Task 1

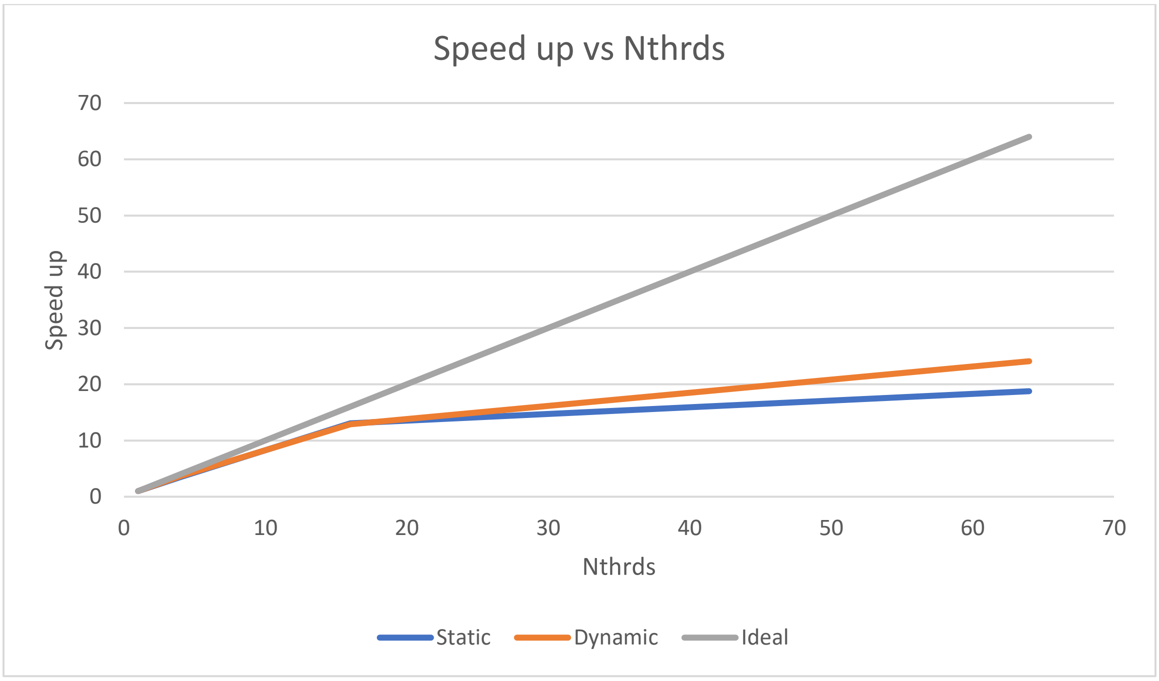
Strong scaling

Static Scheduling

Problem size: 1000 x 1000, Default blocksize: 2			
Nthrds	Execution time (s)	Speed up	Efficiency
1	1.826827e+00	1.00	1.00
4	5.148317e-01	3.51	0.89
16	1.398786e-01	13.06	0.81
64	9.791307e-02	18.76	0.29
256	1.516749e-01	12.13	0.04

Dynamic Scheduling

Problem size: 1000 x 1000, Default blocksize: 2			
Nthrds	Execution time (s)	Speed up	Efficiency
1	1.812126e+00	1.00	1.00
4	4.968913e-01	3.65	0.93
16	1.404060e-01	12.91	0.81
64	7.522422e-02	24.09	0.37
256	1.316195e-01	13.77	0.05



Strong scaling analysis

The two versions of Task 1 that I have implemented are block matrix multiplication with static parallel-for scheduling and dynamic parallel-for scheduling.

Based on the speed-up vs number of threads graph, it can be seen that both implementations' speed-up plateaus past 16 threads. Therefore, adding more threads past this point does not offer significant speed-up. This is expected due to several reasons. First is due to false sharing, since problem size is constant, adding more threads would mean that each thread will have higher chances of having its matrix elements sit on the same cache line. This will cause more cache updates and force the threads to access the memory. The second is due to Amdahl's law. When the number of threads is sufficiently high, the speed-up gain is equal to 1/(1-P), where P is the parallel part of the code. Since the code is not 100% parallelisable, there is a limit for speed-up gains.

Based on the graph, the dynamic scheduling version scales better than the static scheduling version. However, this difference is only observable on a big number of threads(16 threads). This is expected because each thread can request loop iterations during runtime. This offers better load balancing at a high number of threads and consequently faster execution time.

Weak scaling

Static scheduling

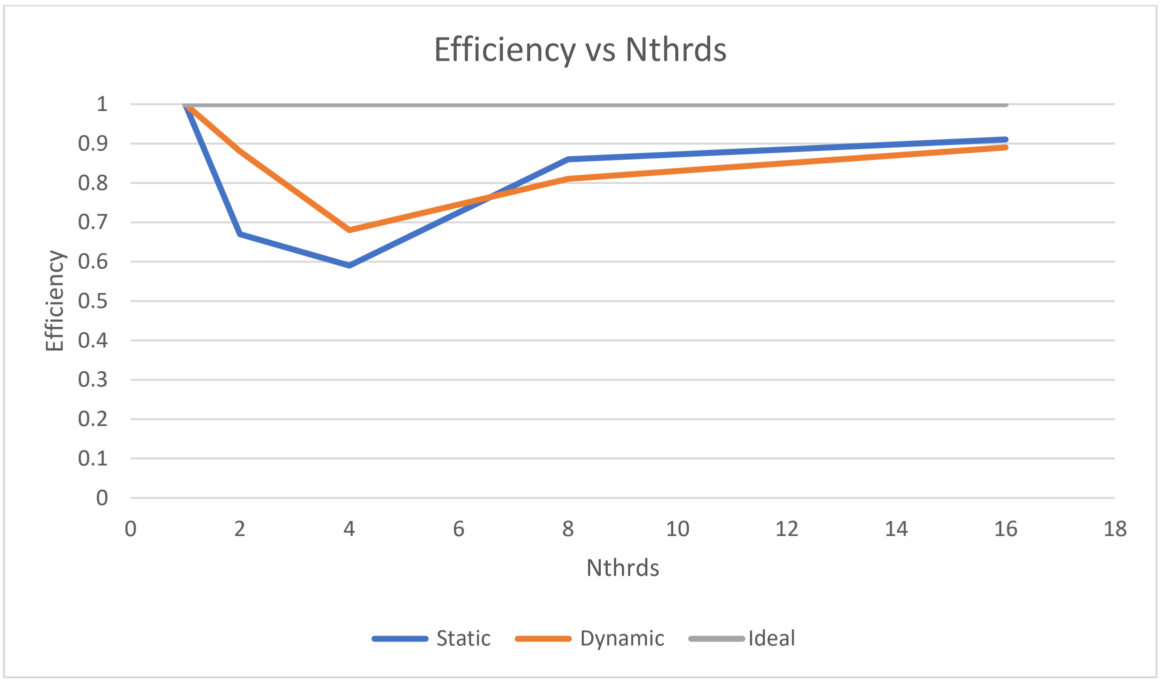
Nthrds	Matrix dimension				
	100	200	400	800	1600
1	1.00	1.00	1.00	1.00	1.00
2	0.97	0.67	0.78	0.99	0.95
4	0.84	0.57	0.59	0.87	0.93
8	0.57	0.52	0.45	0.86	0.91
16	0.61	0.41	0.32	0.76	0.91

