s
        Manufacturer: SK Hynix
        Serial Number: 51DC466C
        Asset Tag: 9876543210
        Part Number: HMA81GS6AFR8N-UH
        Rank: 1
        Configured Memory Speed: 2400 MT/s
        Minimum Voltage: 1.2 V
        Maximum Voltage: 1.2 V
        Configured Voltage: 1.2 V

Handle 0x000B, DMI type 17, 40 bytes
Memory Device
        Array Handle: 0x0008
        Error Information Handle: Not Provided
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 8192 MB
        Form Factor: SODIMM
        Set: None
```

```
Locator: ChannelB-DIMM0
Bank Locator: BANK 2
Type: DDR4
Type Detail: Synchronous Unbuffered (Unregistered)
Speed: 2400 MT/s
Manufacturer: SK Hynix
Serial Number: 51DC45B3
Asset Tag: 9876543210
Part Number: HMA81GS6AFR8N-UH
Rank: 1
Configured Memory Speed: 2400 MT/s
Minimum Voltage: 1.2 V
Maximum Voltage: 1.2 V
Configured Voltage: 1.2 V
```

Software

- Ubuntu 20.04 LTS
- GCC 9
- Swap : 2147479552 bytes ~ 2GB

Performance, Speedup & Efficiency

Case n° 1

In this case study, the main purpose was to analyze the performance of our program in the following build setup:

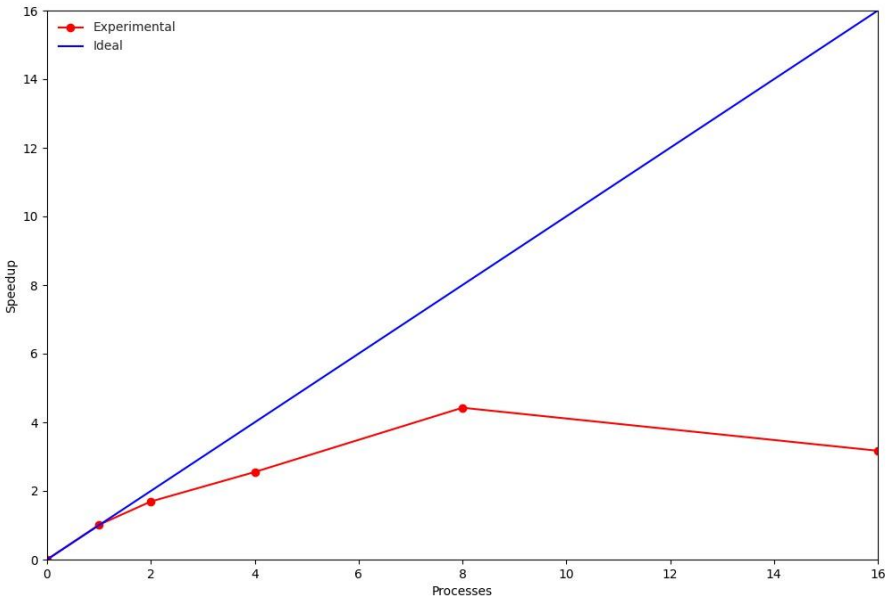
- The sequential program is compiled without any gcc optimization
- The parallel programs are compiled with the gcc optimization -Ox where x = 3

So here we want to highlight the difference between a simple sequential program compared to a parallel one, furthermore the case study is done on multiple sizes that are 1250, 2000, 5000 and with different number of processes (0, 1, 2, 4, 8, 16). In addition, there are many versions of the read of matrix A and B in the files, but we focused on version 1 and version 4. These are the two versions whose results were the most interesting to analyze. In the end, the matrix B was written column wise in order to improve the performance of the dot product.

Size-1250

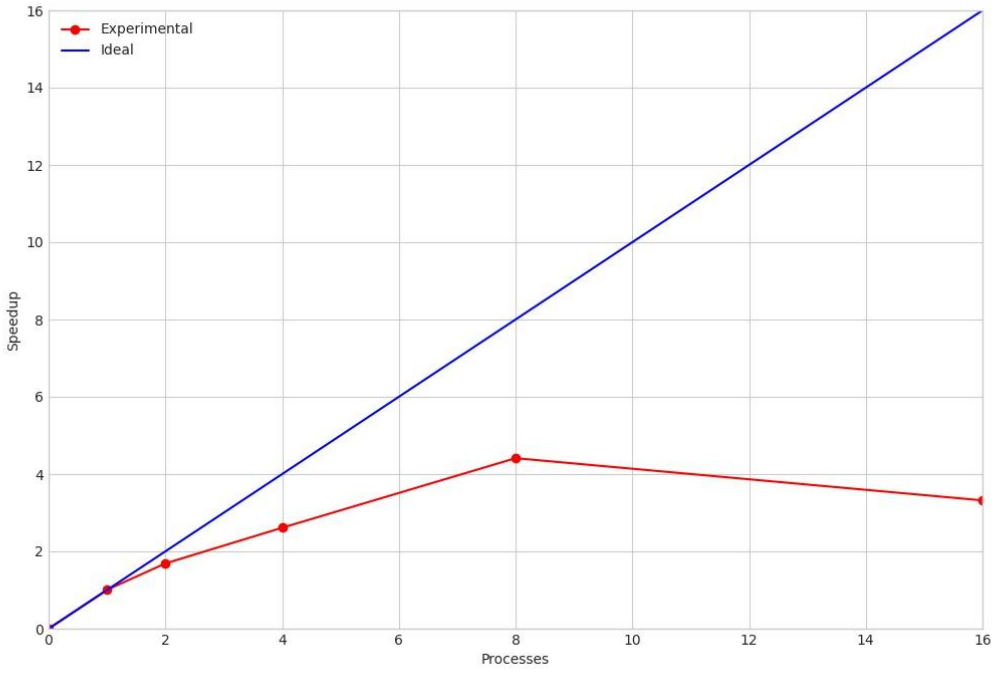
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.0128571 4286	9.5457142 86	0.0257142 8571	9.5842857 14	1	1
Parallel	1	0.0104285 7143	9.4494285 71	0.007	9.4668571 43	1.0124041 77	1.0124041 77
Parallel	2	0.008	5.6458571 43	0.0094285 71429	5.6628571 43	1.6924823 41	0.8462411 705
Parallel	4	0.0087142 85714	3.7062857 14	0.0408571 4286	3.7558571 43	2.5518238 18	0.6379559 545
Parallel	8	0.0112857 1429	2.1147142 86	0.0398571 4286	2.166	4.4248779 84	0.5531097 481
Parallel	16	0.1545714 286	2.2588571 43	0.6087142 857	3.0221428 57	3.1713542 9	0.1982096 431


