COMP4300 Individual Project - Report

Runpeng Luo u7109933

May 26, 2022

Table of Contents

Summary

This report consists of benchmarking each algorithm (ant its variation) among different parameter settings (including number of execution unit, input data size, platform, etc). Explanation on implementations can be refered to `README.md`, related compile and execution can be found in MakeFile, one for each part.

The benchmarking is ran on both Gadi and local Laptop, with following Configuration.

```
# Gadi Platform
normal
Normal priority queue for standard computational intensive jobs
2 x 24-core Intel Xeon Platinum 8274 (Cascade Lake) 3.2 GHz CPUs per node
192GB RAM per node
2 CPU sockets per node, each with 2 NUMA nodes
12 CPU cores per NUMA node
48 GB local RAM per NUMA node
400 GB local SSD disk per node
Max request of 20736 CPU cores, exceptions available on request

Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              96
On-line CPU(s) list: 0-95
Thread(s) per core:  2
Core(s) per socket:  24
Socket(s):           2
NUMA node(s):        4
Vendor ID:           GenuineIntel
CPU family:          6
```

```
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8274 CPU @ 3.20GHz
Stepping:              7
CPU MHz:               3200.000
CPU max MHz:           4000.0000
CPU min MHz:           1200.0000
BogoMIPS:              6400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              36608K

# MacOS Platform

Model Name: MacBook Pro
Model Identifier:   MacBookPro16,1
Processor Name: 6-Core Intel Core i7
Processor Speed:    2.6 GHz
Number of Processors:   1
Total Number of Cores:  6
Memory: 16 GB

Architecture:             x86_64
Byte Order:               Little Endian
Total CPU(s):             12
Thread(s) per core:       2
Core(s) per socket:       6
Socket(s):                1
Vendor:                   GenuineIntel
CPU family:               6
Model:                    158
Model name:               MacBookPro16,1
Stepping:                 10
L1d cache:                32K
L1i cache:                32K
L2 cache:                 256K
L3 cache:                 12M
```

Speed up and efficiency is calculated via following formula

```
# Number of Process = P
# serial run-time = time (when process/thread = 1) = Ts
# Parallel run-time =  time (when process/thread > 1) = Tp
Speed_Up S = Ts/Tp
Efficiency E = S/P
```

Part A

Block Matrix Multiplication by OpenMP

Consider both Gadi and Local laptop's L1 data cache size is 32K, which can hold up to 8000 `int` type variable (4 byte each), since block matrix multiplication requires 3 blocks in a row (A, B, C) for computation within loop, each block can hold around 2667 integers, which leads to block size around 51. Since loop parameter should also be loaded in L1 data cache, I use `48` as the maximum block size for testing and test the suitable block size to use under current L1 cache size.

Both version 1 and version 2 uses `omp for schedule(dynamic)` to parallelize the outer sub-block loop, whereas version2 parallelizing the code by adding another `omp for schedule(dynamic)` to introduce more parallelism, whereas uses `omp atomic` to prevent the race condition.

Running Time Plot&Analysis

The running time plot for block size < 32 is placed in Summplementry Material.

