Impact of Data Distribution, Level of Parallelism, and Communication Frequency on Parallel Data Cube Construction[*]

Ge Yang
Department of Computer and Information Sciences
Ohio State University,
Columbus OH 43210
yangg@cis.ohio-state.edu

Ruoming Jin
Department of Computer and Information Sciences
Ohio State University,
Columbus OH 43210
jinr@cis.ohio-state.edu

Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University,
Columbus OH 43210
agrawal@cis.ohio-state.edu

ABSTRACT

Data cube construction is a commonly used operation in data warehouses. Because of the volume of data that is stored and analyzed in a data warehouse and the amount of computation involved in data cube construction, it is natural to consider parallel machines for this operation.

We have developed a set of parallel algorithms for data cube construction using a new data structure called *aggregation tree*. Our experience has shown that a number of performance trade-offs arise in developing a parallel data cube implementation. In this paper, we focus on three important issues, which are: 1) *Data distribution*, i.e., how the original array is distributed among the processors, 2) *Level of parallelism*, i.e., what parts of the computation are parallelized and sequentialized, and 3) *Frequency of communication*, i.e., does the implementation require frequent interprocessor communication (and less memory) or less frequent communication (and more memory).

We present a detailed experimental study evaluating the above trade-offs. We consider parallel data cube construction with different cube sizes and sparsity levels. Our experimental results show the following: 1) In all cases, reducing the frequency of communication and using higher memory gave better performance, though the difference was relatively small. 2) Choosing data distribution to minimize communication volume made a substantial difference in the performance in most of the cases. 3) Finally, using parallelism at all levels gave better performance, even though it increases the total communication volume.

1. INTRODUCTION

Analysis on large datasets is increasingly guiding business decisions. Retail chains, insurance companies, and telecommunication companies are some of the examples of organizations that have created very large datasets for their decision support systems. A system storing and managing such datasets is typically referred to as a data warehouse and the analysis performed is referred to as On Line Analytical Processing (OLAP) [1].

Computing multiple related group-bys and aggregates is one of the core operations in OLAP applications [1]. Jim Gray has proposed the *cube* operator, which computes group-by aggregations over all possible subsets of the specified dimensions [6]. When datasets are stored as (possibly sparse) arrays, data cube construction involves computing aggregates for all values across all possible subsets of dimensions. If the original (or *initial*) dataset is an n-dimensional array, the data cube includes C^n_m m-dimensional arrays, for $0 \leq m \leq n$. Developing sequential algorithms for constructing data cubes is a well-studied problem [9, 8, 10, 12].

Data cube construction is a compute and data intensive problem. Therefore, it is natural to use parallel computers for data cube construction. There is only a limited body of work on parallel data cube construction [3, 4, 5].

We have developed novel sequential and parallel algorithms for data cube construction, based upon the notion of *aggregation tree*. The use of aggregation tree bounds the total memory requirements for storing intermediate results. In addition, our algorithms meet the important requirements of a data cube construction algorithm, i.e., computing each node from its minimal parent, and ensuring high cache and memory reuse.

However, our work has shown that a number of other factors have a major impact on the performance. The three important factors are, 1) *Data distribution*, i.e., how the original array is distributed among the processors, 2) *Level of parallelism*, i.e., what parts of the computation are parallelized and sequentialized, and 3) *Frequency of communication*, i.e., does the implementation require frequent interprocessor communication (and less memory) or less frequent communication (and more memory).

In this paper, we focus on these three issues and how they impact the performance of parallel data cube construction. Particularly, we examine:

Impact of Data Distribution: We have developed a closed form expression for the total communication volume required by our original parallel algorithm. This expression establishes the relationship between the distribution of the original multidimensional array and the total communication volume. We conduct a detailed experimental study to evaluate the impact of choosing data distribution to minimize communication volume on the overall execution time.

Impact of Level of Parallelism: Our first algorithm sequentializes part of the computation, i.e., if the computation of an array from its parent involves reduction along a dimension that is not distributed, this computation is not done in parallel. We present a variation of this algorithm, which performs a redistribution of the parent array in such cases, and therefore, exploits parallelism at all levels in cube construction. The new algorithm, however, involves higher

[*]This work was supported by NSF grant ACR-9982087, NSF CAREER award ACR-9733520, and NSF grant ACR-0130437.

communication volume and frequency. We present an experimental comparison of the two algorithms.

Impact of Communication Frequency: Our first two algorithms require frequent interprocessor communication. This helps reduce the memory requirements. We have developed a variation of both of our algorithms to require at most n phases of communication, where n is the number of dimensions of the original array. We evaluate the impact of this variation on the performance.

Our experimental results demonstrate the following. 1) In all cases, reducing the frequency of communication and using higher memory gave better performance, though the difference was relatively small. 2) Choosing data distribution to minimize communication volume made a substantial difference in the performance in most of the cases. This difference became more significant with increasing sparsity level in the dataset. 3) Finally, using parallelism at all levels gave better performance, even though it increases the total communication volume. Again, the difference is more significant as the sparsity level in the dataset increases.

The rest of the paper is organized as follows. We further discuss the data cube construction problem in Section 2. Our aggregation tree is presented in Section 3. Our original parallel data cube construction algorithm, its variations for changing the level of parallelism and communication frequency, and quantitative analysis for communication volume are presented in Section 4. Our detailed experimental study is presented in Section 5. We compare our work with related efforts in Section 6 and conclude in Section 7.

