STUDENT PAPER: An Implementation of Parallel Bayesian Network Learning

Joseph S. Haddad
The University of Akron
302 E Buchtel Ave
Akron, OH, 44325, United States
jsh77@zips.uakron.edu

Timothy W. O'Neil
The University of Akron
302 E Buchtel Ave
Akron, OH, 44325, United States
toneil@uakron.edu

Anthony Deeter
The University of Akron
302 E Buchtel Ave
Akron, OH, 44325, United States
aed27@zips.uakron.edu

Zhong-Hui Duan
The University of Akron
302 E Buchtel Ave
Akron, OH, 44325, United States
duan@uakron.edu

ABSTRACT

Bayesian networks may be utilized to infer genetic relations among genes. This has proven useful in providing information about how gene interactions influence life. However, Bayesian network learning is slow as it is an NP-hard algorithm. K2, a search space reduction, helps speed up the algorithm but may introduce bias. The bias arises from the fact that K2 enforces topologies which makes it impossible for subsequent nodes to become parents of previous nodes while the algorithm builds the network. To eliminate this bias, multiple Bayesian networks must be computed to ensure every node has the chance to be a parent to every other node. The purpose of this paper is to propose a hybrid algorithm for generating consensus networks utilizing OpenMP and MPI. This paper evaluates the parallelization of network generation and provides commentary on learning and implementing OpenMP and MPI. The OpenMP and MPI accelerations are implemented in a single library and can be switched on or off. These accelerations are for computing multiple Bayesian networks simultaneously. Methods are developed and tested to evaluate the results of the implemented accelerations. The results show generating networks across multiple cores results in a linear speed-up with negligible overhead. Distributing the generation of networks across multiple machines also introduces linear speed-up, but results in additional overhead.

1. INTRODUCTION

Inferring relations among genes requires a significant amount of data. Bayesian networks may be used to correlate this data and extract relationships among the genes [12]. We do not know what this relationship is, but we do know it has a high likelihood of existing. These relationships can then be used to make testable hypotheses to determine how gene interactions influence life in organisms or humans. As a result, tests can be performed in the lab with more confidence and a reduced chance of wasting time and resources.

This concept has been applied to smaller data sets and shows promising results [12], however remains too slow to be applied to a larger problem. It is our objective to decrease the runtime required to form a network which may reveal genetic interactions. Bayesian network learning, however, is inherently slow because it is an NP-hard algorithm [4]. Search space reduction algorithms may be utilized to reduce the computational complexity. K2 is a great example of a search space reduction algorithm, and is our algorithm of choice. However, it introduces a new problem. K2 restricts the parent hierarchy of genes within the network [4], and thus introduces bias in the computed relations. To achieve high confidence in the generated networks, an abundance of Bayesian networks need to be computed using random search space restrictions. These random search space restrictions (or topologies) remove the bias and provide results which can be interpreted at various levels of confidence.

By eliminating one problem and introducing another, consensus networks enable the ability of parallelization by requiring multiple units of work rather than just one faster unit of work. Other authors describe parallel implementations that can increase the speed of Bayesian network learning [2] [8]. However, no libraries exist which compute multiple Bayesian networks concurrently. This project examines the value of Bayesian network learning within a parallel environment in order to reduce the time needed to generate consensus networks using many topological inputs. This examination is performed through implementation of the said algorithm, exploring methods available such as OpenMP and MPI.

Results from running experiments with varying number of cores and machines are examined and it is found our parallelization has a positive impact. There are a couple caveats, however, such as the over provisioning of resources which leads to waste and potential introduction of latency from cluster parallelism. When the resources are appropriate for the problem size, OpenMP and MPI substantially reduce

the time to generate a consensus network. The reduction in runtime appears to be linear, more so after accounting for introduced latency and overhead.
This paper is an extension to the initial analysis performed on the algorithm and explains the thought processes behind the implementation. The preceding publication shows why the algorithm needs to be sped up, as an increase in samples causes linear growth of the problem and introduction of additional genes causes exponential growth of the problem [5]. After reading this paper, the reader should have a sense of why and how the parallelization was reasoned about and implemented to achieve optimal efficiency.

