Submitted to the special issue of IEEE Computers (December 1996)
"Applications for Shared-Memory Multiprocessors"

Parallel Probabilistic Inference on Cache-coherent Multiprocessors

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

phone: *(415) 725-8814*
fax: *(415) 725-1449*
email: *alexvk@cs.stanford.edu*

Jaswinder Pal Singh
Department of Computer Science
Princeton University
Princeton, NJ 08544

phone: *(609) 258 - 5329*
fax: *(609) 258 - 1771*
email: *jps@cs.princeton.edu*

Abstract

Probabilistic inference is a technique used in expert systems for reasoning under uncertainty. A typical inference task is to determine the probability of some events (say diseases) given evidence about other events (say findings). Inference is conveniently represented as the propagation of evidence in a graph called a belief network. Probabilistic inference is computationally very expensive (NP-hard in general), but is very important both in itself and to calibrate less expensive approximate schemes. It is therefore natural to explore speeding up inference exploiting parallelism. Probabilistic inference affords concurrency at two different levels and presents interesting tradeoffs between load balance and data locality. We present two parallel implementations of probabilistic inference in belief networks. One uses a static assignment of work to processors but sacrifices some available concurrency, while the other uses dynamic assignment to obtain both forms of concurrency but sacrifices some types of data locality. We provide detailed performance measurements and analysis for both implementations on two cache-coherent shared-address-space multiprocessors: a 32-processor Stanford DASH multiprocessor with physically distributed main memory, and a 16-processor bus based SGI Challenge XL with centralized main memory. We find that the static assignment scheme produces better results uniformly over all input networks on both machines since it has better locality properties. To understand scalability with problem size, number of processors and cache organization, we perform software simulations of the multiprocessor execution and provide a detailed characterization of the spatial and temporal data locality. We find that temporal locality is low due to both large size of the working sets and low reuse of data, while spatial locality is high. Our results suggest that probabilistic inference is both successful on moderate-scale cache-coherent machines, as well as a good benchmark for testing the memory and communication architectures of these machines due to its emphasis on the data locality.

Keywords: shared-address-space machines, expert systems, probabilistic inference, exact probabilistic inference, parallel programming, task scheduling, data locality.

Submitted to the special issue of IEEE Computers (December 1996)
"Applications for Shared-Memory Multiprocessors"

1 Introduction

Probabilistic inference is a technique used in expert systems for reasoning under uncertainty, i.e. when a definite logical conclusion is impossible due to the absence of complete information. The inference engine holds information about events and the dependencies among events—expressed as conditional probabilities—and answers queries about some events given evidence about other events. The output in such an engine is the probability of the event(s) specified in the query, and the input is evidence about observed events that may be relevant to the query. For example, in a medical diagnostic expert system the output is the probability of a disease, and the input is the results of medical exams and patient information. The events and dependencies between events are usually represented by a graph called a *belief network*. Probabilistic inference has been applied in medical diagnosis, speech recognition, automated vision, software fault diagnosis, manufacturing yield diagnosis, spacecraft fault diagnosis, scheduling and other areas. It's potential for application is increasing rapidly as more and more areas incorporate the need to make decisions under uncertain circumstances.

Although probabilistic inference is the only self-consistent[1] technique for reasoning under uncertainty, it is rarely used directly in practice because of its computational complexity (*NP*-hard in general). In practical systems, users often resort to different heuristics for approximate inference, or to exact inference in a subnetwork extracted from the network representing the full problem. Since the accuracy and effectiveness of these methods are unclear, performing exact probabilistic inference on realistic problems is important not only in itself, but also to evaluate approximate methods by comparison. Moreover, many structural features of exact inference algorithms are also features of approximate inference techniques [3], so many of the results obtained for speeding up exact inference can carry over to speeding up approximate inference as well. Parallelism is a promising avenue for speeding up both types of inference. Thus, it is important to understand how well the performance of exact probabilistic inference can be improved through parallelism. Since we focus on exact inference in this paper, we shall henceforth use probabilistic inference to mean exact probabilistic inference.

Probabilistic inference is a very data-intensive application. Unlike many scientific applications for high performance computing, where the number of operations grows as $O(n^{3/2})$ (as in many linear algebra algorithms [4]) or as $O(n \log n)$ with large constants (as in hierarchical N-body methods [5]), where n is the size of the data set, the number of operations to perform probabilistic inference is only a small constant multiple of the data set size (both the data set size and the number of operations can grow exponentially with the number of nodes in the belief network). Thus, managing access to data in an architecture-sensitive way is crucial to good performance on modern uniprocessors as well as multiprocessors. The well-known trade-off between the load balance and data locality is heightened in a parallel implementation of probabilistic inference, both because of the irregular structure of the belief networks and because of its data-intensiveness.

We present and evaluate two different parallel implementations of probabilistic inference. One uses a static assignment of work to processors but sacrifices one of the two available forms of concurrency, while the other uses dynamic assignment to obtain both forms of concurrency but sacrifices some types of data locality. We provide performance measurements for both implementations on two cache-coherent shared-address-space multiprocessors: a 32-processor Stanford DASH multiprocessor with physically distributed main memory, and a 16-processor bus-based SGI Challenge XL with centralized main memory. We show that good speedups can be obtained with realistic networks on these moderate-scale multiprocessors when an appropriate partitioning technique is used, even though we start from a highly optimized uniprocessor implementation. In particular, we find that the static assignment scheme that sacrifices some concurrency and load balance performs substantially better than the dynamic scheme that compromises some locality. In fact, the importance of data management makes probabilistic inference a good benchmark for the memory architectures of multiprocessors. We provide a detailed breakdown of performance bottlenecks, using available performance monitoring tools. We also provide a quantitative characterization of the temporal and spatial locality in the parallel programs, using simulation to understand how they might interact with different memory system organizations.

