# A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference

Alexander V. Kozlov
Department of Applied Physics
Stanford University
Stanford, CA 94305–4090

Jaswinder Pal Singh
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

**Abstract**

*Probabilistic inference in belief networks is a promising technique for diagnosis, forecasting and decision analysis tasks. Unfortunately, exact inference can be very expensive computationally. In this paper, we examine whether probabilistic inference can be sped up effectively through parallel computation on real multiprocessors. Our experiments are performed on a 32-processor Stanford DASH multiprocessor, a cache-coherent shared-address-space machine with physically distributed main memory. We find that the major part of the calculation can be moved outside the actual propagation through the network, and yields good speedups. Speedups for the propagation itself depend on the structure of the network and the size of the cliques that the algorithm creates. We demonstrate good speedup on a CPCS subnetwork used for medical diagnosis. This result as well as a tendency for the speedup to increase with the size of the network invites practical application of parallel techniques for large Bayesian networks in expert systems.*

## 1 Introduction

A belief network is a directed acyclic graph (DAG) with a set of conditional probabilities defined on it. The nodes in the network represent events (variables that take on certain values), and the edges represent conditional dependences of the events. Belief networks are increasingly being used in expert systems for diagnosis, forecasting, decision analysis, control theory applications, and intelligent agent modeling. The goal in using them is to capture dependencies that exist in real decision-making problems. To solve a network is to determine the conditional probability of a node or set of nodes (e.g. representing diseases), given the instantiation of evidence about the values of other nodes (e.g. findings).

Two types of approaches are taken to find the desired conditional probabilities. The first is exact algorithms, among which the two dominant ones are the Lauritzen-Spiegelhalter (LS) algorithm [1, 2, 3, 4] and Symbolic Probabilistic Inference (SPI) [5]. For a general query and network, the probabilistic inference problem is $NP$-hard [6] and exact algorithms are computationally very expensive. The second approach is based on approximate search algorithms or Monte-Carlo simulations. For the precision needed by practitioners in many applications, search methods take much less time than exact methods. However, with increasing demand for precision the exact algorithms become increasingly important and simulation methods become relatively more expensive. Besides, even though practitioners may use heuristic methods, they do not know how good those methods are unless they compare their results with exact algorithms for realistic networks. Thus, it is important to solve large networks exactly.

The large computational cost of exact algorithms makes it natural to explore the possibility of parallel computation. However, we do not know of any demonstration of successful parallel algorithms or implementations so far. At the same time, networks that take hours of computation time and are too large to be computed on a uniprocessor already exist. One of them is QMR-DT, the "quick medical reference" network used for medical diagnosis. This network uses findings from a patient to diagnose a disease and to make a decision about treatment. It consists of about 600 disease nodes and 4,000 findings nodes.

Both the SPI and LS algorithms are equally effective for exact probabilistic inference. However, while the SPI algorithm is suited more for single queries, the LS algorithm determines the distribution of conditional probabilities for all nodes in the entire network. Since the LS algorithm produces more information and is more popular, we chose to study it in this paper. We found that the LS algorithm yeilds good speedups on CPCS subnetworks, which are smaller versions of the QMR-DT network.

The rest of the paper is organized as follows. In Section 2, we present a simple example of a belief network and use it to convey the idea of probabilistic inference. Section 3 briefly describes the flow and form of the LS algorithm, and section 4 describes the Stanford DASH multiprocessor on which we performed our experiments. In Sections 5 and 6, we discuss the issues involved in parallelizing the algorithm and some approaches that we could have taken and did take. Section 7 presents our results. Finally, Section 8 provides a discussion of the results and possible future avenues for experimentation.

## 2 An Example Belief Network

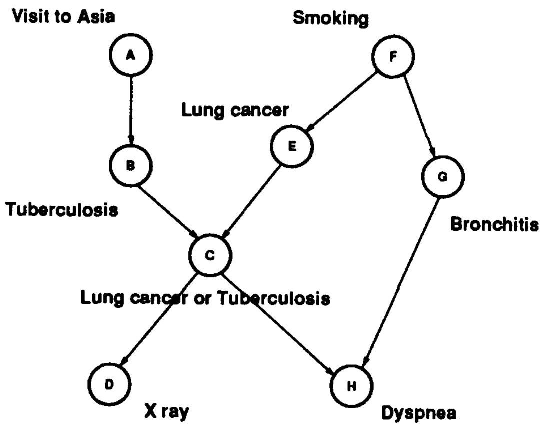

Figure 1: The "Asia" network.

