Parallelization of Module Network Structure Learning and Performance Tuning on SMP

Hongshan Jiang[1], Chunrong Lai[2], Wenguang Chen[1],
Yurong Chen[2], Wei Hu[2], Weimin Zheng[1], and Yimin Zhang[2]

[1]Tsinghua University
Dept. of Computer Science
Beijing, 100084 China
jhs03@mails.tsinghua.edu.cn
{cwg, zwm-dcs}@tsinghua.edu.cn

[2]Intel China Research Center Ltd.
9/F, Raycom Infotech Park A, Zhong Guan Cun
Beijing, 100080 China
{chunrong.lai, yurong.chen, wei.hu,
yimin.zhang}@intel.com

Abstract

As an extension of Bayesian network, module network is an appropriate model for inferring causal network of a mass of variables from insufficient evidences. However learning such a model is still a time-consuming process. In this paper, we propose a parallel implementation of module network learning algorithm using OpenMP. We propose a static task partitioning strategy which distributes sub-search-spaces over worker threads to get the tradeoff between load-balance and software-cache-contention. To overcome performance penalties derived from shared-memory contention, we adopt several optimization techniques such as memory pre-allocation, memory alignment and static function usage. These optimizations have different patterns of influence on the sequential performance and the parallel speedup. Experiments validate the effectiveness of these optimizations. For a 2,200 nodes dataset, they enhance the parallel speedup up to 88%, together with a 2X sequential performance improvement. With resource contentions reduced, workload imbalance becomes the main hurdle to parallel scalability and the program behaviors more stable in various platforms.

1. Introduction

In recent years, Bayesian networks [10] have been widely applied in various fields such as bioinformatics, speech processing and text mining to represent probabilistic influences among stochastic variables, for instance to identify gene regulatory networks from microarray data. In real world applications, it is often the case that the problem domain is so complex that it contains a mass of variables while there are only a few instances available due to expensive cost of acquiring data. In these cases, the amount of data is insufficient to learn the underlying distribution: statistical noises tend to lead spurious dependencies that significantly overfit the data. To fit with these cases, an extension of traditional Bayesian network, namely module network [11], was introduced.

Module network can be viewed as a Bayesian network in which variables in the same module share the same parents and conditional probability distribution. A real world example corresponding to this model is that many genes in a cell are co-regulated by the same factors and exhibit the same expression pattern. Module network introduces the clustering concept into regular Bayesian network and make up its aforementioned defect. Currently it has been successfully used in Gene regulatory analysis [12].

Compared with traditional Bayesian network structure learning [10, 8], module network structure learning has a further structure to learn, namely module assignment. Once the assignment is determined, learning dependency structure of a module network is like that of a Bayesian network except that the targets are changed from nodes to modules. Module network significantly reduces the model complexity and the number of parameters, however learning such a model is still a computation-intensive and time-consuming process.

[0]Supported by Intel Corporation, the National Natural Science Foundation of China under Grant No. 60273007 and the ChinaGrid.

A parallel learning program of module network on distributed memory multiprocessor systems was recently implemented and optimized [9], but as far as we know the parallel version for shared-memory multiprocessor systems has not appeared. We see that the behaviors of parallel programs on shared-memory multi-processors are more difficult to understand. It is mainly due to the facts that there are more types of resource contentions in SMP and communications between processors are implicit. Because of the abundance of SMP systems in recent years and the trend of on-chip multi-core systems in the future [7], implementing such a hotspot application on SMP systems and analyze its characteristics is of great importance.

In this paper, as our main contribution, we propose a parallel module network learning algorithm with OpenMP [4] and present a set of recipes to cope with performance penalties that arise from resource contentions. Experiments show that these optimizations have sound effectiveness.

The rest of this paper is organized as follows. In section 2, we describe a hill climbing algorithm for learning module network. In section 3, the parallelized algorithm is proposed. In section 4, measurements of hotspots and thread profiles by Intel® VTune™ [1] together with corresponding analysis are presented. In section 5, we give some optimization techniques to deal with memory sharing. Experiment results are given and discussed in section 6. In the last section, we give the summary and future works.

