CSE 708: SEMINAR ON PROGRAMMING MASSIVELY PARALLEL SYSTEMS

Implementing Parallel Matrix Multiplication using SUMMA and MPI

SAKSHI MEHRA(50424433)

University at Buffalo The State University of New York

AGENDA

- Matrix Multiplication Definition and Use Case
- Process
- Sequential Approach
- Parallel Approach
- Results
- Conculsion
- References

MATRIX MULTIPLICATION

- Given two matrices Matrix A of size mxn with elements a_{ij} and Matrix B of size nxp with elements b_{jk}) . Matrix C is the product of A and B with size mxp

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, ..., m$ and $j = 1, ..., p$.

NOTE: Number of Columns of A = Number of Rows of B

SEQUENTIAL APPROACH

ITERATIVE ALGORITHM

Complexity:

- The algorithm takes Θ(nmp) time.
- If input are square matrices of size nxn, the runtime is cubic i.e. $\Theta(n^3)$

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- For i from 1 to n:
  - For j from 1 to p:
    - Let sum = 0
    - For k from 1 to m:
      - Set sum $\leftarrow$ sum + $A_{ik} \times B_{kj}$
    - Set $C_{ij} \leftarrow$ sum
- Return C

ikj vs ijk

Speed increases because cache hit increases.

```
0,0 | 0,1 | 0,2 | 0,3
----+-----+-----+----
1,0 | 1,1 | 1,2 | 1,3
----+-----+-----+----
2,0 | 2,1 | 2,2 | 2,3
```

```
0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3
```

```
x[0][0]
        x[0][1]
                x[0][2]
                        x[0][3]
                                x[1][0] etc...
```

```
x[0][0]
                        x[1][0]
                                                x[2][0]
      x[0][1]
                              x[1][1] etc...
```

RESULTS - SEQUENTIAL

Matrix Dimensions	Time(ms) ijk	Read optimized Time(ms) ikj
1000 X 1000	1868	2516
2000 X 2000	29496	26394
3000 X 3000	104528	75631
4000 X 4000	273488	195324
5000 X 5000	1047400	344091
6000 X 6000	5446480	528849

PARALLEL APPROACH –SUMMA(SCALABLE UNIVERSAL MATRIX MULTIPLICATION ALGORITHM)

- Uses a shift algorithm to broadcast
- The SUMMA algorithm computes n partial outer products:

  for k := 0 to n − 1 C[:, :] + = A[:, k] · B[k, :]

- Each row k of B contributes to the n partial outer products
- Communication Phase - Matrix A and Matrix B are divided into submatrices based on the number of processors in the grid and sent to their respective processors.

Data splitting/ Communication Phase

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Sub-Matrix of Matrix A and B are sent to Processors

Calculation Phase/ Computation Phase

- Required Column Sub-Matrix of Matrix A is either within the Processor or is brought in from a different processor
- Calculating result of new column and row in the submatrices.
- For each Processor Partial Resultant Matrix obtained.
- Complete Resultant Matrix of the matrix product betweeen A & B obtained.

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Caculation Description Box

Calculation Phase -> Current Processor = P1

Required Column Sub-Matrix of Matrix A is within Processor 1.

Required Row Sub-Matrix of Matrix B is within Processor 1.

Sub-Matrix A

1	2
5	6

Sub-Matrix B

1	2
5	6

Sub-Matrix C

1	2
5	10

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Caculation Description Box

Calculation Phase -> Current Processor = P1

Calculating result of new column and row in the submatrices.

Sub-Matrix A

0	0
0	0

Sub-Matrix B

0	0
0	0

Sub-Matrix C

11	14
35	46

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Caculation Description Box

Calculation Phase -> Current Processor = P1

Required Column Sub-Matrix of Matrix A is brought from Processor 2 to 1.

Required Row Sub-Matrix of Matrix B is brought from Processor 3 to 1.

