Node Level Primitives for Parallel Exact Inference[*]

Yinglong Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90089, U.S.A.
yinglonx@usc.edu

Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, U.S.A.
prasanna@usc.edu

Abstract

We present node level primitives for parallel exact inference on an arbitrary Bayesian network. We explore the probability representation on each node of Bayesian networks and each clique of junction trees. We study the operations with respect to these probability representations and categorize the operations into four node level primitives: table extension, table multiplication, table division, and table marginalization. Exact inference on Bayesian networks can be implemented based on these node level primitives. We develop parallel algorithms for the above and achieve parallel computational complexity of $O(w^2 r^{(w+1)} N/p)$, $O(Nr^w)$ space complexity and scalability up to $O(r^w)$, where N is the number of cliques in the junction tree, r is the number of states of a random variable, w is the maximal size of the cliques, and p is the number of processors. Experimental results illustrate the scalability of our parallel algorithms for each of these primitives.

1. Introduction

A full joint probability distribution for any real-world systems can be used for inference. However, such a distribution grows intractably large as the number of variables used to model the system grows. Bayesian networks[9] are used to compactly represent joint probability distributions by exploiting conditional independence relationships. They have found applications in a number of domains, including medical diagnosis, credit assessment, data mining, image analysis and genetics [4][10][11][13].

Inference on a Bayesian network is the computation of the conditional probability of the *query variables*, given a set of *evidence variables* as knowledge. Such knowledge is also known as *belief*. Inference on Bayesian networks can be *exact* or *approximate*. Exact inference is NP hard [9]. The most popular exact inference algorithm converts a given Bayesian network into a junction tree, and then performs exact inference in the junction tree [6].

Early work on parallel algorithms for exact inference appears in [5][9] and [12], which formed the basis of scalable parallel implementations discussed in [7] and [8]. However, unlike [7], which focuses on the conversion of Bayesian network to junction tree and [8], which presents the parallelization of exact inference using pointer jumping, we do not focus on the network structure at all. Note that structure level parallelism can not offer large speedups when the size of the cliques of the junction tree or the number of states of the variables increase, making node-level operations the dominating part of the problem. We focus on parallelizing the *table operations only* and start off with a junction tree converted from an arbitrary Bayesian network.

In this paper, we study parallelism in exact inference from the *node level* perspective. Node level means the probability representation and operations on the nodes of Bayesian networks and the cliques of junction trees. We propose four node level primitives and achieve a scalability of $1 \le p \le r^w$, compared to $1 \le p \le n$ of most structure level parallel methods. We implement parallel node level primitives using MPI on state-of-the-art platforms. We also propose an exact inference algorithm based on those parallel node level primitives, which achieves computational complexity of $O(w^2 r^{(w+1)} N/p)$ and space complexity $O(Nr^w)$, $1 \le p \le r^w$, where N is the number of cliques in the junction tree, r is the number of states of a random variable, w is the maximal size of the cliques, and p is the number of processors.

The paper is organized as follows: Section 2 gives a brief background on Bayesian networks and junction trees. Section 3 addresses the node level primitives for exact inference. Section 4 implements the node level primitives and the parallel exact inference algorithm based on these primitives. Experiments are shown in Section 5 and Section 6

[*]This research was partially supported by the National Science Foundation under grant number CNS-0613376 and utilized the DataStar at the San Diego Supercomputer Center. NSF equipment grant CRI-0454407 is gratefully acknowledged.

concludes the paper.

2. Background

Consider a set of n random variables, $\mathcal{W} = \{A_1, A_2, \cdots, A_n\}$. The probability that random variable A_j takes the value a is $P(A_j = a)$ where $a \in \{0, 1, \cdots, r-1\}$ and r is the number of states of variable A_j. A *joint distribution*, $P(\mathcal{W}) = P(A_1, A_2, \cdots, A_n)$, assigns a probability to every possible combination of states of these random variables.

A *Bayesian network* exploits conditional independence to represent a joint distribution more compactly: The variables in the network with directed edges to a certain variable A_j are called parents of A_j, denoted by $pa(A_j)$. Given the values of $pa(A_j)$, A_j is conditionally independent of all other preceding variables. The joint probability distribution thus can be rewritten using less variables as:

$$P(\mathcal{W}) = P(A_1, \cdots, A_n) = \prod_{j=1}^{n} P(A_j | pa(A_j)) \qquad (1)$$

For each variable A_j, we use a *conditional probability table* to describe the conditional probabilities $P(A_j|pa(A_j))$. The details of the conditional probability table will be addressed in Section 3.1.