The rest of the paper is organized as follows. In Section 2 we give a short overview of the probabilistic inference method. In Section 3 we provide some of the details of our uniprocessor implementation, which has been carefully designed and tuned to be close to optimal. In Section 4 we briefly describe the Stanford DASH and SGI Challenge

[1] By self-consistency we mean that a robot using this technique will be consistent in its actions. As was shown [1, 2], any other self-consistent technique has to be isomorphic to the probabilistic method. For a more detailed definition of consistency and the proofs see [2].

XL multiprocessors. In Section 5 we discuss the available parallelism, and in Section 6 possible parallelization strategies. In Section 7 we provide the results of our performance measurements and locality characterization. Finally, in Section 8 we summarize our results and conclude.

2 Probabilistic Inference in Belief Networks

Probabilistic inference answers a question, called a *query*, about the probability of some events given evidence about other events. The events and the dependencies between events are usually represented by a directed acyclic graph, called a *belief network*. Nodes in a belief network denote events and edges denote the dependencies between events. Every node can have a finite number of values, corresponding to possibly observable outcomes of the corresponding events.

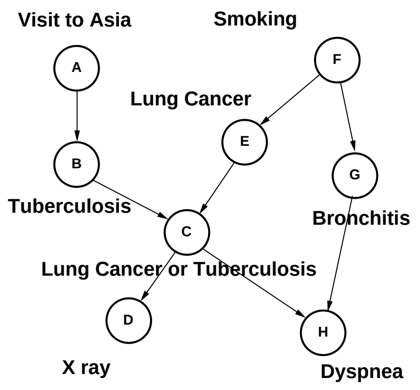

Figure 1: The "Asia" network. Tuberculosis and lung cancer are equally likely to cause shortness of breath (dyspnea), and are also equally likely to cause a positive chest X-ray. Bronchitis is another cause of dyspnea. Tuberculosis is more likely in people who have visited Asia. Smoking can cause lung cancer and bronchitis. All nodes are binary (can have only two possible values).

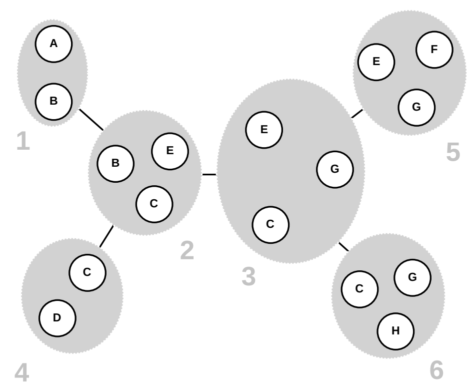

Figure 2: A join tree for the "Asia" network. Nodes in the original network can appear in several adjoining cliques in the join tree. All conditional probabilities in the original networks are wholly contained within one of the cliques.

For example, Fig. 1 shows the well-known "Asia" network [6], all of whose nodes have two possible values. For example, the node x_F ("Smoking") has values "Smokes" and "Doesn't smoke". The dependencies between nodes are given by directed edges: We believe that the probability of lung cancer is likely to be affected by smoking, but the probability of visiting Asia is not. The probability of lung cancer is not affected by the fact of having tuberculosis, provided that we do not have any information about the node "Lung Cancer or Tuberculosis". We associate a conditional probability $p(x_i|Pa(x_i))$ with each node in the network,[2] where $Pa(x_i)$ denotes the set of parents of the node x_i. That is, $p(x_i|Pa(x_i))$ is a set of numbers each being the probability that node x_i takes one of its possible values given that each of its parents takes a certain of its possible values. Then, we can write the joint probability distribution for the n nodes in the network as:

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i|Pa(x_i)). \tag{1}$$

The joint probability distribution assigns a probability to the observation of each particular combination of values for all the nodes in the network (e.g. the probability that node x_A is in the state "Visited", node x_B in the state "Absent", node x_C in the state "Absent", and so on till the last node in the network).

[2] Although we associate just conventional probabilities with the nodes without parents, for example with x_A or x_F, we can consider $p(x_A)$ or $p(x_F)$ as conditional probabilities conditioned on an empty set of nodes.

Probabilistic inference in a belief network can be most generally understood as an evaluation of joint probability sums [7]. Let X_i be the set of query nodes, the nodes whose probabilities we want to obtain. The probability distribution $p(X_i)$—the set of numbers, each being the probability of occurrence of a certain particular combination of states or values of the nodes in X_i—can be obtained from:

$$p(X_i) = \sum_{x_i \notin X_i} p(x_1, x_2, \ldots, x_n), \tag{2}$$

where the summation is over all possible states or values of the nodes that are not in X_i. That is, for each combination of values of the nodes in X_i, we get the probability of that combination by holding those nodes to those values and summing the joint probability distribution over all possible values of all other nodes. Using Bayes' rule, we can evaluate a conditional probability $p(X_i|X_j)$:

$$p(X_i|X_j) \equiv p(X_i \cup X_j)/p(X_j), \tag{3}$$

which gives the probability of observing a particular combination of values for the nodes in the query X_i provided that we know the values of the nodes in X_j. For example, X_i could be the diseases in which we are interested and X_j could be available findings.

The product structure (1) of the joint probability distribution allows to evaluate the sums very efficiently by taking common multipliers out of the summation sign and reducing the total number of operations to perform the summation (2). For example, to sum over node x_A in (2), we notice from Fig. 1 that there are only two factors in the product (1) that contain x_A explicitly: the conditional probability $p(x_B|x_A)$ and the probability $p(x_A)$ itself. Thus, we can take all the six other factors corresponding to the rest of conditional probabilities out of the summation sign as a common multiplier as we sum over the values of x_A.