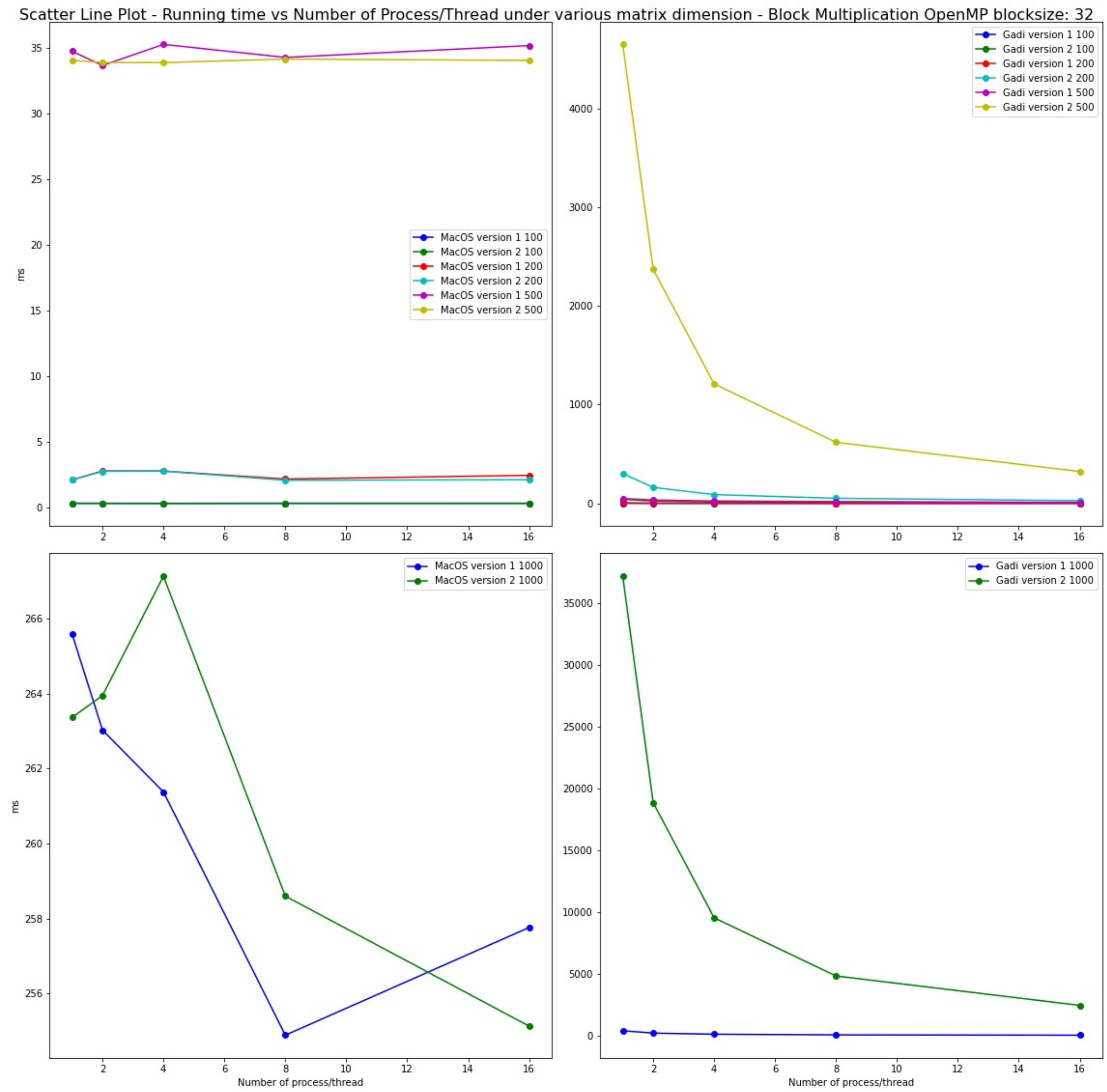


figure 1. Block Multiplication Running Time Plot, Block size 32

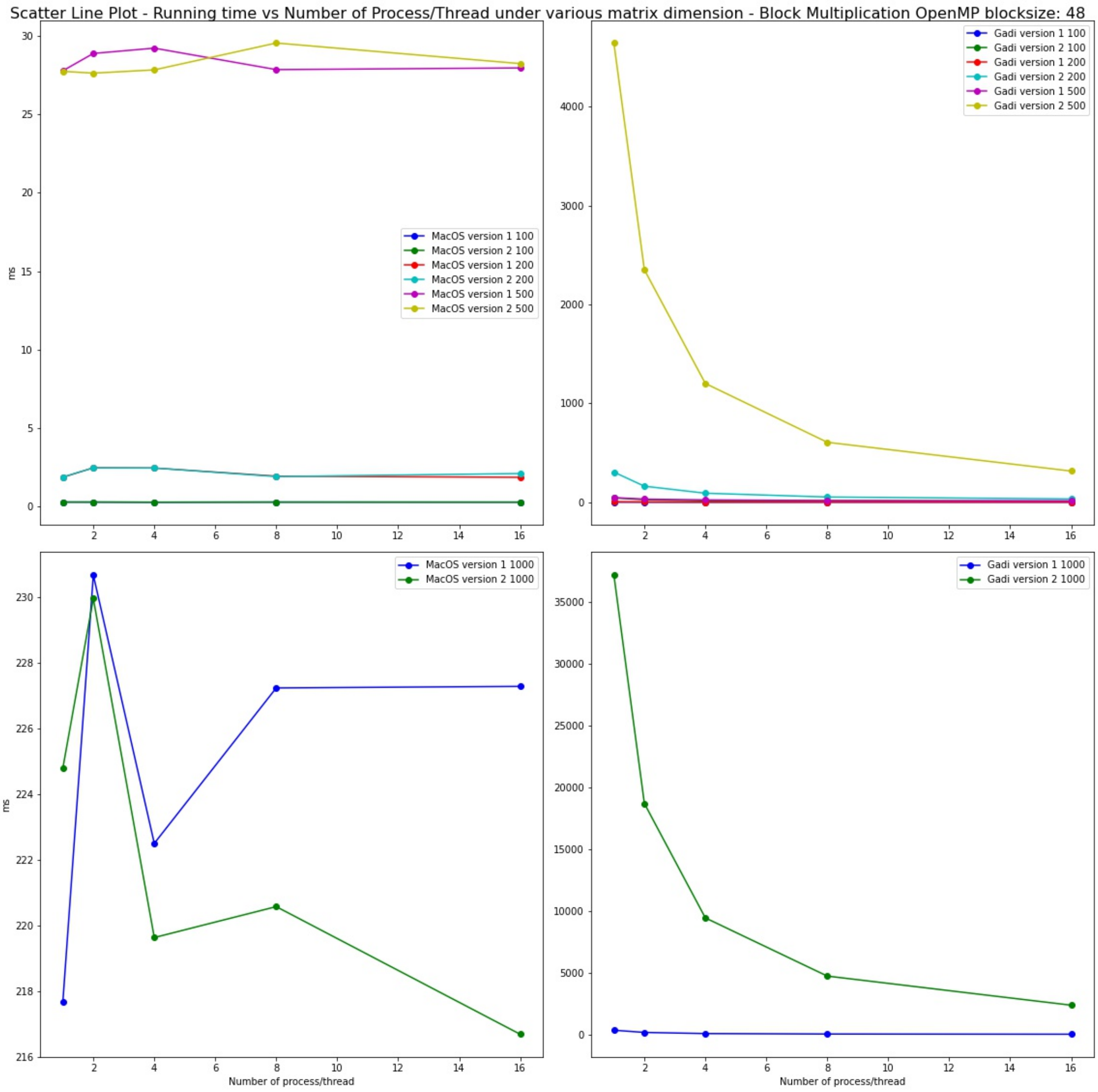


figure 2. Block Multiplication Running Time Plot, Block size 48

Compare between version 1 and version 2, notice that although version2 introduces more parallelism, it still be outperformed by version 1 with regarding to running time. One reason for that is using `critical region`, which introduces more overhead with tradeoff to parallelism. Hence, **version 1 is better than version 2 in the aspect of running time**.
Compare between Gadi and MacOS platform, interesting fact is that MacOS platform even out perform Gadi in all set of input size and block size in version 2, which may due to variant OpenMP implementation in different environment.

num_thread=16(ms)	Mac v1 1000	Mac v2 1000	Gadi v1 1000	Gadi v2 1000
blocksize=1	6381.898	6450.872	590.1621	71566.75
blocksize=8	486.306	487.7262	59.44078	5028.82
blocksize=16	325.5699	326.8301	49.88003	3139.67
blocksize=32	257.7631	255.1364	46.41821	2457.691
blocksize=48	227.2732	216.6882	57.111	2402.853

Consider the blocksize, notice that when blocksize=48, the running time is minimal among all set of input size and blocksize, which also verified my prediction above.

Performance (Speed Up & Efficiency & Scalability) Plot&Analysis

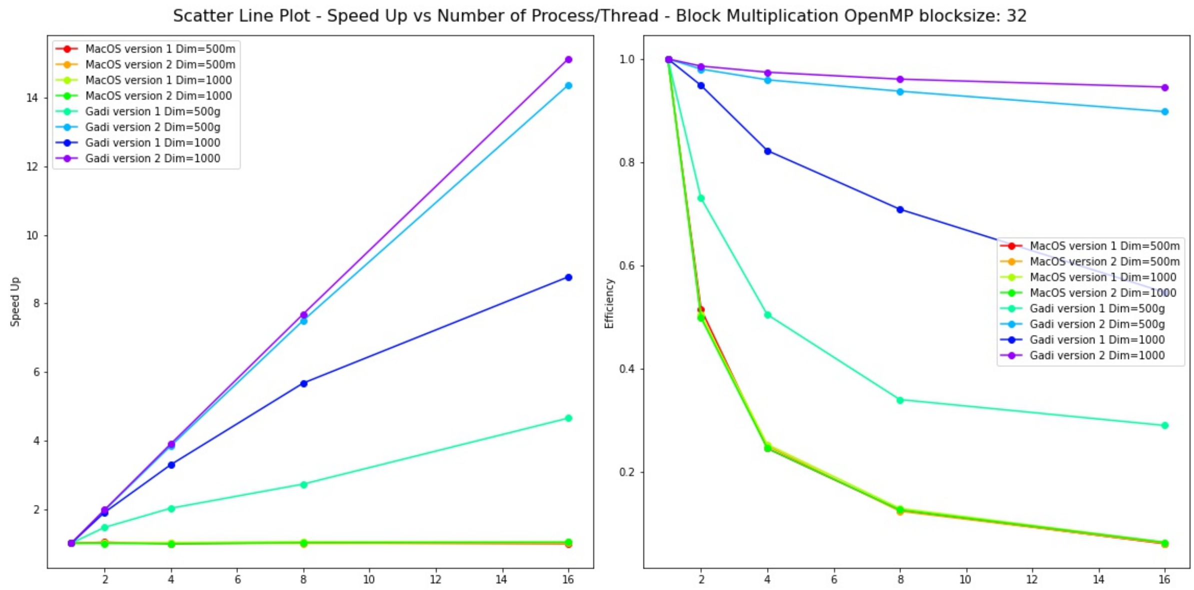


figure 3. Block Multiplication Performance Plot, Block size 32

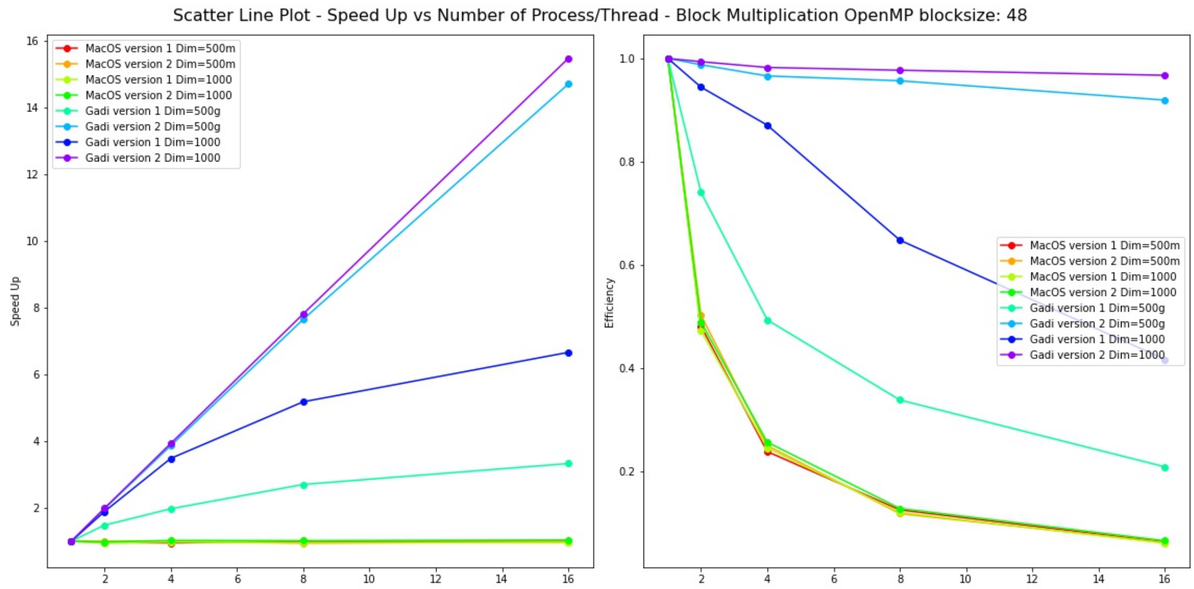


figure 4. Block Multiplication Performance Plot, Block size 48

Gadi Efficiency	v1 500	v2 500	v1 1000	v2 1000
P=1	1	1	1	1
P=2	0.740180	0.987656	0.944265	0.993536
P=4	0.492993	0.965990	0.870595	0.982420
P=8	0.337721	0.956631	0.647629	0.977063
P=16	0.208118	0.919178	0.416403	0.967270

Consider the efficiency table above (block size = 48), notice that the efficiency is fixed with increasing number of thread and fixed input size = 1000 for version 2, indicates that the version 2 block multiplication is strong scalable, whereas the version 1 is not scalable since by increasing thread number and double the input size, the efficiency is not fixed. Which indicates that version 2 implementation is more scalable than version 1 by adding additional parallelism inside the sub-block multiply step. Among the different platform, Gadi has higher speed up and efficiency compare to MacOS in various parameter setting in the plot.