Traditional exact inference using Bayes's rule fails for networks with undirected cycle [9]. Therefore, most inference methods convert a network to a cycle-free hypergraph called a *junction tree*. Each node of the junction tree is called a *clique*. A *potential table* is used for each clique to express the joint probability of all variables represented by a clique. The potential table of clique C_i is denoted as $\psi(C_i)$. The details of potential tables are given in Section 3.2.

The *evidence* in a Bayesian network is a group of observed variables e.g. $E = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$ where $e_k \in \{1, 2, \ldots, n\}$. Given these evidences, we can inquire the distribution of other variables. The variables to be inquired are called *query variables*. Exact inference is the propagation of the evidence throughout the junction tree followed by computation of the updated probability of the query variables. Mathematically, evidence propagation can be represented as:

$$\psi^*(R) = \sum_{Y \backslash R} \psi^*(Y), \qquad \psi^*(X) = \frac{\psi(X)}{\psi(R)} \psi^*(R) \qquad (2)$$

where X is a clique and R is a *separator*, which is the intersection of X and its neighbor clique Y. $Y \backslash R$ is defined as $\{r | r \in Y, r \notin R\}$; $\psi^*(\cdot)$ is the updated potential function.

P(A|B, C, D)	000	001	$\cdots$	111
0	0.4321	0.8275	$\cdots$	0.5273
1	0.5679	0.1725	$\cdots$	0.4727

Table 1. An example segment of conditional probability distribution in table representation.

3. Parallelism in Exact Inference

3.1. Probability Representation on Bayesian Network Nodes

Each node of a Bayesian network denotes a random variable. The probability distribution of a random variable can be expressed as a list of non-negative floating point numbers that sum to 1, where each number indicates the probability in a certain state. Assuming a node A has k parents, r_0 states and its i-th parent has r_i states, the conditional probability distribution of A, $P(A|pa(A))$, can be expressed as a 2-dimensional table of size $\prod_{i=0}^{k} r_i$.

Table 1 is an example of a conditional probability table where each random variable is binary. In Table 1, each column gives the probability distribution of the variable on a given condition set; each row indicates the probability of the variable taking a given state under various condition sets.

3.2. Probability Representation on Junction Tree Cliques

Each node of a junction tree denotes a clique, which is a set of random variables. For each clique C, there is a *potential function* $\psi(C)$ which is proportional to the joint distribution of the random variables in that clique [6]. The discrete form of the potential function is called a *potential table*. A potential table is a list of non-negative real numbers where each number corresponds to a probability of the joint distribution. Assuming a clique contains w random variables and the i-th variable has r_i states, the length of the potential table is $\prod_{i=1}^{w} r_i$.

The potential table of a clique is constructed from some conditional probability tables of the Bayesian network in two steps: First, a potential table is initialized: $\psi(C) = \prod_i P(A_i|pa(A_i))$, where $A_i \cup pa(A_i) \subseteq C$. Second, the initial potential table is updated from the root to the leaves and from the leaves to the root [6]. The second step is quite similar to belief propagation which will be discussed in the next section.

A potential table can be expressed as a 2-dimensional matrix, as shown in Table 2. However, unlike conditional probability tables, there is no conditional set in a potential table. We use row-major ordering to index the table entries,

ψ(A, B, C, D, E, F)	000	001	$\cdots$	111
000	0.0931	0.3917	$\cdots$	0.6821
001	0.2957	0.1725	$\cdots$	0.0942
$\cdots$			$\cdots$	
111	0.3402	0.2013	$\cdots$	0.0994

Table 2. An example segment of clique potential function in table representation.

e.g. in Table 2, $T(9) = \psi(A = 0, B = 0, C = 1, D = 0, E = 0, F = 1) = 0.1725$.

3.3. Belief Propagation among Neighbor Nodes

3.3.1 Deriving the Potential Table of Separators

A *separator* is defined as the set of shared variables of the two cliques.The table representation of a separator can be obtained from the potential tables of adjacent cliques in the junction tree. For example, assuming A and B are two adjacent cliques in a junction tree, the separator between the two cliques is given by $S = A \cap B$. Given the potential table of a clique, obtaining the potential of its subset is termed *marginalization*. Marginalization is illustrated in Figure 1, where $\psi(\{B, C\})$ is a separator potential table, $\psi(\{A, B, C, D, E\})$ and $\psi(\{B, C, F, G\})$ are the potential tables of two adjacent cliques. For example, $\psi(\{B = 0, C = 0\})$ (the gray entry in $\psi(\{B, C\})$) can be obtained by either summing up the values of all gray entries in $\psi(\{A, B, C, D, E\})$ or summing up the values of all gray cells in $\psi(\{B, C, F, G\})$. Note that the states of B and C in all these gray entries are 0.

