4300 Individual Project Report

Qianhui Wang, u7070170

May 29, 2022

Part A

Problem Description

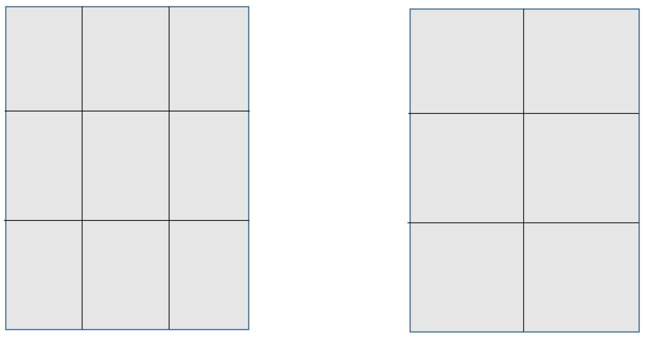
- **Problem inputs**: matrix $\mathbf{A}$ of dimensions $M \times K$ and matrix $\mathbf{B}$ of dimensions $K \times N$
- **Problem output**: matrix $\mathbf{C}$ of dimensions $M \times N$ such that $\mathbf{C} = \mathbf{A} \times \mathbf{B}$
- **Algorithms**: 1. blocked matrix multiplcation; 2. SUMMA; 3. Cannon.
  These algorithms arrange threads/processes into a 2D layout (referred to as "grid" in this report), and operate on submatrices of $\mathbf{A}$ and $\mathbf{B}$. There are two essential preprocessing steps pertaining submatrices that might affect performance:

  1. Partition original matrices into submatrices of certain sizes(dimensions).
     The sizes of submatrices of $\mathbf{A}$ and $\mathbf{B}$ should be conformable such that submatrix multiplication is well-defined. One example of a conformable partition of $\mathbf{A}$ and $\mathbf{B}$ is shown below. $\mathbf{A}$ and $\mathbf{B}$ should have the same number of submatrices along the K dimension (3 in this case). The submatrix $A_{i,k}$ and $B_{k,j}$ should have conformable dimensions $m \times l$ and $l \times n$ for some $m, l, n > 0$.

$$\mathbf{A} = \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} B_{1,1} \\ B_{2,1} \\ B_{3,1} \end{pmatrix}$$

     The size of submatrices can vary. However to quantify the distribution of workload, equally sized submatrices are used, so that computation for one submatrix of $\mathbf{C}$ is counted as a unit of workload. The only submatrices that might have different sizes are those at the boundary of the original matrices.

     In addition, the shape of submatrices can be square or rectangle, depending on the assumed thread/process grid. If threads/processes are arranged in a square grid, then the submatrix shape will be the same as that of the original matrix $\mathbf{C}$. If a rectangle process grid is used, we could make the submatrix a square by following the $\mathbf{C}$ matrix dimension ratio $M : N$ for the grid *row* : *column* ratio. In (b), because $M : N = 3 : 2$, the grid has 3 rows and 2 columns (actual number of rows and cols depends on *nthds*/*nprocs*).



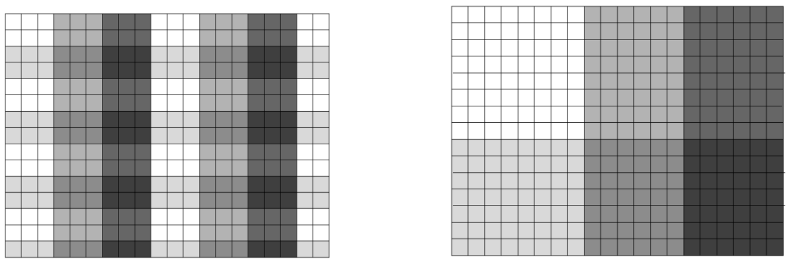
(a) square grid - rectangle submatrix

(b) rectangle grid - square submatrix

Figure 1: submatrix shape

2. Map threads/processes to submatrices, (or distribute submatrices to threads/processes)
   Mapping can be done either statically or dynamically. Static mapping is much easier to implement since there is no need to consider the run time. Hence all implementations implement static mapping in this project, (other than OpenMP where the runtime library can handle dynamic scheduling for the programmers).

   Under static mapping, two major techniques are block distribution and cyclic distribution. Distribution is closely related to the size of submatrices. If the submatrix size is small with respect to the number of available parallel resources (threads/processes), then each thread/process will be assigned multiple units of submatrix. On the other hand, we can maximize the submatrix size such that each thread/process takes at most one or two units. Two distributions of 15x20 matrix on a 2x3 thread/process grid are illustrated below.



(a) multiple units (cyclic) (b) single unit (block)

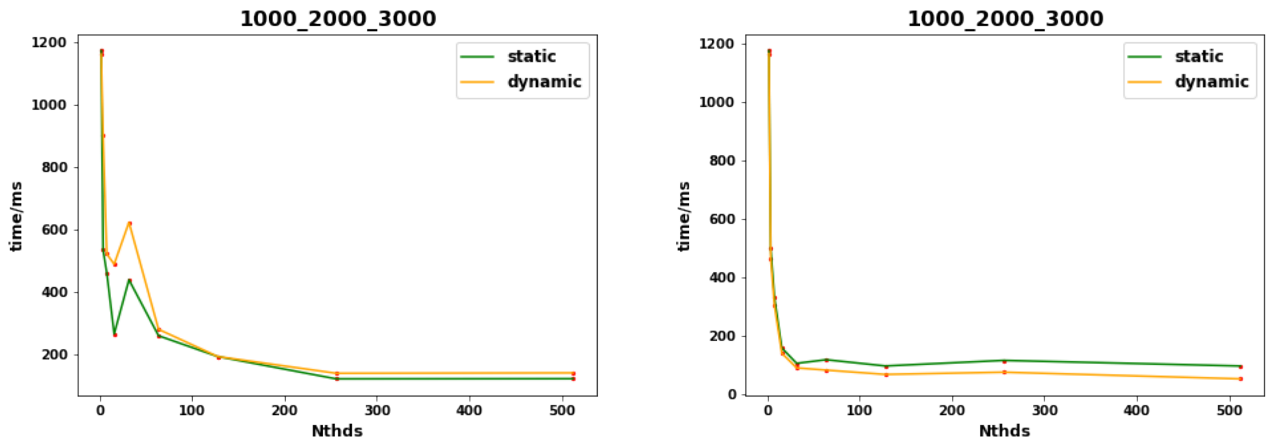
Figure 2: submatrix distribution

   The level of difficulty in implementing cyclic distribution varies with different programming paradigms (and APIs). For OpenMP, we can easily implement cyclic distribution of any-sized submatrices by asking the OMP runtime to implicitly do the work for us, by using "omp for directive" and the "schedule" construct. Distribution is also simple because we operate with shared memory, which the concurrent entities can directly access.

   However, cyclic distribution with MPI is much more difficult, because data needs to be explicitly sent across processes of distinct address space. This requires much data and memory handling if the submatrix size is small (e.g. copying multiple submatrices to contiguous memory for send operations), which will have poor performance consequences. This problem can be alleviated by maximizing the submatrix size. Hence, in Task2-4 implementation of SUMMA and Cannon algorithms, a simplified cyclic distribution is used such that each process takes at most 2 units of work.

Task 1

Implementation design Variants of OpenMP matmul implementations assume a rectangle thread grid, such that all submatrices are of square shape (Figure 1b). In `omp_block.c` and `omp_block_2.c`, the submatrix size is optimized based on the dimensions of the thread grid. The submatrix multiplcation uses a k-i-j loop (the most performant serial matmul structure) to optimize the individual thread work. `omp_block.c` implements a "write-through" scheme: threads update the intermediate multiplcation results directly into the memory region of matrix **C**. On the contrary, `omp_block_2.c` implements a "write-back" scheme: threads allocate private memory to "cache" their intermediate results, and update the final product sums into **C** at the end. Each file compares the two submatrix distribution schemes: static block distribution by the default `schedule(static, n/nthds)`, and dynamic distribution by `schedule(dynamic, 1)`.



