Homework 3

Hai Zhu

due date: October 11, 2017

Programming Assignment

Using flux to run an efficient MPI program to use p cores to do a template calculation on an $n \times n$ matrix $A(0 : n-1, 0 : n-1)$ where the entries are long integers. Do this for $n = 1000$ and $n = 4000$ for $p = 4, 16,$ and 36. The same program must be used for all of the runs, and should work no matter what the entries of A are nor what the function f is, and no matter what $p, n > 1$ are, for p a perfect square such that $p \leq n/2$ (in which case there might be some cores don't run at all).

Procedure: For each combination of n and p:

1. Initialize A using the definition below.
2. Use `MPI_BARRIER`, then have the root process (process 0) call `MPI_WTIME`.
3. Do 10 iterations, defined below.
4. Compute the verification values (see below) and send them to the root process.
5. Have the root process call `MPI_WTIME`.
6. Print out the elapsed time and the verification values.

To initialize A: for all $0 \leq i, j \leq n-1$, $A(i, j) = i + j * n$

Each entry of A can start on whatever processor you choose, but no entry of A can start on more than one processor.

Iterative step: In each iteration, let A_0 denote the previous value of A. Then for all $0 \leq i, j \leq n-1$, the new value of $A(i, j)$ is given by:

- $A(i, j) = A_0(i, j)$ if ($i = n-1$ or 0) or ($j = n-1$ or 0), i.e., it is unchanged along the border of the matrix
- Otherwise, $A(i, j) = f(A_0(i, j), A_0(i+1, j), A_0(i, j+1), A_0(i+1, j+1))$, where f is a function that we will give you.

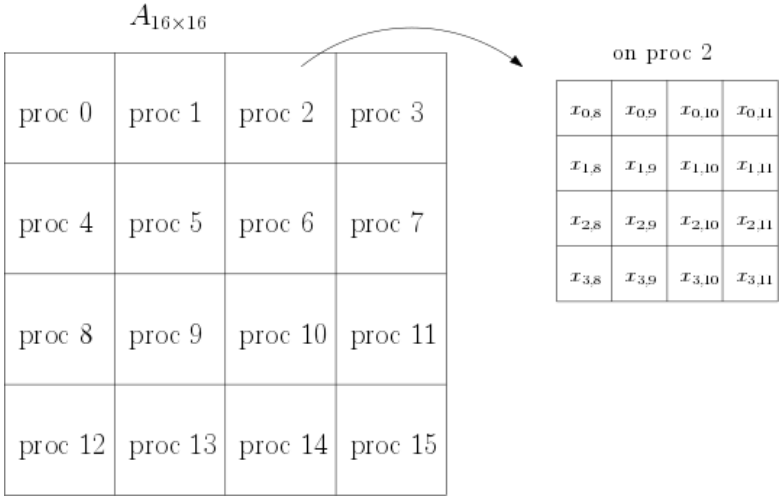
To verify a run: Print the sum of all of the entries of A after the final iteration, and the value of $A(\lfloor n/2 \rfloor, \lfloor n/2 \rfloor)$.

C++ MPI Implementation

1. **decompose matrix**

   In this problem, we want to calculate each entry in an $n \times n$ matrix A iteratively by using p cores. It is intuitive to consider decompose larger matrix into its sub-blocks. And therefore, for p cores, where p is a perfect square, we can decompose matrix A in a way that each core takes care of a sub-block of the original matrix. There won't be any communication in initialization step, we just need to calculate global index for each entry in the sub-block based on the index of processor.