Experimental
Ideal
Speedup
Processes
0
2
4
6
8
10
12
14
16

V4

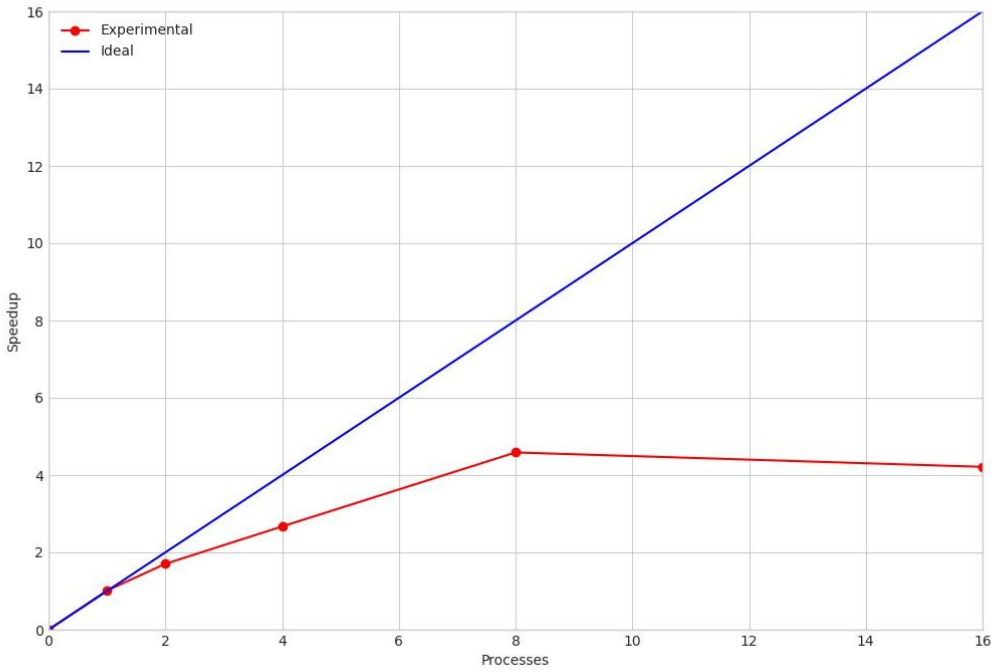
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.0128571 4286	9.5457142 86	0.0257142 8571	9.5842857 14	1	1
Parallel	1	0.01	9.5174285 71	0.007	9.5347142 86	1.0051990 47	1.0051990 47
Parallel	2	0.0078571 42857	5.6648571 43	0.0085714 28571	5.6811428 57	1.6870348 02	0.8435174 009
Parallel	4	0.0088571 42857	3.629	0.0334285 7143	3.6712857 14	2.6106074 17	0.6526518 542
Parallel	8	0.011	2.1177142 86	0.0437142 8571	2.1724285 71	4.4117840 47	0.5514730 059
Parallel	16	0.1341428 571	2.2421428 57	0.5104285 714	2.8868571 43	3.3199722 88	0.2074982 68



Size-2000

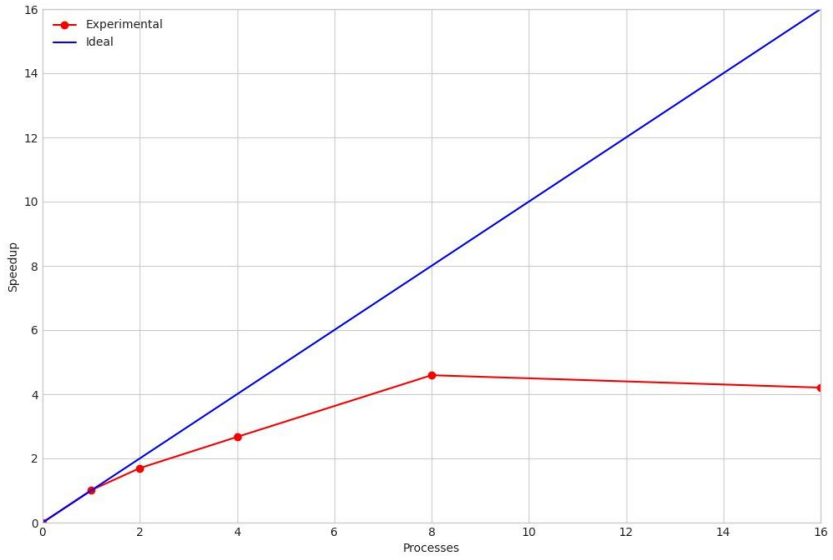
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.04	39.78428571	0.07285714286	39.89714286	1	1
Parallel	1	0.02271428571	39.36585714	0.01771428571	39.40585714	1.012467327	1.012467327
Parallel	2	0.01957142857	23.38714286	0.02071428571	23.42714286	1.703030673	0.8515153363
Parallel	4	0.02271428571	14.88871429	0.02985714286	14.94157143	2.670210633	0.6675526575
Parallel	8	0.02842857143	8.606857143	0.065	8.700428571	4.585652595	0.5732065744
Parallel	16	0.1984285714	8.695428571	0.5747142857	9.468857143	4.213511964	0.2633444978



V4

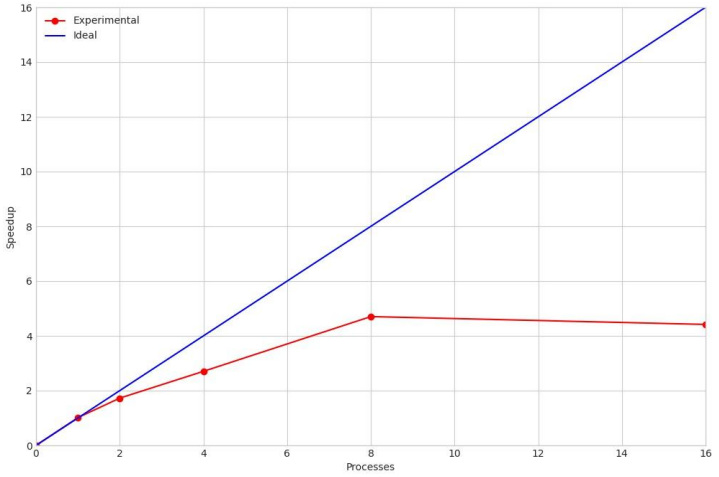
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.04	39.78428571	0.07285714286	39.89714286	1	1
Parallel	1	0.02571428571	39.44185714	0.01771428571	39.48542857	1.010426993	1.010426993
Parallel	2	0.02028571429	23.44571429	0.02571428571	23.49142857	1.698370226	0.8491851131
Parallel	4	0.02128571429	14.89742857	0.02457142857	14.94314286	2.669929829	0.6674824573
Parallel	8	0.02757142857	8.599571429	0.05957142857	8.686714286	4.592892266	0.5741115332
Parallel	16	0.19	8.711428571	0.5831428571	9.484428571	4.206594267	0.2629121417



Size-5000

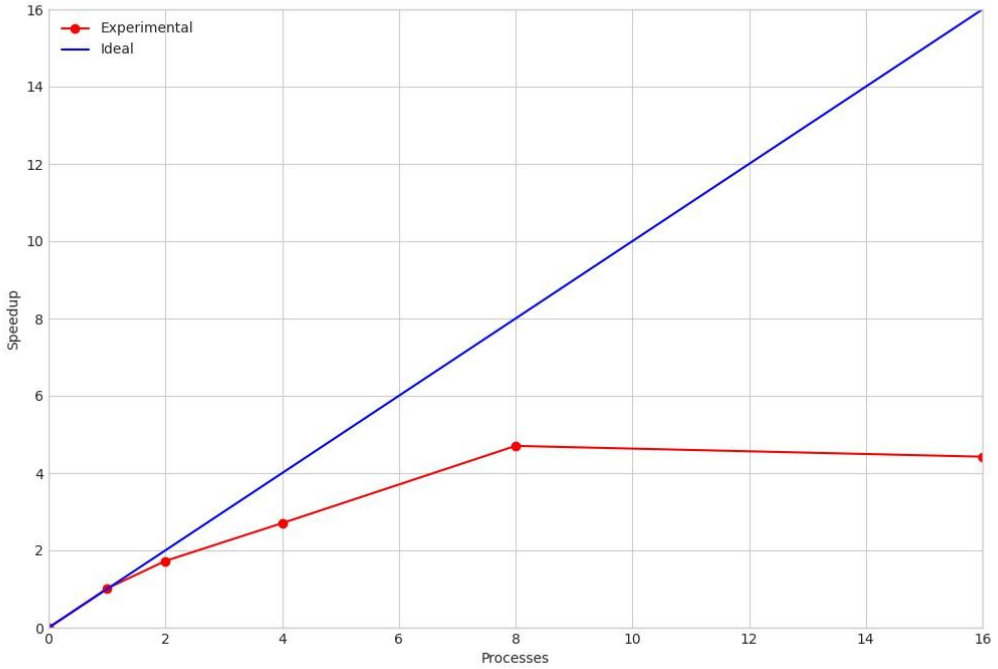
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.1471428571	634.5985714	0.5414285714	635.2871429	1	1
Parallel	1	0.14	626.7638571	0.106	627.0105714	1.013200051	1.013200051
Parallel	2	0.1222857143	367.8394286	0.1235714286	368.0852857	1.725923767	0.8629618834
Parallel	4	0.1404285714	234.4998571	0.2325714286	234.873	2.704811293	0.6762028233
Parallel	8	0.1905714286	134.371	0.4875714286	135.049	4.704123265	0.5880154082
Parallel	16	0.5557142857	138.1802857	5.056	143.7918571	4.418102356	0.2761313973



V4

