Scalable Parallel Implementation of Exact Inference in Bayesian Networks[1]

Vasanth Krishna Namasivayam and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California, Los Angeles
{namasiva, prasanna}@usc.edu

Abstract

We present a scalable parallel implementation for exact inference in Bayesian Networks. We explore two levels of parallelization: top level parallelization which uses pointer jumping to stride across nodes; and node level parallelization which parallelizes the node level computations which are independent from each other. For a junction tree with n cliques, using p processors, the worst-case running time is

$$\frac{n}{p}(\log n) * r^w \quad (1)$$

where w is the clique width and r is the maximum range or number of states of the variable.

We have implemented the algorithm using MPI and OpenMP. We consider three different types of input junction trees: linear junction trees, balanced trees and random junction trees, and obtained speedups of 203, 181 and 190 respectively over 256 processors.

Keywords*: Bayesian Networks, Junction Tree, Partitioning, Scalability, Pointer-Jumping, Loop level parallelization.*

1 Introduction

The joint probability distribution of two discrete random variables X and Y is a function whose domain is the set of ordered pairs (x, y), where x and y are possible values for X and Y, respectively, and whose range is the set of probability values corresponding to the ordered pairs in its domain. A full joint probability distribution for any real-world system can be used for inference, however such distributions grow intractably large as the number of variables used to model the system grows. While dealing with joint probability distributions it is seen that independence and conditional independence relationships can greatly reduce the size of the probability distributions. This property is taken advantage of by belief networks.

Bayesian reasoning has been used in Artificial Intelligence since the 1960s, especially in medical diagnosis [1]. Belief networks have found application in a number of domains, including consumer help desks, nuclear reactor diagnosis, tissue pathology, pattern recognition, credit assessment, data mining[2], image analysis, robotics, genetics[3] and computer network diagnosis [1]. To solve a belief network also known as inference, is to solve the conditional probability of the query nodes given a set of evidence nodes; i.e. we find the probability of the query variable being true, given knowledge of the evidence variables in the network.

There are two main approaches for computing probabilities in a Bayesian network - $exact\ inference$ and $approximate\ inference$. The Lauritzen Speigelhalter [4] algorithm is the most popular exact inference algorithm. Exact inference is NP hard [5] and computationally very expensive. The time complexity of exact inference is exponential with the density of the network and the number of states in the random variable - the number of states of a random variable referring to the set of values it can take.

This paper explores a scalable, parallel algorithm, which is topology independent. There has been considerable work in the field of parallelizing exact inference. These include (D'Ambrosio [6]; Diez and Mira [7]; Kozlov [8]; Kozlov and Singh [9]; Shachter, Andersen, and Szolovits [10] and Pennock [5]). The popular exact inference junction tree algorithm for multiply connected networks (Lauritzen and Spiegelhalter [4]) was also conceived as a parallel algorithm with one processor per clique. DAmbrosio [6] examines the possibility of parallelizing the symbolic probabilistic inference (SPI) algorithm. Two sources of concurrency are identified: topological parallelism and conformal product parallelism, in the former case only independent computations are exploited. Kozlov and Singh [9] present a parallel implementation of the junction tree algorithm, however they

[1]The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. government.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009.

only consider parallelizing independent operations. Pennock [5] presents a logarithmic time Bayesian inference algorithm. The implementation in this paper is based on this algorithm. The algorithm is based on a theoreotical model: the CREW PRAM and does not take into account issues such as communication costs between parallel processors. The state of the art in parallel techniques in exact inference involve partitioning the graph and assigning the partitions to various processors [5]. However, this technique is topology dependent.

Some networks are inherently more parallelizable than others. In some networks, the computation of two different nodes may be independent whereas in others dependencies may exist, hence the computations can not be performed concurrently. Simple partitioning techniques when applied on long-chain topologies will have the same running time as the serial algorithm.

