High Performance Computing Project Report: Implementation of Fast Biclustering Algorithm

Simeng Kuang

December 20, 2012

Abstract

In this report, we mainly described a biclustering algorithm and its parallel implementation. This algorithm, which efficiently detects low rank submatrix within a much larger matrix, is discussed both from the theoretical intuition aspect and its implementation details. Various numerical results were presented to demonstrate the efficiency of our implementation.

1 Introduction

Aiming at detecting low rank submatrix within a much larger matrix, biclustering algorithm is highly related to various techniques in data analysis[1, 2, 3, 4] . In the paper of Rangan[5], a fast biclustering algorithm that relies on a basic geometric property of high-dimensional space were presented. According to a scoring system which based on counting the number of rank-1 2×2 submatrices, the algorithm removes rows and columns that are unlikely to be in the low rank submatrix thus find the low rank one.
There are also other attempts to finding certain kinds of low-rank submatrices, such as nuclear norm minimization [6], adaptive dissection of projective space[7] and spectral biclustering[8].
In this report, In the first part, we constructed the algorithm from its geometrical intuition in an equivalent but slightly different way as in Rangan's paper. We gave a new form of the row and column scores that is more straightforward and easier to implement, 4 update formulas that are used in implementation were also introduced. In the second part, we focused on important implementation details such as problem scale we can work on and ways to accelerate the code. Finally, we closely studied the efficiency of our implementation by presenting various numerical experiment results on different problem scales with both single thread and multi threads.

2 Theoretical Background

Generally speaking, there is one basic problem we will consider. This problem involves finding the largest submatrix of low numerical-rank from a larger matrix. This problem (stated formally below) is often called the 'biclustering' problem in data-analysis[1, 2, 3, 4].

Problem 1. *Assume that we are given an $n \times m$ matrix A, and that an $n_C \times m_C$ submatrix C of A has low numerical-rank k. Assuming that C is the largest such submatrix within A, is it possible to find C quickly?*

In Rangan's paper, the following property of high dimensional space is essential to find a way to solve our problem.

Property 1. *Planar projections of eccentric gaussian distributions are concentrated in non-adjacent quadrants.*

Intuitively, consider a gaussian distribution with a covariance matrix that is numerically low rank(we can assume it is full rank to avoid degeneracy). Since it's numerically low rank, most of its singular values are a lot smaller than the largest few ones. The consequence of this is that if we draw its density function in R^n, it will be very 'thin'. So if we do a random planar projection, it will project the distribution to a subspace that corresponding to small singular values with high probability. The result is that the distribution will also be 'thin', like a line in R^2, in another word the distribution will concentrate in non-adjacent quadrants. A more accurate description of this property can be found in Rangan's paper[5].
With this property, we can tell how likely a row(column) is in the low rank submatrix.
Consider arbitrary 2×2 submatrix of A, we can view it as a planar projection $P(i, j)$, where i, j are the row indexes of this submatrix in A. If the 2×2 matrix is in the low rank submatrix, because of Property 1, its row vectors are more likely to stay in same quadrant or opposite quadrants,but not adjacent ones. If it is not in the low rank submatrix, the probability of staying in each quadrant will be uniform. In another word, we actually find a bias of 2×2 matrices:

Property 2. *For arbitrary row i,j and column k,l of A,consider their intersection, a 2×2 matrix, if its 2 row vectors stay in same quadrant or opposite quadrants, row i,j and column k,l are more likely to be in low rank submatrix.*

If we binarize matrix A to a binary matrix B, meaning set $b_{ij} = \text{sgn}(a_{ij})$, Property 2 is equivalent to the following property of B.

Property 3. *For arbitrary row i,j and column k,l of B,consider their intersection, a 2×2 matrix, if it is rank 1, row i,j and column k,l of A are more likely to be in low rank submatrix.*

Back to property 2, we can transform this bias of 2×2 matrices to bias of rows and columns.