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.147142857	634.5985714	0.541428571	635.2871429	1	1
Parallel	1	0.147428571	626.4268571	0.108285714	626.6821429	1.013731044	1.013731044
Parallel	2	0.123428571	367.7854286	0.131285714	368.0401429	1.726135464	0.863067732
Parallel	4	0.140714286	234.4937143	0.292142857	234.9265714	2.704194502	0.676048626
Parallel	8	0.192	134.3468571	0.522428571	135.0615714	4.703685409	0.587960676
Parallel	16	0.541571429	138.0535714	5.002714286	143.5981429	4.424062388	0.276503899



Case n° 2

In this case study, the main purpose was to analyze the performance of our program in the following build setup:

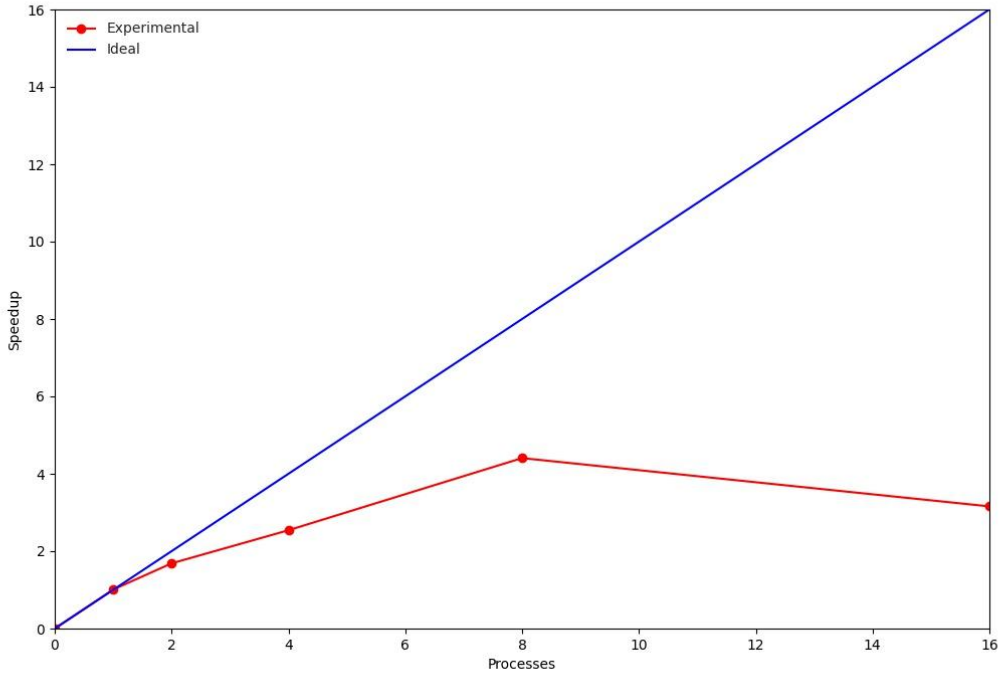
- The sequential program is compiled with gcc optimization -Ox where x = 3
- The parallel programs are compiled with the gcc optimization -Ox where x = 3

So here we want to highlight the difference between the sequential program compared to a parallel one with the same compiler optimization, furthermore the case study is done on multiple sizes that are 1250, 2000, 5000 and with different number of processes (0, 1, 2, 4, 8, 16). In addition, there are many versions of the read of matrix A and B in the files, but we focused on version 1 and version 4. These are the two versions whose results were the most interesting to analyze. In the end, the matrix B was written column wise in order to improve the performance of the dot product.

Size-1250

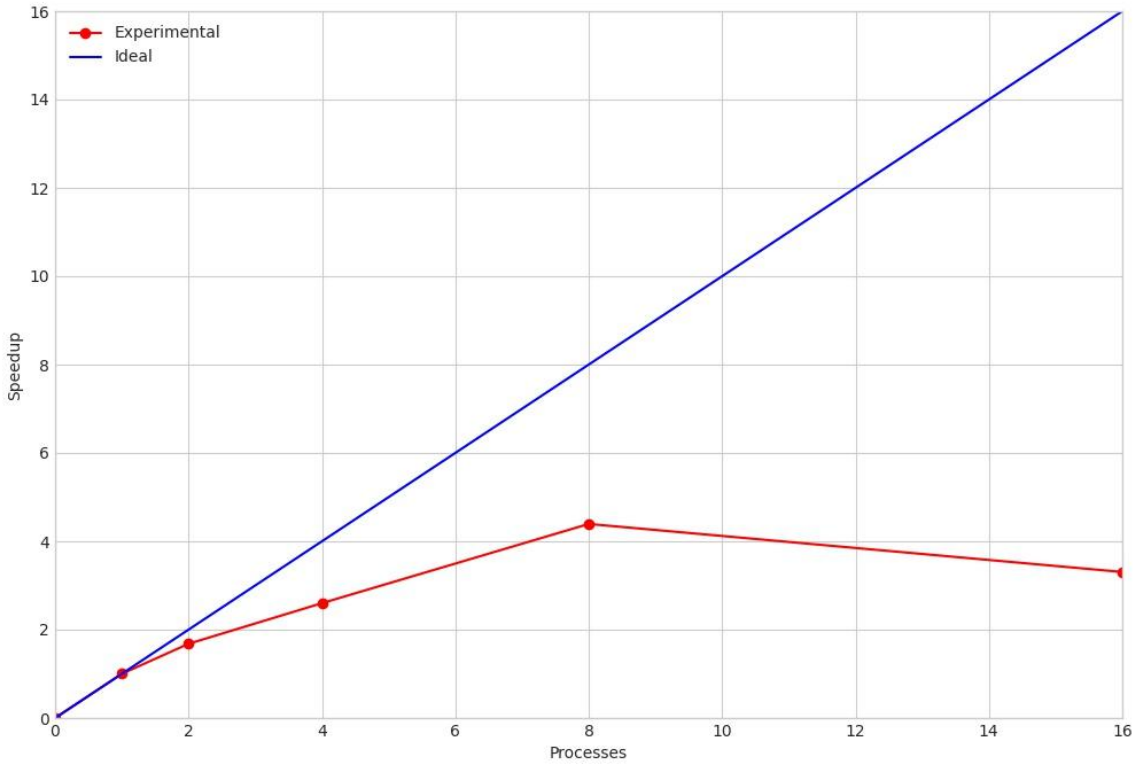
V1

Version	Processes	ReadFromFile	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.01	9.508571429	0.02428571429	9.542857143	1	1
Parallel	1	0.01042857143	9.449428571	0.007	9.466857143	1.008028007	1.008028007
Parallel	2	0.008	5.645857143	0.009428571429	5.662857143	1.685166498	0.842583249
Parallel	4	0.008714285714	3.706285714	0.04085714286	3.755857143	2.540793427	0.635198356
Parallel	8	0.011285714	2.114714286	0.03985714286	2.166	4.405751202	0.550718902
Parallel	16	0.1545714286	2.258857143	0.6087142857	3.022142857	3.157645947	0.197352871



V4

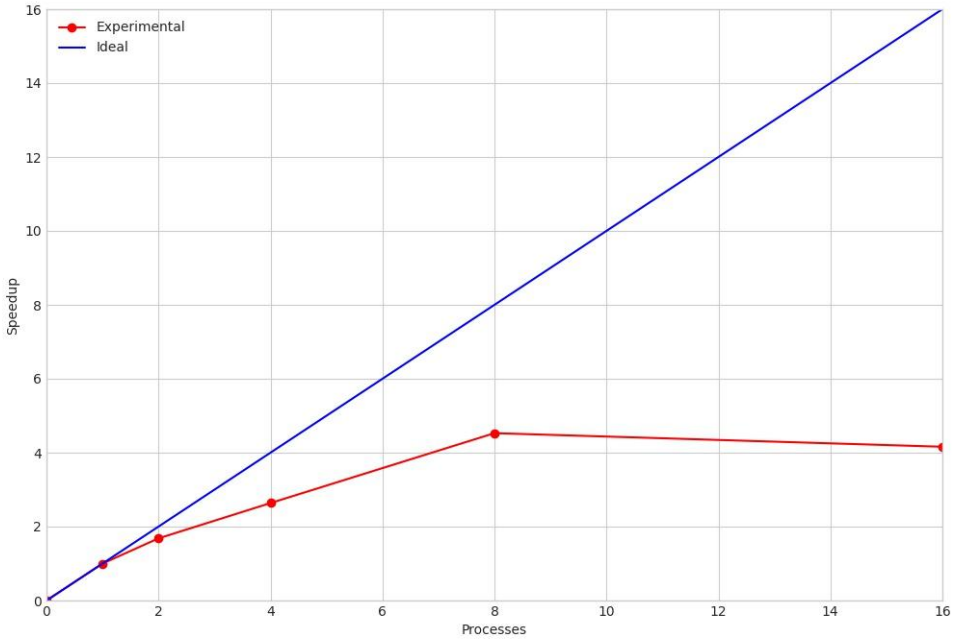
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.01	9.508571429	0.0242857 1429	9.542857143	1	1
Parallel	1	0.01	9.517428571	0.007	9.534714286	1.000854022	1.000854022
Parallel	2	0.0078571 42857	5.664857143	0.0085714 28571	5.681142857	1.679742507	0.8398712 533
Parallel	4	0.0088571 42857	3.629	0.0334285 7143	3.671285714	2.599322931	0.6498307 327
