# Homework 2

total: 15 pts
late penalty: -10% = -1.5 pts
FINAL SCORE: 13.5 pts

Dr. Michal A. Kopera

**ME 471/571**, Spring 2019 **submission due: Mar 8th**

Homework done by: Arash Modaresi Rad & Mark Angelo Mijares

This homework requires you to parallelize the matrix-matrix multiplication and perform a scalability study of your implementation. There are two extra credit opportunities, which require considerable work, but you can score extra points accordingly.

You will submit a document addressing the questions outlined in this assignment, as well as the code you have written. Please remember to clearly comment your code, so it is easily readable. For the report, please use Google Docs or PDF format. The final submission needs to be made by Friday, March 8th. Even though it may seem like a lot of time, it is very important that you start early, and seek feedback and help if you need it along the way.

Please take advantage of the Office Hours (both virtual and in-person) to help you with the homework. Discussion and collaboration between students is also encouraged, but I will not tolerate any plagiarism (i.e. copying each others code, results, comments). If you are using any external help, please acknowledge that in your document. Any copying of the code available on the Internet will be considered plagiarism, and will be dealt with through official BSU channels. By putting your name at the top of the report, you acknowledge that you take responsibility for the submission, and understand all aspects of the submitted work. If you cannot complete this assignment (with help from me during office hours), it will be very difficult for you to continue in this course. Please make effort to implement and understand the required algorithm by yourself.

## Matrix-matrix product in parallel

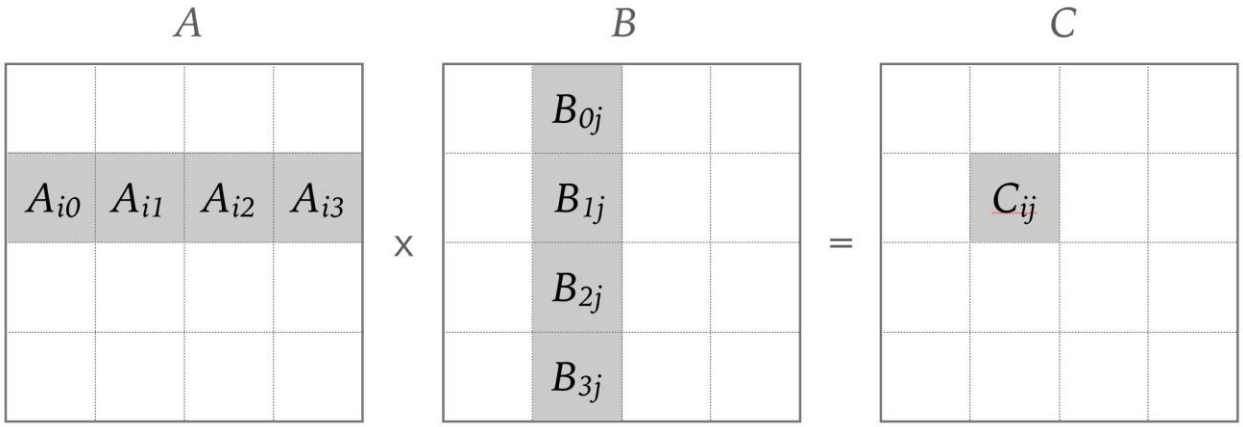
As you know, in order to compute a single entry of matrix C, which is a product of matrices A and B ($C = AB$, and assume $A, B$ are square matrices of size $n \times n$), you need an entire row of matrix A, and an entire column of matrix B. This follows from the simple algorithm you examined in Homework 1:

```
for (i = 0; i < n; i++)   for
(j = 0; j < n; j++)       for
(k = 0; k < n; k++)
      C[i][j] += A[i][k]*B[k][j];
```

To parallelize this operation, we need to decide how we will distribute the data among processes. Assume we have $p$ processes, and $p$ is a square number (i.e. $q = \sqrt{p}$ is an integer), we want to distribute $A, B, C$ among processes such that each process owns a sub-matrix $A_{i,j}, B_{i,j}, C_{i,j}$ of size $n/q \times n/q$, and processes are organized in an $q \times q$ grid with the coordinates $(i, j)$ representing the location of the process in the grid, and the global rank of the process $(i, j)$ can be obtained as $r = i * q + j$. In the example depicted here we use $p = 16$, which leads to $q = 4$. To compute sub-matrix $C_{i,j}$ we then need not only $A_{i,j}$ and $B_{i,j}$, but also the entire row of sub-matrices $A_{i,0}, A_{i,1}, \ldots, A_{i,q-1}$, and the entire column of sub-matrices $B_{0,j}, B_{1,j}, \ldots, B_{q-1,j}$

process grid

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |




$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots + A_{i,q-1}B_{q-1,j}.$$