SUMMA algorithm by MPI

SUMMA version 2 uses OpenMP parallelism to introduce hybrid multi-process/thread feature.

Running Time Plot&Analysis

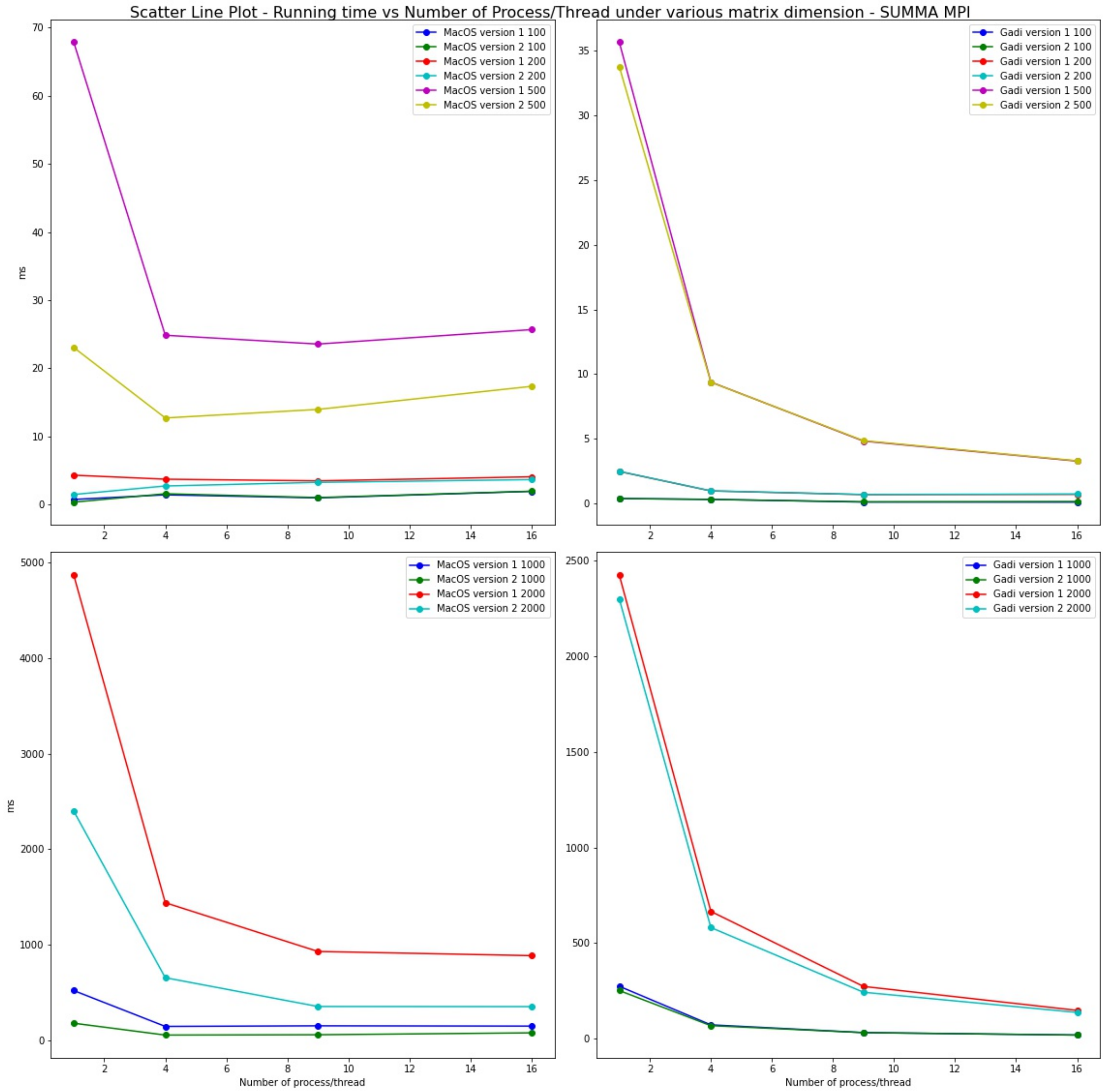


figure 5. SUMMA MPI Running Time Plot

Compare amond different platforms, notice that mac's running time is around double than the Gadi's running time among different input size with increasing processes count. When input size is large, version 2 SUMMA's running time around half of the version 1 SUMMA's running time under mac platform, but the difference is not so obvious compare to Gadi platformm which indicates that parallelising local matrix multiplication can speed up the program in the factor of running time.

Performance (Speed Up & Efficiency & Scalability) Plot&Analysis

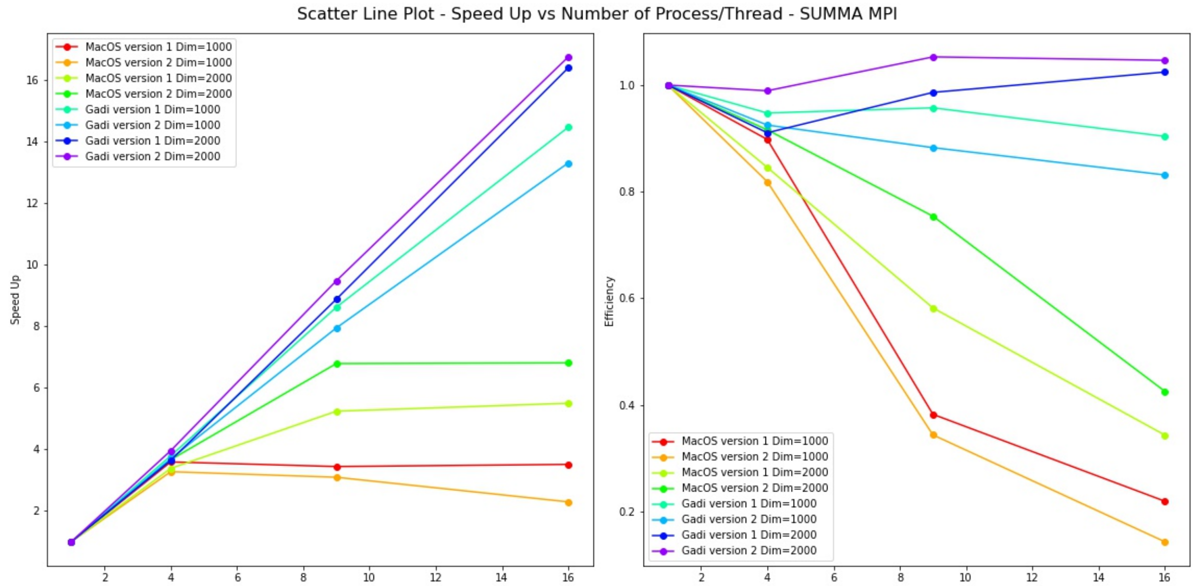


figure 6. SUMMA MPI Performance Plot

Gadi Efficiency	v1 1000	v2 1000	v1 2000	v2 2000
P=1	1	1	1	1
P=4	0.947588	0.924961	0.910588	0.989293
P=9	0.957295	0.882672	0.986253	1.052828
P=16	0.903813	0.831408	1.024404	1.046457

From the result, we fixed the input size to 2000, and notice that for both version 1 and 2, efficiency is around 0.98 - 1.1 with increasing number of processes, the efficiency is fixed during increasing process unit, which indicates that the program is strongly scalable. However, efficiency decreases a lot shown in macos platform with fixed input size and increasing number of processes, which may due to different MPI implementation and system behaviour when more processes be used for a program in different platform.

