

# An Implement of Parallel Module Network Learning Algorithm on Distributed Memory Multiprocessors\*

Long Liu<sup>1</sup>, Wei Hu<sup>2</sup>, Chunrong Lai<sup>2</sup>, Hong-shan Jiang<sup>1</sup>,

Wenguang Chen<sup>1</sup>, Weimin Zheng<sup>1</sup> and Yimin Zhang<sup>2</sup>

<sup>1</sup>Department of Computer Science and Technology,  
Tsinghua University, Beijing, P.R. China

<sup>2</sup>Intel China Research Center,  
Beijing, P.R. China

**Abstract.** As an extension of Bayesian Network, Module Network is good for dealing with problems where there are many variables but only a small set of data available. However it's still time-consuming to learn a Module Network. In this paper, we propose a parallel implementation of the Module Network learning algorithm based on the message passing model. In order to solve the load-imbalance problem introduced by either result caching or intrinsic computation, we propose a grouping strategy, which groups computations by modules and then distributed them cyclically. We tested our algorithm on eight 4-way Intel Xeon multiprocessors. Speedups of 29.26 on 32 processors have been observed. The result shows that our algorithm is effective.

**Keywords:** Module Network, Parallelization, MPI, Cache, Load balance

## 1. Introduction

In the past several years, the Bayesian Network[1,2,3] was widely applied in bioinformatics[4], speech processing[5], text mining[6], coding[7] and so on to learn dependency structure among variables, for example, to discover the relationship of genes from the gene expression data generated by microarray. In many cases of real world, the problem domain is so complex that it contains thousands of variables. But due to the expensive cost to acquire the data, maybe there are only a few hundred instances available. In these complex systems, the amount of data is rarely enough to robustly learn a model of the underlying distribution: statistical noise is likely to lead to spurious dependencies in a model that significantly over fit the data. To make up the defect of the regular Bayesian Network, an extended version of it, namely Module Network[8], was introduced.

---

\* This project is supported by NSFC 60273007 and Intel Corp.

Module Network is an extension of Bayesian Network to deal with problems where there are many variables but less data available. Module Network assumes that variables can be partitioned into sets and the variables within each set have a similar set of dependencies and therefore exhibit a similar behavior. For example, many genes in a cell are organized into modules, in which sets of genes required for the same biological function or response are co-regulated by the same inputs in order to coordinate their joint activity. As another example, when reasoning about thousands of NASDAQ stocks, entire sectors of stocks often respond together to sector-influencing factors.

Module Network combines the concept of clustering and probabilistic model that can be expected employed in wide range of real applications, such as finance, marketing etc. For example, customers can be grouped into modules and each module denotes different consumption characteristics. Currently it has been successfully used in Gene regulatory analysis [9,10].

Although a Module Network can significantly reduce the complexity of model space and the number of parameters compared with its corresponding Bayesian Network, it is still a computation-intensive and time-consuming process to learn a Module Network. It may take several days even a few months to get the final result, depending on the scale of given problem. So it is necessary to look to parallel computing.

In this paper, as our main contribution, we propose a parallel Module Network algorithm using the MPI programming model[11,12] and analytically and empirically validate our parallelization strategy. We study the performance of the parallel Module Network on eight 4-way Intel Xeon multiprocessor systems. To the best of our knowledge, a parallel implementation of the Module Network algorithm has not been reported in the literature.

We now briefly outline the remainder of the paper, and summarize our results. In Section 2, we present the Module Network algorithm and carefully analyze its computational complexity. In Section 3, based on above analysis, we introduce a parallel Module Network algorithm using MPI programming model. In Section 4, we list the experimental results on eight 4-way Intel Xeon multiprocessor systems. We analytically show that the speedup and scaleup of our algorithm approach the optimal as the number of nodes and modules increase. The final section presents the conclusions and the future work.

## 2. The Sequential Module Network Learning Algorithm

A Module Network is an extension of Bayesian Network. It can be viewed simply as a Bayesian Network in which variables in the same module share parents and parameters. In a Module Network, variables are partitioned into modules; variables in a same module share same behaviors. Each module connects with other modules by edges, which start from one node in the module and end at another module. Given an edge from node A to module B, then node A is one parent of module B. Combination

of a module and one of its parents is called a family.