2. DATA CUBE CONSTRUCTION

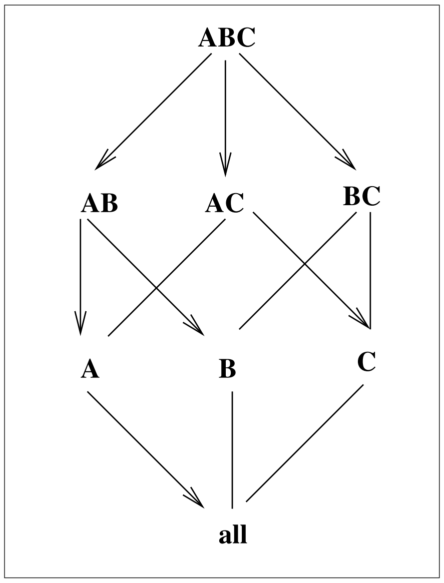

Figure 1: Lattice for data cube construction. Edges with arrows show the minimal spanning tree when $|A| \le |B| \le |C|$

This section further elaborates the issues and challenges in data cube construction. Before that, we also give some general motivation for data cube construction.

Organizations often find it convenient to express facts as elements of a (possibly sparse) multidimensional array. For example, a retail chain may store sales information using a three-dimensional dataset, with item, branch, and time being the three dimensions. An element of the array depicts the quantity of the particular item sold, at the particular branch, and during the particular time-period.

In data warehouses, typical queries can be viewed as *group-by* operations on a multidimensional dataset. For example, a user may be interested in finding sales of a particular item at a particular branch over a long duration of time, or all sales of all items at all branches for a given time-period. The former involves performing an aggregation along the time dimension, whereas the latter involves aggregations along the item and the branch dimensions.

To provide fast response to the users, a data warehouse computes aggregated values for all combinations of values. If the original dataset is n dimensional, this implies computing and storing ${}^nC_m$ m-dimensional arrays, for $0 \le m \le n$. ${}^nC_m$ is the standard combinatorics function, which is defined as

$${}^nC_m = \frac{n \times (n-1) \times \ldots \times (n-m+1)}{m \times (m-1) \times \ldots \times 1}$$

For simplicity, assume that the original dataset is three-dimensional. Let the three dimensions be A, B, and C. The sizes along these dimensions are $|A|$, $|B|$, $|C|$, respectively. Without loss of generality, we assume that $|A| \le |B| \le |C|$. We denote the original array by ABC. Then, data cube construction involves computing arrays AB, BC, AC, A, B, C, and a scalar value *all*. As an example, the array AB has the size $|A| \times |B|$.

Some of the major issues in data cube construction are as follows.

Cache and Memory Reuse: Consider the computation of AB, AC, and BC. These three arrays need to be computed from the initial array ABC. When the array ABC is disk-resident, performance is significantly improved if each portion of the array is read only once. After reading a portion or chunk of the array, corresponding portions of AB, AC, and BC can be updated simultaneously. Even if the array ABC is in main memory, better cache reuse is facilitated by updating portions of AB, AC, and BC simultaneously. The same issue applies at later stages in data cube construction, e.g., in computing A and B from AB.

Using minimal parents: In our example, the arrays AB, BC, and AC need to be computed from ABC, by aggregating values along the dimensions C, A, and B, respectively. However, the array A can be computed from either AB or AC, by aggregating along dimensions B or C. Because $|B| \le |C|$, it requires less computation to compute A from AB. Therefore, AB is referred to as the *minimal* parent of A.

A lattice can be used to denote the options available for computing each array within the cube. This lattice is shown in Figure 1. A data cube construction algorithm chooses a spanning tree of the lattice shown in the figure. The overall computation involved in the construction of the cube is minimized if each array is constructed from the *minimal parent*. Thus, the selection of a *minimal spanning tree* with minimal parents for each node is one of the important considerations in the design of a sequential (or parallel) data cube construction algorithm.

Memory Management: In data cube construction, not only the input datasets are large, but the output produced can be large also. Consider the data cube construction using the minimal spanning tree shown in Figure 1. Sufficient main memory may not be available to hold the arrays AB, AC, BC, A, B, and C at all times. If a portion of the array AB is written to the disk, it may have to be read again for computing A and B. However, if a portion of the array BC is written back, it may not have to be read again.

These issues are addressed by a new data structure, aggregation tree, that we introduce in the next section.

3. SPANNING TREES FOR CUBE CONSTURCTION

This section introduces a data structure that we refer to as the *aggregation tree*. An aggregation tree is parameterized with the ordering of the dimensions. For every unique ordering between the dimensions, the corresponding aggregation tree represents a spanning tree of the data cube lattice we described in the previous section. Aggregation tree has the property that it bounds the total memory requirements for the data cube construction process.

To introduce the aggregation tree, we initially review *prefix tree*, which is a well-known data structure [2].

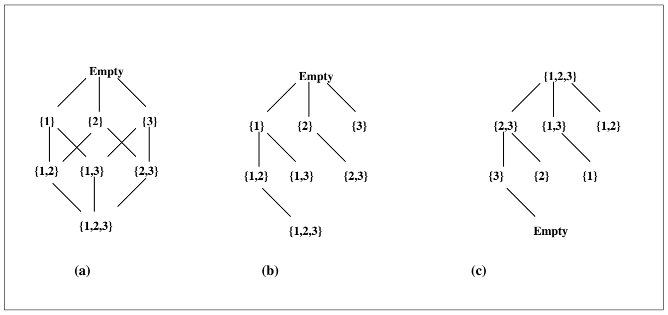

Figure 2: Prefix Lattice (a), Prefix Tree (b), and Aggregation Tree (c) for n = 3

Consider a set $X = \{1, 2, \ldots, n\}$. Let $\rho(X)$ be the power set of X.

