Parallel and Distributed Programming: Matrix-Matrix Mutliplication

Peili Guo

***Abstract*—We describe, implement Matrix-Matrix Multiplication using Fox's and Cannon's algorithm with MPI in C. We test the code on high performance computers using large inputs and compare the efficiency of the code running on different number of cores. Our results achieve super linear speed up with faster data access in cache.**

I. Introduction

A. Background

Matrix multiplication is an important, basic and central operation in linear algebra with a wide range of application in computational problems [5], for example, numerical analysis, financial product pricing, graph theory, etc. [6] Given two matrices, A and B, where A is $n \times m$ matrix and B is $m \times p$ matrix, the matrix product $C = A \times B$ is computed using the following formula:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \times B_{k,j} \tag{1}$$

In this paper, we discuss square dense matrix matrix multiplication, which means that both A and B are $n \times n$ matrices. It is straightforward to see that computing $C = A \times B$ is of $O(n^3)$ time complexity [3]. We need to compute $n \times n$ of C and each $C_{i,j}$ needs to multiply pair wise the i^{th} row in A and the transverse of the j^{th} column in B. We implement both Fox's and Cannon's matrix matrix multiplication algorithm in C using MPI (Message Passing Interface) and test the performance of the parallel code on large parallel computers Uppmax at Uppsala University.

B. Partition

The implementation of matrix matrix multiplication is targeting to execute on large distributed memory architecture, partition of the matrices is required. In this paper, we use Checkerboard (2D-type) partition.[4] The processing elements forms a 2D Cartesian grid. [7] An illustration is displayed below. For the multiplication of matrices A, B of size $n \times n$, we assign a total number of p processors and assume that the n is divisible by $\sqrt{p}$, and p is a perfect square number. In other words, we divide the matrix in to $\sqrt{p} \times \sqrt{p}$ blocks and the size of each block is $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$.

$A^{0,0}$	$A^{0,1}$	$A^{0,2}$	...	$A^{0,p-1}$
$A^{1,0}$	$A^{1,1}$	$A^{1,2}$	...	$A^{1,p-1}$
$A^{2,0}$	$A^{2,1}$	$A^{2,2}$	...	$A^{2,p-1}$
...	...	...	...	...
$A^{p1-1,0}$	$A^{p1-1,1}$	$A^{p-1,2}$	...	$A^{p-1,p-1}$

C. Fox's Algorithm

After creating Checkerboard partition as described in section I-B, we will have a Cartesian grid topology with processors $P_{s,r}$, $s, r = 0, ..., \sqrt{p}$. Matrices A and B are partitioned into blocks and send to the corresponding processor. Fox's algorithm is described as follows [4] :

for $w = 0 : \sqrt{p} - 1$

1) Broadcast block m of A within each block row i where $m = (i + k)$ mod $\sqrt{p}$.
2) Multiply the broadcasted block with the residing B block in each processor on processor $P_{s,r}$: $C_{i,j} = C_{i,j} + A_{i,m} \times B_{m,j}$.
3) Shift the blocks of B one step cyclicly up, use non-blocking communication and overlap with step 2.

end for

D. Cannon's Algorithm

The partition for matrices A and B are the same as Fox's algorithm. Cannon's algorithm require initial shift of the distribution of the matrices A and B [2].

1) Start up phase with initial shift
   a) Shift the i^{th} block row of A i steps cyclicly to left.
   b) Shift the j^{th} block column of B j steps cyclicly to up.
2) Phase two compute phase
   for $w = 0 : \sqrt{p} - 1$
   a) $k = (i + j + s) mod \sqrt{p}$
   b) Compute C in each processor: $C_{i,j} = C_{i,j} + A_{i,k} \times B_{k,j}$
   c) Shift A one step to left cyclicly
   d) Shift B one step to up cyclicly

   end for

II. Implementation in C with MPI

The Checkerboard partition is implemented using:

```
MPI_Cart_create()
MPI_Cart_coords()
```

Blocks are created with:

```
MPI_Type_create_subarray()
MPI_Type_create_resized()
MPI_Type_commit()
```

The input data of matrix A and B are only read on processor 0 to 1D dynamic allocated array, and then use `MPI_Scatterv()` for data distribution.

In the implementation of Fox's algorithm, we use `MPI_Bcast()` to Broadcast A to all processors in the row

communicator. and use `MPI_Isend()` and `MPI_Irecv()` to exchange local array among different processors.

In the implementation of Cannon's algorithm, we use `MPI_Cart_shift()` for the initial shift of A and B. `MPI_Sendrecv_replace()` is used to exchange local array among different processors.

After the parallel computing, the result of matrix C is collected using `MPI_Gatherv()`.

