Parallelism in the SPI Algorithm: An Investigation

By Tony Fountain

A Research Paper submitted to

Oregon State University

in partial fulfillment of the
requirements for the degree of

Master of Science

Completed April 24, 1991
Commencement June 1991.

ACKNOWLEDGMENTS

Thanks to my major professor, Bruce D'Ambrosio, for his support and guidance throughout this project. Thanks also to my other committee members, Professors Tom Dietterich and Vikram Saletore, for their assistance. I would also like to thank several friends and fellow graduate students for their assistance. Thanks to Brad Seevers and Ray Anderson for their help with concepts of parallelism, Lothar Kaul for his help with system support, and especially Nick Flann for discussion and critical feedback on all aspects of this project. Thanks.

Table of Contents

1	Introduction	1
2	Background	3
2.1	Belief Nets	3
2.2	The SPI Algorithm	4
2.3	Conformal product	6
3	Sources of Parallelism	10
3.1	Evaluation Tree Parallelism	10
3.2	Conformal Product Parallelism	12
4	Models	18
4.1	Sequential Model	18
4.2	Parallel Models	19
4.2.1	Conformal Product Model	19
4.2.2	Parallel Query Models	21
4.2.3	Evaluation Tree Parallelism Models	22
5	Simulations	23
5.1	Model Parameters	23
5.2	Nets	24
5.3	Results	25

5.3.1 Shared memory 26
5.3.2 Distributed memory 40
5.3.3 Evaluation Tree Parallelism 50

6 Conclusions 52
6.1 Shared Memory 52
6.2 Distributed Memory 53
6.3 Evaluation Tree Parallelism 54
6.4 Further research 54

Bibliography 57

List of Figures

2.1 A Simple Belief Net 4

2.2 Evaluation Tree for P(A) 5

2.3 Subtree of evaluation tree 7

2.4 Distributions 7

2.5 P(B | E) 8

3.1 Conformal Product Parallelism - 2 elements per process 13

3.2 Conformal Product Parallelism - 1 element per process 15

3.3 Conformal Product Parallelism - 1/2 element per process 17

5.1 Speedup - Dimension (G = 256) 27

5.2 Intel2 Speedup SM 28

5.3 Intel2 Efficiency SM 28

5.4 Intel2 Conformal Products - Multiplies 29

5.5 Intel2 Conformal Products - Log Multiplies 30

5.6 Intel2 Multiplies per Task 30

5.7 Intel2 Processors per Task 31

5.8 Net 4 Speedup SM 32

5.9 Net 4 Efficiency SM 33

5.10 Net 4 Conformal Products - Multiplies 33

5.11 Net 4 Conformal Products - Log Multiplies 34

5.12 Net 4 Multiplies per Task 35

5.13 Net 4 Processors per Task 35

5.14 Net 9 Speedup SM 37

5.15 Net 9 Efficiency SM 37

5.16 Net 9 Conformal Products - Multiplies 38
5.17 Net 9 Conformal Products - Log Multiplies 39
5.18 Net 9 Multiplies per Task 39
5.19 Net 9 Processors per Task 40
5.20 Intel2 Speedup DM 41
5.21 Intel2 Efficiency DM 42
5.22 Intel2 Compute - Communicate DM G = 2 43
5.23 Intel2 Compute - Communicate DM G = 64 43
5.24 Intel2 Compute - Communicate DM G = 256 44
5.25 Net 4 Speedup DM 45
5.26 Net 4 Efficiency DM 45
5.27 Net 4 Compute - Communicate DM G = 2 46
5.28 Net 4 Compute - Communicate DM G = 64 46
5.29 Net 4 Compute - Communicate DM G = 256 47
5.30 Net 9 Speedup DM 48
5.31 Net 9 Efficiency DM 48
5.32 Net 9 Compute - Communicate DM G = 2 49
5.33 Net 9 Compute - Communicate DM G = 64 49
5.34 Net 9 Compute - Communicate DM G = 256 50

List of Tables

5.1 Random Net Descriptions 24
5.2 Net Summary Statistics (G = 256) 26
5.3 Evaluation Tree Parallelism 51

Chapter 1

Introduction

The Symbolic Probabilistic Inference (SPI) algorithm was developed by Bruce D'Ambrosio for efficient calculation of prior probabilities in belief nets [2]. Although the complexity of the SPI algorithm compares favorably with other approaches to probabilistic inference [5], its actual running time is still prohibitively long even for moderately sized nets. As part of our overall research goal of developing decision-theoretic problem solving programs for real world real-time problems, we undertook this investigation into the feasibility of developing a parallel version of the SPI algorithm.

The specific goals of this project were the following:

1. Identify those operations in the SPI algorithm that, if performed in parallel, would lead to some improvement in performance.

2. Estimate the performance of actual parallel implementations of SPI on both shared memory and cube architectures.

3. Perform a preliminary exploration into the practical issues related to developing a parallel algorithm.

To accomplish these goals the following tasks were performed:

1. Examined the algorithm to identify potentially parallelizable constructs and data dependencies.

2. Designed and implemented models to simulate running the SPI algorithm in parallel.

3. Ran the models on test nets under various parameter settings simulating shared memory and cube architectures and various computational grainsizes.

4. Gathered and analyzed the test results.

The remainder of the paper is organized as follows. Chapter 2 provides the necessary background information for understanding the SPI algorithm, including an overview of belief nets and an example of inference in SPI. Chapter 3 explores the sources of parallelism in the SPI algorithm. Chapter 4 describes our approach to estimating the performance of parallel versions of the SPI algorithm. Chapter 5 presents the results from simulated runs of a parallel SPI algorithm. Finally Chapter 6 includes a brief discussion of the results of this project and introduces some issues for future research.

Chapter 2

Background

In order to understand the issues involved in the construction and analysis of a parallel version of the SPI algorithm it is necessary to understand, at least at a general level, the sequential SPI algorithm. Probabilistic inference is the process of computing the posterior probabilities of values of variables in a probabilistic model. In simple terms, one begins with a model of the system of interest in terms of variables that describe the system, the possible values of these variables, and a set of probability distributions that represent the likelihoods that the model variables take on those values. The initial values of the probability distributions represent expectations about variables' values prior to any observations of the actual system. Once observations are made the probability distributions of variables corresponding to the observed features of the system are updated. Given such a model, probabilistic inference is the process of computing the probabilities that certain variables have certain values. SPI is one of a class of algorithms that perform probabilistic inference over a specific type of probabilistic model referred to as belief networks, Bayesian networks, causal nets, or simply belief nets [7].

2.1 Belief Nets

A belief net consists of a set of nodes and a set of arcs. The nodes represent the variables of the model, and the arcs represent influence, causation, or relevance among the variables [7] [6]. Figure 2.1 is a graphical representation of a simple belief net.

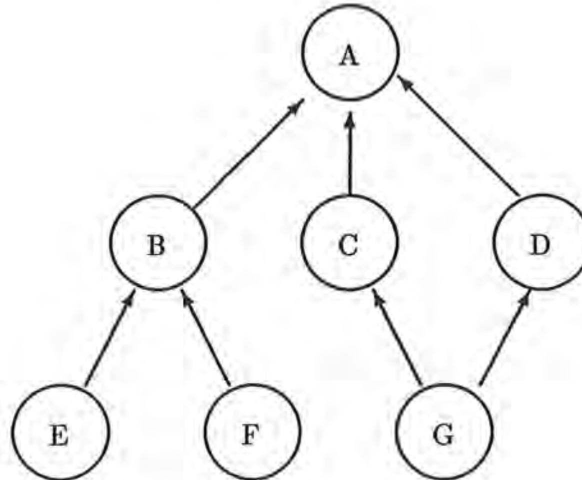

Figure 2.1: A Simple Belief Net

The belief net representation of a probabilistic model consists of a set of marginal probability distributions and a set of conditional probability distributions. A marginal probability distribution is defined for each node with no incoming arcs. A conditional probability distribution is defined for each node with incoming arcs, where each conditioning variable corresponds to an incoming arc. From a belief net representation, any subspace of the full joint probability distribution across the variables of the model can be generated using the product rule [7]. Belief net queries are requests to generate such subspaces. SPI is an algorithm for answering queries over belief nets.

2.2 The SPI Algorithm

There are three stages to the SPI algorithm:

1. Transform the belief net into a representation that makes inference easier.

2. Update the representation in response to observations.

3. Compute posterior probabilities in response to queries, referred to as query processing.

Each of the three stages of the SPI algorithm is of a different computational complexity. The first stage is performed only once and is polynomial in the number of nodes and arcs in the net. The second stage is constant time with respect to the number of nodes or arcs. The third stage is exponential in the worst case [2]. Since stage 3 consumes the largest part of the processing time, we concentrated our analysis on that stage.

Query processing can be thought of as two separate processes: query expansion and evaluation. Query expansion is the process of determining the distributions necessary for answering a query and an order in which to combine these distributions. Query evaluation is the process of combining the distributions of the expanded query to get the result distribution. Although the sequential SPI algorithm interweaves query expansion and evaluation, the entire expanded query can be constructed first and then evaluated. An expanded query can be represented as an evaluation tree. Figure 2.2 illustrates the evaluation tree that is generated in response to a query for the marginal probability of variable A from the belief net of Figure 2.1.

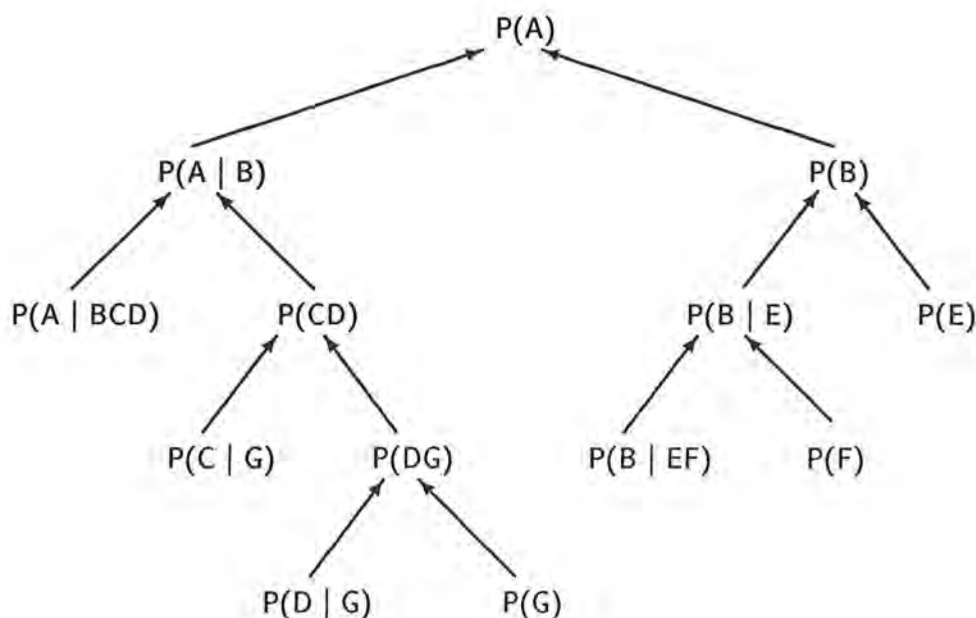

Figure 2.2: Evaluation Tree for P(A)

The expanded query corresponding to the evaluation tree in Figure 2.2 is:

$$P(A) = \sum_{B}(\sum_{CD} P(A|BCD)(\sum_{G} P(C|G)(P(D|G)P(G))))(\sum_{E}(\sum_{F} P(B|EF)P(F))P(E))$$

The complexity of query expansion is $O(n^3)$ in the number of unique variables in the query [3].

Evaluation trees, and thus expanded queries, are not unique. One topic of current research is to find alternative methods for generating expanded queries [8]. For this project we chose one method and used it for all simulations. Current methods for generating evaluation trees are oriented towards efficient sequential evaluation. One important topic for future research is query expansion methods which maximize the parallelism of the query evaluation process.

Evaluation of an expanded query consists of repeatedly combining the distributions together, two at a time, with the conformal product operator, as described in the following section.

2.3 Conformal product

The fundamental operation in query evaluation is the combining of two probability distributions. This is referred to as the conformal product operation in [8]. As an example of the conformal product operation consider the subtree of the evaluation tree of Figure 2.2 which corresponds to the calculation of the P(B | E). As illustrated in Figure 2.3 the $P(B|E)$ is calculated as follows:

$$P(B|E) = \sum_{F} P(B|EF)P(F)$$

.

To calculate the value of P(B | F) first assume that all variables have the same value space of {0, 1} and that the probability distributions for P(B | E) and P(F) are defined as in Figure 2.4.

Then P(B | E) can be calculated as follows. Notice that there are four entries in the conditional probability distribution of P(B | E):

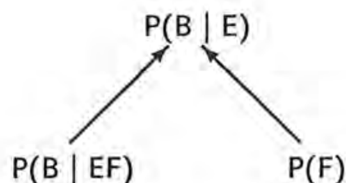

Figure 2.3: Subtree of evaluation tree

P(B | EF):

E F	B = 1	B = 0
1 1	.1	.9
1 0	.2	.8
0 1	.7	.3
0 0	.6	.4

P(F):

1	0
.6	.4

Figure 2.4: Distributions

1. P(B = 1 | E = 1)
2. P(B = 1 | E = 0)
3. P(B = 0 | E = 1)
4. P(B = 0 | E = 0)

Since the variable F is not one of the variables in the result distribution P(B | E), the entries in the full joint probability distribution that distinguish between values of F are summed over. The actual values of the entries in P(B | E) are computed by:

1. $$\begin{aligned} P(B = 1 \mid E = 1) &= P(B = 1 \mid E = 1, F = 1) * P(F = 1) + \\ &\quad P(B = 1 \mid E = 1, F = 0) * P(F = 0) \\ &= 0.1 * 0.6 + 0.2 * 0.4 \\ &= 0.14 \end{aligned}$$

2. $$\begin{aligned} P(B = 1 \mid E = 0) &= P(B = 1 \mid E = 0, F = 1) * P(F = 1) + \\ &\quad P(B = 1 \mid E = 0, F = 0) * P(F = 0) \\ &= 0.7 * 0.6 + 0.6 * 0.4 \\ &= 0.66 \end{aligned}$$

3. $$\begin{aligned} P(B = 0 \mid E = 1) &= P(B = 0 \mid E = 1, F = 1) * P(F = 1) + \\ &\quad P(B = 0 \mid E = 1, F = 0) * P(F = 0) \\ &= 0.9 * 0.6 + 0.8 * 0.4 \\ &= 0.86 \end{aligned}$$

4. $$\begin{aligned} P(B = 0 \mid E = 0) &= P(B = 0 \mid E = 0, F = 1) * P(F = 1) + \\ &\quad P(B = 0 \mid E = 0, F = 0) * P(F = 0) \\ &= 0.3 * 0.6 + 0.4 * 0.4 \\ &= 0.34 \end{aligned}$$

The resulting conditional probability distribution of P(B | E) is shown in table form in Figure 2.5.