2. Module Network Learning Algorithm

Module network structure learning is an optimization problem, in which a very large search space must be explored to find the optimal solution. Because a brutal search will lead to super-exponential computational complexity, we use a greedy hill climbing algorithm to find a local optimal solution.

The hill climbing algorithm first clusters all nodes into modules, none of which has parent nodes. This initial module network model is treated as the start point in the search space (solution space). The algorithm searches all the nearest neighbors of the start point in the search space, then selects the neighbor that has the highest score as the new start point. This procedure iterates until no neighbor has higher score than the current start point (i.e., it has reached a local optimal). Here the "neighbors" of a module network model are of two kinds: dependency neighbors and assignment neighbors. The former can be generated from the current model by adding or deleting a single arc, subject to the acyclic constraint; the latter can be generated from the current model by moving a node from a module to another, subject to the acyclic constraint and without changing module count.

Learning dependency structure and learning module assignment are equally important, so every iteration step combines searches for dependency neighbors and searches for assignment neighbors. In our implementation, these searches are organized as two iterative phases. The algorithm is presented in Figure 1.

Bayesian Information Criteria (BIC) [8] score metric is used in the algorithm for scoring a module network model. It equals to the log likelihood of the data subtract the penalty of the model's complexity. A model M has BIC score with dataset D as below:

$$Score(M, D) = log(P(D|M)) - 0.5 \times log(E) \times C(M) \quad (1)$$

where P(D|M) is the probability that data D is generated from model M; E is the number of evidences in the dataset D, and C(M) is a constant that describes the complexity of model M. From the formula, we can see that the main computation to get a score is to compute the probability P(D|M). According to the conditional

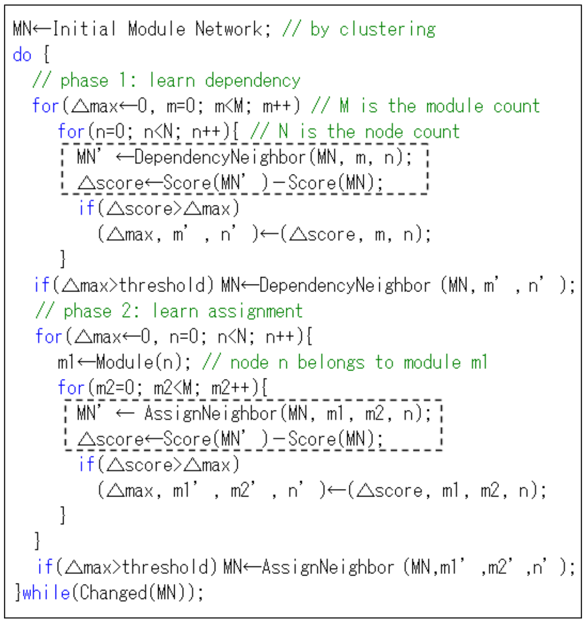

```
MN←Initial Module Network; // by clustering
do {
  // phase 1: learn dependency
  for(△max←0, m=0; m<M; m++) // M is the module count
    for(n=0; n<N; n++){ // N is the node count
     MN' ←DependencyNeighbor(MN, m, n);
     △score←Score(MN' )−Score(MN);
      if(△score>△max)
        (△max, m' , n' )←(△score, m, n);
    }
  if(△max>threshold) MN←DependencyNeighbor (MN, m' , n' );
  // phase 2: learn assignment
  for(△max←0, n=0; n<N; n++){
    m1←Module(n); // node n belongs to module m1
    for(m2=0; m2<M; m2++){
     MN' ← AssignNeighbor(MN, m1, m2, n);
     △score←Score(MN' )−Score(MN);
      if(△score>△max)
        (△max, m1' , m2' , n' )←(△score, m1, m2, n);
    }
  }
  if(△max>threshold) MN←AssignNeighbor (MN,m1' ,m2' ,n' );
}while(Changed(MN));
```

Figure 1. Sequential algorithm

independence in Bayesian network, the total score of a model can be decomposed into the sum of its families' scores:

$$Score(M, D) = \sum_i Score(Family_i, D) \quad (2)$$

Different from that in traditional Bayesian network model, here a family consists of a module (a set of child nodes) and its parent nodes. Since each family's score will not change once it has been computed out, we use a cache to store each family's score after the score has been computed. Consequently when we need to compute a family's score, we look for it in the cache first. If cache hit, we load score from cache directly; otherwise, when encounter a new family, we compute the score from scratch and save it into cache for possible future use.

