

# Optimization and Parallelization of MRF Community Detection Algorithm for a Specific Network

Jun Lu<sup>a,b,\*</sup> and Yuanzhong Zhang<sup>a</sup>

<sup>a</sup>College of Computer Science and Technology, Heilongjiang University, Harbin, 150080, China

<sup>b</sup>Key Laboratory of Database and Parallel Computing of Heilongjiang Province, Harbin, 150080, China

---

## Abstract

Research on the optimization and parallelization of the MRF network community detection algorithm for a specific network is carried out in this paper. Firstly, the principle of the existing algorithm is expounded, the algorithm is analyzed, and some problems are pointed out. Some optimization strategies and rules are proposed, including the extraction of variables and operations from inner loops to outer loops, the merging of related operations in loops, the removal of redundant loops, and the split of loops. In order to achieve better parallelism, OpenMP parallel computing of this method is realized by reversing the order of inner and outer loops. The influence of the density of network edges on the algorithm efficiency is also analyzed in this paper. The optimization and parallel algorithm can be applied to the module partition of Alzheimer's disease gene data, and the efficiency of the algorithm is greatly improved. The optimization strategies and rules proposed in this paper can be further extended to general situations. It is significant in practical applications.

**Keywords:** community detection; MRF; optimization; parallel

(Submitted on June 11, 2019; Revised on July 5, 2019; Accepted on August 10, 2019)

© 2019 Totem Publisher, Inc. All rights reserved.

---

## 1. Introduction

In recent years, the analysis of complex networks such as social networks and biological networks has become an important research direction. A basic problem in the analysis of complex systems is to identify network communities or modules, such as customer segmentation based on interests or preferences [1], recommendation systems in social media consumption platforms [2], and community analysis in behaviour social goals [3]. Community identification and analysis can help understand the functions and organizational principles of a system and predict its future trend. These communities or modules are usually embedded in a large number of network data, so how to quickly and effectively identify and analyze a network community with a rapidly accumulated large amount of data has become an urgent problem to be solved.

In the same network community, highly connected nodes often have the same attributes and form functional modules. Community detection in complex networks was initially considered as a structure-based graph partitioning problem [4]. Its main purpose is to divide the nodes in the network into groups, so that the nodes in the same group are connected intensively, while the nodes in different groups are loosely connected. Various community detection algorithms have been proposed: hierarchical clustering [4], modular-based method [5], heuristic method [6], dynamic algorithm [7], spectral algorithm [8], and statistical model-based method [9]. Among these methods, the method based on statistical models is widely used, because it has a good theoretical foundation and better performance.

Markov random field (MRF) is a general and powerful statistical model technology [10]. It can be represented by undirected graphs and is usually used to deal with problems in computer vision, images, Chinese chunking [11], and so on. Because the problem of network community discovery is similar to the problem of image segmentation, He et al. extended MRF to the network community detection problem. They proposed a new pMRF (pairwise MRF) model for network community identification, which is superior to most existing technologies [12]. Obviously, in the biological gene expression network, this algorithm can also be used for module partitioning. In gene expression networks, a set of genes with similar

\* Corresponding author.

E-mail address: [lujun111\\_lily@sina.com](mailto:lujun111_lily@sina.com)

functions or relatively fixed associations is called a module. Recognition of network modules in biological data can mine the relationship among biological structure, function, and behaviour, and then effectively identify its biological function modules, predict protein function, and predict disease genes.

However, when the network scale increases, the time and space requirements of the algorithm will be greatly increased, so it is necessary to optimize and parallelize the algorithm to improve the efficiency of the algorithm. In this paper, by analyzing the time and space relationship of the algorithm, the relevant principles of space-time optimization are extracted. Finally, good optimization and a parallel effect can be achieved. Furthermore, these optimization principles can be extended to general cases, so they have very good practical significance.

## 2. Overview of NetMRF

The MRF network community partitioning method for specific networks is called NetMRF. The so-called specific network does not have node feature information, that is, there is no node feature that can be used to construct a unitary potential, but only a binary potential that reflects the topological attribute of the network. This network satisfies the following four requirements: (1) reward internal edges between nodes in the same group, (2) punish existing edges in different groups, (3) punish the missing edges between nodes in the same group, and (4) reward non edge in different groups.

NetMRF supports these four requirements by introducing auxiliary complete graphs. In addition, in order to make full use of the adjacency relationship between nodes, NetMRF also innovatively introduces random graphs. In a random graph, the number of nodes and edges is the same as that of a given network graph. Each node has the same degree, and the nodes are randomly connected. There are only a few community structures in the random graph, while the real network usually hides some community structures. By comparing a given network graph with a random graph, the degree of density and sparseness within the community can be evaluated, so the adjacency relationship between nodes can be used fully.

The NetMRF model is an undirected network graph  $G$  with  $n$  nodes and  $m$  edges. The adjacency matrix of  $G$  is  $= (a_{ij})_{n \times n}$ . If node  $i$  and node  $j$  are connected, then  $a_{ij} = 1$ ; otherwise,  $a_{ij} = 0$ . Assuming that the nodes are divided into  $K$  communities,  $c_i (c_i \in \{1, \dots, K\})$  represents the community to which node  $i$  belongs. The auxiliary graph  $D$  of the pMRF model is a complete graph with the same number of nodes as the given network  $G$ .