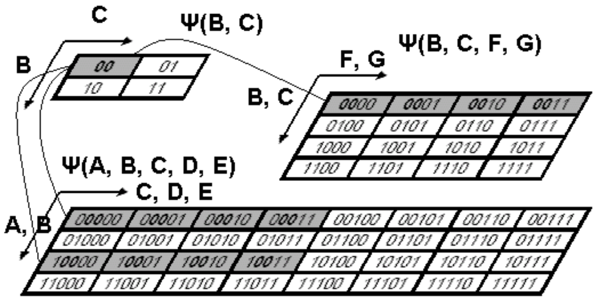


Figure 1. Obtaining separator potential table from clique potential tables. In each entry, the states of the binary random variables in a clique (or separator) are given.

3.3.2 Belief Propagation Between Adjacent Cliques

Assume two cliques A and B are adjacent and A has been updated. We want to use the updated potential table of A to renew the potential table of B. Denote S as the separator between A and B. First, we obtain the potential table of S denoted $\psi(S^*)$ by marginalizing the potential table of A. Then, we obtain the potential table of S denoted $\psi(S)$ by marginalizing the potential table of B, which is . Finally, the potential table of B can be updated by dividing the product of the entries in $\psi(B)$ and $\psi(S^*)$ by the entry in $\psi(S)$, that is, $\psi(B)\psi(S^*)/\psi(S)$. For each entry of $\psi(B)$, the above process can be implemented in parallel.

3.4. Node Level Primitives

In the previous section, we analyzed the node level probability representation and belief propagation. Here, we introduce four node level primitives on these tables: table marginalization, table multiplication, table division, and table extension.

Table marginalization is used to obtain the potential table of a separator from the potential table of a clique. The input to table marginalization is a potential table $\psi(A)$, and the separator between A and one of A's neighbors. The output is the potential table of the separator. Table marginalization can be implemented in parallel, where each processor handles a block of the clique potential table and obtains the separator potential table by combining the results from all the processors.

Table multiplication and **table division** are used in belief propagation among neighbor cliques. They convert multiplication and division between two tables to multiplication and division between corresponding entries. The independence of operations on different entries leads to a parallel implementation.

Table extension is used to simplify multiplication and division of tables. It equalizes the size of two tables involved in multiplication or division. Multiplying entries in the same locations of the two tables (shown in Figure 2), we obtain the result of table multiplication. Similarly, dividing two entries in the same locations of the two tables, if the divisor is not zero, we obtain the result of the table division.

Table extension can benefit the parallelization of table operations. Once the tables are extended, they can be divided into smaller chunks. For example, there is a random vertical surface in Figure 2 which cuts both $Extended\ \psi(\{B, C\})$ and $\psi(\{A, B, C, D, E\})$ into two chunks. Multiplication between $\psi(\{B, C\})$ and $\psi(\{A, B, C, D, E\})$ can be implemented by multiplying the two left chunks and multiplying the two right chunks in parallel.

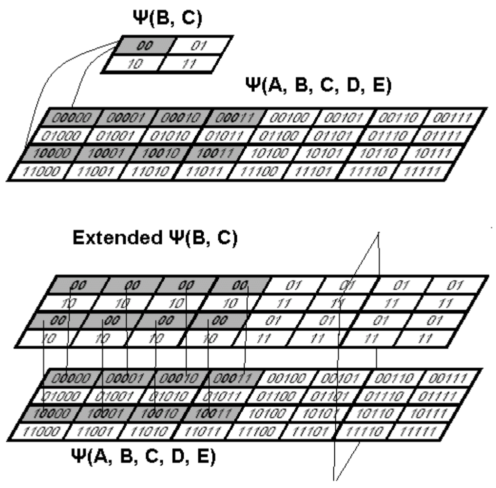


Figure 2. Illustration of table extension. After extension, each entry of one table corresponds to the entry in the same location of the other table.

4. Parallel Algorithms for Node Level Primitives

4.1. Table Extension

Consider the multiplication of tables T_1 and T_2. Let the random variables involved in T_1 be arranged in lexicographic order to form a vector V_1 (likewise with T_2 and V_2). Such a vector is called *variable vector* of a given table. For example, the variable vector of Table 2 is (a, b, c, d, e, f). We define a *mapping vector* $M_{V_2}^{V_1}$ as the corresponding relationship from V_1 to V_2. Specifically, if the i^{th} element of V_1 is the j^{th} element of V_2, then $M_{V_2}^{V_1}(i) = j$. So, vector $M_{V_2}^{V_1}$ has the same length as V_1. Note that each element of a variable vector V_i is a random variable. Suppose the k^{th} random variable in V_i has $(R_k + 1)$ possible states: 0, 1, ...R_k. In an observation, assuming the k^{th} random variable in V_i is in the $x_k^{(i)}$-th state where $x_k^{(i)} = 0, 1, \cdots, R_k$, we define the *value string* as $X^{(i)} = (x_1^{(i)}, ..., x_{|V_i|}^{(i)})$. Denote T_{Big} (T_{Small}) as one of T_1 or T_2 that has the larger (smaller) number of entries. The variable vectors are named as V_{Big} and V_{Small}, respectively. Using these definitions, we formulate the algorithm for table extension shown in Algorithm 1.