Learning a Module Network is to find a Module Network which fits the given evidences best. It involves searching over two combinatorial spaces simultaneously—the space of structures and the space of module assignments. The structure of a Module Network decides the parent variables of a module and the module assignment decides the variables in the module. The Module Network learning algorithm uses an iterative approach with two steps: 1) Optimize the dependency structure according to the current assignment function. 2) Optimize the assignment function given the current dependency structure. We present this algorithm in Figure 1, and intuitively explain it below:

- 1) **Initialization.** Set the initial Module Network, where all nodes are assigned into modules and no edge exists.
- 2) **Structure Search Step.** This step learns the structure assuming that the assignment is fixed. It involves a search over the space of dependency structures. For each neighbor of current Module Network, compute the score increment of Network based on the score of the current Module Network, and then find the maximum. If the score increment is higher than the low threshold, update the network structure by setting this neighbor as the current Module Network. Here a neighbor is such a Module Network-it can be transformed from the current Module Network by adding, removing or reversing one and only one edge. A neighbor should be a legal one, that is, it should be a directed acyclic graph.
- 3) **Module Assignment Search Step.** Assuming that the module network structure is given, this step learns a new assignment function from data. For each neighbor of current Module Network, compute the score increment of Network based on the score of the current Module Network, and then find the maximum. If the score increment is higher than the low threshold, update the network assignments by setting this neighbor as the current Module Network. Here a neighbor is such a Module Network-it can be transformed from the current Module Network by moving one and only one node from one module to another.
- 4) **Convergence Condition.** Repeat steps 2 and 3, until no more score improvement can be obtained. Then the converged structure and assignment construct the final Module Network.

```

1: MN ← Initial Module Network;
2: do
3: {
4:   for (m1 = 0; m1 < M; m1++)
5:     for (n = 0; n < N; n++) {
6:       MN' ← Neighbor (MN, m1, n);
7:       Δs' ← Score (MN') - Score (MN);
8:       if (Δs' > Δs) {
9:         Δs ← Δs'; m' ← m1; n' ← n;
10:      }
11:    }
12:   if (Accept (Δs)) MN ← Neighbor (MN, m', n');
13:   for (n = 0; n < N; n++) {

```

```

14:   m1 ← Module (n) ;
15:   for (m2=0 ; m2<M ; m2++) {
16:     MN' ← Neighbor (MN, m1, m2, n) ;
17:     Δs' ← Score (MN') - Score (MN) ;
18:     if (Δs' > Δs) {
19:       Δs ← Δs' ; m1' ← m1 ; m2' ← m2 ; n' ← n ;
20:     }
21:   }
22: }
23: if (Accept (Δs)) MN ← Neighbor (MN, m1' , m2' , n) ;
24: }until (Convergence) ;

```

**Fig. 1.** The sequential Module Network learning algorithm

We now analyze the computational complexity of the sequential implementation of the Module Network learning algorithm in Figure 1.

Though we cannot give an accurate conclusion about the time cost of iteration due to the complexity of real graph, we can estimate an approximate one. Carefully examining Figure 1, we can learn that learning dependency structure requires  $aMNC$  flops (floating point operation) where  $M$  denotes the number of modules,  $N$  denotes the number of nodes,  $C$  denotes the flops for computing increment of score and  $a$  is a coefficient. Also, learning network assignments requires  $bMNC$  flops where  $b$  is a coefficient. Put these together, we can estimate the computational complexity of the sequential implementation of the Module Network learning algorithm as

$$T_1 \sim cMNC \cdot I \cdot T^{\text{flop}}, \quad (1)$$

Where  $c$  is a coefficient, and  $I$  denotes the number of iterations and  $T^{\text{flop}}$  denotes the time for a floating-point operation.

From (1), we learn that the time to learn a Module Network is relevant to the number of modules and the number of nodes.

The sequential Module Network Learning Algorithm has been implemented in the Probabilistic network library (PNL) [13]. The PNL library, developed by INTEL Corporation, includes a comprehensive set of inference algorithms for Bayesian and Markov networks.

### 3. Parallel Module Network learning Algorithm

#### 3.1 Parallel Algorithm Design

Before parallelizing the Module Network learning algorithm, we optimized the sequential code first. A result cache is used for computation reuse. When we compute

the score increment of each neighbor, the results were stored to the result cache so that they can be used in the next iteration step and need not to be recomputed if both the module and its parents remain unchanged. A cache item is bound with one distinct combination of one module and its parents. If the nodes in the module or its parents have been changed, the score increment would have to be recomputed and stored in a different cache item. In one iteration step, at most three modules or their parents can be changed. When the dependency structure is adjusted, parents of one module will be changed. When the network assignments are adjusted, two modules will be changed since one certain node moved between these two modules. So with the help of result cache, at most three modules need to be recomputed in one iteration and all the others can be obtained from the cache directly.