Consider the speed up, where notice that when input size=2000, the speed up for version 2 is always greater than version 1 in Gadi platform, where efficiency for Gadi version 2 also greater than version 1, in various process count, which indicates that version 2 SUMMA outperforms the version with increasing input size.

Cannon's algorithm by MPI

Cannon version 2 uses OpenMP parallelism to introduce hybrid multi-process/thread feature.

Running Time Plot&Analysis

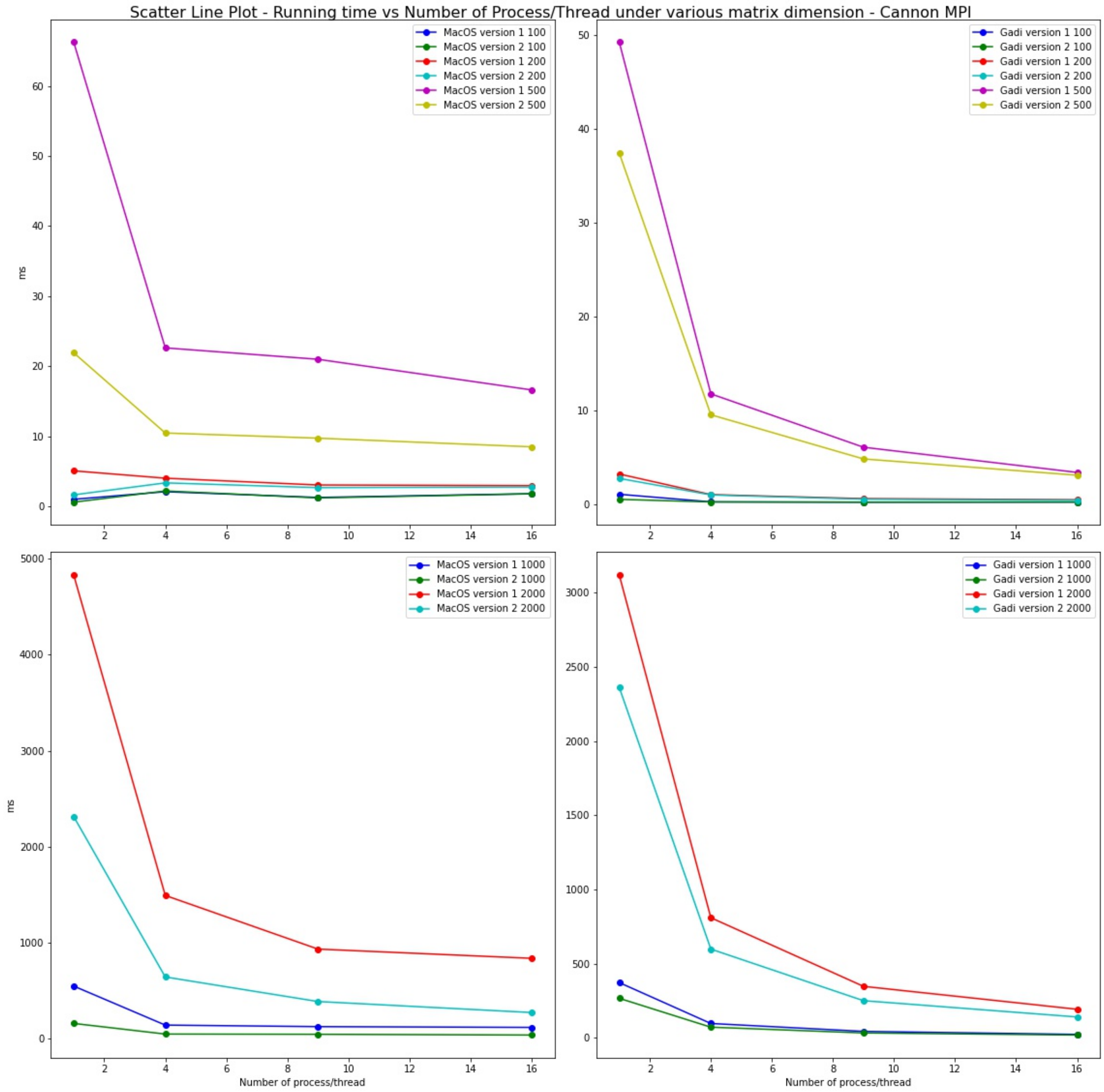


figure 7. Cannon MPI Running Time Plot

Similar to SUMMA's result, Cannon's version 2 outperforms Cannon's version 1 implementation in the order of running time with increasing processes number and input size, under both platform, where compare between MacOS and Gadi, Gadi still outperform the MacOS running time in all aspects. Compare between SUMMA and Cannon algorithm, since benchmarks only test for square matrix, Cannon slightly outperform the SUMMA in the running time aspect, by comparing two scatter plot result from each implementation.

Performance (Speed Up & Efficiency & Scalability) Plot&Analysis

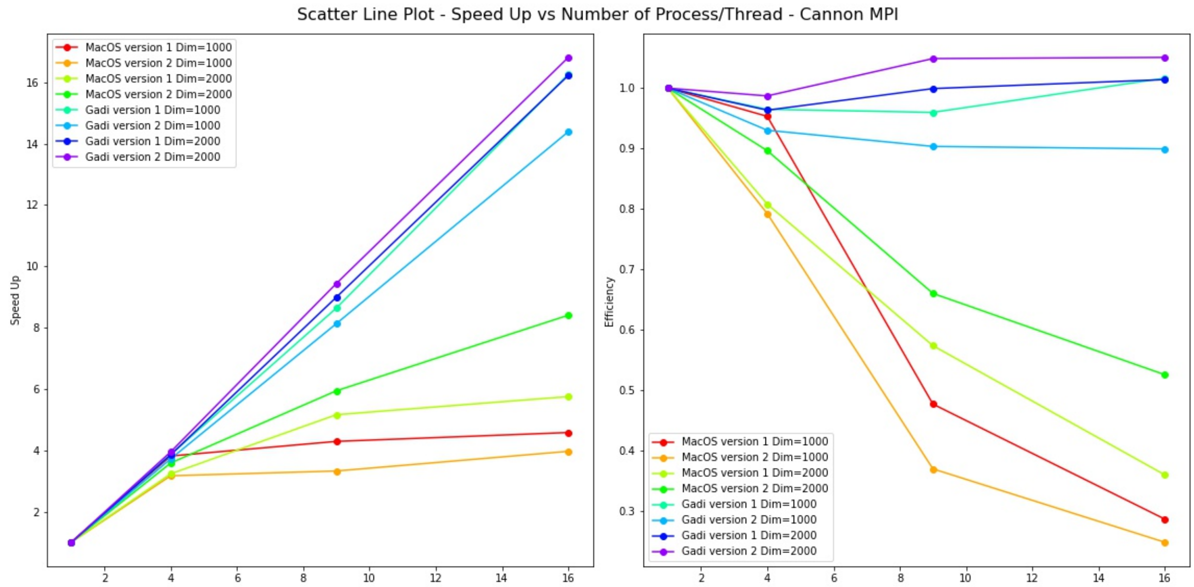


figure 8. Cannon MPI Performance Plot

Gadi Efficiency	v1 1000	v2 1000	v1 2000	v2 2000
P=1	1	1	1	1
P=4	0.964934	0.930255	0.963492	0.987122
P=9	0.959862	0.903481	0.999261	1.048956
P=16	1.016512	0.899415	1.014161	1.050897

We still consider the case when input size is fixed to 2000, for both version 1 and 2, the result shows that the efficiency is tending towards 1~1.1, which indicates the strong scalable feature of the algorithm. Whereas comparing between version 1 and version 2 in Gadi Platform, version 1 always have lower speed up and efficiency compare to version 2, which indicates that version 2 cannon outperforms version 1. Moreover, Gadi's speed up and efficiency is always greater than MacOS platform, which may due to Gadi's platform has better computation resources.

SUMMA&Cannon algorithm by Pthread

Summa(or Cannon) Pthread version 2 uses OpenMP parallelism to introduce hybrid multi-process/thread feature.