Nthrds	Execution time (s)	Efficiency	Problem size
1	5.820384e-03	1.00	100x100
2	1.959703e-02	0.67	200x200
4	4.840297e-02	0.59	400x400
8	1.400664e-01	0.86	800x800
16	5.027097e-01	0.91	1600x1600

Dynamic scheduling

Nthrds	Matrix dimension				
	100	200	400	800	1600
1	1.00	1.00	1.00	1.00	1.00
2	0.88	0.63	0.78	0.94	0.94
4	0.76	0.51	0.68	0.85	0.94
8	0.51	0.41	0.48	0.81	0.90
16	0.23	0.35	0.037	0.73	0.89

Nthrds	Execution time (s)	Efficiency	Problem size
1	5.537961e-03	1.00	100x100
2	1.765553e-02	0.88	200x200
4	4.381583e-02	0.68	400x400
8	1.412876e-01	0.81	800x800
16	5.161379e-01	0.89	1600x1600



Weak scaling analysis

Even though the strong analysis shows that speed-up gain peaked past 16 threads, it does not mean that adding more threads is useless. As can be seen from the efficiency table, increasing the problem size together with threads offers better efficiency. However, the weak scaling is not near the ideal. From the efficiency vs number of threads graph, the efficiency dropped significantly for the static version after just increasing the number of threads to 2 and doubling the problem size. The trend continues at threads=4 and problem size=400x400. It improves past this, and the efficiency plateaus at around 0.9 for both versions.

Unlike the previous analysis, both versions of the implementation offer roughly the same weak scaling. The static version performs worse initially, but it outperforms the dynamic version at threads=8 and problem size=800x800.

Block size comparison

Number of threads: 16, Problem size 800x800

Static scheduling

Block size	Execution time (s)	Speed up
1	2.155735e-01	1.00
2	7.947073e-02	2.70
4	6.086540e-02	3.59
8	6.058515e-02	3.56
16	6.405387e-02	3.36

Dynamic scheduling

Block size	Execution time (s)	Speed up
1	2.249583e-01	1.00
2	7.939889e-02	2.83
4	6.105228e-02	3.68
8	6.212184e-02	3.62
16	6.070364e-02	3.71

Block size analysis

For both versions, increasing the block size increases the speed-up linearly. However, this holds up until the block size of 4. Increasing the block size further does not offer any considerable speed-up. This algorithm assumes that the cache can hold every matrix element of the block in the cache. When b is too large, the algorithm cannot fully take advantage of the cache because the matrix elements are too little. Inversely, when b is too small, the cache can no longer store all the elements, hence the elements might be stored in a slower cache or memory. The best block size here is 4.

Other platform performance comparison

Platform: Personal computer

OS: Linux (Virtual Machine)

CPU: Ryzen 9 4900H(8-threads) Overall cache size:12.5MB Memory: 8GB

Static scheduling

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	1.090718e+00s	1.251788e+00s
4	3.198426e-01s	3.504540e-01s

Dynamic scheduling

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	1.093034e+00s	1.285576e+00s
4	3.071215e-01s	3.456624e-01s

Overall Gadi outperforms my personal computer on both versions of the implementation. Since block matrix multiplication depends on storing blocks of the matrix in the cache, the performance will vary with different cache sizes. Since Gadi has a bigger cache size (35.75MB vs 12.5MB), the algorithm runs better on Gadi.

Task 2

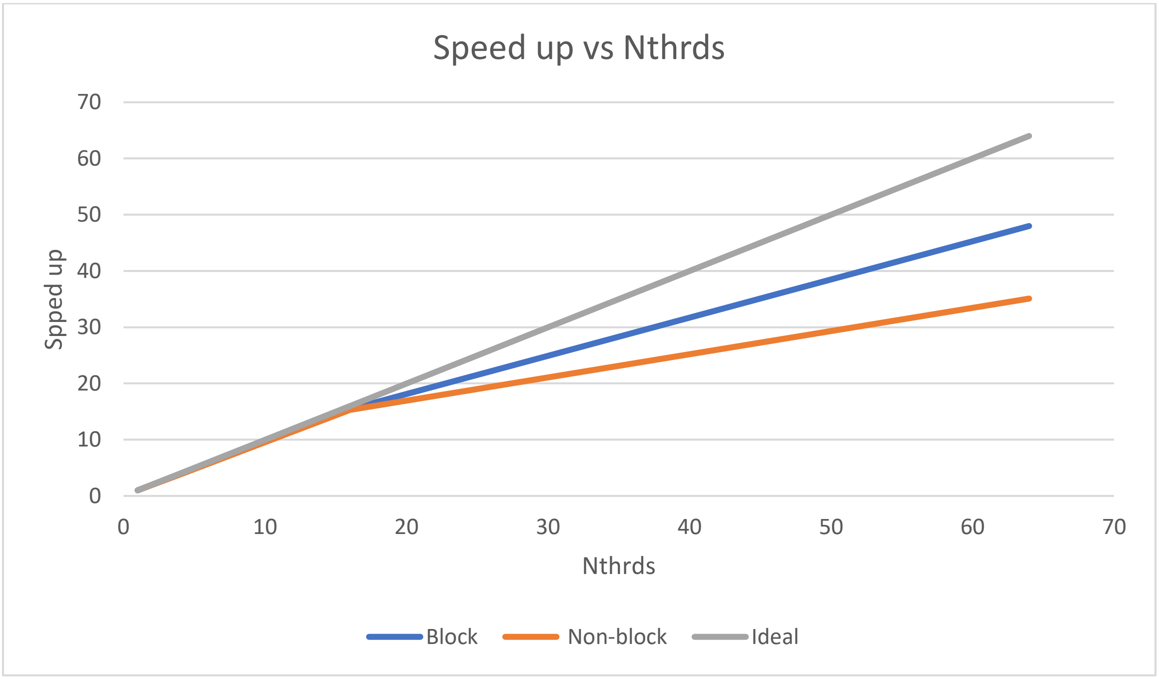
Strong scaling

Blocking

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.651839e+00	1.00	1.00
4	9.288464e-01	3.93	0.98
16	2.367640e-01	15.42	0.96
64	7.607873e-02	47.99	0.75

Non-blocking

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.681730e+00	1.00	1.00
4	9.699209e-01	3.80	0.95
16	2.404390e-01	15.31	0.95
64	1.049200e-01	35.09	0.55



Strong scaling analysis

I have implemented two versions of SUMMA using MPI. One version uses blocking send and receive while the other uses the non-blocking version. For the non-blocking version, the manual **Wait()** operation is used to synchronise the communication.

From this algorithm's strong scaling graph, it can be seen that this algorithm offers better strong scalability than block matrix multiplication. Specifically, this algorithm's speed-up scales closer to the ideal. This is probably due to having more efficient communication. Not just that, since MPI is a distributed memory paradigm, common pitfalls such as false sharing can be avoided.