Property 4. *For a given row i of B, consider all the 2×2 matrices that have row i. the more rank 1 matrices we can find within these matrices, the more likely row i is in the low rank submatrix.*

Thus we actually defined a score for every rows and columns, the next lemma tells us a straightforward form of our score.

Lemma 1. *For arbitrary rows B_i and B_j of B, define a score $S(i, j)$ to be the number of 2×2 submatrices these 2 rows intersects with all possible columns. we have:*

$$S(i, j) = < B_i, B_j >^2 \tag{1}$$

Lemma 2. *For arbitrary row B_i of B, define a score R_i to be the number of 2×2 submatrices this row intersects with all possible columns. we have:*

$$R_i = \sum_{j=1}^{n} S(i,j) = \sum_{j=1}^{n} < B_i, B_j >^2 \tag{2}$$

If lemma 1 is proved, lemma 2 is just add up all the rows, next we prove lemma 1.

Proof of lemma 1:

since B_i B_j are consist of 1 and -1, we consider vector $v = (b_{i1}b_{j1}, b_{i2}b_{j2}, \ldots, b_{im}b_{jm})$, if $v_k = 1$, it means the kth entry of row i and row j are the same, otherwise they are different. Thus 2×2 matrix is rank 1 if and only if $v_k = v_l$. So let the number of 1 in v to be n_1, the number of -1 in v to be n_2. we have:

$$\begin{aligned} n_1 + n_2 &= m & (3)\\ n_1 - n_2 &= < B_i, B_j > & (4)\\ S(i,j) &= n_1^2 + n_2^2 & (5) \end{aligned}$$

So we have

$$S(i,j) = \frac{1}{2}(< B_i, B_j >^2 + m^2) \tag{6}$$

Since we only want to compare different rows scores, we can ignore the constants and the expression, simply we define:

$$S(i,j) = < B_i, B_j >^2 \tag{7}$$

Lemma 1 is proved, so lemma 2 also holds. In summary, we have the scores for rows and columns R_i and C_j as below, we have to change our notation to make things more clear.

Theorem 1. *Let B be a $n \times m$ $\{1, -1\}$ matrix that derived from A. Let A_i be the ith row and R_i be its score. Similarly we define B_j as the jth column and C_j as its scores.*

$$\begin{aligned} R_i &= \sum_{j=1}^{n} < A_i, A_j >^2 & (8)\\ S_i &= \sum_{j=1}^{m} < B_i, B_j >^2 & (9) \end{aligned}$$

3 Algorithm Description

With scores defined in the last section, we can have simple procedure of detecting low rank submatrix.

1. Binarize matrix A to B,
2. Calculate the scores of all remaining rows and columns,
3. Remove rows or columns with lowest score, go to step 2 unless all the rows or columns were removed.

4. Check the rows and columns remained at last, we find the low rank submatrix.

Note that when a row or column is removed, the score of all the remaining rows and columns changes, so we have to recalculate scores after each remove. The naive way to do this is to calculate formulas (8) (9) directly, which is a $O(n^3)$ operation. However, we notice that only small parts in every score are changed, we can use the following update formula to recalculate all the scores.

Theorem 2. *After removing one row $a = (a_1, a_2, \cdots, a_m)$,set the original scores to be R'_i and C'_j. we find the new scores to be:*

$$\begin{aligned} R'_i = &\quad R_i - < A_i, a >^2 i & (10) \\ C'_i = &\quad C_i - m - 2 \sum_{A_k \neq a} < A_k, a > a_i a_{ki} & (11) \end{aligned}$$

If we remove one column, let it be $b = (b_1, b_2, \cdots, b_n)^T$, The new scores are

$$\begin{aligned} C'_i = &\quad C_i - < B_i, b >^2 & (12) \\ R'_i = &\quad R_i - n - 2 \sum_{B_k \neq b} b_i a_{ik} < B_k, b > & (13) \end{aligned}$$

Using these formulas instead of (8)(9) brings the complexity of recalculating scores to $O(n^2)$.