		B	
		1	0
E	1	.14	.86
	0	.66	.34

Figure 2.5: P(B | E)

The number of multiplies in a conformal product is the number of entries in the full joint probability distribution over all of the unique variables in the two distributions to be combined. This number can be calculated by $\prod_{v \in V} n(v)$ where V is the set of unique variables, and $n(v)$ is the number of possible values for the variable v. In the preceeding example V = {B, E, F}, and $n(B) = n(E) = n(F) = 2$. Therefore the number of entries in the full joint probability distribution, and the number of multiplies in the corresponding conformal product, is $2^3 = 8$.

A conformal product operation involves indexing, multiplying, and adding. The number of multiplies is an accurate measure of complexity, since indexing is only $O(m\ log\ n)$, where m is the overall number of multiplies and n is the number of unique variables in the query, and the number of additions is approximately the same as the number of multiplies. The complexity of the query evaluation process, as measured by the number of multiplies, is $O(V^n)$, where V is the maximum number of values of any variable in the query and n is the number of unique variables in the expanded query.

Chapter 3

Sources of Parallelism

Since query processing dominates the processing time of the SPI algorithm, we only considered parallelism in that portion. Furthermore, due to the relatively low complexity of query expansion, we focused our analysis on query evaluation exclusively. The purpose of this chapter is to introduce those aspects of the query evaluation process whose parallelism is analyzed more rigorously in Chapters 4 and 5.

3.1 Evaluation Tree Parallelism

One source of parallelism in the query evaluation process of the SPI algorithm is at the evaluation tree level. The idea behind this approach is that the running time for query evaluation can be reduced by performing some conformal product operations in parallel. Since each non-leaf node in the evaluation tree corresponds to a conformal product operation, and since distributions occur at most once in the evaluation tree, the conformal product operations corresponding to nodes in disjoint subtrees can be performed in parallel. For example, from the evaluation tree of Figure 2.2 it is easy to see that the subtrees rooted at P(A | B) and at P(B) can be evaluated in parallel. More generally, the structure of the evaluation tree expresses the data dependencies among the conformal product operations and bounds on parallelism can be calculated from an analysis of this structure. For example, in the evaluation tree illustrated in Figure 2.2, we can see the following ordering of conformal product operations:

$$CP(DG) < CP(CD) < CP(A|B) < CP(A) \ and \ CP(B|E) < CP(B) < CP(A)$$

where $CP(X)$ represents the conformal product operation that calculates $P(X)$. For example, $CP(DG) \Rightarrow P(DG) = P(D|G) * P(G)$. Recall that the measure of complexity of a conformal product operation is the number of multiplies in its evaluation. Thus a lower bound on the computational time for the evaluation of this query is the length of time it would take to evaluate the longest path in the evaluation tree, where path length is measured in terms of the number of multiplies required for the conformal products in the path. For example, let $M(c)$ be the number of multiplies in the evaluation of conformal product c, $L(p)$ be the length of path p, $P_1 = CP(DG) < CP(CD) < CP(A|B) < CP(A)$, and $P_2 = CP(B|E) < CP(B) < CP(A)$. Then the lengths of the paths P_1 and P_2 can be calculated as follows:

$$\begin{aligned} L(P_1) &= M(CP(DG)) + M(CP(CD)) + M(CP(A|B)) + M(CP(A)) \\ &= 4 + 8 + 16 + 4 \\ &= 32 \end{aligned}$$

$$\begin{aligned} L(P_2) &= M(CP(B|E)) + M(CP(B)) + M(CP(A)) \\ &= 8 + 4 + 4 \\ &= 16 \end{aligned}$$

So the lower bound on computational time for the evaluation of this query is the time it takes to perform 32 floating point multiplies. In this example the longest path in terms of multiplies is also the longest path in terms of the number of conformal products it contains, but since the complexity of conformal products varies there is no guarantee that this will always be the case. Using the same approach as was used in calculating the lower bound, we can calculate an upper bound by summing the number of multiplies for each conformal product in the evaluation tree. The upper bound thus computed is 44 and corresponds roughly to the sequential running time for the evaluation of this query.

The lower bound computed above is, of course, only a theoretical bound, and

is based on several unrealistic assumptions, including:

1. The evaluation of the shorter path, P_2, can be carried out in parallel with the evaluation of P_1.

2. The query evaluation process consists only of conformal products, i.e., there are no inherently sequential operations that cannot be distributed equally among the processors performing the confomal product operations.

3. There are no communication costs for partitioning the evaluation tree among multiple processors.

4. There is no limit on the number of processors, in other words, there is at least one processor available for each subtree of the evaluation tree.

In Chapter 4 we present a more detailed model for estimating the performance of an SPI algorithm that exploits evaluation tree parallelism, and in Chapter 5 we present the results from simulations of this algorithm on sample belief nets and queries. But even with the rough estimates of running time presented above, it is clear that there will not be much performance improvement unless the evaluation trees are relatively bushy.

3.2 Conformal Product Parallelism

A second source of parallelism in the query evaluation process is at the conformal product level. The idea here is to parallelize each conformal product operation. Recall the evaluation tree illustrated in Figure 2.3 for the calculation of P(B | E), which consists of a single conformal product operation $P(B|EF) * P(F)$. Recall also that the result distribution, $P(B|E)$, has four entries. One approach to parallelizing the conformal product is to calculate subsets of the entries in the result distribution in parallel. For example, Figure 3.1 shows a possible evaluation of this conformal product in which two processes are used and each process computes two elements of the result distribution.

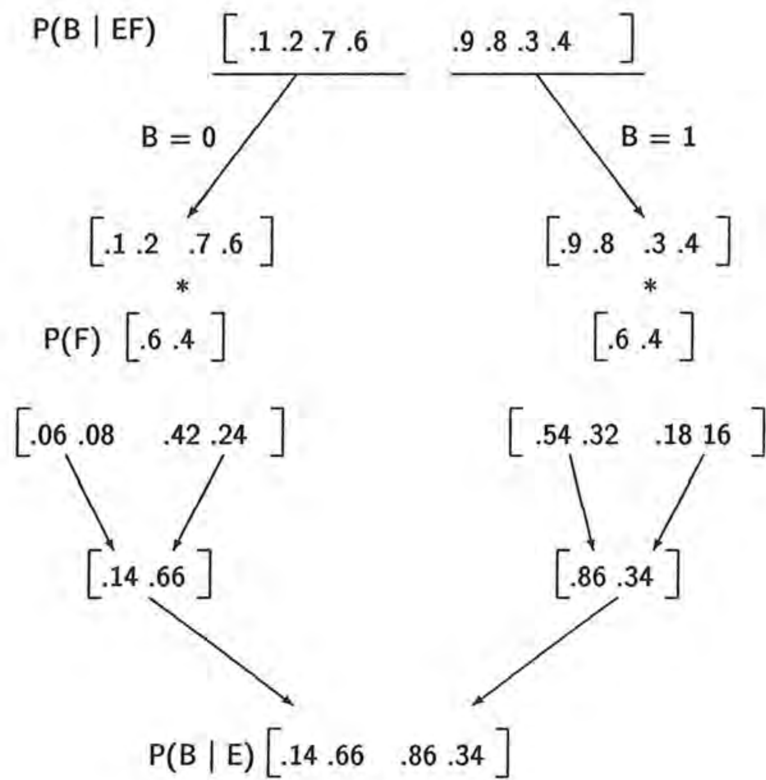

Figure 3.1: Conformal Product Parallelism - 2 elements per process

To understand this figure, begin at the top with the distribution $P(B|EF)$. This distribution is split into two equal subarrays. The left half corresponds to the values of $P(B|EF)$ in which B = 0, the right half corresponds to the values in which B = 1. The entries in each of these two subarrays are multiplied by the appropriate entries in the distribution $P(F)$. Notice that if we assume that the two multiply operations, [.1 .2 .7 .6] * [.6 .4] and [.9 .8 .3 .4] * [.6 .4], are performed in parallel then each of the corresponding processes requires access to the entire distribution $P(F)$. The results of these multiplies are two arrays of four entries each. Together these eight numbers represent the joint probability distribution $P(BF|EF)$. Since the result distribution does not distinguish between the values for variable F, the values of the joint probability distribution which distinguish between the values of F are summed over. So each four element array yields two elements of the result distribution. Of course it would never make sense to partition the problem into such small computations. This example, like the two which follow, is only to illustrate the concept behind conformal product parallelism. Part of the analysis in Chapter 4 is directed towards determining appropriate computational grainsizes.

As a second example, consider partitioning the same conformal product in such a way that each process computes only one result distribution entry instead of two. Figure 3.2 illustrates this evaluation process. In this example there are four processes, each of which computes one element of the result distribution. Each of these four processes requires access to $\frac{1}{4}$ of the distribution for $P(B|EF)$ and the entire distribution $P(F)$. An important point to get from this example is that as parallelism increases, the amount of data for each process decreases, but the overall amount of data needed by all processes increases. In the first example there were 2 processes, and each required 6 numbers for a total of 12. In the second example there were 4 processes, and each required 4 numbers for a total of 16. The reason for this increase in the total amount of data is that we split the first distribution, $P(B|EF)$, over variable B, and B is not an element of the second distribution, $P(F)$. Splitting over variables that occur in both input distributions results in no net increase in the total amount of data required to compute the conformal product.

This is demonstrated in the following example.

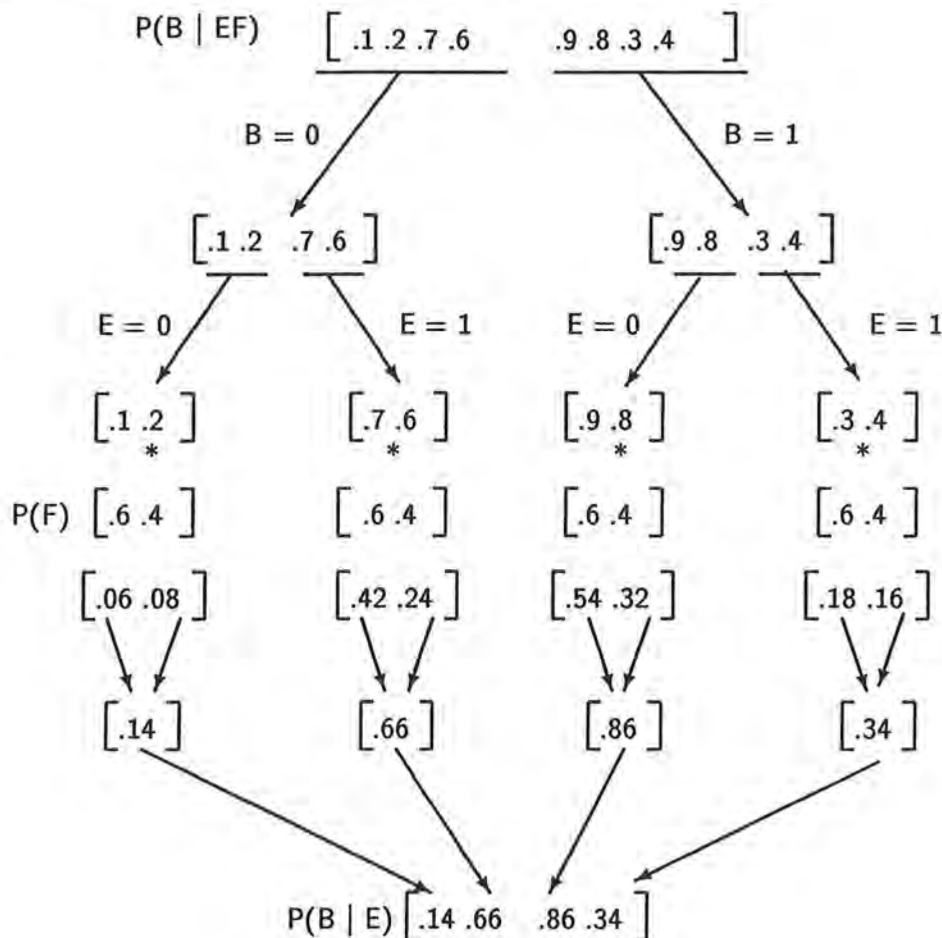

Figure 3.2: Conformal Product Parallelism - 1 element per process

In Figure 3.3 a final example is presented. In this example eight processes are used and each process computes $\frac{1}{2}$ of one element of the result distribution. The two important points to get from this last example are (1) splitting the input distributions on variables that occur in both distributions does not increase the total amount of data required to compute the conformal product, and (2) splitting the input distributions on variables other than those in the result distribution requires either the ability to perform concurrent writes with summing or a separate processing step in which the values that make up the result distribution entries are summed together. In other words, splitting the input distributions on variables

that occur in both distributions minimizes the total data required, and as long as splitting is performed on variables that occur in the result distribution the processes produce independent subsets of the result distribution. If splitting occurs on other variables, this independence is lost. Of course, splitting on more variables increases the amount of parallelism, but at the cost of complicating the process of assembling the result values. For this project, we only investigated the parallelism available from splitting on subsets of the variables in the result distribution.