DEFINITION 1. *$L(n)$ is a lattice (V, E) such that:*

- *The set of nodes V is identical to the power set $\rho(X)$.*
- *The set of edges E denotes the* immediate superset *relationship between elements of the power set, i.e, if $r \in \rho(X)$ and $s \in \rho(X)$, $r = s \cup \{i\}$, and $i \notin s$, then $(r, s) \in E$.*

The lattice $L(n)$ is also referred to as the *prefix* lattice. The lattice we have shown earlier in Figure 1 is a complement of the prefix lattice, and is referred to as the *data cube* lattice.

A prefix tree $P(n)$ is a spanning tree of the prefix lattice $L(n)$. It is defined as follows:

DEFINITION 2. *Given a set $X = \{1, 2, \ldots, n\}$, a prefix tree $P(n)$ is defined as follows:*

(a) ϕ is the root of the tree.

(b) The set of nodes of the tree is identical to the power set $\rho(X)$.

(c) A node $\{x_1, x_2, \ldots, x_m\}$, where $m \leq n$, and $1 \leq x_1 < x_2 < \ldots < x_m \leq n$, has $n - x_m$ children. These children, ordered from left to the right are, $\{x_1, x_2, \ldots, x_m\} \cup \{x_m + 1\}, \ldots, \{x_1, x_2, \ldots, x_m\} \cup \{n\}$.

Given a prefix tree $P(n)$, the corresponding aggregation tree $A(n)$ is constructed by complementing every node in $P(n)$ with respect to the set X. Formally,

DEFINITION 3. *Given a set $X = \{1, 2, \ldots, n\}$ and the prefix tree $P(n)$ as defined earlier, an aggregation tree $A(n)$ is defined as follows:*

(a) If r is a node in $P(n)$, then there is a node r' in $A(n)$, such that $r' = X - r$.

(b) If a node r has a child s in $P(n)$, then the node r' in $A(n)$ has a child s'.

Figure 2 shows the prefix lattice, prefix tree and the aggregation tree for $n = 3$.

```
Construct_Cube($D_1, D_2, \ldots, D_n$)
{
  Evaluate($\{D_1, D_2, \ldots, D_n\}$)
}

Evaluate($l$)
{
  Compute all children of $l$
  For-each children $r$ from right to left
    If $r$ has no children
      Write-back to the disk
    Else Evaluate($r$)
  Write-back $l$ to the disk
}
```

Figure 3: Sequential Data Cube Construction Using the Aggregation Tree

Since an aggregation tree is a spanning tree of the data cube lattice, it can be used for data cube construction. We next present an algorithm that uses the aggregation tree and has minimally bounded memory requirements.

Figure 3 shows this sequential algorithm. Suppose we are computing data cube over n dimensions which are denoted by $D_1, D_2, \ldots, D_n$. The data cube construction algorithm starts by invoking the function *Evaluate* for the root of the aggregation tree.

When the function *Evaluate* is invoked for a node l, all children of l in the aggregation tree are evaluated. This ensures maximal cache and memory reuse, since no portion of the input dataset or an intermediate result needs to be processed more than once. After computing all children of a node, the algorithm progresses in a depth-first fashion, starting with the right-most child. An array is written back to the disk only if it is not going to be used for computing another result. Thus, the only disk traffic in this algorithm is the reading of the original input array, and writing each output (or computed) array once. Moreover, each array is written once in its entirety. Therefore, frequent accesses to the disks are not required.

The depth-first traversal, starting from the right-most child in the aggregation tree, creates a bound on the total memory requirements for storing the intermediate results. Consider data cube construction starting from a three dimensional array ABC, where the sizes of the three dimensions are $|A|$, $|B|$, and $|C|$, respectively. After the three children of the root of the aggregation tree are computed, the memory requirements for holding them in main memory are $M = |A| \times |B| + |A| \times |C| + |B| \times |C|$. The design of the aggregation tree and our algorithm ensure that the total memory requirements for holding output arrays during the entire data cube construction process are bounded by M. The reason is as follows. Suppose the ordering between the three dimensions is C, B, A. After the first step, BC can be written back. Then, the node AC is used for computing the array C. Since $|C| \leq |B| \times |C|$, the memory requirements do not increase above the factor M. After computing C, both AC and C can be written back. Then, A and B are computing from AB. Since $|A| \leq |A| \times |C|$ and $|B| \leq |B| \times |C|$, the total memory requirements again do not increase beyond M. This result generalizes to an arbitrary number of dimensions, though a detailed proof is beyond the scope of this work.

4. PARALLEL ALGORITHMS

This section focuses on algorithms for parallel data cube construction. Initially, we focus on two parallel algorithms, level one parallel and all levels parallel. Then, we discuss an optimization that reduces the communication frequency, and can be applied to both algorithms.

4.1 Level One Parallel Algorithm

Consider a n-dimensional initial array from which the data cube will be constructed. Suppose we will be using a distributed memory parallel machine with 2^p processors. Through out this paper, we will assume that the number of processors used is a power of 2. This assumption corresponds well to the parallel processing configurations used in practice and has been widely used in parallel algorithms and partitioning literature.

We partition the dimension D_i along 2^{k_i} processors, such that $\sum_{i=1}^{n} k_i = p$. Each processor is given a unique label $\{l_1, l_2, \ldots, l_n\}$ such that $0 \le l_i \le 2^{k_i} - 1$. Since $\sum_{i=1}^{n} k_i = p$, it is easy to verify that there are 2^p unique labels. A processor with the label l_i is given the l_i^{th} portion along the dimension D_i.

A processor with the label $l_i = 0$ is considered one of the *lead* processors along the dimension D_i. There are $2^p/2^{k_i}$ lead processors along the dimension D_i. The significance of a lead processor is as follows. If we aggregate along a dimension, then the results are stored in the lead processors along that dimension.