(a) write-through in `omp_block.c` (b) write-back in `omp_block_2.c`

Figure 3: performance benchmark of omp implementations

Performance Figure 3 shows the performance of the variants on matrices of dimensions 1000×2000 and 2000×3000. Implementations using the "write-through" scheme (`omp_block.c`) takes much more time (5-6 times slower) than the "write-back" ones, especially for $nthds < 200$. This is expected since updating intermediate results directly to the contiguous memory of matrix **C** will result in false sharing. Although threads update in distinct memory locations (hence there is no real sharing), they could update the same cache line that these distinct yet contiguous memory locations map to. When threads belonging to different cores update in the same cache line, each update will invalidate the shared cache line of different cores, and will force the threads to access the main memory due to write-misses. In particular, there are 2000 additions for the calculation of a single **C** value, which could mean a maximum of 2000 write-misses just for a single value, and taking into account the size of submatrices and its relation with the cacheline size, write-misses for a single thread will be a lot more. Hence, "write-through" scheme is not desirable for a parallel implementation. In contrast, `omp_block_2.c` uses private thread memory to alleviate false sharing. The private memory will be highly unlikely to be mapped to the same cache line, so threads will not experience read-write misses during their entire submatrix multiplcation process. The only possible write-misses occur when they copy the final results to the memory of **C**, which is much less than the "write-through" scheme. Hence the "write-back" performance will be close to optimal.

In addition, we could observe in Figure 3 that the false sharing effect with "write-through" is most severe with $nthds = 32$ where there is a sudden spike in running time. The poor performance might be due to the specific choice of block size $blk\_sz = 250$ resulting in a data distribution that has most number of threads mapped to a single cache line. On the other hand, because we do not have much false sharing with "write-back", the running time is a smooth decrease with increasing number of threads.

We calculate the speed ups and efficiencies of the "write-back" implementation for the problem size 1000_2000_3000.

nthds	time/ms	speedup	efficiency
1	1601.369	nil	nil
2	1165.810	1.37	0.69
4	463.788	3.46	0.86
8	304.916	5.27	0.65
16	140.875	11.44	0.71
32	90.698	17.78	0.56
64	83.100	19.29	0.30
128	68.353	23.54	0.18

Table 1: speedup & efficiency of "write-back"

The "write-back" implementation is strongly scalable, especially for $nthds \leq 32$, since the efficiency is almost constant by increasing the number of threads with constant problem size. For $nthds > 32$, the overhead of thread creation and termination exceeds the amount of useful work in the parallel region, hence we observe the decrease in efficiency.

We can also study the effect of matrix dimensions on running time. We compare performance with 6 arrangements of matrix dimensions and we keep the total amount of work constant (1000x2000x3000 multiplcations in total). The specific choice of $nthds$ is 32, since this delivers the worst performance with "write-through" scheme. (Table 1)

dimensions (M_K_N)	time/ms (static)	time/ms (dynamic)
2k_1k_3k	252.522	293.350
3k_1k_2k	179.774	216.251
1k_2k_3k	265.885	328.770
3k_2k_1k	372.362	428.534
1k_3k_2k	394.120	572.992
2k_3k_1k	618.864	548.039

Table 2: write-through in `omp_block.c`

The running time roughly increases as the dimension K increases, and this is consistent with what is explained earlier that for a single $\mathbf{C}_{ij}$ calculation, there are K intermediate updates of the product sum. Hence the larger the K value, the more false sharing there is.

For "write-back" scheme (Table 2), the running time does not vary much with matrix dimensions since no false sharing occurs. Since the total work for the 6 dimension arrangements is the same, individual threads will roughly have the same amount of work. Hence, the observation is consistent with the expected performance. The fluctuation in running time can be attributed partly to the runtime difference, and partly to the difference in submatrix size used (chosen based on the ratio $M : N$ of matrix $\mathbf{C}$).

dimensions (M_K_N)	time/ms (static)	time/ms (dynamic)
2k_1k_3k	119.367	98.118
3k_1k_2k	111.117	95.598
1k_2k_3k	105.997	88.848
3k_2k_1k	105.424	98.695
1k_3k_2k	85.430	84.763
2k_3k_1k	87.373	69.801

Table 3: write-back in `omp_block_2.c`

In addition, we notice that the dynamic distribution with "write-back" is more performant than static distribution, whereas with "write-through", it is less performant. It is suspected that the difference is still related to false sharing. Since "write-back" has almost no false sharing, threads can quickly finish a single unit of work and take a next unit when they are ready, there will be little overhead for the runtime scheduling. On the other hand, with false sharing, threads are required to access memory, and because the memory access requires invalidate and refetch the cacheline, the OS scheduler might block the threads until the cache is ready. Hence, the OpenMP runtime will need to wait for threads to be waken and ready, which results in the worsened performance.

In sum, we should be cautious to avoid false sharing when designing a parallel program with shared memory systems.

Task 2

Implementation design As mentioned in the Problem Description, for implementations with MPI, we maximize the submatrix size w.r.t. the number of available processes, to reduce the overhead of data copying in message passing operations. To determine the submatrix size, we need to first arrange processes into a 2D grid. As mentioned before, we can either use a square grid or a rectangular one. For SUMMA, both ways are valid. In particular, for a rectangle grid of dimension $m \times n$ built from the available processes (i.e. $m \times n \leq comm\_sz$), a conformable partition of matrices that also preserves the correctness of SUMMA could be as follows:

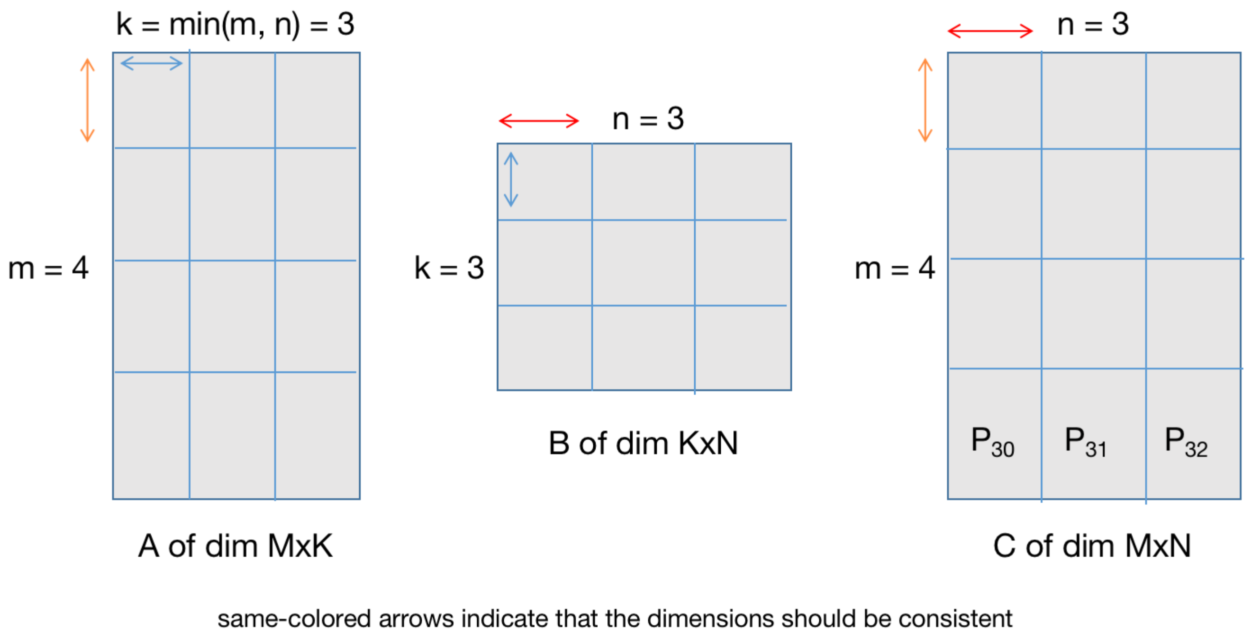


Figure 4: submatrix partition for rectangle process grid

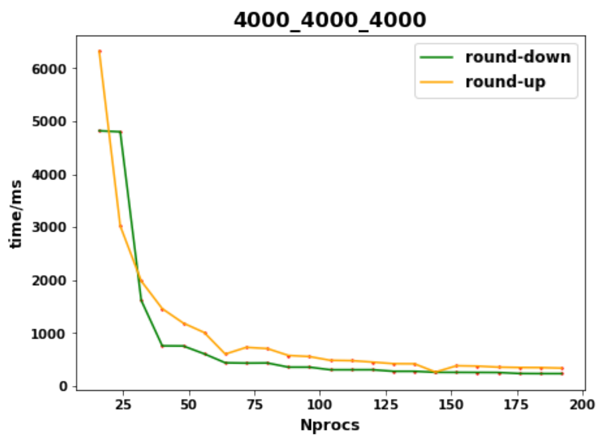
In this example, we assume that $comm\_sz \geq m \times n = 4 \times 3 = 12$. In this mapping scheme, the processes P_{3x} will not keep data copies of any submatrix of $\mathbf{B}$. This mapping still works for SUMMA specifically because it uses the broadcast operation for data exchange, and hence all necessary data copies will be available for these processes during the multiplcation loop. For a conformable partition of $\mathbf{A}$ and $\mathbf{B}$, with dimensions $m \times k$ and $k \times n$, the specific value of k is chosen to be $\min(m, n)$ because any value greater will result in some processes having to keep data copies of more than one unit of submatrices.