We implemented a parallel version of the exact inference algorithm which overcomes this dependency. We exploit parallelization at the node as well as the clique level. Hence our implementation runs on a junction tree with n cliques in $\frac{n}{p}(\log n)*r^w$ time, using p processors, where w is the clique width and r is the maximum range or number of states of the variable, regardless of the topology of the junction tree. We have implemented the algorithm using MPI and OpenMP on the clusters at San Diego Supercomputer Center. We consider three different types of input junction trees: linear junction trees, balanced trees and random junction trees. Our implementation demonstrates linear scalability.

The rest of the paper is organized as follows: in Section 2 we cover the basics of Bayesian inference. Section 3 describes the parallel exact inference algorithm and our implementation. Section 4 describes the experiments which were conducted to prove scalability and efficiency of the parallel algorithm. We also present detailed experimental results obtained. Finally Section 5 concludes the paper and discusses possible future work.

2 Bayesian Inference

A Bayesian network is represented by a graph which is composed of a set of nodes, each node represents a variable, and a set of edges. Each edge connects two nodes, and an edge can have an optional direction assigned to it. An edge between two nodes indicates a relation between the nodes and the direction indicates the causality. Let the variables be $\{X(1), ..., X(n)\}$. The probability that variable $X(i)$ has a value k_i, is given as $P(X(i) = k_i)$ where $k(i)\ \epsilon\ \{k_1, k_2, k_3.....k_r\}$. Any random variable $X(i)$ can take r values, $\{k_1, k_2, k_3.....k_r\}$.

If there is an arc from node A to another node B, then we say that A is a parent of B. If a node has a known value, it is said to be an evidence node. Let $parents(A)$ be the parents of the node A. Then the joint distribution for $X(1)$ through $X(n)$ is represented as the product of the probability distributions $P(X(i)|parents(X(i)))$ for $i = 1$ to n. In mathematical notation the joint probability also known as the conditional probability table (CPT) is expressed as

$$P(A) = \Pi(P(X(i)|parents(X(i)))) \qquad (2)$$

In a network, when we get new information about variables in the network, we update the conditional probability tables, to reflect this new information. This updating is known as $evidence\ propagation$. Once all the beliefs are updated, the conditional probability tables contain the most recent beliefs in any variable and can be queried like a simple database to evaluate probabilities.

Most exact inference algorithms like the popular LS algorithm [4] do not work on the Bayesian network directly, instead they work on an intermediate data structure - the junction tree. Evidence propagation based on Bayes rule can not be applied directly to nonsingly connected networks i.e a Bayesian network with loops. Loops are undirected cycles in the underlying network.

If we ignore the existence of loops, and permit the nodes to communicate with each other as if the network were singly connected, then there is a possibility that the messages may circulate indefinitely in these loops.

Pearl[11] has presented several methods to overcome the problem namely conditioning and stochastic simulation. Lauritzen and Spiegelhalter[4] approached the problem of nonsingly connected networks from a mathematical perspective. This method involves the extraction of an undirected triangulated graph from the directed acyclic graph in the causal network, and the creation of a tree whose vertices are the cliques of this triangulated graph. Such a tree is called "join" or "junction" tree. Probabilites in the original causal network are updated by passing messages among the vertices in this tree.

The steps involved in converting the bayesian network to a junction tree is well documented in earlier papers such as [4]. The steps include moralization, triangulation, identifying cliques and constructing the junction tree.

All our experiments were conducted on a junction tree. Tools are available that convert arbitrary Bayesian networks to a junction tree [12].

3 Parallel Algorithm

In our implementation, the Bayesian network is converted into a junction tree and provided as input. A potential table for any clique in a junction tree, lists the probabilities associated with all the possible combinations of the random variables in that clique. Our implementation accepts one evidence variable coming in at the root of the junction tree.

An extension of this, allowing the evidence to come in at any node in the network, would involve re-rooting the junction tree so that the evidence node is at the head of the new junction tree. The procedure for re-rooting the tree is described in [5]. A parallel version of arc reversal may be used. This technique is possible since in any Bayesian network, the edges can be reversed, and the resulting network would represent the same joint distribution as long as no directed cycle is created [5].

3.1 Parallel Rewrite of CPT

The algorithm we implement overcomes the topological dependencies of other parallel techniques, by the use of a well known parallel technique: pointer jumping. Pointer jumping has been used earlier in devising parallel algorithms, list ranking, minimum spanning tree, etc .