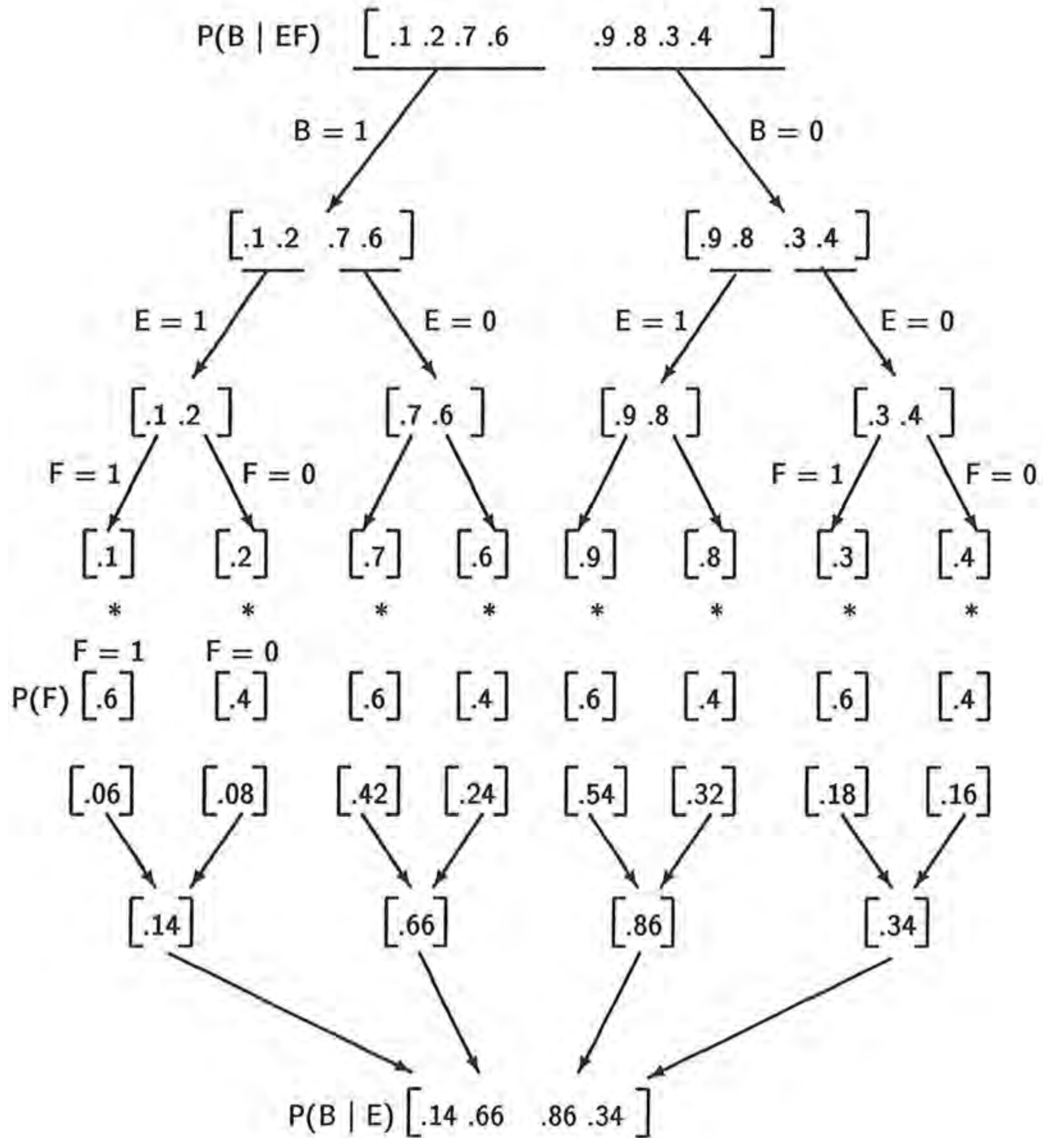

Figure 3.3: Conformal Product Parallelism - 1/2 element per process

Chapter 4

Models

For this analysis we considered conformal product and simple evaluation tree parallelism only. Our goal was to estimate the expected performance, in terms of speedup and efficiency, of parallel evaluation of expanded queries. Our approach was to modify the existing SPI code to create parameterized simulation models, run these models on sample nets, and use the statistics gathered from these runs to estimate the performance of actual parallel implementations.

This approach required the development of several models. First we developed a model of sequential conformal product evalutation. From this model we developed a model for sequential query evaluation. Then we developed a general model for parallel conformal product evaluation. This general model allowed us to model both shared-memory and distributed-memory architectures. From the parallel conformal product model we developed a model of parallel query evaluation. Each of the models was parameterized to allow experimentation with various aspects of the computational environment including task grainsizes, number of processors, and communication costs.

4.1 Sequential Model

To provide a basis against which to measure the performance of the parallel algorithms, we developed the following model to estimate the running time of the sequential algorithm.

Sequential conformal product model: $T_s(c) = \alpha M(c)$

where M(c) is the number of multiplies in conformal product c and α is a constant scaling factor that accounts for factors other than multiplies in the conformal product operation, most notably indexing and additions. M(c) is calculated by $\prod_{v \in V} n(v)$ as explained in section 2.2.1. To simplify analysis we restricted the value space of the variables in the test belief nets to 2 values each. Thus in the following sections $M(c) = 2^n$ where n is the number of unique variables in the 2 input distributions.

Given the model for the sequential running time for a single conformal product, the estimated sequential running time for the evaluation of an entire query is the sum of the times required for each of its conformal products.

Sequential model for a query: $T_s(q) = \sum_{c \in q} T_s(c)$

4.2 Parallel Models

4.2.1 Conformal Product Model

The following model was developed to estimate the running time of a parallel alglorithm.

Parallel conformal product model: $T_p(c) = P + S + W + C$

Where

P is the cost for process initialization,

S is the cost for setting the problem up,

W is the cost for the work done at each processor, and

C is the cost for communication.

The value which was used for W was αG, where G is the computational grainsize specified as the number of floating point multiplies per process. α is the same constant scaling factor that was used in the sequential model.

For both the shared-memory and cube architecture models the following measures were calculated for each conformal product:

T_s = Time for sequential

T_p = Time for parallel

N_u = Number of processors used

The values for G and N_u were determined as follows. First a minimum grainsize, G_{min}, and a maximum number of processors, N_a, were specified. N_a represents the number of processors available. Then, given a particular conformal product to compute, the actual G and N_u values were calculated so that N_u was maximized under the constraints $N_u \leq N_a$ and $G \geq G_{min}$ and $N_u \leq 2^v$, where v is the number of variables in the result distribution. In other words, N_u and G were chosen such that as many as possible of the available processors were used as long as there was enough work for each processor to perform as specified by the minimum grainsize, and there was enough parallelism in the problem to support the desired partitioning.

Distributed-Memory Communication Model

The distributed-memory model includes a specific model of communication for a cube architecture. This model assumes that there is no overlap between the conformal product calculations and the communication between processors. This model also assumes that the data to be sent to processors is arranged into buffers, one buffer for each process.

Total communication cost: $C = C_d + C_r + B$

where

C_d is the communication cost for distributing the data,

C_r is the cost of returning the data, and

B is the cost for building the buffers of data to be distributed.

The data transfer cost calculations were based on the log spanning tree, or broadcast, communication model for hypercubes [4].

Distribution communication cost: $C_d = (D_{max} * C_{st}) + (B_d * ((N_u - 1) * C_b))$

where

D_{max} is the maximum dimension of the cube which is used,

C_{st} is the communication startup time,

B_d is the number of bytes sent to each processor, and

C_b is the communication cost per byte, per link.

The following formula was used to calculate B_d, the number of bytes sent to each processor:

$B_d = B_{total} \ / \ N_u$

where B_{total} is the total number of bytes needed to compute the conformal product. Assuming 4 bytes per word, a lower bound for B_{total} can be calculated by the following formula:

$B_{total} = 4 \ * \ (2^{max(|dist1-vars|,|splitting-vars|)} + 2^{max(|dist2-vars|,|splitting-vars|)})$

where |dist1-vars| and |dist2-vars| represent the number of variables in the 2 input distributions and |splitting-vars| represents the number of result distribution variables on which splitting occurs.

The return communication cost is calculated in the same way as distribution communication costs except that the amount of data returned is less than that sent out.

Return communication cost: $C_r = (D_{max} * C_{st}) + (B_r * ((N_u - 1) * C_b))$

where

B_r is the number of bytes returned from each processor and is calculated by

$B_r = B_{result} \ / \ N_u$

where B_{result} is the total number of bytes in the result distribution. Since there are $2^{|result-dist-vars|}$ entries in the result distribution, $B_{result} = 4 * 2^{|result-dist-vars|}$.

4.2.2 Parallel Query Models

As explained in Chapter 2, query evaluation consists of repeated comformal product operations. Since we were interested in the performance of a parallel SPI algorithm on the task of query evaluation we constructed a model to predict this performance from the models for conformal product operations. The parallel query model is analogous to the sequential query model. The running time for parallel query evaluation is simply the sum of the running times of its conformal products.

Parallel model for a query: $T_p(q) = \sum_{c \in q} T_p(c)$

$N_u(q)$, the number of processors used in evaluating query q, is simply the

maximum of the number used by any of the conformal products in q. Given $T_p(q)$, $T_s(q)$ and $N_u(q)$ the speedup, cost, and efficiency for a query can be calculated according to fomulas given in [1].

Speedup: $S(q) = T_s(q) \,/\, T_p(q)$

Cost: $C(q) = T_p(q) * N_u(q)$

Efficiency: $E(q) = T_s(q) \,/\, C(q) = S(q) \,/\, N_u(q)$

4.2.3 Evaluation Tree Parallelism Models

For evaluation tree parallelism, we only computed a lower bound on running time. As demonstrated in the example in Section 3.1 a lower bound on the running time of an algorithm exploiting evaluation tree parallelism can be calculated by summing the times required to perform each of the conformal products in the longest path of the evaluation tree.

Lower bound on $T_p(q) = \sum_{c \in Lp} T_s(c)$ where Lp is the longest path in the evaluation tree as measured by the amount of time it takes to compute the conformal products in the path.

Chapter 5

Simulations

In order to estimate the performance of parallel query evaluation we instantiated the model parameters, chose sample belief nets and queries, ran simulations and gathered and analyzed the results.

5.1 Model Parameters

The following are the values for model parameters which were used in the simulations described in the following sections of this paper.

α, the scaling factor for multiplies, was 45 micro seconds. This value was derived from repeated timings of the sequential algorithm running on a Sun SPARC-station 1 and is approximately constant with respect to the number of variables in a query.

P, the cost for process initialization, was 0. There are two reasons for essentially ignoring the cost of process initialization. First there is no cost for process initialization in a cube. Second, even for a shared-memory machine, the cost of process initialization is small, about 500 micro seconds on a Sequent Balance, and as task grainsize increases, this cost becomes insignificant.

S, the cost for setting the problem up, was also 0. The main reason we assumed a value of 0 for the set-up cost was that we did not have a more reliable estimate. By ignoring the set-up cost, our model overestimates the actual speedup possible. Determining a realistic estimate for S should be one of the future research tasks.

For the distributed-memory model, the communication cost parameters, C_{st} and C_b, were based on the Intel iPSC2.

C_{st}, the communication start-up time, was 230 micro seconds.

C_b, the communication cost per byte, was .5 micro seconds per byte per link.

B, the cost for building the buffers of data to be distributed, was 0. This cost is a significant factor in the overall running time of the parallel distributed-memory algorithm. By ignoring this cost, our model overestimates the actual speedup possible. Exploring the costs and effects of building data buffers is also a topic for future research.

Simulations were run on a number of nets using the above parameters and various values for G, the computational grainsize, and N_a, the number of processors available. The actual values for G and N_a are given next to the simulation results in which they figure.

5.2 Nets

To obtain the sample belief nets on which to run the simulations, we generated 10 random nets using J. Suermondt's random net generator. In addition to these 10 nets, we ran simulations on a net obtained from Intel Corporation, hereafter referred to as the Intel2 net. Table 5.1 provides a description of the random nets.

Net	Nodes	Arcs	Observations	Query
1	66	3	11	51
2	74	3	2	28
3	70	4.2	6	60
4	71	2.6	16	53
5	46	3	12	17
6	29	4.7	12	22
7	68	3.5	10	43
8	86	4.3	13	27
9	50	4.0	16	27
10	60	3.5	8	36

Table 5.1: Random Net Descriptions

In Table 5.1 values under the nodes column indicate the total number of nodes that are in the belief net; values under the arcs column indicate the average number of incoming arcs to each node; values under the observations column refer to the number of observations posted to the net prior to querying; and the values under the query column indicates which node was queried. These parameters were randomly chosen by the net generator under the following constraints:

$$10 < \text{Nodes} < 100$$
$$1 < \text{Arcs} < 5$$
$$1 < \text{Observations} < 20$$

The query node was chosen at random from the nodes in the net that were not observed.

The Intel2 net has 52 nodes and 28 observations and an average of 2.4 arcs per node.

5.3 Results

Table 5.2 shows the results obtained from running each of the 11 nets under both the shared and distributed-memory models with the minimum grainsize set at 256 and the maximum number of processors limited to 1024.

As a measure of problem complexity, each value in the dimension column represents the log of the number of multiplies of the largest conformal product in the associated query's evaluation. As discussed in Chapter 2, the number of multiplies in a conformal product is equal to the number of entries in the full joint probability distribution over all unique variables in the input distributions. The full joint distribution can be thought of as an n-dimensional table with 2^n entries, where n is the number of unique variables in the input distributions. Since even in moderately-sized belief nets it is common for the evaluation of some conformal products to require millions of multiplies, it is often convenient to describe the

Net	Dimension	N_u	Shared Memory		Distributed Memory	
			Speedup	Efficiency	Speedup	Efficiency
1	15	128	26.62	.10	11.70	.09
2	10	4	2.45	.60	2.19	.55
3	29	1024*	1022.00	.99	20.00	.02
4	12	16	4.83	.30	3.68	.23
5	17	512	75.27	.15	14.88	.03
6	9	2	1.23	.61	1.19	.60
7	26	1024*	1011.00	.99	6.34	.02
8	34	1024*	1023.00	.99	20.14	.02
9	20	1024*	572.58	.56	16.36	.02
10	18	1024*	195.45	.19	11.91	.55
Intel2	14	32	10.19	.32	6.28	.20
* N_u was limited to 1024.						

Table 5.2: Net Summary Statistics (G = 256)

complexity of query evaluation in terms of the dimensionality of the full joint probability distribution, rather than in the number of multiplies it takes to construct this distribution.

From the table, it is easy to see that communication costs and problem size both have a large effect on the performance of parallel query evaluation. Under the shared-memory model the three largest queries all show a near-linear speedup and high efficiency. Under the distributed-memory model the same queries show a top speedup of only 20.14 and an efficiency of only .02. The effects of communication costs and problem size are shown clearly in the Figure 5.1.

In order to understand better the issues affecting the expected performance of parallel query evaluation, three nets were chosen to explore in more detail. These nets were the Intel2 net, random net 4, and random net 9.

5.3.1 Shared memory

Since communication is free in the shared-memory model, the only factors that affect the degree to which a query can be parallelized are the the minimum task grainsize, the size of intermediate result distributions, and the number of processors available.