On the other hand, a more straightforward design uses a square process grid, such that each process has the data copy of a single submatrix of $\mathbf{A}$ and $\mathbf{B}$. In this project, I use the square grid design because the grid dimensions m, k, n will then be the same, and this requires less bookkeeping than the rectangle grid design (3 different values for m, k, n). It turns out that even with the square grid, the actual implementation of SUMMA that needs to be workable for any input matrix size is already quite complex, so I would keep the grid design simple.

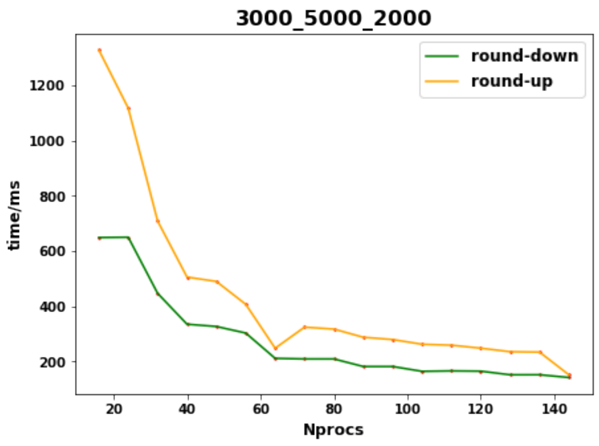
The complexity lies with the number of different submatrix dimensions that we need to keep track of. For the square grid (Figure 1), submatrices will have rectangle shape, hence different row and column dimensions. Also, those submatrices at the edge of the original matrix could have different dimensions, so there will be 4 different submatrix shapes for a single matrix. For **A**, **B**, and **C**, there will be 12 in total. MPI provides construct to create self-defined datatype, for SUMMA (and Cannon), we will build 12 "subarray" types. This is already not an easy task, so I do not use rectangle grid in my designs.

Rather, what I explored follows another direction: how to fully utilize the given parallel resources. For a given number of processes *comm_sz*, we can either build the grid by using the largest square number $\leq$ *comm_sz* ("round-down"), or the smallest square number $>$ *comm_sz* ("round-up"). For the "round-down" method, a small number of processes will never be utilized, so it could be possible that since this approach wastes some parallel resources, it will be less performant than the "round-up" which fully utilizes the available resources. For the "round-up" method, some processes will need to be mapped to 2 units of submatrices, hence it implements a simplified cyclic distribution. To explore the performance difference, both implementations are provided: "round-down" in `mpi_summa.c` and "round-up" in `mpi_summa_2.c`.

Performance We take timings from before P0 (default process for file IO) distributes submatrices to after P0 collects all subresults.



(a) 4k_4k_4k



(b) 3k_5k_2k

Figure 5: benchmark of SUMMA

Clearly, the performance of "round-down" is better than "round-up". The essential cause is with the cyclic mapping i.e. some processes are assigned two units of **A** and **B**, and need to calculate two units of **C**. In my implementations, the particular choice of function for submatrix distribution and collection is `MPI_Alltoallw`, which is one of the MPI functions that support the most generic data communication. This function is a collective function that allows simultaneous sending and receiving of data of any type, from all processes, to all processes inside a communicator. The essential reason for its use is that, we need to deal with at most 4 different "subarray" types with a matrix distribution and it is the only collective function (I found) that supports the send/recv of multiple data types in a single operation, otherwise we are forced to use point-to-point sendrecv which is less efficient. However unfortunately, due to its generality, there will be little optimisation for communication that the MPI runtime can do. To make the matter worse, cyclic mapping requires twice usage of `MPI_Alltoallw`, so submatrix distribution and collection in ”round-up” definitely take more time. In addition, the parallel region will also be slower. First, processes taking 2 units have to complete one more unit of local matrix multiplcation than the rest, and since SUMMA uses the collective broadcast operation to exchange data, faster processes need to wait for those with 2 units of work to complete before broadcast can proceed. Second, processes taking 2 units also need to participate in broadcast twice since they need copies of 2 different submatrices. Hence, the broadcast process will take more time. In essence, the bottleneck is with the operations of processes with 2 units of work.

For the inconsistencies in Figure 5, where ”round-up” has closer to ”round-down” performance for *nproc* = 64 and 144, it is simply because the number used is a square such that no cyclic mapping takes place. However, we still observe that ”round-up” performs worse, since the implementation has to take care of the generic cyclic mapping to ensure correctness and this incurs too much overhead. We also observe the inconsistency on testcase 4k_4k_4k that the performance of ”round-down” is worse than ”round-up” when *nprocs* = 24. This is a special case where *nprocs* is a number closest to a larger square value(25). ”round-up” will be more performant since only one process will need to take 2 units of work, so the overhead of cyclic mapping is minimal; and ”round-down” will waste $24 - 16 = 8$ which is 33% of given parallel resources, and is thus not efficient. This is the special case that demonstrates the importance of utilizing all available parallel resources, but we should note that cyclic mapping only worths it when the problem size is sufficiently large. For example, even with *nprocs* = 24, on a smaller testcase 3k_5k_2k, ”round-up” is still much worse than ”round-down”. Hence, the best situation to use ”round-up” is with a large problem and a small value of *nprocs* (also closest to a larger square).

We calculate the speedup and efficiency of ”round-down” approach (since this is more performant).

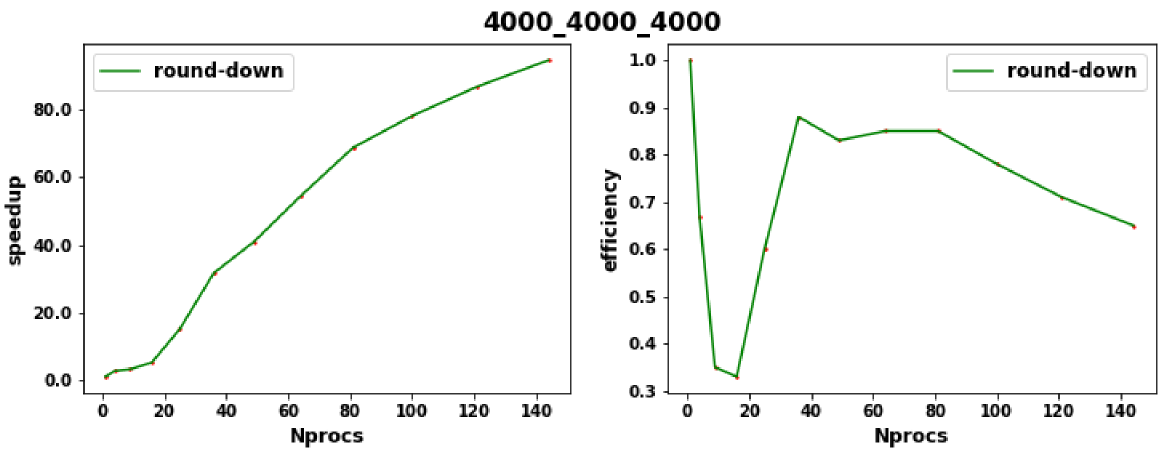


Figure 6: speedup and efficiency of ”round-down”

SUMMA can be considered strongly scalable, especially for *nprocs* within the range of 25 to 81. It is quite interesting to see that for small values of *nprocs*, speed up and efficiency is a lot worse than when *nprocs* is large. My suspection is that since our performance measurement takes into account the overhead of initial data distribution and final collection, the time for such data communication will be more pronounced for a small process grid (larger submatrix size), hence the efficiency will be lower.

One point to mention about correctness is that, cyclic mapping could result in deadlock for broadcasts within column communicators. In Figure 6, the mapping of P0, P1, P2 and P3 creates a cyclic dependency of the blocking column broadcasts, which will cause a overall system deadlock. Hence, care should be taken to use the non-blocking broadcast `MPI_Ibcast` with `MPI_Wait` for the column communicators. This might not have too much performance issue, but it did complicate the implementation.

