ECE 5720 HW3

Chen-Tung Chu cc2396

1. General Remarks

In this assignment, our goal is to compute the SVD of a matrix by implementing one sided Jacobi. For an $m \times n$ matrix A, we want to find three orthogonal matrices: U, V, and Σ such that $AV = U\Sigma$. The first thing we have to do is read through the Jacobi method and try to understand the serial processing code that the professor had given. However, it was in MATLAB code, and MATLAB is a very powerful programming in matrix computation. Our first step is to transfer the Jacobi algorithm from MATLAB code into C code, and check whether our Σ matrix be the same as in the MATLAB code for justification and correctness of our code. The next and final step is to implement OpenMPI with our code for parallel processing. I will explain more detailly on the serial processing section since the implementation of OpenMPI comprises mostly the code from the serial one.

2. Serial Processing (Transfer MATLAB code)

To mimic the setup and get the correct Σ, I also set both row and column to 64 for matrix A, and I defined a parameter to switch whether I want to read the data from `MyMatrix.txt` or I simply want to generate the matrix randomly. For `MyMatrix.txt`, I first imported the data from matrix A as the values for justification. For the randomly matrix, for a $ROW \times COL$ matrix, I generated the data by using the equation `A.data[i][j]=i*A.row+j`. In other words, each row will have the values range from $ROW \times (i-1), 0 \leq i \leq COL$. Figure 1 is my setup structure for matrix in order for convenience. Figure 2 shows the functions I used for this one-sided Jacobi implementation. Figure 3 shows the code of the initialization of matrix A. Figure 4 shows the example of a randomly generated matrix with 16*16 dimension (I neglected the floating points for simplicity).

```
// Define matrix structure
typedef struct {
  int row;
  int col;
  float **data;
} matrix;
```

Figure 1. Definition of a matrix structure

```
25  matrix matrixInit(int row, int col);
26  void matrixFree(matrix matA);
27  matrix matrixCopy(matrix matA);
28  matrix matrixMult(matrix matA, matrix matB);
29  matrix matrixTrans(matrix matA);
30  matrix twoSubmatrix(matrix matA, int col0, int col1);
31  void DispMatrix(matrix matA);
32  float* readMatrix(const char *filename, int row, int col);
33  void replace(matrix A, matrix sub, int lower, int step, int upper);
34  matrix submatrix(matrix mtx, int lower, int step, int upper);
35  matrix odd_even_sub(matrix mtx, int start);
36  void sort(float arr[], int n);
```

Figure 2. Functions used in the program

Figure 2 shows the functions I implemented in the program. I will simply describe the functionality of each function as follows:

- `matrixInit`: Initialize a matrix, the return value will be a `row*col` identity matrix.
- `matrixFree`: Free the memory address of the matrix.
- `matrixCopy`: Duplicate the input matrix.
- `matrixMult`: Compute the matrix multiplication and return the result matrix.
- `matrixTrans`: Return the transpose matrix of the input matrix.
- `matrixtwoSubmatrix`: Return the `row*2` submatrix of the input matrix.
- `matrixTrans`: Return the transpose matrix of the input matrix.
- `DispMatrix`: Print out the matrix.
- `readMatrix`: Read the file from the input file path name, then return a float array with data from the file.
- `replace`: Update (replace) the values of the target matrix from the other matrix, with lower bound and upper bound for the boundary of the target matrix. I also set up a step argument for convenience.
- `submatrix`: Return the submatrix of the input matrix.
- `odd_even_sub`: Return the even/odd column submatrix of the input matrix.
- `sort`: Sort the input array, here I use bubble sort.

```
73    if(read_matrix == 1){
74      float *A_temp = readmatrix("MyMatrix.txt", ROW, COL);
75      int kcount = 0;
76      for(int i = 0; i < ROW; i++){
77        for(int j = 0; j < COL; j++){
78          A.data[i][j] = A_temp[kcount++];
79        }
80      }
81    }
82    else{
83      for (i = 0; i < A.row; i++){
84        for(j = 0; j < A.col; j++){
85          A.data[i][j] = i * A.row + j;
86        }
87      }
88    }
```