   So for square $n \times n$ matrix A, we are zooming out ,and looking at a coarse grid of $\sqrt{p} \times \sqrt{p}$. And each grid (processor) is only responsible for a roughly size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sub matrix. For example, the following graph gives an illustration of the decomposition for a 16×16 matrix on a 4×4 grid parallel system,



   The rest of the work is to figure out which proc is taking care of which entry, and we need to take care of the index change from matrix A to its submatrix, and vice versa.

```
1   //index
2   int idx = floor(procID/sqrt(p));     //the row current proc is in the ...
        proc grid
3   int idy = procID - sqrt(p)*idx;      //the column current proc is in ...
        the grid
4   int nLocal = ceil(n/sqrt(p));        //submatrix size(roughly, not ...
        all are squares)
5   ...                                  //last row, last column modification
6
7   for(int i=0; i<nLocalx; i++){
8     matrixALocal[i] = new long long[nLocaly+1];
9     for(int j=0; j<nLocaly; j++){
10      iGlobal = idx*nLocal+i;          //global row index
11      jGlobal = idy*nLocal+j;          //global column index
12      matrixALocal[i][j] = iGlobal + jGlobal*n;
13    }
14    matrixALocal[nLocalx] = new long long[nLocaly+1];
15  }
```

2. **communication**

   Initialization step doesn't require any communication, as we have illustrated in the previous decomposing matrix step. And at each iteration, each entry gets updated based on its previous lower, right, and lower right entry values. And this turns into the communication between processors, each processor needs to receive value from lower, right, and lower right processor, before updating its own submatrix.

   In my implementation, I declared a 2d array of size $(\text{nLocalx}+1)\times(\text{nLocaly}+1)$. And the reason it has 1 row and column larger than it is supposed to be is because this could give us a uniform way of calculating each entry. And that only one 2d array is required is because the way each entry gets updated. If we go from left to right, up and down, once an entry gets its new value in this iteration step, it will never be used again. So we only need to ask for space of one 2d array.

   Also two 1d array and one integer to store values to be sent and received. In `MPI_Send` process, it works as illustrated below. The upper most row and left most column gets copied to `aRow` and `aClo`, and $A_{0,0}$ gets copied to `aEntry`. And in `MPI_Recv` process, `aRow`, `aCol`, and `aEntry` get copied to the ghost column and row in the submatrix, and wait for being used in current iteration. It is possible to get this work with just one array. But with two arrays, life is easier.