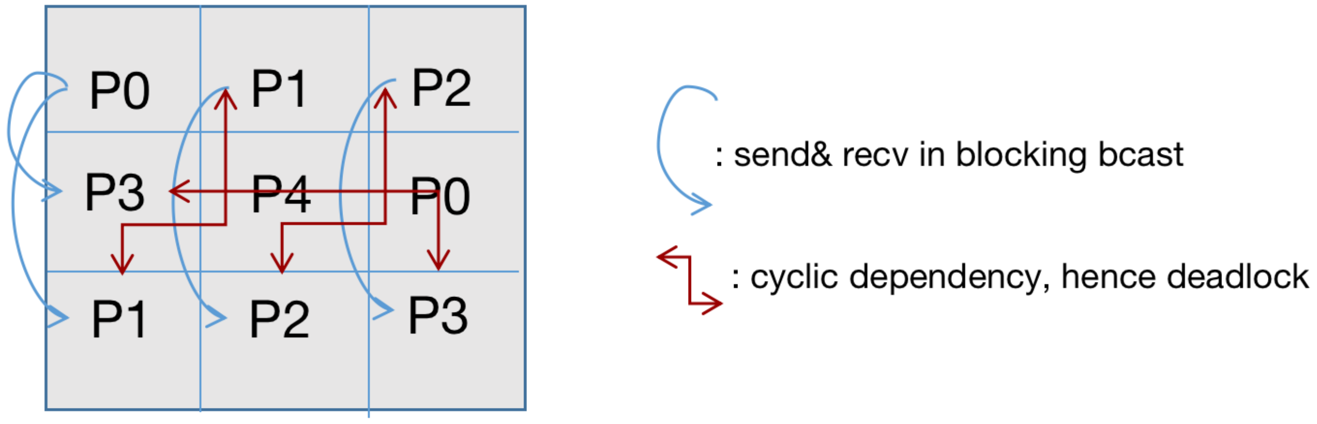

P0
P1
P2
P3
P4
P0
P1
P2
P3
: send& recv in blocking bcast
: cyclic dependency, hence deadlock

Figure 7: deadlock with column broadcasts

We also compare performance for different matrix dimensions (Figure 8). It is expected that there should not be too much performance variance when we keep the total amount of work and the *nprocs* constant. This is because we assume a fixed square process grid, and the work is evenly distributed to all processes. There is no sharing of data, so work is completed individually. So it should be the case that time taken will not vary too much. However, we still observe a slight (if not negligible) increasing trend in Figure 8, and it seems to be the case that the dimension M is the decisive factor that causes this increase, and that the effect is most pronounced with smaller *nprocs*. Hence I suspect the reason lies with overhead of data communcation, but it is less apparent to me how a single dimension would affect the overall performance, so it might also be the case that the data collected is not representative of the overall behaviour. Hence, more study can be done with greater problem size and more extreme variation in these dimensions.

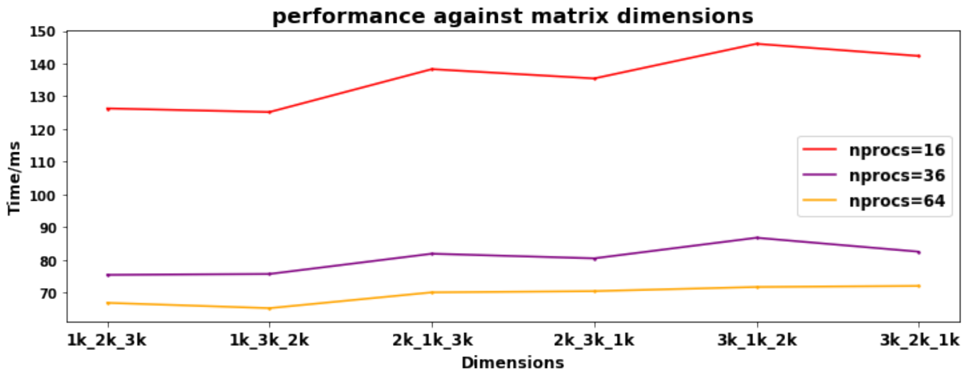


Figure 8: different matrix dimensions with "round-down"

Suggestions In sum, the "round-down" scheme that disregards some available processes will perform even better than the "round-up" scheme that forcefully utilizes all resources with cyclic mapping. But the disclaimer is that the actual performance varies heavily with the specific MPI functions and algorithmic approaches used, so it might be possible to implement SUMMA in others way that can utilize more resources while improving the performance, even with cyclic mapping. Some alternative approaches to look into could be 1. use `MPI_Scatterv` and `MPI_Gatterv` for `P0` to distribute data and collect results. Since these functions are more restrictive in the number of data type supported and they only allow a single sender/receiver, they usually are more efficient; 2. allocate sufficient memory to reduce data copy, "round-down" currently allocates memory only enough to keep a single submatrix, yet during the SUMMA broadcast loop, the original data cannot be overwritten, so a temporary array is used for broadcast, and there is overhead with copying data in-out. Hence, one way to reduce overhead is to allocate the full matrix size for every process. 3. use native MPI functions like `MPI_Cart_create` to explore performance of different rectangle layout. It is possible that by varying the rectangle grid size, one can improve the resource utilisation from "round-down" and at the same time have a performance gain. It might seem weird at first why my second implementation does not implement this way given that I have already thought about it. The primary reson is still that I wish to explore whether it is worthy to "forcefully" utilize all parallel resources given, and the empirical results suggest that there is no worth in doing so, especially with cyclic mapping. I assume this is the same reason why MPI provides functions that only supports building process grid out of the available processes but throw errors once the grid size exceeds $comm\_sz$.

Task 3

Implementation design Similar to SUMMA, a "round-down" and a "round-up" method is implemented for Cannon. Process layout and submatrix partition follows the same strategy. The only difference for the two algorithms is basically in the matrix multiplcation algorithms themselves (i.e. broadcast for SUMMA and sendrecv for Cannon). One step I took further is to implement the data distribution and collection with `MPI_Scatterv` and `MPI_Gatherv` (Suggestion 1 of Task 2) for the "round-down" program. Here the distribution and collection occurs in 2 phases (). For scatter, phase 1 takes advantage of the row-major memory layout of C lanaguage, to distribute full rows of submatrices to processes that are in the same column with P0 (col 0). These processes then distribute the corresponding submatrices to their row members in phase 2. Note that for Cannon, the intial shift of **A** can be integrated in phase 2 so as to save time for a separate shift operation. However, the initial shift of **B** cannot be integrated because the shift is done within columns, and we cannot take advantage of the C language memory arrangement to do so. Hence a separate shift for **B** is still required. The 2-phase gather operates in the opposite order of the 2-phase scatter, where in phase 1, processes in col 0 collects submatrices to form the full rows and in phase 2, P0 collects the full rows to form the complete **C**.

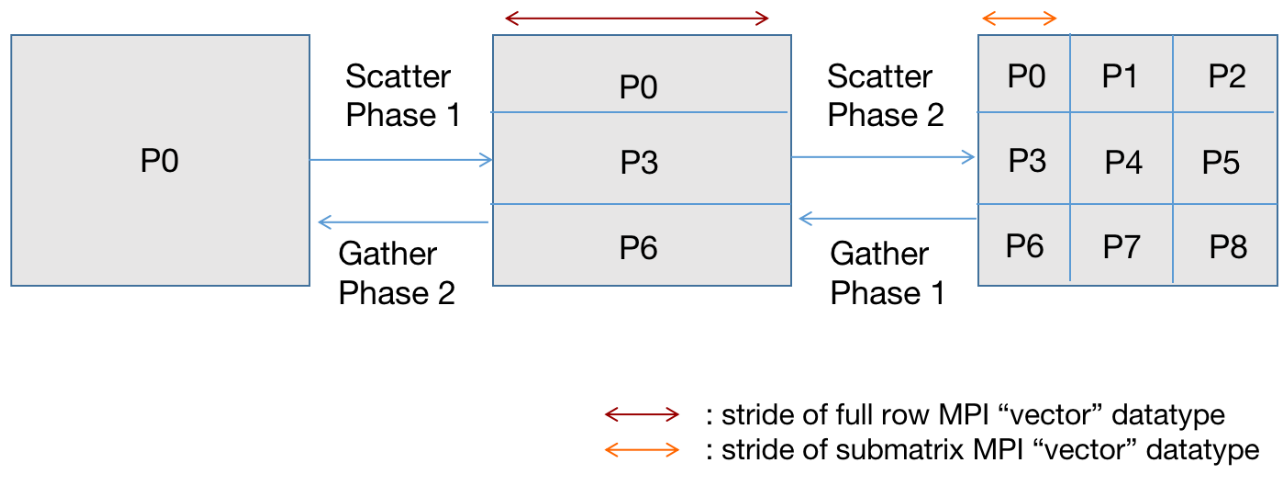
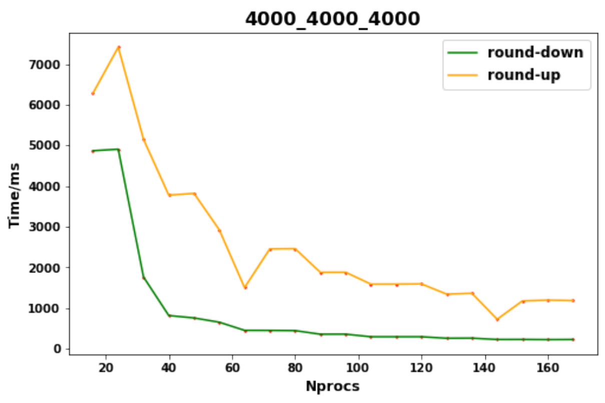