Parallel	8	0.011	2.117714286	0.0437142 8571	2.172428571	4.392713882	0.5490892 352
Parallel	16	0.1341428 571	2.242142857	0.5104285 714	2.886857143	3.305621536	0.2066013 46



Size-2000

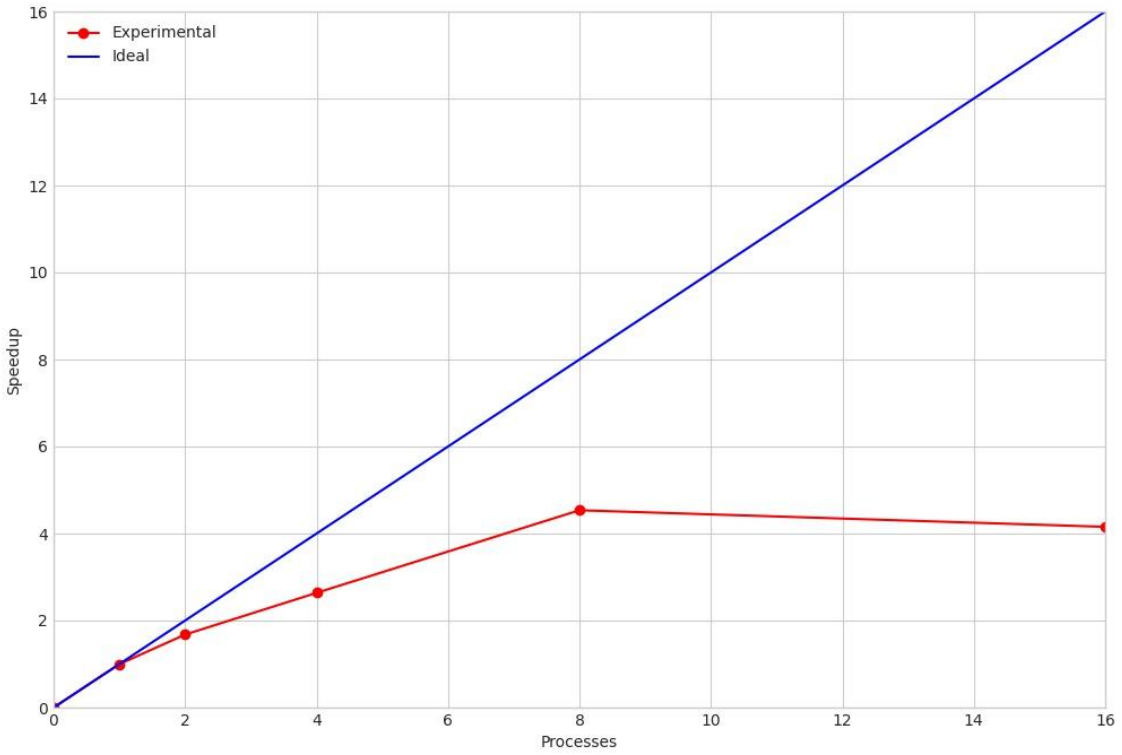
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.03571428571	39.25	0.1114285714	39.39714286	1	1
Parallel	1	0.02271428571	39.36585714	0.01771428571	39.40585714	0.9997788581	0.9997788581
Parallel	2	0.01957142857	23.38714286	0.02071428571	23.42714286	1.681687908	0.8408439539
Parallel	4	0.02271428571	14.88871429	0.02985714286	14.94157143	2.636746948	0.6591867369
Parallel	8	0.02842857143	8.606857143	0.065	8.700428571	4.528184162	0.5660230202
Parallel	16	0.1984285714	8.695428571	0.5747142857	9.468857143	4.160707281	0.2600442051



V4

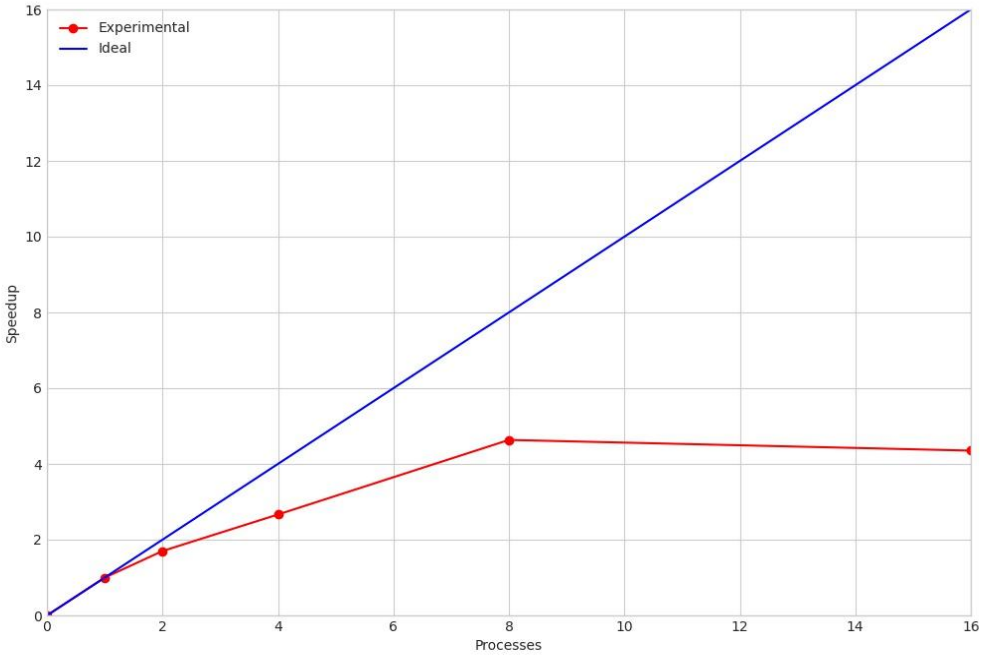
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.03571428571	39.25	0.1114285714	39.39714286	1	1
Parallel	1	0.02571428571	39.44185714	0.01771428571	39.48542857	0.9977640938	0.9977640938
Parallel	2	0.02028571429	23.44571429	0.02571428571	23.49142857	1.677085867	0.8385429336
Parallel	4	0.02128571429	14.89742857	0.02457142857	14.94314286	2.636469666	0.6591174165
Parallel	8	0.02757142857	8.599571429	0.05957142857	8.686714286	4.535333103	0.5669166379
Parallel	16	0.19	8.711428571	0.5831428571	9.484428571	4.153876278	0.2596172674



Size-5000

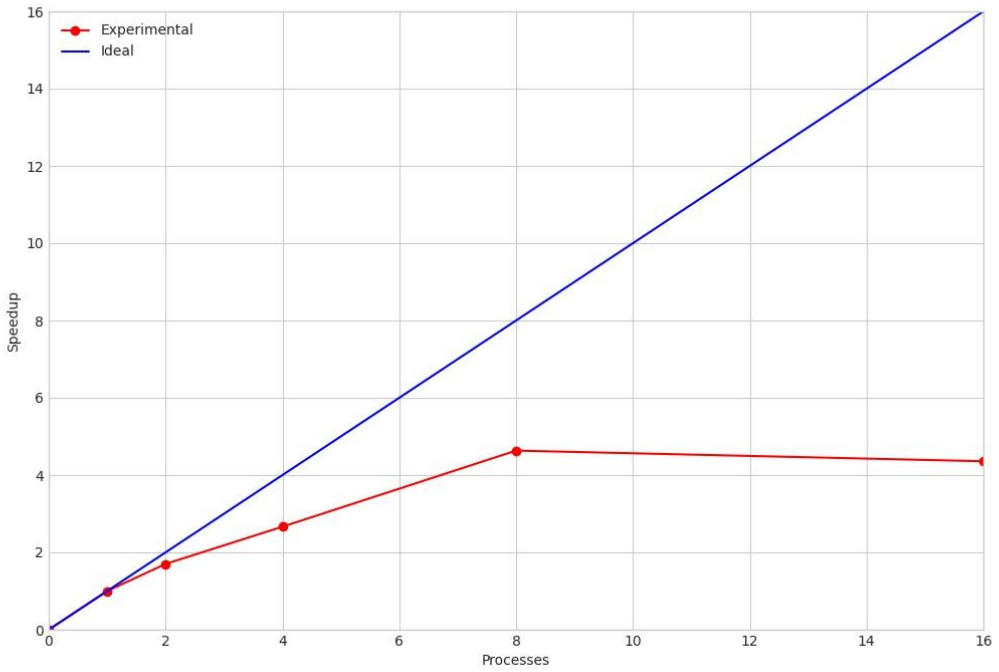
V1

Version	Processes	ReadFrom File	MatrixDot	WriteTonFile	Elapsed	Speedup	Efficiency
Serial	1	0.148571428 6	625.0714286	0.541428571 4	625.7614286	1	1
Parallel	1	0.14	626.7638571	0.106	627.0105714	0.998007780 2	0.998007780 2
Parallel	2	0.122285714 3	367.8394286	0.123571428 6	368.0852857	1.700044671	0.850022335 7
Parallel	4	0.140428571 4	234.4998571	0.232571428 6	234.873	2.664254421	0.666063605 2
Parallel	8	0.190571428 6	134.371	0.487571428 6	135.049	4.633588002	0.579198502 6
Parallel	16	0.555714285 7	138.1802857	5.056	143.7918571	4.351855807	0.271990988



V4

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.1485714286	625.0714286	0.5414285714	625.7614286	1	1
Parallel	1	0.1474285714	626.4268571	0.1082857143	626.6821429	0.9985308114	0.9985308114
Parallel	2	0.1234285714	367.7854286	0.1312857143	368.0401429	1.700253194	0.8501265972
Parallel	4	0.1407142857	234.4937143	0.2921428571	234.9265714	2.663646878	0.6659117195
Parallel	8	0.192	134.3468571	0.5224285714	135.0615714	4.633156733	0.5791445912
Parallel	16	0.5415714286	138.0535714	5.002714286	143.5981429	4.357726473	0.2723579045