The key to pointer jumping is to rewrite the conditional probability table of each clique, in terms of its grandparent. Mathematically this operation can be represented as:

$$P(A(j)|A(j-2)) = \sum_{A_{j-1}} P(A(j)|A(j-1))P(A(j-1)|A(j-2)) \quad (3)$$

The equation represents how the conditional probability table for any node in a Bayesian network can be written in terms of it's grandparent. The same equation can be extended for junction trees; the potential table in a junction tree being the product of the conditional probability tables of the individual variables.

3.2 Parallel Update of Cliques

Each clique has r^w entries in the potential table, which are updated every iteration. Every iteration involves rewriting the potential table of a clique in terms of it's grandparents. The first step involves dividing the potential entries by the separator set entries, to obtain the conditional probability table. Mathematically this operation is represented as:

$$\phi_X = \phi_X / \phi_R \quad (4)$$

where ϕ_X is the belief potential associated with cluster X and ϕ_R is the potential associated with cluster R. ϕ_Y is the belief potential associated with Y, Y is the parent of X. The final clique update operation: Step 5 of Figure 4 is mathematically represented as:

$$\phi_{Rold} = \phi_R \quad (5)$$

$$\phi_R = \sum_{Y/R} \phi_Y \quad (6)$$

$$\phi_X = \frac{\phi_R}{\phi_{rold}} \phi_X \quad (7)$$

These operations are independent. Each clique update involves dividing all the potential entries by the old seperator set entries, followed by multiplying them with the new seperator set entries. In our implementation, when any processor is idle, the independent clique updating operations are parallelized, sending the required entries of the potential table to the idle processor.

Figure 1 presents the parallel algorithm illustrating the pointer jumping operations. Figure 2 presents the parallel algorithm including the node level parallelization.

3.3 Implementation Details

The MPI commands used were MPI_Send and MPI_Recv. The MPI_Send performs a basic send. We specify the destination processor, the datatype of the information being sent and the number of bytes being sent. The MPI_Recv performs a basic receive. We once again specify the source processor, the datatype and the number of bytes being received. These commands are a part of the MPICH library for C. The OpenMP directive used was $\#pragma\ omp\ parallel$. The connectivity matrix is a $n \times n$ matrix, which stores information about the connections between various cliques. By traversing this matrix row-by-row, it is possible to ascertain the parent of the parent of any clique.

We created our own data structure, The data structure stores the following parameters:

* The indices of the nodes in each clique
* The clique connectivity matrix
* The number of nodes in each clique
* The ranges of the nodes in each clique
* The entries in the potential tables
* The nodes in the separator sets
* The separator set potential tables

The input junction tree with n cliques is partitioned among the p processors. Each processor is allocated n/p cliques, and all the information about those cliques are stored in the local memory of that processor. During the pointer jumping operation, if the clique is present locally in the same processor, the local memory is accessed to update the potential tables. If the clique has been allocated to some other processor, the data structure is sent using MPI_Send to the corresponding processor, where the updating of the potential tables takes place. The number of operations taking place during the updating of a clique is r^w multiply and divide operations, where w is the number of random variables in a clique and r is the number of states of the random variable.

1: Mark the Evidence Node/Root Clique as done
2: For all children of root clique compute P(Aj)=P(Aj|Aroot=evidence)
3: While there is a clique not marked done
4: For each clique Aj in parallel
5: **if** potential table of parent_clique not present in local memory **then**
6: MPI_SEND(potential table from parent_clique to clique)
7: **end if**
8: **if** parent of clique not marked done **then**
9: Compute Equation (2)
10: **else**
11: Divide all potential entries by old Separator Set entries
12: Multiply all potential entries by new Separator Set entries
13: **end if**
14: **if** parent(Aj) is marked done **then**
15: the mark Aj done
16: **else**
17: Ascertain parent(parent(clique)) from connectivity matrix
18: Reassign parent(clique) to parent(parent(clique))
19: **end if**
20: Update all Separator Set entries