The energy function of the NetMRF model is the sum of the pairwise potentials of all pairs of nodes in a given network, and the minimum value of the energy function corresponds to the possible optimal community partition. The pairwise potential of a pair of nodes is calculated by the difference between the edge density of a given network and the edge density of a random graph. The pairwise potential between node  $i$  and node  $j$  can be defined as Equation (1).

$$\theta_{ij}(c_i, c_j; a_{ij}) = -(-1)^{\delta(c_i, c_j)} \left( \frac{d_i d_j}{2m} - a_{ij} \right) \quad (1)$$

Where  $d_i$  is the degree of node  $i$  in undirected network graph  $G$  and  $d_j$  is the degree of node  $j$  in graph  $G$ . If  $c_i = c_j$ , the value of  $\delta(c_i, c_j)$  is 1; otherwise, the value of  $\delta(c_i, c_j)$  is 0. In Equation (1),  $d_i d_j / (2m)$  is the expected density of the edges between node  $i$  and node  $j$  in a random graph.

In this way, the energy function of the pairwise potential of the NetMRF model can be defined as Equation (2).

$$E(C; A) = \sum_{i \neq j} \theta_{ij}(c_i, c_j; a_{ij}) = \sum_{i \neq j} [ -(-1)^{\delta(c_i, c_j)} \left( \frac{d_i d_j}{2m} - a_{ij} \right) ] \quad (2)$$

Then, the Gibbs distribution  $P(C|A) \propto \exp\{-\beta E(C; A)\}$  can be used to change  $\beta$  to calculate the posterior probability of partition  $C$  of a given network topology  $A$ , as shown in Equation (3).

$$P(C|A) = \frac{1}{Z(A)} \prod_{i \neq j} \exp\{(-1)^{\delta(c_i, c_j)} \beta \left( \frac{d_i d_j}{2m} - a_{ij} \right)\} \quad (3)$$

$Z(A)$  is a normalization term, which relies on adjacent matrix  $A$  to ensure  $P(C|A)$  is a probability distribution. From Equation (3), it can be seen that the smaller the energy function, the greater the posterior probability. Finally, community partition  $C$  on  $n$  nodes can be estimated as a posterior global maximum of the pMRF model, as shown in Equation (4).

$$C_{max} = \operatorname{argmax}_c P(C|A) \quad (4)$$

In order to obtain the best community partition  $C$  that maximizes the posterior probability  $P(C|A)$  of the NetMRF model, messages sent from node  $i$  to node  $j$  can be iteratively computed with Equation (5) based on all messages received by node  $i$  from its other neighbor  $k$ .

$$\varphi_{c_i}^{i \rightarrow j} \leftarrow \sum_{k \in N(i) \setminus j} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} - a_{ik} \right) + \varphi_{c_k}^{k \rightarrow i} \right] \right\} \quad (5)$$

$N(i)$  is the neighborhood of node  $i$  in complete graph  $D$ . When the algorithm converges,  $\mu_i(c_i)$  about the variable *max-belie* can be calculated as shown in Equation (6).

$$\mu_i(c_i) \leftarrow \sum_{k \in N(i)} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} - a_{ik} \right) + \varphi_{c_k}^{k \rightarrow i} \right] \right\} \quad (6)$$

*Max-beliefs* is a scoring function whose maximum value corresponds to the community configuration with the largest joint probability. In order to find the joint maximum posterior configuration, for each variable  $C_i$ , the state with the biggest maximum confidence is selected based on Equation (7).

$$\hat{c}_i = \operatorname{argmax}_{c_i \in \{1, \dots, K\}} \mu_i(c_i) \quad (7)$$

In the algorithm, messages on all edges of the complete graph  $D$  need to be computed in iteration for each time.  $n(n-1)$  messages are needed in total. An efficient maximum sum version of BP is used to calculate the marginal probability. The message sent from node  $i$  to neighbor  $j$  can be calculated as shown in Equation (8).

$$\begin{aligned} \varphi_{c_i}^{i \rightarrow j} \leftarrow & \sum_{k \in N_G(i) \setminus j} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} - 1 \right) + \varphi_{c_k}^{k \rightarrow i} \right] \right\} \\ & + \sum_{k! \in N_G(i)} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} + \mu_k(c_k) \right) \right] \right\} \end{aligned} \quad (8)$$

$N_G(i)$  is the neighborhood of node  $i$  in network  $G$ .

Each node can send the same message to all of non-neighboring in its original network, and messages sent to non-neighboring can be replaced by external fields [13]. Messages sent from node  $i$  to neighbor  $j$  can be computed based on an auxiliary external field, as shown in Equation (9).

$$\varphi_{c_i}^{i \rightarrow j} \leftarrow \sum_{k \in N_G(i) \setminus j} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} - 1 \right) + \varphi_{c_k}^{k \rightarrow i} \right] \right\} + h_{c_i}(d_i) \quad (9)$$

$N_G(i)$  is the neighborhood of node  $i$  on network  $G$ , and  $h_{c_i}(d_i)$  is the external field. The external field is a function about the degree value of the node. It needs to calculate  $L$  external fields, where  $L$  is the number of different degrees.  $h_{c_i}(d_i)$  can be computed as shown in Equation (10).

$$h_{c_i}(d_i) = \sum_{k=1}^n \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \frac{d_i d_k}{2m} + \mu_k(c_k) \right] \right\} \quad (10)$$

The maximum beliefs  $\mu_i(c_i)$  can be calculated based on the auxiliary external field, as shown in Equation (11).

$$\mu_i(c_i) \leftarrow \sum_{k \in N_G(i)} \left\{ \max_{c_k} \left[ (-1)^{\delta(c_i, c_k)} \beta \left( \frac{d_i d_k}{2m} - 1 \right) + \varphi_{c_k}^{k \rightarrow i} \right] \right\} + h_{c_i}(d_i) \quad (11)$$

In order to find the fixed point of Equation (9) in linear time, the message  $\varphi_{c_i}^{i \rightarrow j}$  needs to be updated and  $\mu_i(c_i)$  needs to be recalculated. Furthermore, the field  $h_{c_i}(d_i)$  needs to be updated by adding the new contributions and subtracting the old contributions. All these works should be repeated until the message converges to a fixed point.