Thus a more accurate description of our algorithm is:

1. Binarize matrix A to B,
2. Initialize the scores of all rows and columns using (8) and (9)
3. Remove rows or columns with lowest score,
4. Update all the scores using (10),(11),(12) and (13),
5. If all the rows or columns are removed, go to step 6, otherwise go to step 3,
6. Check the rows and columns remained at last, we find the low rank submatrix.

Now we can head to the next implementation part.

4 Implementation

In this section, we will first relate our algorithm to its most direct application on genetic data and give a revision of the algorithm to fit the practical needs. Then we will discuss the problem scale we are working on and its relation to implementation. Finally we will discuss technical details of optimizing the performance of both sequential and parallel programs.

4.1 Genetic data and supervised implementation

Gene-expression data typically involves various gene-expression levels measured across many patients. Typically patients are classified into 'cases' and 'controls'. Moreover, there are typically many genes which are correlated across the entire population — including both cases and controls. In this case, the largest low-rank submatrix within the gene-expression data will just be the features across both cases and controls, thus helpless in finding correlation between the decease and gene. Instead, we are interested in finding low rank submatrices that is in the cases group and excluding the co-exist low rank structures among the whole population at the same time. This type of submatrix would pinpoint genes useful for diagnosis and discrimination between case and control status.
Mathematically, instead of a single matrix A, we now have two input, the $n \times m$ cases matrix A and $d \times m$ controls matrix Z. Our objective is to find a low rank submatrix in A that does not share the structure in Z. To achieve this, we simply modify our scores, but still use the procedure we described in the last section.
Intuitively, we also count the rank 1 2×2 matrices, but when we find a rank 1 2×2 in Z, we minus 1 on the score instead of add 1. As a result, equation (8)-(13) are modified into:

Theorem 3. *1. Let A be a $n \times m$ $\{1, -1\}$ matrix. Let A_i be the ith row and R_i be its score. Similarly we define B_j as the jth column and C_j as its scores.*

Let Z be a $d \times m$ $\{1, -1\}$ matrix and Z_i and K_j be rows and columns.

Define $< \cdot, \cdot >$ as the product of 2 vectors.

Then we can write scores as:

$$R_i = \sum_j < A_j, A_i >^2 - \sum_j < Z_j, A_i >^2 \tag{14}$$

$$C_i = \sum_j < B_j, B_i >^2 - \sum_j < B_j, B_i >< K_j, K_i > \tag{15}$$

2. After removing one row $a = (a_1, a_2, \cdots, a_m)$,set the scores as R'_i and C'_j. we find the updates:

$$R'_i = R_i - < A_i, a >^2 \; i \tag{16}$$

$$C'_i = C_i - m - 2 \sum_{A_k \neq a} < A_k, a > a_i a_{ki} + \sum_{Z_k} < Z_k, a > a_k z_{ki} \tag{17}$$

If we remove one column, let it be $b = (b_1, b_2, \cdots, b_n)^T$, and the corresponding column in Z is $k = (k_1, k_2, \cdots, k_d)^T$. We have the update:

$$C'_i = C_i - < B_i, b > [< B_i, b > - < K_i, k >] \tag{18}$$

$$R'_i = R_i - n + d - 2 \sum_{B_k \neq b} b_i a_{ik} [< B_k, b > - < K_k, k >] \tag{19}$$

With these new scores we defined, we can use the same procedure described in last section to analysis genetic data. Related sample results are presented in the Numerical Result section.

4.2 Problem Scales

In this section, we will discuss what are the scales of existing genetic data and what scale we can work on. How to compress and store data in real machine is also discussed.
Our algorithm is essentially a sequential algorithm, the order of removing rows and columns is irreversible. In this case, the main operation of the algorithm is the updating all the scores. This is also what we should optimize and parallelize. Since it is a $O(n^2)$ operation and during each update we only reading entries $O(1)$ times, our algorithm is essentially a memory bound problem. As a result how to store and read memory is important issue.
Firstly, the genetic data we can work on typically of three different scales.