Overall, the blocking version performs and scales better than the non-blocking version. Therefore, we can speculate that the blocking send and receive is faster than the non-blocking version with manual wait operation. This might be because **Wait()** operation has to check on the parameter &**request** before continuing the thread's execution, thus adding slightly more waiting times. The increased wait time contributes to higher total latency between communication.

Weak scaling

Blocking

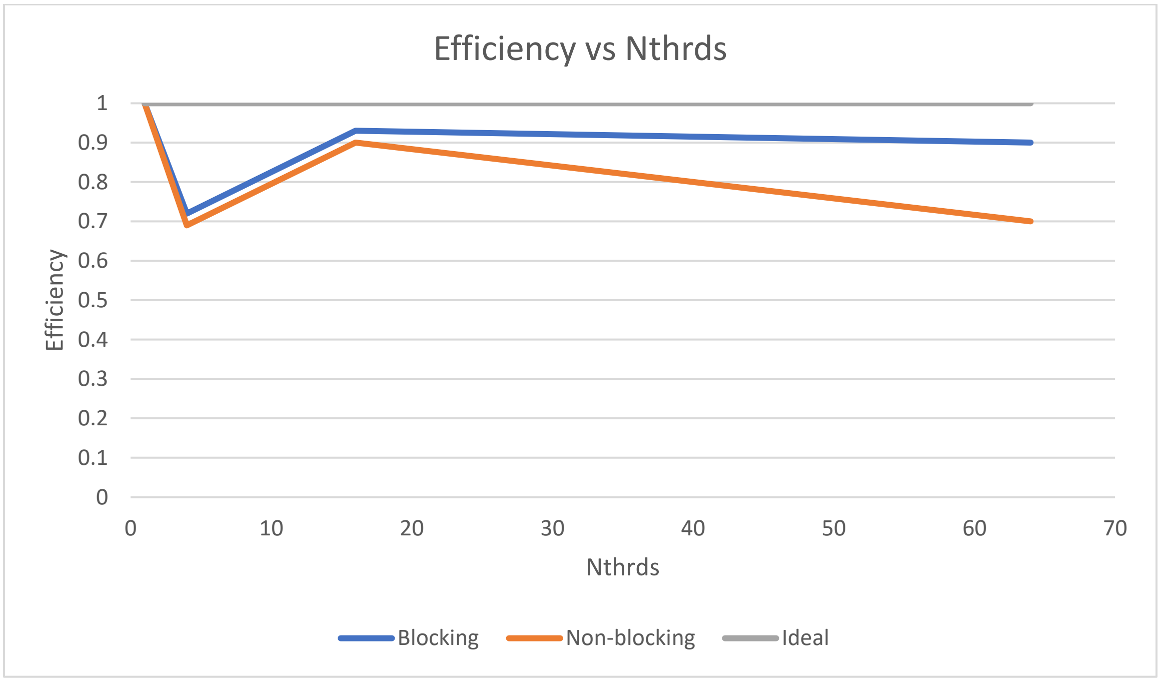
Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.20	0.72	0.97	0.98
16	0.02	0.40	0.93	0.99
64	0.0003	0.005	0.24	0.90

Nthrds	Execution time (s)	Efficiency	Problem size
1	8.795100e-05	1.00	24x24
4	1.224407e-03	0.72	96x96
16	1.640834e-02	0.93	384x384
64	2.679027e-01	0.90	1536x1536

Non-blocking

Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.22	0.69	0.96	0.98
16	0.03	0.42	0.90	0.99
64	0.0004	0.005	0.26	0.70

Nthrds	Execution time (s)	Efficiency	Problem size
1	9.486100e-05	1.00	24x24
4	1.444974e-03	0.69	96x96
16	1.708740e-02	0.90	384x384
64	3.481885e-01	0.70	1536x1536



Weak scaling analysis

Same as the previous algorithm, the efficiency drops when threads=4 and problem size=96x96. However, it goes back up from there at threads=16 and problem size=384x384. The blocking version's efficiency plateaus at an efficiency of 0.90 while the non-blocking version's efficiency goes down to 0.70. Therefore, the blocking version is weaklier scalable than the non-blocking version.

The difference in performance is probably due to the aforementioned **Wait()** operation's overhead. Increasing the number of threads consequently increases the number of communications. Accordingly, the latency of the communication increases as well. Increasing the problem size does not help in reducing or masking this effect since problem size only affects bandwidth, not the latency between send and receive operations.

Other platform performance comparison

Blocking

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.664144e+00s	3.067378e+00s
4	9.699021e-01s	8.367918e-01s

Non-blocking

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.003399e+00s	3.629637e+00s
4	8.429875e-01s	9.376417e-01s

Overall, the performance difference is a mixed bag. The blocking version performs better on Gadi while the non-blocking version performs better on my PC. The performance difference here is probably due to the different inter-core communication layout and network topology each platform's CPU has.

Task 3

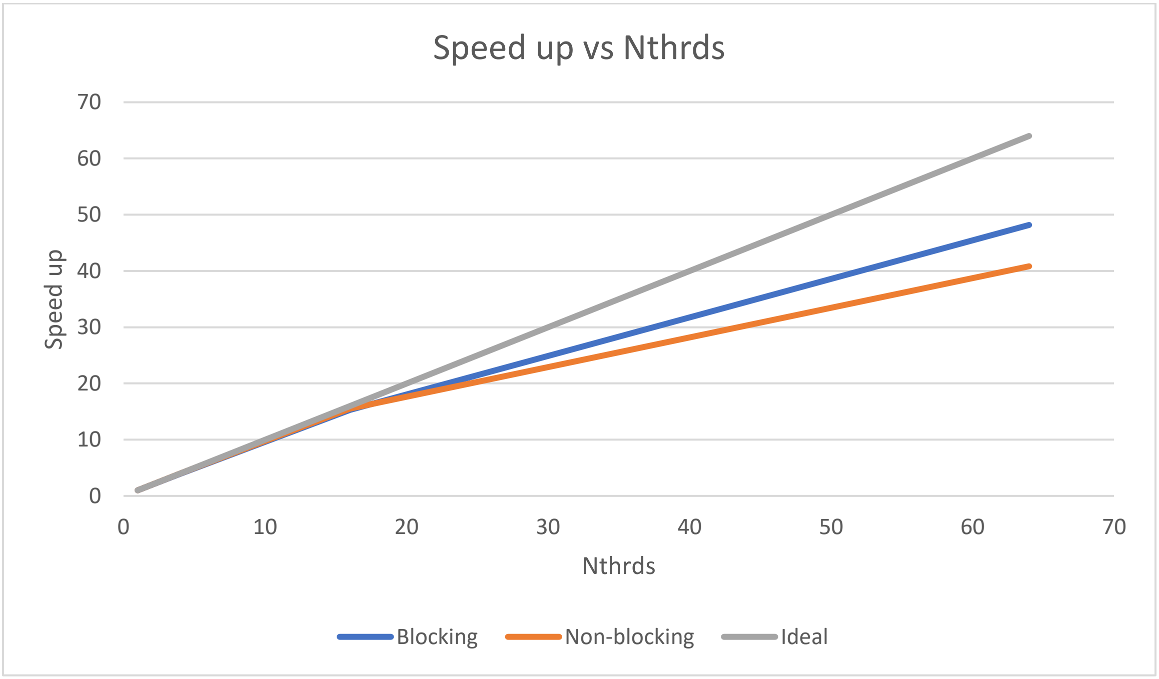
Strong scaling

Blocking

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.620370e+00	1.00	1.00
4	9.241169e-01	3.92	0.98
16	2.361742e-01	15.33	0.96
64	7.513449e-02	48.18	0.75