Sub-Matrix A

3	4
7	8

Sub-Matrix B

9	10
13	14

Sub-Matrix C

38	44
98	116

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Caculation Description Box

Calculation Phase -> Current Processor = P1

Calculating result of new column and row in the submatrices.

Sub-Matrix A

0	0
0	0

Sub-Matrix B

0	0
0	0

Sub-Matrix C

90	100
202	228

Matrix A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix B

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrix C

90	100	0	0
202	228	0	0
0	0	0	0
0	0	0	0

Processor Grid

P1	P2
P3	P4

Caculation Description Box

Calculation Phase -> Current Processor = P1

Processor 1 Partial Resultant Matrix obtained.

Sub-Matrix A

0	0
0	0

Sub-Matrix B

0	0
0	0

Sub-Matrix C

0	0
0	0

Algorithm 3 SUMMA Algorithm

for $k = 0$ to $n - 1$ **do**
  for all $i = 1$ to p_r **do**
    owner of $A(i, k)$ broadcasts it to whole processor row;
  end for
  for all $j = 1$ to p_c **do**
    owner of $B(k, j)$ broadcasts it to whole processor column;
  end for
  Receive $A(i, k)$ into Acol
  Receive $B(k, j)$ into Brow
  $C_{myproc} = C_{myproc} + \text{Acol} * \text{Brow}$
end for

SUMMA Performance

Time = $\frac{2}{p} * n^3 + \alpha \log p * \frac{n}{b} + \beta logp * n^2/s$

- Parallel Efficiency =

  1/(1 + α * log p * p / (2*β*n^2) + β * log p * s/(2*n))
- ~Same β term as Cannon, except for log p factor

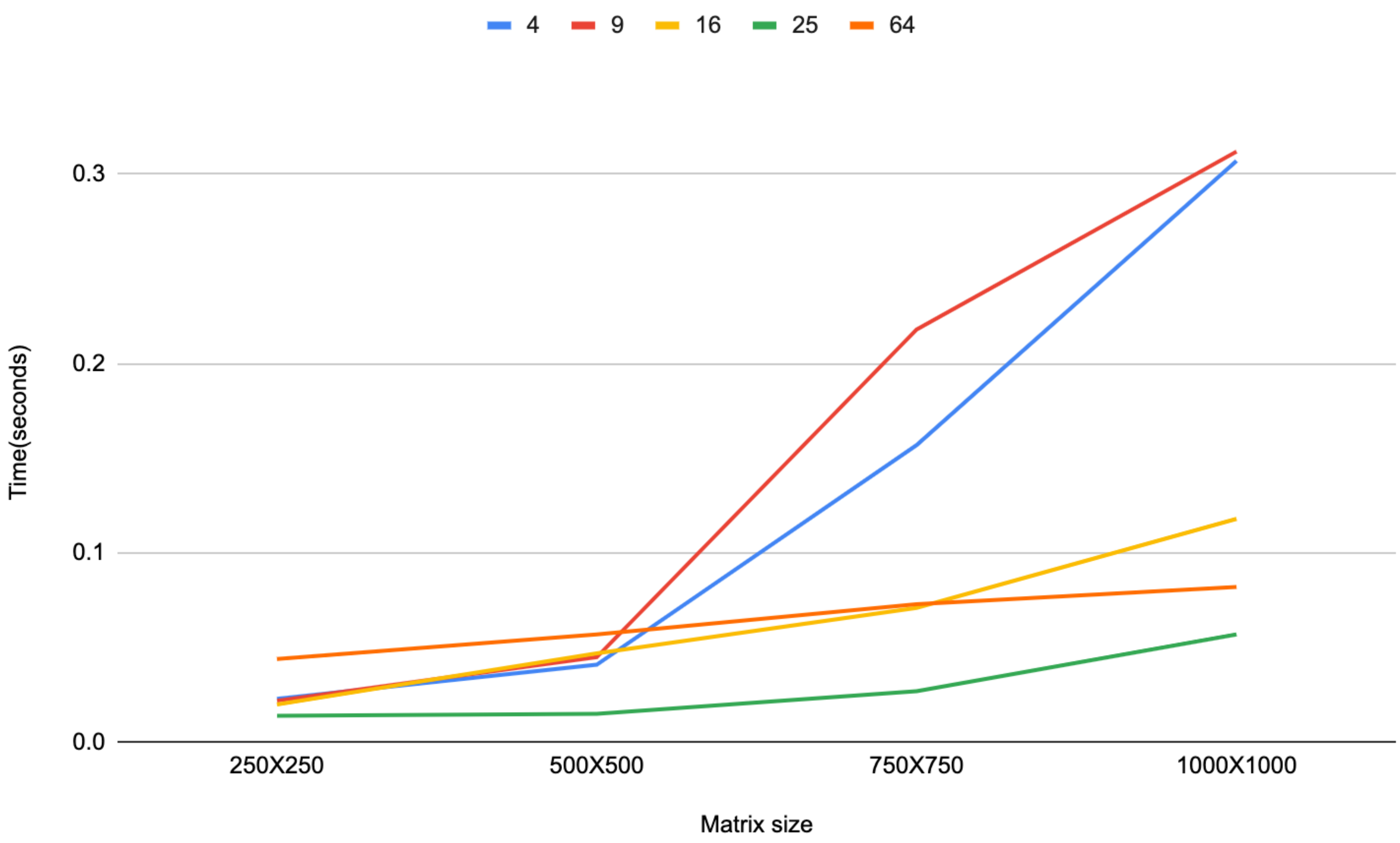
  log p grows slowly so this is ok
- Latency (α) term can be larger, depending on b