P0
Scatter Phase 1
Gather Phase 2
P0
P3
P6
Scatter Phase 2
Gather Phase 1
P0
P1
P2
P3
P4
P5
P6
P7
P8
: stride of full row MPI "vector" datatype
: stride of submatrix MPI "vector" datatype

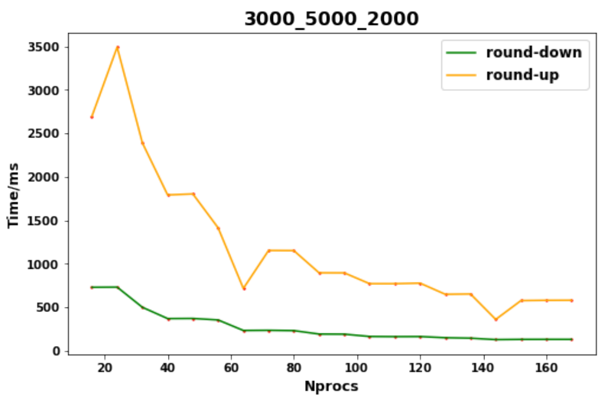
Figure 9: 2-phase scatter-gather

Also as mentioned in Task 2, a concern related to cyclic mapping is the occurrence of deadlock with blocking column operations. Since cyclic shifts also involve participation of all members in a column communicator, just like the blocking broadcast calls, if the shifts are implemented in blocking modes, a cyclic dependency can exist amongst the blocked column operations, and thus results in deadlock. To prevent this, similarly we use non-blocking operations (e.g. `MPI_Isend` and `MPI_Wait`) to achieve the same purpose.

Performance Like in SUMMA, the "round-up" approach is less performant than the "round-down" one (Figure 9), because cyclic mapping incurs much more overhead in data distribution and collection, and overhead in the parallel Cannon loop that requires synchronisation at each step. Specifically for "round-up", we notice that there exists sudden dips in running time for certain values of *nprocs*, and then the running time rises again. These dips occur at $nprocs = 16, 64, 144$, which are all perfect square numbers that do not require cyclic mapping, hence this runtime improvement is consistent with our expectations. However, we would have expected the performance with perfectly squared *nprocs* for "round-up" to be similar to "round-down", since there is no cyclic mapping and the algorithms used are essentially the same; yet from the results, it seems that "round-down" is still much better. There are two possible reasons for this: 1. as with SUMMA, there is implementation overhead to support the generic version of cyclic mapping in "round-up". 2. different choices of communication functions for data distribution and collection. As mentioned, "round-down" uses 2-phase scatter and gather, whereas "round-up" uses alltoall (easier for cyclic mapping). Both use the same (or similarly performant) sendrecv operations for shifts in Cannon algorithm. Hence, it must be the case that MPI does more optimisation for the 2-phase scatter and gather than for alltoall. This suggests the potential of improving SUMMA's performance with 2-phase scatter-gather as well. And overall, there is little meaning in using cyclic mapping for Cannon.



(a) 4k_4k_4k



(b) 3k_5k_2k

Figure 10: benchmark of Cannon

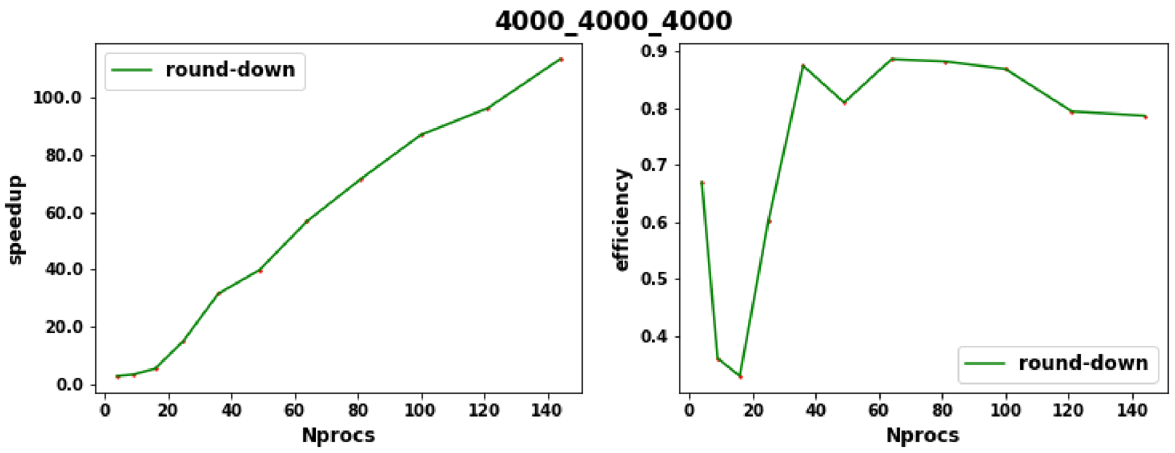
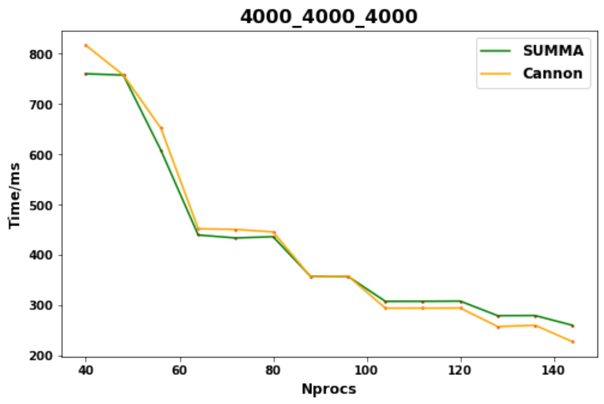


Figure 11: speedup and efficiency of "round-down"

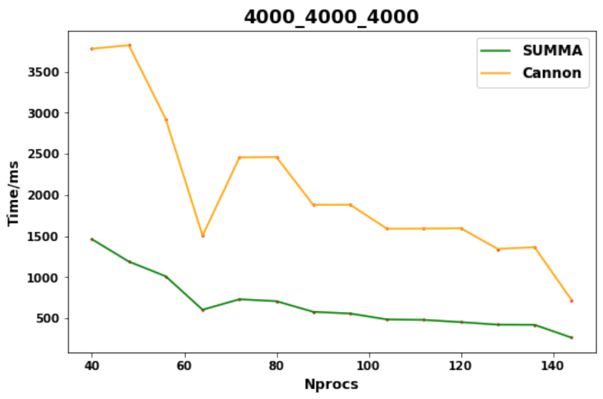
We measure the speedup and efficiency for "round-down". The efficiency is relatively constant between *nprocs* = 36 to 100, with a slight drop at *nprocs* = 49 that could be caused by the underlying node topology used in gadi. As with SUMMA, the low efficiency at *nprocs* = 4 and *nprocs* = 9 may be caused by the fact that the process grid is small, hence each process has to communicate more data, therefore the communication overhead overrides the actual time spent on doing useful matmul work. Overall, Cannon can be considered as having strong scalability.

We can also compare the performance difference between SUMMA and Cannon implementations. From our theoretical complexity analysis of these algorithms (in the Lecture on Matrix), we expect Cannon to be more performant than SUMMA since it only requires point-to-point communication, and does not have the $O(p)$ broadcast overhead (p is the number of processes in a row/column communicator). However, our empirical results show that SUMMA is in fact faster than Cannon. For "round-up", both SUMMA and Cannon use `MPI_Alltoallw` for data distribution and collection, so the cause of such a huge performance difference could only be in the algorithms themselves. We know that the essential difference between SUMMA and Cannon is that SUMMA uses broadcast for data exchange, whereas Cannon uses shifts. Hence, in practice, the collective `MPI_Bcast` is used for SUMMA, and the point-to-point `MPI_Sendrecv_replace` is used for

Cannon. The empirical results suggest `MPI_Bcast` is faster than `MPI_Sendrecv_replace`. According to the MPI implementation, `MPI_Sendrecv_replace` uses a separate buffer to store the data to be sent, then uses non-blocking sends and receives for the rest of operations. If this is true, then there will not be performance issue related to the inherent cyclic structure of the shift operations. As such, it is surprising to learn that the running time of such point-to-point operation is worse than a collective one. Perhaps the reason is that the underlying node topology is not optimised such that to send to a destination node, the message has to travel through a lot of intermediate nodes; whereas for broadcast, MPI automatically arranges in a tree-structure for best optimisation. On the other hand, for "round-down" implementations, we observe that the performance of Cannon is much similar to SUMMA, and even outperforms SUMMA for high values of *nprocs*, I believe this performance similarity is a result of Cannon using the better 2-phase scatter-gatter operations for data distribution and collection operations. Otherwise, if we use the same scatter-gather for SUMMA "round-down", the performance of SUMMA will still be better as it uses the practically more efficient broadcast operation.