Running Time Plot&Analysis

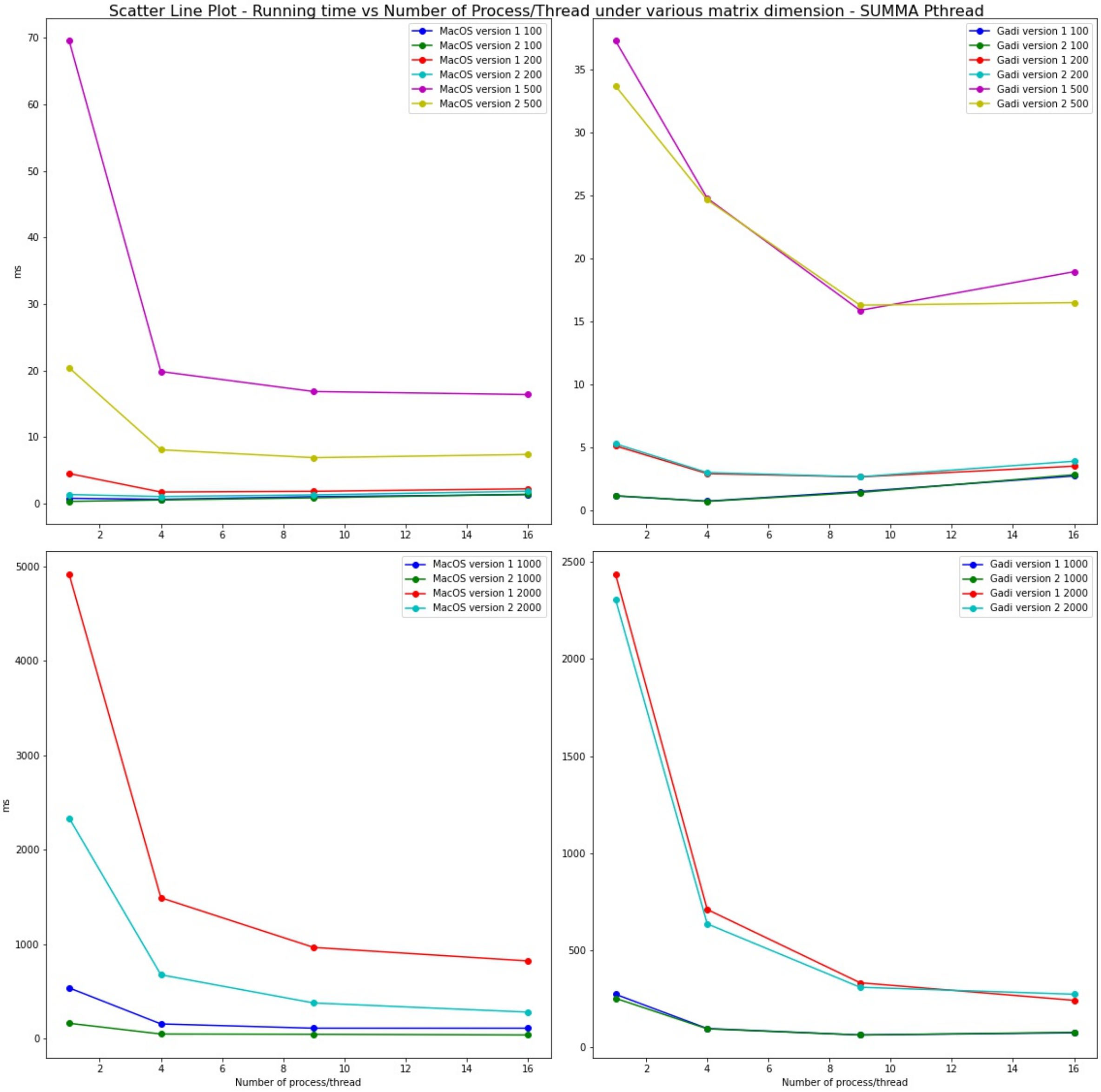


figure 9. SUMMA Pthread Running Time Plot

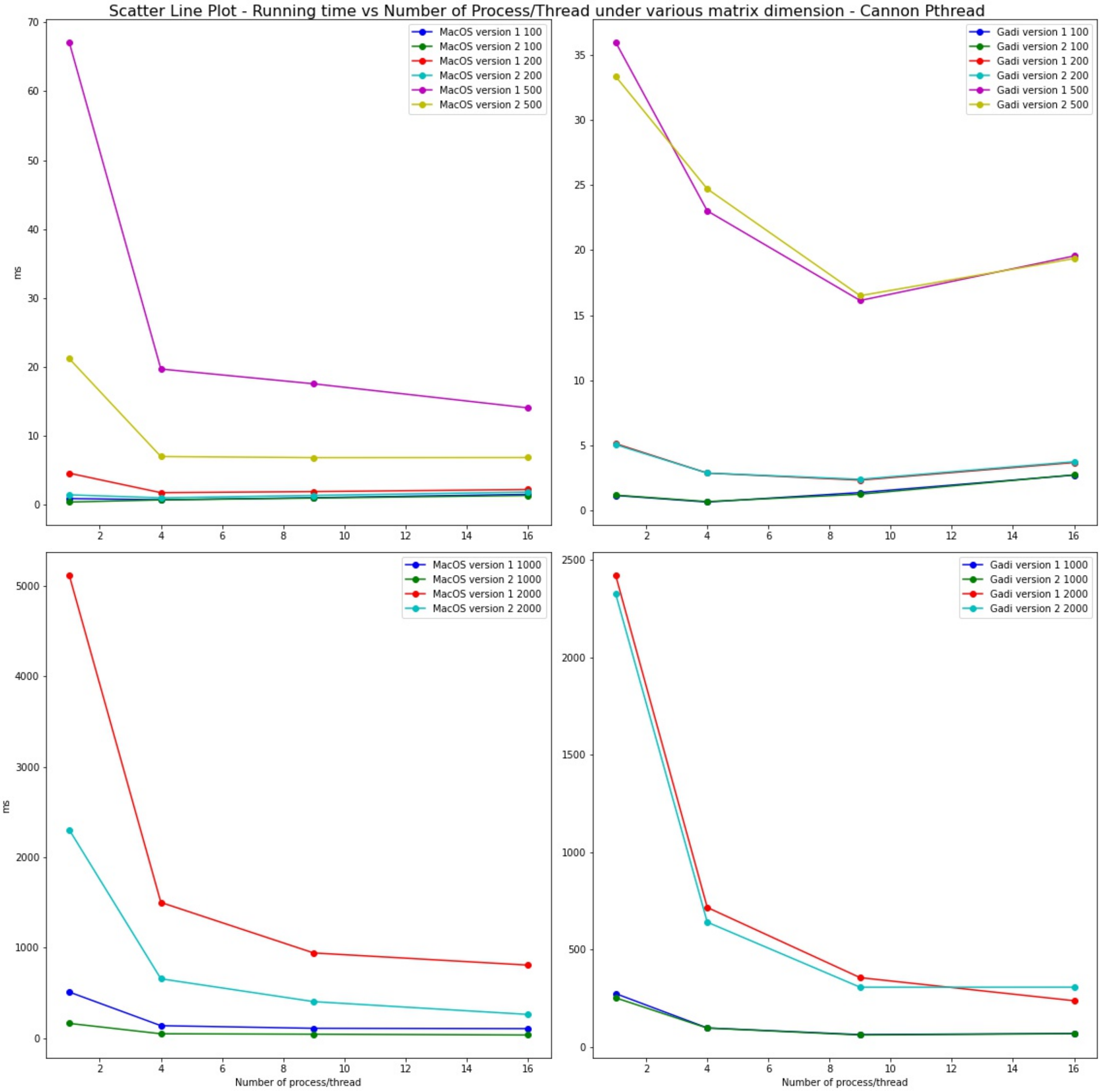


figure 10. Cannon Pthread Running Time Plot

Consider input size from 500 to 2000, where in MacOS platform, summa pthread version 2 (or cannon pthread version 2) is around halve the running time of version 1, due to hybrid openmp method, in Gadi, the difference is not so obvious, but the plot can still reflect the advantage on using openmp as assistence on local matrix multiplication.

Compare between SUMMA and Cannon Pthread, in the plot, notice that Cannon Pthread indeed outperform SUMMA Pthread when input is square matrix, but the advantage on Cannon Pthread is not so obvious (In Gadi, for 16 process and input size 2000, the SUMMA pthread version 1 elapsed time is 241.5891ms, where Cannon pthread version 1 elapsed time is 236.7551ms).