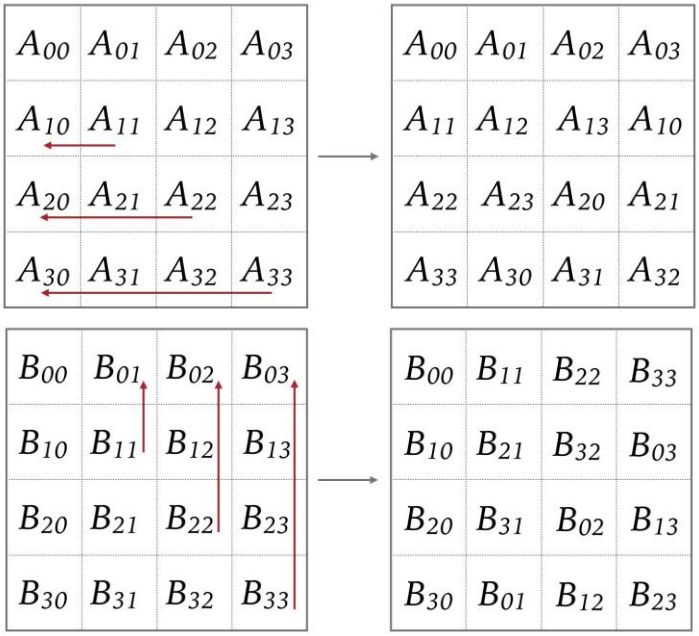
The naive algorithm would then require to gather $n/q$ rows of A and $n/q$ columns of B onto rank $(i, j)$, and then compute $C_{i,j}$. This would require us to store a total of $2q + 1$ submatrices of size $n/q \times n/q$ on each process, for a total of

$(2q + 1) \cdot \frac{n}{q} \cdot \frac{n}{q} = \frac{(2q + 1)n^2}{q^2} \approx \frac{2n^2}{\sqrt{p}}$ numbers. The total memory required for all processes is then approximately $\frac{2n^2}{\sqrt{p}} \cdot p = 2n^2\sqrt{p}$, which is considerably more than the serial algorithm $p$

memory footprint of $3n^2$ (the space needed to store three $n \times n$ matrices). To alleviate that problem, a Canon algorithm was devised, which assumes that we only store a single $A_{i,j}$, $B_{i,j}$, $C_{i,j}$ on each process $(i, j)$, leading to a per-process memory requirement of $3n_2/p$, and $3n^2$ for all processes. The Canon algorithm is executed in the following steps:

1) Initially, process $(i, j)$ owns sub-matrices $A_{i,j}$, $B_{i,j}$, $C_{i,j}$

2) We perform initial alignment of sub-matrices within the process array by sending each block $A_{i,j}$ to process $(i, (j - i + q) \bmod q)$, and each block $B_{i,j}$ to process $((i - j + q) \bmod q, j)$ - see image below (left is before shift, right is after)




3) Each rank multiplies its local $A$ and $B$ sub-matrix, and increments its local $C_{i,j} += A_{loc}B_{loc}$. For example, $C_{1,2} += A_{1,3}B_{3,2}$.

4) Each local sub-matrix $A$ is shifted to the neighbor directly to the left, i.e. $(i, j) \rightarrow (i, j - 1)$, and each sub-matrix $B$ is shifted to the neighbor directly above, i.e. $(i, j) \rightarrow (i - 1, j)$. This


shift assumes periodicity, so the left-most process sends its $A$ to the right-most process in the same row, and top-most process sends its $B$ to the bottom-most process.