  When b=1, get α * log p * n

  As b grows to n/s, term shrinks to

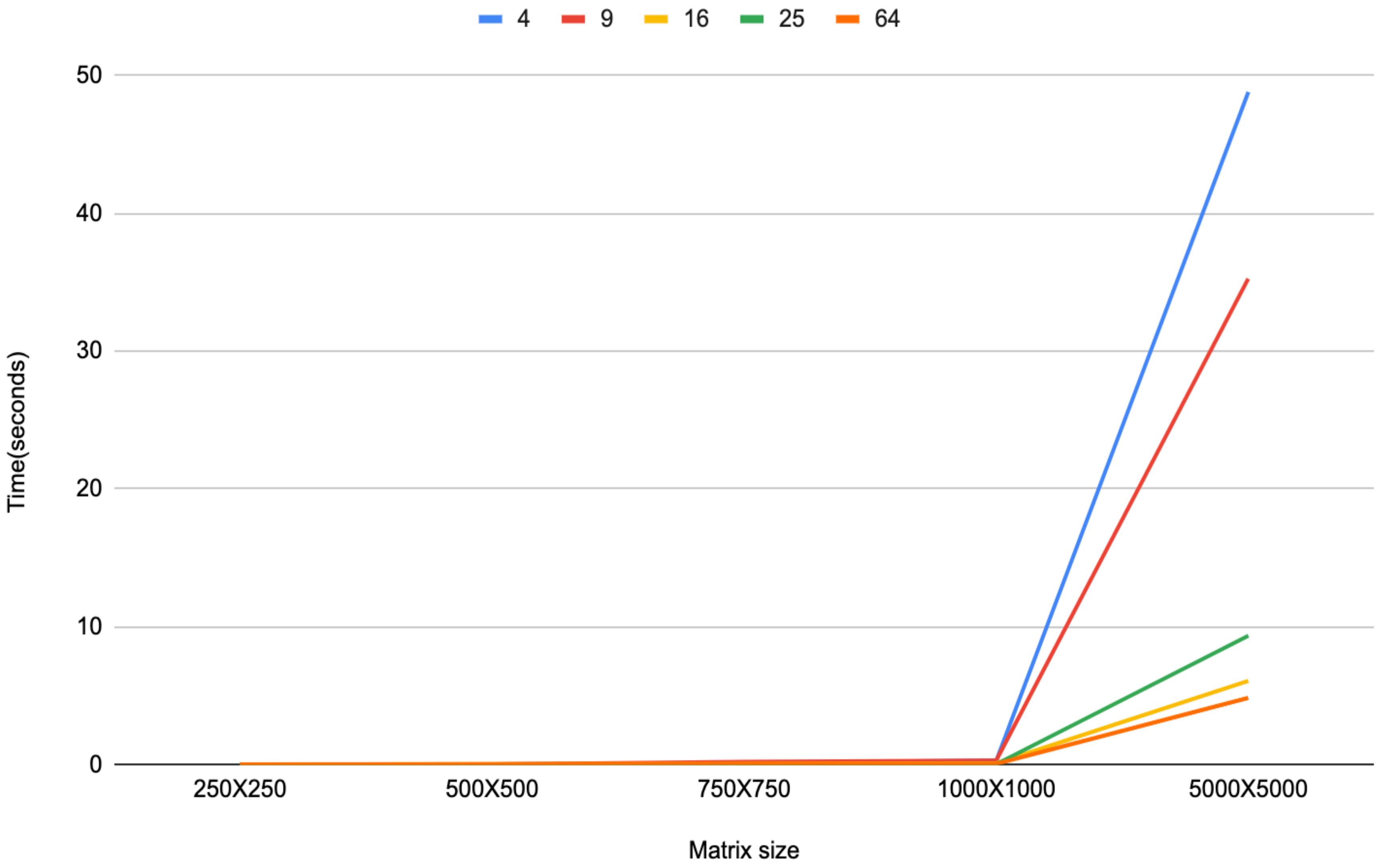
  α * log p * s (log p times Cannon)
- Temporary storage grows like 2*b*n/s
- Can change b to tradeoff latency cost with memory