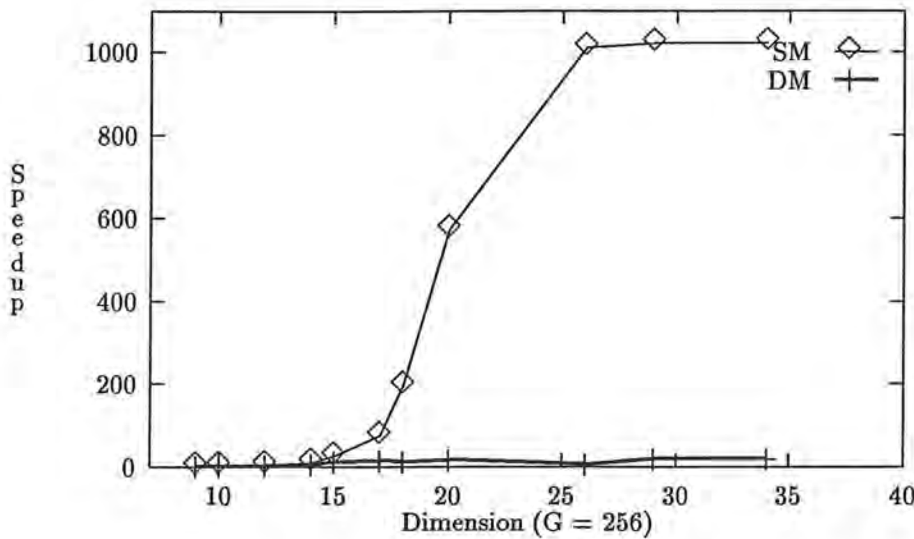

Figure 5.1: Speedup - Dimension (G = 256)

Task grainsize affects possible parallelism in the obvious way; if a conformal product is smaller than the minimum grainsize, then it can't be parallelized. The sizes of intermediate result distributions limit exploitable parallelism since splitting can occur only on those variables that occur in the result distributions as explained in Chapter 3. Finally, on the environment side, parallelism can be limited by the number of processors available. In each of the following nets we identify those factors that directly affect the predicted speedup and efficiency.

Intel2 Net

Figure 5.2 shows the expected speedup of a query on the Intel2 graph under the shared-memory model for grainsizes 2, 16, 64, and 256. As one would expect, without a communication cost the best speedups are achieved with the smallest minimum grainsize. Efficiency is also highest with the smallest grainsize as shown in Figure 5.3.

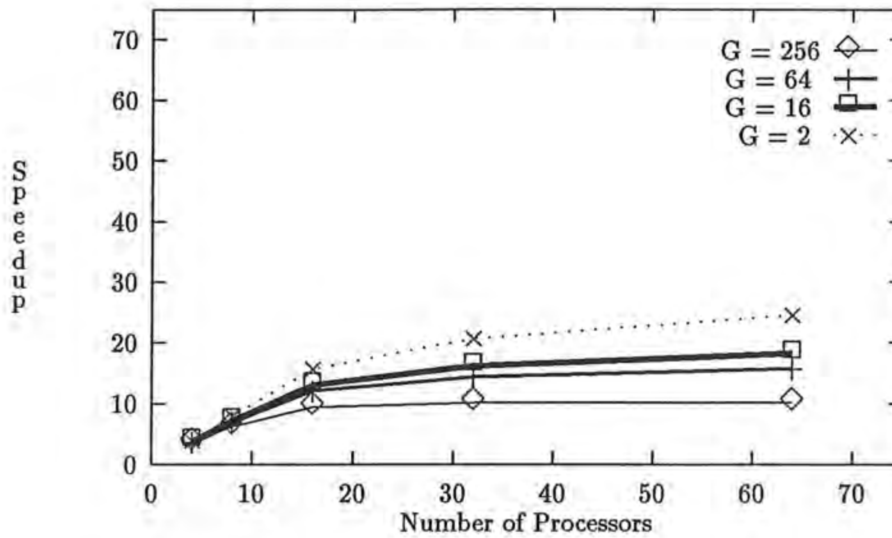

Figure 5.2: Intel2 Speedup SM

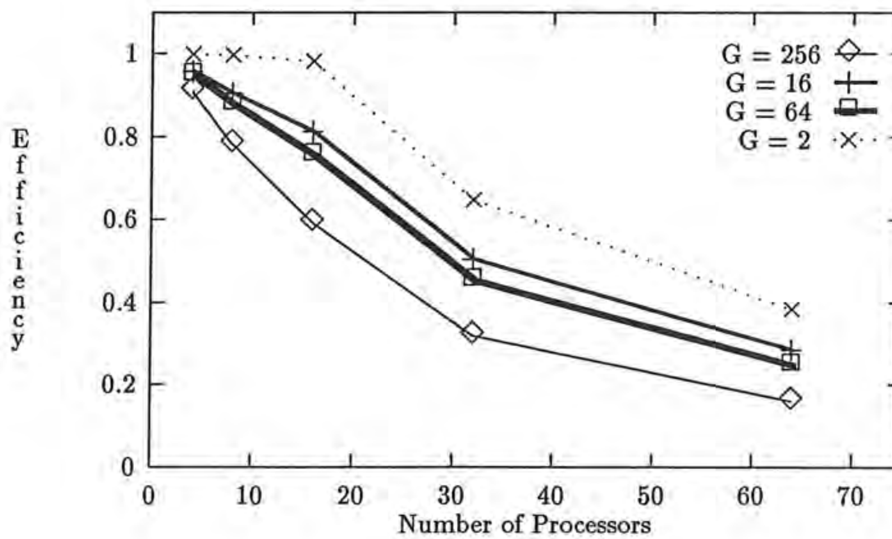

Figure 5.3: Intel2 Efficiency SM

But even with the smallest minimum grainsize, speedup and efficiency are poor. A more reasonable minimum grainsize of 256 causes performance to degrade further. Speedup tops out at 10.19 using 32 processors with an efficiency of only .32. A partial explanation of this poor performance can be obtained by considering the complexity of the individual conformal products contained in the query as shown in Figure 5.4

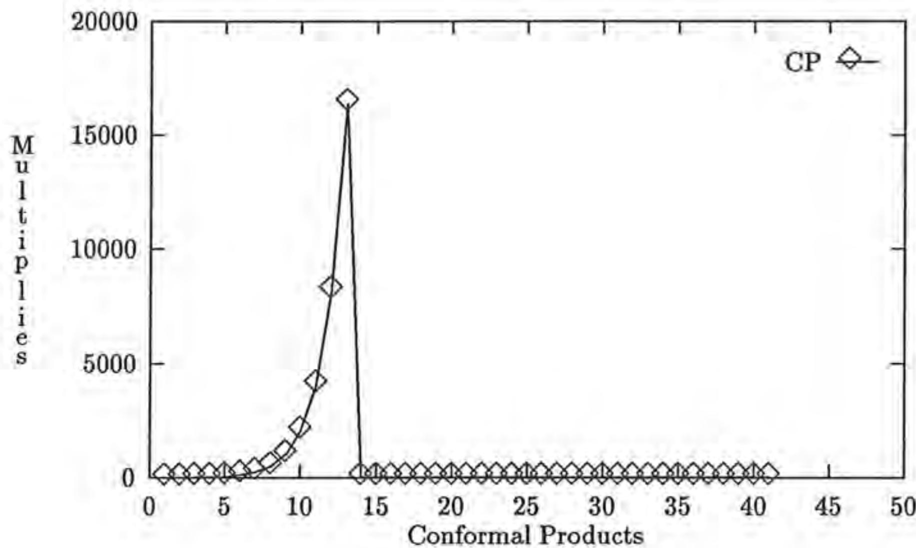

Figure 5.4: Intel2 Conformal Products - Multiplies

From Figure 5.4 is it easy to see that one conformal product dominates the query evaluation while the majority are too small to parallelize. Figure 5.5 shows the same information, but, instead of multiplies the Y axis shows dimension, i.e. log of the graph 5.4's Y axis.

A partial explanation for the predicted poor performance is the number of small conformal products that cannot be parallelized. But this is only a very small part of the explanation, since those conformal products below the minimum grainsize, i.e. with dimension less than 8, make up only about 1% of the overall work. Consider the graph of Figure 5.6, which shows how the problem would be split given a minimum grainsize of 256 and an unlimited number of processors.

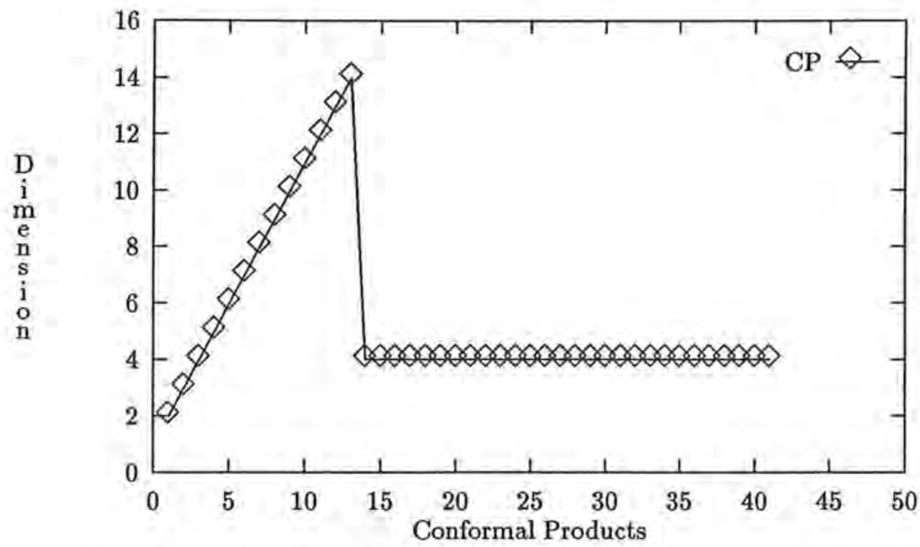

Figure 5.5: Intel2 Conformal Products - Log Multiplies

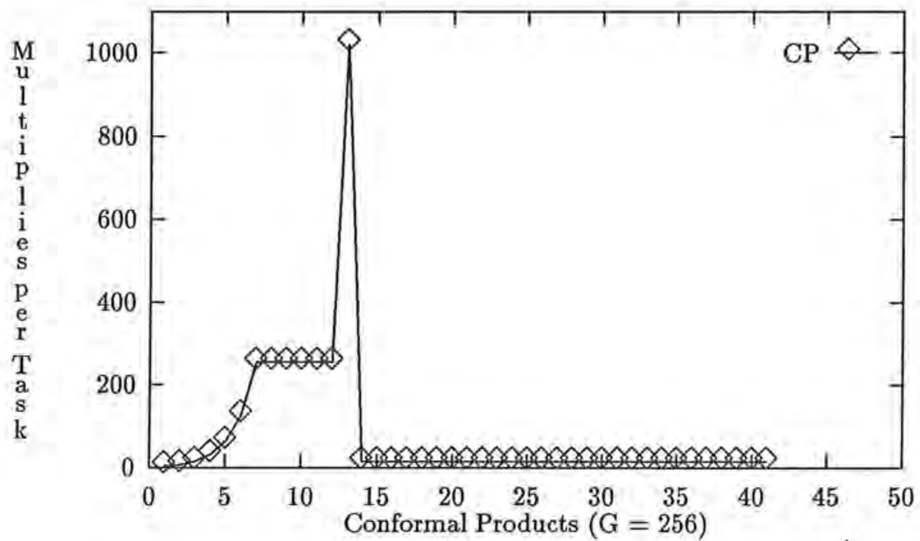

Figure 5.6: Intel2 Multiplies per Task

In Figure 5.6 the source of the poor performance can be seen. When offered unlimited processors all of the larger conformal products, except for the largest, split into tasks of size 256. Figure 5.7 shows the number of processors used by each conformal product. This represents the expected speedup for each conformal product.

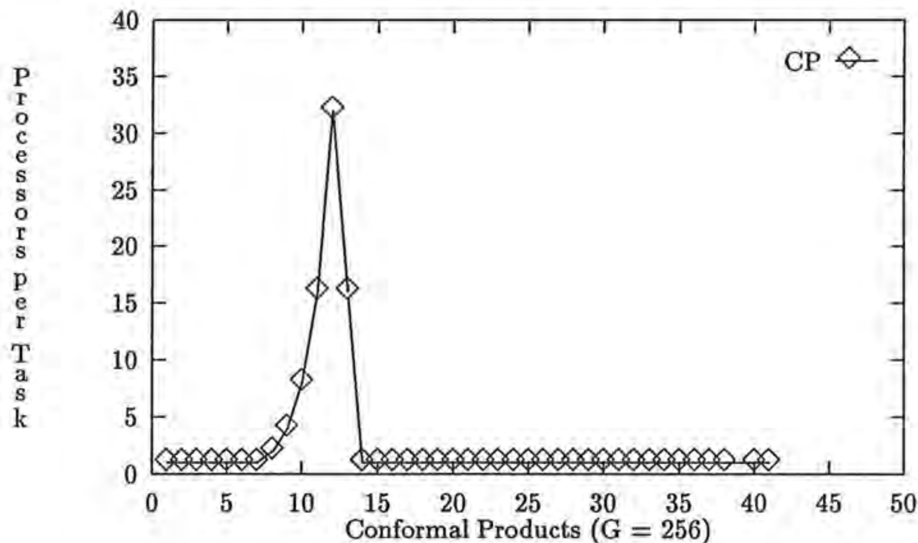

Figure 5.7: Intel2 Processors per Task

From Figure 5.7 we can see that the largest conformal product, number 13, used only 16 processors. Whereas conformal product 12, which was half as large as conformal product 13, used twice as many processors. Together Figures 5.6 and 5.7 show that conformal product 12, of dimension 14, is a processing bottleneck. Although it requires 2^{14} multiplies it only has 4 result variables, or, in other words, it can only be split into 16 subtasks.

Result: Run out of parallelism in problem. Lack of variables in the result distribution of the largest conformal product limited splitting.

Random Net 4

Figures 5.8 and 5.9 show the expected speedup and efficiency of query evaluation on random net 4. In contrast to the Intel2 net, random net 4 shows near-linear speedup for up to 32 processors when the minimum grainsize is 16, and for up to 64 processors when the minimum grainsize is 2.