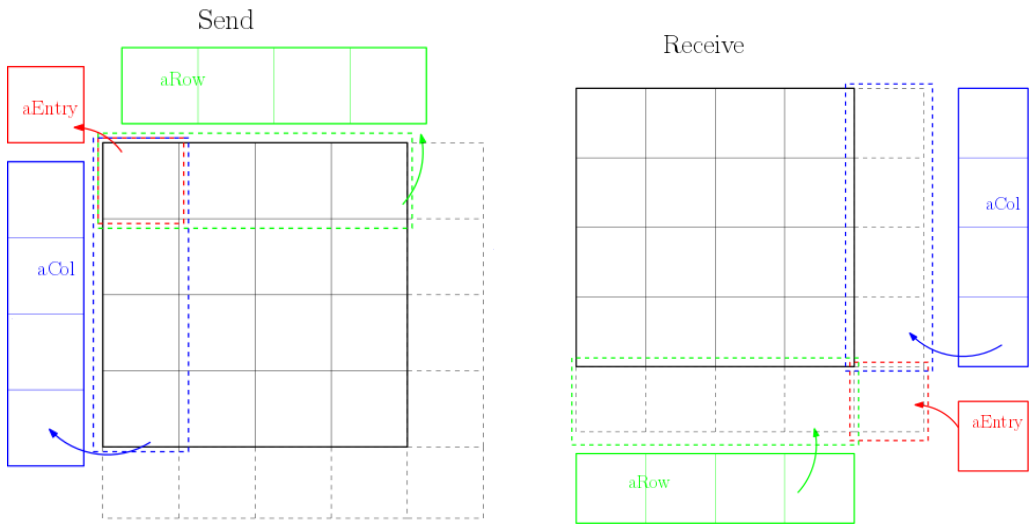


Figure 1: Send and Receive Process

The following contains majority of the communication and update implementation. There are some tweaks that are not included just to make the illustration of MPI send, receive, and updating steps clearer. (I use nonblocking send, and blocking receive.)

```
1   //declare submatrix, row, and column vector
2   long long** matrixALocal = new long long*[nLocalx+1];
3   long long* aCol = new long long[nLocalx]();
4   long long* aRow = new long long[nLocaly]();
5   long long aEntry = 0;
6
```

```
7    ...                              //assignment of aCol, aRow, aEntry, ...
         and Alocal
8
9    //10 iteration
10   for(int iter=0; iter<10; iter++){
11
12     //MPI_Send
13     if( not 1st row ){
14       ierr = MPI_Isend(aRow to proc 1 row above);
15       if( not 1st column ){
16         ierr = MPI_Isend(aEntry to proc upper left);
17       }
18     }
19     if( not 1st column ){
20       ierr = MPI_Isend(aCol to proc 1 column left);
21     }
22
23     //MPI_Recv
24     if( not last row ){
25       ierr = MPI_Recv(aRow from proc 1 row below);
26       if( not last column ){
27         ierr = MPI_Recv(aEntry from proc lower right);
28       }
29     }
30     if( not last column ){
31       ierr = MPI_Recv(aCol from proc 1 column right);
32     }
33
34     MPI_Barrier(MPI_COMM_WORLD);
35     ...                            //assign aRow, aCol, aEntry to Alocal
36
37     //update Alocal
38     for(int i=idxILow; i<idxIHi; i++){
39       for(int j=idyJLow; j<idyJHi; j++){
40         matrixALocal[i][j] = f( matrixALocal[i][j], ...
               matrixALocal[i+1][j], matrixALocal[i][j+1], ...
               matrixALocal[i+1][j+1]);
41       }
42     }
```

3. **flux, and result**
   The module I first used are

```
1) gcc/5.4.0                    4) mkl/11.3.3
2) openmpi/1.10.2/gcc/5.4.0     5) gromacs/5.1.2/openmpi/1.10.2/gcc/5.4.0
3) boost/1.61.0
```

   The time result is not that good. The longest run time, 4 processors, matrix size 4000×4000 took about 4 minutes or so. And then I used intel's compiler

```
1) intel/17.0.1   2) openmpi/1.10.2/intel/17.0.1
```

   The run time seems to be better. The actual code also gots tweaked a bit every now and then. And the following is one of most recent run time result returned on flux,

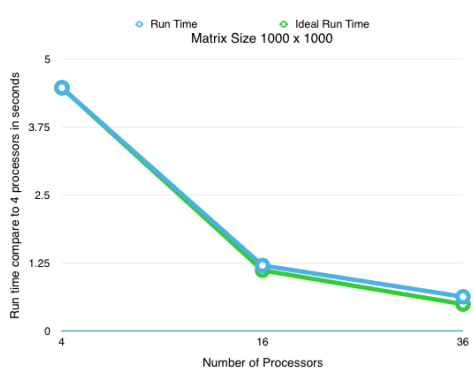
```
================================================================================
  Matrix A size is 1000 X 1000 running on 4 processors.
  Elapsed wall clock time = 4.47497 seconds.
  A[n/2][n/2] value is A[500,500]=1 from proc 3.
  All sum is 3193342943
================================================================================
  Matrix A size is 1000 X 1000 running on 16 processors.
  Elapsed wall clock time = 1.20926 seconds.
  A[n/2][n/2] value is A[500,500]=1 from proc 10.
  All sum is 3193342943
================================================================================
  Matrix A size is 1000 X 1000 running on 36 processors.
  Elapsed wall clock time = 0.633271 seconds.
  A[n/2][n/2] value is A[500,500]=1 from proc 14.
  All sum is 3193342943
================================================================================
  Matrix A size is 4000 X 4000 running on 4 processors.
  Elapsed wall clock time = 70.846 seconds.
  A[n/2][n/2] value is A[2000,2000]=1 from proc 3.
  All sum is 146653871345
================================================================================
  Matrix A size is 4000 X 4000 running on 16 processors.
  Elapsed wall clock time = 17.8065 seconds.
  A[n/2][n/2] value is A[2000,2000]=1 from proc 10.
  All sum is 146653871345
================================================================================
  Matrix A size is 4000 X 4000 running on 36 processors.
  Elapsed wall clock time = 9.90452 seconds.
  A[n/2][n/2] value is A[2000,2000]=1 from proc 14.
  All sum is 146653871345
```