Figure 1. Algorithm for parallel exact inference

3.4 Analysis

Theoretically best speedups are obtained for junction trees with long linear chains compared to the straight forward technique of partitioning. This is because the pointer jumping technique overcomes the dependencies between the cliques present in a long chain; whereas a straightforward parallel technique would have the same time complexity as a serial code. Inference on a junction tree with n cliques with n processors takes $O(\log n)$ time. Inference on a balanced junction tree with n cliques takes $O(\log \log n)$ time. As the clique-width increases the execution time increases. This is because the time complexity is exponential with increase in clique width. The number of floating point operations during each parallel clique update is r^w: the r^w entries in the potential table are updated. The communication cost is $separator\ set\ width * r^2$. The separator set width can not exceed clique width w. During each clique update, the conditional probability table of the parent clique is accessed. The number of entries in a CPT for a random variable with r states is r^2. Hence the MPI communication overheads are minimal as compared to the computational

1: Mark the Evidence Node/Root Clique as done
2: For all children of root clique compute P(Aj)=P(Aj|Aroot=evidence)
3: While there is a clique not marked done
4: For each clique Aj in parallel
5: **if** potential table of parent_clique not present in local memory **then**
6: MPI_SEND(potential table from parent_clique to clique)
7: **end if**
8: **if** parent of clique not marked done **then**
9: OMP parallelize (public potential table)
10: Compute Equation (2)
11: **else**
12: Divide all potential entries by old Separator Set entries
13: Multiply all potential entries by new Separator Set entries
14: End OMP
15: **end if**
16: **if** parent(Aj)is marked done **then**
17: the mark Aj done
18: **else**
19: Ascertain parent(parent(clique)) from connectivity matrix
20: Reassign parent(clique) to parent(parent(clique))
21: **end if**
22: Update all Separator Set entries

Figure 2. Algorithm for parallel exact inference incorporating loop level parallelization

cost during a single clique update. For a junction tree with n cliques, using n processors, the worst-case running time is

$$(\log n) * (r^w + w * r^2) \tag{8}$$

where w is the clique width and r is the maximum range or number of states of the variable.

If p processors are used, the worst case running time is

$$\frac{n}{p}(\log n) * (r^w + w * r^2) \tag{9}$$

Since $w >> 2$, the time complexity can be reduced as

$$\frac{n}{p}(\log n) * r^w \tag{10}$$

4 Experimental Results

4.1 Computing Facilities

We used the USC SMP, the SDSC DataStar and the TeraGrid for our experiments. The SMP at USC is a SunFire 15K system. It has 64 UltraSPARC III 1.2 GHz processors.

It has a 150 MHz Sun Fireplane redundant 18X18 data, address, and response crossbar interconnect. The operating system is SUN OS 5.9 with MPICH for communication. For larger experimental runs we accessed the computing resources at the San Diego Supercomputer Center at UCSD. One of the machines is a DataStar cluster. It has 1024 IBM P655 nodes running at 1.5 GHz with 2 GB of memory per processor. The theoreotical peak performance of this machine is 15 TeraFLOPS. It uses a federation interconnect. Furthermore each channel is connected to a GPFS (parallel file system) through a fiber channel. The second large machine we ran the experiments on is the Teragrid machine - Rachel, a SMP machine. It has 1.15 GHz EV7 processors with 256 Gbytes of shared memory. It runs Tru64 Unix operating system with MPICH 1.2.6 for communication.

4.2 Experiments

We used a moderate sized input graphs of 512 and 1,024 nodes. We used three types of input junction trees: linear networks, balanced networks and arbitrary networks. The specification of the input graphs are as follows:
* $Linear\ Junction\ Trees$: Number of nodes = 512, 1024; Number of States = 2, 4, 16; maxOutDegree = maxInDegree=1; clique width = 2
* $Balanced\ Junction\ Trees$: Number of nodes = 512, 1024; Number of States = 2, 4, 16; maxInDegree=1;maxOutDegree=2; clique width = 2
* $Arbitrary\ Junction\ Tree$: Number of nodes = 512, 1024; Number of States = 2, 4, 16; maxDegree=5; clique width = 2
The junction trees were generated synthetically. We implemented tree generating codes, where the number of nodes, the number of states, maximum degree can be specified. The potential tables were populated with random numbers between 0 and 1. Functional verification of the exact inference implementation was performed by comparing results for the junction trees against the Bayesian Network Toolbox [13].