5) Compute $C_{i,j} += A_{loc}B_{loc}$, where $A_{loc}$, $B_{loc}$ are currently locally stored sub-matrices by process $(i, j)$. For example, after the first shift $C_{1,2} += A_{1,0}B_{0,2}$

6) Repeat steps 4 and 5 for a total of $q - 1$ times, until the complete $C_{i,j}$ is computed:

$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots + A_{i,q-1}B_{q-1,j}$$

**Complete the following tasks:**

(a) (5pts) Assuming that the time required to execute a parallel message-passing algorithm is: $T_p = T_{comput} + T_{latency} + T_{bandwidth}$, where $T_{comput} = \gamma \cdot ops$ is the computation time required to perform $ops$ floating point operations, $T_{latency} = \alpha \cdot m$ is the latency incurred by sending $m$ messages, and $T_{bandwidth} = \beta \cdot d$ is the time required to send $d$ double precision numbers, create a performance model for the Cannon algorithm and derive an expression for speed-up ($S_p = T_1/T_p$)

and efficiency ($E_p = S_p/p$). How does the efficiency depend on the number of processes $p$? Can the Cannon algorithm scale strongly?

Considering that matrices A and B are of size n * n and that p is the number of processors; each process would do $\sqrt{p}$ times multiplications o submatrices size of $(\frac{n}{\sqrt{p}}) \times (\frac{n}{\sqrt{p}})$ . Therefore, the computation time required to perform ops floating point operations per each process is $\frac{n^3}{p}$.

$$T_{comput} = \frac{n^3}{p} * \gamma$$

And $T_{latency} + T_{bandwidth}$ of a shift is defined as:

$$T_{latency} + T_{bandwidth} = \alpha * m + \beta * d = \alpha * 1 + \beta * \frac{n^2}{p}$$

so, for our two shift operations we have:

$$T_{latency} + T_{bandwidth} = 2(\alpha + \beta * \frac{n^2}{p})$$

and considering each of the single step shifts in the for loop ($\sqrt{p}\ times$) we can write

$$T_{latency} + T_{bandwidth} = 2\sqrt{p}(\alpha + \beta * \frac{n^2}{p})$$

Summing up produces the total time as

$$T_{comput} + T_{latency} + T_{bandwidth} = \frac{n^3}{p} * \gamma + 2\sqrt{p}(\alpha + \beta * \frac{n^2}{p})$$

For a single processor we have:

$$T_1 = \frac{n^3}{p} * \gamma$$

The speed-up ($S_p = T_1/T_p$) would be:

$$S_p = \frac{\frac{n^3}{p} * \gamma}{\frac{n^3}{p} * \gamma + 2\sqrt{p}(\alpha + \beta * \frac{n^2}{p})} = \frac{n^3 * \gamma}{n^3 * \gamma + 2p^{3/2}\ (\alpha + \beta * \frac{n^2}{p})} \quad (1)$$

and the efficiency ($E_p = S_p/p$):

$$E_p = \frac{\frac{n^3 * \gamma}{n^3 * \gamma + 2p^{3/2} \ (\alpha + \beta * \frac{n^2}{p})}}{p} = \frac{n^3 * \gamma}{p * n^3 * \gamma + 2p^{5/2} \ (\alpha + \beta * \frac{n^2}{p})} \quad (2)$$

it is better to express efficiency as 1/(1+…)

Ideally, we can say $T_p = T_1 / s$

T_p = T_1 / p in the ideal case, what is s?

$E_p = S_p/p = T_1/(T_p * p) = T_1/(p * T_1/ s) = 1$

So, in an ideal situation we can expect 100% efficiency as we increase the number of processors. But in reality as can be seen in equation (2) computation, latency and bandwidth costs will not allow 100% efficiency to be achieved.

One would expect that Cannon algorithm scale strongly for large problem sizes as for large problem sizes the compute time portion of equation (2) will severely be reduced. However, the efficiency will drop due to increase in latency and bandwidth costs. As a result, plotting the strong scaling efficiency would allow us to chose the optimum number of processors for a given problem size.

5 PTS

##########

(b) (8 pts) Implement Cannon latency and bandwidth costs as outlined above. To compute local matrix-matrix products on each processor, use the best algorithm you came up with in Homework 1. For the clarity of code, encapsulate your matrix-matrix algorithm in a function, so you do not paste the same code over and over again.

Your implementation of Cannon itself looks correct, but you are timing the scatter and gather of serial matrix, which will affect your timing significantly.