Figure 3. Initialization of matirx A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

*Figure 4. Generate a 16*16 matrix*

The next step is to setup Σ, U, threshold and sum. For simplicity, I directly used the result from MATLAB as my threshold value. Later, we come into our main loop, I set up the default value of `maxsweep` to 100 as my stop point of the main loop. We use Brent-Luk annihilation ordering, and I used `twoSubmatrix` function to get the 2×2 submatrix of $A * A'$, in other words, we want the same result from both codes. Figure 5 is the C code I used in comparison of the MATLAB code. In my code, `a_temp` is equal to a in MATLAB.

```
S_twocol = twoSubmatrix(S, j, k);
V_twocol = twoSubmatrix(V, j, k);
ST_twocol = matrixTrans(S_twocol);
a_temp = matrixMult(ST_twocol, S_twocol);
```

```
% get n/2 working 2 by 2 submatrices of A'*A
      a = A(:,[2*k-1 2*k])'*A(:,[2*k-1 2*k]);
```

*Figure 5. Codes to generate 2*2 submatrices*

The next step we can compute `sum` and `tau`, the operation is pretty much the same, as well as getting the value of tangent, cosine, and sin. After we get these values, we can start to rotate the columns of A. In MATLAB, it combined the values of $[cos,\ sin, -sin, cos]$ into a matrix G and replace the value of the submatrix of A via matrix multiplication. Since the matrix is relatively small, I used a for loop to calculate the matrix multiplication of the G matrix in MATLAB, same operation in matrix V. Figure 6 shows the same result operation of replacing the values of A and V (Note: I used the matrix S as a copy of A)

```
for (p = 0; p < S.row; p++)
{
  S.data[p][j] = S_twocol.data[p][0]*c - S_twocol.data[p][1]*s;
  S.data[p][k] = S_twocol.data[p][0]*s + S_twocol.data[p][1]*c;
}
```

```
A(:, [2*k-1 2*k]) = A(:, [2*k-1 2*k])*G;
```

Figure 6. Rotate columns of matrix A

After the rotation (k sweeps), we got the matrices $A^{(k)}$ and $V^{(k)}$ and we need to get $A^{(k+1)}$ and $V^{(k+1)}$. For that we compute another sweep composed of compound rotations. We started to permute columns of A and V. One thing to remember is that MATLAB counts from 1 instead of 0, so when we read the code from MATLAB, especially when facing odd and even columns permutation. Implementing matrix computation in C is relatively harder than in MATLAB, so I came up with three auxiliary functions to help me with the one-liner code in MATLAB. First, like the initialization of odd and even matrices in MATLAB, I wrote a function called `odd_even_sub` to build the odd/even submatrices in A, there are two parameters for the input of the function, the first parameter is the original matrix we want to make into odd or even submatrix; and the second parameter is the integer that determines the submatrix I want to extract. If the input integer is odd, then the function will return the even submatrix; vice versa, it will return the odd submatrix from A if the input parameter is an even number. Mind that in C, it starts with index 0, so we have to do the opposite operation in MATLAB. Figure 7 is the function description of `odd_even_sub`.

```
matrix odd_even_sub(matrix mtx, int start){
  int col;
  if(start % 2 == 0){
    col = (mtx.col - start)/2;
  }
  else{
    col = mtx.col/2;
  }
  matrix res = matrixInit(mtx.row, col);
  int k = 0;
  for(int j = start; j < mtx.col; j+=2){
    for(int i = 0; i < res.row; i++){
      res.data[i][k] = mtx.data[i][j];
    }
    k++;
  }
  return res;
}
```