Compared with the existing methods, the NetMRF method achieves better effect and speed.

### 3. BP Algorithm in NetMRF Method

The NetMRF method can be used to identify and analyze a variety of network communities, and these networks only provide topological information, such as image processing, medical data processing, etc. In this paper, the data of Alzheimer's disease is processed in the experiment. Firstly, these data need to be converted into a probability graph model. Secondly, the node information (node number and label) and edge information (starting point and end point) are initialized. Then, the main program flow, including the BP algorithm, begins to be executed. The BP algorithm is the most time-consuming of the whole program.

In the original BP algorithm of NetMRF, the outermost loop is controlled by the maximum number of iterations (which are configured according to requirement). If the algorithm does not reach the maximum number of iterations and converge in advance, and then it will return the number of iterations and exit the BP algorithm. The second layer loop is controlled by the edge number of the undirected graph. Obviously, the number of edges in the network determines the speed of the algorithm. The more edges there are, the more time is consumed. The first edge to be processed with node  $i$  as the starting node is different from the other edges to be processed with node  $i$  as the starting node. The former requires more data to be updated by the function named *mod\_bp\_iter\_update\_psi*, while the latter requires less data to be updated by the function named *mod\_bp\_iter\_update\_message\_only*. Therefore, the computation time is mainly concentrated on the former.

Before entering the function named *mod\_bp\_iter\_update\_psi*, it needs to be pre-processed, that is, the number of different degrees in the network is counted from the first node and stored in the variable *count1*. Furthermore, the different degrees are stored in the array named *arrayd1*. Firstly, the function *mod\_bp\_iter\_update\_psi* needs to calculate which position in *arrayd1* is the degree value of the current node  $i$ , so that when the *belief* and *message* of node  $i$  are updated, the influence of the non-neighboring nodes of node  $i$  can be approximated by the column of the external field matrix. Then, the influence of the current node in all the external fields can be subtracted, and the values of *belief* and *message* can be calculated and normalized. Errors also can be computed based on different cases. Finally, the influence of node  $i$  stored in the array *external\_field\_node* on all external fields is updated. The pseudocode of the specific algorithm is shown in Figure 1.

In Figure 1, lines 1-6 calculate which position in *arrayd1* is the degree value of the current node  $i$ , and the result is placed in the counter variables  $s$ . Lines 7-11 subtract the influence of the current node in all external fields. Lines 12-32 compute and normalize *belief*. Lines 33-41 compute and normalize *message*. Lines 42-70 update data. Lines 73-75 return the value of the function based on the value of *averr*.

In the function *bp\_converge*, there is a total of  $2m$  ( $m$  is the number of edges in the graph, each edge needs to be processed twice: from point 1 to point 2 and from point 2 to point 1) times to decide whether to call the function *mod\_bp\_iter\_update\_psi* according to different circumstances. When variable *averr* is *true*, the average integrity error of BP loop iteration will be triggered, and when variable *averr* is false, the maximum error of BP loop iteration will be triggered.

In this algorithm, when the amount of network data is large, the computing speed will be seriously affected. For the graph model with sparse edges and dense edges, even if the number of nodes is the same, the computing speed will also be different. That is to say, the sparse/dense degree of edges will also affect the computing speed of the algorithm. Obviously, the BP algorithm still needs further optimization and parallel processing in order to improve the operation efficiency to a greater extent.

Some optimization strategies and rules are proposed for this algorithm in this paper, which greatly improves the efficiency of the algorithm. Furthermore, some optimization strategies and rules are not only limited to the algorithm, but also universal to other algorithms, so they have good practical significance.