The use of result cache greatly improves the performance of the sequential dependency structure learning algorithm, which reduces the execution time to one third of the original sequential code. However, it also makes the workload of learning dependency structure more imbalanced among iterations. An iteration with all scores in the cache would be much faster than another iteration that requires to recomputed scores for modules. In order to have satisfactory parallel speedups, we must take the imbalance among the iterations into consideration.

There are 2 levels of potential parallelism in the Module Network learning algorithm: module level and node level.

First we consider whether it is acceptable to parallelize the Module Network in module level. For a module level parallelism, tasks are partitioned by the combination of the module that the start nodes belong to and the end modules of the edges. We find that under this strategy the workload of each processor would be imbalanced, since the amount of computations associated with every module depends on the nodes inside the module and the structure of the network.

So we decide to choose the node level parallelism strategy, which could be more load-balanced potentially. But the problem is still not solved yet. If we simply assign computations to processors in a continuous way, it still results in imbalanced workload.

The calculation of one score increment may vary widely because:

(1) Modules to which it is related have different amount of nodes and parents, and cache associated with it is unavailable. In this situation, the score increments have to be recomputed completely. The more nodes and parents one module owns, the more time computations associated with the module will cost.

(2) Two modules have same amount of nodes and parents, but the cache is available for only one module. Computation will be much slower without the cache.

Because of the above reasons, if we just simply assign these computations to processors one by one disregarding the differences among them, we cannot ensure that the workload for each processor is balanced.

But we noticed that two computations would cost nearly the same time if they belong to the same module and the caches associated with the module are all available or all unavailable. We also observed that there is no order constraint among all computations in one iteration step, so they can be executed in free order. Therefore we can group these computations according to the modules they belong to before they are assigned to processors. Then the grouped computations are divided cyclically into

processors to achieve an almost balanced partition (see [14] for more knowledge about parallel hill-climbing algorithms). To implement this strategy, we modified line 4 of the sequential Module Network learning algorithm as the following:

```

4. for ( $m_2=0; m_2 < M; m_2++$ )
5.   foreach ( $n$  in  $m_2$ ) {

```

Based on the above adjustment, we propose our parallelism algorithm as shown in Figure 2. In each search step, the network modules are distributed to different processors. Each processor evaluates DAG neighbors by adding, deleting or reversing edges to assigned modules of current structure. Then scores of all neighbors are collected and the highest one is selected.

```

1: PID ← MPI_comm_rank();
2: P ← MPI_comm_size();
3: MN ← Initial Module Network;
4:  $m_1 \leftarrow 0, m_2 \leftarrow 0, n \leftarrow \text{PID}$ ;
5: ( $m_1, m_2, n$ ) ← Redirect ( $m_1, m_2, n$ );
6: do {
7:   for ( $; m_1 < M;$ ) {
8:      $MN' \leftarrow \text{Neighbor}(MN, m_1, n)$ ;
9:      $\Delta s' \leftarrow \text{Score}(MN') - \text{Score}(MN)$ ;
10:    if ( $\Delta s' > \Delta s$ ) {
11:       $\Delta s^* \leftarrow \Delta s'; m_1^* \leftarrow m_1; n^* \leftarrow n$ ;
12:    }
13:    ( $m_1, m_2, n$ ) ← Redirect ( $m_1, m_2, n+P$ );
14:  }
15: Communicate to get global max score;
16:  $\Delta s \leftarrow \text{Max}(\Delta s^*); m \leftarrow m^*; n \leftarrow n^*$ ;
17: if (Accept ( $\Delta s$ ))  $MN \leftarrow \text{Neighbor}(MN, m', n')$ ;
18: for ( $n = \text{PID}; n < N; n += P$ ) {
19:    $m_1 \leftarrow \text{Module}(n)$ ;
20:   for ( $m = 0; m < M; m++$ ) {
21:      $MN' \leftarrow \text{Neighbor}(MN, m_1, m_2, n)$ ;
22:      $\Delta s' \leftarrow \text{Score}(MN') - \text{Score}(MN)$ ;
23:     if ( $\Delta s' > \Delta s$ ) {
24:        $\Delta s^* \leftarrow \Delta s'; m_1^* \leftarrow m_1; m_2^* \leftarrow m_2; n^* \leftarrow n$ ;
25:     }
26:   }
27: }
28: Communicate to get global max score;
29:  $\Delta s \leftarrow \text{Max}(\Delta s^*); m_1' \leftarrow m_1^*; m_2' \leftarrow m_2^*; n' \leftarrow n^*$ ;
30: if (Accept ( $\Delta s$ ))  $MN \leftarrow \text{Neighbor}(MN, m_1', m_2', n')$ ;
31: }until (Convergence);