A simple example of a belief network is the "Asia" network [1] (see Fig. 1), which consists of 8 nodes. We assume that each node can take two values: *true* or *false*. An edge in the network represents a probabilistic dependence expressed in terms of conditional probabilities. For instance, there are four independent conditional probabilities associated with events $c$, $g$, and $h$:

$$\begin{aligned} p(h = \textit{true} \mid c = \textit{true}, g = \textit{true}) &= 0.90\,. \\ p(h = \textit{true} \mid c = \textit{true}, g = \textit{false}) &= 0.70\,, \\ p(h = \textit{true} \mid c = \textit{false}, g = \textit{true}) &= 0.80\,, \\ p(h = \textit{true} \mid c = \textit{false}, g = \textit{false}) &= 0.10\,. \end{aligned}$$

A query one might bring to this network is to determine the probability of a parent node (disease) given evidence about child nodes (findings); for instance, what is the probability of bronchitis given that the person in question visited Asia and has a positive X-ray result. The mathematical formulation of this query is to find the value of $P(g = \textit{true} \mid a = \textit{true}, d = \textit{true})$. Given all conditional probabilities corresponding to different arcs in the network, the conditional probability for an arbitrary set of nodes can be evaluated by Bayes' theorem:

$$p(g=1 \mid a=1, d=1) = \frac{p(g=1, a=1, d=1)}{p(a=1, d=1)};$$

$$p(x_{i_1}, \dots, x_{i_k}) = \textstyle\sum_{x_i \notin \{x_{i_1}, \dots, x_{i_k}\}} p(x_1, \dots, x_n);$$

$$p(x_1, \dots, x_n) = \textstyle\prod_{i=1}^{n} p(x_i|\pi_i),$$

where 1 and 0 represent *true* and *false*, a $\{i_1, i_2, \dots, i_k\}$ is a set of $k$ nodes in the query, $p(x_1, \dots, x_n)$ is full joint probability distribution for a network consisting of $n$ nodes, and $\pi_i$ represents the set of parents for a node $x_i$. The summation is over all nodes other than $i_1, i_2, \dots, i_k$. The second equation reflects the fact that the probability of an event is a sum of joint probabilities over all other variables. And the third equation results from the definition of conditional probabilities.

In a singly connected or tree network, the summation in the second expression above is simplified since the summation over the values of a node can be done during the process of multiplication by factoring out a common multiplier. For example, if we had only nodes $a, b, c, d$ in the Asia network, the full joint probability distribution in the network would be:

$$p(a, b, c, d) = p(d|c)p(c|b)p(b|a)p(a).$$

To sum this probability over, say, nodes $b$ and $c$, we might sum over all 16 terms (each of the terms is a probability for a particular values of $a$, $b$, $c$, and $d$). To evaluate the query $p(a, d)$ we would make $48 = 16 \times 3$ multiplications (3 is the number of multiplications per term). This can be done faster by factoring out probabilities $p(b|a)p(a)$ before summing over node $c$. The minimum number of multiplications for the evaluation of the $p(a, d)$ query is 20. Any query can be calculated by factoring out probabilities. Different factoring strategies represent different algorithms for exact probabilistic inference. In one of them, calculation of a query is represented by message passing up and down the network [7].

```
    for(i = 0; i < gp->clique.numChildren; i++) {
       gc = &globTree->globCliques[gp->clique.childCliqueIndices[i]];
       if(gc->clique.numSeps) {
         if(gc->msgIndices)
           for(j = from; j < to; j++)
             gp->clique.potentials[j] *= gc->clique.msg[gc->msgIndices[j]];
           else
             for(j = from; j < to; j++)
           gp->clique.potentials[j] *= gc->clique.msg[MsgIndex(j, gp, gc)];
       }
 }
```

Figure 2: A piece of code for the in-clique calculations.

In [7] the evaluation of a query in a tree network consists of sending so-called $\lambda$ messages from the bottom of the tree to the top and $\pi$ messages from the top to the bottom, modifying and accumulating node properties along the way. The message upward is a summed product of all probabilities below the given node and the message down is information for updating the node prior probabilities. The message passing algorithm is not so straightforward when a network has a loop. For example, if none of the nodes in the cycle C–E–F–G–H in Fig. 1 is instantiated with evidence, a message from a node can eventually come back to itself during the upward pass, which makes the summation wrong. For networks with loops inference can be propagated by converting a network to a *join* tree, and then sending messages up and down the join tree. A join tree is a tree of sets in which if a node is in set 1 and set 2, it is also in all sets between these sets [3]. One of the exact algorithms that does probabilistic inference in a multiply connected network is the LS algorithm, which we shall outline in the next section.