Performance (Speed Up & Efficiency & Scalability) Plot&Analysis

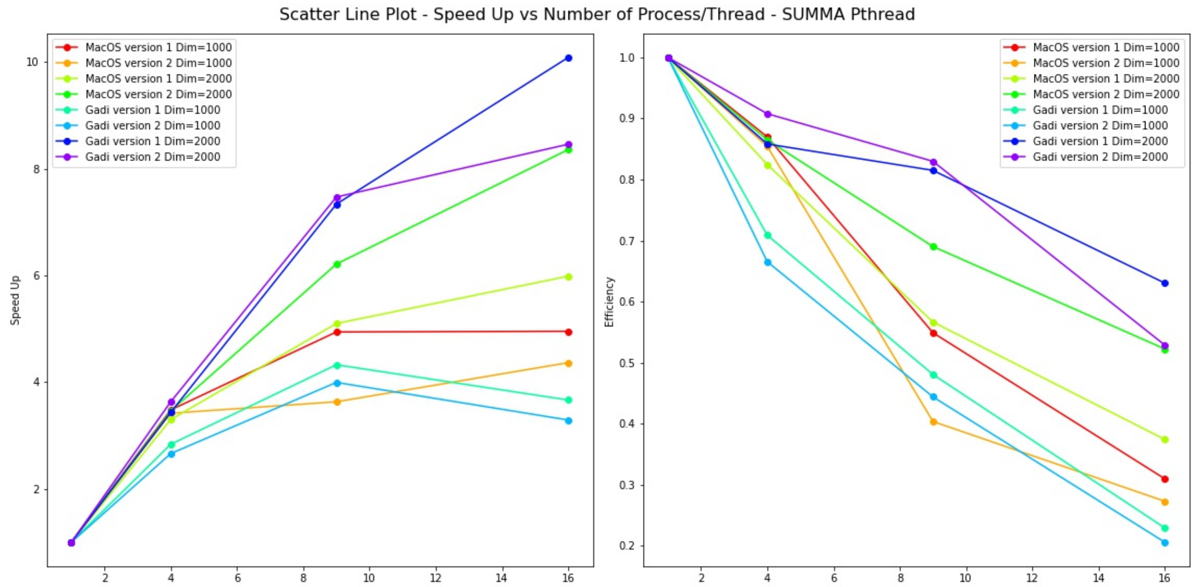


figure 11. SUMMA Pthread Performance Time Plot

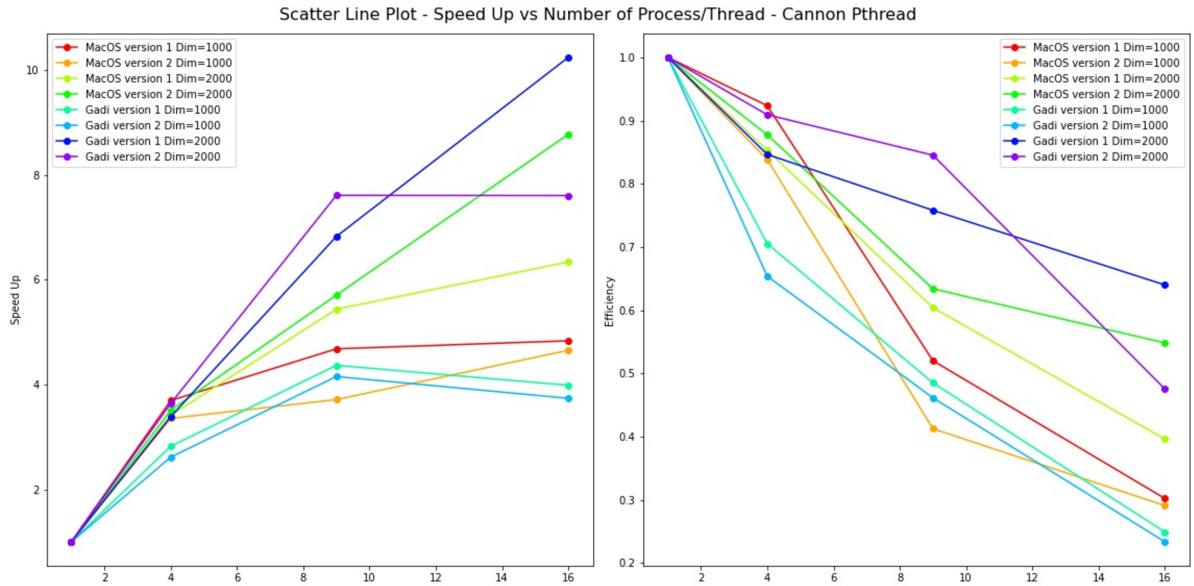


figure 12. Cannon Pthread Performance Plot

Consider the efficiency plot for both SUMMA pthread and Cannon pthread, notice that the efficiency with fix input size and increasing number of thread clearly lead to efficiency drop, whereas when input size from 1000 to 2000, and number of thread from 4 to 9, the efficiency is not fixed and inverse proportional to the increasing input size and number of threads, which indicates that both SUMMA Pthread and Cannon Pthread are not scalable.

Part B

Parallel Sample Sort MPI

The version one MPI sample sort uses `MPI_Send`/`MPI_Recv` for most of the communication, and all the process should wait for master process to obtain the sample and send to them before generating local splitter, whereas verions 2 uses various advanced MPI Communcation method includes `MPI_Alltoallv`, `MPI_Scatterv`, `MPI_gatherv` to speed up the communcation, where process don't need to wait for master process to do the sampling, but master process will distribute the portion first and let all processes to generate sample in parallel.

Running Time Plot&Analysis

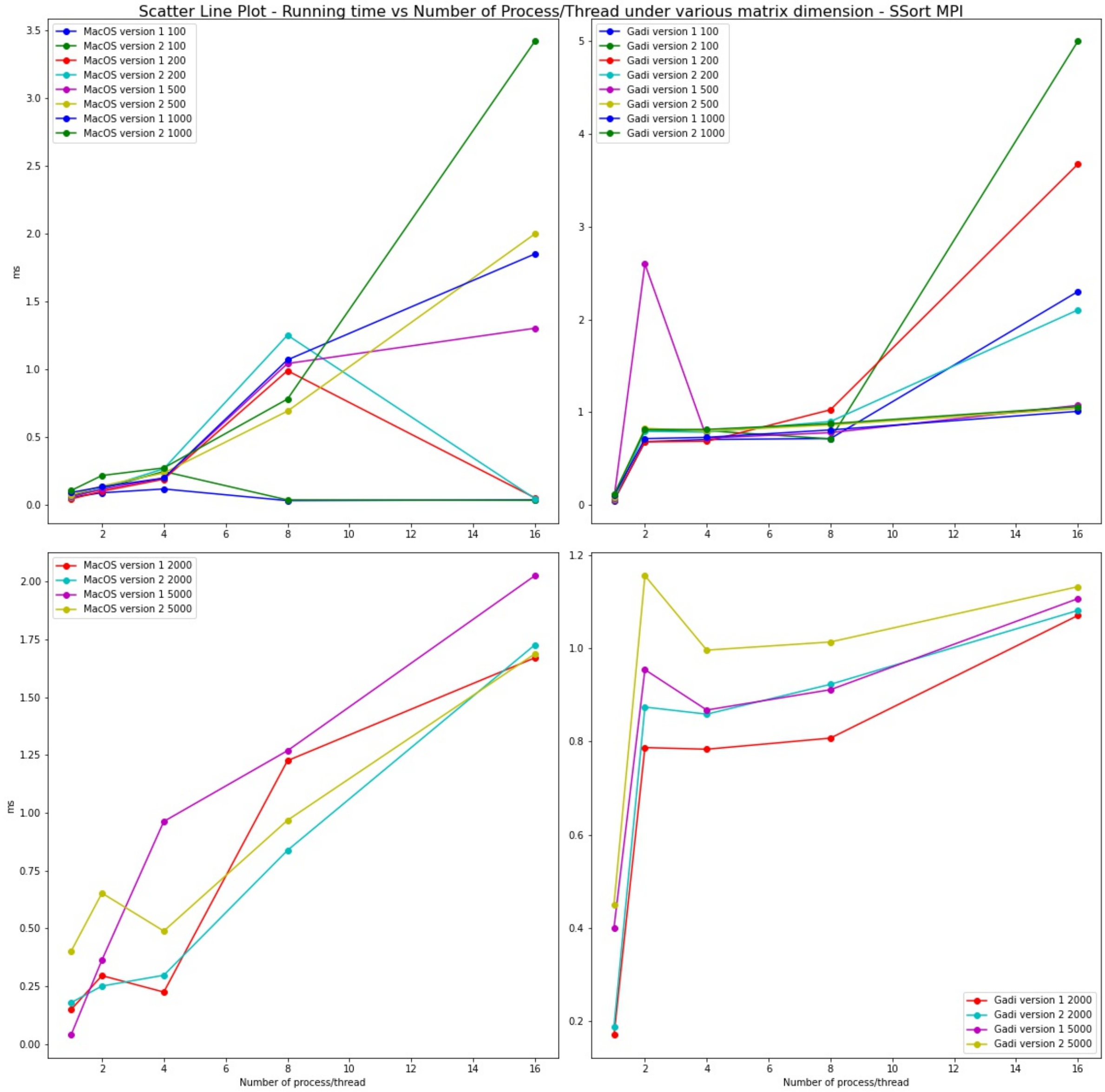


figure 13. Sample Sort MPI Running Time Plot

Consider the running time result, note that verions 2 has lower running time compare to version when input size >= 500, where Gadi running time still outperform MacOS's running time when input size is large.