4.3 Scalability

4.3.1 Results of MPI based Implementation

Figure 3 presents the results obtained on the DataStar Cluster for the three types of junction trees: linear, balanced and random. Similar results were obtained for a 512 clique tree on the DataStar and the Teragrid [14].

4.3.2 Results of OpenMP Implementation

We explored a possible avenue for speedup by parallelizing clique level updating. We implemented the parallel exact inference algorithm in OpenMP and the execution times for the linear, balanced and arbitrary junction trees of 1024 nodes are shown in Figure 4. We also integrated top level pointer jumping with loop level OpenMP parallelization. These results are shown in Figure 5 and 6.

We observed that for a given set of input junction trees, the MPI implementation outperforms the OpenMP implementation by a factor of 1.5. We also observed that incorporating loop level parallelization in the OpenMP implementation, improves the time performance by a factor of 1.6. We got a maximum speedup of 202 for a linear junction tree, 181 for a balanced tree and 190 for a random tree.

4.4 Baseline comparisons

4.4.1 Automatic parallelization of serial code

To analyze scalability, we parallelized the serial version of the exact inference algorithm by inserting OpenMP directives and ran it on 1 to 32 processors. We compared the execution times obtained versus our implementation of the exact inference algorithm (see Figure 7). Each junction tree had 1024 nodes and 16 states. The OpenMP directives used were $pragma\ omp\ parallel$ and $parallel\ for$. When we compare the execution times we see that the parallelized serial code of the exact inference algorithm initially outperforms the implementation of the pointer jumping for exact inference. However the pointer jumping extracts data independent parallelism at the clique level, hence it is more scalable as seen in the results.

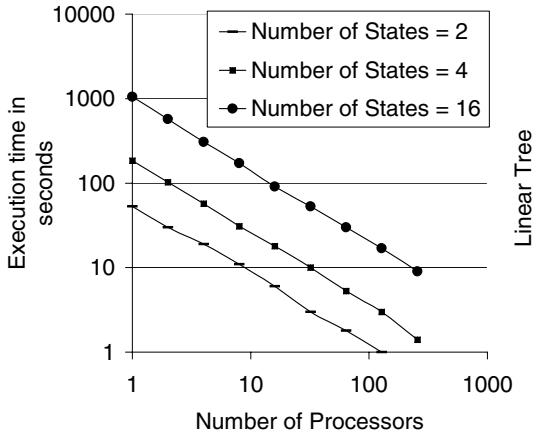
4.4.2 Probabilistic Network Library (PNL)

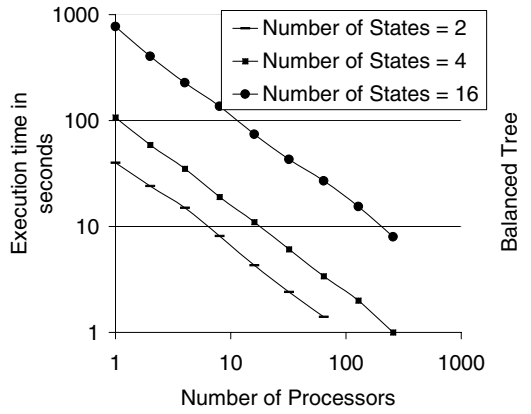
PNL[15] is a full function, free, open source, graphical models library released under a BSD style license. PNL has a parallel version which uses OpenMP.

We conducted experiments to explore the scalability of the exact inference algorithm using the PNL library. The inputs were linear, balanced and random graphs. The graph had 1024 cliques, each random variable having 16 states. We observed that the algorithm using PNL had less scalability as compared to our parallel pointer jumping technique. The results are shown in Figure 8.

5 Conclusion

We have presented an implementation of a parallel algorithm for exact inference on junction trees. It accepts an input as one evidence variable, coming in at the root of the n variable junction tree and runs in $\frac{n}{p}(\log n) * r^w$ time. We have demonstrated scalability of the parallel technique. There are several avenues for improving and extending the current implementation.