The level one parallel algorithm is presented in Figure 4.

```
Construct_Cube($D_1, D_2, \ldots, D_n$)
{
  Evaluate($\{D_1, D_2, \ldots, D_n\}$) on each processor
}

Evaluate($l$)
{
  Locally aggregate all children of $l$
  Forall children $r$ from right to left
    Let $r' = X - r = \{D_{i1}, \ldots, D_{im}\}$
    If the processor is the lead processor along $D_{i1}, \ldots, D_{im}$
      Communicate with other processors to finalize portion of $r$
      If $r$ has no children
        Write-back the portion to the disk
      Else Evaluate($r$)
  Write-back $l$ to the disk
}
```

Figure 4: Level One Parallel Data Cube Construction

We explain this algorithm with the help of an example. Consider data cube construction with $n = 3$ and $p = 3$. Let $k_1 = k_2 = k_3 = 1$, i.e., each of the three dimensions is partitioned along 2 processors. Initially, all 8 processors process the portions of $D_1D_2D_3$ they own to compute partial results for each of D_1D_2, D_1D_3, and D_2D_3.

Next, consider a processor with the label $\{0, l_2, l_3\}$. This processor communicates with the corresponding processor $\{1, l_2, l_3\}$ to compute the final values for the $\frac{1}{4^{th}}$ portion of the array D_2D_3. Similarly, a processor with the label $\{l_1, 0, l_3\}$ communicates with the corresponding processor $\{l_1, 1, l_3\}$ to get the final value for the $\frac{1}{4^{th}}$ portion of the array D_1D_3.

Consider the computation of D_1 from D_1D_3. Only 4 of the 8 processors, i.e., the ones with a label $\{l_1, 0, l_3\}$, perform this computation. These 4 processors process the portion of D_1D_3 they own to compute partial result for D_1. Then, 2 of the processors with the label $\{l_1, 0, 0\}$ communicate with the corresponding processor $\{l_1, 0, 1\}$ to each compute the final values for the half portion of the array D_1. Computation of D_2 and D_3 from D_2D_3 proceeds in a similar fashion.

4.2 All Levels Parallel Algorithm

In the algorithm we described above, the parallelism is limited. As we go down the tree, the computation of children and the process of writing back results take place only on the lead processors, and other processors don't do anything after sending out their portions to the their corresponding lead processors. In this subsection we present another algorithm, referred to as the All Levels Parallel Algorithm. This algorithm further increases the parallelism by partitioning the children in the aggregation tree.

There are no lead processors in this algorithm. Each child is further partitioned into several sections. Each processor keeps an *owned section* of each child and needs to communicate with other processors in order to finalize its owned section. We call the process *finalization* when a processor receives corresponding sections from other processors and aggregates the incoming sections to its owned section. The owned section can be determined by the label of the processor. Moreover, for a d dimensional child in the aggregation tree, we associate with it a vector $\{k_i\}$ of length d to denote along which dimensions the child is partitioned.

We make the same assumptions and use the same terminology as used for presenting the first algorithm. The algorithm is presented in Figure 5.

```
Construct_Cube($D_1, D_2, \ldots, D_n$)
{
  Evaluate($\{D_1, D_2, \ldots, D_n\}$) on each processor
}

Evaluate($l$)
{
  Locally aggregate all children of $l$
  Forall children $r = \{D_{i_1}, \ldots, D_{i_m}\}$ from right to left on each processor
    Let $r' = X - r = \{D_{j_1}, \ldots, D_{j_{n-m}}\}$
    If $\sum_{t=1}^{n-m} 2^{k_{j_t}} > 1$
      Further partition $r$ into $\sum_{t=1}^{n-m} 2^{k_{j_t}}$ sections
      Communicate with other corresponding processors to finalize the
            owned section $r_0$ of $r$
      If $r$ has no children
        Write-back the owned section $r_0$ to the disk
      Else Evaluate($r_0$)
    Else If $r$ has no children
        Write-back $r$ to the disk
      Else Evaluate($r$)
  Write-back $l$ to the disk
}
```

Figure 5: All Levels Parallel Data Cube Construction

To better explain this algorithm, we consider data cube construction with $n = 3$ and $p = 3$. Let $\{k_1 = 0, k_2 = 1, k_3 = 2\}$ be associated with $D_1D_2D_3$. We assume that all dimensions D_i are of the same size. Initially all 8 processors process the portions of $D_1D_2D_3$ they own to compute partial results for each of D_1D_2, D_1D_3, and D_2D_3.

Then on each processor, we further partition D_1D_3 and D_1D_2 into 2 and 4 portions, respectively. There is no need to partition D_2D_3, since D_1 is not partitioned. The further partitioning of D_1D_2 is along both dimensions D_1 and D_2, and the partitioning of D_1D_3 is along dimension D_1 only. After further partitioning, the vector associated with D_1D_2 is $\{k_1 = 1, k_2 = 2\}$, and the vector associated with D_1D_3 is $\{k_1 = 1, k_3 = 2\}$. Note that in our implementation, we choose the dimension with the highest size to further partition a child. This, we believe, minimizes the communication volume.

Within the four processors with the label $\{0, l_2, l_3\}$, each processor keeps its owned section of D_1D_2. The processor $\{0,0,0\}$ keeps the first section of D_1D_2 as its owned section, the processor $\{0,0,1\}$ keeps the second section, and so on. Processors then send all other sections except the owned section to corresponding processors. For example, the processor $\{0,0,0\}$ sends the second section of D_1D_2 to processor $\{0,0,1\}$, the third section to processor $\{0,1,0\}$, and so on. Processors $\{0,0,1\}$, $\{0,1,0\}$ and $\{0,1,1\}$ do the similar thing in parallel. After finalization, each processor with the label $\{0, l_2, l_3\}$ gets the final values for the $\frac{1}{8^{th}}$ portion of the array D_1D_2. The similar process takes place in parallel within processors with the label $\{1, l_2, l_3\}$ and after the finalization, each of the four processors $\{1, l_2, l_3\}$ also has the $\frac{1}{8^{th}}$ portion of the array D_1D_2.