- Cancer data. This data set's size if about $10^5 \times 10^7$, meaning there are about 10^5 patients and 10^7 gene.
- Bipolar disorder data. This data set's size if about $10^4 \times 10^6$.
- Austin spectrum disorder data. This data is 10^2*10^4.

To continue our discussion, we will first explain how we can store our data. Since we are working on $1, -1$ matrix, The cheapest way to store is 1 bit per entry. we simply map 1 to 1, -1 to 0 so the information of 1 entry is stored in 1 bit. The next step is to store these bits in given data type in C, in our implementation, you can choose data types from **bool, short, int, long**.
There are 2 advantage of this storage strategy. Firstly, instead of storing 1 entry in 1 given data type in C, for example **Bool**, it takes at least 8 bits to store 1 entry. So the size of data in computer will be 8 times larger. Secondly, with data stored in bits, we can apply bitwise operation in C, which is very fast. For example, if we store 2 rows in 2 **int** variable a and b. the product of this 2 rows is simply the XOR operation and a Popcount[1] function on int.

$$< A_i, A_j >= 2\text{popcount}(a\hat{\ }b) - 16 \tag{20}$$

Using this strategy, we can now compute the actual memory we need to store the data we mentioned before. We have:

cancer	$10^5 \times 10^7$	100GB
bipolar	$10^4 \times 10^6$	1GB
ASD	$10^2 \times 10^4$	1MB

We can see from this chart that the cancer data is to big to run. There are 2 reasons:

1. We have basically 2 ways of reading data. The first one is to read the whole data set into RAM at the beginning of our program and read from RAM when we want to use them. The second way is to read the data from hard disk every time we update scores. Obviously the first way is

[1]The popcount function returns the number of '1' in a single int

preferred. However, when we have a 100GB data, We need a 100GB RAM to store them, which is not practical for common users. But if we use the second way, we have to read 100GB 10^7 times, just the time of transfering data will kill our program.

2. An alternative way is to use multiple computers to parallel our program so we can store data in different computers and read data to each one's RAM only once. In this situation, the communication between computers will be a big problem because rows and columns interacts with different ones frequently.

As a result, we will give up to run the cancer data and focus on the Bipolar disorder data's scale and read all the data into RAM once.

4.3 Programming Details

Since multiple computer(MPI) is not suit for our problem, we will first work on the single thread version and then parallel it using OpenMP. Several optimize techniques that are actually gain us speed are presented below:

1. Faster Bitwise operation Bitwise operation takes half of the time in the program, especially the popcount function. So if we can make popcount fast, the whole program will be faster. Here we used the 'parallel popcount' described by Anderson[Anderson]:

   ```
   static const int S[] = {1, 2, 4, 8, 16};
   static const int B[] = {0x55555555, 0x33333333, 0
      x0F0F0F0F, 0x00FF00FF, 0x0000FFFF};
   static inline int Popcount(unsigned int v){
     unsigned int c;
     c = v - ((v >> 1) & B[0]);
     c = ((c >> S[1]) & B[1]) + (c & B[1]);
     c = ((c >> S[2]) + c) & B[2];
     c = ((c >> S[3]) + c) & B[3];
     c = ((c >> S[4]) + c) & B[4];
     return c;
   }
   ```

   This method of popcount add bit number in parallel, takes only 12 operations, which is the same as the lookup-table method, but avoids the memory and potential cache misses of a table.

2. Optimization of Loops.
   As we know, nested for loops kills speed, in our code, because of the data storage strategy, there will be a small for loop with length 16(the length of **short**) inside other loops, we unrolled it as the pseudo code showed below:

   ```
   for(i=0;i<16;i++){
     Operation[i];
     }
   ```

   $\Downarrow$

```
  Operation[0];
  Operation[1];
  ...
  Operation[15];
```

This change gives about 30% speedup.