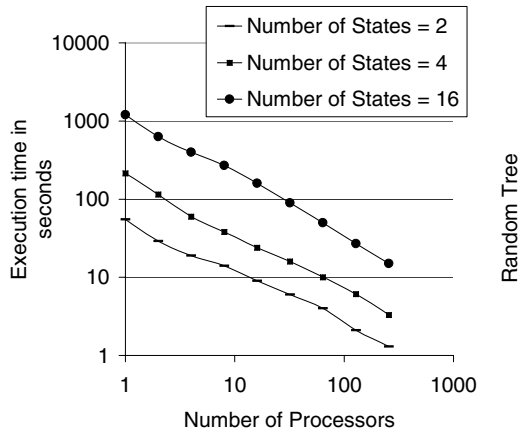
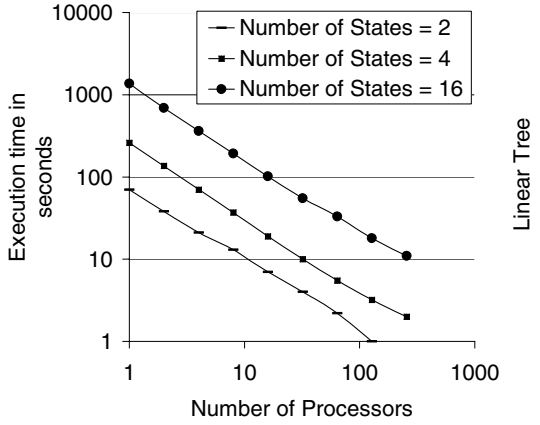
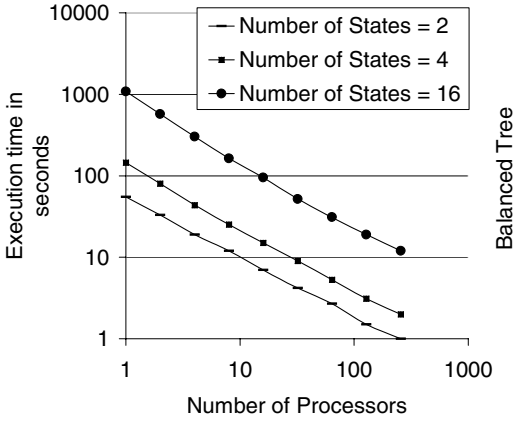


Figure 3. Execution times using MPI on DataStar





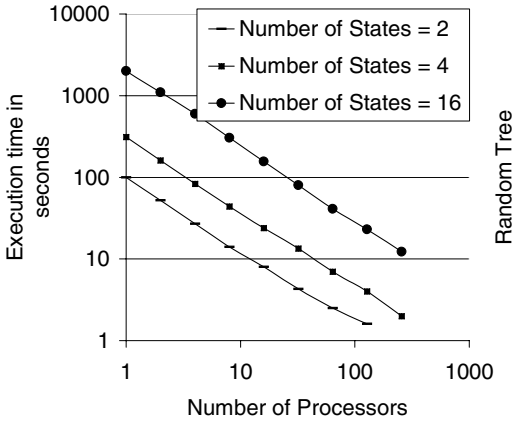
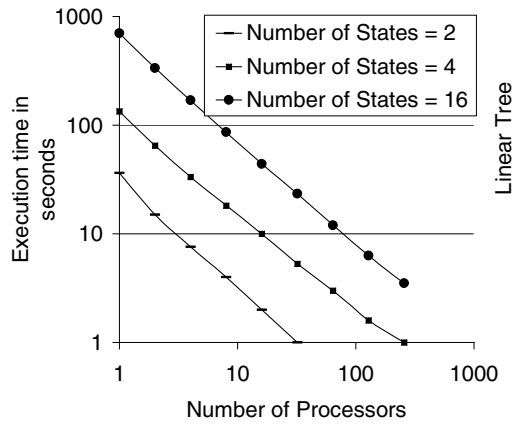
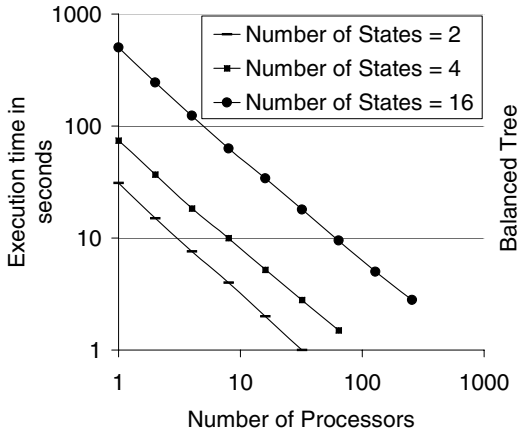