Now we consider the computation of D_2 from D_1D_2. Since each processor keeps $\frac{1}{8^{th}}$ portion of D_1D_2, we need to compute D_2 from the owned section of D_1D_2 on each processor. We further partition D_2 into 2 sections. Processor $\{0,0,0\}$ exchanges non-owned section of D_2 with processor $\{0,1,0\}$, processor $\{0,0,1\}$ exchanges non-owned section of D_2 with processor $\{0,1,1\}$, and so on. After the finalization, each processor has the $\frac{1}{8^{th}}$ portion of D_2. The computation of D_1 and D_3 proceeds in a similar fashion.

In this algorithm, since each processor stores an owned section of result, we obtain more parallelism by involving all processors in the parallel computation of children and the process of writing back results. But there is trade-off between parallelism and the communication volume. In All Levels Parallel Algorithm, the communication volume for the first level is the same as that in Level One Parallel Algorithm. But for the other levels, All Levels Parallel Algorithm increases the communication volume. In the example above, we have to communicate for D_2 when we compute it from D_1D_2, for D_1 is further partitioned. But in Level One Parallel Algorithm, it is not necessary to communicate for D_2 since dimension D_1 is not partitioned. The impact of this on the observed execution times will be evaluated experimentally in the next section.

4.3 Parallel Data Cube Construction with Low Communication Frequency

Both Level One Parallel Algorithm and All Levels Parallel Algorithm have high communication frequency, since we communicate for each child one by one in depth-first order in the aggregation tree. This has the advantage of reducing the required memory, but can also slow down the performance because of communication latency.

In this subsection, we present optimized versions with low communication frequency for these two algorithms, in which communication for a given level of the tree is done concurrently. The optimized Level One Parallel Algorithm is shown in Figure 6. The All Levels Parallel Algorithm can be optimized similarly, and the detailes are not included here.

We take the aggregation tree in Figure 2 (c) as an example to explain how the optimized Level One Parallel Algorithm works, assuming numbers in the aggregation tree as the subscripts of three dimensions. Let $n = 3$, $p = 3$, and $k_1 = k_2 = k_3 = 1$. Initially, we aggregate D_1D_2, D_1D_3 and D_2D_3 from $D_1D_2D_3$ on each processor.

Instead of communicating for D_1D_2, D_1D_3 and D_2D_3 one by one, we only communicate once for all the children at the first level. The processor with the label $\{0, l_2, l_3\}$ communicates with the corresponding processor $\{1, l_2, l_3\}$ to compute the final value

```
Construct_Cube($TreeT$)
{
  Let $n$ be the depth of the tree
  Evaluate($T, n-1$) on each processor
}

Evaluate($T, l$)
{
  Let $r_1, r_2, ..., r_t$ be all the nodes at level $n - l$ of the tree $T$
  Locally aggregate $r_1, r_2, ..., r_t$
  Do the following in parallel on each processor and for all nodes $r_1, r_2, ..., r_t$
    Let $r_i' = X - r_i = \{D_{i1}, \ldots, D_{im}\}$, $i = 1, 2, ..., t$
    If the processor is the lead processor along $D_{i1}, \ldots, D_{im}$
      Communicate with other processors to finalize portion of $r_i$ in parallel
      If $r_i$ has no children
        Write-back to the disk
  If $l > 0$
    Evaluate($T, l-1$)
}
```

Figure 6: Optimized Level One Parallel Algorithm with Low Communication Frequency

for the $\frac{1}{4^{th}}$ portion of the array D_2D_3. In the mean time, a processor with the label $\{l_1, 0, l_3\}$ communicates with the corresponding processor with the label $\{l_1, 1, l_3\}$ to get the final value for the $\frac{1}{4^{th}}$ portion of the array D_1D_3, and a processor with the label $\{l_1, l_2, 0\}$ communicates with the corresponding processor with the label $\{l_1, l_2, 1\}$ to get the final value for the $\frac{1}{4^{th}}$ portion of the array D_1D_2. Similarly, we communicate for D_1, D_2, D_3 simultaneously when we compute D_1 from D_1D_3, D_2 and D_3 from D_2D_3 on corresponding lead processors to get the final values for the half portion of the array D_1, D_2, D_3.

In Level One Parallel Algorithm, we need 6 phases of communications for 3-dimensional data cube construction. But using the optimized algorithm, we only need to communicate twice. Thus, the communication frequency is reduced by 66.67%.

5. EXPERIMENTAL RESULTS

We have conducted a series of experiments to show the impact of communication frequency, data distribution, and parallelism on data cube construction.

Our experiments have been performed on a cluster with 16 Sun Microsystem Ultra Enterprise 450's, with 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each of the nodes has a 4 GB system disk and a 18 GB data disk. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8.

In constructing data cubes, the initial multidimensional array can be stored in a dense format or a sparse format [12]. A dense format is typically used when 40% of array elements have a non-zero value. In this format, storage is used for all elements of the array, even if their value is zero. In a sparse format, only non-zero values are stored. However, additional space is required for determining the position of each non-zero element. We use *chunk-offset compression*, used in other data cube construction efforts [12]. Along with each non-zero element, its offset within the chunk is also stored. After aggregation, all resulting arrays are always stored in the dense format. This is because the probability of having zero-valued elements is much smaller after aggregating along a dimension.