In this algorithm, we convert scalar i, the index of T_{Big}, into a value string $X^{(i)}$, so that we can find its corresponding element in T_{Small}. To convert value string $Y^{(i)}$ into a scalar j is given by the following equation:

$$j = \sum_k Y^{(i)}(k)(R_k + 1)^k \qquad (3)$$

Algorithm 1 Table-Extension: TExt($T_{Small}, V_{Small}, V_{Big}$)

```
Input: $T_{Small}$, variable vectors $V_{Small}$ and $V_{Big}$
Output: $T_{Big}$
  if $V_{Small} == V_{Big}$ then
    $T_{Big} = T_{Small}$
    return
  end if
  if $M_{V_{Big}}^{V_{Small}}$ has not been computed then
    Find mapping vector $M_{V_{Big}}^{V_{Small}}$ from $V_{Small}$ and $V_{Big}$
  end if
  Initialize $T_{Big}$
  for $i$ = 1 to $|T_{Big}|$ in parallel do
    Convert $i$ into value string $X^{(i)}$
    Create value string $Y^{(i)}$ from $X^{(i)}$ using $M_{V_{Big}}^{V_{Small}}$
    Convert $Y^{(i)}$ into index $j$
    $T_{Big}(i) = T_{Small}(j)$
  end for
```

We analyze the complexity of this algorithm using the well known concurrent read exclusive write parallel random access machine (CREW PRAM) model [9]. This model assumes a shared, global memory that processors can read from, but not write to, simultaneously.

In Algorithm 1, finding mapping vector $M_{V_{Big}}^{V_{Small}}$ costs $O(|V_{Small}|\log(|V_{Big}|))$, where $|V_i|$ gives the number of elements in V_i. With an allocated memory block, initializing a table costs constant time. Converting a scalar to a string takes $O(|V_{Big}|\sum_i R_i)$. Creating $Y(i)$ and filling T_{Big} take $O(|V_{Big}|)$ time. Assuming there are p available processors, the computational complexity of this algorithm is $O(|V_{Small}|\log(|V_{Big}|) + (|T_{Big}| \cdot |V_{Big}|\sum_i R_i)/p)$, where $|T_i|$ is the number of entries in table T_i. Assuming there are at most w random variables involved in a potential table and the maximum number of states of those random variables is r, $|V_i|$ is bounded by w and $|T_i|$ is bounded by r^w. As $O(w \log w)$ can be ignored compared to $O(r^w)$ when r and w are large, the computational complexity can be simplified to $O(r^w \cdot |V_{Big}|\sum_i R_i/p)$ where $1 \le p \le |T_{Big}|$. As $|V_{Big}| \le w$ and $\sum_i R_i \le w \cdot r$, the complexity is bounded by $O(w^2 r^{(w+1)}/p)$, $1 \le p \le |T_{Big}|$.

In Algorithm 1, each processor needs T_{Small}, V_{Small} and V_{Big}. We only consider $V_{Small} \le V_{Big} - 1$. In this case, T_{Small} is bounded by r^{w-1}. Each processor handles a segment of T_{Big}, assuming there are p processors. The total length of the segment is $|T_{Big}|$. Therefore, the space complexity for a processor is bounded by $O(r^w)$.

4.2. Table Multiplication and Division

After we obtain the extended table, the multiplication of two tables is straightforward and easy to parallelize. We first combine the variable vectors of T_1 and T_2 to form a union vector $V = V_1 \cup V_2$. Table T_1 and T_2 are then extended according to V. After the extension, each entry in T_1 corresponds to a unique entry in T_2, and vice versa. Thus, the table multiplication becomes entry-wise multiplication. The process is summarized in Algorithm 2.

Algorithm 2 Multiplication: TMul(T_1, T_2, V_1, V_2, R)

```
Input: Potential tables $T_1$ and $T_2$, variable vectors $V_1$, $V_2$, variable range $R$
Output: Table $T$ as the multiplication of $T_1, T_2$
  Create union variable vector $V = V_1 \cup V_2$
  Initialize $T$ according to $V$, where $|T| = \prod_i R_{V(i)}$
  $T'_1 =$ TExt($T_1, V_1, V$)
  $T'_2 =$ TExt($T_2, V_2, V$)
  for $i = 1$ to $|T|$ in parallel do
    $T(i) = T'_1(i) * T'_2(i)$;
  end for
```

