Master 1 MoSIG

# Parallel Algorithms and Programming

Lab 5 report

Member:
Son-Tung DO
Johana MARKU

Grenoble, April 17, 2019

# 1 Fox Algorithm

We successfully implemented the Fox's algorithm with MPI.

Since we do not handle the data distribution in this exercises, all the processes are initialized with a value $a$ and $b$ as a block of matrix A and B. Also a variable $c$ to store the result of computation of the final matrix in the same block.

Given the number or process available N, we always create a matrix of size $n \times n$ where $n = \sqrt{N}$ if $N \mod 2 = 0$ or $\sqrt{N-1}$ if $N \mod 2 \neq 0$

We create a grid process, and 2 sub-communicator are for row and column. The row communicator is used for broadcasting the value of matrix A as every stage, the whole row using the same value of a certain block in that row. The column matrix is used for shifting the value of matrix B.

*N.B: The code provided has the matrix value initialized in each block corresponded to the same rank of the process holding it for the purpose of testing, the code for random value also provided*

# 2 Cannon Algorithm

We successfully implemented the algorithm with MPI.

The same setup is used from the previous exercise: Matrix value initialization, Grid size, sub-communicator. The different comes from the algorithm itself, when we have to do a pre-skewing and post-skewing stage before and after the loop run.

There are optimization we did in this exercise, we created more local variables to store values of matrix A and B during the shifting. Because of that, we avoid modifying the original value of matrix A and B. Thanks to this, the post-skewing stage is unnecessary. This will save us some communication cost to restore the initial state of the matrix.

*N.B: The optimization is done in file ex2-op.c*

# 3 Data distribution

For this exercises, we tried to utilized the vector datatype of MPI by define a typeMatCol, and type MatRow. While doing the Scatterv function, we stuck dealing with distributing the MatB while MatA is run perfectly.

The only thing, we noticed is that the blocks of value that got problem are all supposed to have the same data. We have that problem and don't know how to fix it. We did try to used both dynamic allocation and static allocation.

# 4 Performance Evaluation

We did some performance test with Fox Algorithm, Cannon Algorithm, also with a sequential code (written by ourselves)

```
Performance counter stats for 'mpirun -n 64 ./ex1':

     52160.572306      task-clock (msec)         #   23.311 CPUs utilized
          239,729      context-switches          #    0.005 M/sec
           53,888      cpu-migrations            #    0.001 M/sec
          263,539      page-faults               #    0.005 M/sec
  144,663,274,711      cycles                    #    2.773 GHz                      (50.95%)
   41,422,459,797      instructions              #    0.29  insn per cycle           (63.41%)
    9,143,703,012      branches                  #  175.299 M/sec                    (63.03%)
       85,878,529      branch-misses             #    0.94% of all branches          (62.67%)
   11,816,350,157      L1-dcache-loads           #  226.538 M/sec                    (62.30%)
      251,219,612      L1-dcache-load-misses     #    2.13% of all L1-dcache hits    (62.32%)
       31,174,136      LLC-loads                 #    0.598 M/sec                    (50.20%)
        4,625,911      LLC-load-misses           #   14.84% of all LL-cache hits     (50.72%)

      2.237566214 seconds time elapsed
```

Figure 1: Fox algorithm with 8x8 matrix

```
Performance counter stats for 'mpirun -n 64 ./ex2':

     58236.937831      task-clock (msec)         #   24.493 CPUs utilized
          292,254      context-switches          #    0.005 M/sec
           54,002      cpu-migrations            #    0.927 K/sec
          263,832      page-faults               #    0.005 M/sec
  161,953,265,688      cycles                    #    2.781 GHz                      (50.62%)
   46,264,153,570      instructions              #    0.29  insn per cycle           (63.13%)
   10,329,897,081      branches                  #  177.377 M/sec                    (62.85%)
       90,991,829      branch-misses             #    0.88% of all branches          (62.73%)
   13,356,160,516      L1-dcache-loads           #  229.342 M/sec                    (62.54%)
      280,486,560      L1-dcache-load-misses     #    2.10% of all L1-dcache hits    (62.42%)
       32,486,627      LLC-loads                 #    0.558 M/sec                    (50.18%)
        5,067,407      LLC-load-misses           #   15.60% of all LL-cache hits     (50.46%)

      2.377730854 seconds time elapsed
```

Figure 2: Cannon algorithm with 8x8 matrix

```
Performance counter stats for 'mpirun -n 64 ./ex2-op':

     56279.062778      task-clock (msec)         #   23.791 CPUs utilized
          318,358      context-switches          #    0.006 M/sec
           55,810      cpu-migrations            #    0.992 K/sec
          263,474      page-faults               #    0.005 M/sec
  156,844,996,423      cycles                    #    2.787 GHz                      (50.27%)
   42,703,411,774      instructions              #    0.27  insn per cycle           (62.96%)
    9,430,690,073      branches                  #  167.570 M/sec                    (62.93%)
       92,212,928      branch-misses             #    0.98% of all branches          (62.90%)
   12,310,408,203      L1-dcache-loads           #  218.739 M/sec                    (62.80%)
      304,046,317      L1-dcache-load-misses     #    2.47% of all L1-dcache hits    (62.81%)
       29,167,822      LLC-loads                 #    0.518 M/sec                    (49.96%)
        4,780,283      LLC-load-misses           #   16.39% of all LL-cache hits     (50.28%)

      2.365564677 seconds time elapsed
```

Figure 3: Cannon algorithm with 8x8 matrix (optimized version)

```
Performance counter stats for 'mpirun -n 64 ./seq':

       175.417895      task-clock (msec)         #    0.362 CPUs utilized
              409      context-switches          #    0.002 M/sec
              134      cpu-migrations            #    0.764 K/sec
           19,608      page-faults               #    0.112 M/sec
      533,461,438      cycles                    #    3.041 GHz                      (52.08%)
      604,911,838      instructions              #    1.13  insn per cycle           (68.31%)
      135,044,141      branches                  #  769.842 M/sec                    (69.04%)
        2,772,900      branch-misses             #    2.05% of all branches          (79.44%)
      171,242,657      L1-dcache-loads           #  976.198 M/sec                    (76.67%)
       11,774,676      L1-dcache-load-misses     #    6.88% of all L1-dcache hits    (71.33%)
        1,414,084      LLC-loads                 #    8.061 M/sec                    (54.93%)
          238,117      LLC-load-misses           #   16.84% of all LL-cache hits     (48.05%)

      0.484986967 seconds time elapsed
```

Figure 4: Sequential with 8x8 matrix

We realized that our implementation has a very simple local computation as 1 value is hold in one process. This make the sequential time so small as they don't have communication cost.

There is one remark is our Canon optimized version actually run faster than the normal implementation. But since the size of the matrix that we test is not very big, the amount is not much.