Non-blocking

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.722918e+00	1.00	1.00
4	9.371984e-01	3.97	0.99
16	2.400343e-01	15.51	0.97
64	9.115878e-02	40.83	0.64



Strong scalability analysis

One version of the Cannon's algorithm I have implemented uses the normal blocking send and receive operations. On the other hand, the second version uses non-blocking send and receive with a manual **Wait()** operation. This is similar to the previous task.

Based on the speed-up vs number of threads graph, the speed-up scales linearly up until threads=16. From that point onwards, the speed-up scales with a gradient lower than the ideal speed-up. Same as SUMMA, this algorithm is more strongly scalable compared to block matrix multiplication.

The blocking version of the implementation scales better than the non-blocking version. Again, this is probably due to the same reasoning; increased latency due to the **Wait()** operation. One thing to note is that this algorithm scales almost similar to SUMMA. Moreover, the execution time is also about the same. This similarity is probably due to both algorithms having theoretically identical time complexity and a nearly identical number of words moved

Weak Scaling

Blocking

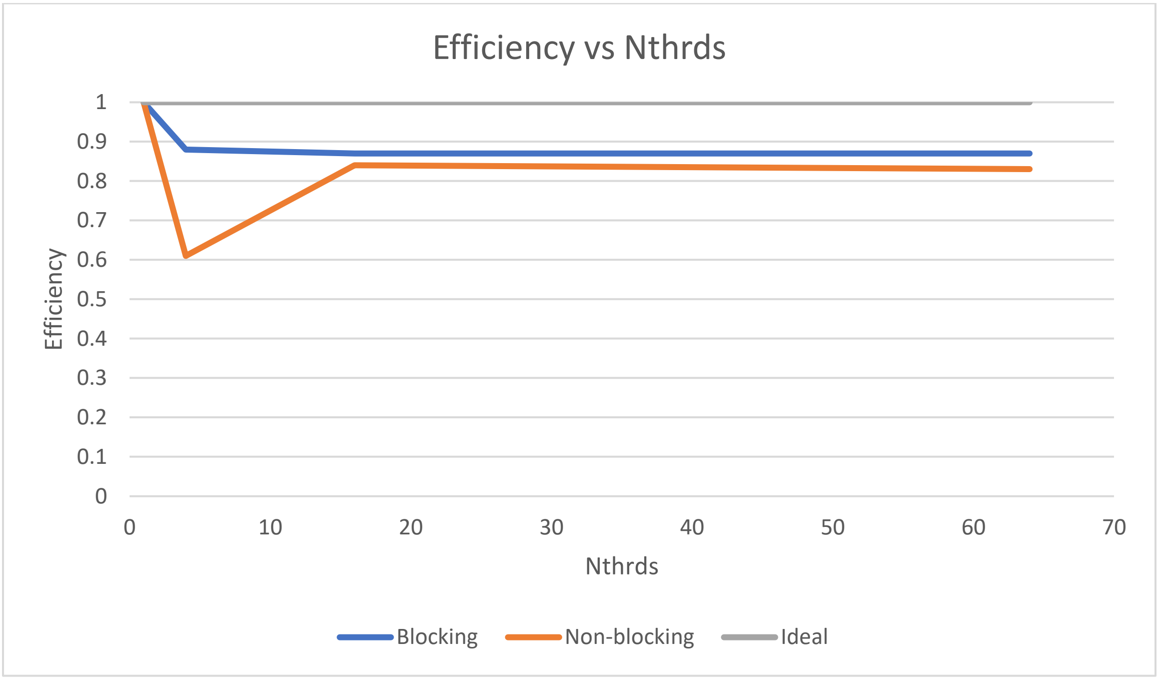
Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.10	0.88	1.00	0.99
16	0.02	0.40	0.87	0.97
64	0.0008	0.007	0.26	0.87

Nthrds	Execution time (s)	Efficiency	Problem size
1	9.689700e-05	1.00	24x24
4	1.095417e-03	0.88	96x96
16	1.672359e-02	0.87	384x384
64	2.641804e-01	0.87	1536x1536

Non-blocking

Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.09	0.61	1.00	0.98
16	0.03	0.34	0.84	0.95
64	0.0005	0.01	0.17	0.83

Nthrds	Execution time (s)	Efficiency	Problem size
1	9.587000e-05	1.00	24x24
4	1.609051e-03	0.61	96x96
16	1.756864e-02	0.84	384x384
64	2.837349e-01	0.83	1536x1536



Weak scaling analysis

From the above graph, two trends can be observed. For the blocking version, the efficiency dropped as soon as the number of threads and problem size grew. Then, the efficiency stayed constant at 0.88. The non-blocking version, on the other hand, has its efficiency dropped, risen, and plateaus. The efficiency dropped at threads=4 and problem size=24x24, rose at threads=16 and problem size=96x96, and finally plateaus after that.

Overall, the blocking version performs better than the non-blocking one. As previously, this might be due to the added latency of the **Wait()** operation. However, this difference is only significant at a lower number of threads. At a bigger number of threads, the performance difference is minor. Therefore, the added latency is not relevant when threads are high. In SUMMA, each thread has to broadcast to its neighbouring column or row threads hence the latency adds up. Meanwhile, in cannon’s algorithm, the communication is one-to-one between partnering threads. For this reason, the latency doesn’t add up and results in better scaling.

Other platform performance comparison

Blocking

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.528307e+00s	3.204122e+00s
4	9.426624e-01s	8.845780e-01s

Non-blocking

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.601712e+00s	3.270846e+00s
4	9.614085e-01s	8.824139e-01s

Weirdly, both versions ran better on my PC compared to Gadi. If we combine this result with the performance comparison from the previous task, we can conclude that MPI runs better on my PC's CPU. This is perhaps due to the better inter-core communication on AMD Ryzen compared to Intel Xeon. It also may be due to MPI communication preferring Ryzen's core network topology.