III. Experiment and Result

For the experiment of the code, we first check the correctness of the code with small input, for example 3x3, and 4x4 matrix matrix multiplication. After confirming the correctness of the code, we systematically run our code on Rackham, Uppmax with input of matrices of size 3600×3600, 7488×7488, and 9072×9072 on 1, 4, 9, 16, 36, 64, 81, 144, 256 processors.

A. Correctness check

Using the following 3×3 matrices A and B, we test the multiplication of $A \times B$ on both 1 core and 9 cores using both Fox's and Cannon's algorithm.

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & 10 & 11 \end{bmatrix} \times \begin{bmatrix} 13 & 14 & 15 \\ 20 & 21 & 22 \\ 24 & 25 & 26 \end{bmatrix}$$

We get the correct results:

$$\begin{bmatrix} 125 & 131 & 137 \\ 353 & 371 & 389 \\ 581 & 611 & 641 \end{bmatrix}$$

To reconfirm the correctness of code, we try the following 4×4 matrices A and B, we test the multiplication of $A \times B$ on both 1 core, 4 cores, and 16 cores using both Fox's and Cannon's algorithm.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 \end{bmatrix}$$

We get the correct results:

$$\begin{bmatrix} 280 & 290 & 300 & 310 \\ 696 & 722 & 748 & 774 \\ 1112 & 1154 & 1196 & 1238 \\ 1528 & 1586 & 1644 & 1702 \end{bmatrix}$$

B. Result

The summary of execution time using Fox's algorithm is presented in Table I. We can see that the parallel part run time reduce significiantly when we increase the number of processors assigned for the matrix matrix multiplication. From running on single core to 4 cores, the parallel matrix multiplication part achieved 7 times speed up for 3600×3600 matrices, and around 5 times speed up for 7488×7488 and 9072×9072 matrices. The speed up exceeds the theoretical speed up that with p cores we are expecting p times speed up. It is plain to see similar speed up from running on multi cores using Cannon's algorithm in Table II. The parallel of both Fox's and Cannon's algorithms is very efficient. For 3600×3600 matrices, we achieve the best performance at running on 64 cores using Cannon's algorithm at 10.58s and on 81 cores using Fox's algorithm at 9.81s. When assigning more than 64 cores for the previous mentioned matrices, the communication overhead is getting larger and larger and the code runs slower. For 7488×7488 and 9072×9072 matrices, the parallel part of the code still improves the performance using both algorithms up to 256 cores. At the same time, it is easy to see that the overhead is getting large, the parallel section of the code only improved less than 20% for the parallel part when comparing the performance from running on 144 cores and 256 cores. Needless to say, the file input time become very significant when we run more than 16 cores for 3600×3600 matrices, 36 cores for 7488×7488 matrices, and 64 cores for 9072×9072 matrices.

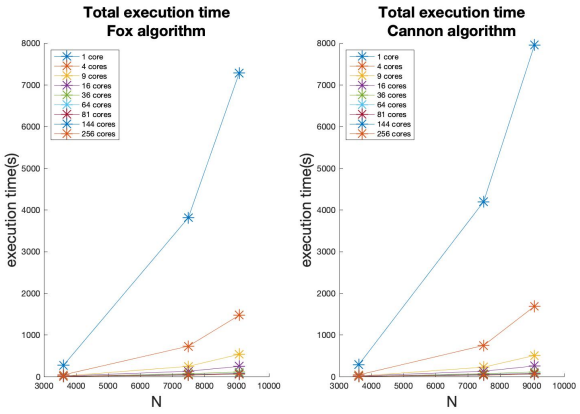

Fig. 1. Comparison of total execution time of matrix matrix multiplication running on different cores using Fox's and Cannon's algorithm

In Figure 1 we plot the total execution time again the matrix size n. We all agree that running on single processor, the total time increase rapidly. The algorithms is of $O(n^3)$ time complexity. Running on multi-core, the execution time increase slightly, yet the exact increasing in time from multiply 3600×3600 matrices to 7488×7488 matrices, to 9072×9072 matrices is difficult to directly observe in Figure 1 because of the scale of the figure. When taking a careful look at Table I we can see that running on 64 cores, the total time for $n = 3600$ is $12.88s$, $n = 7488$ is $51.58s$, and $n = 9072$ is 87.42.