When representing V_1 and V_2 as bit strings, creating the union variable vector costs O($|V_1| + |V_2|$). By allocating memory of size $\prod_i R_{V(i)}$, initialization of T can be completed in constant time. As we have analyzed in Algorithm 1, extending T_1 and T_2 costs $O(|V|\log(|V|) + |T| \cdot |V|\sum_i R_i/p)$, since $|V| > |V_1|$ and $|V| > |V_2|$. The entry-wise multiplication requires $O(|T|)$ time. Note that after table extension, both T_1 and T_2 have the same size $|T|$. The total computational complexity is $O(|V|\log(|V|) + (|T| \cdot |V|\sum_i R_i + |T|)/p)$ and the scalability range is $1 \le p \le |T|$. As $|T|$ is also bounded by r^w, when r and w are large, the computational complexity can be approximated by $O(w^2 r^{(w+1)}/p)$, $1 \le p \le |T|$.

Algorithm 3 Division: TDiv(T_1, T_2, V_1, V_2, R)

```
Input: Potential table $T_1$ and $T_2$, variable vector $V_1$, $V_2$, variable range $R$
Output: Table $T$ as the quotient of $T_1/T_2$
  Create union variable vector $V = V_1 \cup V_2$
  Initialize $T$ according to $V$, where $|T| = \prod_i R_{V(i)}$
  $T'_1 =$ TExt($T_1, V_1, V$)
  $T'_2 =$ TExt($T_2, V_2, V$)
  for $i = 1$ to $|T|$ in parallel do
    if $T'_2(i)$ == 0 then
      $T(i) = 0$;
    else
      $T(i) = T'_1(i)/T'_2(i)$;
    end if
  end for
```

In table multiplication, the total length of table assigned to each processor is T, bounded by $O(r^w)$. The lengths of V_1, V_2 and R are all bounded by $O(w)$, which can be ignored compared to $O(r^w)$. So, the space complexity for Algorithm 2 is $O(r^w)$.

The algorithm for table division is very similar to that of table multiplication. In case the denominator equals 0, we simply let the result be 0, according to the definition of conditional probability. The computational complexity and space complexity of table division are the same as those of table multiplication.

4.3. Table Marginalization

The input to table marginalization is a big table T_{Big} with variable vector V_{Big}, and a smaller variable vector $V_{Small} \subseteq V_{Big}$. The output is a small table T_{Small} and variable vector V_{Small}.

Algorithm 4 Marginalization: TMarg($T_{Big}, V_{Big}, V_{Small}$)

```
Input: Potential table $T_{Big}$, variable vector $V_{Big}$ and $V_{Small}$
Output: Result table $T_{Small}$ whose variable vector is $V_{Small}$
  if $M_{V_{Big}}^{V_{Small}}$ has not been computed then
    Find mapping vector $M_{V_{Big}}^{V_{Small}}$ from $V_{Small}$ to $V_{Big}$
  end if
  Initialize $T_{Small}$ by letting all entries be 0
  for $i = 1$ to $|T_{Big}|$ in parallel do
    Convert $i$ into value string $X^{(i)}$
    Create value string $Y^{(i)}$ from $X^{(i)}$ using $M_{V_{Big}}^{V_{Small}}$
    Convert $Y^{(i)}$ into index $j$
    $T_{Small}(j) = T_{Small}(j) + T_{Big}(i)$
  end for
```

In Algorithm 4, each processor receives V_{Big}, V_{Small} and a segment of T_{Big} of length $\lfloor T_{Big}/p \rfloor$. Each processor produces a segment of T_{Small} from the segment of T_{Big}. The computational complexity of marginalization is $O(|V_{Small}|\log(|V_{Big}|) + |T_{Big}| \cdot |V_{Big}|\sum_i R_i/p)$, which can be bounded by $O(w^2 r^{(w+1)}/p)$, $1 \le p \le |T_{Big}|$. As T_{Big}, $T_{Small}, V_{Big}, V_{Small}, M_{V_{Big}}^{V_{Small}}$ are stored in the memory and T_{Big} is dominant, the space complexity is given by $O(r^w)$.

4.4. Exact Inference with Node Level Primitives

The process of exact inference with node level primitives in junction trees is given in Algorithm 5. The input to this algorithm includes a junction tree with the potential tables for its cliques, the evidence variables and the query variables. The outputs are the probability tables for the query variables.