One aspect is quite surprising that the running time is not decrease when number of processes increase with large input size (>=500), one reason for that can be since the input data is generated using `Random()` upper bounded by `1000`, the result may lead to a lot of duplicated elements, where during the local sampling from the global list, sample data from the global list may not perfectly align the structure of the data, which would lead to local bucket to be unevenly distributed. Also, the program ran in one process even outperform multi process, one reason could be since the data in each local bucket is not distributed evenly due to randomness, which introduces more communcation overhead to send/recv more elements, whereas single process don't need to communicate and can directly apply `quick sort` to sort the global list.

However, when input size = 200 in MacOS platform, the running time indeed decreased with increasing process number, which indicates that the program may still speed up the linear process in some cases when random sampling can reflect the global population.

Performance (Speed Up & Efficiency & Scalability) Plot&Analysis

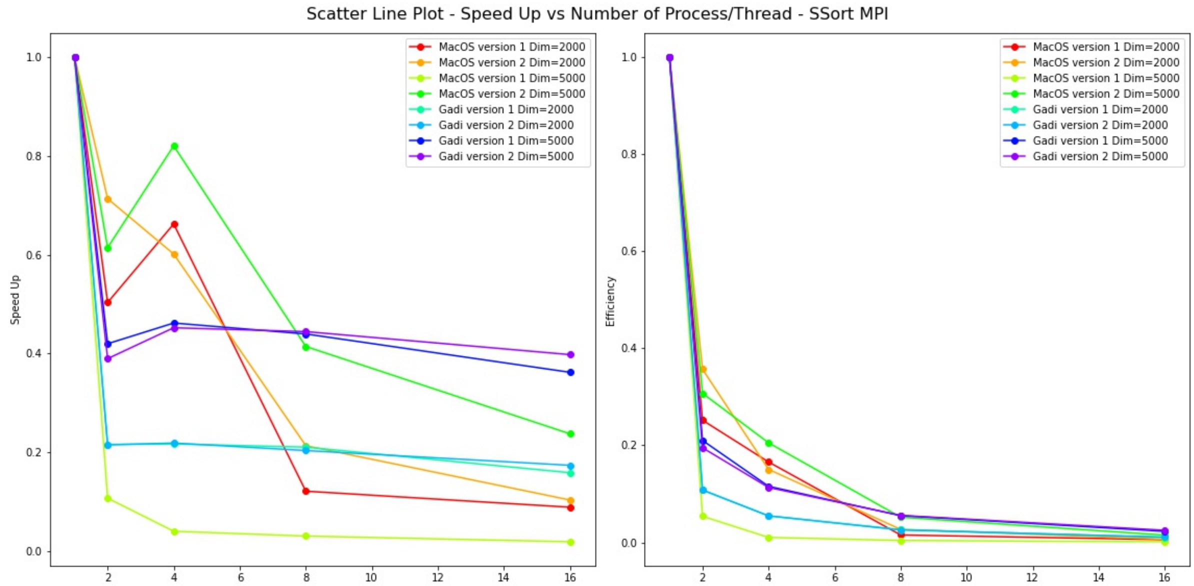


figure 14. Sample Sort MPI Performance Plot

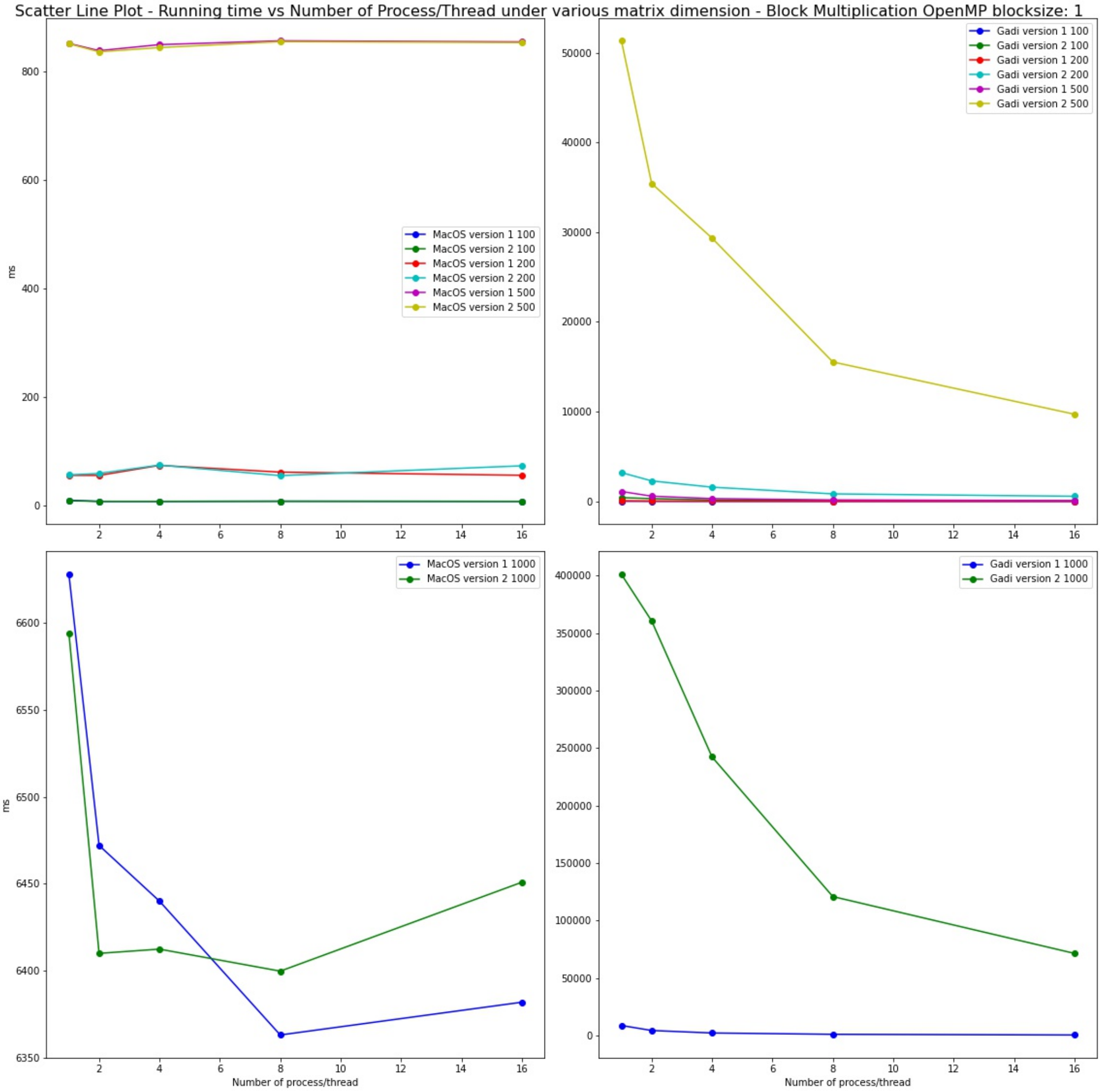
Gadi Efficiency	v1 2000	v2 2000	v1 5000	v2 5000
P=1	1	1	1	1
P=2	0.108071	0.107580	0.209914	0.194835
P=4	0.054279	0.054745	0.115428	0.113042
P=8	0.026337	0.025474	0.054952	0.055547
P=16	0.009932	0.010870	0.022626	0.024864

From the running time analysis, it is not surprising that the speed up even lead to 0 when input size is 5000, which due to uneven random sampling and communication overhead.