IV. Discussion

To evaluate the performance of the code, we plot the strong scale speed up for both algorithms in Figure 2, where we keep the work load constant and increase the number of processors. The black dotted line is the theoretical speed up of parallel code. It would be embarrassing to miss the fact that the parallel implementation of our code is better than the theoretically speed up. Indeed, the reason behind is faster access to local array data with more cache hit. On Rackham, where the code is tested, has L1 cache of $32kB$ and L2 cache of $256kB$, which

TABLE I
SUMMARY OF EXECUTION TIME USING FOX'S ALGORITHM

N	Cores	Parallel time(s)	Readfile time(s)	Total time (s)
3600	1	269.24	6.47	275.71
3600	4	36.75	7.65	44.40
3600	9	14.65	7.65	22.31
3600	16	7.69	7.72	15.41
3600	36	3.07	7.83	10.89
3600	64	2.25	7.96	10.21
3600	81	2.39	7.42	9.81
3600	144	3.33	7.75	11.08
3600	256	5.45	7.43	12.88
7488	1	3791.30	25.96	3817.26
7488	4	696.33	32.99	729.32
7488	9	214.17	33.87	248.05
7488	16	97.07	32.70	129.77
7488	36	39.66	34.29	73.95
7488	64	18.60	32.98	51.58
7488	81	20.19	32.09	52.28
7488	144	9.66	33.43	43.09
7488	256	8.35	32.10	40.45
9072	1	7249.36	37.93	7287.29
9072	4	1435.55	46.77	1482.32
9072	9	493.60	48.10	541.70
9072	16	204.16	48.05	252.21
9072	36	63.68	49.22	112.90
9072	64	38.22	49.20	87.42
9072	81	28.27	47.00	75.27
9072	144	17.50	49.48	66.98
9072	256	14.97	47.11	62.08

Parallel time is only for the part of computing local matrix matrix multiplication.

TABLE II
SUMMARY OF EXECUTION TIME USING CANNON'S ALGORITHM

N	Cores	Parallel time(s)	Readfile time(s)	Total time (s)
3600	1	284.18	7.31	291.49
3600	4	36.81	7.91	44.72
3600	9	12.68	7.37	20.05
3600	16	7.49	8.00	15.49
3600	36	3.09	8.12	11.20
3600	64	2.23	8.34	10.58
3600	81	2.38	8.21	10.59
3600	144	3.29	7.82	11.10
3600	256	5.29	7.80	13.09
7488	1	4168.77	29.76	4198.53
7488	4	717.51	33.22	750.73
7488	9	201.21	31.99	233.21
7488	16	97.71	34.28	131.99
7488	36	41.17	35.54	76.71
7488	64	18.16	35.99	54.15
7488	81	19.81	35.69	55.50
7488	144	9.36	33.88	43.24
7488	256	8.41	33.72	42.13
9072	1	7916.71	42.82	7959.54
9072	4	1644.30	47.46	1691.77
9072	9	461.85	46.01	507.86
9072	16	212.02	50.24	262.26
9072	36	61.55	50.58	112.13
9072	64	35.94	51.16	87.10
9072	81	27.43	52.31	79.75
9072	144	16.98	49.50	66.49
9072	256	15.36	49.78	65.15

Parallel time is only for the part of computing local matrix matrix multiplication.

can fit 4096 and 32768 doubles respectively. Both algorithm's partition the original matrices into blocks of size $\frac{n}{\sqrt{p}}$. The matrices are stored in 1D dynamic allocated array, to perform the local multiplication, we need to multiply pairwise a row of A with a column of B. For example when computing $C_{1,1}$, we need to load $A_{1,:}$ and $B_{:,1}$. $A_{1,1}$,$A_{1,2}$,$A_{1,3}$... are continues allocated, but $B_{1,1}$,$B_{2,1}$,$B3,1$,... are not. If $B_{1,1}$ is loaded into cache line, So is $B_{1,2}$, $B_{1,3}$, until the limit of cache size. If we have a cache miss for $B_{2,1}$ in the same cache line in L1, we will search for it in the next level of Cache. The corresponding time to access in Cache is different, For L1 cache, there is 1-5 cycles latency, and 10-20 cycles for L2 cache, and much slower for L3 cache [8]. For the smaller size of local A and B, the cache miss significantly reduced, therefore we observe a better than theoretical speed up.

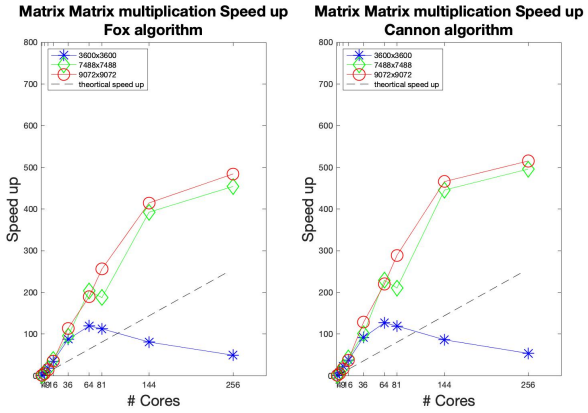

Fig. 2. Strong scalability of matrix matrix multiplication with Fox's and Cannon's algorithm