2. BACKGROUND

2.1 Bayesian Networks

Bayesian networks capture qualitative relationships among variables within a directed acyclic graph (or DAG). Nodes within the DAG represent variables, and edges represent dependencies between the variables [6] [11]. Bayesian networks have a search space which grows exponentially when introducing new nodes and not placing restrictions on the structure of the network. This complication can be overcome by using the K2 algorithm. The K2 algorithm reduces the computational cost of learning by imposing restraints on parent node connections via topological ordering [4]. Here, a topology refers to a hierarchical structure of parenthood that the K2 algorithm will utilize to reduce overall computational complexity while scoring data relationships. Restricting the parent ordering, however, creates an issue of bias, which is inherent within a constraint-based search space reduction [12]. Sriram [12] proposed a solution to this issue by creating a consensus network, or the combination of multiple Bayesian networks derived from several topological inputs. To eliminate the bias created by these restraints, many randomly generated topologies are used. By increasing the number of topological inputs, the consensus network has a greater chance of reflecting the true nature of the gene interactions with higher levels of confidence.

2.2 OpenMP

OpenMP or (Open Multi-Processing) is a cross-platform, multilingual application programming interface (API) which enables shared-memory parallel programming on a single machine. The OpenMP specification consists of compiler directives and library functions used to parallelize portions of a program's control flow [10]. The most rudimentary example of OpenMP would be to distribute a for-loop across multiple threads.
An advisory board of top entities in computation controls its specification [1] which can be implemented by various compilers to target specific system capabilities and architectures. The specification includes language-specific APIs, compiler directives, and standardized environment variables [10]. The model of OpenMP is comparable to the fork-join model, but provides additional convenience (cross-platform) features through compiler directives. These directives consist of, but are not limited to, barriers, critical regions, variable atomicity, shared memory, and reductions [10].
OpenMP enables parallel code portability at a level which would not be achievable while retaining an ideal code climate. OpenMP, by nature allows simple and straight-forward parallelization of loops with a compiler directive that targets the system for which the program is compiled on. Without OpenMP, the program would have to include many different libraries and routines to achieve parallel code across different systems. The result of this would be a program which only works on a specific set of machines, or a code base which is hard to maintain and debug when changes are made to the underlying algorithm.

2.3 MPI

MPI (or Message Passing Interface) is a standard which outlines network-routed (a)synchronous communication between machines [9]. MPI enables executing programs across multiple machines in a cluster and passing messages between them to schedule work or share information.
Execution of a program which utilizes MPI is most often performed with a tool. This tool is responsible for forwarding appropriate parameters to each program in order to specify the information required for the processes to communicate. Upon program start, the MPI execution environment must be initialized using the MPI library methods [9]. The initialization sequence results in augmented program arguments (to remove arguments passed by the execution tool) and the rank of the program in the MPI environment [9]. This information allows the program to proceed as normal while being a small part in a larger sum.

3. METHODOLOGY

Testing was performed on the Blue Waters petascale machine at the University of Illinois at Urbana-Champaign. The facility is maintained by Cray and consists of 22,640 Cray XE6 machines and 3,072 XK7 machines, which are CPU-only and GPU-accelerated machines respectively. The XE6 machines consist of two 16 core AMD processors with 64 GBs of RAM. The XK7 machines consist of a single 16 core AMD processor with 32 GBs of RAM and a NVIDIA K20X GPU [7].
Cray XE6 machines were used to perform all tests utilizing purely synthetic data. OpenMP and MPI were implemented by the Cray Compiler, Cray C version 8.3.10. The synthetic data is in the form of a gene-by-sample matrix consisting of the presence or absence of each gene within the sample. This data was generated according to a model we defined. We then ensured the result of the consensus network(s) matched our model to validate functionality and evaluate a degree of correctness for our algorithm. Each test was run five times with the mean, standard deviation, and standard error calculated to measure runtime consistency.
The library being used to run the tests is available online [3]. This library was implemented as described in this paper.