## 3 Lauritzen-Spiegelhalter Algorithm

The LS algorithm first converts a multiply connected network into a tree of *clusters*, or sets of nodes. As mentioned earlier, inference propagation is simple in a tree. Probability distributions are calculated for the clusters during a message propagation process. After the propagation, the individual node probabilities are calculated from the cluster distributions.

The process of cluster tree formation consists of a few transformations to the original network to build the necessary structure. First, we drop the directions of the edges. Then, we connect parents of the same child, or *moralize* the graph. After that, we *triangulate* the resulting network and find the ranks of vertices according to a maximum cardinality search. Finally, we determine cliques, complete fully interconnected subsets, in the obtained network and the corresponding clique tree. Moralizing the network assures that a probability distribution for a node and its parents will belong to a single clique. Triangulation eliminates cycles of length four or more. Maximum cardinality search helps to find a clique ordering with a running intersection property: the nodes of a clique that are also contained in previous cliques are all contained in *one* previous clique (which then becomes the parent of the first clique in the clique tree):

$$C_n \cap \left(C_{n-1} \cup C_{n-2} \cup \cdots \cup C_1\right) \subseteq C_i$$
$$1 \leq i \leq (n-1).$$

The above nodes of a clique that are also contained in a previous clique constitute the *separators* of the clique, $S_n = C_n \cap (C_{n-1} \cup C_{n-2} \cup \cdots \cup C_1)$. The separators determine the size of the message that the clique will communicate with its parent. The rest of the nodes in the clique constitute the clique *residual*, $R_n = C_n \backslash S_n$ ($\backslash$ is substraction for sets). For each clique we can write:

$$p(C_1 \cup \cdots \cup C_n) = p(R_n|S_n)p(C_1 \cup \cdots \cup C_{n-1}),$$

where the distribution, which is often called *potentials*, for each clique is normalized.

The propagation of messages is very similar to that in a tree of nodes. The difference is that each message now contains $v^{|S_n|}$ numbers and a clique has $v^{|C_n|}$ potentials, where $v$ is the number of values that a node variable (event) takes. Updating potentials consists of multiplying potentials by messages, and calculating a message up or down consists of summing potentials by residue indices (Fig. 2). In the code, potentials of a clique from index `from` to index `to` are updated by corresponding messages from the clique children. If the message indices are precomputed, the clique potential is multiplied by a message whose index is stored in a `msgIndices[]` array. Since the operation of updating a potential is just a single multiplication by a message from the message array, good speedups through parallelism on general-purpose machines are achievable only on networks in which larger independent tasks can be created. There are two possible levels at which independent tasks can be generated: across the cliques (topological parallelism) and within the cliques (in-clique parallelism). Before we discuss our parallel implementation that exploits both types of parallelism (in both cases with and without index preevaluation), let us describe the multiprocessor that we run our experiments on.

## 4 The Stanford DASH multiprocessor

The DASH machine [8] is a state-of-the-art, experimental multiprocessor built at Stanford University. It supports an implicit shared address space communication abstraction in hardware (as opposed to explicit message-passing as in the Intel iPSC's and Paragon or the Connection Machine CM-5), with hardware-supported cache-coherence as well. The shared address space abstraction greatly simplifies the parallel programming task, particularly for irregular problems that require dynamic partitioning and scheduling such as the one considered in this paper. The two machines we used have 16 and 32 processors organized in 4 and 8 clusters.[1] A cluster comprises 4 MIPS R3000 processors connected by a shared bus, and clusters are connected together in a mesh network. Every processor has a 64KB first-level cache memory and a 256KB second-level cache, and every cluster has an equal fraction of the physical memory on the machine. That is, memory is logically shared but physically distributed. All caches in the system are kept coherent in hardware using a distributed directory-based protocol. Our parallel programs are written in C using the **parmacs** macro package from Argonne National Laboratories for parallelism constructs.

[1]The prototype actually has 64 processors in 16 clusters, but is broken up into separate machines in usual operation.

## 5 Available Parallelism and Parallelization Strategies

A program implementing the LS algorithm (we obtained our serial C program from Adam Galper at Stanford Medical School) can be divided into four phases.

- **Clique Tree Generation (CTG)** – includes moralization, triangulation, clique tree formation and generation of the clique potentials
- **Clique Tree Precomputation (CTPC)** – preliminary computation and initialization of the copy tree (the algorithm requires two copies of a tree, one called the permanent tree and the other the copy tree; all propagations and modifications for a given instantiation are done in the copy tree)
- **Clique Tree Propagation (CTP)** – message propagation in the tree, up and down
- **Belief Computation (BC)** – calculation of the node probabilities from the clique distributions

The phases that take most of the time are CTPC and CTP. These are the focus of our parallelization effort since parallelizing most of the other parts is easy.[2] The CTPC part is the initialization required for the propagation part, CTP, that follows.