From Equation 2, we infer that the delta score between a neighbor and the current model equals to the delta score of the changed family. Therefore the problem of computing score of each neighbor model is changed to computing scores of neighbor families.

Analyze the computational complexity of this algorithm is pretty hard due to the search path is tightly correlated to the training dataset and the initial clustering. In addition, using of score caches complicated the situation. In [9] an approximate estimation is given.

An implementation of the sequential module network learning algorithm is available in the Probabilistic Network Library (PNL) [2]. It constructs the start point of our work.

3. Workload Parallelization

To parallelize the hill-climbing structure learning algorithm, we first identify the most computational intensive modules, finding parallel opportunities within it. By using VTune$^{\mathrm{TM}}$ performance analyzer, we find most of the execution time is spent on computing scores of neighbor families. On the other hand, each family score can be computed independently. Therefore each family score computation becomes a single workload unit for parallelization.

Neglecting score cache factor, the workload distribution of family score computations is inherently not even. Computing scores of neighbor families from the same family have close workloads while computing scores of neighbor families from different families may vary greatly in workloads.

Unfortunately the introduction of score caches exaggerates this workload imbalance. In all rounds except the first round, only newly encountered neighbor families need to be computed their scores from scratch, other neighbor families' scores can be obtained from score caches. The newly encountered neighbor families include those neighbors of the just changed family of current model in the previous round and a few neighbors of the unchanged family due to the possible release of some acyclic constraint.

In our implementation, score caches are saved as sparse matrix elements and are indexed by family parents' numbers. Family children's (nodes inside the module) numbers are not used as indices for two reasons. The first is that higher index dimensions lead to more performance penalty when retrieving content from sparse matrix. The second is that computation of delta score of changing children of a family has much lower complexity than that of changing parents of a family. As a result, every sparse matrix is bound to a module. In addition, the score caches are only used in the dependency-learning phase which makes choice by altering parents. In assignment-learning phase, delta score of neighbor families are computed directly.

Now we talk about issues related to parallelization.

Granularity. If we choose to partition each family score computation into tasks, the granularity is fine but the parallelism and synchronization overhead become heavy. Otherwise choosing to parallelize with module granularity will lead to intolerable workload imbalance. As a trade-off, we treat each family score computation as a single workload unit in dependency-learning phase, while treat delta score computations of those neighbor families derived from moving one node as a single workload unit in assignment-learning phase.

Scheduling. In dependency-learning phase we notice that resource contention caused by shared-memory usage hinder us from using a workload balanced scheduling strategy. Sparse matrices for caching scores are implemented as hash tables with linked lists, which makes each sparse matrix a critical resource. But from the second round of learning process, from-scratch computations concentrate in one module (i.e. related to the same sparse matrix). So synchronization overhead for using critical section will be heavy. Therefore we make a copy of sparse matrices for every worker thread. At this point, we choose static scheduling strategy so that the search space mapped in each spare matrix can be divided into fixed nonoverlapped sub-spaces which are thereafter assigned to different worker threads. To use the common acyclic judgment, actually we group nodes according to the module they belong to and then schedule workloads statically. This makes things more complex. But as in one round of the iteration process at most two modules change their components by one node, the nonoverlapping feature of sub-spaces is maintained in an approximate way. In assignment-learning phase we simply use a dynamic scheduling strategy to gain good workload balance.

Synchronization. As OpenMP only support simple reduction variable, we set Δmax and related indices as threadprivate variables. Each worker thread compute its local maximum score in Δmax, then in the critical section at the end of each phase, the global maximum score is collected.