Cannon algorithm is implemented in cannon_alg.c

8 pts

matrix-matrix algorithm and some other frequently used functions are implemented in functions.c

run_cannon_alg.qsub is used for submitting jobs to R1

the run script you used varies the matrix size, but not the processor count

##########

Your scatter and gather operations do not work correctly, as you are not sending blocks of A and B to processes, but instead chop the memory of matrix A and B into p chunks and send it over. As a result, local a and b will hold rows of A and B, and not blocks. Also, you are testing your algorithm on matrices of all ones, which is way too simple test. I used some other numbers, and get wrong results.

(c) (2 pts) Show that your algorithm produces correct result by comparing the resulting matrix with a matrix computed by a serial algorithm.

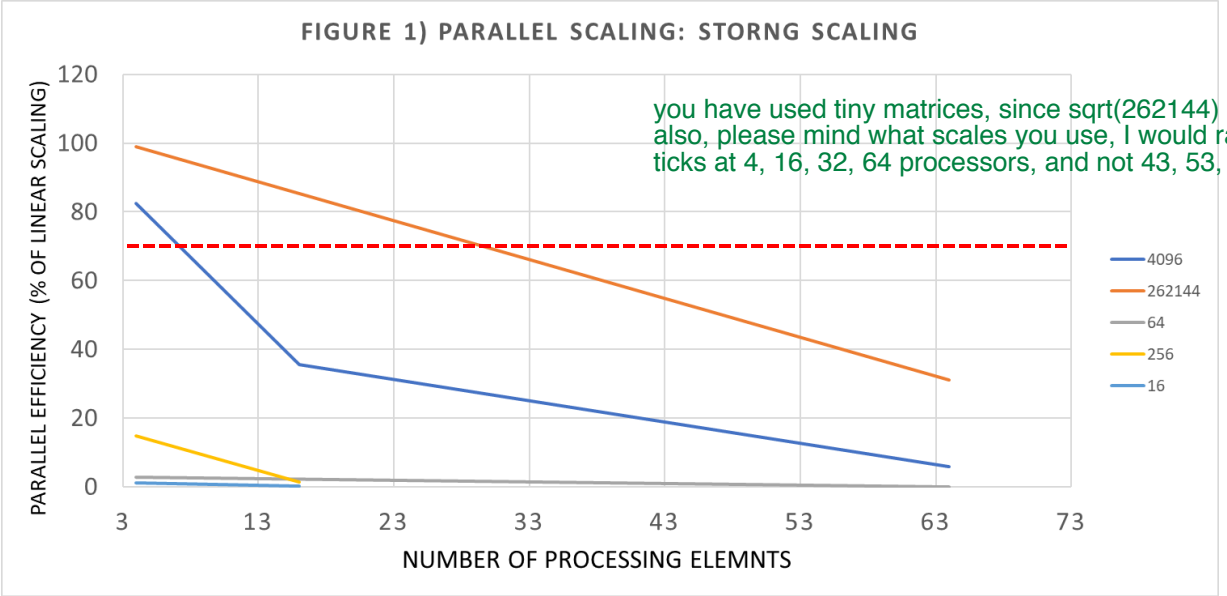
A section is implemented for computing serial matrix-matrix multiplication and estimating its time in cannon_alg.c

the sum of elements of C_serial and C from parallel are used to compare they are equall. 0 pts

##########################################################################################

(d) (5 pts) Create both strong and weak scaling plots for *p* ranging from 1 to 128 processes. 2 pts