CTPC includes copying the tree in the global memory, initializing the copy tree, and precomputation of message indices array. The last part is optional in our code. The sequential code we received computes message indices, mentioned above, on-the-fly during the propagation phases. This program therefore computes the same message indices both times that it sends a message (up and down). Since the values of the indices depend only on the composition of the tree, which is known before propagation, it is possible to preevaluate and store the messages during the CTPC phase and simply look them up during the propagation. The computation of message indices takes a large portion of the serial computation time. Since the precomputation of indices outside the propagation represents completely independent tasks for each clique, it is very parallel. In fact, it speeds up much better then the actual propagation of messages. The drawback of this approach is the memory requirement for storing the precomputed indices. The array of message indices is

[2]Although finding the triangulation which adds the minimum number of arcs in CTG has been known to be *NP*-complete [9], heuristic algorithms with $O(n+e)$ computation time exist, where $n$ and $e$ are the number of nodes and edges in the network.

stored with the relevant clique. The size of the indices array is equal to the rather substantial size of the parent potential array, and hence can more than double the memory requirements of the program. The exact memory overhead depends on the topology of the join tree.

When indices are preevaluated during CTPC, the actual propagation part, CTP, contains only the necessary multiplications and summations for the clique potentials and messages. Parallelism can be exploited at two levels in this phase. First, inference propagation in the clique tree can be parallelized across cliques, since cliques that are not related by an ancestor-descendent relationship can be processed in parallel. This is called topological parallelism. The degree of topological parallelism available depends on the branching factor of the clique tree and the amount of work in different branches. If the clique tree is bushy, the cliques can be partitioned into groups of distinct subtrees. We will later see that a network of practical importance, the CPCS network, satisfies this condition. Since networks may not have a bushy enough topology to provide enough parallelism across cliques, we considered a second form of parallelism, parallelism of the computations within the cliques.

The computation within a clique consists of updating potentials by messages or calculating messages from clique potentials. The computational cost of these calculations is typically not very high. On average, updating potentials or calculating messages takes only a few simple operations per potential. With modern processors this takes only a few clock cycles and the major contribution to the running time is memory access time. Nontheless, parallelization within cliques can be important when cliques are large. Since the belief networks of practical interest are too large to fit into a single cache or local memory, preserving locality of data reference during propagation is potentially very important.

## 6 The Parallel Implementations

The critical issues for a parallel implementation are load balancing and data locality. The size of the potential array in a clique varies exponentially with the size of the clique. Typically the size of a clique varies from two nodes to tens of nodes. Consequently, the work in different cliques varies by orders of magnitude and it is difficult to obtain load balance performance for a general network by distributing the workload statically among processors. The solution is dynamic task scheduling.

If only topological parallelism is desired, the sequence of leaves (in the traversal of the tree) is placed into a shared array in the shared address space. Access to the array is protected by multiple locks, one per entry, to avoid excessive centralization and contention. A processor first checks for available entries in the leaf array. If an entry is available, the processor retrieves it and begins working along the corresponding new branch in the tree. The entry search is organized in a way to preserve locality. A processor chooses a branch which is close to the one it has just been working on, so that different processors work on different subtrees as far as possible. The calculations in different branches of the clique tree are totally independent.

The leaf array implements topological parallelism, which, as we shall see, can be very limited for random networks. Better load balancing can be achieved only by reducing the size of the tasks that form the units of concurrency, i.e. by dividing the calculation inside a given clique between different processors. The partitioning and scheduling of tasks for load balancing becomes a lot more dynamic now, and is managed by a dynamic task queueing system. One way to implement dynamic task queueing is to use a single global task queue that all processors access to obtain and insert tasks. This introduces contention at the global queue, particularly if tasks are small, and does not preserve locality of data reference across tasks. These problems can be alleviated using distributed queues withtask stealing. In this method, every processor has its own local queue. Calculations on a clique are split into chunks and placed in the local queue. Before the topological parallelism is exhausted all tasks in the local queue are executed by the owning processor. If a processor finds no available entries in its own queue, it steals tasks from other processors' queues to balance the workload at some cost in data locality. Data locality is preserved since most of the time a processor accesses its own queue. To maximize the data locality, a processor remembers the pointer to the last stolen task and accesses the same queue the next time.