Figure 2 illustrates the parallelized algorithm.

```
#pragma omp threadprivate(△max, m' , m" , n' )
MN←Initial Module Network; // by clustering
do {
#pragma omp parallel private(m2,△score)
  for(△max←0, m2=0; m2<M; m2++) // M is the module count
#pragma omp for private(m1,n) schedule(static) nowait
    for(m1=0; m1<M; m1++)
      foreach(n in m1){
       MN' ←StructNeighbor(MN, m2, n);
       △score←Score(MN' )−Score(MN);
        if(△score>△max) (△max, m' ,n' )←(△score, m2, n);
      }
#pragma omp critical(GET_MAX_SCORE)
  reduce (△max, m' , n' ); // to get the global maximum
  if(△max>threshold) MN←StructNeighbor (MN, m' , n' );
#pragma omp parallel
#pragma omp for private(n,m1,m2,△score) schedule(dynamic)
  for(△max←0, n=0; n<N; n++){
    m1←Module(n); // node n belongs to module m1
    for(m2=0; m2<M; m2++){
      MN' ← AssignNeighbor(MN, m1, m2, n);
      △score←Score(MN' )−Score(MN);
      if(△score>△max)
        (△max, m' , m" , n' )←(△score, m1, m2, n);
    }
  }
#pragma omp critical(GET_MAX_SCORE)
  reduce (△max, m' , m" , n' ); // to get the global maximum
  if(△max>threshold) MN←AssignNeighbor (MN, m' , m" , n);
}while(Changed(MN));
```

Figure 2. Parallel algorithm using OpenMP

4. First Round Performance Analysis

4.1. Baseline Program and Hardware

The first group of issues before the performance tuning process are the baseline construction and hardware settings.

According to the descriptions in section 3 we developed a package, which includes high level optimizations like common sub-expression elevation and redundancy elimination [5]. We compile this package with options "-O3 -openmp", hence constructed our baseline program.

The hardware platforms are two Xeon^{TM} SMP machines. One is a 4-way multiprocessor system named QP (quad-processors) machine, the other is a Unisys-Es7000 system which consists of 16 processors. Their key characteristics are listed in Table 1.

Processor prefetcher settings are important hardware settings which can influence application performance. Whether or not to set a prefetcher on should

Platform		QP-machine	Unisys-Es7000
Processor		4-way 2.8GHz	16-way 3.0GHz
	L1 d-cache	8KB, hit latency 2 cycles	8KB, hit latency 2 cycles
	L2 cache	512KB, hit latency $\sim$ 10 cycles	512KB, hit latency $\sim$ 10 cycles
	L3 cache	2MB, hit latency 30+ cycles	4MB, hit latency 30+ cycles
	L4 cache	None	32MB on-board, 300+ cycles
	FSB	400MHz	400MHz
Interconn.		FSB	Crossbar
Memory		2GB, peak bandwidth 2.1GB/s	8GB, peak bandwidth 3.2GB/s
OS		Linux 2.4.20smp	Linux 2.4.20smp

Table 1. Hardware configuration

be determined by memory access pattern of the target application. From analysis in section 3, we know that accessing score caches will lead to irregular memory access pattern, since score caches are implemented as items in hash table. On the other hand, when computing family score from scratch, the matrix access stride has regular pattern. Therefore we need some quantitative measurements to judge which has dominant impact.

In Intel's Pentium^{TM}4 and Xeon^{TM}processors, there are two hardware-based prefetchers which can prefetch data from memory to cache. The first one is called stride prefetcher which tries to stay some cache lines ahead of current data access locations, the other one is called adjacent line prefetcher which fetches the adjacent cache line (in same sector). These Prefetchers settings can be changed through BIOS options.

As anticipated, measurements show that prefetcher settings do impact system performance. Table 2 lists the running time metrics of the baseline program with different prefetcher settings on these two machines, which are normalized to the running time with 4-threads in the default setting of corresponding machines. The four-thread (4T) relative speedups are also listed in the table. It is seen that the benefit of the prefetchers decreases as processor numbers increase. And the best relative speedups are achieved if turning off the prefetchers. So we turn off both prefetchers in these machines for the following experiments.

After we determine the baseline program and hardware settings, it is seen that the parallel speedup varies greatly on different platforms and is not good enough on both of them. As the last line of Table 2 shows, the 4T speedup is 2.56X on the QP-machine and 1.74X on Unisys-es7000. It needs to be pointed out that the aw-

	QP-machine			Unisys-Es7000		
St./Ad.	seq.	Par.	Sp.	Seq.	Par.	Sp.
on/on	2.37	1.02	2.33	1.70	1.05	1.62
on/off	2.38	0.95	2.50	1.71	0.99	1.73
off/on	2.42	**1.00**	2.42	1.72	1.11	1.55
off/off	2.47	0.96	**2.56**	1.74	**1.00**	**1.74**

Table 2. Normalized running time and 4T speedups with prefetcher settings

ful performance is not result from the parallelization strategies. In fact, both theoretical analysis (in section 3) and profiling (in section 4.2) confirm that we have a relatively optimum version according to high-level task partition.