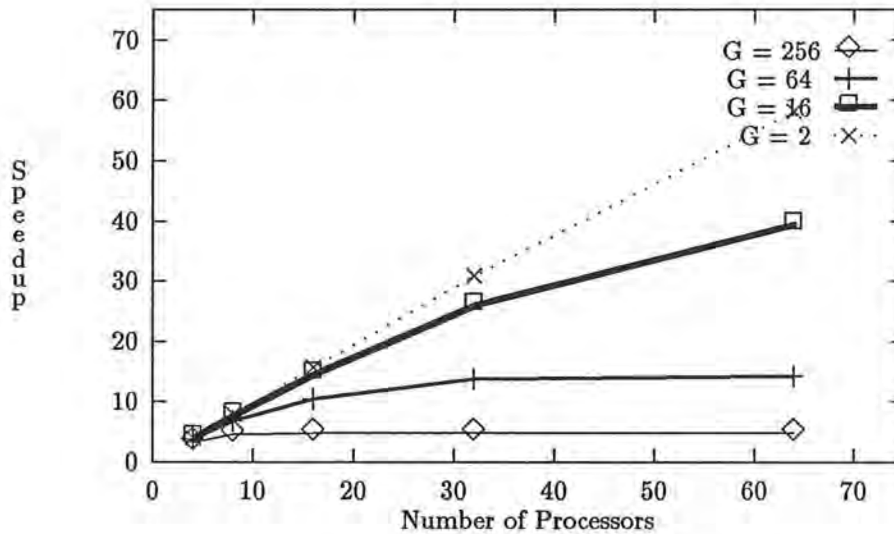

Figure 5.8: Net 4 Speedup SM

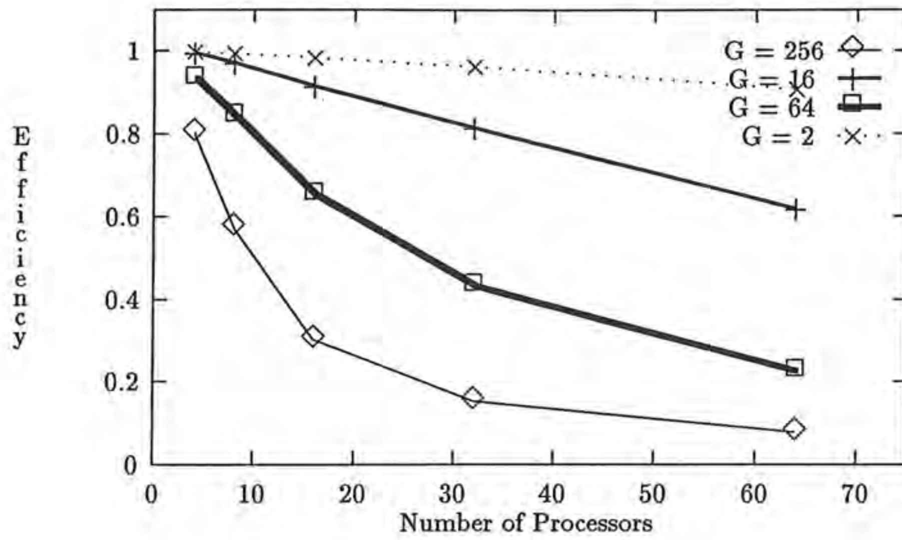

Figure 5.9: Net 4 Efficiency SM

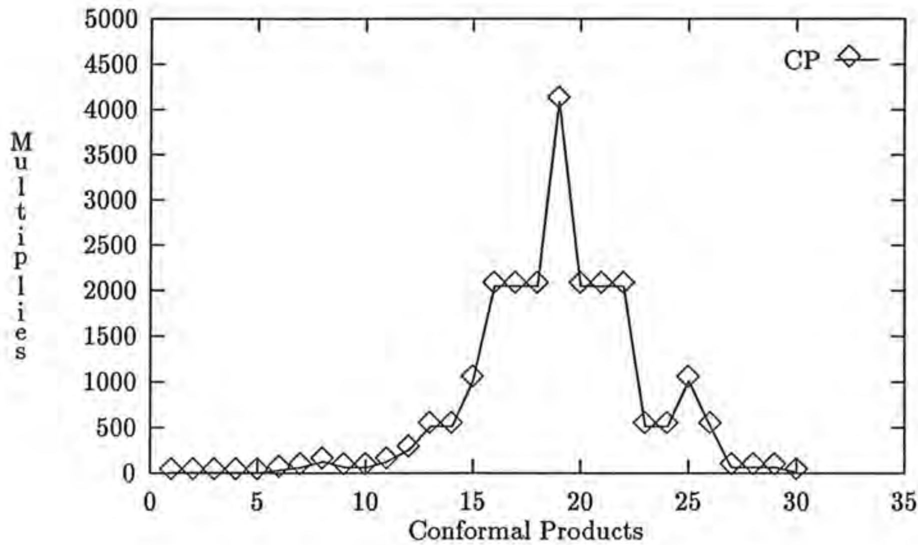

Figure 5.10: Net 4 Conformal Products - Multiplies

Figure 5.10 shows the complexity of the conformal products performed in query evaluation, and Figure 5.11 displays the same information but in terms of dimension rather than multiplies. Together these graphs, 5.10 and 5.11, suggest an explantion for the speedup and efficiency behavior. The largest conformal product is only of dimension 12, which explains the poor performance for large grainsize settings. It is only twice as large as the conformal products which come before and after it, that suggests the absence of any processing bottleneck.

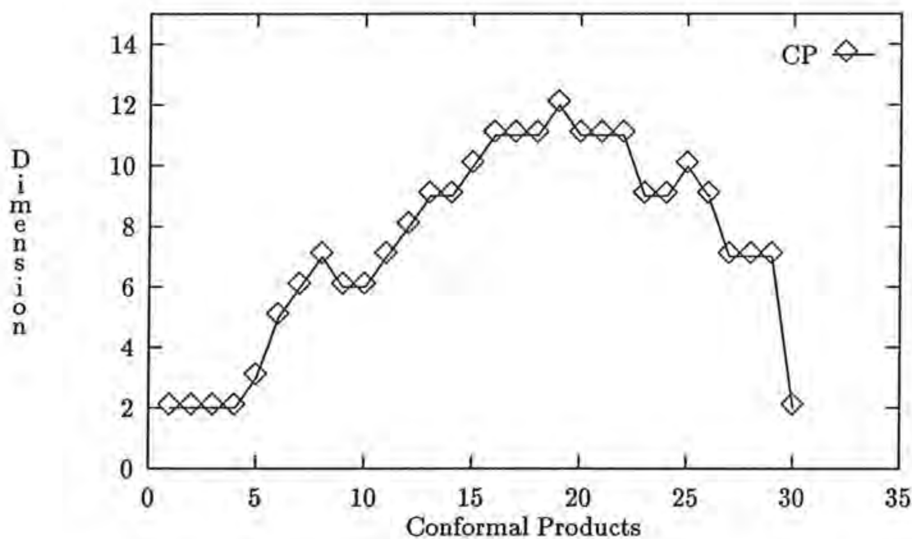

Figure 5.11: Net 4 Conformal Products - Log Multiplies

Figure 5.12 shows the possible parallelism given unlimited processors and a minimum grainsize of 256. The absence of a conformal product with more than 256 multiplies indicates that the all of the result distributions contained enough variables to support complete splitting.

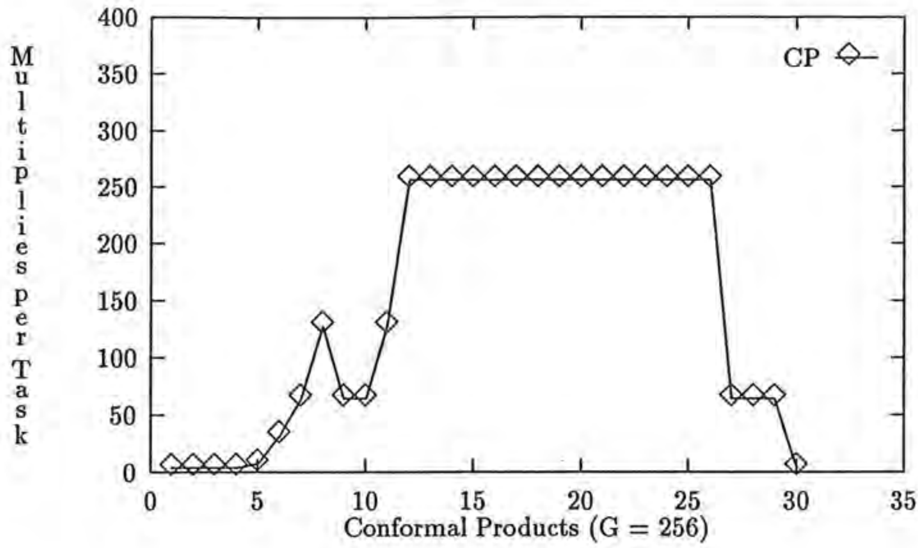

Figure 5.12: Net 4 Multiplies per Task

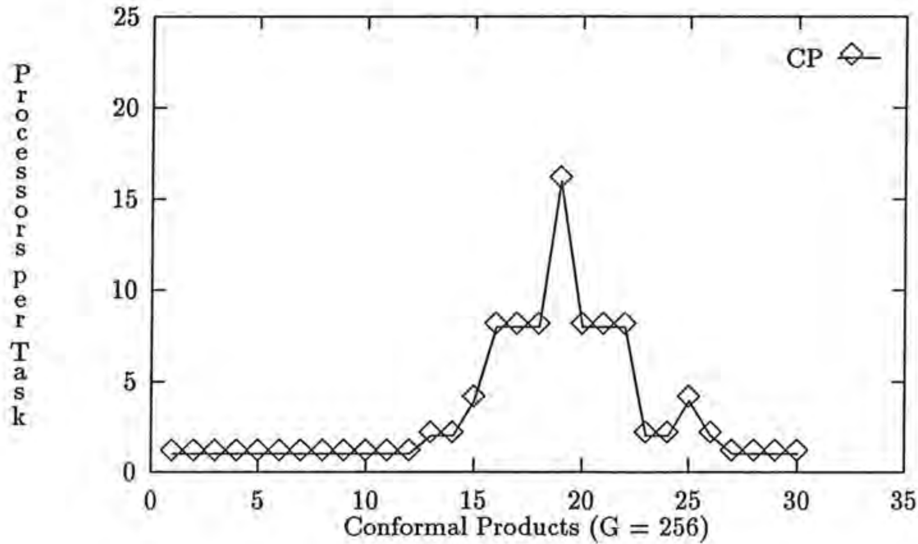

Figure 5.13: Net 4 Processors per Task

Figure 5.13 shows the number of processors that could be used by each conformal product when the minimum grainsize is 256. This figure, together with Figure 5.12, shows that the factor limiting expected speedup and efficiency is the small size of the conformal products, not the lack of splitting variables. As the minimum grainsize is lowered speedup and efficiency increase as shown in Figures 5.9 and 5.9.

Result: The number of small conformal product limited the splitting and there the speedup. The problem was to small to be parallelized effectively.

Random Net 9

As shown in Figures 5.14 and 5.15 random net 9 exhibits near-linear speedup and high efficiency for up to (at least) 256 processors. This performance is maintained with little change across grainsize settings of 2, 16, 64, and 256.

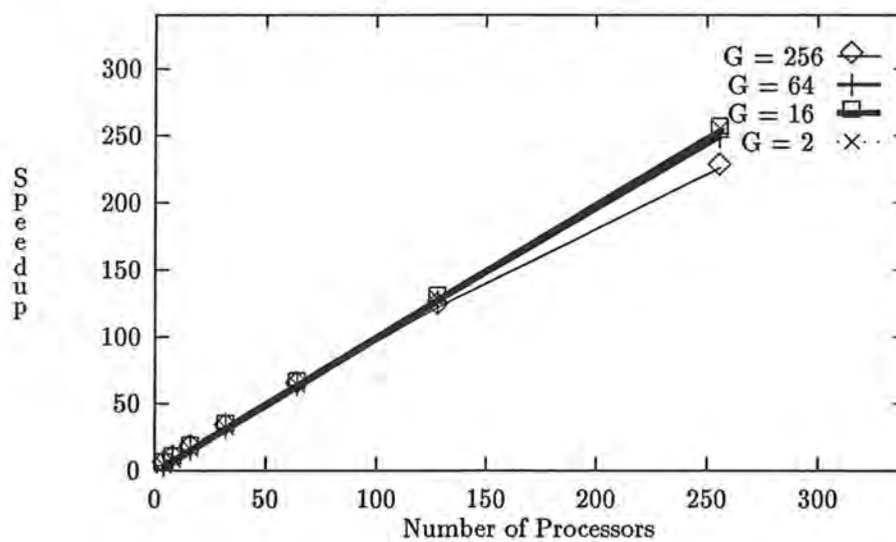

Figure 5.14: Net 9 Speedup SM

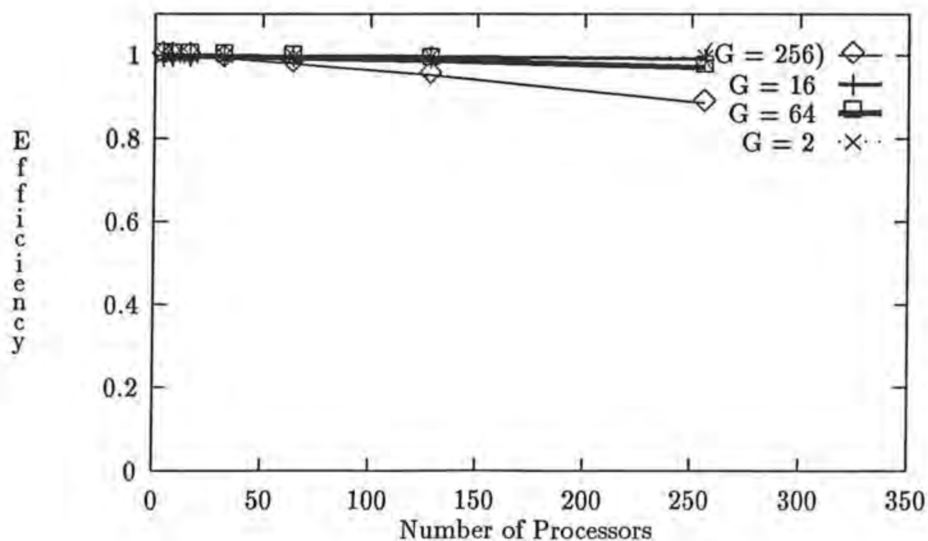

Figure 5.15: Net 9 Efficiency SM

As shown in Figure 5.16, the query evaluation for random net 9 required 41

conformal products, with complexities ranging from 4 multiplies to 2^{20}. An interesting feature of this problem is that there are two dominating conformal products, rather than one, as in the Intel2 net and the random net 4.

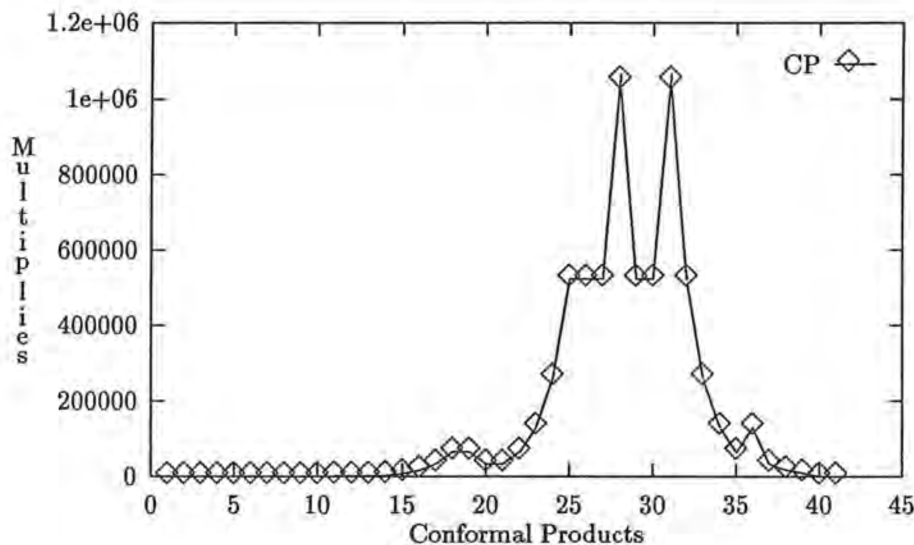

Figure 5.16: Net 9 Conformal Products - Multiplies

Figure 5.17 shows the complexity of conformal products in terms of dimension. An important point is the lack of great disparity between the complexity of neighboring conformal products. The largest single drop in relative magnitude between any 2 neighboring conformal products is from 2^{13}, conformal product 39, to 2^7, conformal product 40. With a grainsize of 256, complete splitting of conformal product 40 would require only 32 processors and 5 splitting variables in the result distribution. From Figure 5.18 we can see that all of the conformal products were either smaller than the minimum grainsize, conformal products 1 though 8 and 40 through 41, or else contained enough splitting variables to allow complete splitting.