Figure 7. odd_even_sub function

After getting the even/odd submatrix, the first thing we do is to permute the left and right boundary of matrix *A*. Once we changed the values of *A*, we can permute the other values with odd or even submatrices. For clean code and convenience, I also wrote two functions called `replace` and `submatrix`. The `replace` function is to replace the current matrix's values with another matrix. The `submatrix` is just as the name of the function, it returns the submatrix we want to extract, and I also set up the "step" parameter to deal with the submatrix in MATLAB like `A(:, 5:2:n-1)`. Figure 8 is the codes for `replace` and `submatrix`.

```
void replace(matrix A, matrix sub, int lower, int step, int upper){
  for(int j = lower, k = 0; j <= upper; j+= step, k++){
    for(int i = 0; i < sub.row; i++){
      A.data[i][j] = sub.data[i][k];
    }
  }
}

matrix submatrix(matrix mtx, int lower, int step, int upper){
  matrix res = matrixInit(mtx.row, (upper - lower + 1)/ step);

  for (int j = lower, k = 0; j <= upper; j += step, k++)
  {
    for (int i = 0; i < res.row ; i++)
    {
      res.data[i][k] = mtx.data[i][j];
    }
  }
  return res;
}
```

Figure 8. replace & submatrix function

Same operation with matrix V, we execute the same code again but change the matrix. After the loop, we finished the permutations of the columns of matrices A and V (In my code, I use S to represent A in MATLAB. Figure 9 shows the whole code for column permutation in my C implementation and the original MATLAB version.

```
matrix temp_even = odd_even_sub(S, 1);
matrix temp_odd = odd_even_sub(S, 2);

replace(S, submatrix(S, 1, 1, 1), 2, 1, 2);
replace(S, submatrix(S, COL-2, 1, COL-1), COL-1, 1, COL-1);

replace(S, submatrix(temp_odd, 0, 1, temp_odd.col -2), 4, 2, COL-2);
replace(S, submatrix(temp_even, 1, 1, temp_even.col - 1), 1, 2, COL-3);

temp_even = odd_even_sub(V, 1);
temp_odd = odd_even_sub(V, 2);

replace(V, submatrix(V, 1, 1, 1), 2, 1, 2);
replace(V, submatrix(V, COL-2, 1, COL-2), COL-1, 1, COL-1);

replace(V, submatrix(temp_odd, 0, 1, temp_odd.col -2), 4, 2, COL-2);
replace(V, submatrix(temp_even, 1, 1, temp_even.col - 1), 1, 2, COL-3);
```

```
temp_even = A(:,2:2:end);
temp_odd = A(:,3:2:end);
A(:,3) = A(:,2);                    % this is the "left" boundary
A(:,n) = A(:,n-1);                  % this is the "right" boundary
A(:,5:2:n-1) = temp_odd(:,1:end-1);
A(:,2:2:n-2) = temp_even(:,2:end);

temp_even = V(:,2:2:end);
temp_odd = V(:,3:2:end);
V(:,3) = V(:,2);
V(:,n) = V(:,n-1);
V(:,5:2:n-1) = temp_odd(:,1:end-1);
V(:,2:2:n-2) = temp_even(:,2:end);
```

Figure 9. Codes for columns permutation

After we finished the sweep in the main loop, we now have the orthogonal matrix of the original matrix A, which is S ,and $S = U * \Sigma$, now we perform the extraction of U and sigma, which is the diagonal matrix of Σ. Since the major goal of this serial process is to make sure my one-sided Jacobi algorithm is correct, and the main function is to check the matrix Σ is the same as in the MATLAB. So, I only checked for the matrix Σ. The matrix sigma is shown in MATLAB as `sigma=sqrt(diag(A'*A))`, I first multiplied the matrices S and S' and get the matrix `sig_temp`, then I perform two simple for loops to grab the square root values in `sig_temp` and stored it into a float array `sg`. The next step is to sort the array in descending order via a simple bubble sort function I created. And for the matrix Σ, we just simply create a matrix in $ROW \times 1$ dimension and stored with the values in the sorted array `sg`. Now we get the complete SVD matrix Σ, and we can print out the eight largest singular values.