In Figure 3 we plot the weak scale speed up of both algorithms, where we keep relatively the workload constant in each processor. $c = \frac{n}{\sqrt[3]{p}}$, where c is constant. We observe that for both algorithm, we have better speed up when we keep c at 3600, that corresponds to running 3600×3600 on 1 core, 7488×7488 on 9 cores, and 9072×9072 on 16 cores. There are more cache hit in L1 and L2 cache [1], so the time for loading one data on average is faster, and there is not much communication required running on 9 and 16 cores (for loop size is of 3 and 4 respectively). However, the same effect cannot be observed for the rest. For example when we carefully observe the black line with '+' marker that plot the scale of 3600×3600 on 16 cores, 7488×7488 on 144 cores, and 9072×9072 on 256 cores, the cache hit are around the same level, since we can fit most of them in L2 cache for computing a single value of $C_{i,j}$ but not able to fit the block in L1.

It is interesting to see the decline of speed up when using more than 64 cores for 3600×3600 matrices. The communication time is greater than the potential reduce with cache hit. We are likely to observe the similar pattern for 7488×7488 and 9072×9072 matrices, that after an optimal number of cores, the speed up will reduce.

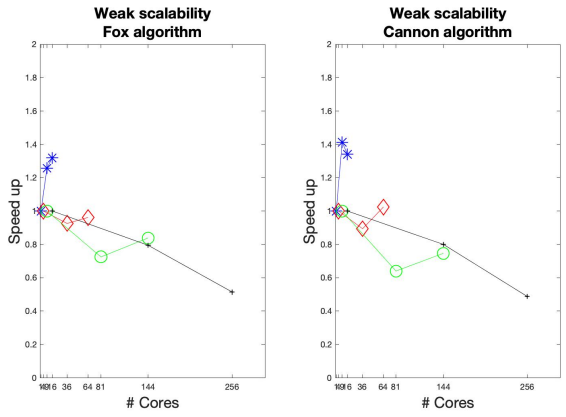

Fig. 3. Weak scalability of matrix matrix multiplication using Fox's and Cannon's algorithm

V. Conclusion

In this paper, we successfully implement matrix matrix multiplication using Fox's and Cannon's algorithm using MPI on large distributed memory system Rackham at Uppsala university. We test the code systematically using 3 different sizes of input matrices, namely 3600×3600, 7488×7488, and 9072×9072. Our code is tested on both correctness and performance. The performance of Fox's and Cannon's algorithm is very similar. The best performance are the following, based on parallel time, bracket time is total run time:

3600×3600 2.23s (10.58s) with Fox on 81 cores
7488×7488 8.35s (40.45s) with Fox on 256 cores
9072×9072 14.97s (62.08s) with Fox on 256 cores

The speed up for the implementations exceed the theoretical speed up and achieved almost 500 times speed up when running on 256 cores for some test cases. We conclude that the faster data access with more cache hit is the key for the successful implementation.

We notice some limitations of the code where the file input time becomes significant when running on more than 16 cores for 3600×3600 matrices, 36 cores for 7488×7488 matrices, and 64 cores for 9072×9072 matrices. The next priority for improving the code is to have more efficient data input and output. For example using binary files or store the matrices in different parts and let each processor read the allocated local part.

VI. Appendix

Rackham.uppmax.uu.se properties:
Model name: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Available cores: 40
Compiler: gcc version 9.1.0 (GCC)
mpirun (Open MPI) 3.1.3

References

[1] Anchev, N., & Gusev, M., & Ristov, Sasko., & Atanasovski, B., (2012). Optimal Cache Replacement Policy for Matrix Multiplication. [https://link-springer-com.ezproxy.its.uu.se/content/pdf/10.1007%2F978-3-642-37169-1_7.pdf]

[2] Cannon, L.E., (1969) A cellular computer to implement the Kalman filter algorithm. https://scholarworks.montana.edu/xmlui/handle/1/4168

[3] Cormen, TH, & Leiserson, CE, & Rivest, RL (2009). Introduction to Algorithms, MIT Press, Cambridge. pp.594- 602. Available from: ProQuest Ebook Central. [12 May 2019].

[4] Fox, G.C., & Otto,S.W., &Hey, A.J.G.,(1987) Matrix algorithms on a hypercube I: Matrix multiplication https://doi.org/10.1016/0167-8191(87)90060-3.

[5] Lee, H.J., &Robertson, J.P., & Fortes, J.A.B., (1997) Generalized Cannon's algorithm for parallel matrix multiplication [doi:10.1145/263580.263591]

[6] Pan, V.Y.,(2017) Fast matrix multiplication and its algebric neighbourhood [https://doi.org/10.1070/SM8833]

[7] Schatz, M.D., & Van de Geijn, R.A.,& Poulson, J.,(2016) Parallel Matrix Multiplication: A Systematic Journey [DOI.10.1137/140993478]

[8] Lecture notes for High Performance Programming, (2019) Uppsala University