Task 4A

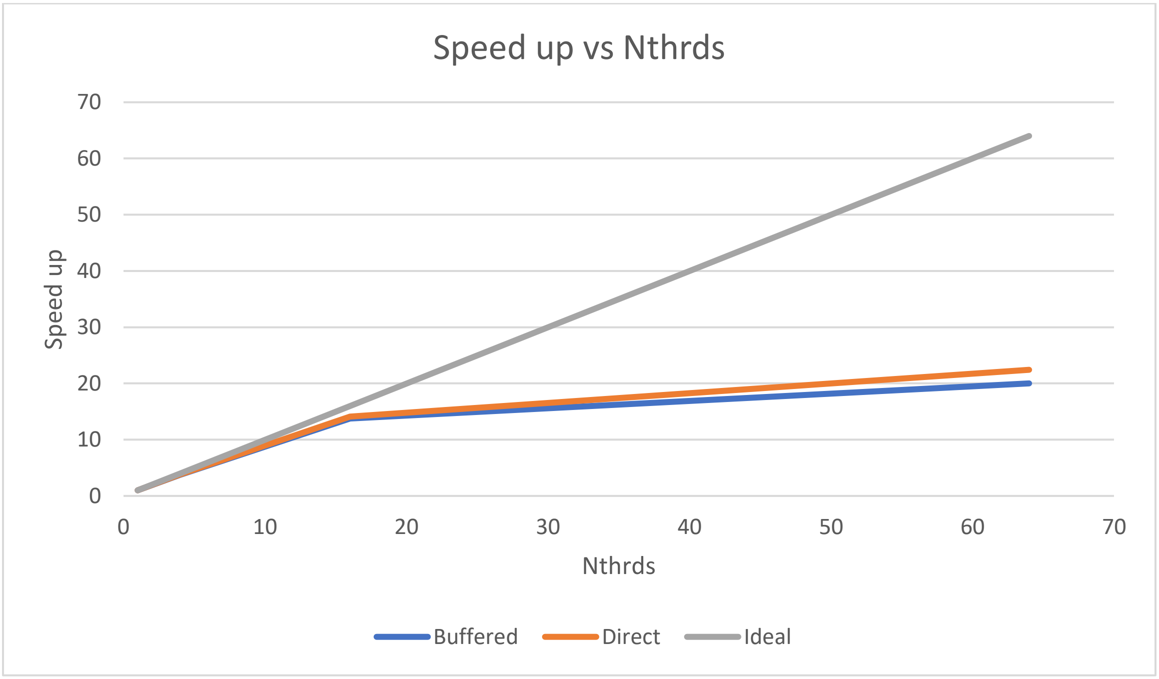
Strong scaling

Buffered

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.632057e+00	1.00	1.00
4	9.728301e-01	3.73	0.93
16	2.643721e-01	13.75	0.86
64	1.806979e-01	20.01	0.31

Direct

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.789078e+00s	1.00	1.00
4	9.961960e-01	3.80	0.95
16	2.685609e-01	14.11	0.88
64	1.689169e-01	22.43	0.35



Strong scaling analysis

The two versions of SUMMA in pthreads that I have implemented are buffered and direct. Direct means that the results of the multiplications are directly written on the C matrix. Meanwhile, buffered means that each thread has its local copy of the C matrix, and results are combined at the very end when the algorithm has finished.

Unlike SUMMA in MPI, this pthread version scales much worse. The initial scaling is similar up to threads=16 but changes drastically at threads=64. At threads=64, the efficiency of the speed-up is only on average 0.3 while the MPI version averages more than double at 0.7. The performance difference is probably due to the different paradigms of both APIs. Since Pthread is a shared memory paradigm, it is susceptible to false sharing and performs worse when results have to be combined at the end. On the flipside, MPI has a very optimized collective communication. Operations like **gather()** can be used to combine results from all threads without having to worry about false sharing. Moreover, the collective communication in MPI also has better time complexity. The gather operation has a time complexity of O(log P+n) while the pthread version that I have implemented has a complexity of O(n).

When comparing the buffered and direct versions, there is hardly any noticeable difference. Each version scales the same up until threads=64. At that point, the direct version scales slightly better with a speed-up of 22.43 compared to the buffered's 20.01. The direct version performed better probably due to the lower loop overhead. Since results are written directly, only one loop is needed. Meanwhile, buffered needs two loops, the first for calculating and storing to the local copy and the second for transferring to the main copy.

Weak scaling

Buffered

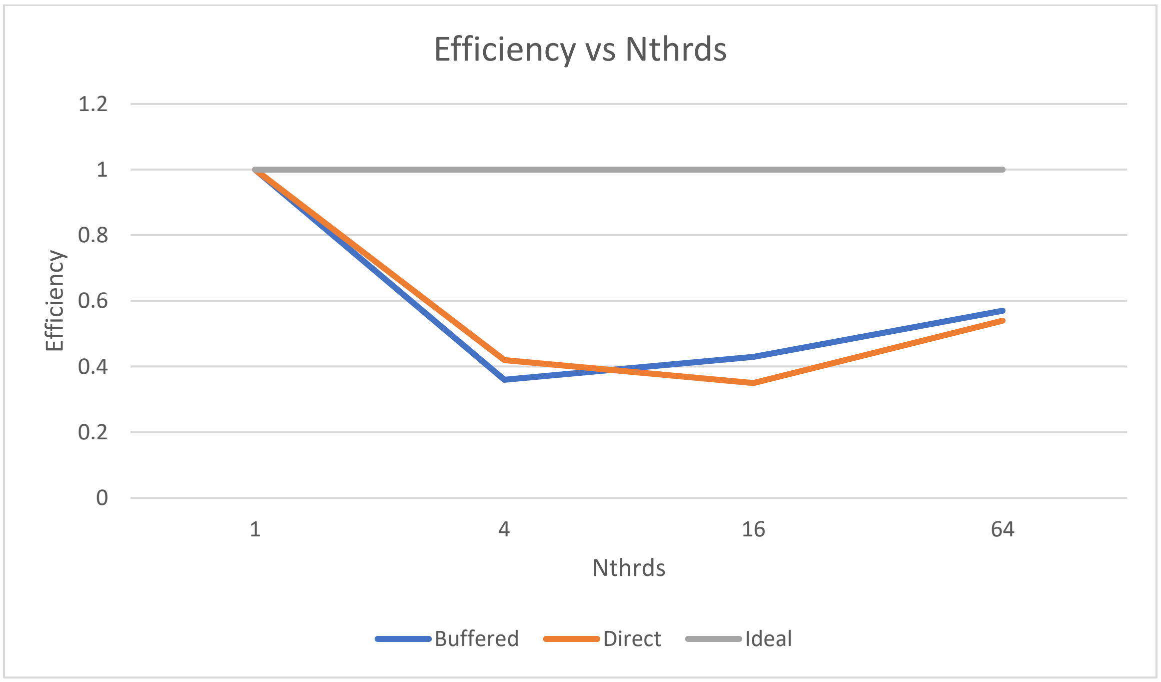
Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.12	0.36	0.83	0.98
16	0.01	0.10	0.43	0.92
64	0.0007	0.02	0.16	0.57

Nthrds	Execution time (s)	Efficiency	Problem size
1	1.380444e-04	1.00	24x24
4	3.211975e-03	0.36	96x96
16	3.455997e-02	0.43	384x384
64	4.160411e-01	0.57	1536x1536

Direct

Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.12	0.42	0.73	0.98
16	0.01	0.20	0.35	0.94
64	0.0008	0.03	0.12	0.54

Nthrds	Execution time (s)	Efficiency	Problem size
1	1.761913e-04	1.00	24x24
4	4.384041e-03	0.42	96x96
16	4.234505e-02	0.35	384x384
64	4.007030e-01	0.54	1536x1536



Weak scaling analysis

From the Efficiency vs number of threads graph, the efficiency dropped significantly as soon as the number of threads and problem size increased. The efficiency gets better at a higher number of threads, but it does not improve much.