Case n° 3

In this case study, the main purpose was to analyze the performance of our program in the following build setup:

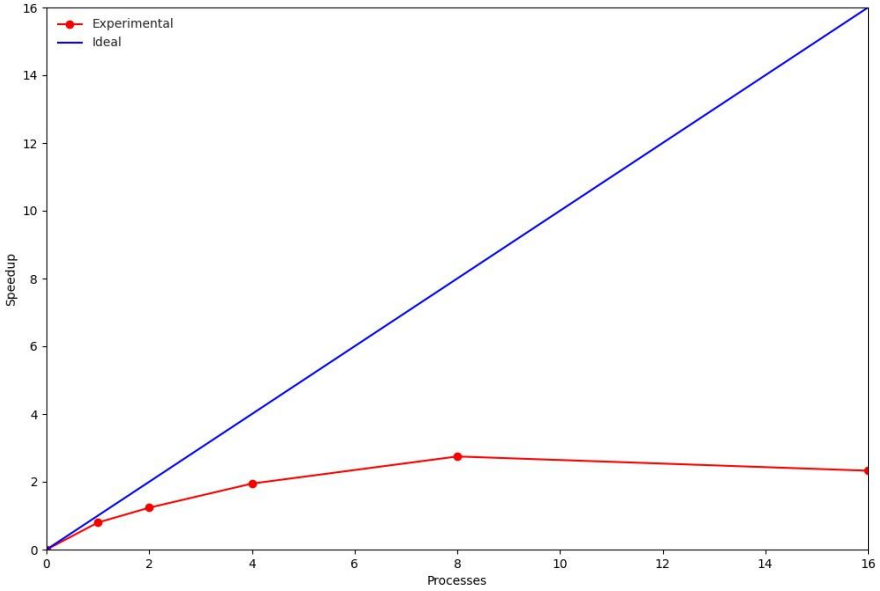
- The sequential program is compiled without gcc optimization
- The parallel programs are compiled without the gcc optimization

So here we want to highlight the difference between the sequential program compared to a parallel one without compiler optimization, furthermore the case study is done on multiple sizes that are 1250, 2000, 5000 and with different number of processes (0, 1, 2, 4, 8, 16). In addition, there are many versions of the read of matrix A and B in the files, but we focused on version 1 and version 4. These are the two versions whose results were the most interesting to analyze. In the end, the matrix B was written column wise in order to improve the performance of the dot product.

Size-1250

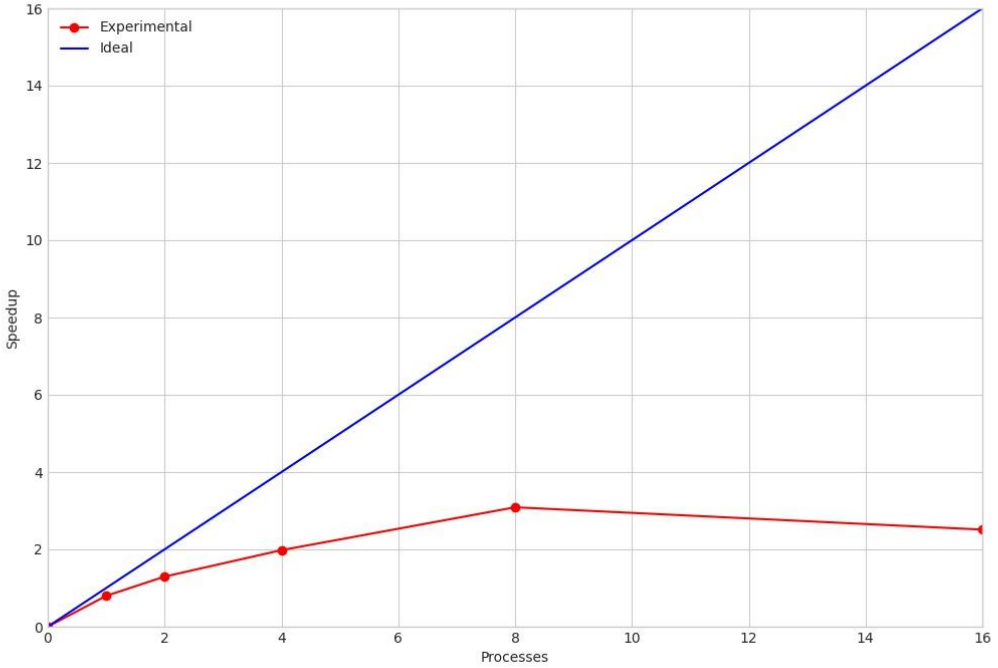
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.0128571 4286	9.5457142 86	0.0257142 8571	9.5842857 14	1	1
Parallel	1	0.009	12.001285 71	0.0072857 14286	12.018142 86	0.7974847 552	0.7974847 552
Parallel	2	0.0081428 57143	7.7404285 71	0.0102857 1429	7.7591428 57	1.2352248 04	0.6176124 02
Parallel	4	0.0101428 5714	4.8527142 86	0.0592857 1429	4.9222857 14	1.9471209 66	0.4867802 415
Parallel	8	0.0137142 8571	3.2358571 43	0.2384285 714	3.4878571 43	2.7479008 81	0.3434876 101
Parallel	16	0.1497142 857	3.2761428 57	0.6918571 429	4.1175714 29	2.3276549 98	0.1454784 374



V4

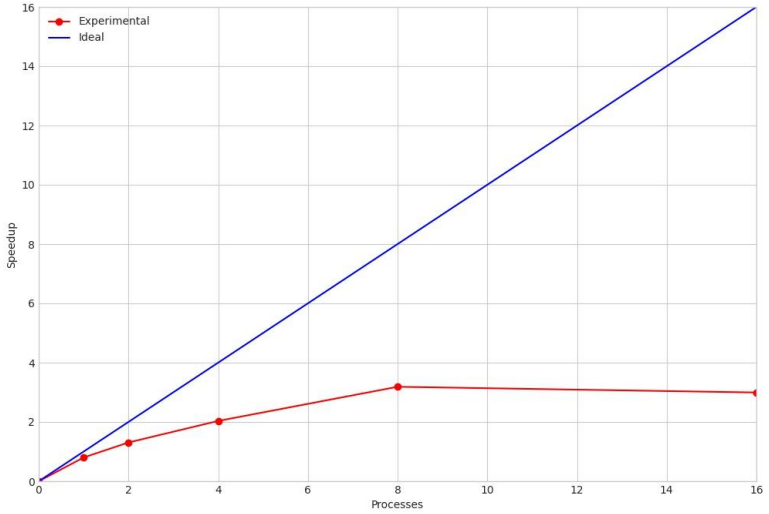
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.0128571 4286	9.5457142 86	0.0257142 8571	9.5842857 14	1	1
Parallel	1	0.0115714 2857	12.011	0.0075714 28571	12.030285 71	0.7966798 081	0.7966798 081
Parallel	2	0.0081428 57143	7.3824285 71	0.0094285 71429	7.4001428 57	1.2951487 42	0.6475743 712
Parallel	4	0.012	4.7885714 29	0.0417142 8571	4.8422857 14	1.9792895 92	0.4948223 979
Parallel	8	0.0118571 4286	3.0312857 14	0.0591428 5714	3.1024285 71	3.0892848 92	0.3861606 115
Parallel	16	0.1454285 714	3.1001428 57	0.5697142 857	3.8158571 43	2.5116993	0.1569812 062



Size-2000

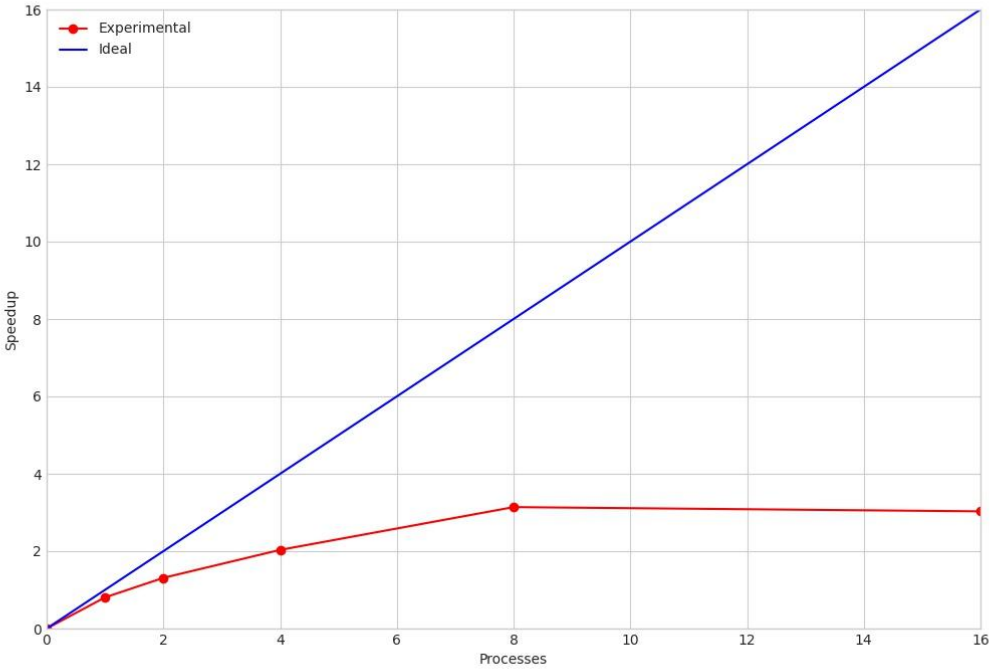
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.04	39.78428571	0.07285714286	39.89714286	1	1
Parallel	1	0.024	49.82857143	0.01885714286	49.87128571	0.8000022916	0.8000022916
Parallel	2	0.02057142857	30.48185714	0.02957142857	30.532	1.306732047	0.6533660235