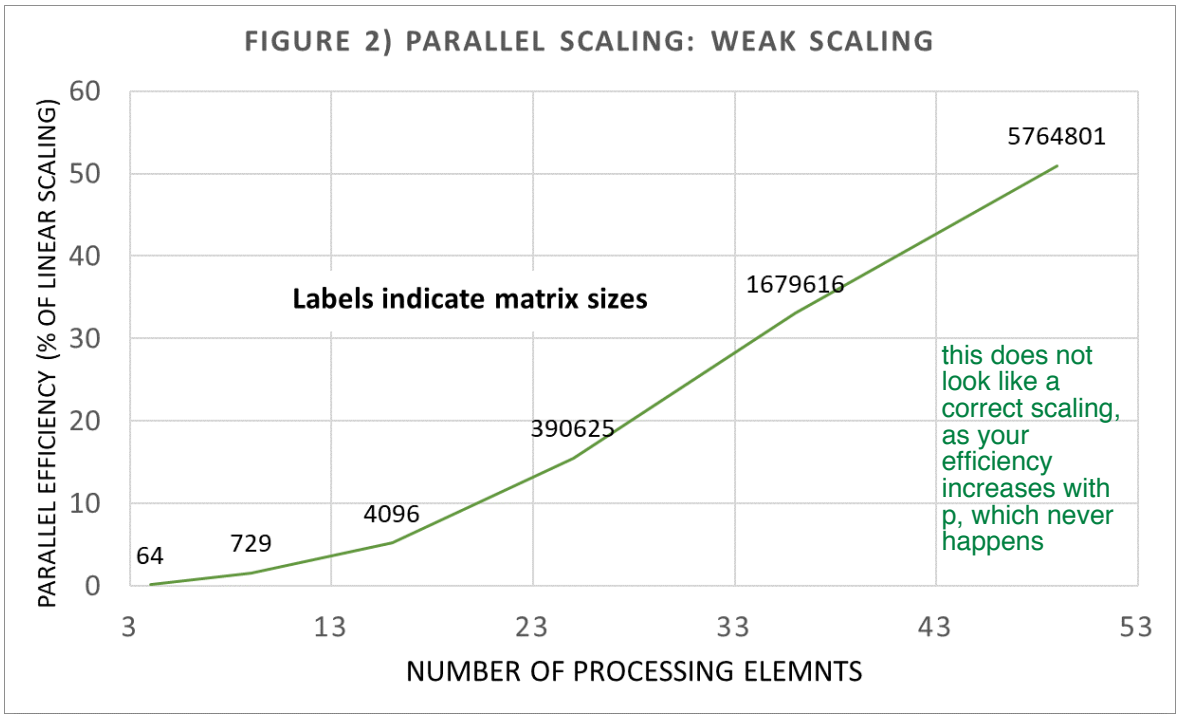
You should be able to perform strong scaling for matrix size of at least 2048 by 2048 (but you can try larger sizes to see if it fits in memory). Comment on both scaling plots.




you have used tiny matrices, since sqrt(262144) = 512
also, please mind what scales you use, I would rather see ticks at 4, 16, 32, 64 processors, and not 43, 53, 63

I asked for matrices size up to 2048x2048 or so, naturally for tiny sizes you will not get any benefit, as it is much more efficient to solve them in serial

As evident from figure (1), implementing cannon algorithm is ineffective for small matrix sizes. Assuming that obtaining more than 75% efficiency is reasonable criteria for us in tradeoff between having a larger number of processors (more expense) and faster computation, it can be seen that matrix sizes of 4096 and 262144 can be used in cannon algorithm with high efficiency. For matrix size of 4096, 4-8 processors can produce high efficiency and for matrix size 262144, 4-29 processors produce high efficiencies.




For a weak scaling problem, instead of keeping the problem size fixed, we increase the problem size relative to the number of cores. This helps us to make an educated guess at the amount of computing time we need. As illustrated in figure (2), parallel efficiency relevant to weak scaling increases as we increase the number of processors as well as the matrix size.