Processors	250 X 250	500 X 500	750 X 750	1000 X 1000	5000 X 5000	10000 X 10000
4	0.02265	0.0411683	0.15687	0.306646	48.7919	~
9	0.021585	0.10323	0.218011	0.312147	44.7802	~
16	0.0196146	0.0469481	0.0710892	0.118495	6.08682	89.0198
25	0.0136477	0.014682	0.0272895	0.0574698	9.36909	76.853
64	0.0441307	0.0566216	0.0728194	0.0821494	4.8575	34.8661
121	0.951168	0.881796	1.00411	1.01392	1.82618	23.2469
144	1.85181	1.86337	1.87949	1.88167	2.56661	21.5294
225	1.05108	1.09527	3.30053	1.09419	3.98347	13.3343
625	0.372938	0.117778	0.203478	0.160495	0.234925	1.78691



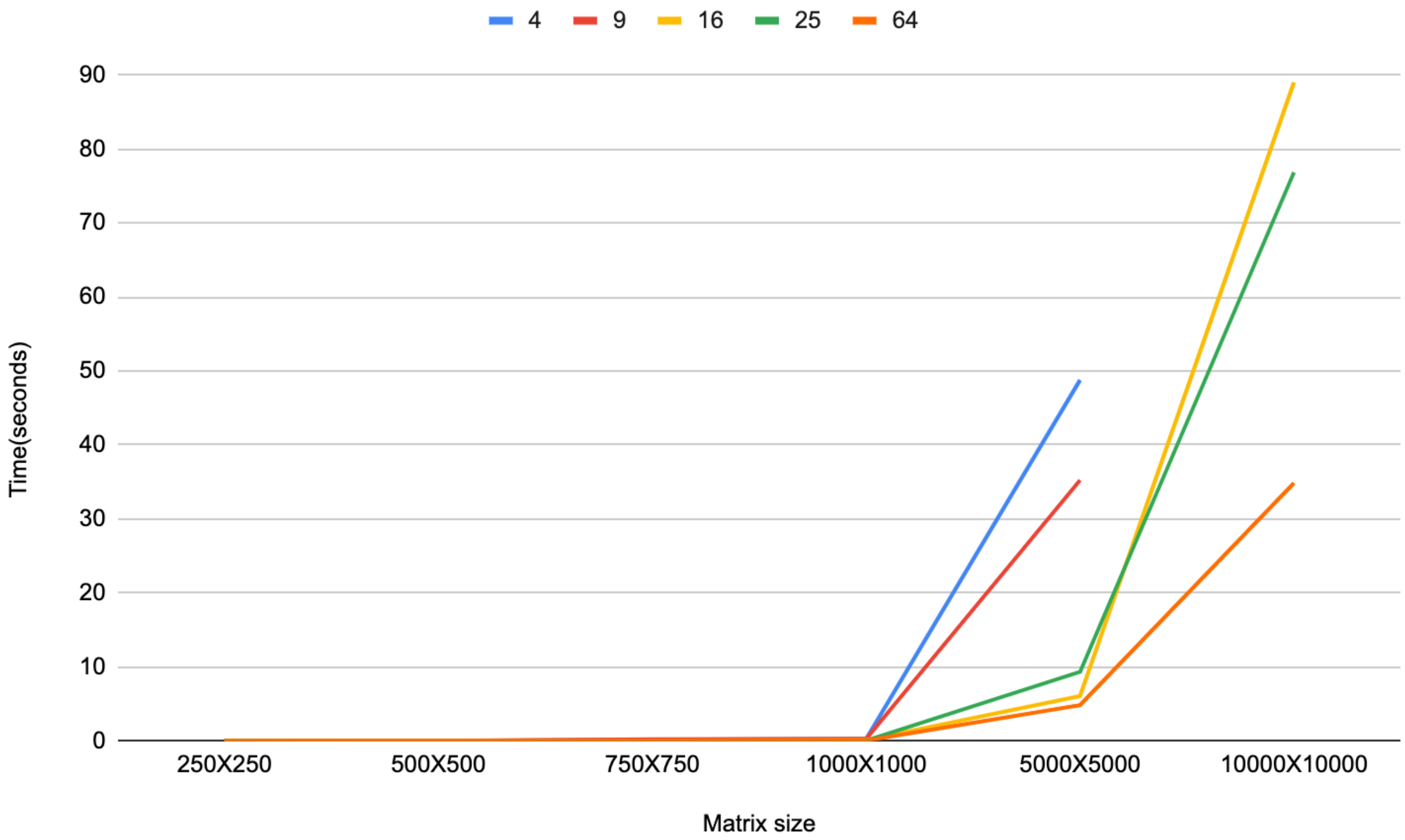
Observations:

- For data sizes upto 500 X 500, the performance is almost same across all nodes.
- After 500 X 500, the performance with lesser number of nodes worsens drastically.
- At 1000 X 1000, using number of nodes 4 and 9, yields large runtime. The performance with using number of nodes as 16, 25, 64, yields similar results, 25 number of nodes giving slightly better results.



Observations:

- With matrix sizes increasing above 1000, the runtime increases sharply for all node count.
- The difference in runtime with using 16, 25, and 64 nodes is also visible at this size.
- Using 64 number of nodes gives the least runtime.



Observations:

- Above 5000 X 5000, the runtimes increase exponentially, 64 number of nodes work best.

Results summary

Matrix size	Number of nodes with best runtime
250 X 250	4
500 X 500	25
750 X 750	25
1000 X 1000	25
5000 X 5000	64
10000 X 10000	64

Key Takeaways

1. Using higher number of nodes(25+) gives worse results for smaller matrix sizes, since the communication time between nodes becomes the largest factor in runtime.

2. The use of larger number of nodes is beneficial for large matrix sizes and 64 nodes work best.

3. Runtimes with less than 64 number of nodes increase exponentially for matrix sizes above 5000 X 5000.

References

- http://www.cs.csi.cuny.edu/~gu/teaching/courses/csc76010/slides/Matrix%20Multiplication%20by%20Nur.pdf
- https://cs.iupui.edu/~fgsong/LearnHPC/summa/index.html
- https://www.andrew.cmu.edu/user/haewonj/documents/codml19_full_summa.pdf

Thank you!