**Algorithm 1** *mod\_bp\_iter\_update\_psi*

**Input:** start node  $i$  of edge,  
end node  $k$  of edge,  
Shock-proof variable *dumping\_rate* to compute message

**Output:** error of message

```

1  for e=0 to count1-1 do
2    if arrayd1[e]==degree(i) then
3      break;
4    end if
5    s=s+1;
6  end for
7  for q=0 to Q-1 do
8    for m=0 to count1-1 do
9      external_field[q][m] = external_field[q][m]
        -external_field_node[q][m][i];
10   end for
11 end for
12 for q=0 to Q-1 do
13   for j=0 to N-1 do
14     if i != j then
15       for e=0 to L(i)-1 do
16         Computing haveEdge related to belief
17         if k=j then
18           neighbor[ci][k] = max;
19         end if
20       end for
21     end if
22   end for
23   pom_psi[ci] = haveEdge + external_field[ci][s];
24 end for
25 for ci=0 to Q-1 do
26   if pom_psi[ci] < normalMin then
27     normalMin = pom_psi[ci];
28   end if
29 end for
30 for ci=0 to Q-1 do
31   real_psi[i][ci] = pom_psi[ci] - normalMin;
32 end for
33 for ci=0 to Q-1 do
34   psii_iter[ci][k] = pom_psi[ci] - neighbor[ci][k];
35 end for
36 normalMin1 = 1000000.0;
37 for ci=0 to Q-1 do
38   if psii_iter[ci][k] < normalMin1 then
39     normalMin1 = psii_iter[ci][k];
40   end if
41 end for
42 for q=0 to Q-1 do
43   for m=0 to count1-1 do
44     max = 0.0;
45     for qq=0 to Q-1 do
46       if q=qq then
47         b=-1.0;
48       else
49         b=1.0;
50       end if
51       ttt = beta * arrayd1[m] * di * b / TM
        + real_psi[i][qq];
52       if max<ttt then
53         max=ttt;
54       end if
55     end for
56     external_field_node[q][m][i] = max;
57     external_field[q][m] += max;
58   end for
59 end for
60 for ci=0 to Q-1 do
61   Compute the current error mydiff
62   if averr=true then
63     mymaxdiff += mydiff;   ecount++;
64   else
65     if mydiff>mymaxdiff then
66       mymaxdiff = mydiff;
67     end if
68   end if
69   Update other related data to compute haveEdge
70 end for
71 if averr =true then
72   return mymaxdiff / ecount
73 else
74   return mymaxdiff
75 end if

```

Figure 1. *Mod\_bp\_iter\_update\_psi***4. Optimization of NetMRF Algorithm**

The network community partitioning method of MRF in a specific network is mainly realized by the BP algorithm. The *mod\_bp\_iter\_update\_psi* function in Section 3 is the core of the BP algorithm and consumes the most computing time. Many operations of this function can be further optimized.

**4.1. Optimization of Counter**

In the first step of the algorithm (lines 1-6) in Section 3, the value of counter  $s$  will affect belief calculation. This step is equivalent to calculating which position in *arrayd1* is the degree value of the current node  $i$  when *mod\_bp\_iter\_update\_psi* function is executed every time. This means that for many edges with node  $i$  as the start node, the calculation will be repeated. Therefore, the  $s$  value corresponding to each starting node needs to be calculated and saved in array ( $s[ ]$ ) before the BP operation. When necessary,  $s[i]$  can be used directly without recalculating, and thus the amount of computation can be reduced. It can be shown in Algorithm 2 in Figure 2.

**Algorithm 2** Optimization of counter

```

1  for i=0 to N-1 do
2    s[i]=0;
3  end for
4  for L=0 to count1-1 do
5    if arrayd1[L]==degree(i) then
6      break;

```

```

6      end if
7      s[i]=s[i]+1;
8  end for
9  end for

```

Figure 2. Optimization of counter

#### 4.2. Combine the Computing of External Field Impact Values

In Section 3, the second step of the algorithm (lines 7-11) needs to subtract the influence of the current node from the external field information. In fact, only a small part of the *external\_field* array information is used in step 3 (lines 12-32), and only a small amount of information is needed to subtract the influence of the current node to calculate the *belief* value.

$$pom\_psi[ci] = haveEdge + (external\_field[ci][s[i]] - external\_field\_node[ci][s[i]][i])$$

In this way, the second step of the external field information minus the influence of the current node can be combined with the update of the external field information in the fifth step (lines 42-70). Thus, the computational complexity can be reduced.

$$external\_field[ci][m] = external\_field[ci][m] - external\_field\_node[ci][m][i] + max$$

$$external\_field\_node[ci][m][i] = max$$

#### 4.3. Optimization of Belief Computing

In the third step of the algorithm (lines 12-32) in Section 3, when *belief* is computed, it involves multiple loops. It is found that for each node *j*, if node *i* is not equal to node *j*, then for each edge *e* of node *i*, if node *j* happens to be the other end of *e*, then *haveEdge* and the contribution of all neighboring nodes of node *i* need to be calculated. The above logic is essentially equivalent to the other node *j* of each edge *e* of node *i*, and *haveEdge* and the contribution of all neighboring nodes of node *i* are calculated directly. This reduces loop and improves the operational efficiency. The improved algorithm is shown in Algorithm 3 in Figure 3.

**Algorithm 3** Optimization of belief computing

```

1  for q=0 to Q-1 do
2      for e=0 to L(i)-1 do
3          Computing haveEdge related to belief
4          if k=j then
5              neighbor[ci][k] = max;
6          end if
7      end for
8      pom_psi[ci] = haveEdge + external_field[ci][s];
9  end for
10 for ci=0 to Q-1 do
11     if pom_psi[ci] < normalMin then
12         normalMin = pom_psi[ci];
13     end if
14 end for
15 for ci=0 to Q-1 do
16     real_psi[i][ci] = pom_psi[ci] - normalMin;
17 end for

```

Figure 3. Optimization of belief computing

#### 4.4. Merge Relevant Operations of Message

Some loop program sections with little correlation can be merged if the numbers of loops are the same. Furthermore, in order to calculate the maximum (minimum) value in an array, the first value of the array should be assigned to the minimum variable firstly and then compared one by one, instead of giving the minimum value that does not appear in the array to the minimum variable. Although there seems to be only one time for reduction operation, if the calculation is in a large number of loops or within multiple loops, the number of operations saved is considerable. In this case, the maximum number of loops that call the *mod\_bp\_iter\_update\_psi* function is  $2m \times time$  (*time* is the maximum number of iterations). Therefore, *message* computing can be optimized as Algorithm 4 in Figure 4.