In Algorithm 5, $pa(A_i)$ is the parent of A_i in JT. $T(A_j)$ denotes the potential table of clique A_j. $V(A_j)$ is the variable vector of A_j. In array A, the cliques of JT are arranged by the Breadth First Search (BFS) order. $E == e$ judges if E is consistent with evidence e. $\delta(expr)$ is 1 if $expr$ is true; otherwise it is 0. Thus, $T(A_i) \cdot \delta(E == e)$ makes all entries in $T(A_i)$ inconsistent with e to be 0. The next two for-loops perform *evidence collection* to propagate evidence from leaf cliques to the root, and *evidence distribution* to propagate evidence from the root to leaf cliques. Evidence collection and distribution ensure the evidence is absorbed by all cliques [6]. Finally, we obtain the probability tables of query variables by marginalizing clique potential tables. $V(Q)$ is a variable vector consisting of query variables.

Assume the number of cliques in JT is N. The complexity of BFS is $O(N)$, since JT has N cliques and $N-1$ edges. In evidence collection and distribution, evidence propagation from one clique to another requires two table marginalization, two table extensions, one table multiplication and one table division. There are at most $O(N)$ node level operations. If we compute the BFS order of cliques *a priori*, the scalability of Algorithm 5 depends on the node level primitives. Therefore, the computational complexity is $O(w^2 r^{(w+1)} N/p)$, $1 \leq p \leq r^w$.

The dominant part of Algorithm 5 consists of four for-loops. Each iteration of these for-loops only utilizes a constant number of primitives. As the space complexity for each primitive is bounded by $O(r^w)$, the space complexity for each iteration is also bounded by $O(r^w)$. Therefore, the space complexity for Algorithm 5 is $O(Nr^w)$.

5. Implementation and Experimental Results

5.1. Computing Facilities

We implement the node level primitives using Message Passing Interface (MPI) on the DataStar Cluster at the San Diego Supercomputer Center (SDSC)[2] and on the clusters at the USC Center for High-Performance Computing and Communications (HPCC)[3].

The DataStar Cluster at SDSC employs IBM P655 nodes running at 1.5 GHz with 2 GB of memory per processor. This machine uses a Federation interconnect, and has a theoretical peak performance of 15 Tera-FLOPS. The DataStar Cluster runs Unix with MPICH. IBM Loadleveler was used to submit the jobs to the batch queue.

The USC HPCC is a Sun Microsystems & Dell Linux cluster. A 2-Gigabit/second low-latency Myrinet network connects most of the nodes. The HPCC nodes are Dual Intel Xeon 3.2 GHz with 2 GB memory, running USCLinux with MPICH and Portable Batch System (PBS).

5.2. Data Distribution

Note that we use the row-major ordering to index table entries. The potential table entries are distributed among the processors by the following way: Assuming the potential table is $T = (T_0, \cdots, T_{|T|-1})$, the k-th processor is in charge of $|T|/p$ entries: $(T_{k\cdot\lfloor|T|/p\rfloor}, \cdots, T_{\min((k+1)\cdot\lfloor|T|/p\rfloor, |T|-1)})$. V_1, V_2 (or V_{Small}, V_{Big}), R and the potential table of the separator (T_2 in Algorithm 2 and 3) are broadcast to all processors.

Algorithm 5 Exact Inference on Junction Tree

```
Input: Junction tree $JT$, evidence $e$ for variable $E$, query
  variables $Q$, variable range $R$
Output: Probability table of $Q$
  $A$ = Order cliques of $JT$ by BFS
  for $i$=1 to $|A|$ do
    if $E \in A_i$ then
      $T(A_i) = T(A_i) \cdot \delta(E == e)$
    end if
  end for
  for $i$ = $|A|$ to 2 by -1 do
    $V_{sepset} = V(pa(A_i)) \cap V(A_i)$
    $T_{sepset}$=TMarg($T(pa(A_i))$, $V(pa(A_i))$, $V_{sepset}$)
    $T^*_{sepset}$=TMarg($T(A_i))$, $V(A_i)$, $V_{sepset}$)
    $T_{divsep}$=TDiv($T^*_{sepset}$, $T_{sepset}$, $V_{sepset}$, $V_{sepset}$, $R$)
    $T^*$=TMul($T(pa(A_i))$, $T_{divsep}$, $V(pa(A_i))$, $V_{sepset}$,
    $R$)
    Update potential table by $T(pa(A_i)) = T^*$
  end for
  for $i$ = 2 to $|A|$ do
    $V_{sepset}$=$V(pa(A_i)) \cap V(A_i)$
    $T^*_{sepset}$=TMarg($T(pa(A_i))$, $V(pa(A_i))$, $V_{sepset}$)
    $T_{sepset}$=TMarg($T(A_i)$, $V(A_i)$, $V_{sepset}$)
    $T_{divsep}$=TDiv($T^*_{sepset}$, $T_{sepset}$, $V_{sepset}$, $V_{sepset}$, $R$)
    $T^*$=TMul($T(A_i)$, $T_{divsep}$, $V(A_i)$, $V_{sepset}$, $R$)
    Update potential table by $T(A_i) = T^*$
  end for
  for $i$=1 to $|A|$ do
    if $Q \in A_i$ then
      $T_{query}$=TMarg($T(A_i)$, $V(A_i)$, $V(Q)$)
    end if
  end for
```