3. Improvement in memory access

   Since we store several matrix entries in one variable, we have to access the data several times to get the information of each entry. The technique here is to store this variable in a temporary variable so that this temporary one can stay in the cache, so when we read this variable, the reading is very fast. The changes in our code is:

```
 Operation[0, array[s]];
 Operation[2, array[s]];
 ...
 Operation[15, array[s]];
```

$$\Downarrow$$

```
  temp = array[s];
  Operation[0,temp];
  Operation[1,temp];
  ...
  Operation[15,temp];
```

4. Parallel technique Parallel the sequential code using openMP is not straightforward. A main difficulty is the frequent writing access from different cores to a same variable. More specifically, take an example of update formula (13). For different k, the term $< B_k, b >$ is calculated by different core, so all the cores will try to add its $< B_k, b >$ to a same score R_i'. An obvious choice is to synchronize this operation, but it only makes the program much slower even you are using multiple threads. Our strategy is create a temporary array to store the increments on scores, after we get all the increment, we add increments on all the cores to scores. So the scores will only be accessed x times, x is the number of threads we used. The pseudo code is showed below:

```
#pragma omp parallel for
 for(k=0;k<n;k++){
  int s;
  ...;
  s=...;
  ...;
#pragma omp critical
{
  Ri+= s;
}
}
```

$$\Downarrow$$

```
#pragma omp parallel
 int Ri_private;
#pragma omp for
 for(k=0;k<n;k++){
  int s;
  ...;
  s=...;
  ...;
  Ri_private+= s;
 }
#pragma omp critical
{
  Ri+=Ri_private;
}
```

Using this technique the multi-thread program achieves speedup of 2.9 for 4 cores and 5.6 for 12 cores.

These optimization techniques do accelerate our program to a great extant, numerical evidences are showed in the following numerical results section.

5 Numerical Results

There are 4 parts in this section. In the first one we define the measurements we will use in the following three parts to present the efficiency of our program. In the second part, numerical evidences of how the optimization techniques accelerate the program are presented. In the third part, we used the ASD data to test and compared the performance of the multi-thread program with different core-numbers. In the last one, we used large random matrix as input to test the performance of the multi-thread program on large matrix. Estimation of running time on Bipolar disorder data were also made.

5.1 Measurement of performance

A direct measurement is the running time of the program, but it is inconvenient when we want to compare the performance of different implementation on different data size. Therefore we define a measurement called `Time Per Unit` to compare each program's speed. We also use `Speedup` with its general definition to measure the efficiency of parallel program. Their formal definition are given below:

$$\text{Time Per Unit} = \frac{\text{running time/s}}{nm(n+m)} \tag{21}$$

$$\text{Speedup} = \frac{\text{Sequential time/s}}{\text{parallel time/s}} \tag{22}$$

5.2 Effects of Optimization Techniques

In this section, we ran several versions of our program on the ASD data sets. The data matrices has a 193×6449 case matrix A and a 128×6449 controls matrix.

In figure 1, the x-axis represent program version. Version 1 is the original program with no optimization. In version 2, we added optimization technique 1 described in last section, Version 3 corresponds to technique 1 and 2, Version 4 corresponds to technique 1,2 and 3, while the final version 5 contains all the techniques and an additional `-O3` compile flag. The left y-axis is the timing on this data. The secondary y-axis is the `Time Per Unit`.