4.2. Thread Profile and Hotspot Functions

To gain the running metrics of our program and measure which parts influence the whole performance to what extent, we use VTune^{TM} analyzer, which consists of thread profiling and sampling profiling, to get thread profile and hotspot functions.

The thread profiling process creates thread profile which can be expected to give hints to understand or optimize the parallel speedup. The process is based on the instrumentation. The codes are in fact recompiled with -openmp_profile instead of -openmp. Instrument codes are added to the OpenMP pragma or even the Pthreads/Winthreads primitives. Finally by running those instrumented binary, VTune can give a report of run-time ratio of the parallel-part, sequential-part, load-imbalance, barrier as well as the synchronization percentage and the overheads ratio. In our experiments the thread profile only increases 5%∼8% overhead to the program. So that the gained thread profile of baseline program is trustworthy.

	Para. ratio	Seq. ratio	Imba. / barr.	Locks / Sync.	Over-heads
QP	94.0%	0.5%	4.1%	0.9%	0.4%
Uni.	90.1%	0.3%	6.5%	0.5%	0.3%

Table 3. Thread profile of the baseline

Table 3 lists the thread profiles. We found that when running in parallel most of the time spends on the parallel part. The most serious parallelism-limiting factor is the imbalance part that arises from the *LearnDependency* function because of the static scheduling strategy there. But the number 4.1% or 6.5% does not occupy a large ratio. At this way the parallel speedup should be much better than what we got. As we have seen that the aggregate running time increases much from uni-thread (UT) running to multi-threads (MT) running. It is highly possible that some operations run much slower in multi-processors configuration than in uni-processor configuration.

In the next step, we switch to VTune sampling process to get bottlenecks on run time. In this process, interrupts are generated on every interval of given number of hardware events based on hardware counters, and then samples are noted down on current PC. Thus the sampling process can give a run-time hotspot instructions distribution for interested events. When the binary is built with debug information, for example built with -g in gcc or -zi in Visual studio compiler or in icc compiler, the distribution can be seen in higher level such as function-level, class-level or module-level.

In Table 4, we list those hotspot functions which cost up to 90% of the running time on both machines with one thread and four threads, respectively. These hotspot functions can be divided into four modules, as Table 4 shows. The first module is *ModuleNet*, which contains the main part of functions for learning module network. Among them, function *ScoreCacheAccess* accesses score caches; function *ProcessingStatisticalData*, which include log computations, computes score from the sufficient statistics [8]. The second module is *Libpthread.so*, which deals with primitive operations of pthead. The third module is *Libc-2.3.2.so*, which is mainly about memory operations. The last module is *libguide.so*, which include those part of OpenMP library calls.

Neglecting *libguide.so*, which relates to OpenMP runtime environment, we can see from Table 4 that vector functions such as *push_back*, *erase*, *assign* etc. significantly increase their percentage as thread count increased. Other functions such as *ScoreCacheAccess* and *free* have the same tendency but less degree. The scalability problem in vector functions, *malloc* and *free* give us hints to think about the cost of dynamic memory allocation/release. While the prolonged accessing time of *ScoreCacheAccess* makes us to pay attention to memory layout [5].

4.3. Memory Hierarchy Statistics

For further assurance, we get the memory-hierarchy micro-architectural statistics with VTune^{TM} in Table 5. As this table shows, the Level 1 cache misses, FSB (front-side-bus) activities, 64KB aliasing, and DTLB (Data Translation Look-aside Buffer) page walks all increase from 1T to 4T. These data confirm that the memory behavior in MP (Multi-Processors) case is not