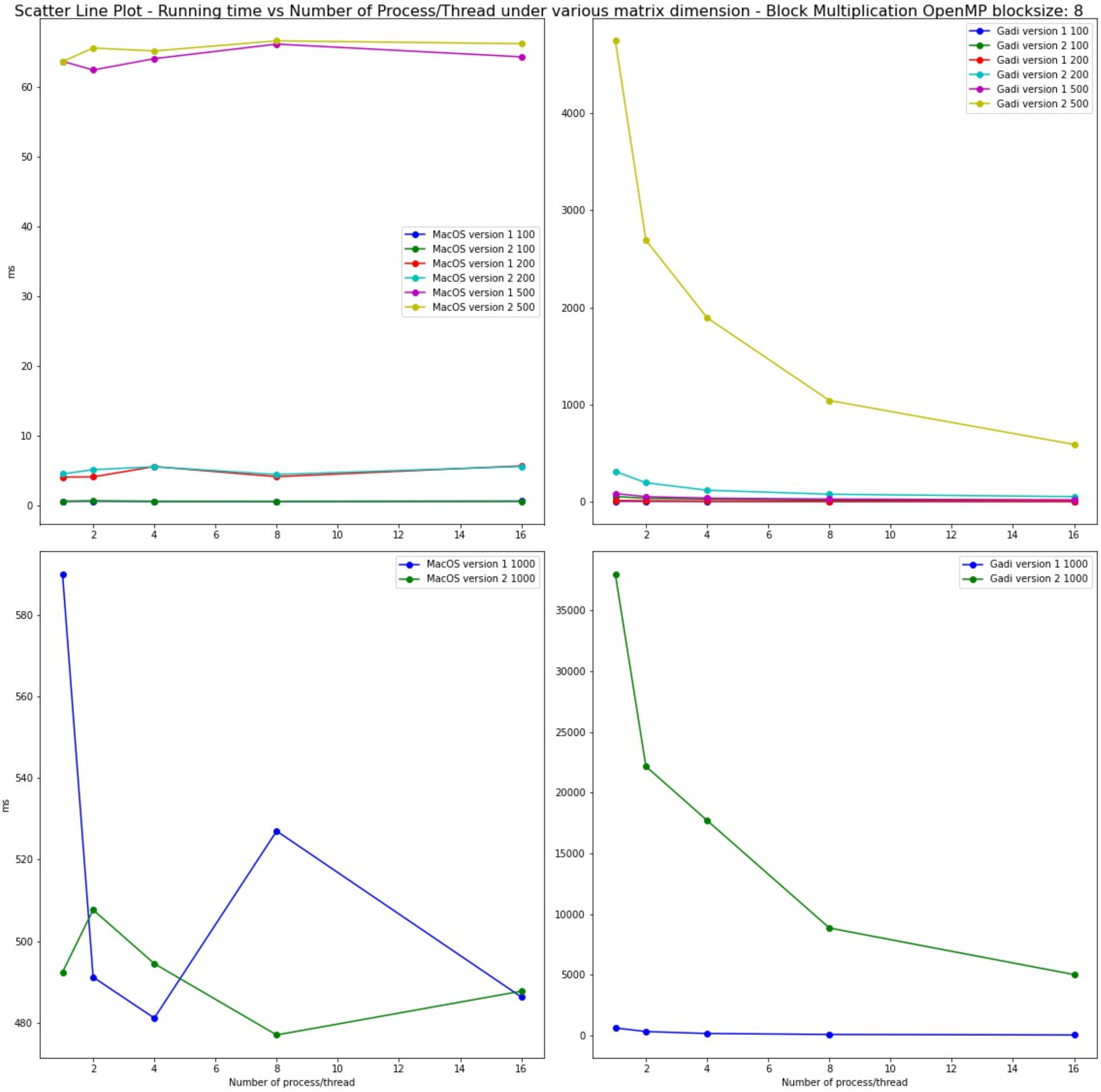
For the efficiency part, the result didn't provide the evidence that the program is strongly scalable, but it shows the weak scalability, consider the number of process is increased from 4 to 8, where input size is from 2000 to 5000, notice that for both version 1 and 2, the efficiency is fixed respectively.

In addition, Gadi's efficiency&speed up still outperform the MacOS platform result, as previous analysis shown. Moreover, version 2 still outperform version 1 samplesort regard to efficiency and speed up (for size 5000, v1 efficiency for p = 16 is 0.023 < 0.025 for v2 p = 16).

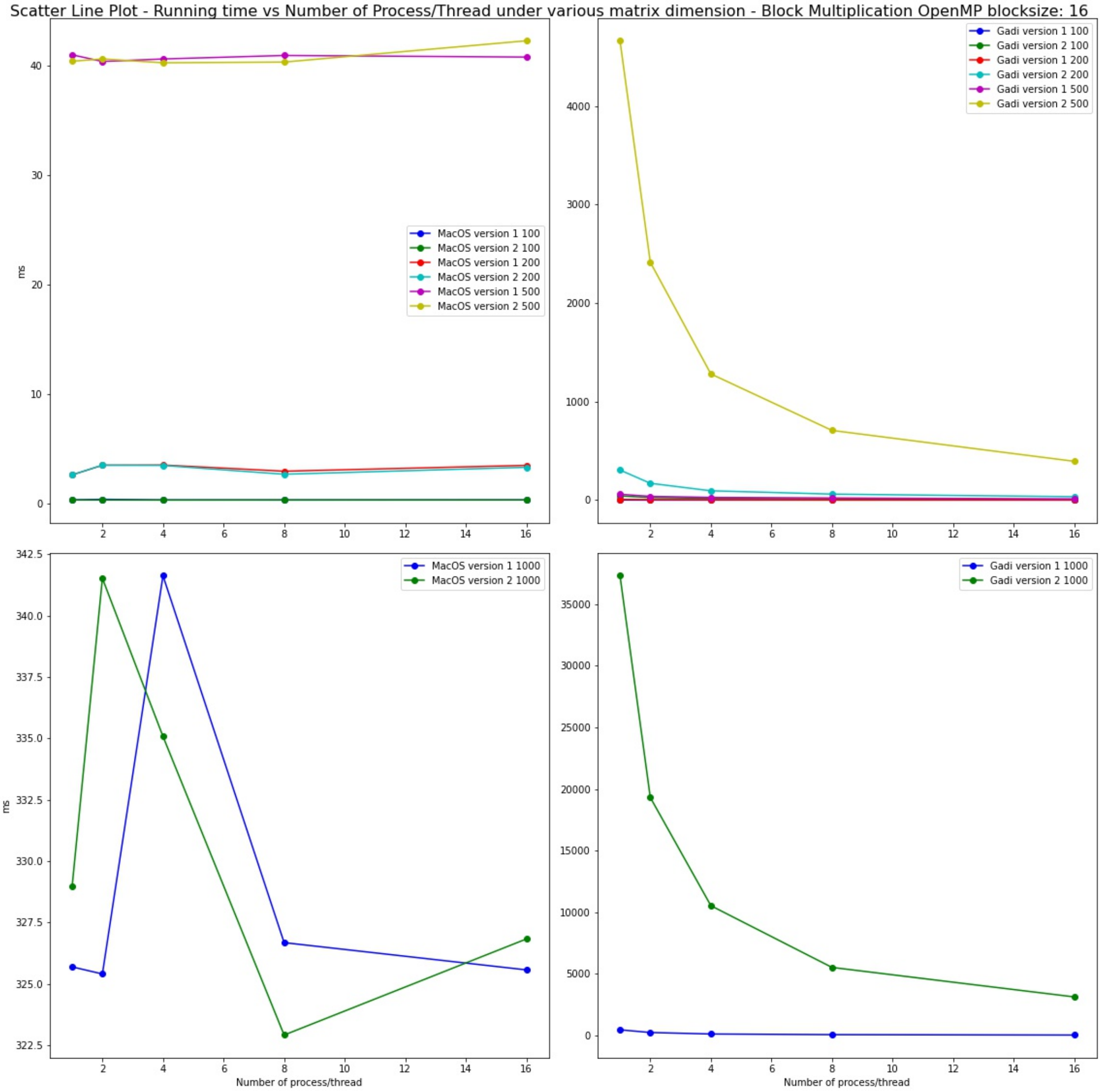
Supplimentry Material



suppl 1. Block Multiplication Running Time Plot, Block size 1



suppl 2. Block Multiplication Running Time Plot, Block size 8



suppl 3. Block Multiplication Running Time Plot, Block size 16