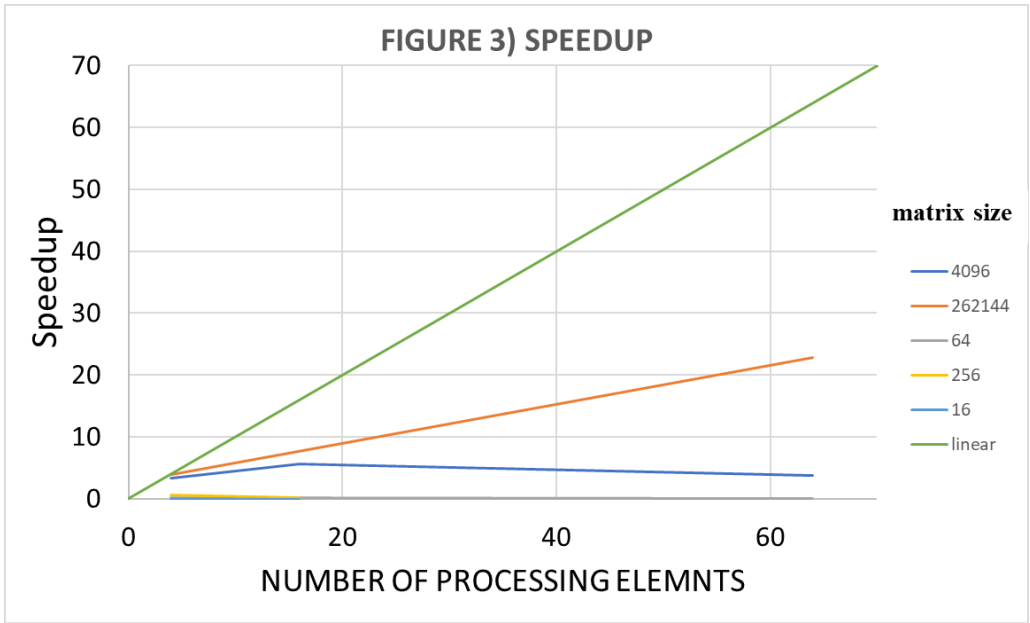


Figure (3) shows that as the size of the matrix increases, we obtain better speedup values. However, none reaches the ideal speedup due to the presence of latency and band wide costs shown in equation (1).

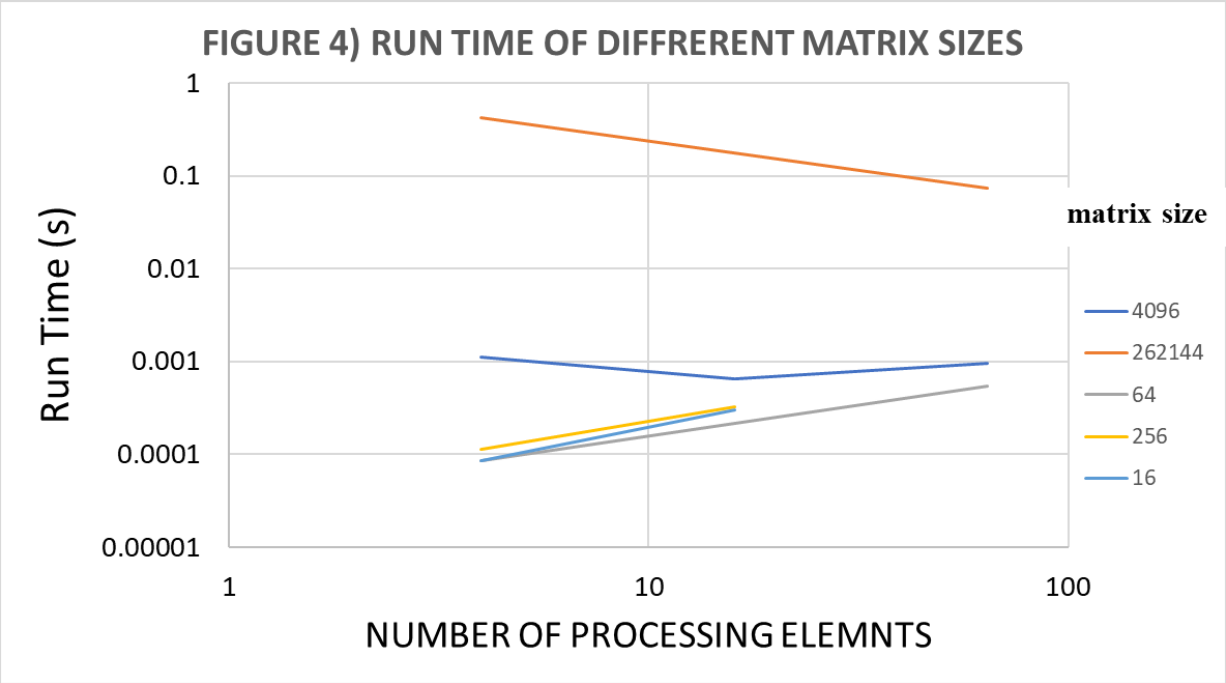



Figure (4) shows that as expected the run time of the smaller matrix sizes (i.e., 16, 64, and 256) increases as more processors are being used and this mainly because the problem is too small for computation algorithm and we are increasing communication in as we increase the number of processors. On the other hand, matrix sizes 4096 follows almost a constant runt time and solving matrix size 262144 with more processes reduces the run time.