3.1 Processors

The first natural step in parallelizing computation is to attempt to use multiple cores (or threads) simultaneously on the machine. This can be done by running multiple instances of the program, or by implementing code which takes advantage of multiple threads. Analyzing the program reveals a couple potential places for parallelization. There are many for-loops which perform actions which are independent from one another. The for-loops identified for inspection are the generation of topologies and the iteration over the topologies to generate networks.
The generation of topologies results in a a predetermined number of topologies filled into an array. This operation

can be easily parallelized across multiple cores as they are independent. The appropriate tool to perform this parallelization is OpenMP. OpenMP was implemented with a simple compiler directive which sped up computation.

```
#pragma omp parallel for
for (...) { }
```

Iterating over the topologies to generate networks can also be parallelized. The creation of Bayesian networks are independent from one another, and thus, networks can be asynchronously generated. Implementation of this parallelization is straight-forward as Bayesian network computation does not mutate its data set. This prevents us from having to replicate the memory and increase the space complexity of the algorithm. OpenMP was implemented again as shown above. Additionally, within the parallel for, the resulting network must be appended to the consensus network. The consensus network, however, is not thread-safe and must be operated on within a critical section. A critical section specifies that the code can only be executed on one thread at a time.

```
#pragma omp critical
for (...) { }
```

This ensures the networks are properly summed together, otherwise, an addition may be lost. For example, if `Thread A` and `Thread B` attempt to increment a variable at the same time, they may both access the value before the other commits the new value. This will result in a lost operation, as the threads are not aware of one another.
To measure the resulting computational runtime decrease, multiple tests were performed with varying number of processors. A single set of synthetic data was used which consisted of 10 genes and 10,000 samples. Using an exclusively reserved machine, tests were run by varying the number of processors (up to 32) and measuring the algorithm performance for the creation of 160 Bayesian networks per gene 1600 total). We have reached the resource limits on the systems which we have access to, and cannot test beyond 32 cores. The selection of 10 genes and 160 Bayesian networks was arbitrarily chosen as sufficient means to measure computation time.

3.2 Cluster Parallelism

Distributing work across multiple machines requires a different approach than that of OpenMP. OpenMP cannot share memory across machines so it cannot be applied to this situation. MPI is optimal for this situation as it allows machines to send messages back and forth to share memory and communicate their responsibilities and results. Distributing the Bayesian network learning process across multiple machines doesn't make much sense because each step is dependent on the previous, so the result would be a slower computation since calculations couldn't happen in parallel and there would be added network latency. The main candidate for distribution would be the computation of a Bayesian network (or the iteration over the topologies), because networks are computed independent of one another and there is a large backlog of networks which need to be computed. Distributing the work with MPI is surprisingly simple, as the topologies are randomly generated. This means there is no communication required prior to beginning computation. Upon initialization, each machine must determine its rank and role by augmenting the arguments, this may be done like so.

```
int main(int argc, char **argv) {
  int forkIndex = 0, forkSize = 1;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &forkIndex);
  MPI_Comm_size(MPI_COMM_WORLD, &forkSize);

  ...
}
```

Each machine can then determine how much work it needs to do by dividing the number of requested topologies per gene by the number of machines in the swarm.

```
int top_d = topologies / forkSize;
int top_r = topologies % forkSize;
if (forkIndex < top_r) ++top_d;
topologies = top_d;
```

When the machines complete their share of the computation they communicate to coalesce the computed networks into a consensus network. The master machine then saves the consensus network to the disk and completes any other required computations which are simple enough not to require being distributed across machines.
Tests are conducted to measure the impact on runtime when multiple machines are used. The same data is used from the above (processors) test. Tests were run on dedicated machines utilizing 16 processors and computing 60 Bayesian networks per gene (600 total). The selection of 10 genes and 60 Bayesian networks was arbitrarily chosen as sufficient means to measure computation time.

4. RESULTS AND DISCUSSION

In the following tables, the standard deviation is represented by the letter `s` and the standard error is denoted by `se`. This standard deviation and error is in regards to the algorithm runtime, not the accuracy of the algorithm.

4.1 Processors

When increasing the number of processors, the resulting runtime decrease appears to be linear. The linear nature of the results removes the necessity for further testing between the number of cores tested. Figure 1 illustrates that as the number of processors increase, the runtime decreases at approximately the same rate. Exact results may be seen in Table 1.

Table 1: Runtimes for the program across increasing numbers of processors.