Since sparse formats are frequently used in data warehouses, all our experiments have been conducted using arrays stored in a sparse format. A sparse array is characterized by *sparsity*, which is the fraction of elements that have a non-zero value. We have experimented with different levels of sparsity.

5.1 Impact of Communication Frequency

In order to see the impact of communication frequency, we applied the two algorithms and their corresponding optimized versions presented in Section 4 to $64 \times 64 \times 64 \times 64$ dataset. We experimented with three different levels of sparsity, 25%, 5%, 2% on 8 processors. A four-dimensional dataset can be partitioned in three ways on 8 processors (i.e. when $p = 3$). These three options are, $k_1 = 0, k_2 = k_3 = k_4 = 1$, $k_1 = k_2 = 0, k_3 = 1, k_4 = 2$, and $k_1 = k_2 = k_3 = 0, k_4 = 3$. We refer to these three options as three dimensional, two dimensional, and one dimensional partitions, respectively. Figure 7 and Figure 8 show the results for Level One Parallel Algorithm and its optimized version, and the results for All Levels Parallel Algorithm and its optimized version, respectively.

Note that the advantage of the original algorithms, which have higher communication frequency, is reduced memory requirements. However, memory requirements did not appear to be a significant factor in these experiments, as sufficient memory was always available in all cases. For both algorithms, the optimized version which reduces the communication frequency outperforms the original version regardless of the data distribution and the sparsity. This is consistent with the fact that the communication frequency is determined by the structure of the aggregation tree and is not affected by the data distribution and the sparsity. Communication frequency of the optimized versions is determined by the number of levels in the aggregation tree, whereas the communication frequency of the original versions is determined by the number of children in the aggregation tree.

However, the relative differences between the corresponding original and communication frequency optimized algorithms is quite small (within 5%) in almost all cases. An exception is when the sparsity level is 25% and the data distribution is one dimensional. Thus, if memory is a constraint, the versions with reduced memory requirements will be preferable.

Since the optimized version outperforms the original one, we used only optimized versions of Level One Parallel Algorithm and All Levels Parallel Algorithm to test the impact of the data distribution and the parallelism in the following subsections. For simplicity, Level One Parallel Algorithm and All Levels Parallel Algorithm will stand for their corresponding optimized versions in the following subsections.

5.2 Impact of Data Distribution

We now report a series of experiments to evaluate the impact of data distribution on the performance. We experimented primarily with the Level One Parallel, using a number of different datasets and sparsity levels.

The first set of experimental results are obtained from $64 \times 64 \times 64 \times 64$ dataset. We experimented with three different levels of sparsity, 25%, 10%, and 5%. The results on 8 processors for Level One Parallel Algorithm are presented in Figure 9. The sequential execution time were 22.5, 12.4, and 8.6 seconds, with sparsity levels of 25%, 10%, and 5%, respectively.

Our results from the previous section suggest that when $|D_1| = |D_2| = |D_3| = |D_4|$, partitioning more dimensions reduces the communication volume. Our results from Figure 9 validate this. Three dimensional partition outperforms both two dimensional and one dimensional partitions at all three sparsity levels. The version with two dimensional partition is slower by 7%, 12%, and 19%, when the sparsity level is 25%, 10% and 5%, respectively. The version with one dimensional partition is slower by 31%, 43%, and 53% over the three cases. The ratio of communication to computation increases as the array becomes more sparse. Therefore, a greater performance difference between different versions is observed.

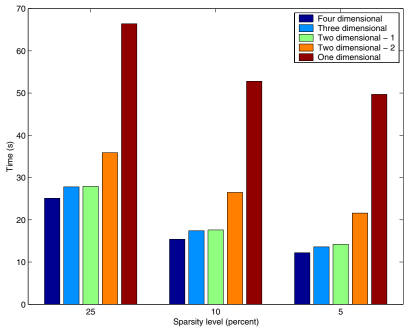

Figure 11: Impact of Data Distribution for Level One Parallel Algorithm, 128^4 dataset, 16 processors

The speedups of the three-dimensional version were 5.34, 4.22, and 3.39, with the sparsity levels of 25%, 10%, and 5%, respectively. We believe that these are good speedups considering the small problem size and high ratio of communication to computation.

As we had stated earlier, the Level One Parallel Algorithm sequentializes a part of the computation after the first level of the aggregation tree. With different choices for partitioning, the amount of computation of performed on different nodes is, therefore, different. So, this could be another factor behind the observed difference in execution time. However, the dominant part of the computation in data cube construction is at the first level and is not affected by the partitioning choice made. Therefore, we can conclude that the difference in performance seen as a result of the partitioning choice made is primarily because of the difference in communication volume.

Next, we consider $128 \times 128 \times 128 \times 128$ arrays with sparsity levels of 25%, 10%, and 5%. Figure 10 shows experimental results on 8 processors for Level One Parallel Algorithm. Again, the problem can be partitioned in three ways and we have implemented all three. The sequential execution time for 25%, 10%, and 5% cases are 321, 154, and 97 seconds, respectively.

The experimental results again validate our theoretical result that three dimensional partition is better than two dimensional or one dimensional partition. The version with two dimensional partition is slower by 8%, 15% and 16% with sparsity levels of 25%, 10%, and 5% respectively. The version with one dimensional partition is slower by 30%, 42%, and 51% over the three cases. The speedups of the three dimensional versions are 6.39, 5.31, and 4.52, with sparsity levels of 25%, 10%, and 5%, respectively. The speedups reported here are higher because of the larger dataset, which results in relatively lower communication to computation ratio.