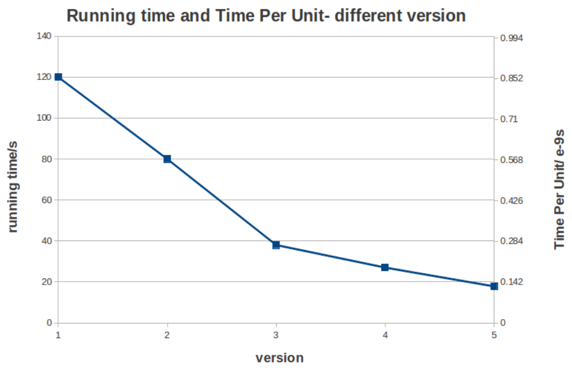

Figure 1: Sequential Program Efficiency

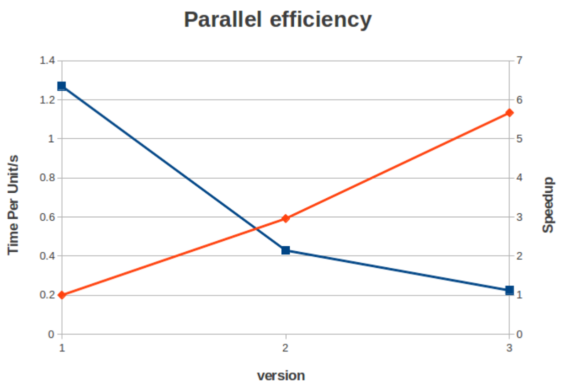

Figure 2: Parallel Program Efficiency

We can easily see from figure 1, the performance was enhanced after each optimization technique was added. In total we improved TPU from $0.852e-9$ to $0.127e-10$, which means the program is 7 times faster.

In figure 2, we investigated the efficiency of parallel version versus the sequential program final version 5. The x-axis represent version, we used 1,4,12 cores to run the program. The left y-axis is the TPU and the right y-axis represent speedup.

We can see that the multi-thread program achieves speedup of 2.9 for 4 cores and 5.6 for 12 cores. With 12 cores the TPU goes down to $2.2e-10$, which is 40 times faster than version 0.

5.3 Performance and thread Numbers

We present results of program performance with different thread numbers on random data matrix. Notice that in practical data, the ratio of column number and row number is about 100, our randomly generated matrix will also has this ratio. Particularly in this section, we chose data matrix of size 320×12800 and 320×51200, using 1,2,4,6,8,10 and 12 threads to run the program.