Consider the data flow pattern in Fig. 3. The message from each of the children updates potentials of the parent, and the potentials of the parent are used to calculate the message to the next level of the hierarchy. For simplicity let us assume that there was enough topological parallelism that the children of the clique in Fig. 3 were calculated on different processors 1, 2 and 3. The message from child 1, 2, and 3 in this case will be in the local memory of the corresponding processor. Suppose the size of the potential array is big enough that we can divide it into chunks

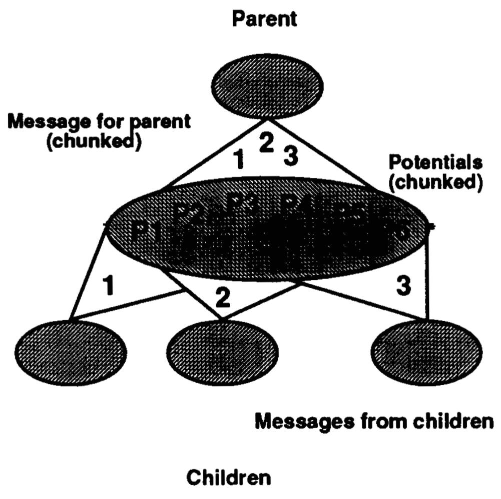

Figure 3: When a potential is updated by a message, message form each child updates each of the potentials in the parent. Message up is calculated based on all potentials. Dedicating a group of processor to working on only this group of messages or potential gives the best data locality.

P1, P2, P3, P4, P5 and P6. There are two ways to divide the potential calculation into tasks that are assigned to processors: one is to create a task for each chunk of potentials; the other to create a task for each chunk of message(s). In the first case, the processor assigned a chunk of potentials is responsible for updating that chunk with all incoming messages; in the second the processor assigned a message uses it to update all "chunks" of potentials. The first solution has a number of advantages. First, since the potential array is as a rule bigger than a message array, the overhead of communicating messages is smaller than that of of communicating potentials. Second, since each chunk of potentials is assigned to only one processor, locking of the potentials is not necessary and the synchronization overhead is significantly lower. Finally, if the communicated message array is small, it might remain in the cache for the next task picked by the processor.

The optimal chunk size varies with the size and structure of the network. The optimal chunk is smaller for smaller networks as well as for propagation close to the top of the tree (with less available topological parallelism). In our implementation the chunk size changes dynamically depending on the size of the clique, but it is always bigger than some minimum size. The minimum size is a parameter in the code which, after some experimenting, we set to 1024 potentials and 64 messages per task.

Our implementation thus exploits both topological and in-clique parallelism. In the absence of task stealing (which is an option in the implementation) only topological parallelism is exploited. The implementation also tries to exploit as much data locality as possible.

The structure of the join tree is by and large determined by the problem. In many cases load balance can be improved by constructing the join tree can be constructed with the view of parallelization ahead. In general the problem is analogous to construction an optimal evaluation tree [5] and is computationally expensive. In the current implementation, we have chosen a simple way to construct a tree which facilitates speedup. Each clique was prescribed a computational cost. In the current version computational cost has been calculated as a simple sum of the number of potentials, messages, and some constant determined by the overhead of starting a computation on a clique. Since edges in the tree can be reversed in a way that any node can be made the root of the tree, we balanced the workload by chosing a proper root of the tree which gives the best initial load balance for all its children.

Parallelization of the message indices precomputation is easy with this task management system. To calculate the mapping between potentials and messages we only need a few parameters from both cliques (child and parent) which easily fit in the local cache. Calculations are independent of each other and therefore are well managed by distributed queues. In the absence of memory allocation for the copy tree and the copy tree initialization the CTPC part would have given perfect speedup.

## 7 Results

Clique trees can be divided into two types: small fan-out (few children per node) and large fan-out (many children per node). In the first type of tree, parallelism can be mostly exploited only within cliques; in the second type, topological parallelism is available as well. While topological parallelism does not require much communication between different processors, parallelsm within cliques is more closely coupled and its efficiency is therefore questionable. Our measurements show that larger speedups are available in the clique trees with larger fan-out.