The summation of the joint probability distribution is particularly easy in a linear chain of nodes or in a tree-structured network, since every factor in the product (1) involves at most two nodes and the structure of the network guarantees the absence of loops in the summation sequence. Thus, in the well-known Lauritzen-Spiegelhalter (LS) algorithm [6], as in most other algorithms for probabilistic inference, a general network is first converted to a *join tree* and the summation is represented as a propagation of messages in this tree. The join tree nodes are traditionally called *cliques* since they were obtained as completely interconnected subsets, or cliques, of a moralized (with interconnected parents) and triangulated network [6]. One of the possible join trees for the "Asia" network is shown in Fig. 2. A simple node from the original network can appear in several adjacent join tree nodes. The way the join tree is constructed, these copies are made so that the factors in the joint probability distribution (1) are each wholly contained within one of the cliques. The probability distribution of the nodes in a clique is traditionally called the clique *potentials*. The product of all clique potentials represents the same joint probability distribution as the joint probability distribution of the original network.

Let us look at an example of the process of the summation as it propagates through a network. Consider computing the probability $p(x_H)$ in Fig. 2. Clique $X_1 = \{x_A, x_B\}$ sends a message containing a partial sum over x_A to the clique X_2. Clique $X_4 = \{x_C, x_D\}$ sends a similar message, the potentials summed over x_D, to the same clique X_2. The clique $X_2 = \{x_B, x_C, x_E\}$ multiplies its potentials by the corresponding entries from the messages $X_1 \rightarrow X_2$ and $X_4 \rightarrow X_2$, sums the potentials over x_B, and sends a message to X_3. The clique X_3, in its turn, receives messages from X_2 and X_5 and sends a message to X_6. Finally, the clique $X_6 = \{x_C, x_G, x_H\}$ sums its potentials over x_C and x_G to yield the desired distribution $p(x_H)$.

As we can see, the computations in each clique require only a few operations per datum in the clique. Each potential is multiplied by a single entry in the message array from each but one of its neighbors, and is summed for a message to the remaining neighbor. The computational complexity of the problem arises from the size of the potentials arrays in the cliques and the message arrays that carry messages between the cliques. Since the state space for potentials is a direct product of the state spaces for each of the individual nodes in the clique, the number of potentials per clique grows exponentially with the number of nodes in the clique, and hence so does the number of operations for probabilistic inference. The size of the cliques depends on the density of the interconnections in the original network and can be very large—in modern large practical networks potentials may require as much as 500 MB of RAM for their representation.

We can think of the computations as associated with individual message entries in a message array. The computations associated with a message entry are summing some of source clique potentials to form an outgoing

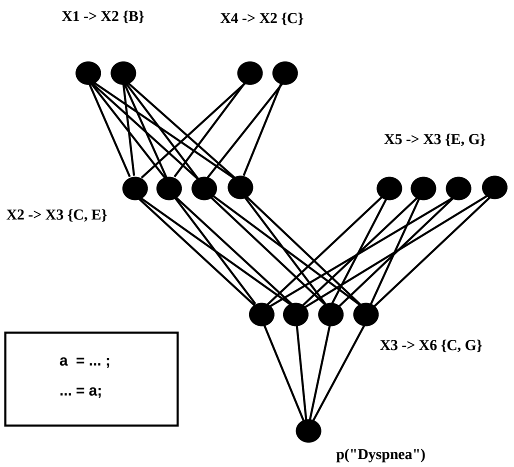

Figure 3: The data dependencies of different computations for the evaluation of $p(x_H)$. A circle represents the set of instructions to compute an entry in the message array, i.e. to sum the potentials of the source clique and to multiply the potentials of the destination clique. A group of circles represents a message between two cliques. For example clique X_1 sends a message $X_1 \rightarrow X_2$ consisting of two numbers to clique X_2. All dependencies are true-dependencies (see insert in the lower left corner) and are through the clique potentials. The dependencies between $X_1 \rightarrow X_2$ and $X_4 \rightarrow X_2$ messages as well as $X_2 \rightarrow X_3$ and $X_5 \rightarrow X_3$ messages are not shown for simplicity. Although the graph is very densely interconnected vertically, the computation of each message is completely parallel. Since the size of the message arrays is usually very large (thousands), the problem has enough parallelism algorithmically.

message array entry and multiplying some of the destination clique potentials by that message entry. The data dependencies among message array entry computations are shown in Fig. 3. All the dependencies are true-dependencies, carried through the potentials of the cliques. The computations not connected by a dependence edge in Fig. 3 can be done in parallel. We can see that computation of different entries in a message between two cliques is completely parallel, i.e. each entry in the message array can be computed independently. Since the number of messages is usually very large (thousands) in networks that are computationally expensive and have large cliques, the computationally expensive problems have a lot of parallelism. However, the processors would need to exchange data if two computations connected by a data dependence arc are to be performed on different processors. For example, even though we can avoid interprocessor communication on the path $X_4 \rightarrow X_2 \rightarrow X_3 \rightarrow X_6$ in Fig. 3 with only two processors by assigning the first half of each message array along the path to one processor and the second half to the second,[3] interprocessor communication can not be avoided for either a larger number of processors or for the other messages. In general, the inherent interprocessor communication can be high due to the high interconnection density of the dependence graph, but can be reduced by clever assignment of the message entry computations to processors.

The joint probability summation algorithm presented here represents the structure of computations in the modern algorithms for probabilistic inference. The two most popular algorithms are the Lauritzen-Spiegelhalter (LS) algorithm [6] and the Symbolic Probabilistic Inference (SPI) algorithm [7] based on the optimal factoring approach described above. While the LS algorithm builds the whole join tree in the memory and evaluates the probabilities of all nodes in the network each time a query is made, the SPI algorithm is query-oriented and computes only the conditional probabilities that are required for the query. The LS algorithm is convenient when we want to know the probabilities of all nodes in the network. The SPI algorithm is more suited for queries with a few nodes and is substantially more economical in terms of memory requirements and the computation time in this case. Despite these differences, the structure of the computation in the two algorithms is essentially identical

[3] The intrinsic reason for the absence of the dependencies between these two groups is that the node x_C is at the beginning of all message node sets on the path $X_4 \rightarrow X_2 \rightarrow X_3 \rightarrow X_6$. This separation into groups of independent computations will be the basis of our heuristics for sorting nodes in the node sets, which we describe below in Section 3.

and can be reduced to the summation of the join tree that we have considered above.