5.3. Experimental Results

To test the performance of individual node level primitives, we randomly generate two potential tables given $w = 15$ and $r = 2, 3$. We use double precision and the memory needed by a clique potential table is 256KB ($r = 2$) and 110MB ($r = 3$). The separator set we used has 5 random variables, so the small table is 0.25KB ($r = 2$) and 2KB ($r = 3$). The memory taken by V_1, V_2 and R is ignored compared to the size of the clique potential table. We ran the program with 1, 2, 4, 8, 16, 32, 64 and 128 processors. The execution times are shown in Figures 3.

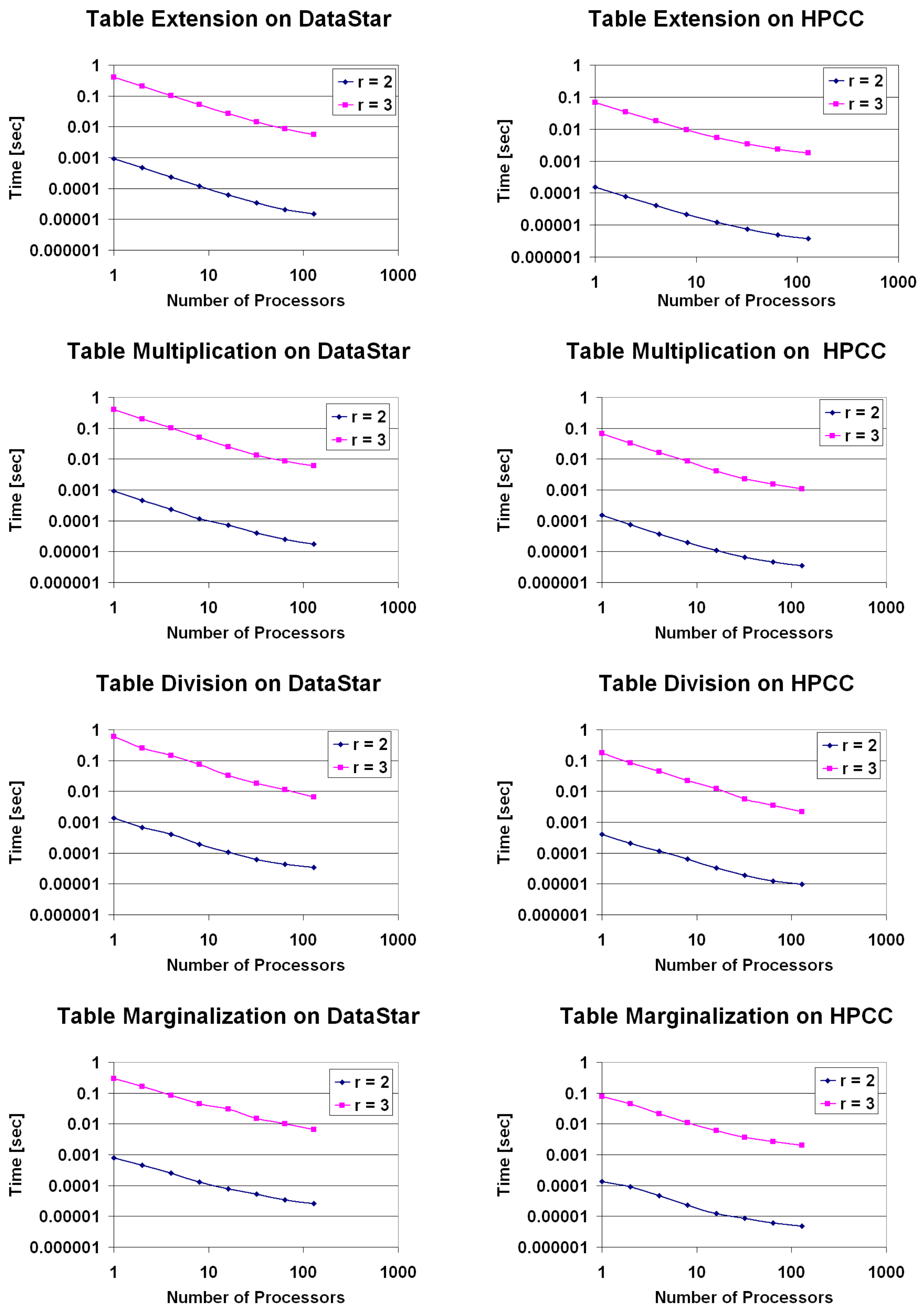


Figure 3. Execution time of node level primitives on DataStar and HPCC.