3. Results for serial processing

With the completion of transferring the MATLAB code into C code, now we can examine whether my one-sided algorithm is correct or not. I extracted the matrix A's values and stored it into a text file called `MyMatrix.txt`. I set the `read_matrix` value to 1 so it can read the values from the text file. After executing my code, the result of eight largest singular values in matrix Σ and the comparison with `Sigma` in MATLAB are shown in Figure 10.

```
The eight largest singular values are:
64.0000
63.0000
62.0000
61.0000
60.0000
59.0000
58.0000
57.0000
(base) chentungchu@dhcp-vl2042-8288 hw3 %
```

Sigma

64×1 double

	1
1	64.0000
2	63.0000
3	62.0000
4	61.0000
5	60.0000
6	59.0000
7	58.0000
8	57.0000

Figure 10. Result of matrix Sigma

We can find that the result of my code and the matrix Sigma in MATLAB are exactly the same (here I set the datatype to `double`, but for parallel processing I will set to `MPI_FLOAT`), thus I can assure that the one-sided Jacobi algorithm I wrote is correct. And now I can move forward to implement my Jacobi algorithm finding SVD of a matrix with OpenMPI.

4. Implementation with OpenMPI

With OpenMPI, we have to set up some initialization for the MPI environment. I set up four MPI status for my status and request initialization. After that, the process is the same as the serial processing one, first to get the matrix A, the copy of A called S, and the matrix V. Then I start the iteration in main loop to perform full sweep. The main difference is that implementing OpenMPI requires buffers, and after setting up the buffers, we can perform different situations for different rounds and ranks. For each situation, we have to check the rank of every PEs and the conditions of the rank for column permutation. Once we finish the permutations and value replacements from right to last row, and we checked the MPI barriers, after it meets the criteria, we can terminate the MPI and to calculate the Matrix Σ and V.

I started testing the code with a 64×64 matrix generated with number $i * 64 + j$, where $0 \leq i < 64,\ 0 \leq j < 64$, Figure 11 shows the result of different ranks out of PEs (in here I set 12 PEs).

```
Spent 59.51377 msec to find the SVD, rank 4 out of 12 PEs
Spent 51.64647 msec to find the SVD, rank 6 out of 12 PEs
Spent 59.65105 msec to find the SVD, rank 10 out of 12 PEs
Spent 55.58344 msec to find the SVD, rank 0 out of 12 PEs
Spent 55.65985 msec to find the SVD, rank 8 out of 12 PEs
Spent 51.59059 msec to find the SVD, rank 2 out of 12 PEs
Spent 47.75801 msec to find the SVD, rank 3 out of 12 PEs
Spent 55.74197 msec to find the SVD, rank 5 out of 12 PEs
Spent 47.76083 msec to find the SVD, rank 9 out of 12 PEs
Spent 43.78613 msec to find the SVD, rank 7 out of 12 PEs
Spent 51.74760 msec to find the SVD, rank 1 out of 12 PEs
Spent 43.78954 msec to find the SVD, rank 11 out of 12 PEs
```

Figure 11. Different ranks processing in 12 PEs

I also experimented the four different dimension setups in matrix A with 12 PEs. Table 1 is the execution time difference in milliseconds between serial processing and OpenMPI.

Matrix Size / Process Type	32*32	64*64	128*128	256*256	512*512	1024*1024
Serial	198.679	1382.849	11391.888	92617.539	MLE	MLE
OpenMPI	1.000	235.923	3508.448	34540.606	647416.874	--

Table 1. Time difference between serial and OpenMPI (ms)

From the table above we can see the significant execution time difference between serial processing and OpenMPI parallel computation. With 12 PEs separating the matrix computation, the average compute dime drastically diminished. Take 256*256 matrix for example, implementing Jacobi algorithm with OpenMPI with 12 processors is approximately x2.68 faster than conventional serial processing. For further experiment, I tried to expand more on the dimension of the matrix on both methods. As I experimented with 512*512 matrix, both my computer and the Ubuntu virtual machine could not compute the matrix with such high dimension. Turned out that the serial computing method may require memory capacity for over 32 GB. Perhaps it needs more memory to compute the matrix with such high dimension. For the parallel implementation with OpenMPI, we can find out despite such high dimension of a matrix, it can still compute the SVD successfully.