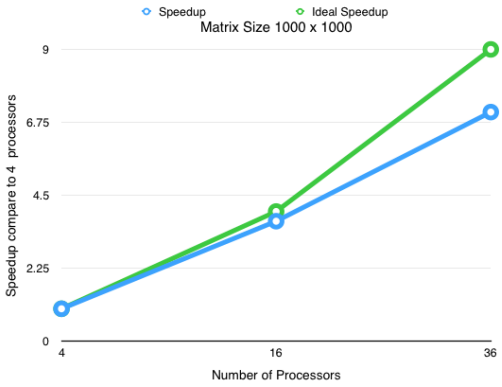
These results can be obtained by submitting `.pbs` file like the following

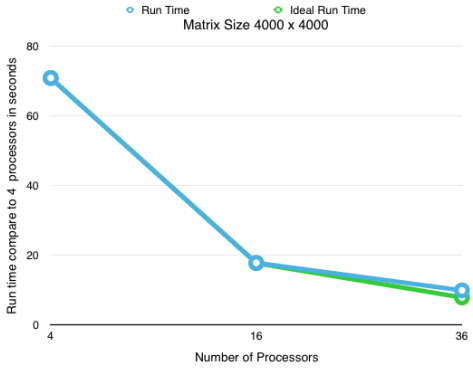
```
#!/bin/sh
#PBS -S /bin/sh
#PBS -N hw3_hszhu
#PBS -A eecs587-f17_flux
#PBS -l qos=flux
#PBS -l procs=36,mem=1gb,walltime=5:00
#PBS -q flux
#PBS -M hszhu@umich.edu
#PBS -m abe
#PBS -j oe
#PBS -V
# Let PBS handle your output
```

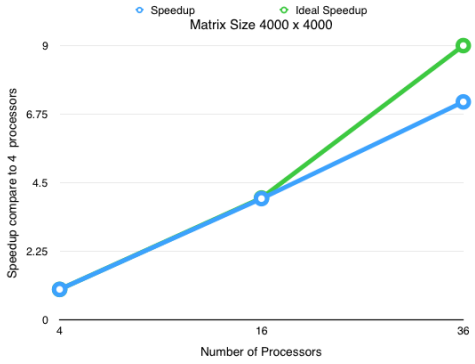
```
cd ~/EECS587/hw3
mpirun -np 4 ./hw3_hszhu 1000
mpirun -np 16 ./hw3_hszhu 1000
mpirun -np 36 ./hw3_hszhu 1000
mpirun -np 4 ./hw3_hszhu 4000
mpirun -np 16 ./hw3_hszhu 4000
mpirun -np 36 ./hw3_hszhu 4000
```

Graphically, the speed up plot looks like, (all compared to 4 processors)









The sums all checked out. And I also have a serial version code written firstly just to verify the parallel code to working correctly. I didn't get perfect speed up. And I think there are a few reasons that cause this. First of all, the message passing would cause overhead. And also the total number of boundary entries need to be sent and received increases as we increase the number of processors. So the communication time won't scale as for loop. Secondly, outside the for loop, there are also some extra computations to do copy work, or figure out index. These operations use constant amount of time regardless of number of processors. And thirdly, I am asking for a subblock that is one row and column larger than it needs to be, just to make sure that writing the computation of each entry is consistent, so that we don't need to break it into different cases. That might not be a good idea.

These 3 sources of extra computation are the reasons that it doesn't scale well when you use more and more processors. And this can be confirmed by modifying the code a little bit. Just

comment out the for loop, and let the code run. The resulting time will shows how these 3 sources scale. And the results are as follows,

```
For matrix size 1000 x 1000:
        proc num                4               16               36
        run time(s)        0.0270119         0.072226         0.257634
For matrix size 4000 x 4000:
        proc num                4               16               36
        run time(s)        0.0272171        0.0662482         0.315714
```

These are the extra amount of computation that run besides the computation of f function inside the for loop. And we can tell from the above results that they are actually increasing as we use more and more cores.