Parallel	4	0.02628571429	19.55285714	0.03628571429	19.61528571	2.033982244	0.5084955061
Parallel	8	0.03157142857	12.39914286	0.093	12.52328571	3.185836667	0.3982295837
Parallel	16	0.1995714286	12.46628571	0.6597142857	13.32557143	2.994028667	0.1871267917



V4

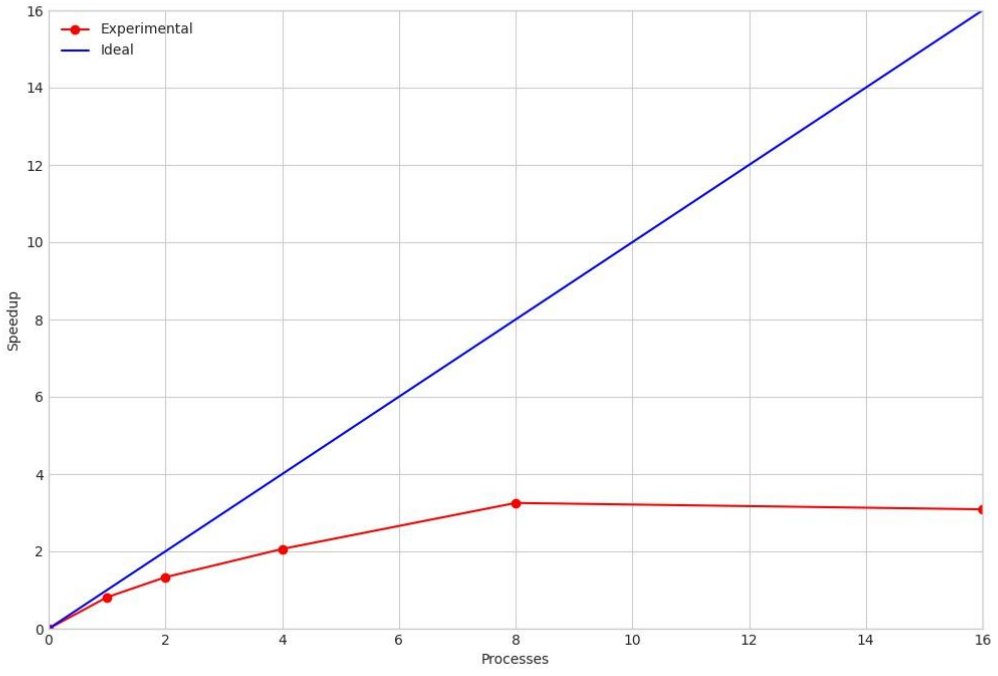
Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.04	39.78428571	0.07285714286	39.89714286	1	1
Parallel	1	0.02657142857	49.49014286	0.01885714286	49.53542857	0.8054264192	0.8054264192
Parallel	2	0.02157142857	30.39271429	0.03628571429	30.451	1.310207969	0.6551039844
Parallel	4	0.02457142857	19.55342857	0.04585714286	19.62357143	2.033123433	0.5082808576
Parallel	8	0.03085714286	12.481	0.2	12.71185714	3.138577032	0.3923221289
Parallel	16	0.1795714286	12.42542857	0.5655714286	13.17014286	3.029362953	0.1893351846



Size-5000

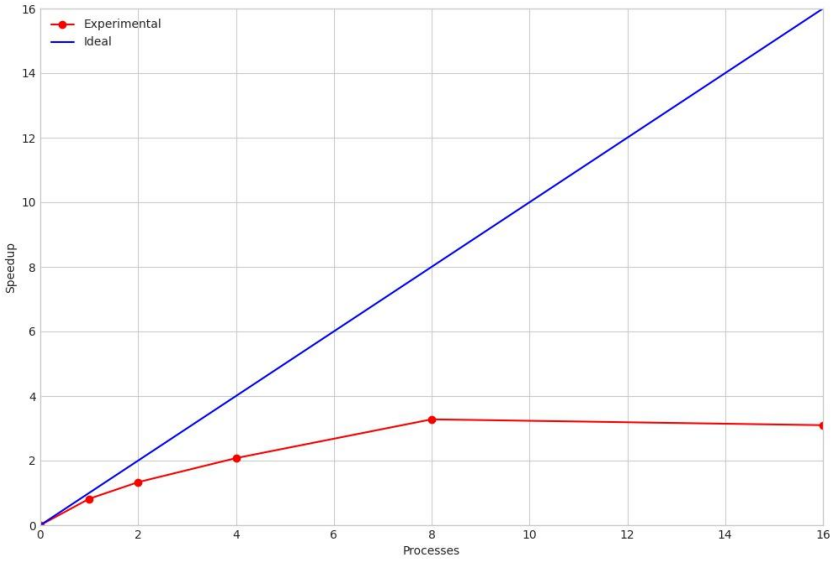
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.1471428571	634.5985714	0.5414285714	635.2871429	1	1
Parallel	1	0.1414285714	780.998	0.1131428571	781.2528571	0.8131645069	0.8131645069
Parallel	2	0.1275714286	476.6782857	0.2064285714	477.0124286	1.331804173	0.6659020864
Parallel	4	0.1562857143	307.5581429	0.3342857143	308.0488571	2.062293458	0.5155733645
Parallel	8	0.2008571429	194.3702857	0.7188571429	195.2897143	3.253049681	0.4066312102
Parallel	16	0.5798571429	198.9677143	6.108714286	205.6562857	3.089072336	0.1930670021



V4

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.147142857	634.5985714	0.541428571	635.2871429	1	1
Parallel	1	0.146428571	776.4421429	0.114571429	776.7031429	0.817927864	0.817927864
Parallel	2	0.127	476.4534286	0.191714286	476.7721429	1.332475381	0.666237691
Parallel	4	0.158857143	305.3902857	0.353571429	305.9025714	2.076763003	0.519190751
Parallel	8	0.199571429	192.9007143	0.696285714	193.7964286	3.278115843	0.409764480
Parallel	16	0.55	198.0167143	6.538	205.1045714	3.097381684	0.193586355



Considerations

Basically we compared the programs with O0 and O3 modifiers and we noticed that the maximum speedups reachable are of 4.70 and 4.60 in the first and second case, while in the third case (Only O0 modifier, both sequential and parallel programs) the maximum speedup is 3.30. Looking at the absolute elapsed times in the tables we can see that the O3 modifier allows us to boost the performances of the parallel programs more than the performance of the sequential ones.