Table 1: Parameters of the completely random networks used to test the parallel LS program. $n$ – number of nodes, $e$ – average number of edges per node, $v$ – number of values per node, $c$ – number of cliques, $l$ – number of leaves in the clique tree, $\max|C_i|$ - maximum size of a clique, $\max|S_i|$ - maximum size of separator set, Memory – global memory required in Mb, Time – uniprocessor execution time for (CTPC + PC) part in seconds. The networks were generated with J. Suermondt's code.

| network | $n$ | $e$ | $v$ | $c$ | $l$ | $\max\|C_i\|$ | $\max\|S_i\|$ | Memory | Time |
|---|---|---|---|---|---|---|---|---|---|
| A | 54 | 2.2 | 2 | 31 | 16 | 21 | 17 | 51 | 351 |
| B | 18 | 2.2 | 4 | 9 | 6 | 10 | 9 | 45 | 241 |
| C | 54 | 2.0 | 2 | 34 | 15 | 17 | 16 | 13 | 85 |
| D | 80 | 1.4 | 2 | 55 | 29 | 16 | 14 | 3.1 | 19 |
| E | 18 | 2.0 | 4 | 10 | 5 | 8 | 6 | 3.9 | 13 |
| F | 54 | 1.7 | 2 | 38 | 20 | 14 | 12 | 1.0 | 7.2 |
| G | 18 | 1.4 | 4 | 12 | 6 | 10 | 8 | 0.1 | 0.16 |

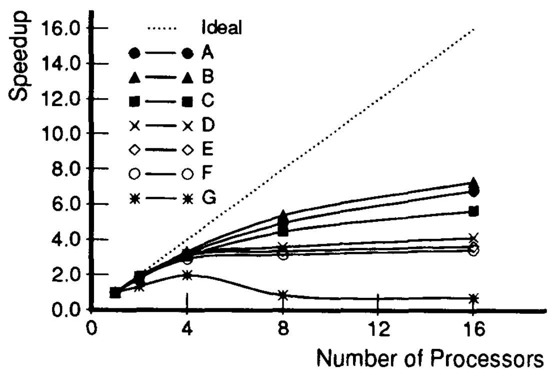

Figure 4: Speedups for different random generated networks.

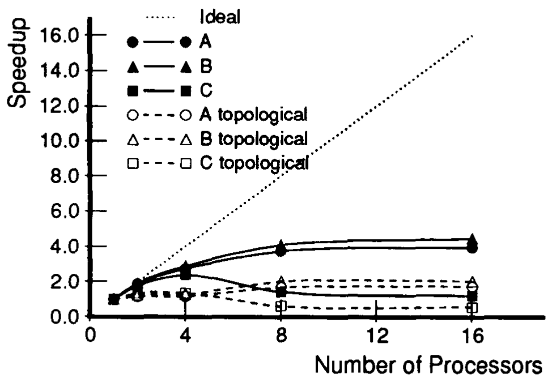

Figure 5: Topological and total speedups of the CTP part for selected networks: A, B, and C

## 7.1 Random networks

Since our implementation does not assume anything about the structure of the tree, we first test it with seven small random networks that we generated artificially with J. Suermondt's program.[3] The random networks are sorted according to the uniprocessor computation time in Table 6. A clear correlation exists between the memory requirement and the execution time of a network.

The speedups for the (CTPC + CTP) phases of the program are given in Fig. 4. The larger network speedups are better for two reasons. First, the larger networks have more topological parallelism. Second, the cliques in the tree are larger and the calculations within cliques can be parallelized more efficiently. The performance of the smallest network G decreases beginning with 4 processors due to larger ratio of the communication and synchronization time to the computation time.

[3]This program takes a fully connected network and removes nodes till it meets given specifications. The program was obtained from Bruce D'Ambrosio.

While the CTPC phase usually takes an order of magnitude more time than the CTP phase and has a large positive impact on the overall speedup, it is interesting to look at the speeding up of the CTP phase in isolation. This speedup is much lower (Fig. 5) since the CTP phase is much less parallel. On the same plot we show speedups with only topological parallelism. The available topological parallelism in the