(a) "round-down"



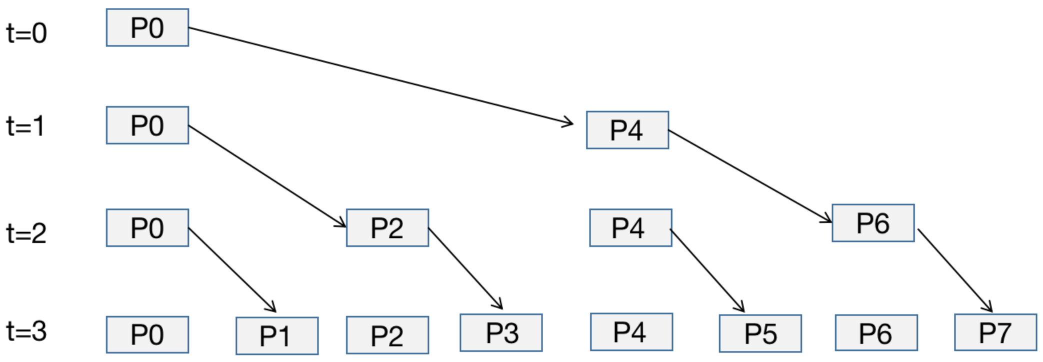
(b) "round-up"

Figure 12: SUMMA vs Cannon

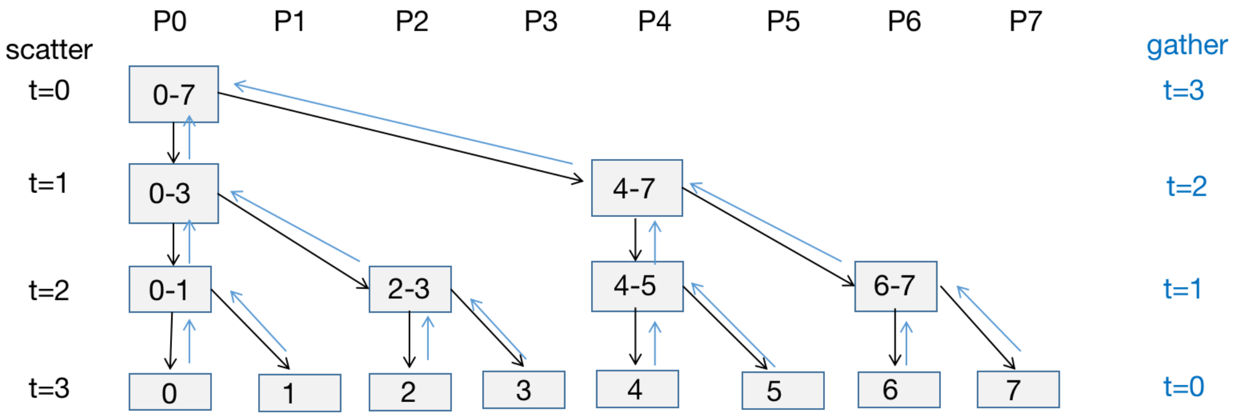
Suggestions Since we have noticed an inconsistency between the empirical results and the theoretical complexity analysis of the two algorithms, the next step could be to implement SUMMA with the same scatter-gather operations as in Cannon, to verify whether it is the case that the sendrecv_replace operation is slower than broadcast in practice. If this is the case, then it suggests either that the sendrecv_replace operation is not implemented in the way that I have assumed earlier, or that Cannon would not be applicable in practice. (This might explain why SUMMA is used widely in industry but not Cannon.)

Task 4

Implementation design The Pthread versions of the MPI-equivalent SUMMA and Cannon implementations are provided. The algorithmic workflow is essentially the same, i.e. `pth_summa(_2).c` and `pth_cannon(_2).c` are an equivalent implementation of `mpi_summa(_2).c` and `mpi_cannon(_2).c` respectively. Data communcation operations (Figure 13) such as alltoallw, scatter-gather, broadcast (and non-blocking ibroadcast), sendrecv_replace, non-blocking isend are implemented with semaphores and barriers. To simulate MPI, I assume that data to be sent are all copied into intermediate buffers and from there receiver copies to their destination memory address.



(a) broadcast



(b) scatter-gatter

Figure 13: logarithmic algorithms

Performance Note that since now we are operating on a shared memory system, the particular "round-up" scheme (`pth_summa_2.c`) I used for SUMMA involves allocation of the full size **A**, **B** and **C** for each thread, and because of this large storage requirement on a shared memory system, the performance of the program will be affected if it operates on a large program size and a high number of *nthds* because it will overflow the system memory. (It works for MPI because each process has a dedicated core). Hence it is better to avoid setting as inputs the combination of large *nthds* and large problem size.

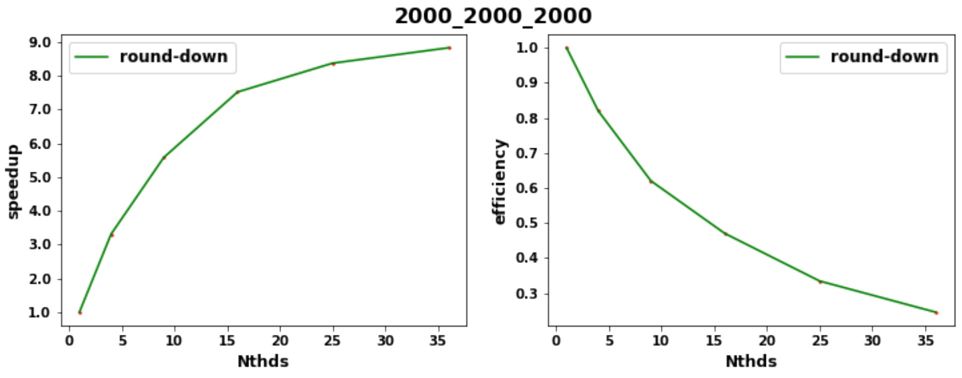


Figure 14: speedup and efficiency of SUMMA in Pthread

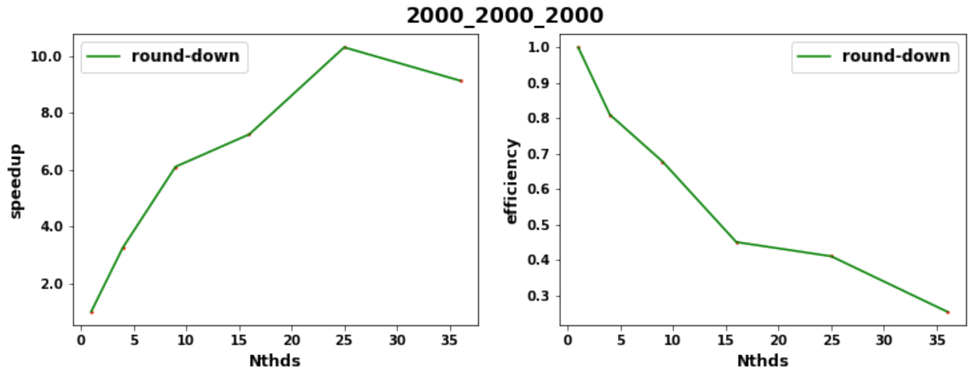
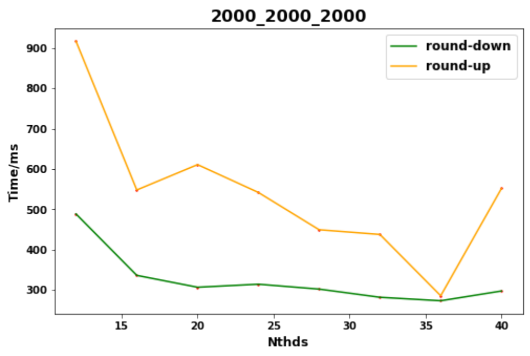
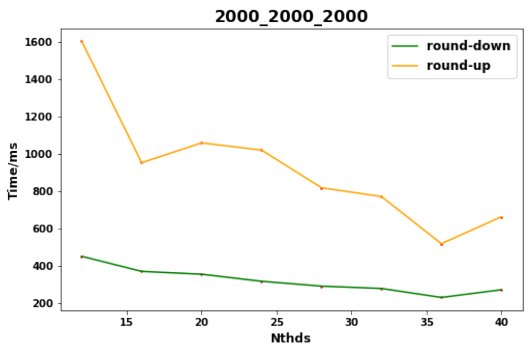


Figure 15: speedup and efficiency of Cannon in Pthread

From Figure 14 and 15, we observe that the "round-down" approach for both SUMMA and Cannon implemented in Pthread is no longer strongly scalable, as their efficiencies decrease with a increasing number of threads for a fixed problem size. (Recall that for MPI versions, they are strongly scalable). This difference is essentially due to the fact that my own implementation of the MPI 2-phase scatter-gather operations (despite O(logp)), broadcast operation (despite O(logp)), and sendrecv_replace operation is less efficient and not as optimised as the industrially used ones.

