# Report to Test 1

Student: Shuzhan Sun

## Flow of the Code

Considering the realistic case that the input and output matrices are very large, it is very expensive to read and distribute the matrix from one process, or to collect the calculated matrix to single process. Thus, in my implementation, I do the parallel matrix multiplication in such a way:

1. Each process generates its local matrix blocks  and , where the number of rows or columns in each block is managed with block-distribution. In “row distributed matrix multiply”, each process generates some rows of matrix A and the whole matrix B. In “Cannon’s algorithm”, each process generates a small block of matrix A and a small block of matrix B;
2. Each process multiplies the local block matrices and add to the final matrix , either with “row distributed matrix multiply” or with “Cannon’s algorithm”;
3. Eventually, each process outputs its local C component to a txt file.

The timing only counts the step 1) and step 2) above.

In the MKL code, I use the build-in function *cblas\_dgemm* in MKL library and follow a [similar example code](https://software.intel.com/sites/default/files/managed/2d/f8/mkl_2018_tutorial_c.pdf) provided by Intel. *cblas\_dgemm*, by default, will parallel the matrix multiplication. In the code, I set the number of parallel threads using the *mkl\_set\_num\_threads* routine.

## Sanity Check

I validate the code with s simple matrix multiplication. I use 4 processes and put the output files in the subdirectory “SanityCheck”. All the codes give the correct result.



## Timing for Fixed Problem Size

Matrix A is , matrix B is , the elapsed time vs. number of process:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | numP = 1 | numP = 2 | numP = 4 | numP = 8 | numP = 16 |
| Row distributed | 27.2757s | 13.4578s | 6.5548s | 4.09674s | 1.97949s |
| Cannon | 27.4036s |  | 6.29411s |  | 1.86352s |
| MKL | 1.24298s | 0.42297s | 0.287631s | 0.214842s | 0.127659s |
| Sequential | 18.1071s |  |  |  |  |

The speedup , where the sequential time  is from the sequential code. The efficiency . The Karp-Flatt e is calculated with K-F Metric: .

Plot these data and parameters in the following graphs. The Cannon algorithm has a better speedup than that of the row distributed multiplication. Also, efficiency shows the Cannon’s algorithm uses the machines more effectively. The Karp-Flatt e shows that the Cannon’s algorithm has smaller serial fraction.





## Isoefficiency Analysis for Scaled Problem Size

To study the scalability of two algorithms, I run at different problem sizes, but all matrices A, B, and C are square  matrices. The execution times are:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | N | numP = 1 | numP = 2 | numP = 4 | numP = 8 | numP = 16 |
| Row distributed | 1000 | 2.34665 | 0.951703 | 0.689691 | 0.375425 | 0.167259 |
| 2000 | 17.4985 | 9.16869 | 5.21004 | 2.90943 | 1.54816 |
| Cannon | 1000 | 2.25013s |  | 0.55816s |  | 0.172257s |
| 2000 | 18.2531s |  | 4.77207s |  | 1.3632s |

From this, the Cannon’s algorithm scales better when problem size increases. The overall efficiency of Cannon’s algorithm is better.