Cores	Mean Time	*s*	*se*
1	396.348	3.192	1.427
2	269.023	0.530	0.237
4	137.359	0.629	0.281
8	76.169	0.220	0.090
16	40.359	0.307	0.137
32	22.172	0.144	0.064

This linear decrease is consistent with how OpenMP distributes its work. OpenMP distributes the task of an independent Bayesian network computation across multiple threads simultaneously. These independent tasks are non-blocking and do not lock one another, and thus have very

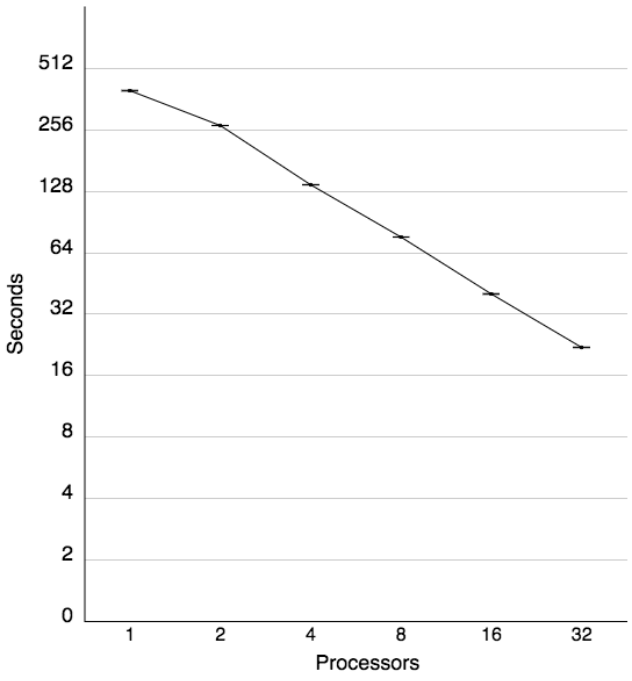

Figure 1: Illustrates runtime decrease as the number of processors increase. The decline is nearly linear.

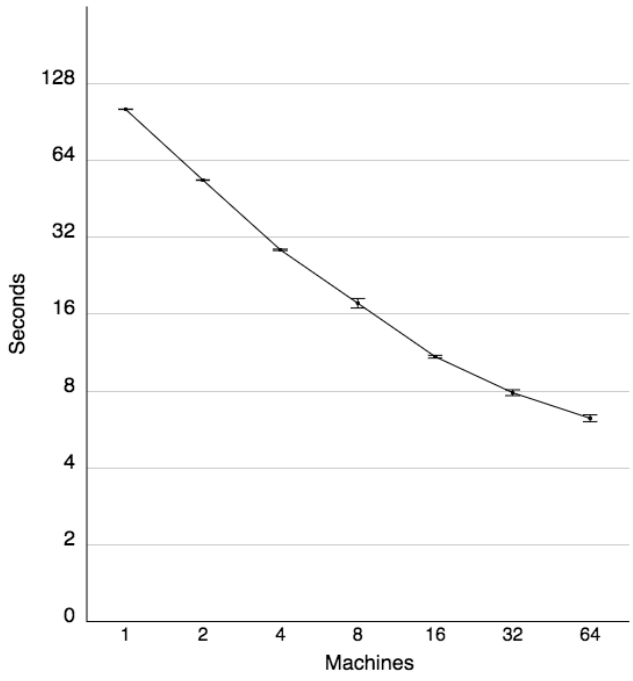

Figure 2: Illustrates runtime decrease as the number of machines increase. The decline is nearly linear.

little contention. There is one lock after each computation which appends the network to the consensus network, but is negligible to the total time taken to compute the Bayesian networks. OpenMP results in such low runtime standard error because it works with memory within the program and requires no network communication like MPI. The reduction of standard error as the number of threads increase may be due to the kernel. The kernel is responsible for scheduling threads and ensuring other work on the system gets done. The increase in threads means there are more threads which may go uninterrupted by the kernel scheduling something else from the operating system.

4.2 Cluster Parallelism

The resulting runtime decrease also appears to be linear while increasing the number of machines. However, as the number of machines increase, overhead also increases. Figure 2 demonstrates that as the number of machines increase, there is much more variation introduced and overhead in the runtime.

Observing 64 machines and leading up to 64 machines, it can be noted that the reduction in runtime becomes less and less and then starts increasing. This increase in runtime happens when the inflection point has been reached for the given set of data. At some point, it takes longer to send the data over the network than it would be to simply compute more data on fewer machines. There are some potential modifications which can be made to mitigate this overhead (such as asynchronous coalescing), but it cannot be eliminated completely. It is important to note that an increase in resources does not necessarily mean an increase in performance, nor always one for one; see Table 2 for test results.