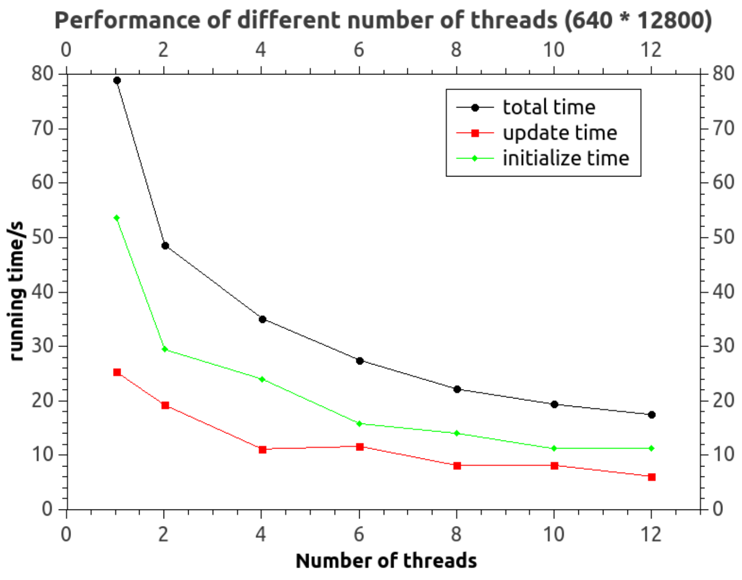

Figure 3: 640×12800

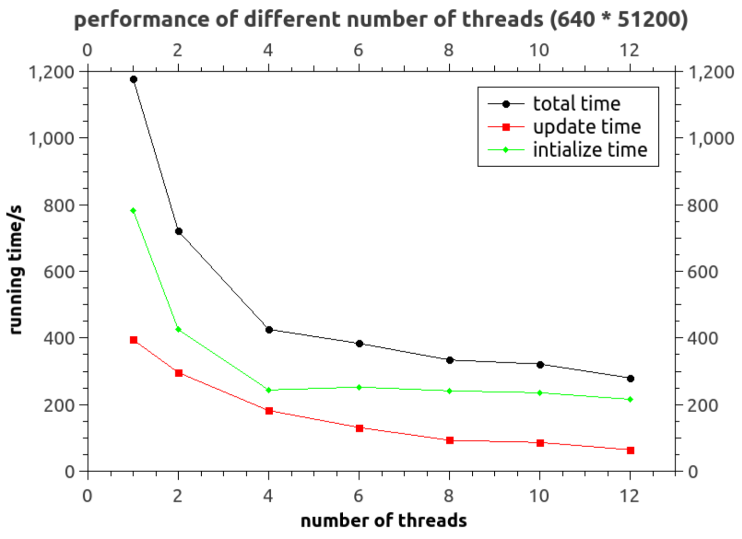

Figure 4: 640×51200

In Figure 3 and 4, x-axis represent the number of threads we used. y-axis represent the running time. Black, green and red lines are total time, update time and initialize time, respectively. In Figure 5, we plot the speedup of different cores.

We can see that our parallel program do accelerate the computation to a great extent. 12 cores give us 4 to 4.5 speedup. However, we can also see that when the thread number is bigger then 8, the speedup does not increase dramatically, this is due to the synchronization and the different efficiency between cores(The situation that some cores are waiting for other cores will occur).

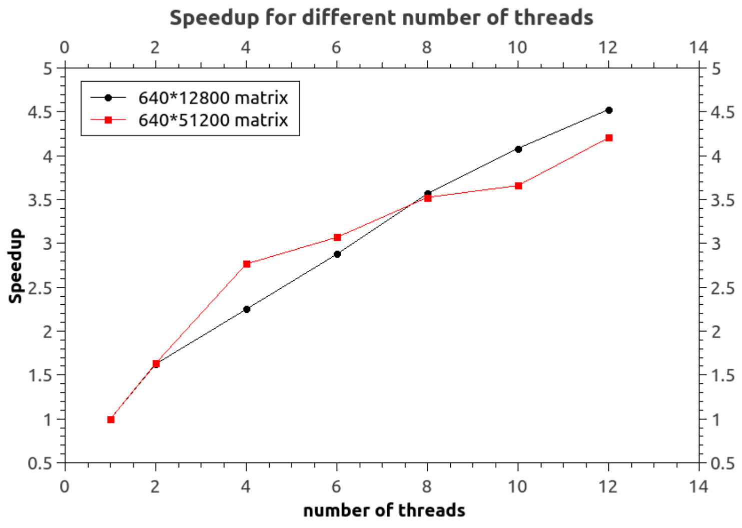

Figure 5: !!!!!

5.4 Performance on large data

In this section, we investigate the performance on matrix with different size.

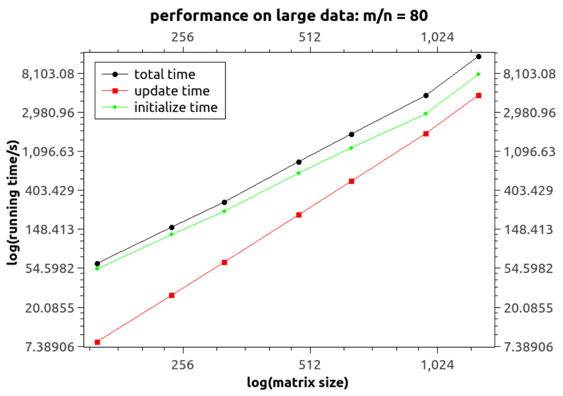

Figure 6: 640 × 12800

Again we choose random matrices that is around the ratio of 100. We investigate matrices with 2 given ratio: 20, 80. 20 is close to the ratio of ASD data, while 80 is close to the ratio of bipolar data.

In figure 6 and 7, we used the log-log axis, x-axis represents the row number, y-axis represents the running time. In figure 8, we investigated the TPU of our program on data with different size. The black and red line represents matrices with ratio 20 and 80, respectively.