For the buffered version, the efficiency increased at threads=16, while for the direct version, the efficiency only increased at threads=64. Overall, similar to the strong analysis graph, there is barely any noticeable difference between the two versions in this form as scaling. It can be said that the direct version performed better at lower threads and problem size while the reverse for the buffered version. This is perhaps due to loop overhead not having any significant performance penalty when the loop size is big. Hence, the performance penalty is masked by a bigger problem size.

Compared to the MPI's version of the implementation, the pthread version scales much worse overall.

Other platform performance comparison

Buffered

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.678782e+00s	3.050156e+00s
4	9.892249e-01s	8.114111e-01s

Direct

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.638580e+00s	3.293332e+00s
4	1.020467e+00s	9.132640e-01s

Both implementations ran better on my PC compared to Gadi. Since there are no "real communications" going on here and cache size is not an assumption in the algorithm, we can speculate that the performance difference might be due to faster instructions completion. My PC's CPU probably can complete more floating-point operations per second compared to Gadi's CPU hence resulting in faster execution time.

Task 4B

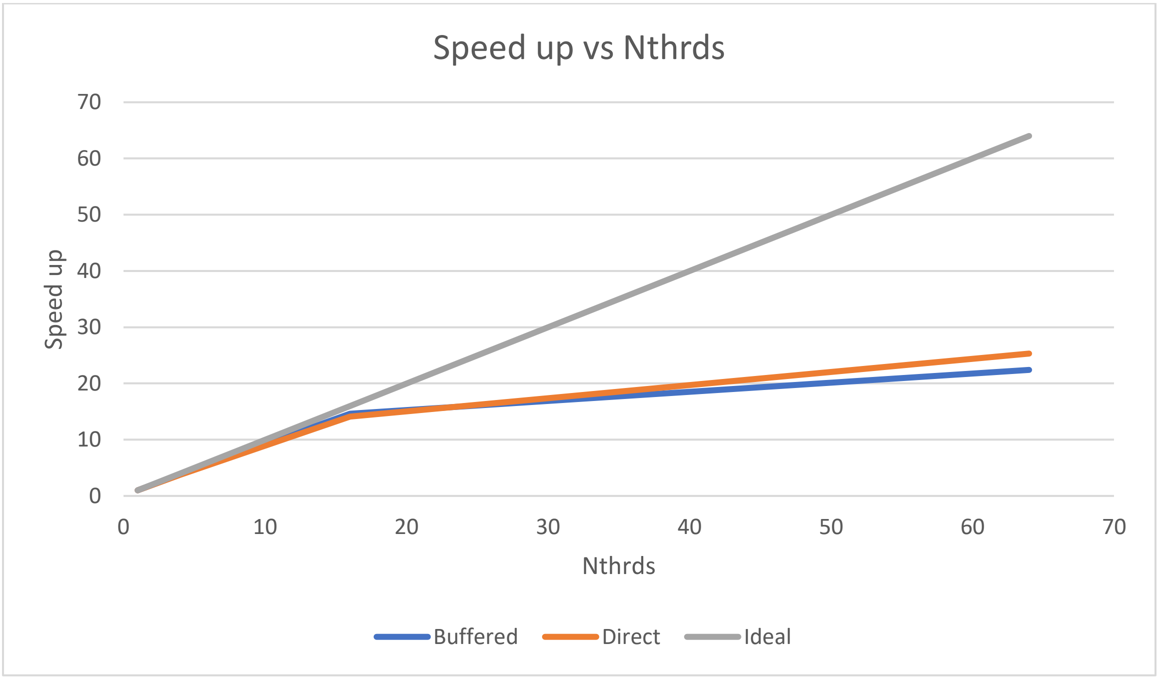
Strong scaling

Buffered

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.581706e+00	1.00	1.00
4	9.413040e-01	3.80	0.95
16	2.451880e-01	14.61	0.91
64	1.598141e-01	22.41	0.35

Direct

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	3.735652e+00	1.00	1.00
4	9.973941e-01	3.74	0.94
16	2.647130e-01	14.11	0.88
64	1.476061e-01	25.31	0.40



Strong analysis scaling

Similar to Pthread SUMMA, the two versions of the cannon's algorithm that I have implemented are buffered and direct.

As can be seen from the graph, the algorithm scales linearly until threads=16. From there, the speed-up efficiency drops by a significant amount. At threads=64, the speed-up efficiency is only on average 0.37 for both versions. This is drastically lower compared to the MPI version of the implementation where for threads=64, the average efficiency is 0.7. This difference in performance can again be attributed to the different paradigms each API used. MPI does not suffer from false sharing and communications are better because of the optimized collective communication.

Both versions scale roughly the same again in this algorithm. The direct version scales slightly better at higher threads, but the difference is not significant.

Weak scaling

Buffered

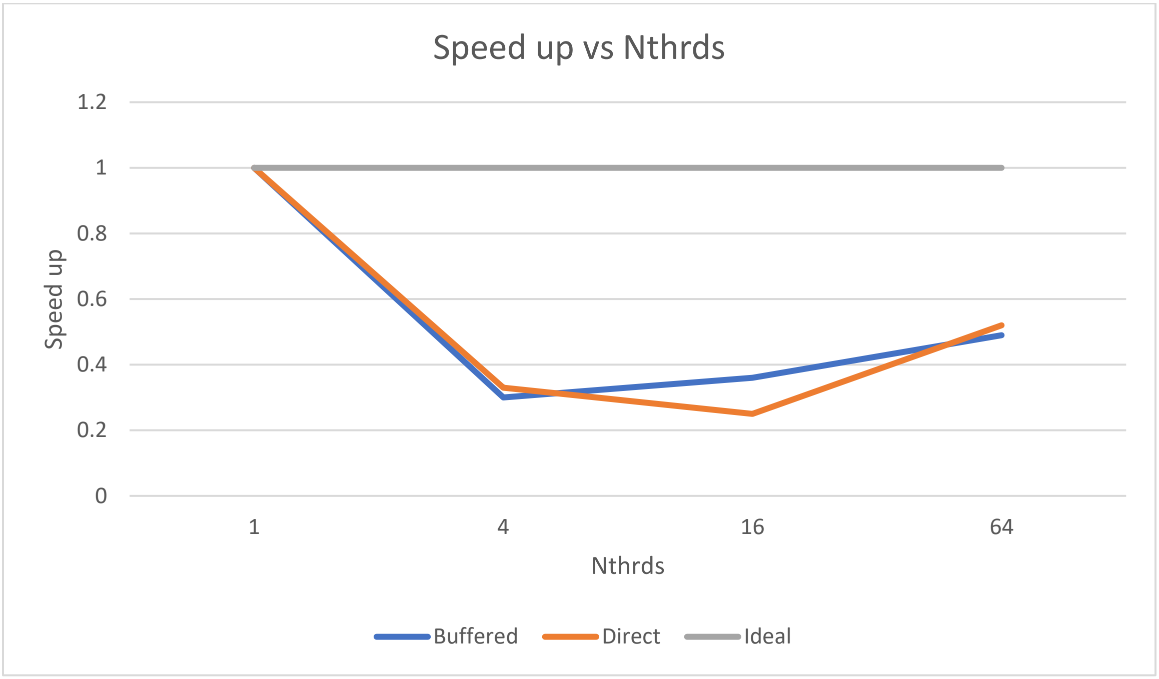
Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.10	0.30	0.72	0.99
16	0.01	0.11	0.36	0.94
64	0.0009	0.02	0.11	0.49