3 Uniprocessor Implementation

Our starting point for parallelism is a new optimized uniprocessor program for probabilistic inference that we wrote. It is based on a lazy algorithm that updates the join tree in depth-first search order [8]. It is more economical than the original LS algorithm in that it updates only a part of the tree for simple queries—given precomputation of the clique potentials—as opposed to the whole tree, and can perform incremental probabilistic inference.[4] Clique potentials are updated only if necessary. The time for inference is still governed by the size of the cliques and grows linearly with the total number of potentials in the cliques (exponentially in the number of nodes), but with smaller constants. Since different queries are likely to require different computation time, the experimental results in this paper are for the complete update of the join tree.

The uniprocessor program we constructed is substantially faster than the program for the Lauritzen-Spiegelhalter algorithm that we originally obtained and used in [9]. The speedup is largely due to the optimization of two bottlenecks in the original program. First, the original program spent most of its execution time evaluating the mapping between clique potentials and message entries, i.e. determining which potentials should be multiplied by which incoming message entries. We optimized this by evaluating the index for the next potential based on the previous index, and by optimizing the index computation in general by precomputing some of the information. This reduced the percentage of time spent on index computation to about 50%. Second, the original program was not sensitive to the memory architecture—particularly the use of caches and multiword cache lines—and spent up to 80% of its execution time on memory stalls. To exploit locality we have to rearrange potentials in the cliques so that the potentials that are most often summed together are close in the address space. Unfortunately, this requirement can not be satisfied since different outgoing messages are likely to be potentials summed over values of different nodes and we do not know apriori which queries are going to be made. Thus, we used a greedy strategy of sorting the simple nodes in the cliques—and thus the potentials to be summed and multiplied—according to the number of times they appear in different cliques in the join tree. This strategy achieves good data locality for an average query, substantially reducing the memory system time.

These optimizations resulted in large uniprocessor speedups. For example, the probabilistic inference time for the practically important CPCS subnetwork (the Computer-based Patient Case Study 179-node subnetwork with up to 8 parents per node) was reduced from 100 seconds to under a second. Thus, this particular network no longer needs parallelism, and we do not consider it in this paper. Larger networks exhibited smaller, but still substantial uniprocessor speedups through these optimizations.

4 Multiprocessor Platforms

We ran our experiments on two machines which support an implicit shared address space communication abstraction as well as coherent caching in hardware. The shared address space abstraction greatly simplifies the parallel programming task, particularly for applications with irregular data access patterns such as the this one. Our parallel programs are written in C using the `parmacs` macro package from Argonne National Laboratories for parallelism constructs.

4.1 The Stanford DASH Multiprocessor

The DASH machine [10] is an experimental multiprocessor built at Stanford University. The machine we used has 32 processors (33 MHz clock speed) organized in 8 clusters.[5] A cluster comprises 4 MIPS R3000 processors and 32MB local memory 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

[4] We have also implemented the optimal factoring or SPI algorithm in parallel, but since it is more query-dependent and demonstrates the same trends for parallelization as our incremental algorithm, we do not consider parallel optimal factoring in this paper

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

fraction of the physical memory on the machine. That is, memory is logically shared but physically distributed, and four processors share a local main memory unit. The caches in the system are kept coherent between clusters in hardware using a distributed directory-based protocol.

There are several types of memory (read) latencies on DASH. A miss in the first-level cache satisfied in the second-level cache has a stall time of 12 processor clock cycles. A miss in the second-level cache satisfied locally in the processor cluster—which we call a *local* miss—results in a stall time of 29 processor clock cycles. Finally, if the data are in some remote cluster, the processor is forced to initiate a request on the mesh network, which stalls the processor for 100 or more processor clock cycles depending on the location of the data and the state of the directory. We call the latter miss a *remote* miss. Since local misses are much less expensive than remote misses, it is important to reduce intercluster communication and to allocate (place) data in physical memories so that if capacity misses are important then they are kept local.

4.2 The SGI Challenge Multiprocessor

For our experiments, we used a commercially available shared-address-space SGI Challenge XL machine with 512 MB centralized memory. It has 16 MIPS R4400 processors (100 MHz clock speed) connected to a shared bus. The consistency between processors is maintained by a snoopy-cache-coherence bus protocol. The processors had a bigger cache line size (128 bytes as opposed to 16 bytes for DASH), a smaller first-level cache (16 KB as opposed to 64 KB for DASH), and a larger second-level cache (1 MB as opposed to 256 KB for DASH).

Since all the processors are connected to a single bus, the difference between memory latency due to a second-level cache miss satisfied in the main memory and one satisfied in one of the other processor caches is not large (100 versus 130 processor cycles without contention, respectively). Thus, there are no "remote" misses and no data placement problem, and communication is not much more expensive then capacity misses.

5 Available Parallelism

There are two types of parallelism available in belief networks: *topological* and *in-clique* [9]. Topological parallelism is due to the fact that computations in cliques that do not have an ancestor-descendent relationship in the join tree are independent. In-clique parallelism is due to the fact that even the computations for different incoming or outgoing message entries in the same clique (node of the join tree) are independent. The amount of topological parallelism is very limited in randomly generated networks, and is larger but still moderate in practical networks due to the way people tend to construct them. The vast majority of the computation tends to be concentrated in large cliques.

6 Parallelization Strategies and Multiprocessor Implementation

Two major strategies for managing parallelism are static and dynamic assignment of tasks to processes.[6] Static assignment means that once the input description is known it is predetermined which processes will perform which tasks, and this is not subject to change at runtime. It has low runtime task management overhead and allows better control over data locality, and is therefore preferable when the workload distribution among tasks is predictable enough that static assignment can lead to good load balance. However, if the workload distribution among tasks is not predictable, then dynamic assignment must be used. For unpredictable computations, dynamic assignment is usually implemented using a system of task queues from which processors take tasks at runtime when they become idle.