**Algorithm 4** Optimization of message computing

```

1  normalMin1 = pom_psi[0] - neighbor[0][k];
2  psii_iter[0]=normalMin1;
3  for ci=1 to Q-1 do
4    psii_iter[ci][k] = pom_psi[ci] - neighbor[ci][k];
5    if psii_iter[ci][k] < normalMin1 then
6      normalMin1 = psii_iter[ci][k];
7    end if
8  end for

```

Figure 4. Optimization of message computing

#### 4.5. Optimization of Data Updating

The simply repeated computing part of the loop can be calculated ahead of time outside the loop and temporarily store the result of the operation. Then, the temporary result can be directly used inside the loop, and thus the amount of calculation can be reduced.

If it needs to be selectively processed based on the value of the loop variable in the process of the loop, then the loop can be split, which reduces the judgment operation in the new loop body after the split. Thus, the operation efficiency is improved.

Therefore, updating the effect of node  $i$  (stored in array *external\_field\_node*) on all external fields can be optimized as Algorithm 5 in Figure 5.

**Algorithm 5** Optimization of data updating

```

1  for q=0 to Q-1 do
2    for m=0 to count1-1 do
3      tmp=beta*arrayd1[m]*di/TM;
4      max=-tmp+real_psi[i][q];
5      for qq=0 to q-1 do
6        ttt=tmp+real_psi[i][qq];
7        if (max < ttt) then
8          max = ttt;
9        end if
10       end for
11       for qq=q+1 to Q-1 do
12         ttt=tmp+real_psi[i][qq];
13         if (max < ttt) then
14           max = ttt;
15         end if
16       end for
17     end for
18   end for

```

Figure 5. Optimization of data updating

#### 4.6. Extraction of Selection Operation

The Boolean variable *averr* is judged many times in the program, and different processing is carried out according to the judgement result: when *averr* is *true*, the average integrity error of BP loop iteration will be triggered, and when *averr* is *false*, the maximum error of BP loop iteration will be triggered. Although the processing of the Boolean variable is not complicated, its judgment operations are placed in the loop and the execution times are considerable. If the judgment operations of the Boolean variable are extracted to the outermost loop, a different function is called according to their judgment value, that is, the original function can be divided into two functions. The difference between two functions is only that the relevant operations are directly executed according to different *averr* values. For example, for the original function *mod\_bp\_iter\_update\_psi*, lines 62-68 and lines 71-75 can be optimized.

This means that the original function *mod\_bp\_iter\_update\_psi* can be split two functions: *mod\_bp\_iter\_update\_psi\_1* and *mod\_bp\_iter\_update\_psi\_2*. When *averr* is equal to *true*, *mod\_bp\_iter\_update\_psi\_1* function will be called. This function is optimized. Lines 62-68 become:

```

mymaxdiff += mydiff;
ecount++;
Lines 71-75 become:

```

```

return mymaxdiff / ecount

```

When *averr* is equal to *false*, *mod\_bp\_iter\_update\_psi\_2* function will be called. This function is optimized, and lines 62-68 become:

```

if mydiff > mymaxdiff then
  mymaxdiff = mydiff;

```

*end if*

Lines 71-75 become:

*return mymaxdiff*

Obviously, in the optimization algorithm, the judgment of the Boolean variable *averr* only needs to be executed once. Compared with the original algorithm, which executes judgments many times by calling functions in a loop, it reduces the program running time.

In addition to the main function *mod\_bp\_iter\_update\_psi* in BP algorithm, other functions can be optimized similarly. After optimization, the efficiency of the whole algorithm is greatly improved.

This algorithm can be used to process the data of Alzheimer's disease and then research on dividing the genetic data of Alzheimer's disease into modules. In order to calculate the data processing time more accurately, data of each group is run three times, and then the average value is taken as the final processing time. Table 1 shows the processing time of edge-dense network graph data by the original algorithm, and Table 2 shows the processing time of edge-sparse network graph data by the original algorithm.

Table 1. Dense data processing time of the original algorithm (s)

Node	First	Second	Third	Average
3000	176.716	176.458	176.651	176.608
4000	412.524	412.384	412.641	412.516
5000	796.598	796.798	796.406	796.601
6000	1365.6	1365.92	1368.21	1366.577
7000	2154.37	2153.66	2159.68	2155.903
8000	3201.45	3199.96	3193.34	3198.250
9000	4560.1	4567.47	4561.53	4563.033
10000	6198.1	6203.1	6247.34	6216.180

Table 2. Sparse data processing time of the original algorithm (s)

Node	First	Second	Third	Average
3000	20.171	20.185	20.196	20.184
4000	46.093	46.113	46.139	46.115
5000	88.594	88.793	88.716	88.701
6000	150.964	150.546	150.636	150.715
7000	236.467	237.133	237.727	237.109
8000	349.023	350.888	348.737	349.549
9000	492.091	491.382	490.736	491.403
10000	667.862	667.657	667.723	667.747

The nodes in Table 1 and Table 2 were selected according to certain conditions of gene data. In the gene network graph, nodes represent different individuals, while edges represent internal relationships among individuals. The closer the relationship between individuals, the more edges there are and the greater the amount of calculation. Table 1 has the same number of nodes as Table 2, but the corresponding number of edges is ten times than that of Table 2. Tables 3 and 4 show dense data and sparse data processing time after optimization, respectively.