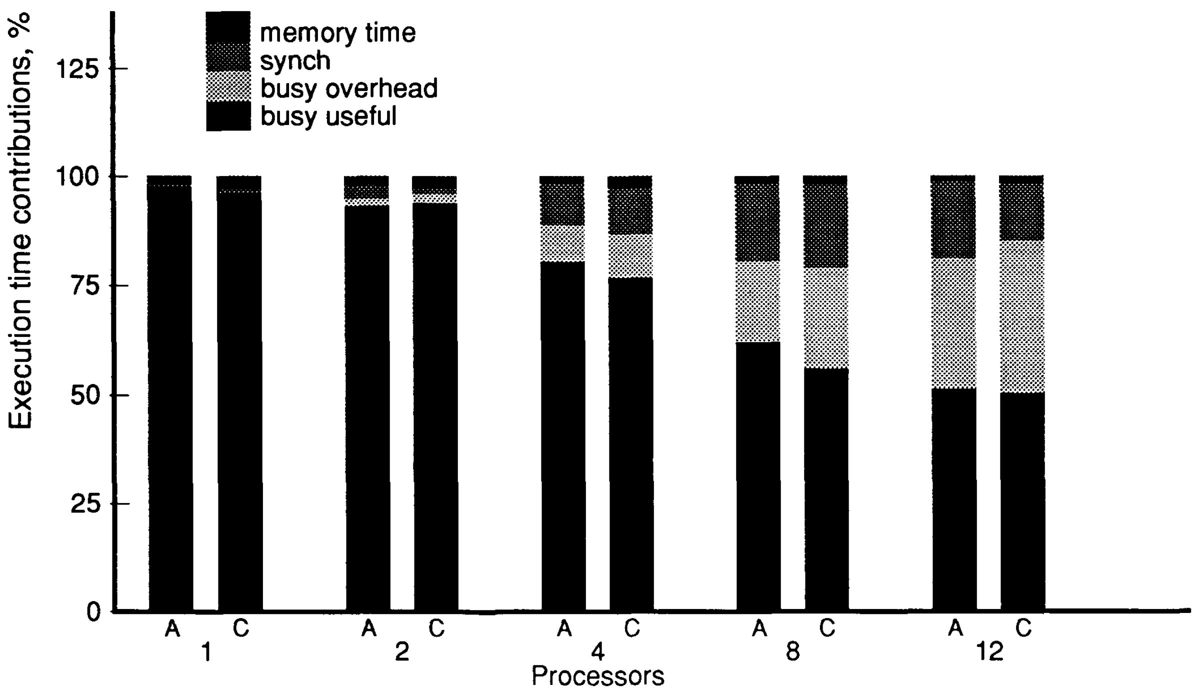

Figure 6: Contributions to the execution time for the A and C networks.

random generated networks is very low, and the in-clique speedup significantly contributes to the total speedup. Even though these networks each have up to 16 different branches (leaves), the work is concentrated in the top part of the tree, where only in-clique parallelism is present. For example, parallelization within cliques is the major contribution to the better performance of the network A compared to the network C (Fig. 4). These networks have almost the same structure (number of cliques and leaves), but the clique sizes of the network A are larger. The available topological parallelism is exhausted already at 4 processors.

In Fig. 6 we use the hardware performance monitor on DASH to examine the breakdown of execution time into time spent doing useful work (busy useful), time spent doing extra work in the parallel program that did not need to be done in the serial program (busy overhead), time spent waiting at synchronization points (synch) and time spent stalled on second-level cache misses (the time a processor spends stalled on first-level cache misses that are satisfied in its second-level cache is included in busy time, since we cannot measure it with the performance monitor). We used a uniprocessor execution time as the busy useful time. The breakdown shown is for the A and C networks. The execution time is normalized to 100% for each number of processors, so that we examine the percent of parallel execution time spent in each component, averaged over all processors. The contribution of the memory stall time is low, showing good data locality in the calculations. Memory-remote time and hence interprocessor communication is found to not be large. Synchronization time, however, is a fast-growing source of overhead.

The major source of performance loss is busy overhead. Detailed measurements of the program, particularly of the CTP phase, shows that the largest contribution to the synchronization and buzy overhead times is time spend on waiting for the entries in the queues. This corresponds to calculations at the parts of the tree where parallelism is absent. In this case, given the small size of a clique, all other processors have to wait till one of them finishes its work. In

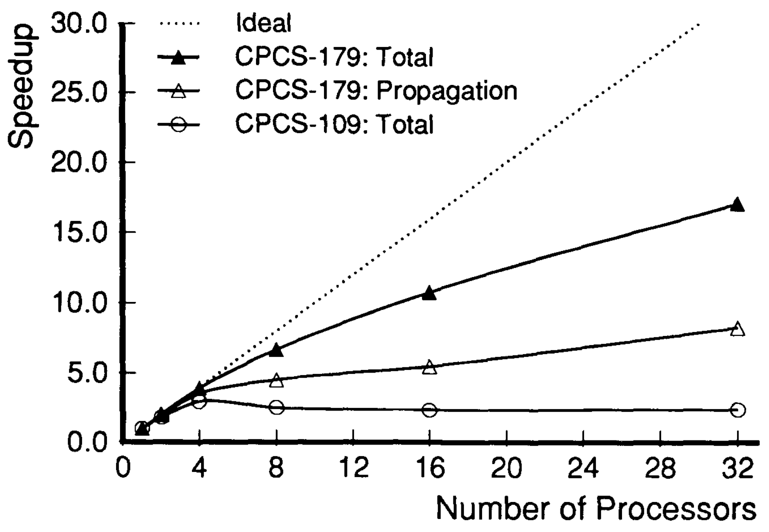

Figure 7: Speedups for the CPCS-109 and CPCS-179 subnetworks.

larger networks with either bigger size of the cliques or larger number of smaller cliques, load balance should be better.