Nthrds	Execution time (s)	Efficiency	Problem size
1	1.208782e-04	1.00	24x24
4	2.959013e-03	0.30	96x96
16	3.516889e-02	0.36	384x384
64	4.136758e-01	0.49	1536x1536

Direct

Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.12	0.33	0.65	0.97
16	0.01	0.14	0.25	0.98
64	0.008	0.02	0.13	0.52

Nthrds	Execution time (s)	Efficiency	Problem size
1	1.568794e-04	1.00	24x24
4	3.107071e-03	0.33	96x96
16	5.182695e-02	0.25	384x384
64	4.148810e-01	0.52	1536x1536



Weak scaling analysis

For both versions, the efficiency dropped drastically when threads=4 and problem size=96x96. The efficiency dropped from 1.00 to roughly 0.3. It is worse for the direct version since increasing the threads and problem size reduces the efficiency further to 0.25. However, the efficiency does rise again when the thread and problem size is very large, threads=64 and problem size=1536x1536. Unlike the direct version, the buffered version has its efficiency rise up immediately after the initial drop.

Except for threads=16 and problem size=384x384, both versions' efficiencies are roughly the same across all threads and problem sizes. This similarity might be because of the aforementioned negligible loop overhead. Since the loop overhead is masked by a higher problem size, both versions have no other differences and performed about the same.

Comparing this version of the implementation with task 3, we can see that this implementation scales much worse overall.

Other platform performance comparison

Buffered

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.499814e+00s	3.205738e+00s
4	9.428871e-01s	9.159741e-01s

Direct

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	3.671131e+00s	3.227953e+00s
4	1.012698e+00s	9.238150e-01s

Again, both versions of the implementations ran faster on my PC compared to Gadi. Similar to the previous algorithm, since communication and cache size is not at play here, the difference is probably due to my PC's CPU being overall faster than Gadi's.

Part B

Task B description:

For task B, I have implemented an improved version of SUMMA using MPI and OpenMP hybrid. I have improved SUMMA by adding two modifications. The first is by parallelising the broadcast in the initial communication phase. This is done by having one thread broadcasting the A matrix while having the other thread broadcasting the B matrix. The second is by parallelising the local matrix multiplication done at the end of each iteration. To make it even better, I have also improved the matrix multiplication by making it a block matrix multiplication.

The motivation for this task is to utilise available hardware better. By using MPI only, certain parallelisable part of the code is left as serial. This is a waste since according to Amdahl's law, we can gain better speed-up by having more parallelisable parts in the code. To overcome this, we can use OpenMP in conjunction with MPI. The shared memory parallelism offered by OpenMP can help in parallelising the serial part without needing any synchronisation from other MPI threads. Through this approach, each node can be used for the MPI part of the code, while each node's core can be used for the OpenMP section of the code.

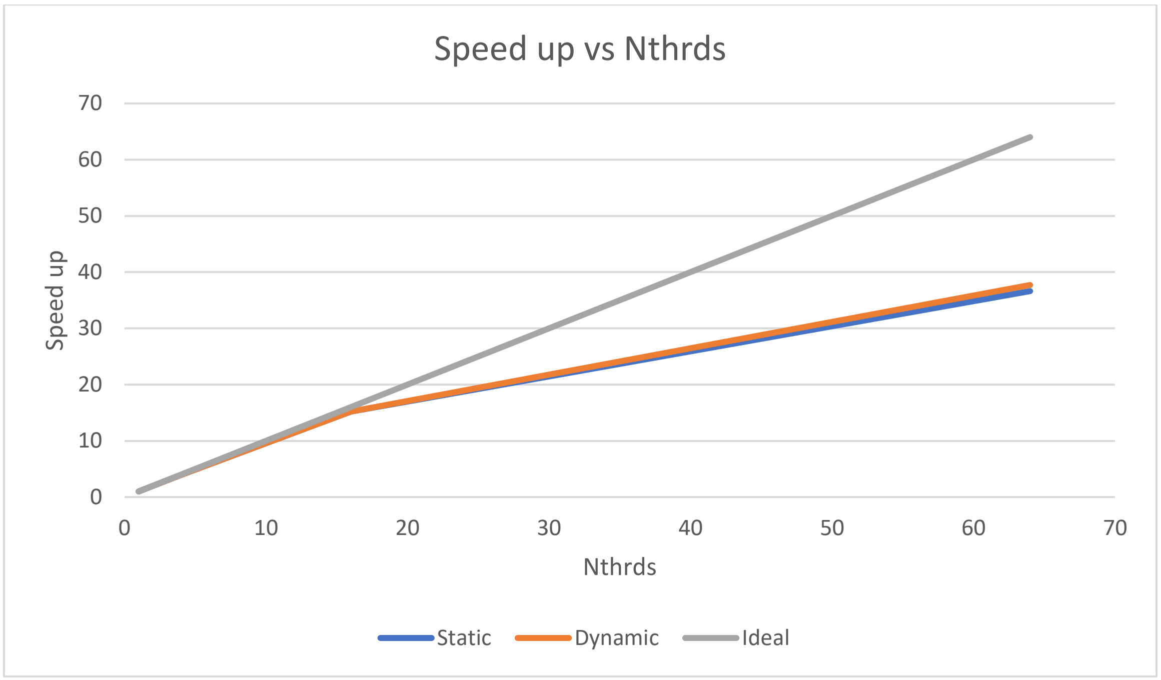
Strong scaling

Static scheduling

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	2.565677e+00	1.00	1.00
4	6.508627e-01	3.95	0.99
16	1.685320e-01	15.23	0.95
64	7.010459e-02	36.62	0.57

Dynamic scheduling

Problem size: 1000 x 1000			
Nthrds	Execution time (s)	Speed up	Efficiency
1	2.628051e+00	1.00	1.00
4	6.674232e-01	3.94	0.99
16	1.724159e-01	15.24	0.95
64	6.969602e-02	37.71	0.59



Strong scaling analysis

The two different versions I have implemented in this task are similar to Task 1. Particularly, I have implemented the block matrix multiplication part of this algorithm with either static parallel for scheduling or dynamic scheduling.