Figure 4. Execution times using OpenMP on DataStar





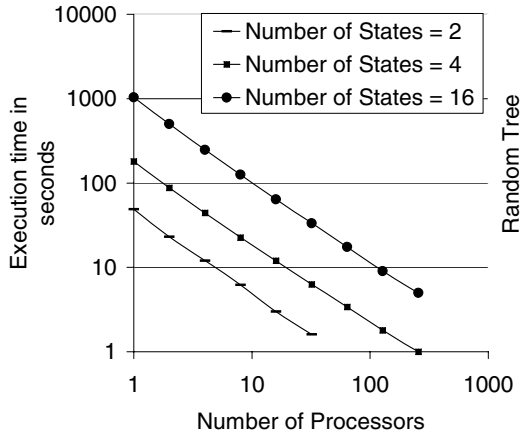
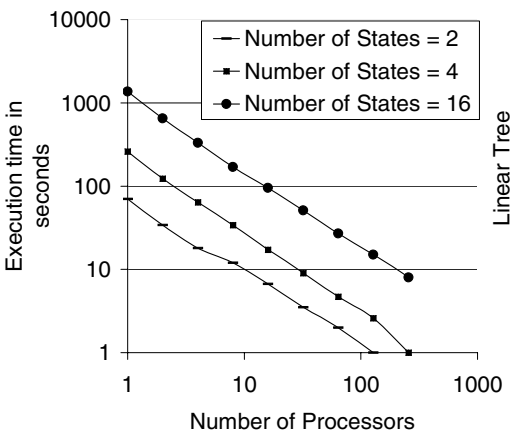
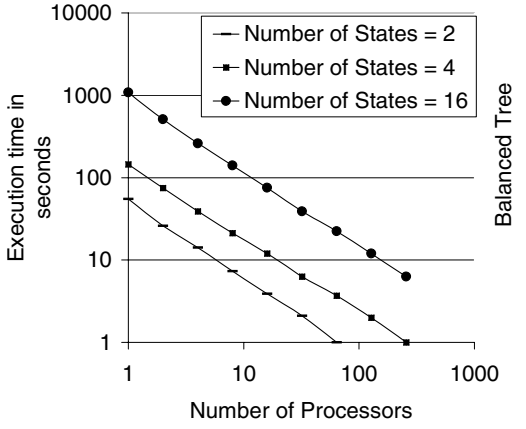