Of the two types of parallelism available in this application, the workload distribution for in-clique parallelism is predictable; i.e. given that a clique will be involved for a given query, we can determine ahead of time how the basic computations within the clique should be subdivided among processes to obtain load balance as long as we know the number of clique potentials and message entries. However, the distribution of work across cliques (for

[6] We assume one process per processor in this paper, and use the terms interchangeably. A task in this context is a part of the computation that can be performed in parallel with the other tasks.

topological parallelism) cannot be scheduled statically without a lot of extra computation, all the more because we do not know what queries will be made.

Our first parallel implementation uses static assignment, but exploits only in-clique parallelism. It therefore does not have to worry about which cliques will be used. We assign a contiguous set of entries in a message from one clique to another to a single processor. That processor is then responsible for computing those message entries by summing over source clique potentials, and for multiplying destination clique potentials by the computed message entries. If the number of messages is smaller than the number of processors, the computations for each message entry are subdivided among a subset of processors (each processor computes part of the sum for a message array entry, and then all processors communicate partial sums to each other). The computations on different cliques (more correctly, on different edges of the clique tree since computation is associated with messages in our formulation) are separated by a barrier. For example, the message $X_2 \rightarrow X_3$ in Fig. 3 consists of four entries and thus the computation of the message can be subdivided between four processors. When the computation of the message $X_2 \rightarrow X_3$ is completed, all processors meet at a barrier and continue on computation of the message $X_5 \rightarrow X_3$. For the large networks, the number of messages between cliques that matter is several thousands, so the computation has enough parallelism.

The dynamic task scheduling program exploits both topological and in-clique parallelism, and is oriented towards first exploiting the more coarse-grained topological parallelism to reduce interprocessor communication by dividing processors between different branches of the join tree. It represents an optimized version of the dynamic scheduler we used in our parallel Lauritzen-Spiegelhalter algorithm implementation [9]. To allow more concurrency at the topological level, we subdivide the update of one clique potentials by another clique potentials into two stages. First, we compute all message entries from the source clique and store them in a message array (per source-destination clique pair), and then we update potentials of the destination clique by the stored message array entries. Thus, the computations of messages and potentials can be done independently and several processors can work simultaneously on computing messages for the same destination clique.

Every clique-level message or potentials computation is further broken up into tasks (to exploit in-clique parallelism as well), each of which is conceptually either the computation of a single message entry or the computation of a single potential. First, the leaves of the tree are placed into a global task queue. A processor picks up a leaf from the global queue and places the (likely many) tasks for that leaf into its local, but shared, task queue. If its task queue becomes empty and there are no more cliques in the global task queue, a processor tries to steal tasks from other neighboring queues (the goal is to compute the whole clique on a single processor, exploiting topological parallelism). In fact, to reduce task management overhead, the individual tasks are not really enqueued and dequeued from a task queue, but rather the task management and stealing is done through the manipulation of task indices assigned to processors. To reduce stealing overhead and exploit locality, a processor steals several tasks at a time. In order to further preserve locality when stealing, a processor steals tasks from processors that work on the same clique or a nearby clique and steals them from the end of the potentials or message array, not from the beginning of the array from where the tasks are taken for processing by the processor which owns the queue.

7 Results

We first describe the input belief networks that we use, and then the parallel performance results and analysis, including a measurement of the spatial and temporal locality properties of the programs.

7.1 Input Networks

We used six networks represented in Table 1 for our performance measurements. Since there is no such thing as a typical belief network, the first four networks—which we call the AA, A, B, and C networks—were generated to have a random interconnection structure.[7] The computation time for these networks varies from 5 to 260 seconds and the global memory requirements from 8 MB to 164 MB. As we will see later, the networks exhibit some common as well as some distinguishing data locality properties.

[7] The networks were generated by randomly removing edges from a completely interconnected network until it has a required average number of edges per node. These are the same networks that we used in [9].

Table 1: Parameters of the networks we used to test the multiprocessor implementation. 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, $\max \| X_i \|$ – maximum size of a potentials array in double precision numbers, $\max \| X_i \cap X_j \|$ – maximum size of a message array in double precision numbers, Memory – global memory requirement in MB, Time – uniprocessor (DASH) propagation time in seconds.

network	n	e	v	c	l	$\max \| X_i \|$	$\max \| X_i \cap X_j \|$	Memory	Time
AA	54	2.4	2	30	12	16 M	1 M	164	260
A	54	2.2	2	32	16	512 K	256 K	23	33
B	18	2.2	4	8	4	1 M	64 K	10	5
C	54	2.0	2	30	14	512 K	64 K	8	8
TL8	16	4	4	9	7	256 K	64 K	16	10
MC8	16	4.5	4	9	1	256 K	64 K	16	10

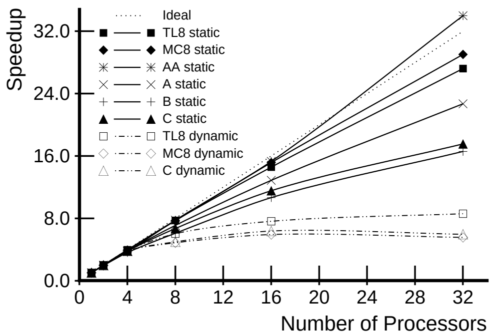

Figure 4: Speedups on the DASH multiprocessor for the static and dynamic assignment programs. The AA network displays superlinear speedup.

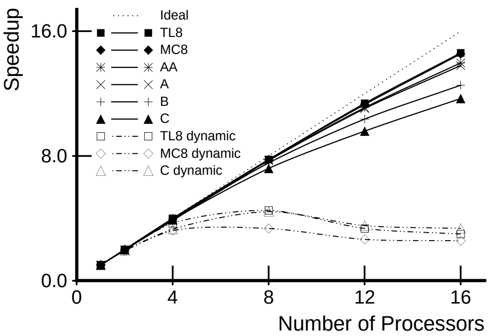

Figure 5: Speedups on the SGI Challenge multiprocessor for the static and dynamic assignment programs.