The polyline graph can be used to show the efficiency of the optimization algorithm more clearly. Figures 6 and 7 show a comparison of the running time of different data quantities of Alzheimer's disease gene data before and after optimization.

Table 3. Dense data processing time of the optimization algorithm (s)

Node	First	Second	Third	Average
3000	5.90106	5.85378	5.95574	5.904
4000	14.824	14.638	14.8202	14.761
5000	26.9479	28.1487	26.7469	27.281
6000	40.8522	40.5392	40.9579	40.783
7000	61.2183	60.8283	61.4357	61.161
8000	86.0759	86.7632	85.7602	86.200
9000	115.707	114.297	115.198	115.067
10000	147.447	147.708	146.043	147.066

Table 4. Sparse data processing time of the optimization algorithm (s)

Node	First	Second	Third	Average
3000	0.630785	0.490285	0.501147	0.541
4000	0.918575	0.946323	0.942303	0.936
5000	1.81004	1.57645	1.58681	1.658
6000	2.59126	2.82106	2.61722	2.677
7000	4.90444	5.43291	5.52687	5.288
8000	7.26354	7.44252	8.3276	7.678
9000	11.4181	11.6191	11.2714	11.436
10000	15.0421	15.1497	15.152	15.115

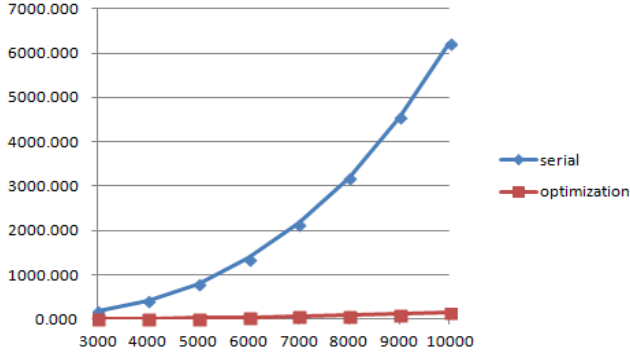


Figure 6. Computing speed of dense edge network

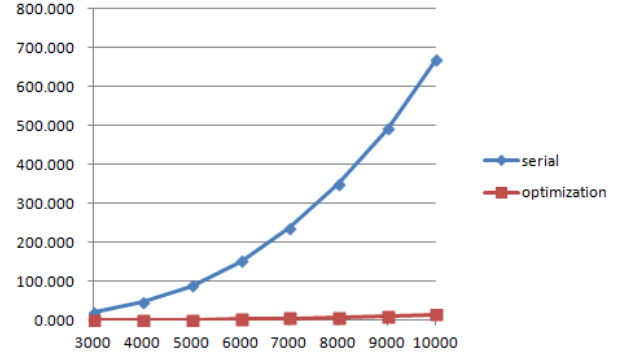


Figure 7. Computing speed of sparse edge network

The network shown in Figures 6 and 7 has the same number of nodes, but the density of edges is different, and the difference is ten times. It can be seen that the edge density of Figure 6 is higher, and the operation efficiency is not as fast as that of Figure 7 with sparse edges. However, the efficiency of sparse network processing is not much faster and does not show a linear growth.

In order to see the time effect more clearly, Figure 8 shows the multiples of the time spent before and after optimizing dense data and sparse data. Each node in Figure 8 is calculated from  $a/b$ , where  $a$  is the time spent by dense (sparse) data before optimizing and  $b$  is the time spent by dense (sparse) data after optimizing.

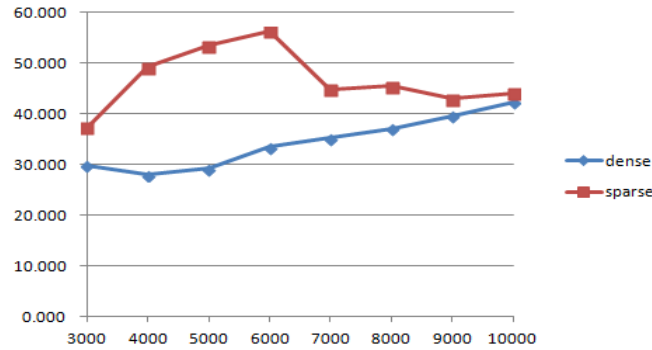


Figure 8. Multiples of the time spent before and after optimizing

As can be seen from Figure 8, the optimization efficiency of sparse data does not increase monotonously with the increase in the number of nodes, but the optimization efficiency of sparse data is generally higher than that of dense data.

The main reason for the above phenomenon is that the optimization strategy of the BP algorithm is mainly used in the loop of the *mod\_bp\_iter\_update\_psi* function for the number of network nodes ( $N$ ) or the number of blocks ( $Q$ ), while the number of times that the outer loop (the second cycle) controlled by edges performs the *mod\_bp\_iter\_update\_psi* function by conditional selection is limited. That is, this function can only be performed on the first edge of a node as the starting node, so the optimization effect does not increase linearly but tends to be similar to the increase in nodes. In addition, according to the relevant conditions in different situations, the algorithm will also affect the final selection results and operational efficiency.

## 5. Parallelization of BP Algorithm

Because the parallel technology has achieved good results in related research [14], parallel technology can also be used in this algorithm to further improve the operational efficiency.