Finally, we have also executed the same dataset on 16 processors. A four-dimensional dataset can be partitioned in five ways on 16 processors (i.e. when $p = 4$). These five options are, $k_1 = k_2 = k_3 = k_4 = 1$, $k_1 = 0, k_2 = k_3 = 1, k_4 = 2$, $k_1 = k_2 = 0, k_3 = k_4 = 2$, $k_1 = k_2 = 0, k_3 = 1, k_4 = 3$, and $k_1 = k_2 = k_3 = 0, k_4 = 4$.

The first, second, and the fifth option represent unique choices for four dimensional, three dimensional, and one dimensional par-

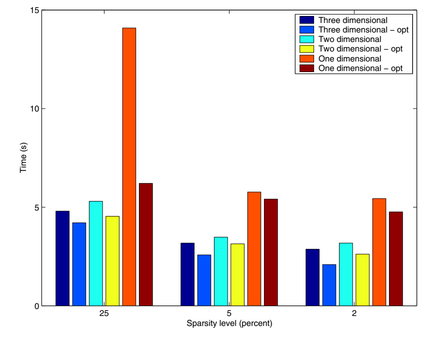

Figure 7: Impact of Communication Frequency for Level One Parallel Algorithm, 64^4 dataset, 8 processors

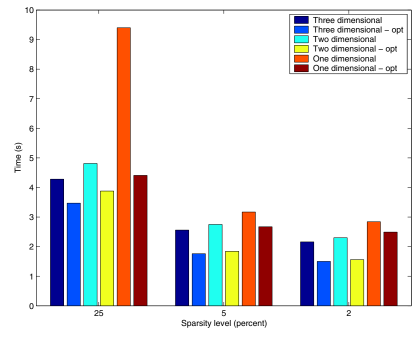

Figure 8: Impact of Communication Frequency for All Levels Parallel Algorithm, 64^4 dataset, 8 processors

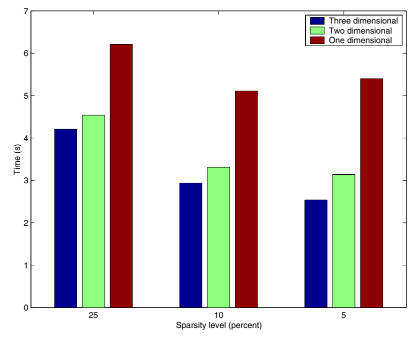

Figure 9: Impact of Data Distribution for Level One Parallel Algorithm, 64^4 dataset, 8 processors

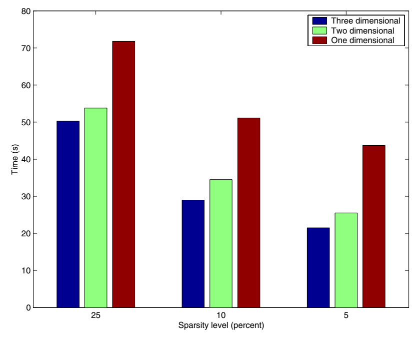

Figure 10: Impact of Data Distribution for Level One Parallel Algorithm, 128^4 dataset, 8 processors

tition. There are two different choices for two dimensional partition. Results from these five partitions, and for sparsity levels of 25%, 10%, and 5%, are shown in Figure 11.

The relative performance of the five versions is as predicted by the theoretical analysis we have done. The version with four dimensional partition always gives the best performance, followed by the version with three dimensional partition, the two dimensional version with $k_1 = k_2 = 0, k_3 = k_4 = 2$, the other two dimensional version, and the finally the one dimensional version. In fact, with sparsity level of 5%, there is more than 4 times performance difference between the best and the worst version.

The speedups of the best version are 12.79, 10.0, and 7.95, with sparsity levels of 25%, 10%, and 5%, respectively.

5.3 Impact of Level of Parallelism

When we presented All Levels Parallel Algorithm in Section 4, we pointed out that though All Levels Parallel Algorithm obtained more parallelism than Level One Parallel Algorithm, it also increased the communication volume. In order to see the trade-off, we experimented with $128 \times 128 \times 128 \times 128$ dataset on 8 processors for both algorithms. We only considered the three dimensional partition, since we have shown that three dimensional partition outperforms the other partitioning choices for $128 \times 128 \times 128 \times 128$ dataset. The experimental results are presented in Figure 14.

We have two observations from Figure 14. First, All Levels Parallel Algorithm has better performance than Level One Parallel Algorithm, though its communication volume is greater than the latter. As we had stated earlier, both algorithms have the same communication volume at the first level. Since the dominant part of computation is at the first level, the increase of communication volume at other levels has less impact on the execution time than

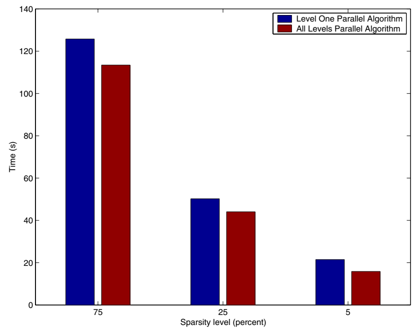

Figure 14: Impact of Parallelism, 128^4 dataset, 8 processors, Three Dimensional Partition

the increase of parallelism does. Therefore, All Levels Parallel Algorithm gets better performance.

The second observation is that the sparser the dataset is, the better performance All Levels Parallel Algorithm achieves over Level One Parallel Algorithm. For the three sparsity levels 75%, 25%, 5%, All Levels Parallel Algorithm reduces the execution time compared to Level One Parallel Algorithm by 9.79%, 12.26%, 26.22%, respectively. This is because the ratio of the communication to computation is higher in the sparser dataset.

More comparisons between the two algorithms can be performed by examining the corresponding results from the previous subsection, where the impact of data distribution was studied for both the algorithms. Overall, the trends are the same as explained above.