The two other networks have been constructed to resemble some networks of practical interest. The TL8 network is a two-level network with 8 four-valued nodes in the first level and 8 four-valued nodes in the second level. Each node in the second level has all 8 nodes in the first level as its parents. Two-level networks are often used in diagnosis problems, such as medical diagnosis in which one level consists of diseases and the other of findings. The MC8 network is a Markov chain network consisting of 16 four-valued nodes. Each node in the network conditionally depends on up to 8 of its predecessors. The join trees for the TL8 and MC8 networks consist of 8 cliques with 9 nodes in each, requiring 262,144 potentials (2MB of RAM per clique if represented by double precision numbers). While the join tree for the MC8 network can be represented as a linear chain of 8 cliques, the join tree for the TL8 network can be represented as a tree with 7 branches, thus exhibiting additional topological parallelism that can be exploited by our dynamic task scheduler. The size of all but the AA network has been chosen to fit a single DASH cluster memory to avoid allocating memory in different clusters when the program is executed in a single cluster (on less than 4 processors).

7.2 Multiprocessor Speedups

Fig. 4 and Fig. 5 show speedups on the DASH and SGI Challenge multiprocessors. The speedups depend on the size and topology of the networks, and are generally better for larger networks with larger cliques (the superlinear speedup for the AA network will be explained at the end of this section). However, they are uniformly better for the static task assignment than for dynamic assignment. For the static scheme, they are better for the

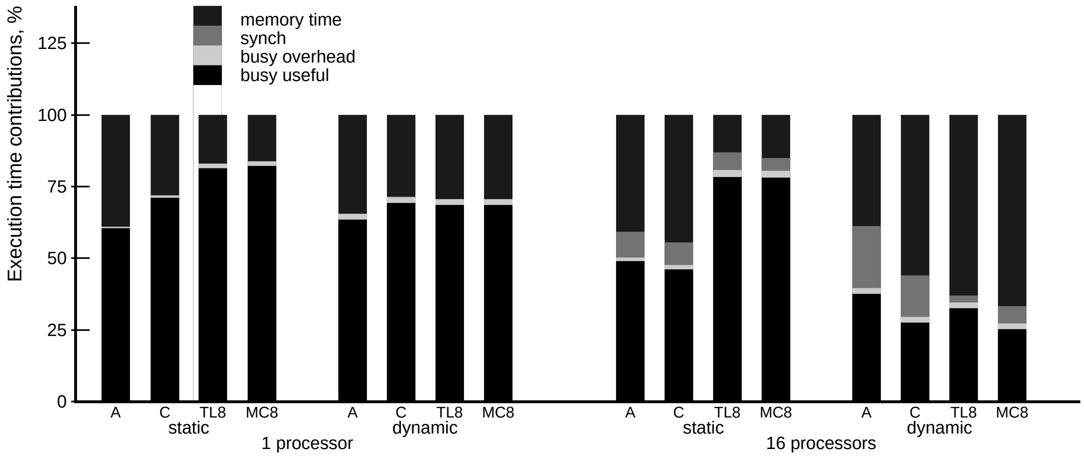

Figure 6: The computation time decomposition on DASH for the A, C, TL8, and MC8 networks for the static and dynamic task scheduling. Busy useful is the time which a computer would spend on the computations on a single processor. Busy overhead is the extra time processors have to spend on computations—initialization and finalization within each task—when executed on a multiprocessor. For static task scheduling, the synchronization time is the time spent waiting at synchronization points. For the dynamic task scheduling, synchronization time also includes the time spent on the computations needed to steal tasks, so do not reflect only load balance. The memory system time is the time spent on memory stalls.

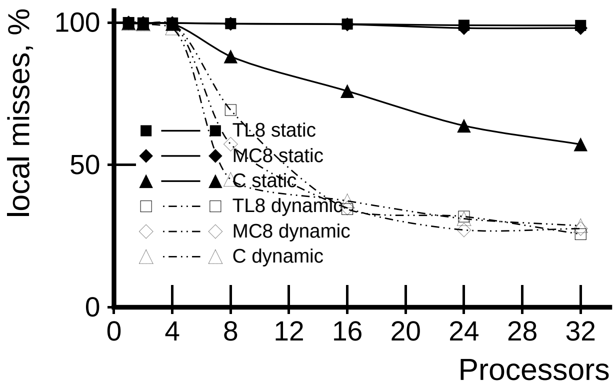

Figure 7: The relative number of local misses in the second-level cache for the TL8, MC8, and C networks for the static and dynamic assignment.

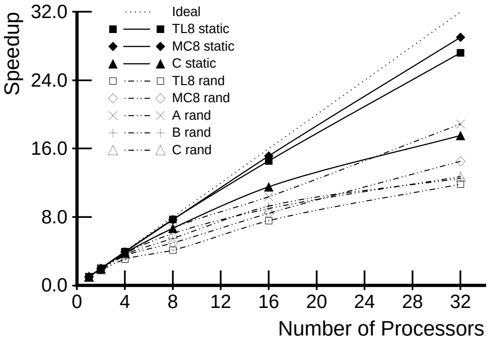

Figure 8: The speedups (DASH) compared with a random node ordering in the cliques.

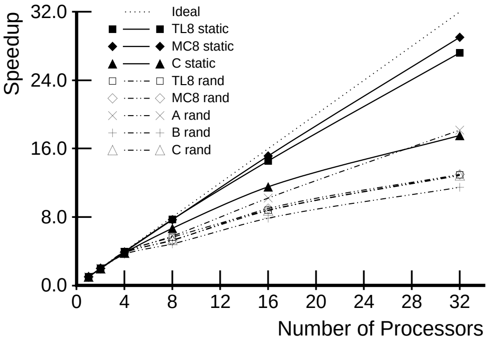

Figure 9: The speedups (DASH) compared with a random data placement strategy.

structured networks than for the random ones. To understand the reasons, we measured how the processors spend their time using the `prof` and `pixie` profiling tools, and explicit timing measurements around synchronization events. `pixie` is a program that counts basic blocks and measures what the execution time would have been had there been no cache misses, while `prof` uses PC-sampling to profile the actual execution time spent in different parts of the code. The resulting decomposition of execution time (shown for some networks in Fig. 6) shows that the worse performance of the dynamic scheme compared to the static scheme, as well of the static scheme when using the random C network, is explained by the increase in the time spent stalled on the memory system.