Figures 5.18 and 5.19 show the degree of splitting possible and the number of processors used given a minimum grainsize of 256 and unlimited processors. From these graphs it is easy to see that the problem is big enough and well-behaved enough (no processing bottlenecks due to lack of splitting variables) that speedup and efficiency are limited only by the number of processors available.

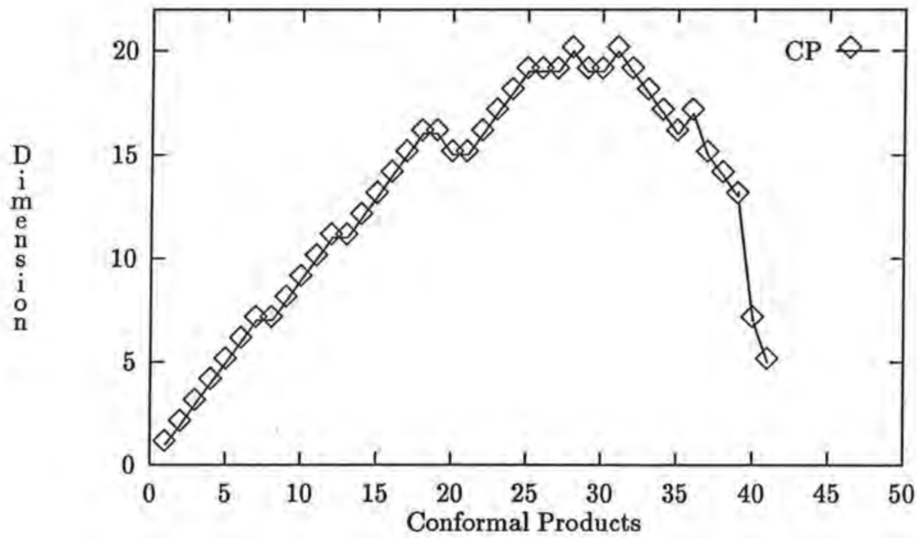

Figure 5.17: Net 9 Conformal Products - Log Multiplies

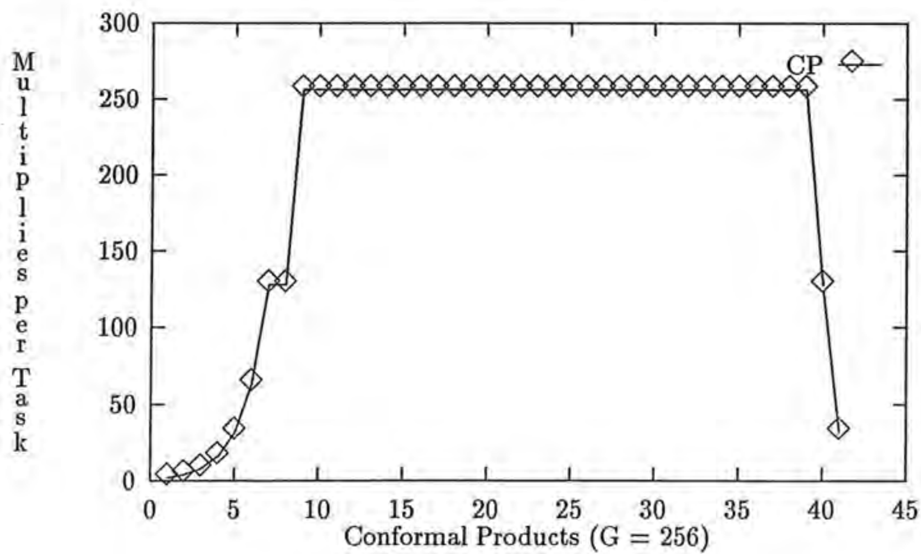

Figure 5.18: Net 9 Multiplies per Task

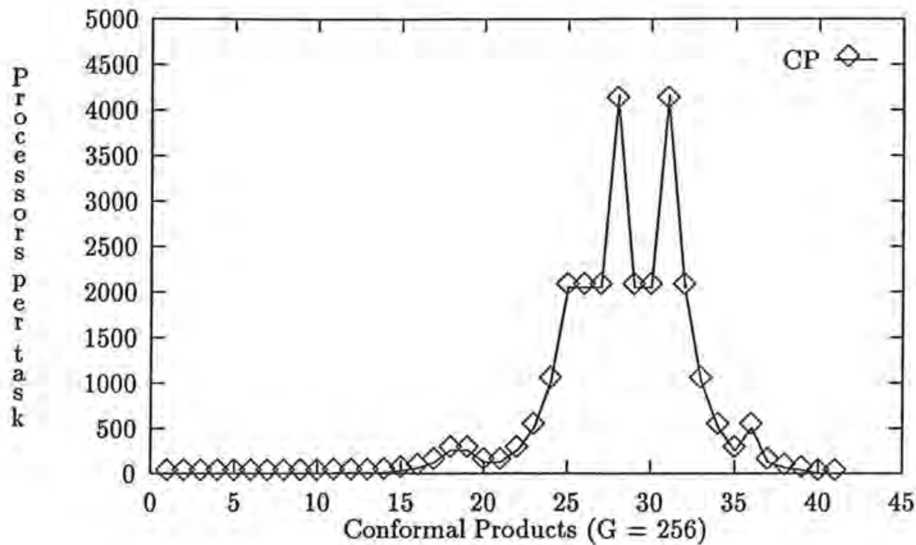

Figure 5.19: Net 9 Processors per Task

Result: Troblem is big enough and contains enough parallelism to get near-linear speedup and high efficiency. We would expect to run out of processors before exhausting the possible parallelism.

5.3.2 Distributed memory

The purpose of the distributed memory model was to determine the effect of communication costs on speedup and efficiency of parallel query evaluation. The speedup and efficiency figures of Table 5.2 suggest that parallel conformal product evaluation is communication-intensive. And the graph in Figure 5.1 shows just how dismal the predicted performance is. In the following sections we examine the results of simulations of the Intel2 net and random nets 4 and 9 under the distributed-memory model.

Intel2 Net

Figure 5.20 shows the speedup curves for the Intel2 net under minimum grainsize settings of 2, 16, 64, and 256. The number of processors ranges from 4 to 64. The first thing one notices is that speedup is poor under all conditions.

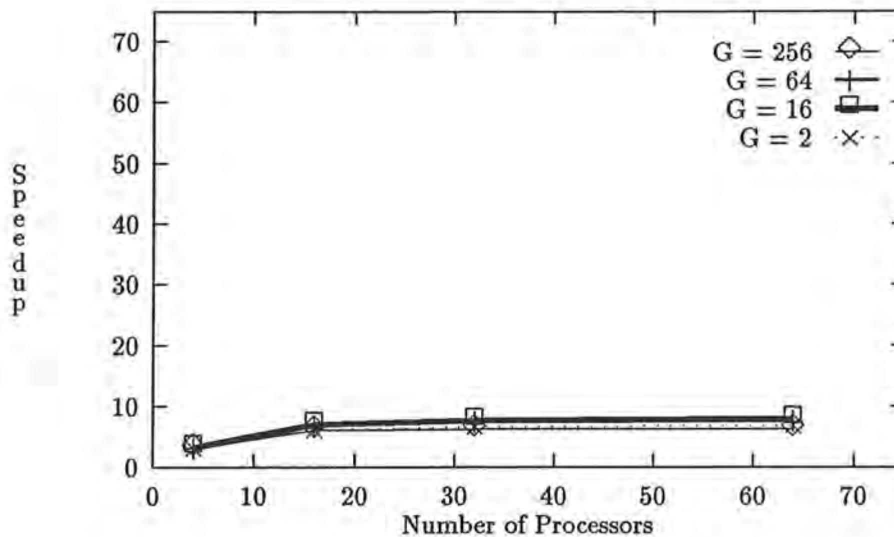

Figure 5.20: Intel2 Speedup DM

Figure 5.21 shows the efficiency curves for the same set of simulations and, as with the speedup curves, not only is the performance poor but grainsize seems to have had no effect. We know from the analysis of the shared-memory simulations that without communication costs, speedup and efficiency depend in part on the minimum grainsize. To understand the relationship between communication costs, grainsize settings, and speedup and efficiency consider Figure 5.22.

Figure 5.22 shows the relationship between time and number of processors for a simulation of query evaluation in the Intel2 net with a minimum grainsize of 2. There are 3 measures of time: total time, compute time, and communication time. The general trend displayed in the graph is that with few processors there is little splitting so communication time is low and compute time is high. As the number of processors increases, more splitting is possible, so the compute time decreases and the communication time increases. The explanation for the reduction in compute time is that with more processors, the amount of work performed by any one processor is reduced. The explanation for the increase in communication time is that with more processors the overall amount of data sent out increases, as discussed in Chapter 3, and some of this data has to travel farther (across additional

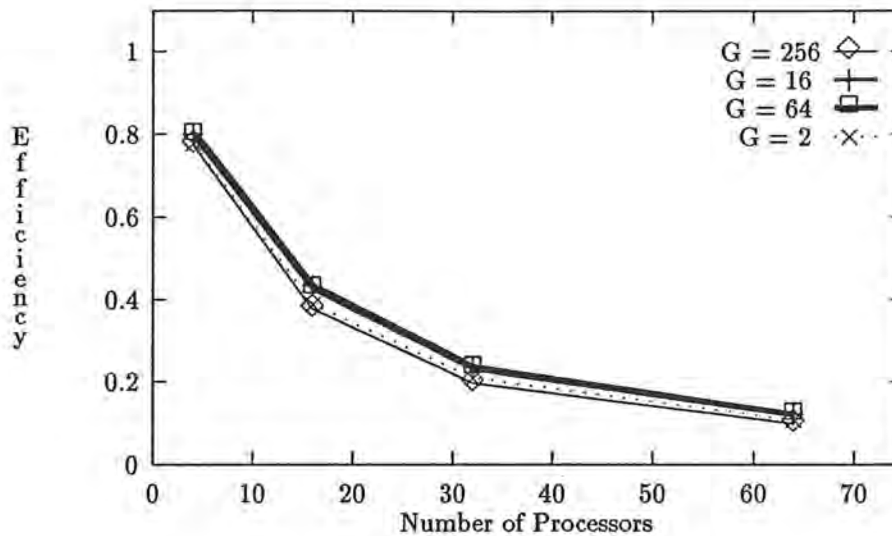

Figure 5.21: Intel2 Efficiency DM

dimensions of the cube). The general result is that the compute time saved by splitting the problem is consumed in the communication time. Figure 5.23 show the effects of increasing the minimum grainsize from 2 to 64.

In Figure 5.23 we see that for up to 32 processors, with a minimum grainsize of 64, the compute time is greater than the communication time. At 32 processors compute time and communication time are approximately equal. With more than 32 processors compute time drops slightly below communication time. The trend is similar to that shown in Figure 5.22. Compute time starts high and drops quickly, while communication time starts low and climbs slowly. Both compute time and communication time level off quickly. As a final look at the relationship between compute and communicate time with the Intel2 net, consider the graph of Figure 5.24 in which the minimum grainsize is 256.

In Figure 5.24 we see the same general trend as in Figures 5.23 and 5.22. The difference is that since the grainsize is 256, less splitting occurs and thus the compute time stays higher than in the previous simulations and the communication time stays low. Whereas in the simulations with smaller grainsizes, the communication costs eventually became greater than the compute costs, in this case the two costs never

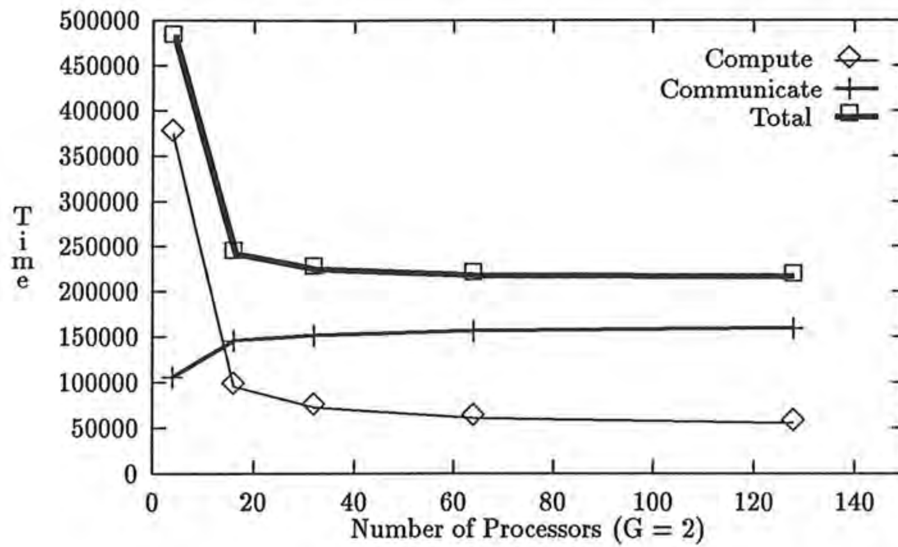

Figure 5.22: Intel2 Compute - Communicate DM G = 2

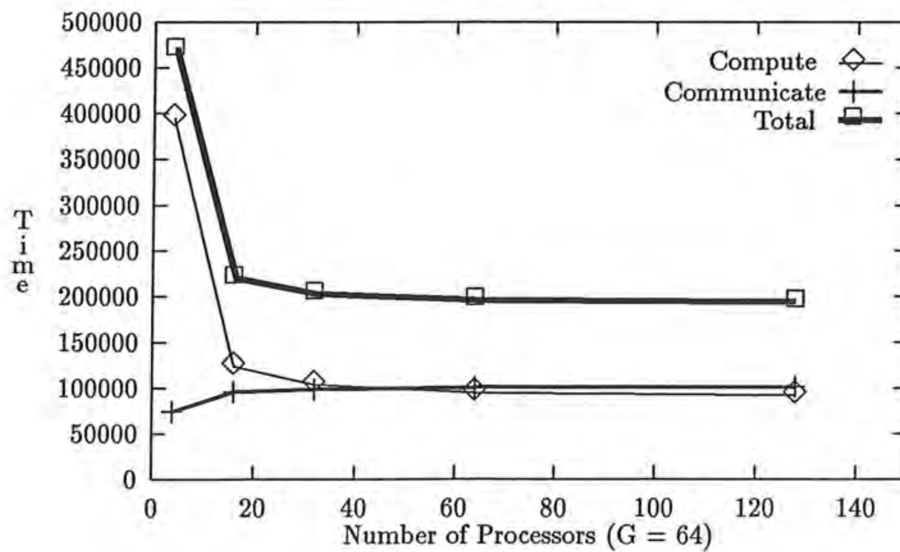

Figure 5.23: Intel2 Compute - Communicate DM G = 64

become close.