We also noticed that in every trial the max speedup is reached with 8 processes and it varies between 3.30 and 4.70, that is consistent with our system that has 4 physical cores but 8 threads.

Other considerations

The MPI documentation states that:
“

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user’s buffer after a read operation completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee that data has been transferred to the storage device.
”

(*MPI: A Message-Passing Interface Standard*, n.d., 506)

also:

“

In addition, in order to prevent subsequent shared offsetaccesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses.

…

Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel.
”

(*MPI: A Message-Passing Interface Standard*, n.d., 525)

This means that the time measured for the read operation is the time taken by a single process to read its portion of the A matrix (without waiting the others) and the time taken for reading the whole matrix B. While in the case of the write it is only the time taken for “initializing the access” and not writing the relative piece of the resulting C matrix.

Finally, it is worth mentioning that the times resulting from the measures of the parallel programs are the maximum times between all the processes, i.e. if we have that one process that takes longer to compute its portion of the resulting C matrix we use its measured time because only when the slower process completes its task the matrix will be fully calculated.

Test case

In the *test* folder there is the file *test.c* which is the file that contains the test cases. We used the decorator pattern to define the test functions.

We tested the matrix dot matrix product several times by asserting that the expected result and the one returned by our function are the same.

API

Public Functions

Type	Name
void	**init** (double * a, double * b, int rows_a, int columns_a, int rows_b, int columns_b) This function initializes the two matrices A and B needed in the program.
void	**matrix_dot_matrix** (double * a, double * b, double * c, int n_rows_A, int n_columns_A, int n_rows_B, int n_columns_B) This function makes the dot product between the matrix A and the matrix B that is expected to be allocated columnwise.
void	**print_matrices** (double * a, double * b, double * c, int rows_a, int columns_a, int rows_b, int columns_b) Prints the matrices to stdout. C should be the result of AxB.

Public Functions Documentation

Function `init`

```
void init (
    double * a,
    double * b,
    int rows_a,
    int columns_a,
    int rows_b,
    int columns_b
)
```

Parameters:

- `a` pointer to the A matrix used in matrix dot product
- `b` pointer to the B matrix used in matrix dot product
- `rows_a` number of rows of A
- `columns_a` number of columns of A
- `rows_b` number of rows of B
- `columns_b` number of columns of B

Function `matrix_dot_matrix`

```
void matrix_dot_matrix (
    double * a,
    double * b,
    double * c,
    int n_rows_A,
    int n_columns_A,
    int n_rows_B,
    int n_columns_B
)
```

Parameters:

- `a` pointer to the A matrix used in the dot product.
- `b` pointer to the B matrix used in the dot product.
- `c` pointer to the C matrix used to store the result of dot product.
- `n_rows_A` number of rows of A
- `n_columns_A` number of columns of A
- `n_rows_B` number of rows of B
- `n_columns_B` number of columns of B

Function `print_matrices`

```
void print_matrices (
    double * a,
    double * b,
    double * c,
    int rows_a,
    int columns_a,
    int rows_b,
    int columns_b
)
```

Parameters:

- `a` pointer to the A matrix used in the dot product.
- `b` pointer to the B matrix used in the dot product.
- `c` pointer to the C matrix used to store the result of dot product.
- `rows_a` number of rows of A
- `columns_a` number of columns of A
- `rows_b` number of rows of B
- `columns_b` number of columns of B

How to run

1. Create a build directory and launch cmake
   ```
   mkdir build
   cd build
   cmake ..
   ```
1. Generate executables with `make`
2. To generate measures (TAKES A LOT OF TIME! Our measures are already included so you should skip this step) run `make generate_measures`
3. To extract mean times and speedup curves from them run `make extract_measures`

Results can be found in the `measures/measure` directory, divided by problem size and the gcc optimization option used.

Extras

We have also tried running our program on a raspberry pi cluster. The front end is a pi3, while the worker nodes are 4 pi4.

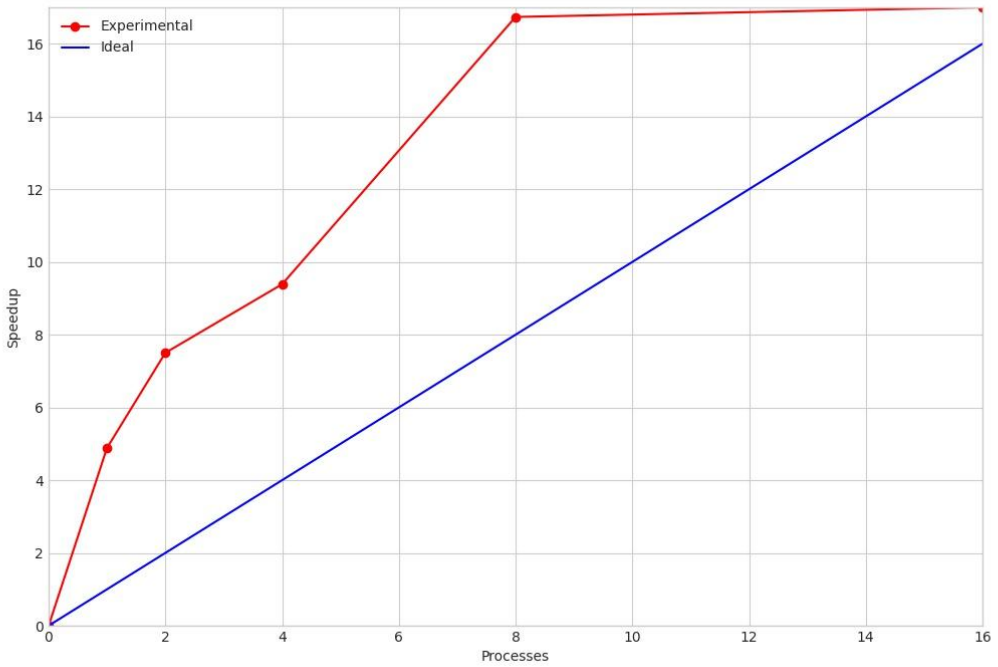
We immediately noticed that the serial program was slower than the serial program in the previous configuration due to the difference between the single thread performance of the processors.

The parallel programs run much faster with respect to the serial program on the same configuration, as we are using physically different devices. Each device uses up to 4 processes.