From figure 6 and 7, we can see that the timing indicate that our program is indeed $O(n^3)$, because the slope is 3. From figure 8, we observed that the TPU is decreasing when we use larger matrices, this is because when the matrix is larger, the time of other lower order operation (such as the synchronized operation in last section) can be neglected.

In conclusion, the program at last achieves a TPU of $1.2e-10$ on large data matrix. As a result, we can predict the time we need to run the bipolar data:

$$\begin{aligned}\text{estimated time} &= 1.2 \times 10^{-10} \times 10240 \times 819200 \times 829440 \\ &\approx 834941s = 231h = 9d\end{aligned}$$

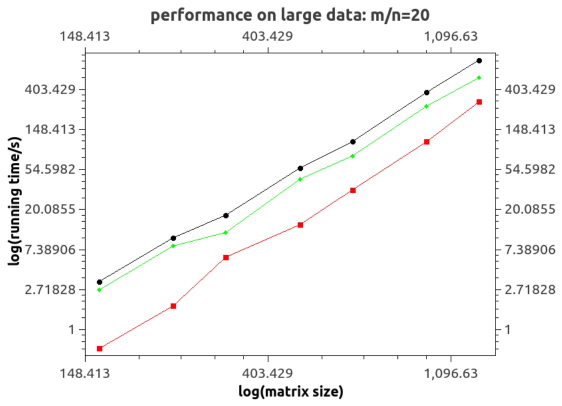

Figure 7: 640 × 51200

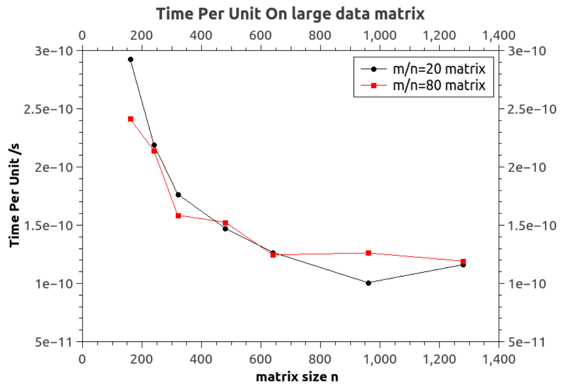

Figure 8: TPU

6 Summary

In this report, we discussed the biclustering algorithm introduced by Rangan[5] from both its theoretical intuition and implementation. For the first half, we found new formulas for the scores that are easier to understand and implement. For the second half, we implemented the algorithm in C and used openMP the get a parallel version. We discussed several optimization techniques to make the program faster, these techniques made the program 40 times faster comparing to the first version. Achieved a TPU of $1.2e-10$, which eventually enabled us to run the target dipolar disorder data.

References

[1] E. Michielssen, A. Boag, *IEEE Trans. Antennas Propag.* **44**(8)(1996) 1086–1093.

[2] S. Chandrasekaran, M. Gu, T. Pals, *SIAM J. Matrix Anal. Appl.* **28**(2006) 603–622.

[3] E. Candes, L. Demanet, Y. Lexing, *SIAM J. Sci. Comput.* **29**(6) (2007) 2064–2093.

[4] F. Woolfe, E. Liberty, V. Rokhlin, M. Tygert, *PNAS* **104**(51) (2007) 20167–20172.

[5] A.V. Rangan, *J. Comput. Phys.* **231**(7) (2012)9,

[6] X.V. Doan, S.A. Vavasis, *¡arXiv:1011.1839v1¿* (2010)

[7] A.V. Rangan , *J. Comp. Phys.* **231**(1) (2012) 215–222.

[8] Y. Kluger, R. Basri, J.T. Chang, M. Gerstein, *Genome Res.* **13**(4):(2003)703–716.

[9] http://graphics.stanford.edu/ seander/bithacks.html#CountBitsSetParallel.