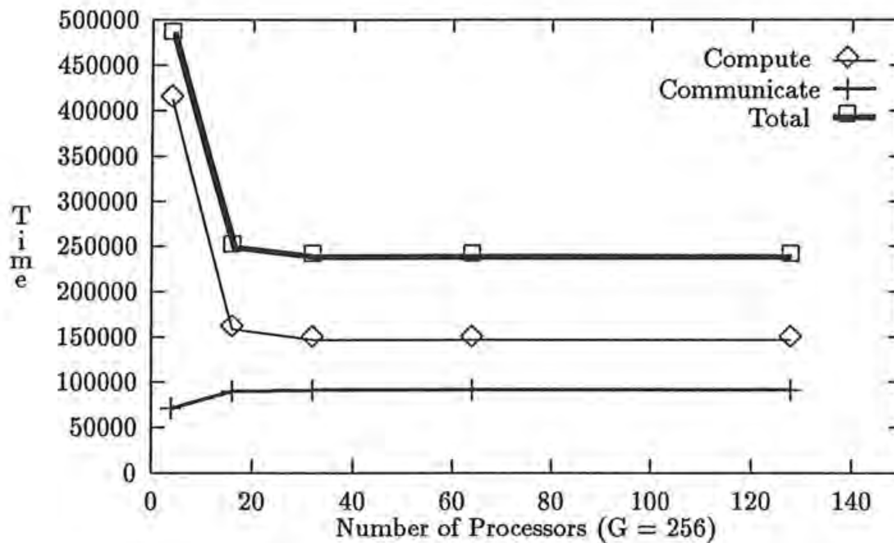

Figure 5.24: Intel2 Compute - Communicate DM G = 256

Result: With more than 16 processors, the additional cost for communication consumes any reduction in compute time.

Random Net 4

Figure 5.25 shows the speedup curves for parallel query evaluation on random net 4 with minimum grainsize settings of 2, 16, 64, and 256, with the number of processors ranging from 4 to 32. As with the Intel2 net, speedup levels off quickly. One difference between the speedup graphs of random net 4 and the Intel2 net is the distinctly poor performance on the random net 4 when the grainsize was set at 256. This performance is also reflected in the efficiency graph of Figure 5.26.

The poor speedup and efficiency performance at grainsize 256 can be explained by the small sizes of the conformal products in the query, as discussed in the section on random net 4 under shared-memory simulations. As with the Intel2 net, the relationships between compute time and communication time for random net 4 are given for minimum grainsize settings of 2, 64, and 256, in Figures 5.27, 5.28, and 5.29, respectively.

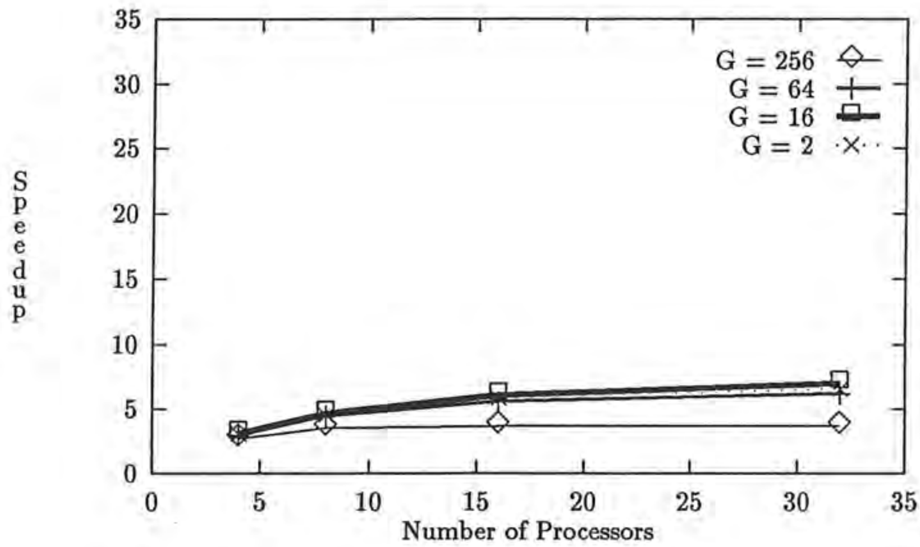

Figure 5.25: Net 4 Speedup DM

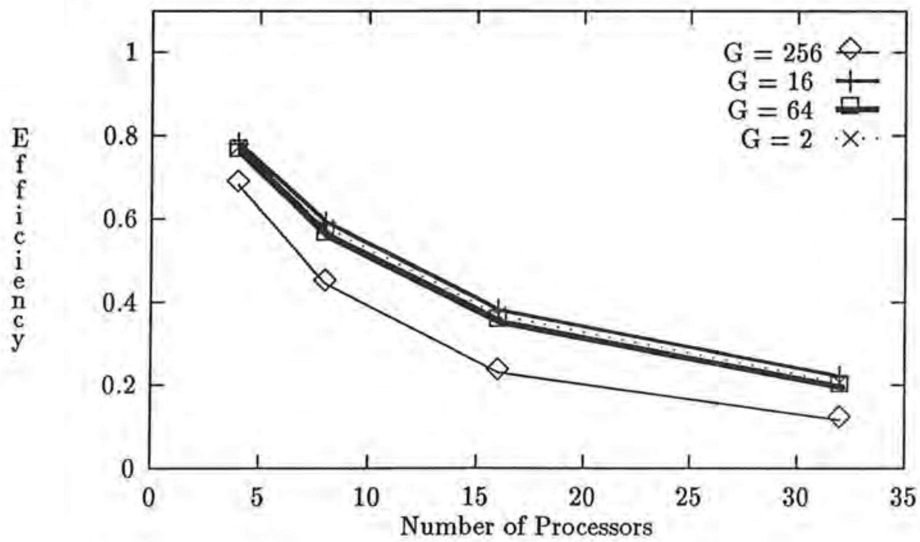

Figure 5.26: Net 4 Efficiency DM

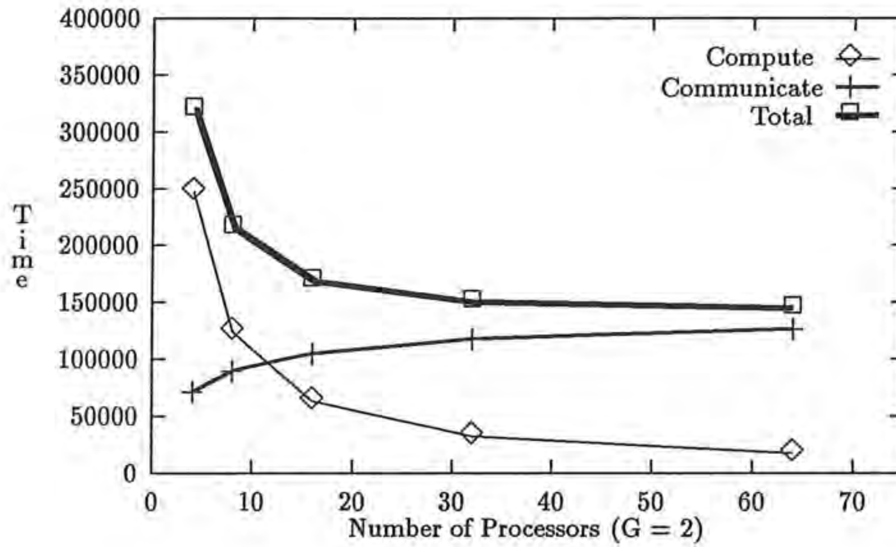

Figure 5.27: Net 4 Compute - Communicate DM G = 2

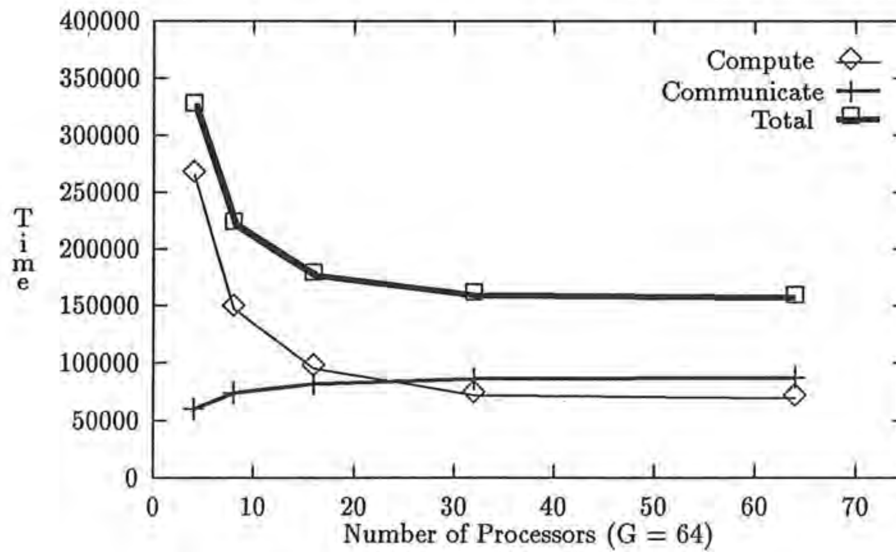

Figure 5.28: Net 4 Compute - Communicate DM G = 64

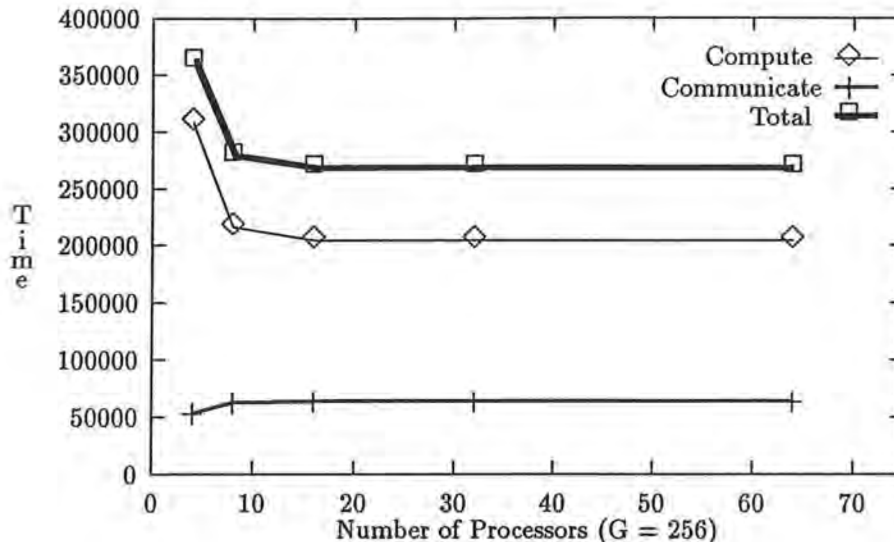

Figure 5.29: Net 4 Compute - Communicate DM G = 256

For random net 4 the relationships between compute and communication times are similar to those of the Intel2 net. With a small number of processors compute time is high and communication time is low. As the number of processors increases compute time decreases and communication time increases, until at some point the maximum possible splitting occurs and the time measures level out.

Random Net 9

Figure 5.30 shows the speedup curves for parallel query evaluation on random net 9 with minimum grainsize settings of 2, 16, 64, and 256. The number of processors ranges from 4 to 256. A comparison of this graph with the speedup graph of this net under the shared-memory model (Figure 5.14) shows the effect of communication costs. The shared-memory model predicted near-linear speedup. Under the distributed-memory model speedup tops out at about 16. The efficiency graph in Figure 5.31 also reflects this poor performance.

As with the Intel2 net and the random 4 net, the compute and communication times are given for grainsize settings of 2, 64, and 256. Figures 5.32, 5.33, and 5.34 present this information.