Both versions of the implementation scale linearly as ideal up until threads=16. After that, the speed-up gain is much lower compared to the ideal. For threads=64, the speed-up efficiency averages at only 0.58 for both versions. The speed-up no longer scales linearly due to Amdahl's law. When the number of threads is large, the speed-up is dependent on the parallel part of the algorithm. Considering that communication between threads is serial, this algorithm is not 100% parallelisable. Accordingly, the speed-up efficiency plateau is normal.

There is almost no performance difference between the two versions. The dynamic version performs marginally better, but the difference is too small to draw an inference or conclusion. This is expected since the block matrix multiplication is only a small part of the overall algorithm. Since the base algorithm is SUMMA, communication plays a more significant role in differentiating the performance. Therefore, because both versions used the same blocking send and receive operation, their performance is roughly the same.

Compared to SUMMA implemented with just MPI, the overall execution time is faster on the hybrid MPI/OpenMPI implementation. Thus, parallelising the remaining serial matrix multiplication section of the code does indeed provide slightly better performance. However, both versions still scale the same on the strong scaling analysis.

Weak scaling

Static scheduling

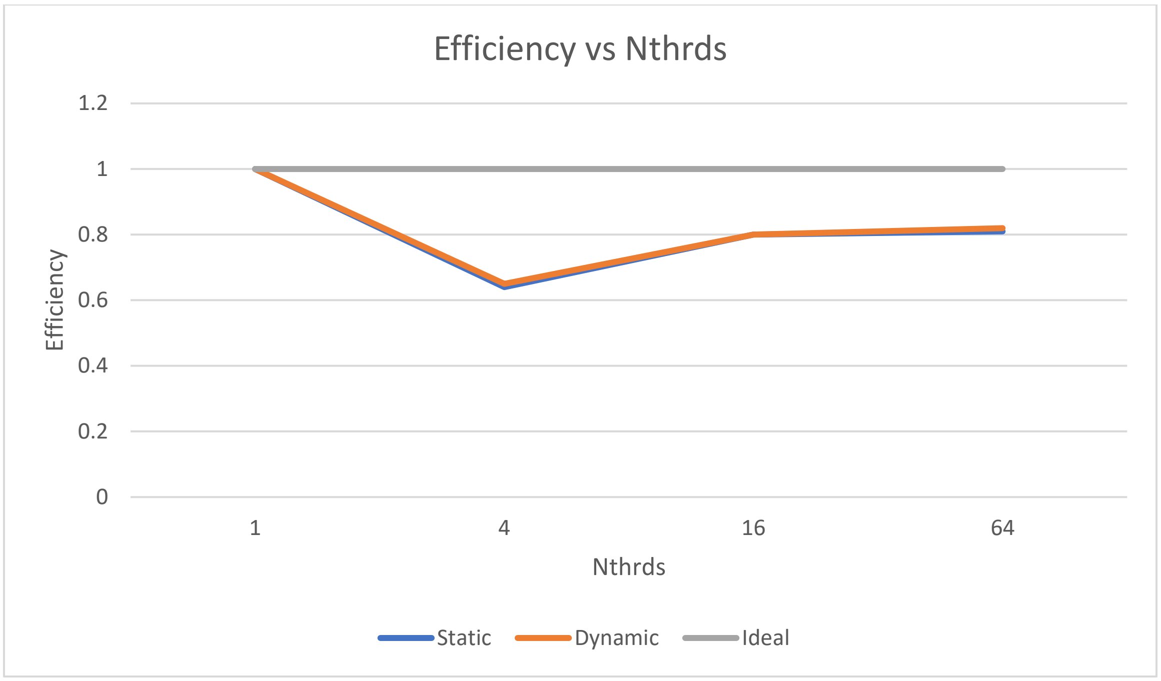
Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.10	0.64	0.97	0.92
16	0.005	0.12	0.80	0.97
64	0.0001	0.002	0.10	0.81

Nthrds	Execution time (s)	Efficiency	Problem size
1	9.863300e-05	1.00	24x24
4	9.274080e-04	0.64	96x96
16	1.155471e-02	0.80	384x384
64	1.856669e-01	0.81	1536x1536

Dynamic Scheduling

Nthrds	Matrix dimension			
	24	96	384	1536
1	1.00	1.00	1.00	1.00
4	0.08	0.65	0.97	1.00
16	0.002	0.11	0.80	0.97
64	0.0001	0.002	0.09	0.82

Nthrds	Execution time (s)	Efficiency	Problem size
1	9.375900e-05	1.00	24x24
4	9.182360e-04	0.65	96x96
16	1.170900e-02	0.80	384x384
64	1.876171e-01	0.82	1536x1536



Weak scaling analysis

The speed-up efficiency trends here are the same as in previous tasks. The efficiency dropped at threads=4 and problem size=96x96. Then, it rose back and plateaus at threads=16 and problem size=384x384. The overall weak scaling is good, the lowest efficiency is 0.6 while the plateau is at 0.8.

Both versions have no difference in performance. The speed-up efficiency is equal throughout all number threads and problem sizes. The explanation for this is similar to the strong scaling analysis; the block matrix multiplication part of the code is too small to provide a performance difference. Therefore, no matter what schedule the loop has, the performance benefit is hard to notice.

Block size comparison

Number of threads:16, Problem size: 800x800

Static scheduling

Block size	Execution time (s)	Speed up
1	2.036419e-01	1.00
2	2.044299e-01	0.99
4	1.290468e-01	1.60
8	1.273029e-01	1.60

Dynamic scheduling

Block size	Execution time (s)	Speed up
1	1.931142e-01	1.00
2	1.973471e-01	0.98
4	1.450838e-01	1.33
8	1.097518e-01	1.76

Similar to Task1's block matrix multiplication, increasing block size offers some speed-up but only to a certain extent. The logic here is similar. If the block size is too small, then the matrix elements are too large to be stored in the cache. On the other hand, if it's too big, then the temporal locality of the cache cannot be fully taken advantage of. It is important to note that block size speed-up here is smaller compared to block matrix multiplication. This is because of the algorithm itself. In this implementation, block matrix multiplication is just a small part of the main SUMMA algorithm, therefore it can just be considered micro-optimization. In task 1, the main algorithm is the block matrix multiplication, hence optimizing the block size offers a more significant speed-up.

Other platform performance comparison

Static

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	2.729112e+00s	1.955130e+00s
4	7.298713e-01s	6.301974e-01s

Dynamic

Problem size: 1000x1000	Execution time (s)	
Nthrds	Gadi	PC
1	2.556561e+00s	1.927688e+00s
4	6.831977e-01s	6.030997e-01s

Both versions of the implementation ran faster on my PC compared to Gadi. Similar to Task 2 and Task 3, this might be because of MPI communication running better on AMD Ryzen's core topology. The advantage of having a bigger cache for Gadi's Xeon CPU does not provide any advantage here. This is because the block matrix multiplication part of this code is small. As a result, it does not provide a drastic difference as seen earlier in Task 1. To conclude, the advantage of having a bigger cache is bottlenecked by having poorer inter-core communication performance.