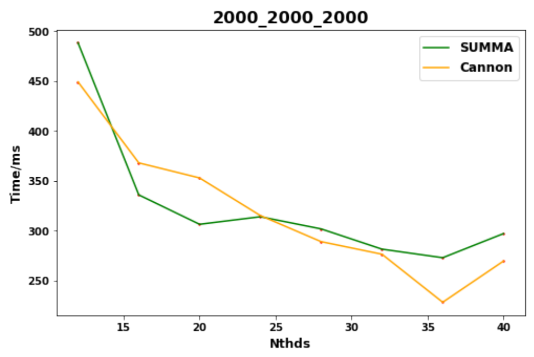


(a) SUMMA in Pthread

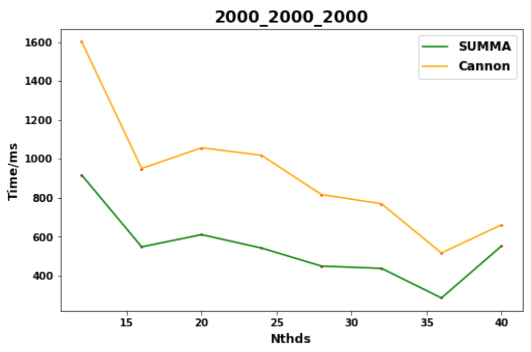


(b) Cannon in Pthread

Figure 16: "round-down" vs "round-up"



(a) ”round-down”



(b) ”round-up”

Figure 17: SUMMA vs Cannon in Pthread

The Pthread implementations of ”round-down” vs ”round-up” have consistent performance (Figure 16) as our earlier expectations: ”round-up” has much overhead associated with cyclic mapping, and hence performs much worse than ”round-down” for both SUMMA and Cannon. The running time dips of ”round-up” in both (a) and (b) occur at $nthds = 16$ and $nthds = 36$ which are square numbers that do not require cyclic mapping. Hence, our self-written Pthread implementations also confirm that cyclic mapping is not useful for SUMMA and Cannon.

We can also compare the Pthread versions of SUMMA and Cannon (Figure 17). Interestingly, with ”round-down” (a), Cannon overtakes SUMMA for larger values of $nthds$, and that the performance improvement (running time decrease) of SUMMA is more gentle than Cannon. In the earlier Task 3, I suspect that this is because my MPI version of SUMMA uses alltoallw whereas Cannon uses scatter-gather, and that SUMMA will be more performant than Cannon by using the same 2-phase scatter-gather operation. Hence to (dis)prove my suspection, in Task 4, I used the same 2-phase scatter-gather operations in ”round-down” for both SUMMA and Cannon. However, this performance result in Figure 17(a), consistent with the result in Figure 12(a), suggests my earlier suspection is wrong, and that scatter-gather has no effect on the performance difference between SUMMA and Cannon. Hence, it must be the case that the performance difference is a result from the algorithm difference of SUMMA and Cannon (i.e. broadcast vs sendrecv).

On the other hand, for ”round-up”, Cannon is stll worse than SUMMA. Hence, we might conclude that sendrecv with cyclic mapping will perform worse than broadcast with cyclic mapping; and that sendrecv alone works better than broadcast. Further analysis of the underlying cause of this interesting behaviour is required, yet one perspective that might worth exploring is that data shift operations (Cannon) are afterall cyclic whereas broadcasts (SUMMA) are tree-structured acyclic. This might impact the communcation sequence in some way, to my suspection.

Part B

Problem Description

Parallel sample sort is implemented with all four APIs : Pthread, OpenMP, MPI and CUDA.

- **Problem input**: an unsorted array a of length n
- **Problem output**: an array *output* that contains the sorted array a
- **Algorithm workflow (Sample Sort)**:
  1. Random bucket assignment for splitter selection
  2. Local splitter $(nthds/nprocs - 1)$ selection
  3. Broadcast local splitters to form global splitters $(nthds/nprocs * (nthds/nprocs - 1))$
  4. Global splitter sort and final splitter $(nthds/nprocs - 1)$ selection
  5. Bucket sort
  6. Broadcast bucket counts
  7. Write result to correct output position
- **Implementation design (Highlights):**
  - `pth_sort.c`:
    Step 1. Use `rand()` to implement random bucket assignment. (Not counted towards performance measurement because of its large overhead)
    Step 3. Broadcast into distributed memory (i.e. use Pthread to simulate MPI). Implement MPI alltoall broadcast (`MPI_Allgather`) operation with conditional variable.
  - `pth_sort_2.c`:
    Step 1. Use cyclic distribution to simulate random bucket assignment. The assignment approach is not really "random" but we argue that since the array is not sorted, we still distribute random values into each bucket. This improves efficiency and does not affect correctness.
    Step 3. Broadcast into distributed memory (i.e. use Pthread to simulate MPI). Implement MPI alltoall broadcast (`MPI_Allgather`) operation with semaphores.
  - `omp_sort.c`:
    Step 1. Use cyclic distribution to simulate random bucket assignment.
    Step 3. Broadcast local splitters into shared memory.
    Step 4. Sort global splitters and select final splitters in shared memory by Thread 0
  - `omp_sort_2.c`:
    Step 1. Use block distribution to simulate random bucket assignment.
    Step 3. Broadcast local splitters into shared memory.
    Step 4. Sort global splitters in shared memory by Thread 0; but select final splitters in private memory by each thread

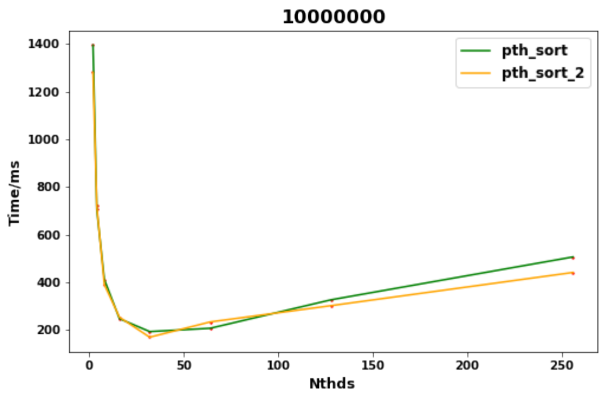
- mpi_sort.c:
  Step 1. Broadcast the entire array a by P0 such that all processes own a copy.
  Processes select buckets based on block distribution individually.
  Step 3. Broadcast into distributed memory by MPI_Allgather.
  Step 5. Processes sort buckets with their local copy of a.
  Step 6. Gather bucket counts by P0.
  Step 7. Gather sorted buckets by P0.

- mpi_sort_2.c:
  Step 1. Use block distribution by MPI_Scatterv to achieve random bucket assignment.
  Step 3. P0 collects all local splitters by MPI_Gather.
  Step 4. P0 sorts global splitters and select final splitters.
  Step 5 & 6. P0 assigns final buckets for all processes.
  P0 broadcasts bucket counts.
  P0 send buckets to respective processes point-to-point.
  Individual processes sort received buckets locally.
  Step 7. Gather sorted buckets by P0.

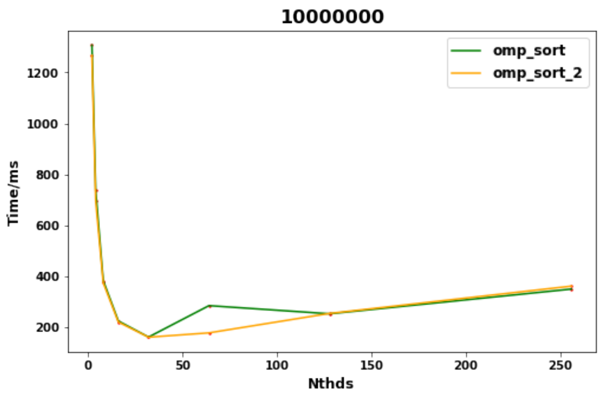
- cuda_sort.c:
  Only allow one thread block.
  Step 1. Use cyclic distribution to simulate random bucket assignment.