Module name / Function name		QP-1T	QP-4T	Uni.-1T	Uni.-4T
ModuleNet		48.6%	↑61.9%	46.4%	43.2%
	ScoreCacheAccess	2.7%	↑3.6%	3.9%	↑5.0%
	ProcessingStatisticalData	8.4%	2.8%	7.8%	3.2%
	Vector functions	2.0%	↑15.8%	1.7%	↑12.9%
Libpthread.so		32.2%	22.7%	31.2%	22.7%
Libc-2.3.2.so		10.0%	8.6%	11.4%	↑23.5%
	_malloc	3.8%	2.7%	4.3%	↑13.1%
	_free	2.3%	↑3.3%	3.7%	↑8.6%
Libguide.so			6.2%		9.6%

Table 4. The hotspot functions

as good as in UP (Uni-Processor) case. It gives the clue that memory contention of multi-processors may be the main bottleneck of workload scalability.

	QP-1T	QP-4T	Uni.-1T	Uni.-4T
Instru. Loads	36.5%	36.5%	36.5%	36.5%
L1 miss	2.4%	↑3.3%	3.3%	↑3.6%
L2 miss	7.0%	2.6%	5.1%	4.7%
FSB activities	0.6%	↑8.8%	0.5%	↑9.8%
DTLB walks	0.11%	↑0.13%	0.09%	↑0.13%
64KB aliasing	2.8%	↑4.7%	4.3%	↑4.6%

Table 5. Memory performance metrics

5. Memory Issues and Optimizations

The implementation of aforementioned parallel module network learning algorithm is based on PNL [2], which is implemented with C++ and STL (Standard Template Library) [3]. The *push_back/erase* methods of vectors in STL provide friendly interface to programmers but also implicitly use heap memory allocation/release. Due to increased competition for shared resources like the memory bandwidth and more interferes among multi-processors for cache coherence, excessive use of dynamic memory allocation/release leads to performance degrade on SMP. We used the HOARD [6] multiprocessor memory allocator trying to make those lock-free, but the performance is not improved. In the following, we will illuminate those memory issues that mostly influence the workload scalability of our program. However, it is out of the scope of this paper to describe how to optimize memory usage on SMP systems in general.

The first issue is the frequency of memory allocation and release. As memory is a shared resource in SMP system, allocating memory for multiple concurrent threads is inefficient. Also, frequent invocations to memory allocation and release cause memory fragmentation and random memory layout. Further more, the operations are more expensive when vectors need to expand allocated space. All of these limit workload scalability.

The optimizations adopted for this issue are memory pre-allocation and static function usage. Memory pre-allocation is applied for those frequently accessed buffers or vectors. At the beginning of a code region (usually a loop region), we allocate a memory block whose length reaches the maximum length needed to run in this region. At the end of the region, we release the memory block. This remarkably reduces the frequency of memory allocation and release.

Static functions are used as substitutes for virtual functions of an object. It can reduce the overheads of construction and destruction of an object. In addition, compared with invoking virtual functions, calling static function also saves the cost of virtual tables' access. As additional costs, few more parameters need to be transferred when invoking the static functions.

The second issue is memory layout. Memory layout problem is often caused by dynamic memory allocation/release. Compiler can perform memory alignment and padding automatically for global or local variables according to compile pragma in source code. But it can not automatically align memory for dynamically allocated memory yet. This may introduce several performance issues. Intel manuals mention that the split loads or blocked store forwarding penalties due to un-alignment access take the major part of sequential penalty. Here we point out two more kinds of penalties. One is false sharing[5], which means that if the buffers are not aligned by cache line granularity, i.e. many memory lines are each partitioned by more than one processor, there will be more cache invalidates for these falsely shared data. The other kind of penalties comes from the aliasing problem. Intel reports that the processors in the test platforms have an aliasing penalty when "64 KBytes aliasing"[1] exists. The reason is that the processor is doing way prediction based on that part of virtual address, so that different data with the same value for bit 0 through 15 of the linear address will compete for the same cache lines.

The optimization adopted for memory layout problem is manual memory alignment. After dynamic allocation of a large block, we compute a memory address closely after the gained memory handle, which is aligned according to the cache line size of the target

[1]The newer Intel processor has less aliasing penalties because there are only "4MB aliasing"

architecture.