The random Bayesian networks used for exact inference were generated by the Bayesian Network Toolbox for Matlab[1]. We generated three networks with 1024 nodes. The number of states r was set as 2 and 3. As the potential table grew exponentially with w, we controlled the size of the cliques by limiting the degree of the nodes in the Bayesian networks. The maximal clique size w are 20, 14 and 11 respectively when we converted these Bayesian networks to junction trees. The number of cliques are 244, 318 and 427 respectively. We performed the exact inference with 1, 2, 4, 8, 16, 32, 64 and 128 processors. The results are shown in Figures 4.

The experimental results on two clusters show scalable performance of our implementations of all the primitives and of exact inference with various parameters, where the execution time decreases as the number of processors increases. Such results illustrate the efficiency and scalability of the proposed primitives. Note that we are not concerned with the individual speedups, which can be improved further, but the general scalability of our results.

6. Conclusions

In this paper, we categorized table-based operations in exact inference into four node level primitives: table extension, table multiplication, table division and table marginalization. We developed scalable parallel algorithms for these primitives, as well as exact inference using them. As part of our future work, we plan to study the analysis of the exact inference algorithm using a more realistic system model to take into account the message passing costs. Additionally, we will also explore the load balancing issues that might arise in the situation when r and w are different across different cliques, leading to different sized tables.

References

[1] The Bayesian Network Toolbox for Matlab (BNT). http://bnt.sourceforge.net/.

[2] Datastar Cluster at the San Diego Supercomputer Center (SDSC). http://www.sdsc.edu/us/resources/datastar/.

[3] USC center for High-Performance Computing and Communications (HPCC). http://www.usc.edu/hpcc/.

[4] D. Heckerman. Bayesian networks for data mining. In *In Data Mining and Knowledge Discovery*, 1997.

[5] A. V. Kozlov and J. P. Singh. A parallel lauritzen-spiegelhalter algorithm for probabilistic inference. In *Supercomputing*, pages 320–329, 1994.

[6] S. L. Lauritzen and D. J. Spiegelhalter. Local computation with probabilities and graphical structures and their application to expert systems. *J. Royal Statistical Society B*, 50:157–224, 1988.

[7] V. K. Namasivayam, A. Pathak, and V. K. Prasanna. Scalable parallel implementation of Bayesian network to junction tree conversion for exact inference. In *Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pages 167–176, 2006.

[8] V. K. Namasivayam and V. K. Prasanna. Scalable parallel implementation of exact inference in bayesian networks. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 143–150, 2006.

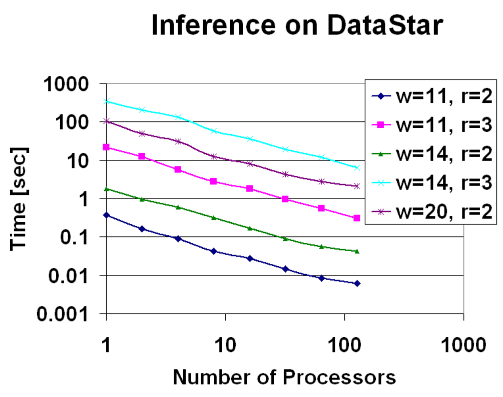
[9] D. Pennock. Logarithmic time parallel Bayesian inference. In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, pages 431–438, 1998.

[10] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[11] E. Segal, B. Taskar, A. Gasch, N. Friedman, and D. Koller. Rich probabilistic models for gene expression. In *9th International Conference on Intelligent Systems for Molecular Biology*, pages 243–252, 2001.

[12] R. D. Shachter, S. K. Andersen, and P. Szolovits. Global conditioning for probabilistic inference in belief networks. In *In Proceedings of the Tenth Conference on Uncertainty in Artifficial Intelligence*, pages 514–522, 1994.

[13] L. Yin, C.-H. Huang, and S. Rajasekaran. Parallel data mining of bayesian networks from gene expression data. In *Poster Book of the 8th International Conference on Research in Computational Molecular Biology*, pages 122–123, 2004.



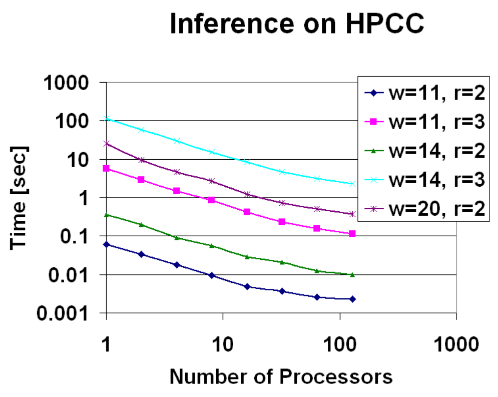


Figure 4. Execution time of exact inference on DataStar and HPCC.