To explain the increase in memory system time, we measured the relative number of locally versus remotely satisfied second-level cache misses on DASH using a hardware performance monitor. This hardware monitors bus transactions in a cluster without perturbing the program execution. It sees all second-level cache misses coming from each processor in the cluster, and can tell whether misses are satisfied within the cluster or have to be satisfied remotely (by another cluster). The results for some networks are shown in Fig. 7. The ratio of local to remote misses stays close to 100% for the TL8 and MC8 networks, in which we were able to take advantage of the dependence structure to partition the network with a minimum of cross-processor dependencies and to place data appropriately in cluster memories. However, it quickly becomes much lower for the program with dynamic assignment for all networks, both due to the inherent communication caused by processors picking up dependent tasks unpredictably along the same path in the tree, and due to the difficulty of placing data appropriately given this unpredictability of which processors will perform what tasks. Note that since there is one memory per cluster composed of four processors, data placement and remote misses only become an issue after four processors (if everything fits in one cluster's memory when less than four processors are used). For the static assignment program, the ratio of local to remote misses also decreases with an increase in the number of clusters for random networks, albeit less than in the dynamic case, since the random connectivity of these networks prevents us from eliminating communication or placing data very well at page granularity. Most real networks have a lot of structure, so we should be able to partition them much better than the random networks.

We can also reduce the number of remote misses by proper data management. In Sections 3 we described our heuristic to sort the nodes in the clique node sets to achieve a better data locality and to reduce execution time for a uniprocessor. The same strategy helps to minimize interprocessor communication for a multiprocessor since it makes the dependencies between message array entry computations (see Fig. 3) almost vertical. To demonstrate the effect of the ordering strategy, Fig. 8 compares the speedups with those obtained using a random ordering of nodes.

The same ordering strategy also allows to efficiently allocate memory pages in different clusters. For example, if the dependencies are almost vertical, potentials at the beginning of the clique arrays are likely to be summed into messages at the beginning of the message array, and the messages at the beginning of the message array are likely to update potentials at the beginning of the next clique array. The allocation of the adjoint contiguous parts of the arrays on the corresponding clusters decreases the memory system time and improves performance. Fig. 9 shows the effect of this allocation strategy on the performance of the parallel program by comparing with

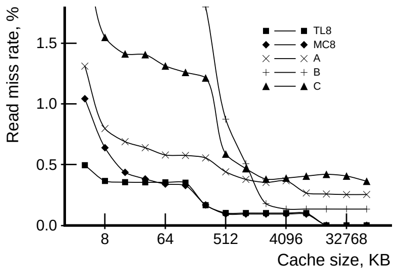

Figure 10: Read miss rate as a function of cache size. Simulation for 16 processors and 64 byte cache line size with 4-way set associative caches.

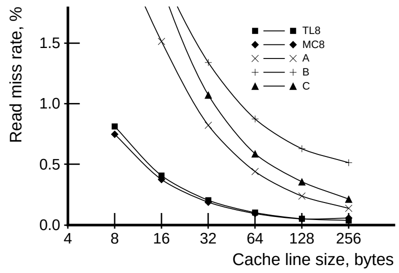

Figure 11: Read miss rate as a function of cache line size. Simulation for 16 processors and 512 KB, 4-way set associative cache size per processor.

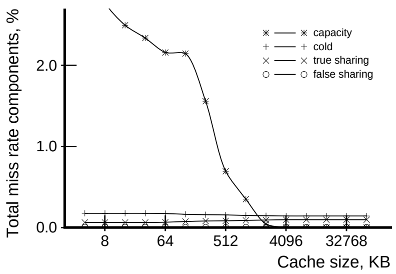

Figure 12: Miss rate decomposition on capacity, cold, true sharing and false sharing misses for the B network as a function of cache size. Simulation for 16 processors and 64 byte cache line size with 4-way set associative caches.

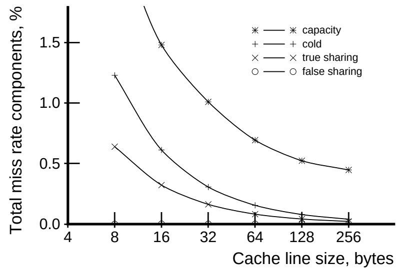

Figure 13: Miss rate decomposition on capacity, cold, true sharing and false sharing misses for the B network as a function of cache line size. Simulation for 16 processors and 512 KB, 4-way set associative cache size per processor.

a random page allocation strategy. Although page allocation was also implemented for the dynamic version, the effect is much less pronounced due to the stochastic nature of the task stealing process.

The efficient memory allocation allows to obtain a superlinear speedup on DASH for the large AA network (see Fig. 4). While the uniprocessor execution has to use memory in multiple clusters (the generated data structures do not fit into a single cluster memory and are distributed in multiple clusters for this network), beyond a certain number of clusters multiprocessor execution with our partitioning is likely to use mostly local memory, effectively reducing the memory system time. This reduction in memory system time from 202 seconds to 160 seconds compensates the extra 25 seconds spent on synchronization for the 32 processor program execution, resulting in superlinear speedup.

These results show that we can achieve good speedups using good task partitioning and data placement on a distributed memory machine. Since the memory system performance is so important for our parallel program, let us now examine the data locality behavior in some detail, to understand how it might scale with some important memory system parameters, problem size and number of processors.