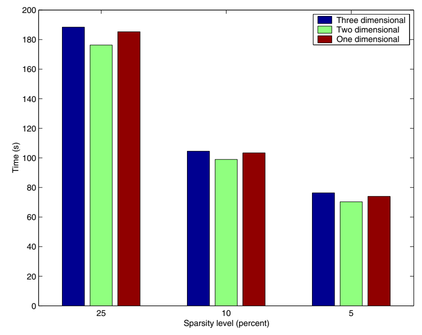

Figure 12: Impact of Data Distribution for Level One Parallel Algorithm, $32 \times 64 \times 256 \times 2048$ dataset, 8 processors

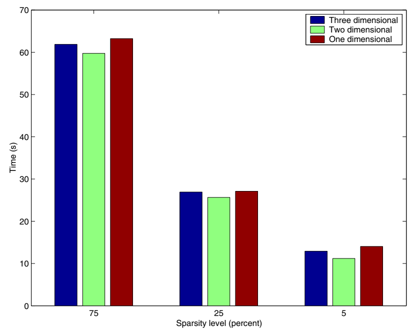

Figure 13: Impact of Data Distribution for Level One Parallel Algorithm, $64^3 \times 512$ dataset, 8 processors

6. RELATED WORK

Since Gray *et al.* [6] proposed the data cube operator, techniques for data cube construction have been extensively studied for both relational databases [9, 8] and multidimensional datasets [12, 10]. Our work belongs to the latter group. Zhao *et. al* [12] use MMST (Minimum Memory Spanning Tree) with optimal dimension order to reduce memory requirements in sequential data cube construction. Our primary focus has been on parallelization, including ordering of dimensions and partitioning to minimize communication volume. In the process, we have used the aggregation tree to bound the total memory requirements, without requiring frequent writing to the disks. Tam [10] uses MNST (Minimum Number Spanning Tree) to reduce computing cost. Again, our primary contribution is in looking at parallelization.

Goil *et. al* [4, 5] did the initial work on parallelizing data cube construction starting from multidimensional arrays. In comparison, our work has focused on evaluating the impacts of level of parallelism, data distribution, and communication frequency on the performance. It should also be noted that their work has used only single dimensional partitions, whereas our experiments and mathematical analysis have demonstrated that multidimensional partitions often perform better. Recently, Dehne *et. al* [3] have studied the problem of parallelizing data cube. They focus on a *shared-disk* model where all processors access data from a common set of disks. Because there is no need to partition the data-set, they can partition the tree. In comparison, we have focused on a shared-nothing model, which we believe is also more commonly used in practice. Their effort does not consider the memory requirements issue either. Ng *et al.* have reported parallel algorithms for iceberg-cube computation [7]. The algorithms they have parallelized are very different from the original cube construction problem that we have focused on.

7. CONCLUSIONS

In this paper, we have focused on a number of trade-offs that arise in parallel data cube construction. We have presented a set of algorithms that represent different choices that could be made for level of parallelism and frequency of communication. For our basic algorithm, we have analyzed the total communication volume and shown how it relates to the data distribution of the original array.

We have carried out a set of experiments to evaluate the impact of data distribution, frequency of communication, and level of parallelism, as the cube size and the sparsity of the dataset are varied. Our results can be summarized as follows: 1) In all cases, reducing the frequency of communication and using higher memory gave better performance, though the difference was relatively small. 2) Choosing data distribution to minimize communication volume made a substantial difference in the performance in most of the cases. This difference became more significant with increasing sparsity level in the dataset. 3) Finally, using parallelism at all levels gave better performance, even though it increases the total communication volume. Again, the difference is more significant as the sparsity level in the dataset increases.

8. REFERENCES

[1] S. Chaudhuri and U. Dayal. An overview of datawarehousing and olap technology. *ACM SIGMOD Record*, 26(1), 1997.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[3] Frank Dehne, Todd Eavis, Susanne Hambrusch, and Andrew Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases: An International Journal (Special Issue on Parallel and Distributed Data Mining), to appear*, 2002.

[4] Sanjay Goil and Alok Choudhary. High performance OLAP and data mining on parallel computers. Technical Report CPDC-TR-97-05, Center for Parallel and Distributed Computing, Northwestern University, December 1997.

[5] Sanjay Goil and Alok Choudhary. PARSIMONY: An infrastructure for parallel multidimensional analysis and data mining. *Journal of Parallel and Distributed Computing*, 61(3):285–321, March 2001.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregational Operator for Generalizing Group-Bys, Cross-Tabs, and Sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, 1995.

[7] Raymong Ng, Alan Wagner, and Yu Yin. Iceberg-cube computation with pc clusters. In *Proceedings of ACM SIGMOD*, 2001.

[8] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 23rd Int. Conf. Very Large Data Bases*, pages 263–277, Athens, Greece, August 1997.

[9] S.Agrawal, R. Agrawal, P. M.Desphpande, A. Gupta, J.F.Naughton, R. Ramakrishnan, and S.Sarawagi. On the computation of multidimensional aggregates. In *Proc 1996 Int. Conf. Very Large Data Bases*, pages 506–521, Bombay, India, September 1996.

[10] Yin Jenny Tam. Datacube: Its implementation and application in olap mining. Master's thesis, Simon Fraser University, September 1998.

[11] Ge Yang, Ruoming Jin, and Gagan Agrawal. Implementing data cube construction using a cluster middleware: Algorithms, implementation experience and performance evaluation. In *The 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin,Germany, May 2002.

[12] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. An array based algorithm for simultaneous multidimensional aggregates. In *Prceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170. ACM Press, June 1997.