Figure 5. Execution times using loop level parallelization on DataStar





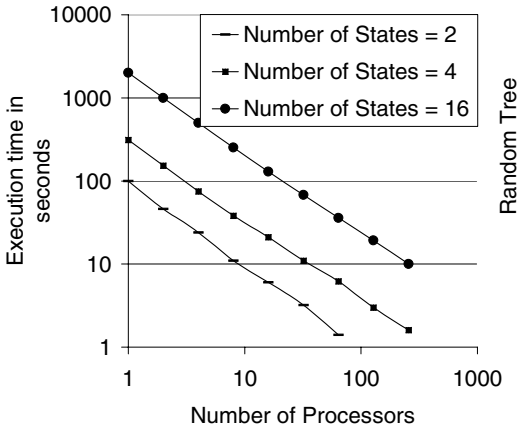


Figure 6. Execution times using loop level parallelization on DataStar

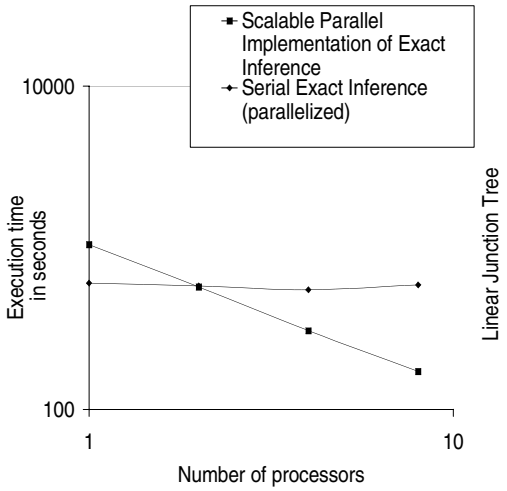


Figure 7. Execution times on linear junction tree

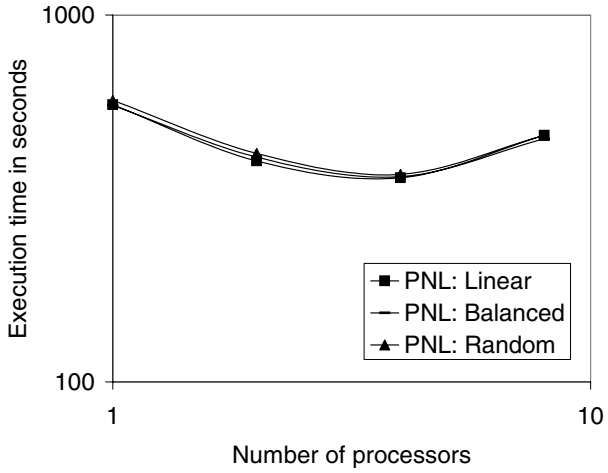


Figure 8. Execution time for exact inference using PNL

We can extend the number of the evidence variables and we need not limit them to coming at the root of the junction tree. We can also parallelize the operations involved in converting the Bayesian network to a junction tree. There are opportunities for exploring load balancing, when clique sizes are unequal. We expect to include additional results in the full version of the paper [14].

6 Acknowledgements

We would like to thank Mahidhar Tatineni, SDSC and the other consulting staff at SDSC.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. Prentice Hall, 1995.

[2] D. Heckerman, "Bayesian networks for data mining," in *Data Mining and Knowledge Discovery*, 1997.

[3] E. Segal, B. Taskar, A. Gasch, N. Friedman, and D. Koller, "Rich probabilistic models for gene expression," in *ISMB*, 2001, pp. 243–252.

[4] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," in *J. of the Royal Statistical Society*, 1988, pp. 157–224.

[5] D. M. Pennock, "Logarithmic time parallel bayesian inference," in *[UAI-98]*, USA, July 1998, pp. 431–438.

[6] B. D'Ambrosio, "Parallelizing probabilistic inference: Some early explorations," in *(UAI-92)*, USA, 1992, pp. 59–66.

[7] F. J. Diez and J. Mira, "Distributed inference in bayesian networks," in *Cybernetics and Systems 25(1)*, January 1994, pp. 39–61.

[8] A. V. Kozlov, "Parallel implementations of probabilistic inference," in *Computer 29(12)*, Dec 96, pp. 33–40.

[9] A. V. Kozlov and J. P. Singh, "A parallel lauritzen-spiegelhalter algorithm for probabilistic inference," in *SC 94*, 1994, pp. 33–40.

[10] R. D. Shachter, S. K. Andersen, and P. Szolovits, "Global conditioning for probabilistic inference in belief networks," in *(UAI- 94)*, 1994, pp. 514–522.

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[12] *Bayesian Network Generator, http://www.kddresearch.org/Groups/Probabilistic-Reasoning/resources.shtml*.

[13] K. Murphy, *Bayesian Network Toolbox, http://bnt.sourceforge.net/*.

[14] V. K. Namasivayam and V. K. Prasanna, *Scalable Parallel Implementation of Exact Inference in Bayesian Networks, http://ceng.usc.edu/ prasanna/pubs.phpallpubs*.

[15] Intel, *Probabilistic Network Library, http://www.intel.com/technology/computing/pnl/index.htm*.