```

**Fig. 2.** The parallel Module Network learning algorithm

### 3.2 Computational Complexity Analysis

From Figure 2, we can learn that each iteration of our parallel Module Network learning algorithm consists of an asynchronous computation phase followed by a synchronous communication phase. For computation tasks have been divided evenly to  $P$  processors, we expect that the computation time for each processor to decrease to

$$T_p = T_l/P \sim cMNC \cdot I \cdot T^{\text{flop}} / P. \quad (2)$$

Also, for each processor, in each iteration, only a local maximum increment of score is computed. So before all the processors update the dependency structure or assignments of the Module Network, it's necessary for them to communicate with each other to get the global maximum increment of score and the indispensable information for adjusting the network. We can estimate the communication time for the parallel Module Network learning algorithm to be

$$T_s \sim I \cdot (T^{\text{reduce}} + T^{\text{bcast}}), \quad (3)$$

where  $T^{\text{reduce}}$  denotes the time required by ‘‘MPI\_Allreduce’’ and  $T^{\text{bcast}}$  denotes the time required by ‘‘MPI\_Bcast’’.

We now can combine (2) and (3) to estimate the computational complexity of the parallel Module Network learning algorithm as

$$T_2 = T_p + T_s \sim cMNC \cdot I \cdot T^{\text{flop}} / P + I \cdot (T^{\text{reduce}} + T^{\text{bcast}}). \quad (4)$$

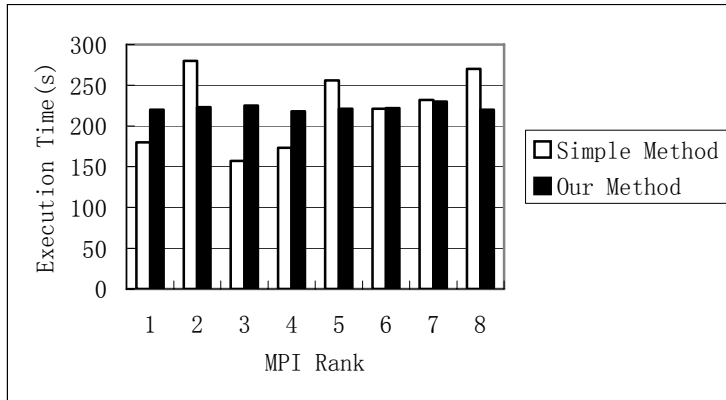
## 4. Performance and Scalability Analysis

Experiments are carried out on an 8-node cluster, each node featuring 4-way SMP of Pentium-III Xeon 700MHz CPU and 1GB RAM. The software environment is Linux 2.4.20 and mpich-1.2.5. All nodes are connected by 100Mb/s switch. Time were measured with the function ‘‘MPI\_Wtime()’’ and the I/O time were excluded from the execution time. Ten measurements were made for each test case and the average number is reported.

We generated four test data sets. The smallest one contains 1,110 nodes that are assigned to 37 modules. To ensure that the network structure is imbalanced, the

number of nodes in each module varies from 1 to 157 randomly. The three other data sets are 2,220 nodes with 74 modules, 4440 nodes with 148 modules and 8880 nodes with 296 modules.

We measured the parallel execution time of our algorithm for the largest data set on 8 processors. For comparison purpose, we have also implemented a naïve parallel algorithm of Module Network in which computations are simply assigned to processors without being organized according to the modules they belong to. We call it the *simple method*. The parallel execution time of the simple method also was measured. The results are shown in Figure 3.