The commonly used parallel technologies can be divided into two categories according to the type of multi-core: multi-core parallel computing based on CPU and multi-core parallel computing based on GPU. The former has fewer kernels and each core has strong computing power, so each parallel unit can perform complex computing. Multiple CPU cores can share memory. Generally speaking, switching between serial and parallel codes costs less time; the latter has more kernels and each core has weak computing power, which is suitable for a large number of parallel units, but the program executed by each parallel unit is simpler. Especially for serial/parallel handover, when the program needs to be executed in GPU, it takes a long time to switch the program flow from CPU to GPU and copies a large amount of data from CPU to device (GPU) memory. On the other hand, when the program is executed in GPU, long switching time is also required to transfer the running results from device (GPU) memory to CPU memory. If the number of parallel units in the program is small, the time saved by parallel execution is not enough to offset the switching time between CPU and GPU, which will lead to low parallel efficiency and sometimes even cause the parallel program to be slower than serial programs.

In this paper, the original program structure is analyzed. In the outermost loop, which is controlled by the number of iterations, and the second loop, which is controlled by the number of edges, the result data of the last loop need to be used in the current loop, so parallel technology cannot be used in the outermost loop and the second loop. Parallel computation can only be performed for the third and more inner loops, such as dealing with loops in the *mod\_bp\_iter\_update\_psi* function. However, after analysis, it is found that most of the loops that may be converted to parallel operations are controlled by the number of communities, and the number of communities is usually not set very high, so the parallel efficiency is low. The number of some loops is controlled by the variable *count1*, which stores the number of node degree in the network. In general, the number of degrees is larger than the number of communities, which means that *count1* is more fit for parallelism than the number of community loops controlled by *Q*. However, because *count1* is still not large enough (usually only a few thousand), it is still not suitable to use GPU parallel technology. Thus, OpenMP, which is CPU parallel technology, is adopted in this paper. The workstation parameter Core (s) per socket used in this experiment is 2, which can adopt hyper-threading technology. Generally speaking, if the CPU core number is *m*, the number of threads can be set up between *m-2m*. When the amount of computation allocated by each thread is quite different, that is, when the amount of computation is unbalanced, *2m* threads can be set up. The workstation used in this experiment has *m* = 20 cores. Because the workload of each thread is basically balanced, only 20 threads can be opened.

In order to facilitate parallel processing in the future, sometimes the position of inner and outer loops can be reversed, and it is more advantageous to move the loop with a higher number of loops to the outside. For example, the loop structure controlled by the variable *count1* belongs to the inner loop of the loop controlled by the number of communities *Q*. After analysis, it is found that the two loops can be inverted both inside and outside because of the irrelevance of the data processed by the two loops. Thus, the influence of the node *i* stored in the update array *external\_field\_node* on all the external fields can be converted into more efficient parallel computing. The algorithm is shown in Figure 9.

<p><b>Algorithm 6</b> Parallel algorithm</p> <pre> 1 #pragma omp parallel num_threads(Number_parallel) 2 { 3   int ci,qq; 4   double tmp,max,ttt; 5   #pragma omp for private(m) schedule(dynamic) 6   for m=0 to count1-1 do 7     tmp=beta*arrayd1[m]*di/TM; 8     for q=0 to Q-1 do 9       max=-tmp+real_psi[i][q]; 10      for qq=0 to q-1 do 11        ttt=tmp+real_psi[i][qq]; 12        if (max &lt; ttt) then 13          max = ttt;</pre>	<pre> 14   end if 15   end for 16   for qq=q+1 to Q-1 do 17     ttt=tmp+real_psi[i][qq]; 18     if (max &lt; ttt) then 19       max = ttt; 20   end if 21   end for 22   external_field[q][m]=external_field[q][m] 23   -external_field_node[q][m][i]+max; 24   external_field_node[q][m][i] = max; 25 end for 26 }</pre>
---	---

Figure 9. Parallel algorithm

Table 5 is the time for parallel processing of sparse and dense data on the basis of optimized code. Each time in Table 5 is the average value of running the algorithm three times. The parallel thread number is 20.

Table 5. Processing time on sparse and dense data in parallel (s)

Node	3000	4000	5000	6000	7000	8000	9000	10000
Dense	4.399	7.805	13.013	20.351	30.105	40.785	55.295	75.792
Sparse	0.731	1.089	1.688	2.309	2.986	3.684	5.157	6.377

Figures 10 and Figure 11 show the comparison of the run time between the optimized and parallel algorithm after the optimization program is parallelized. Figure 10 is the computing speed before and after dense network parallelization. Figure 11 is the computing speed before and after sparse network parallelization.