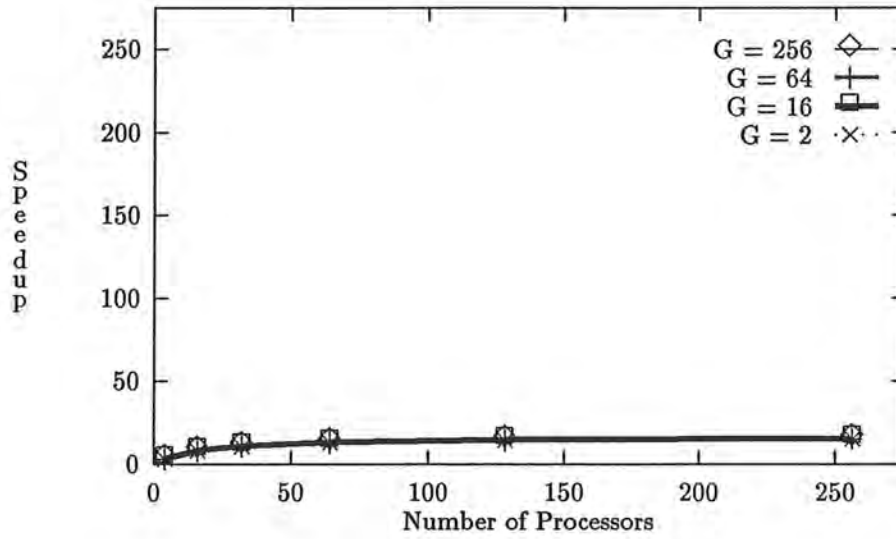

Figure 5.30: Net 9 Speedup DM

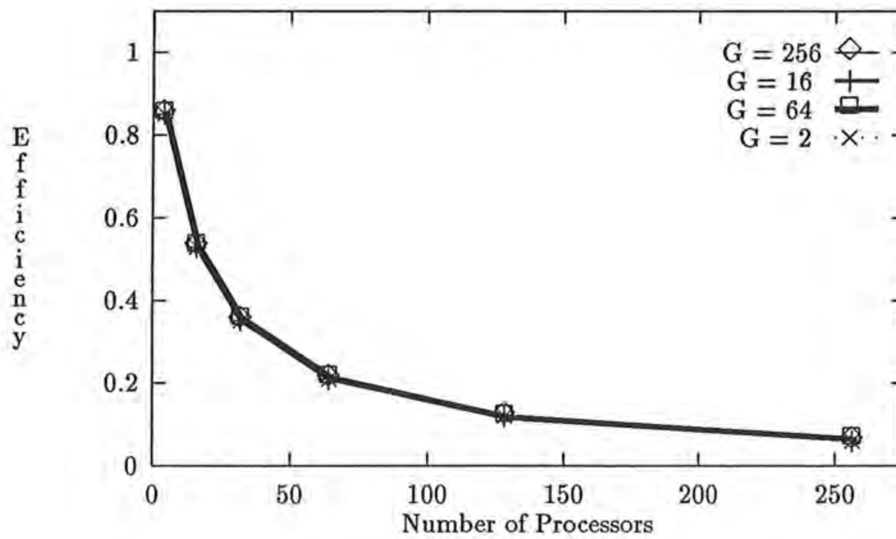

Figure 5.31: Net 9 Efficiency DM

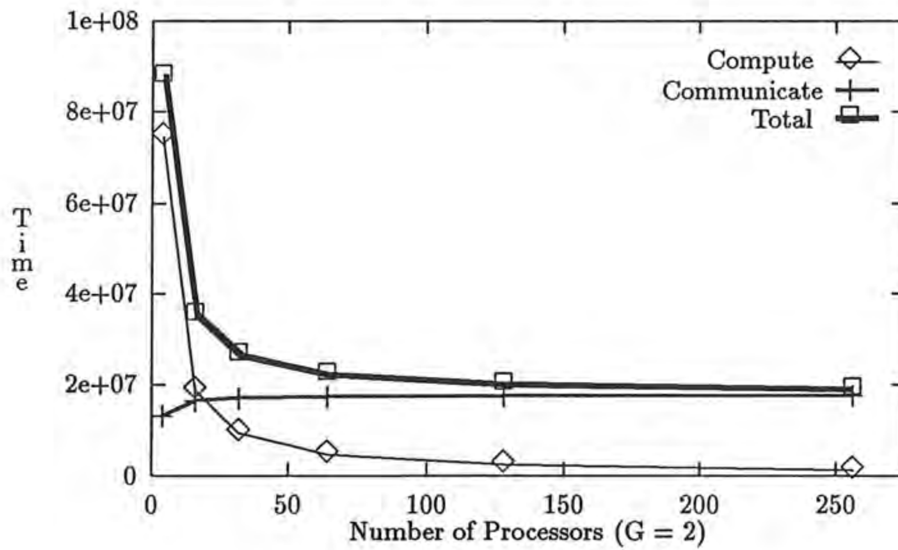

Figure 5.32: Net 9 Compute - Communicate DM G = 2

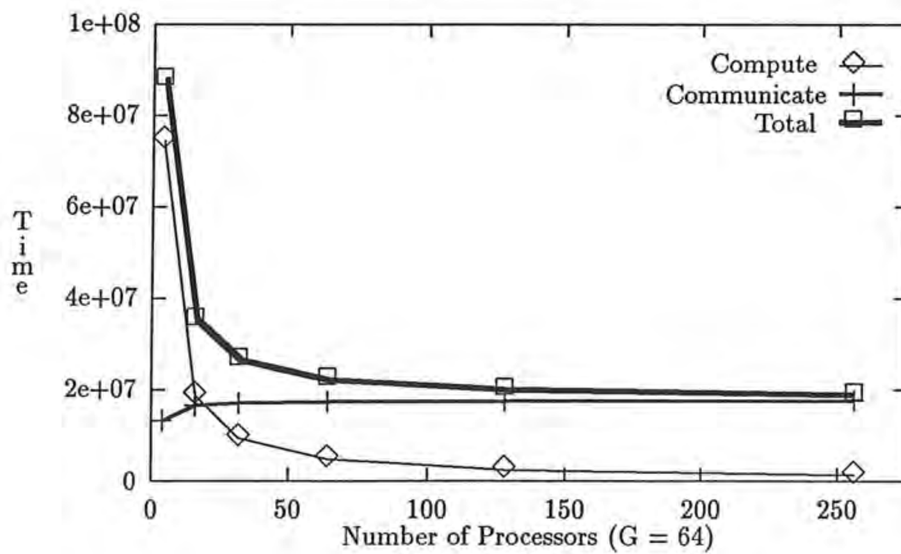

Figure 5.33: Net 9 Compute - Communicate DM G = 64

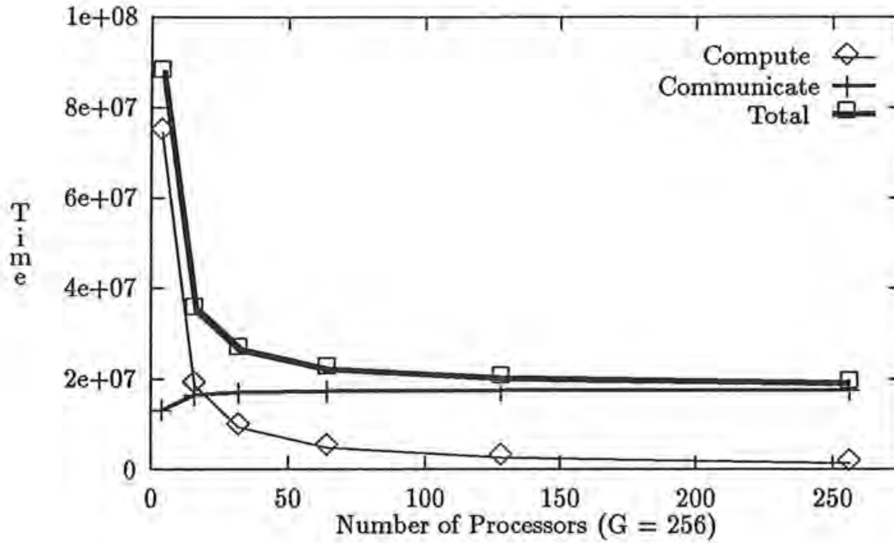

Figure 5.34: Net 9 Compute - Communicate DM G = 256

Figures 5.32, 5.33, and 5.34 all look the identical. The conclusion to draw from this is that for this net, within the bounds explored, grainsize is not an important issue.

Result: In order to understand the results for random net 9 under the distributed-memory model, it is important to remember the high complexity of the query evaluation. As described in the shared-memory section, many of the conformal products in this query are of sufficient size to split across 256 processors with a minimum grainsize of 256. This explains why speedup was near-linear for this query with 256 processors and a minimum grainsize of 256 under the shared-memory model. This also explains why the compute-communicate cost graphs are so similar for the three grainsizes 2, 64, 256.

5.3.3 Evaluation Tree Parallelism

To investigate the potential parallelism available at the evaluation tree level we calculated the longest path in the evaluation tree and summarized this information in Table 5.3.

In Table 5.3 the entries in the Number of CPs column is the number of con-

Net	Number of CPs	Longest Path	% of Time
1	39	27	.999
2	21	13	.884
3	51	44	.999
4	30	27	.999
5	32	30	.999
6	30	19	.973
7	50	43	.999
8	88	56	.999
9	41	41	1.00
10	36	31	.999
Intel2	41	41	1.00

Table 5.3: Evaluation Tree Parallelism

formal products in the evaluation of the query. Entries in the Longest Path column represent the number of conformal products in the longest path. Entries in the % of Time column are the percentages of the overall time for query evaluation spent in evaluating the conformal products in the longest path. The time to compute the longest path consumes almost the entire query evaluation time. This means that those conformal products that were not in the longest path were of low complexity. The conclusion to be drawn from this data is that there is no evaluation tree parallelism to be exploited in the evaluation of these queries. But, as mentioned in Chapter 2, evaluation trees are not unique, and therefore another topic for future research is to explore the possiblity of generating more balanced and bushy evaluation trees.

Chapter 6

Conclusions

For this project we investigated two sources of parallelism in the query evaluation process of the SPI algorithm: evaluation tree parallelism and conformal product parallelism. Rather than implementing parallel algorithms or performing a purely analytic analysis we chose instead to modify an existing sequential program to simulate parallel query evaluation. This simulation allowed us to control various features of the computation, in particular subtask decomposition and communication models. It thus provided a method for exploring the effects of these features on the time complexity of query evaluation. The work described in this paper represents only a few sample points from the space of explorations possible using this approach. These points were chosen to give us information about two aspects of the problem, the amount of parallelism in query evaluation, in particular in the intermediate computations, and the relative costs of communications. The shared-memory model addresses the first of these issues, the distributed-memory model the second.

6.1 Shared Memory

The results from Chapter 5 suggest that if conformal products are big enough and contain enough result variables, then reasonable speedup and efficiency can be achieved through conformal product parallelism. The results show that parallelism is limited when the expanded queries contain a large proportion of small conformal products, as demonstrated by random net 4, or when one or more of the large conformal products contains few result variables, as demonstrated by the Intel2

net. These results should provide some direction for future work on query expansion methods. Of course, since communication and setup costs were ignored these results are upper-bounds on speedup and efficiency. The effects of including communication costs in performance measures was the subject of the distributed-memory model simulations. Setup costs and their effects on parallelism provide a topic for future research, as described below.

6.2 Distributed Memory

For the distributed-memory model the results from Chapter 5 show that communication costs dominate and make the spanning tree distribution approach infeasible. Our hypothesis was that as conformal product complexity increased, the compute time would decrease faster than communication time would increase. This turned out to be true, and explains the slight speedup observed. But the effect was smaller than expected. To understand why recall from Chapter 4 the formula for calculating the amount of data sent out.

$$B_{total} = 4 * (2^{max(|dist1-vars|,|splitting-vars|)} + 2^{max(|dist2-vars|,|splitting-vars|)})$$

Recall also that the overall compute-time complexity of the conformal product operation is exponential in the number of unique variables in the two input distributions. Let $all - vars$ represent the set of unique variables, i.e., $all - vars = dist1 - vars \bigcup dist2 - vars$. Since $dist1 - vars$, $dist2 - vars$, and $splitting - vars$ are all subsets of all-vars we know that $B_{total} < 4 * 2^{|all-vars|}$. With respect to communication costs a best case situation would be one in which |dist1-vars| = |dist2-vars| = |all-vars|/2, and $dist1 - vars \bigcap dist2 - vars = \phi$, and |splitting-vars| $\leq$ |dist1-vars|. In other words, if the variables in $all - vars$ are split evenly across the two input distributions and splitting occurs at most on half the variables, then the amount of data that must be sent out is exponential in |all-vars| / 2. As it turned out this best case never occurred. In every case |all-vars| - (max |dist1-vars|, |dist2-vars|) = 1. In other words, for all of the test nets the size of one of the input distributions was one variable smaller than the number of variables in

the full joint distribution. Therefore compute time is exponential in the number of $all-vars$ and communication time is exponential in the number of $all-vars$ minus 1. The reason why the distributed-memory simulations showed any speedup at all was the lower constant associated with communication, .5 microseconds per byte, as compared with the constant associated with multiplication, 45 microseconds per multiply. One possible topic for future research is to determine if this asymmetrical distribution of variables is an inherent part of the problem or simply an artifact of the query expansion process.

6.3 Evaluation Tree Parallelism

Given the shape of the evaluation trees generated by the current query expansion procedure, there is no evaluation tree parallelism to exploit. A partial solution to the lack of evaluation tree parallelism is bushier trees, and one topic of future research is alternative methods for query expansion.

6.4 Further research

This work represents a first look at parallelism in the SPI algorithm. Among the many issues that deserve further attention are the following.

- Splitting the conformal product calculation on only those variables that occur in the result distribution has been shown to produce processing bottlenecks for some queries. The alternative is to split on all variables, as long as the minimum grainsize constraint is satisfied, and then sum the results together. This approach would increase parallelism but complicate the data dependencies among subtasks. It is not clear at this time whether the increased parallelism would compensate for the increased complexity of communication that this strategy requires.

- Explore the ignored parameters of the model, in particular S, the setup time. Except for communication, the set-up time includes the time required for all

operations that take place between the time a query is received and the time at which subtasks begin computing their respective portions of the result distribution. This includes the time to construct the expanded query and the time required to put together command packages for the subtasks. Since we have not considered ways to parallelize these operations, it is only fair to assume that they are sequential. As pointed out in [9] and [4], the amount of sequential code limits the potential speedup according to Amdahl's law. We believe that this cost is small relative to the costs of calculating the conformal product, but a more complete analysis would investigate this issue more rigorously.

- The belief nets that were used in this project all had a uniform value space of dimension 2. Since it seems likely that many real-world nets will have variables with greater dimension and variability it is important to explore the effects that these changes have on possible parallelism.

- Since communication is so expensive in the distributed-memory model any followup work to this project should investigate alternative data distribution schemes. It appears that any approach to parallel conformal product evaluation in which one processor is responsible for dividing up the problem, sending out the data associated with subtasks, and assembling the results together will be communication-intensive. A different approach would be to partition the belief net across the processors in such a fashion that data is local to the processors whose computation requires it.

- The results presented in Chapter 5 are all sensitive to the structure of the expanded query, or the evaluation tree. As mentioned in Chapter 2, evaluation trees are not unique and the generation of efficient evaluation trees is a topic of current research. For the tests reported on in this paper, all evaluation trees were generated by a procedure oriented towards efficient sequential evaluation. The primary heuristic of that procedure is one to keep the dimension of the intermediate result distributions as low as possible, thus minimizing the

number of necessary multiplies. It seems like this strategy for generating evaluation trees would also benefit the parallel evaluation process, but we do not know for certain at this time that there is not an approach which maintains low dimensionality and yet produces evaluation trees which are parallelizable.

Bibliography

[1] Selim G. Akl, "The Design and Analysis of Parallel Algorithms", Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[2] Bruce D'Ambrosio, Symbolic Probabilistic Inference in Belief Nets, OSU Dept. of Computer Science Tech Report 90-30-1.

[3] Bruce D'Ambrosio, Personal communication.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, "Solving Problems On Concurrent Processors", Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[5] Zhaoyu Li, Complexity of Probabilistic Inference in Belief Nets – An Experimental Study, OSU Masters Thesis, 1990.

[6] Robert Oliver and James Smith, "Influence Diagrams, Belief Nets and Decision Analysis", John Wiley and Sons, New York, 1990.

[7] Judea Pearl, "Probabilistic Reasoning in Intelligent Systems", Morgan Kaufmann, San Mateo, CA, 1988.

[8] Ross D. Shachter, Bruce D'Ambrosio, and Brendan Del Favero, Symbolic Probabilistic Inference: A Probabilistic Perspective.

[9] Michael Quinn, "Designing Efficient Algorithms For Parallel Computers", McGraw-Hill, New York, 1987.