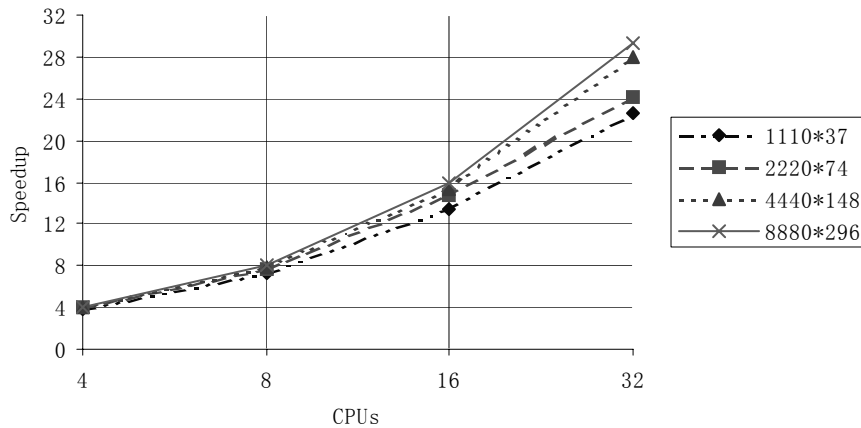


**Fig.3.** Execution time for 8880 nodes, 296 modules and 8 processors

In Figure 3, the execution time of each MPI rank is shown for both the simple method and our method. It is noticed that the execution time of the simple method in different MPI ranks varies widely. It indicates that the simple method suffers from load imbalance seriously. To the contrary, the execution time of our parallel method is much more stable in different MPI ranks, which verifies the effectiveness of our strategy to solve the load imbalance problem. .

The speedup factors measured for 4, 8, 16 and 32 processors for the different data sets are shown in Figure 4.

Speedup of 22.67- 29.26 on 32 processors is obtained for various dataset. For the largest data set we tested, i.e. 8880 nodes with 296 modules, the speedup is 29.26 on 32 processors, which is almost liner.



**Fig. 4.** Scalability of the parallel Module Network on different data sets.

## 5. Conclusion and Future Work

This paper described a parallel implementation for the Module Network learning algorithm. In order to solve the load-imbalance problem introduced by either result caching or intrinsic computation, we proposed to group computations by modules and then distributed them cyclically. The experiments on a cluster show satisfactory results. Speedups of 22.67-29.26 on 32 processors are obtained for different data sets on eight 4-way Intel PIII Xeon servers, which are connected with 100Mbps fast Ethernet. We believe the grouping strategy used in this work is also beneficial to other parallel applications that are suffered from the load imbalance problem.

## References

- [1] K. Murphy, A Brief Introduction to Graphical Models and Bayesian Networks, 1998. [Online]. Available:<http://www.ai.mit.edu/~murphyk/Bayes/bnintro.html>
- [2] D. Heckerman. A tutorial on learning with Bayesian networks. In M. I. Jordan, ed., Learning in Graphical Models. 1998.
- [3] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Mach. Learn.*, 20:197–243,1995.
- [4] D. S. Salzberg, S. and S. Kasif, Computational Methods in Molecular Biology. Elsevier Publ, 1998.
- [5] J. Bilmes, “Graphical models and automatic speech recognition,” in Mathematical Foundations of Speech and Language Processing, ser. Institute of Mathematical Analysis Volumes in Mathematics Series. Springer-Verlag, 2003.
- [6] B. Taskar, E. Segal and D. Koller, “Probabilistic clustering in relational data,” in Seventeenth International Joint Conference on Artificial Intelligence, Seattle, Washington, 8 2001, pp. 870–876.

- [7] B. J. Frey, Graphical Models for Machine Learning and Digital Communication. MIT Press: Cambridge, MA, 1998.
- [8] Learning Module Networks. E. Segal, D. Pe'er, A. Regev, D. Koller, and N. Friedman. In Proc. Nineteenth Conf. on Uncertainty in Artificial Intelligence (UAI), 2003.
- [9] Learning module networks from genome-wide location and expression data. Xu X, Wang L, Ding D. FEBS Lett. 2004 Dec 17;578(3):297-304.
- [10] Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data. Eran Segal, Michael Shapira, Aviv Regev, Dana Pe'er, David Botstein, Daphne Koller and Nir Friedman. Nature Genetics 34, 166 - 176 (2003).
- [11] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. The MIT Press, Cambridge, MA (1996)
- [12] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. The MIT Press, Cambridge, MA (1997)
- [13] "Pnl: Probabilistic network library." [Online]. Available: <http://www.intel.com/research/mrl/pnl/>
- [14] L.Wai et al, "A Parallel Learning Algorithm for Bayesian Inference Networks", IEEE Transactions on Knowledge and Data Engineering, Jan. 2002, pp 93-105