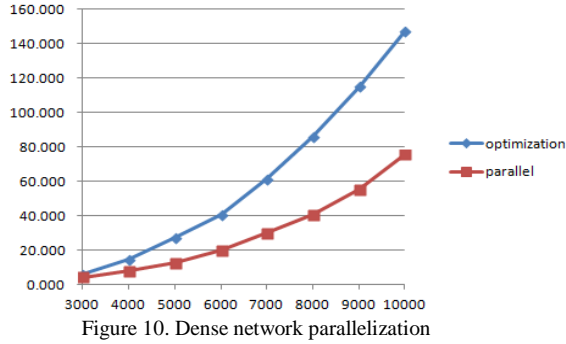


Figure 10. Dense network parallelization

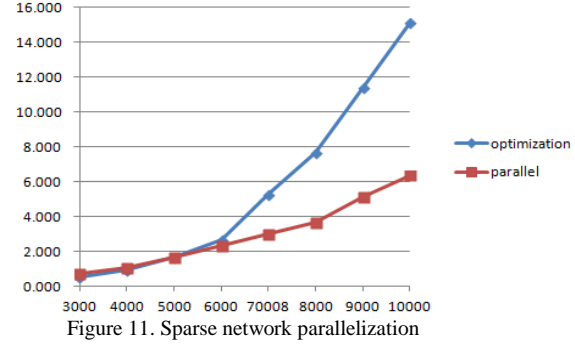


Figure 11. Sparse network parallelization

As can be seen from Figures 10 and 11, the parallel efficiency of sparse edge networks with the same number of nodes is faster than that of dense edge networks, but for networks with fewer nodes, the efficiency of the two networks is similar. The main reason is that each edge needs to be computed in the BP algorithm, and the processing result of the former edge needs to be used by the next edge, which makes it impossible to use the edge as the parallel unit to carry out parallel processing. Only part of the operations of each edge can be processed in parallel, which greatly reduces the parallel ability and becomes the bottleneck to further improve the computing efficiency.

## 6. Conclusions

In this paper, the optimization and parallel research of the MRF network community partitioning algorithm for a specific network are carried out. Firstly, the principle of the algorithm is elaborated. Then, based on the analysis of the existing algorithms, some optimization strategies and rules are put forward, including the extraction of variables and operations from inner loops to outer loops, the merging of related operations of loops, the removal of redundant loops, and the split of loops. In order to achieve better parallelism, OpenMP parallel computing is realized by reversing the order of inner and outer loops. The influence of the density of network edges on the efficiency of the algorithm is analyzed in this paper, that is, the higher the density of network edges, the slower the operation speed, the lower the operation efficiency, and the worse the effect of optimization and parallel operation. On the contrary, the rarer the density of network edges, the faster the computing, the higher the operation efficiency, and the better the effect of optimization and parallel operation. In this paper, the algorithm is applied to the module partition of Alzheimer's disease gene data. The experimental results show that the efficiency of the algorithm is greatly improved after optimization and parallel processing. In particular, most of the optimization strategies and rules proposed in this algorithm can be extended to general situations, so they have good practical significance.

## References

1. S. Fortunato and D. Hric, "Community Detection in Networks: A User Guide," *Physics Reports*, Vol. 659, pp. 1-44, July 2016
2. D. Bernardes, M. Diaby, R. Fournier, F. FogelmanSouli é and E. Viennet, "A Social Formalism and Survey for Recommender Systems," *ACM SIGKDD Explorations Newsletter*, Vol. 16, No. 2, pp. 20-37, May 2015
3. S. Pool, F. Bonchi, and M. Leeuwen, "Description-Driven Community Detection," *ACM Transactions on Intelligent Systems and Technology*, Vol. 5, No. 2, pp. 1-28, April 2014
4. M. Girvan and M. E. J. Newman, "Community Structure in Social and Biological Networks," *Proceedings of the National Academy of Sciences*, Vol. 99, No. 12, pp. 7821-7826, 2002
5. M. E. J. Newman and M. Girvan, "Finding and Evaluating Community Structure in Networks," *Physical Review E*, Vol. 69, No. 2, pp. 026113, 2004
6. U. N. Raghavan, R. Albert, and S. Kumara, "Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks," *Physical Review E*, Vol. 76, No. 3, pp. 036106, 2007
7. M. Rosvall and C. T. Bergstrom, "Maps of Random Walks on Complex Networks Reveal Community Structure," *Proceedings of the National Academy of Sciences*, Vol. 105, No. 4, pp. 1118-1123, 2008
8. Y. Li, K. He, D. Bindel, and J. E. Hopcroft, "Uncovering the Small Community Structure in Large Networks: A Local Spectral Approach," in *Proceedings of the 24th International World Wide Web Conference*, pp. 658-668, New York, USA,

September 2015

9. T. Martin, B. Ball, and M. E. J. Newman, "Structural Inference for Uncertain Networks," *Physical Review E*, Vol. 93, No. 1, pp. 012306, 2016
10. S. Nowozin and C. H. Lampert, "Structured Learning and Prediction in Computer Vision," *Foundations and Trends in Computer Graphics and Vision*, Vol. 6, No. 3-4, pp. 185-365, March 2011
11. G. Sun, F. Lang, and Y. Xue, "Chinese Chunking Method based on Conditional Random Fields and Semantic Classes," *Journal of Harbin Institute of Technology*, Vol. 43, No. 7, pp. 135-139, July 2011
12. D. X. He, X. X. You, Z. Y. Feng, D. Jin, X. Yang, and W. X. Zhang, "A Network-Specific Markov Random Field Approach to Community Detection," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pp. 306-313, New Orleans, Louisiana USA, February 2018
13. A. Decelle, F. Krzakala, C. Moore, and L. Zdeborov, "Inference and Phase Transitions in the Detection of Modules in Sparse Networks," *Physical Review Letters*, Vol. 107, No. 6, pp. 065701, 2011
14. S. P. Huang, Z. L. Su, X. J. Yue, and X. L. Deng, "Feature Template-based Parallel Computation Technique for Conditional Random Fields," *Computer Science and Application*, Vol. 3, No. 5, pp. 251-256, August 2013

**Jun Lu** is a professor and master's supervisor in the College of Computer Science and Technology at Heilongjiang University. Her research interests include computational biology, data coding, parallel computing, and machine learning.

**Yuanzhong Zhang**, Male, Jan 25, 1992. He is a graduate student in the College of Computer Science and Technology, Heilongjiang University. Research Interests include parallel computing and machine learning.