With the OpenMPI implementation, I set up the statements of the non-blocking communication to check rank and iteration conditions for synchronization, if both the rank and the iteration number is zero, then we have to wait for all the process to finish to

receive the overlapped matrix computation (working with 2*2 submatrix of matrix $A * A'$). Figure 12 shows the statements of different situations for different rounds and ranks.

```
156 >   if(rank==0 && itr==0)
171
172 >   else if (rank==0 && itr!=0)
197
198 >   else if (rank!=0 && rank!=(npes-1))
223
224 >   else if (rank==(npes-1) && itr!=(round-1))
249
250 >   else if (rank==(npes-1) && itr==(round-1))
```

Figure 12. Statements for different rounds and ranks

The later process is the same as in the serial program, but with two rounds of conditional statements. After the main loop, we also perform the same process in the serial code.

5. Test Instructions

For the files in the `cc2396.zip`, to run `cc2396_hw3.c` on SURM, we need two files, `cc2396_hw3.sub` and `cc2396_hw3.sh`. For `cc2396_hw3.sub`, I set my job name as cc2396 and I requested 6, which is the maximum server of the virtual machine, and 12 tasks for my configuration. The path I set was the testing path on my Ubuntu VM. Figure 13 shows the code and path setup of `cc2396_hw3.sub`. For `cc2396_hw3.sh`, I simply changed the execution file. Figure 14 shows the code.

```
1   #!/bin/bash
2   #SBATCH -J cc2396                   # Job name
3   #SBATCH -o output/cc2396.o%j        # Name of stdout output file(%j expands to jobId)
4   #SBATCH -e output/cc2396.e%j        # Name of stderr output file(%j expands to jobId)
5   #SBATCH --nodes=6                   # Total number of nodes requested
6   #SBATCH --ntasks=12                 # Total number of tasks to be configured for.
7   #SBATCH --tasks-per-node=2          # sets number of tasks to run on each node.
8   #SBATCH --cpus-per-task=1           # number of cpus needed by each task
9                                       # (if task is "make -j3" number should be 3).
10  #SBATCH --get-user-env              # tells sbatch to retrieve the users login environment.
11  #SBATCH -t 00:10:00                 # Run time (hh:mm:ss)
12  #SBATCH --mem-per-cpu=1000          # memory required per allocated CPU
13  #SBATCH --mem=1000M                 # memory required per node
14  #SBATCH --partition=six             # Which queue it should run on.
15
16  ./cc2396_hw3.sh
17  # cd /home/cc2396/hw3/output; ./cc2396_hw3.sh
```

Figure 13. cc2396_hw3.sub code

```
1   #!/bin/bash
2
3   # module load openmpi-4.0.1
4   # mpirun -np 12 /bin/hostname
5   mpirun -np 12 ./cc2396 --mca opal_warn_on_missing_libcuda 0
```

Figure 14. cc2396_hw3.sh code

As for testing part, for those who want to test on their own machine may need to change the path directory in order to let OpenMPI to compile successfully and run the executable. I also added the serial main code at the end of the OpenMPI code for further justification and comparison.

6. Conclusion

From the experiments above, we can understand that despite a serial approach may still compute Jacobi algorithm to find the SVD. However, as the matrix size grows larger, the serial process causes larger acquirement for memory usage. Eventually, the machine cannot handle the use of memory. In my experiment, when the matrix size came to 512*512, the 32GB-RAM computer could saturate the memory. Thus, we implement the same algorithm with OpenMPI approach. We can see that from the statistics, with OpenMPI parallel method, it was way faster than the serial one, it is obviously that with a number of processors compute the large matrix, with numerous tasks. This simply points out the importance of parallel computing. It can break through the limitation of serial processing. But we need to set up some statements for the internode communication and synchronization in the paralleling process. In conclusion, the optimal way for this kind of matrix computation is parallel processing.

7. Reference

[1] https://github.com/Amoiensis/Matrix_hub/tree/master/code_src/Matrix_Hub_v1.43
[2] https://github.com/Pruthvish-E/c-matrix-library
[3] https://www.mathworks.com/help/matlab/ref/norm.html
[4] https://www.mathworks.com/help/matlab/ref/qr.html
[5] http://www.netlib.org/utk/papers/mpi-book/node44.html
[6] Course lecture