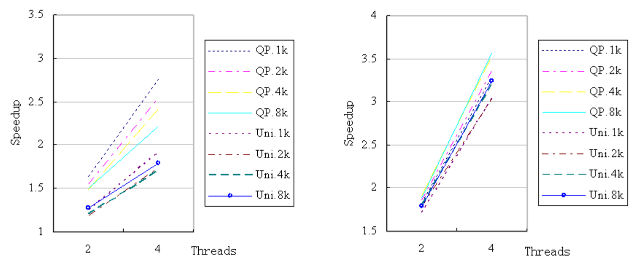
To estimate the impact of each optimization on final performance, we separate out the static-function-usage optimization and construct two optimized versions. The first one called MEMORY only uses memory pre-allocation and alignment. The second one called FINAL uses all of the aforementioned optimizations.

Although these optimizations are somewhat straightforward, they improve both the sequential performance and parallel speedup significantly. Also as we will describe in Section 6, these optimizations have different patterns of influence on the sequential performance and the parallel speedup.

6. Experiment Results

Experiments were carried out on the machines as described in Table 1. Hardware settings and compiler options are set as section 4.1 mentioned. The wall time was measured by computing the elapsed time between two invocations of *gettimeofday*, meanwhile the time for clustering the initial model were excluded from the execution time. Five measurements were made for each test case, and the average value is reported.

The dataset scaling graphs for the baseline version and the final version are shown in Figure 3. There are four candidate datasets generated from Bayesian network models which contain 1,100, 2,200, 4,400 or 8,800 nodes. It is seen that the speedups become much better in the final version. In addition, different from baseline version, whose speedup often drops with increasing dataset, the final version increases its speedup with increasing dataset. Another benefit we noticed from the two figures is that after the code optimizations the speedup variance between the two platforms becomes more similar. Thus the speedups can be understood more easily.



(a) baseline (b) after optimization

Figure 3. Dataset scaling graphs

Table 6 lists the performance metrics of our three parallel versions in the two test machines, with the dataset of 2,200 nodes grouping to 74 modules. We list the total sequential running time, the running time in the *dependency-learning* phases and *assignment-learning* phases in Table 6(a), respectively. We also list the speedups of 4 threads in Table 6(b).

(a)	QP-machine			Unisys-es7000		
Time	Total	Dep.	Ass.	Total	Dep.	Ass.
BASE.	14.8	4.50	10.3	14.6	3.99	10.6
MEM.	10.5	4.04	6.43	10.1	3.92	6.21
FINAL	7.35	3.66	3.69	7.00	3.58	3.42

(b)	QP-machine			Unisys-es7000		
Sp.(4T)	Total	Dep.	Ass.	Total	Dep.	Ass.
BASE.	2.32	1.38	3.28	1.56	0.90	2.04
MEM.	3.33	3.01	3.57	2.37	2.12	2.57
FINAL	3.35	2.97	3.83	2.94	2.37	3.92

Table 6. Performance metrics with dataset of 2,200 nodes

We deduced Table 7 from Table 6. It lists the relative improvements of sequential performance and parallel speedup using two different optimizations, i.e. memory related optimization and static function usage.

(a)	QP-machine			Unisys-es7000		
Time%	Total	Dep.	Ass.	Total	Dep.	Ass.
Mem.	41.8	11.4	61.0	44.3	1.9	71.2
Static	42.4	10.3	74.3	44.7	9.6	81.4

(b)	QP-machine			Unisys-es7000		
Sp.(4T)%	Total	Dep.	Ass.	Total	Dep.	Ass.
Mem.	44	118	8.7	52	135	26
Static	0.60	-1.2	7.4	24	12	52

Table 7. Relative Improvements by different optimizations

In Table 7, it is seen that the memory related optimizations improve not only the sequential performance by larger than 40%, but also the relative parallel speedup by more than 40%. On the other hand, static function usage results in lower parallel speedup improvement compared to its sequential counterpart. Memory related optimizations and static function usage have different effects in the two phases of the learning process, i.e. *dependency-learning* phase and *assignment-learning* phase. Memory related optimizations have lower impact on the *dependency-learning* phase according to the sequential performance while it has higher impact according to parallel speedup. Static function usage has consistent impact patterns with higher impact on *assignment-learning* phase. The reason will be described later in detail.

From the result, we can conclude that memory related optimizations reduce not only memory accessing overheads but also interferences between processors while static function usage only reduces memory accessing overheads.