7.3 Cache performance and data locality

To understand how the temporal and spatial locality in the program scale with the problem size and number of processors, which provides insights into how performance might scale with different systems, we performed software simulations of the multiprocessor execution of our program with different cache parameters. The simulations were done using the TangoLite execution-driven reference generator coupled to a memory system simulator. The simulator models a cache-coherent multiprocessor with a physically distributed shared address space and a directory based cache-coherence protocol. It is very flexible and divides cache miss rates according to a classification described in [11].

To analyze temporal locality, we measure the working sets of the program by plotting the miss rate versus the size of the per-processor cache used in the simulations, assuming a sixteen processor execution and 4-way set-associative caches with a 64-byte cache line size. The results are shown in Fig. 10. For the TL8 and MC8 networks, the working set sizes are approximately the size of one clique potentials array divided by the number of processors, i.e. 2MB/16 = 128 KB. For randomly generated networks, the dependence is not so distinct since we have a wider distribution over the sizes of the cliques, except for the B network. All the work for the B network turns out to be concentrated in one clique of size 8MB (see Table 1), which produces a sharp rise in the total execution time around a 512 KB cache. This is confirmed by the decomposition of the miss rate into capacity, cold, true sharing and false sharing for this particular network in Fig. 12. For large networks the capacity or cold misses constitute the major part of all misses since the clique potentials arrays are typically several MB and it is clique potentials that are reused. Thus, temporal locality in the application is quite limited.

To analyze spatial locality, we measure the dependence of the miss rate on cache line size (the sizes of the processor caches were kept constant at a relatively large size of 512 KB). The results are shown in Fig. 11 and Fig. 13. The dependence characterizes the spatial locality of the program and the amount of data sharing between processors. All components of the miss rate decrease for random networks as the cache line size increases. False sharing of data among processors is insignificant compared to other sources of misses for the B or other randomly generated networks. For the TL8 and MC8 networks, its effect becomes important only beyond 128 byte cache line. In general, the application has good spatial locality for today's cache line sizes (up to 128 bytes).

8 Discussion and Conclusions

We have presented two parallel implementations of exact probabilistic inference which exploit concurrency at different levels, and have measured and analyzed their performance. We found that a simple static partitioning technique that exploits only one type of parallelism (parallelism between different message entry computations) produces better results than a dynamic technique that tries to exploit all available parallelism (focusing first on the more coarse-grained topological parallelism). The better performance of the static scheme is explained by the better data locality property, which is more difficult to retain in the dynamic scheme due to both the unpredictability of topological parallelism and the stochastic nature of the task stealing algorithm, both of which make good data placement difficult. Compared with load balance, which is relatively easy to obtain with both schemes, data locality is a more critical issue for this application. In fact, locality is a critical issue even for a uniprocessor implementation of probabilistic inference. This emphasis on data locality makes probabilistic inference a good benchmark for the memory and communication architectures of multiprocessors.

While speedups degrade as the number of processors grows, they increase as the inference networks become larger. Speedups for the networks of different structure that we examined—randomly generated networks as well as networks representing practical problems—are close to ideal on moderate-scale shared-address-space machines with distributed as well as centralized main memory. For machines with distributed memory, we provided an efficient data placement algorithm. We showed that with this placement strategy the parallel program scales very well. Speedups become superlinear when the networks are too large to fit in a single processor's memory. Since the static scheme is also able to keep inherent communication low in most cases, we expect the application to scale well for large networks on large multiprocessors.

To further understand scalability with problem size, number of processors and cache organization, we analyzed the locality properties of the parallel program in detail using multiprocessor simulation. We found that working sets tend to grow exponentially with the size of the original network (linearly with data set size) and the reuse of data is limited, leading to poor temporal locality. On the other hand, spatial locality is quite good and we expect it to grow with the size of the networks. Thus, the application can efficiently utilize the long cache lines on modern microprocessors, particularly for inference in large networks.

Since computationally expensive networks usually have large cliques and hence a large amount of the in-clique parallelism, the networks that are most important to parallelize are also easily parallelizable with good speedups. Networks that are not dominated by large cliques are possible, but they are typically not expensive to compute even on a uniprocessor and do not need parallelism in practice. Thus, exploiting only the in-clique parallelism in our static approach is satisfactory. If there are cases where it isn't, it is always possible to extend the static approach with dynamic task stealing for load balancing.

Acknowledgments

We thank Adam Galper for providing us with the original sequential Lauritzen-Spiegelhalter implementation from which we started, Daphne Koller, Wray Buntine, and Malcolm Pradhan for discussions, John Hennessy for his support and guidance, and ARPA for financial support under contract no. N00039-91-C-0138.

References

[1] R. T. Cox. Probability, frequency, and reasonable expectation. *Am. Jour. Phys.*, 14:1 – 13, 1946.

[2] Edward T. Jaynes. Probability theory: The logic of science. Unpublished, but publicly available through ftp://bayes.wustl.edu/Jaynes.book, 1995.

[3] Alexander V. Kozlov and Jaswinder Pal Singh. Approximate probabilistic inference in Bayesian belief networks based on sensitivities. *submitted to the Journal of Artificial Intelligence Research*, 1995.

[4] Edward Eric Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization.* PhD thesis, Stanford University, 1993.

[5] Jaswinder Pal Singh. *Parallel hierarchical N-body methods and their implications for multiprocessors.* PhD thesis, Stanford University, 1993.

[6] 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.

[7] Zhaoyu Li and Bruce D'Ambrosio. Efficient inference in Bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11(1):55 – 81, 1994.

[8] Alexander V. Kozlov and Jaswinder Pal Singh. Sensitivities: an alternative to conditional probabilities for Bayesian belief networks. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 376 – 385. Morgan Kaufmann, 1995.

[9] Alexander V. Kozlov and Jaswinder Pal Singh. A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Proceedings of the Supercomputing'94 conference*, pages 320 – 329, held in November 1994, Washinton, DC, 1994. IEEE Computer Society Press.

[10] 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.

[11] Steve Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24 – 36, 1995.