- cuda_sort_2.c:
  Allow multiple thread blocks.
  Use multiple kernels to achieve synchronisation between thread blocks.
  Step 1. Use cyclic distribution to simulate random bucket assignment.
  Step 4. Global splitter sort and final splitter selection in Host.

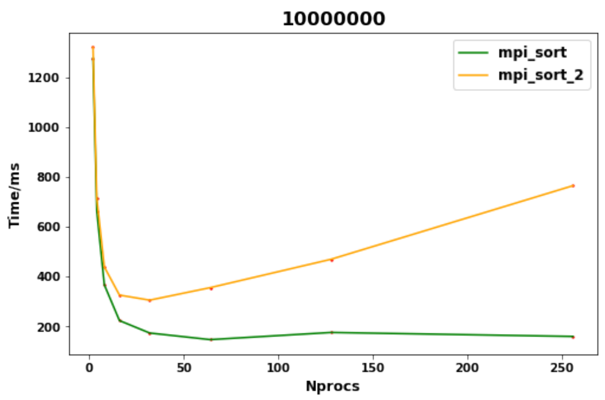
Performance We compare the performance of sample sort implemented with these four APIs.



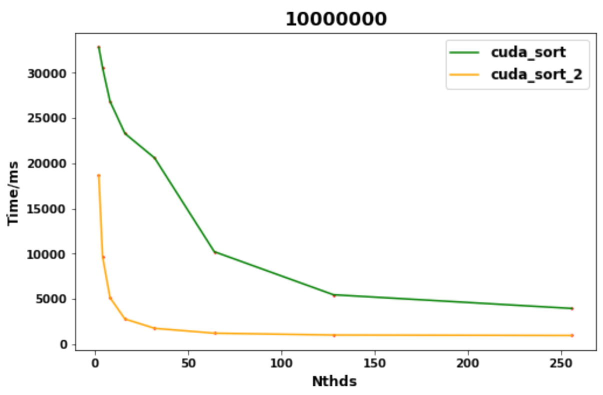
(a) Pthread



(b) OpenMP



(c) MPI



(d) CUDA

Figure 18: Performance of Sample Sort

The Pthread and OpenMP implementations are similarly performant because there is not much variation in terms of the algorithm structures. As mentioned before, `pth_sort.c` and `pth_sort_2.c` differ in terms of how they assign random buckets, i.e. `pth_sort.c` uses `rand()` (not measured) and `pth_sort_2.c` uses cyclic. They both simulate the MPI distributed memory system, by exchanging data with a communicator. The main data exchange in sample sort is the alltoall broadcast of local splitters. `pth_sort.c` uses conditional variable whereas `pth_sort_2.c` uses semaphores. Both ways implement *nthds* phases of sending and receiving to neighbours, in a cyclic fashion. Hence, the algorithms' time complexity should be similar. From (a), it appears that with increasing *nthds*, semaphores outperform conditional variable. Hence, it must be due to the reason that the conditional variable implementation requires a larger critical section, hence serializing more part of the program.

For OpenMP (b), `omp_sort.c` will be less performant than `omp_sort_2.c`, since there is one more barrier used in the parallel region. This additional barrier is used to wait for P0 to sort the global splitters and select final splitters in the shared memory. Only after P0 finishes can other processes fill their own buckets and perform bucket sort. In `omp_sort_2.c`, this barrier is removed by replacing the approach of P0 (as the leader) doing the selection for the rest, with the shared parallel approach of asking all processes doing the selection in their private memory separately. Hence, this removes the dependency and speeds up the sort. The discrepancy at value $nthds = 64$ is due to runtime difference.

For MPI (c), `mpi_sort.c` is significantly more performant than `mpi_sort_2.c`. The reason is because `mpi_sort.c` distributes the work of filling buckets (after final splitters have been selected) to individual processes by broadcasting the input array a to all processes. At first, it might seem that broadcasting the entire input will incur much overhead, so I seek for an alternative (`mpi_sort_2.c`) that does not require broadcasting the whole input. Yet this means that P0 has to fill every single buckets by taking an input element, then iterating all buckets to see which slot to put it in. This is a much more costly process, as the number of buckets increase, the more buckets P0 has to iterate just for putting a single element into a correct slot. This explains why the running time of `mpi_sort_2.c` increases drastically with large *nprocs*. Hence, this suggests that broadcasting the input to all processes and let them fill their own buckets is the right way to go.

For Pthread, OpenMP, and MPI, the general increase in running time after reaching the lowest point can be attributed to two factors: 1. overhead of thread creation and destruction w.r.t. problem size. 2. in splitter selection, the number of global splitters is of $O(nthds^2)$, hence the more threads, the more splitters need to be broadcasted. On the other hand, we see that CUDA has a general decrease trend for the running time. I believe its simply because the hardware architecture of GPU has a lot of small cores for data processing, and thus has more parallel computing resources than the rest of the platforms (e.g. running 1000+ threads would be no worries for CUDA but hard for Pthread and OpenMP).

For CUDA (d), `cuda_sort.c` is much less performant than `cuda_sort_2.c` because `cuda_sort.c` only supports single thread block operations, whereas `cuda_sort_2.c` supports any number of thread blocks. The running time of `cuda_sort_2.c` is taken with a default 1 thread per block. We do observe an inverse relationship between *nthds* and *time* for this setup. This means that speedup is linear, but when we evaluate its efficiency (Figure 19), we still observe a continuous decrease, which suggests the implementation is not strongly scalable. However, the implementation is in fact weakly scalable (Table 4), as its efficiency is almost constant when we increase $nthds = num\_blk * 1$ at the same rate as problem size.

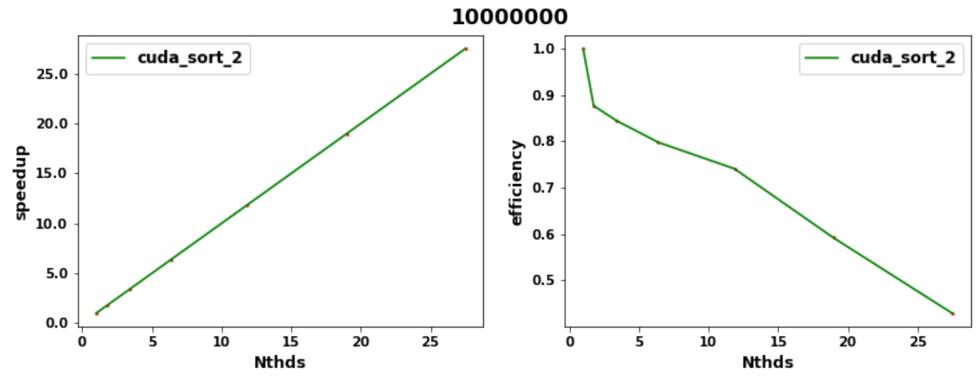


Figure 19: speedup and efficiency of sample sort in CUDA

num_blk	problem size	time (ms)	serial time (ms)	speedup	efficiency
1	1000000	3177.4	3177.4	1	1
2	2000000	3279.0	6273.8	1.91	0.95
4	4000000	3448.16	13613	3.95	0.99
8	8000000	3651.12	28238.6	7.73	0.97
16	16000000	4637.9	69449	14.97	0.94

Table 4: Weak scalability of `cuda_sort_2.c`

Also, another interesting finding with CUDA thread blocks is that for a fixed total number of threads, the more blocks we use, the better the performance. In the following example (Table 5), the total number of threads used is 32 and the problem size is 1000000.

num_blk	thd_per_blk	time/ms	speedup	efficiency
1	32	20582.2	1	1
2	16	11260.7	1.82	0.91
4	8	6238.6	3.3	0.82
8	4	3704.3	5.56	0.69
16	2	2438.8	8.44	0.52
32	1	1732.8	11.878	0.37

Table 5: Block configurations with `cuda_sort_2.c`

The running time is inversely associated with num_block, and the speedup factor almost is comparable to in Figure 19. What makes this surprising is that, in Figure 19, speedup is brought by increasing the number of available threads, whereas in Table 5, we have speedup just by distributing/rearranging the threads amongst multiple thread blocks. Hence, this suggests that for CUDA, the block configuration plays a very important role (decisive almost) in performance and programs should utilize this feature as much as possible. This also explains why in Figure 18(d), `cuda_sort.c` is so much worse than `cuda_sort_2.c`, as the single block configuration is the worst one amongst all possible configurations.

Suggestions Since CUDA operates on a completely different GPU architecture, it might be worthwhile to implement other parallel programs with CUDA to compare the performance difference, either cross-platform, or just to explore some of the surprising features that the NVIDIA GPU offers.

End