Figure 5) comparison of computation time on basis of given matrix size between serial and parallel approaches

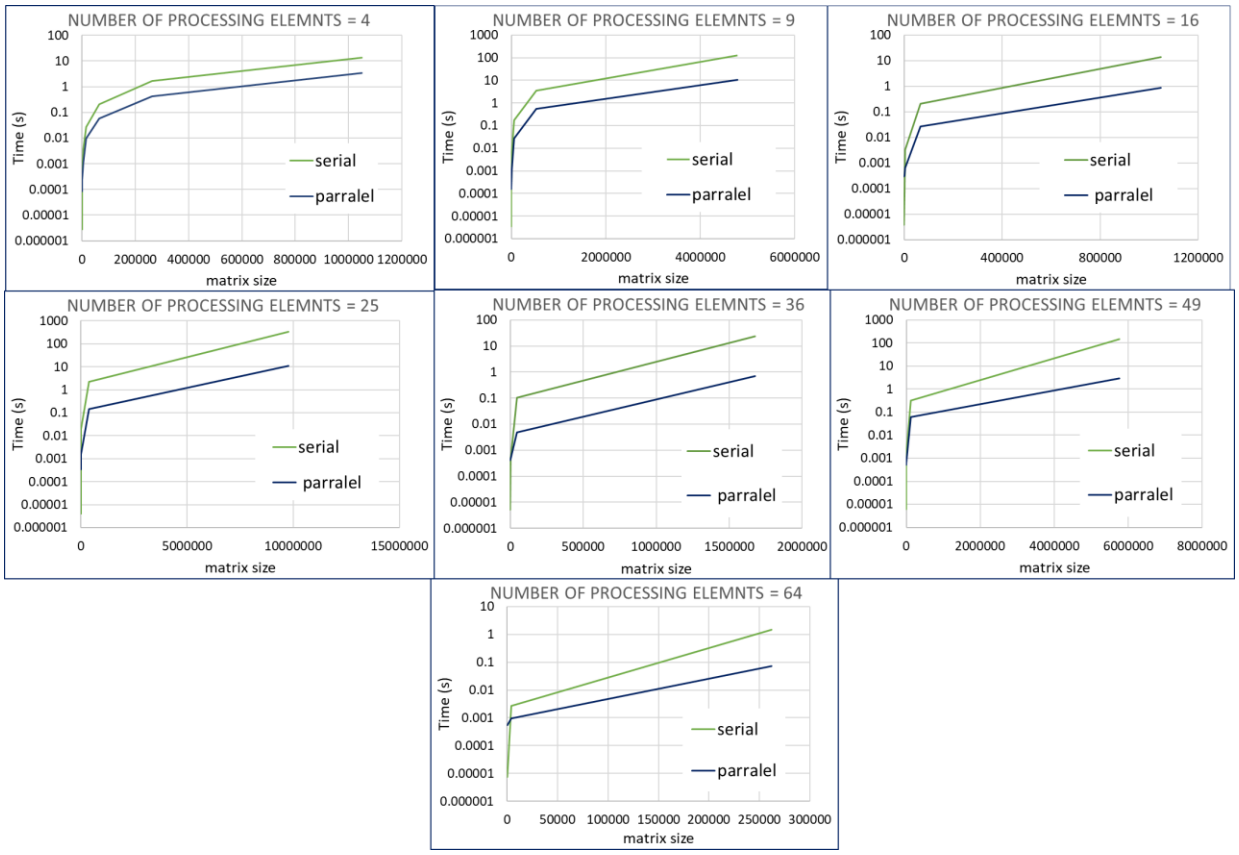



Figure (5) clearly shows that for smaller sizes of matrix have smaller computation time in serial as opposed to parallel. Increasing the matrix size will reduce the computation time and that is where cannon algorithm preforms best.

##########################################################################

(e) (Extra Credit) Can you come up with estimates for $\alpha, \beta, \gamma$ from point (a)? You could create simple programs to estimate how much time is required to compute a single FLOP ($\gamma$), what is the latency required to set-up a single MPI message ($\alpha$) and how long does it take to send a single double precision number over the R1 network ($\beta$). When you substitute your estimates to your performance model and plot a strong scaling efficiency, how does it compare with your actual strong scaling for the Cannon algorithm?

##########################################################################

(f) (Extra Credit) Another parallel matrix matrix algorithm is the DNS algorithm (Dekel, Nassimi, Aahni). Contrary to the Cannon algorithm it performs a 3D memory decomposition. You can find many references in the Internet. For this extra credit assignment, implement the DNS algorithm, create a strong and weak scaling plots, and compare your results against the Cannon algorithm.