The standard error generally increases with the increase in machines, but this is not always true. There does not seem to be a correlation between an increase or decrease in machines with an increase or decrease in standard error, except for the general rule stated above. This is consistent with the fact that networks are very unpredictable. Pings may vary wildly depending on other network traffic and the route which packets decide to take. Additionally, there may be other noisy peers on the network hogging bandwidth and causing slower transmissions. On clusters across the world wide web, traffic may have to travel through geographical displacement and suffer packet loss or increases in latency. The only thing consistent with the standard error is that it is not consistent.

5. CONCLUSION

By generating a consensus network out of many Bayesian networks, researchers may screen and infer new gene interactions. This allows researchers to feel more confident about testing hypotheses in the lab, such that their resources and time will not be wasted.

We have concluded that utilizing parallelization through means of OpenMP and MPI substantially reduces the time to generate a consensus network. However, as demonstrated in the graphs above, an increase in resources must be tailored to the problem at hand. Increasing the resources too significantly becomes detrimental, resulting in costly waste; see Table 2.

Future work may involve parallelizing the coalescing of consensus networks in effort to reduce the overhead introduced when increasing cluster parallelism. Additionally, all matrix operations are currently done on a single-thread. These operations (in some cases) contain thousands of rows and columns being applied to an expensive mathematical function.

Table 2: Runtimes for the program across increasing numbers of machines.

Nodes	Mean Time	*s*	*se*
1	102.204	0.361	0.161
2	53.451	0.272	0.122
4	28.656	0.383	0.171
8	17.8	1.812	0.810
16	10.917	0.327	0.134
32	7.862	0.462	0.207
64	6.259	0.444	0.198
128	6.739	0.430	0.193
256	7.904	1.110	0.496
512	7.241	0.246	0.110
1024	8.845	1.105	0.494

These operations are ideal for the GPU as it can perform the arithmetic across several thousand of threads simultaneously. As such, the motivation for this is that CUDA (or other means of GPGPU acceleration) has the potential to speed the algorithm up by several orders of magnitude.

6. REFLECTIONS

Working on this project gave me a massive amount of experience, which far surpassed what I thought it would. I gained experience in professional writing for journal publications and renewed my skills in proofreading. I also gained exposure to a whole new aspect of project organization which I was not used to: meetings with advisors, progress reports, and demos. I feel like this has really helped foster my professional identity and prepared me more for higher education and the workforce. Additionally, I flexed my problem solving skills while implementing the algorithm and begun refactoring. The refactoring had to be done in such a fashion to allow for parallelization. This presented some challenges because there were also memory considerations to make things sharable over the network (MPI). Overall, I learned many invaluable skills which will be applied to my future education and work. Notably, I performed my first publication [5] and gave a presentation at the associated conference, then proceeded to present a poster version of the paper at GLBIO 2016 to draw attention to the work.

7. ACKNOWLEDGMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.
Additional financial support was provided by the Buchtel College of Arts and Sciences at The University of Akron. Further support was provided by a grant from the Choose Ohio First Bioinformatics scholarship.
The data, statements, and views within this paper are solely the responsibility of the authors.

8. REFERENCES

[1] About the OpenMP ARB and OpenMP.org: *http://openmp.org/wp/about-openmp/*.
[2] Altekar, G. et al. 2004. Parallel metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. Bioinformatics.
[3] Bayesian Learning source code: *https://github.com/Timer/bayesian-learning*.
[4] Cooper, G.F. and Herskovits, E. 1992. A Bayesian method for the induction of probabilistic networks from data. Machine Learning.
[5] Haddad, J.S. et al. 2016. Analysis of Parallel Bayesian Network Learning. Proceedings of the 31st International Conference on Computers and Their Applications.
[6] Korb, K. and Nicholson, A. 2003. *Bayesian artificial intelligence*. Chapman and Hall/CRC.
[7] Lessons Learned From the Analysis of System Failures at Petascale: The Case of Blue Waters: *https://courses.engr.illinois.edu/ece542/sp2014/finalexam/papers/bluewaters.pdf*.
[8] Misra, S. et al. 2014. Parallel Bayesian network structure learning for genome-scale gene networks. International Conference for High Performance Computing, Networking, Storage and Analysis.
[9] MPI: A Message-Passing Interface Standard: *http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf*.
[10] OpenMP Application Program Interface: *http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf*.
[11] Pearl, J. 1998. *Probabilistic inference in intelligent systems*. Morgan Kaufmann Publishers.
[12] Sriram, A. 2011. *Predicting Gene Relations Using Bayesian Networks*. MS thesis, University of Akron.