Size-2000

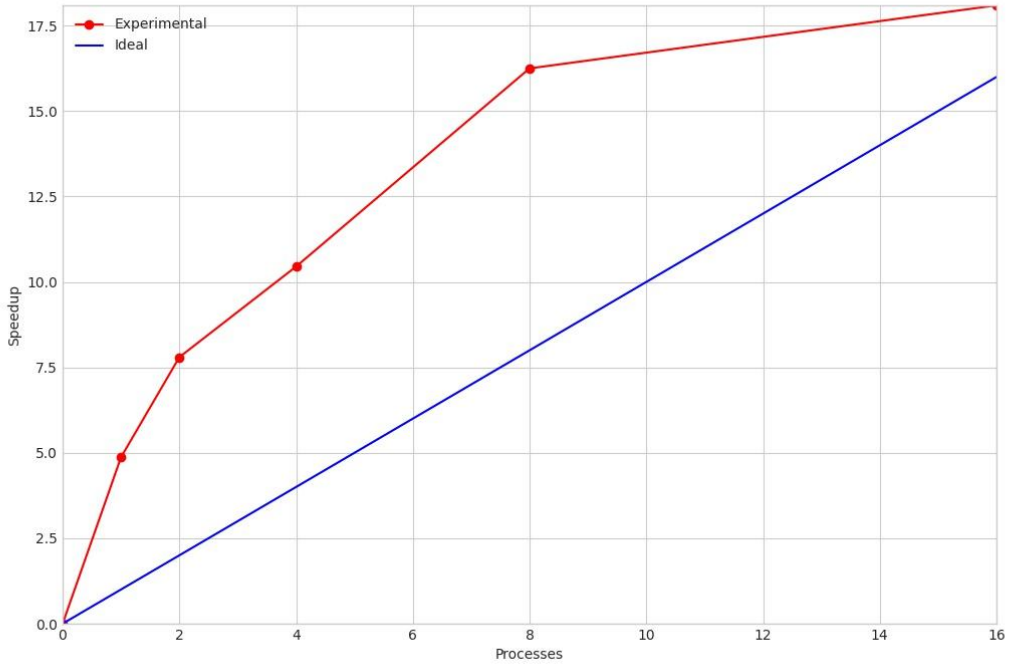
V1

Version	Processes	ReadFrom File	Matrix Dot	WriteTo File	Elapsed	Speedup	Efficiency
Serial	1	0.161428571	368.2	1.37	369.7314286	1	1
Parallel	1	1.042857143	69.63357143	4.985714286	75.662	4.886619817	4.886619817
Parallel	2	0.232285714	35.09757143	13.95585714	49.28585714	7.501775357	3.750887679
Parallel	4	0.311714285	18.78014286	20.29542857	39.38757143	9.387007504	2.346751876
Parallel	8	1.6	9.413857143	11.07814286	22.09171429	16.73620362	2.092025452
Parallel	16	1.301428571	4.730857143	15.71614286	21.74885714	17.00003941	1.062502463



V4

Version	Processes	ReadFromFile	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	0.1614285714	368.2	1.37	369.7314286	1	1
Parallel	1	1.022142857	69.62671429	5.296714286	75.94557143	4.868373779	4.868373779
Parallel	2	0.2314285714	35.12085714	12.057	47.40914286	7.798736832	3.899368416
Parallel	4	0.2418571429	18.74328571	16.39142857	35.37642857	10.45134977	2.612837442
Parallel	8	0.7857142857	9.394428571	12.57228571	22.75242857	16.25019621	2.031274526
Parallel	16	1.258714286	4.724857143	14.44942857	20.433	18.09481861	1.130926163



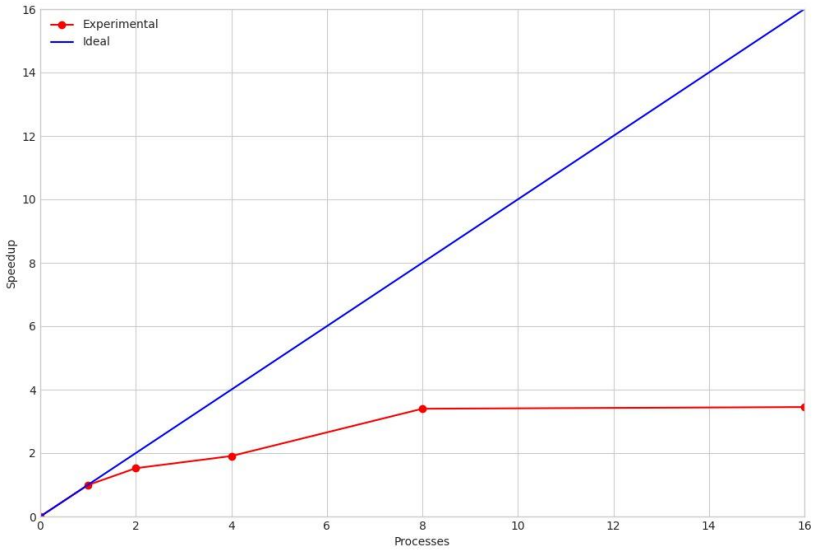
Why is this happening?

At first we did not notice that the sequential program was run on the cluster frontend which is a raspberry pi3. On the contrary the parallel algorithm was executed on the cluster workers, which are raspberry pi4. So the reason why we have very high speedups is that the sequential algorithm is executed on an older device.
As we found out by looking at this very useful table the pi3 processor, which is a Cortex-A53 has a more shallow pipe, made of 8 stages, than the pi4 processor which is a Cortex-A72. Also the Cortex-A72 also has out-of-order-execution.
We decided to execute the serial program on a pi4 and then compared the results with the previous “mistake”.

Size-2000 with serial on pi4

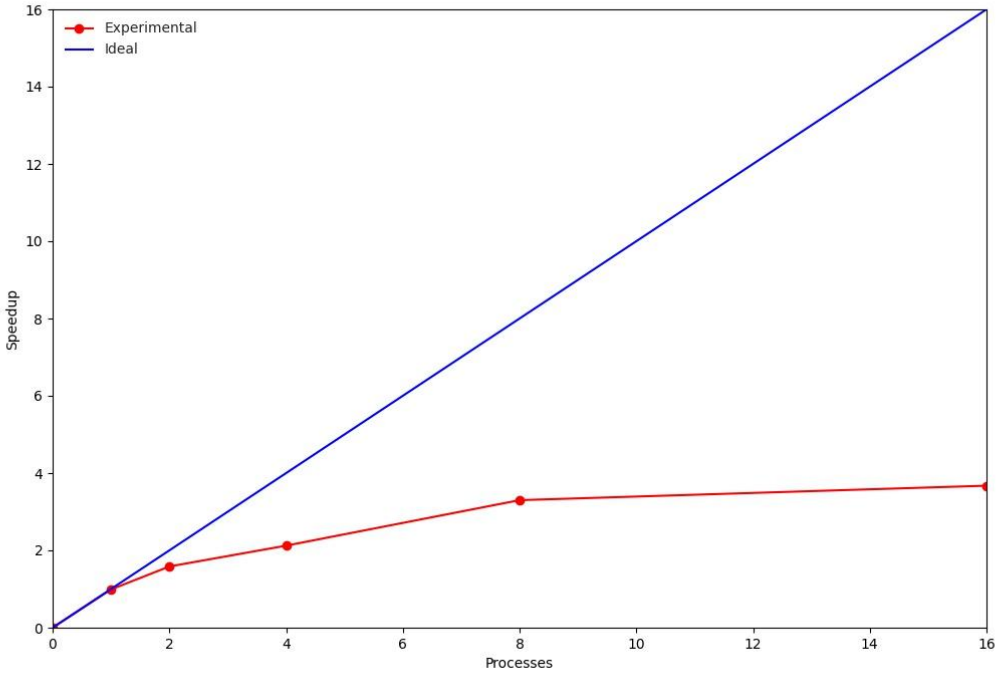
V1

Version	Processes	ReadFrom File	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	1.012857143	69.88857143	4.17	75.07142857	1	1
Parallel	1	1.042857143	69.63357143	4.985714286	75.662	0.992194609	0.992194609
Parallel	2	0.232285714	35.09757143	13.95585714	49.28585714	1.523183991	0.761591954
Parallel	4	0.311714285	18.78014286	20.29542857	39.38757143	1.905967437	0.476491859
Parallel	8	1.6	9.413857143	11.07814286	22.09171429	3.398171236	0.424771407
Parallel	16	1.301428571	4.730857143	15.71614286	21.74885714	3.451741307	0.215733831



V4

Version	Processes	ReadFromFile	MatrixDot	WriteToFile	Elapsed	Speedup	Efficiency
Serial	1	1.012857143	69.88857143	4.17	75.07142857	1	1
Parallel	1	1.022142857	69.62671429	5.296714286	75.94557143	0.988489877	0.988489877
Parallel	2	0.2314285714	35.12085714	12.057	47.40914286	1.583479978	0.791739989
Parallel	4	0.2418571429	18.74328571	16.39142857	35.37642857	2.122074828	0.530518707
Parallel	8	0.7857142857	9.394428571	12.57228571	22.75242857	3.299490792	0.412436349
Parallel	16	1.258714286	4.724857143	14.44942857	20.433	3.674028707	0.229626794



Final considerations

What we can confirm is that the serial program alone executes much faster on the pi4, rather than on the pi3 due to the above stated reasons. We can see that on the pi3 it took 370 seconds, while on the pi4 it only took 75 seconds. We have nearly **5 of speedup**.
We can also see that the parallel programs execute similarly in both cases of executions, with the sequential on the pi3 and then on the pi4.