In the *dependency-learning* phase, using score cache makes the situation complicated. Firstly, the last line of Table 6(b) shows that although we have satisfactory parallel speedup of the *assignment-learning* phase in the final version, the parallel speedup of the *dependency-learning* phase is far from acceptable. The reason is that to avoid redundant cache computations we adopt static scheduling strategy, which results in workload imbalance. After resource contentions being reduced by aforementioned optimizations, workload imbalance becomes a major hurdle to the parallel scalability. Secondly, less sequential performance improvements are made in the *dependency-learning* phase, as illustrated in Table 7(a). The reason is that loading score caches, which dominates the *dependency-learning* phase, is rarely influenced by these optimizations in sequential mode. Thirdly, with memory optimizations we achieved remarkable improvements in the parallel speedup in dependency-learning phase, as illustrated in Table 7(b). This is because that memory alignment greatly impacts parallel behavior of loading score caches. Fourthly, static function usage only influences those from-scratch computations, which include those miss-cache computations in the *dependency-learning* phase and all computations in the *assignment-learning* phase. Therefore, more parallel speedup improvements are achieved in the *assignment-learning* phase by using static functions.

Further evaluation of the parallel scalability of our program on Unisys-es7000 machine was performed, with dataset of 8,800 nodes running on 16 threads. The result is consistent with the conclusion we made.

One thing we should emphasize here is that those “naïve optimizations” such as the manual redundant computation removal, computation reuse and loop unrolling [5], are observed to slightly reduce the parallel speedup. It is because that those optimizations mainly focus on hotspots, which are most likely in parallel regions of the program. Thus those optimizations lead to less parallel ratio, and finally less relative speedup according to the Amdahl’s Law.

7. Summary

In this paper, we presented a parallel implementation of module network learning algorithm with OpenMP. The key contributions of this paper are as follows. First, we give parallelization of an emerging application using OpenMP. Second, we identify some performance penalties and give effective recipes. Considering the 2,200 nodes dataset, the optimizations enhance the parallel speedup up to 88% (2.94/1.56 - 1), together with a 2X (14.6/7) sequential performance improvement. Third, we analyze impact patterns of different optimizations on sequential performance and parallel speedup.

The results from this paper emphasize the importance of a memory-friendly program to increase the parallel efficiency of a shared-memory multi-processor, especially the importance of control the random data layout caused by the dynamic memory allocation. At the same time, the efficiency of memory usage of STL vectors on SMP needs also be improved.

References

[1] Intel vtune performance analyzer. Intel Corporation. http://developer.intel.com/software/products/vtune/index.htm.

[2] Pnl: Probabilistic network library. Intel Corporation. http://sourceforge.net/projects/openpnl/.

[3] Standard template library programmer’s guide. SGI Corporation. http://www.sgi.com/tech/stl/.

[4] *OpenMP C and C++ Application Program Interface, 2.0 Edition*. The OpenMP Architecture Review Board, Mar. 2002.

[5] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[6] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications, 2000. http://www.cs.umass.edu/ emery/hoard/.

[7] S. Borkar, P. Dubey, and et al. Platform 2015: Intel processor and platform evolution for the next decade, 2005. ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Platform_2015.pdf.

[8] D. Heckerman. A tutorial on learning with bayesian networks, 1996. ftp://ftp.research.microsoft.com/pub/tr/tr-95-06.pdf.

[9] L. Liu, W. Hu, C. Lai, H. Jiang, W. Chen, W. Zheng, and Y. Zhang. Parallel module network learning on distributed memory multiprocessors. In *2005 International Conference on Parallel Processing Workshops (ICPPW’05)*, pages 129–134, 2005.

[10] K. Murphy. A brief introduction to graphical models and bayesian networks, 1998. http://www.cs.ubc.ca/~murphyk/Bayes/bayes.html.

[11] E. Segal, D. Pe’er, A. Regev, D. Koller, and N. Friedman. Learning module networks. *The Journal of Machine Learning Research*, 6:557–588, Sept. 2005.

[12] X. Xu, L. Wang, and D. Ding. Learning module networks from genome-wide location and expression data. *FEBS Lett.*, 578(3):297–304, Dec. 2004.