For the artificial networks we generate, then, we find that speedups depend heavily on network topology as well as clique size. Speedups are good for networks with a lot of topological parallelism, and relatively poor for networks with random structure, worse for networks with small cliques (network C) than those with large cliques (network A). Let us now examine the more interesting situation of a real network, the CPCS medical diagnostic network, to see how these effects play out in it.

## 7.2 CPCS network

The structure of a network is by and large determined by the problem but to some extent can be constructed with parallelization in mind. The join tree formation procedure can be done in a variety of ways and also can hinder or facilitate a better speedup. The question of the join tree construction was not in the scope of this paper. The encouraging fact in our simulations is that speedup is growing with the size of the network.

Since it is difficult to get a speedup for a general network, we tried to analyze a network of practical importance, the CPCS (Clinical Patient Case Study) network. The network is used for internal medicine diagnosis of a patient with abdominal pain. The CPCS network contains 448 nodes and over 900 edges and has never been solved exactly. The obstacle is the memory required to solve it.

A few representative subnetworks of CPCS which can be solved exactly have been created. The largest available ones are CPCS-109 and CPCS-179, consisting of 109 and 179 nodes. The uniprocessor belief propagation time for the CPCS-109 and CPCS-179 is 3.5 and 236 second respectively on DASH. Figure 7 plots the speedups for the whole propagation (CTPC+CTP) for both networks as well as speedup of the propagation part (CTP) for the larger network. The running time is reduced to about 10 seconds on 32 processors for both networks.

## 8 Discussion and Conclusions

We have reported early results on the parallelization of the Lauritzen-Spiegelhalter algorithm for probabilistic inference in belief networks. The efficiency of parallelization in the propagation phase of the algorithm depends on the topology of the clique tree as well as the size of the cliques. However, good overall speedups are achieved since the major part of the computation is independent of the topology and is parallel. Speedups are better for trees with large fanout since computations in different clique can be done independently. This corresponds to parallelization of the conformal product evaluation at the topological level. Speedups are also better when cliques are large, in which case parallelism can be efficiently exploited within cliques. These facts are encouraging since they suggest that speedups will be better for more expensive networks, and these are the ones where speedups are more important.

We have demonstrated good speedups on a medical diagnosis network (the only real network we had access to) on a 32-processor Stanford DASH machine. The architecture of the machine (in particular, its support oof cache coherent shared address space) simplified the programming task and optimized the data exchange. Future work includes obtaining and experimenting with larger networks than the ones examined in this paper, and using better synchronization and partitioning strategies that reduce the number of queue accesses needed. We would also like to explore the relationship between the clique tree generation and parallel performance. The clique tree generation might be tailored to produce trees with larger fanout. However, this is only useful if it doesn't add too much overhead to the generation phase. Finally, we would like to explore the parallelization of approximate probabilistic inference algorithms, and examine the tradeoffs in using exact versus simulation-based methods for real networks given the availability of parallel machines.

## Acknowledgements

We would like to thank Adam Galper, David Heckerman, Max Henrion, Eric Horvitz and Ross Shachter for discussing the area of probabilistic inference with us, Bruce D'Abmrosio and Adam Galper for providing us with networks and sequential code, and Greg Provan and Malcolm Pradhan for providing us with the CPCS subnetworks. We would also like to thank John Hennessy for his support, and DARPA for its financial support under contract no. N00039-91-C-0138.

## References

[1] S. S. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, B 50:253 - 258, 1988.

[2] Richard E Neapolitan. *Probabilistic Reasoning in Expert Systems*. Wiley, New York, 1990.

[3] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269 - 282, 1990.

[4] Finn Verner Jensen, Kristian G. Olesen, and Stig Kjaer Andersen. An algebra of bayesian belief universes for knowledge-based systems. *Networks*, 20:637 - 659, 1990.

[5] Zhaoyu Li and Bruce D'Ambrosio. Efficient inference in bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 1994. to be published.

[6] G. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42:393 - 405, 1990.

[7] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of plausible inference.* Morgan Kaufmann Publishers, San Mateo, California, 1988.

[8] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148 - 159, May 1990.

[9] M. Yannakakis. Computing the minimum fill-in is *NP*-complete. *Journal of Algebraic and Discrete Methods*, 2(1):77 - 79, 1981.