Communication and Memory Optimal Parallel Data Cube Construction

Ruoming Jin  Ge Yang  Karthik Vaidyanathan  Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{jinr.yangg,vaidyana,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.

This paper addresses a number of algorithmic issues in parallel data cube construction. First, we present an aggregation tree for sequential (and parallel) data cube construction, which has minimally bounded memory requirements. An aggregation tree is parameterized by the ordering of dimensions. We present a parallel algorithm based upon the aggregation tree. We analyze the interprocessor communication volume and construct a closed form expression for it. We prove that the same ordering of the dimensions minimizes both the computational and communication requirements. We also describe a method for partitioning the initial array and prove that it minimizes the communication volume.

Experimental results from implementation of our algorithms on a cluster of workstations validate our theoretical results.

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).

Computing multiple related group-bys and aggregates is one of the core operations in OLAP applications. Jim Gray has proposed the cube operator, which computes group-by aggregations over all possible subsets of the specified dimensions [5]. 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 [10, 13, 11].

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 [2, 3, 4].

There are a number of issues in designing a parallel data cube algorithm, which can be summarized as follows. The first three of these four issues are also applicable to sequential data cube construction.

Cache and Memory Reuse: Since the input datasets are typically very large, cache and memory reuse is an important consideration. Consider an initial 3 dimensional array ABC. Arrays AB, AC, and BC are computed from ABC by aggregating along C, B, and A dimensions, respectively. When a portion of the array ABC is read from a disk (or main memory), it is important to update corresponding portions of AB, AC, and BC simultaneously, and avoid reading ABC several times.

Using minimal parents: Next, consider the computation of the array A. It can be computed either from the array AB or the array AC. If the size of the dimension B is smaller than the size of the dimension C, it requires less computation to use AB. In this case, AB is referred to as the minimal parent of A. Obviously, a sequential or parallel algorithm is more efficient if it computes each array from its minimal parent.

Memory Management: The size of the output produced by a data cube construction algorithm can be extremely large. Therefore, it is important to carefully use the available main memory.

Communication Volume: Because of the size of the output produced and need for aggregating along all dimensions, parallel data cube construction can involve a large interprocessor communication volume. In a distributed memory machine, minimizing communication volume is a key to achieving high parallel performance.

In this paper, we address a number of algorithmic issues for parallel data cube construction. The main contributions of this paper can be summarized as follows.

- We have developed a data-structure called aggregation tree, which ensures maximal cache and memory reuse in data cube construction. Moreover, we show that the size of the intermediate results that need to be held in main memory are bounded when a data cube is constructed by a right to left, depth-first traversal of the aggregation tree.
- We present a parallel algorithm for data cube construction. We develop a closed form expression for the communication volume required for parallel data cube construction using the aggregation tree.
- The aggregation tree is parameterized by the ordering of dimensions. If the original array is n-dimensional, there are \( n! \) instantiations of the aggregation tree. We show that the same ordering of the dimensions ensures that each array is computed from its minimal parent, as well as minimizes the communication volume.
- The communication volume is further dependent upon the partitioning of the original array between the processors. We have developed an algorithm for partitioning the array. We
show that our approach minimizes the interprocessor communication volume.

- We have implemented our parallel algorithm on a cluster of workstations. We present experimental results that validate our theoretical results on partitioning.

The rest of the paper is organized as follows. We further discuss the data cube construction problem in Section 2. Our aggregation tree is introduced in Section 3. The same section also establishes the key properties of this data-structure. A parallel data cube construction algorithm that uses the aggregation tree is described in Section 4. We also analyze the communication volume in this section. Selecting the ordering of the dimensions and partitioning between the processors are addressed in Section 5. Experimental results are presented in Section 6. We compare our work is related efforts in Section 7 and conclude in Section 8.

2. DATA CUBE CONSTRUCTION

![Lattice for data cube construction](image)

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

This section further elaborates on the four issues in parallel data cube construction we listed in the previous section. 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 combination of values. If the original dataset is n-dimensional, this implies computing and storing $nC_m$ m-dimensional arrays, for $0 \leq m \leq n$. $nC_m$ 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 $|A| \leq |B| \leq |C|$. 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 $|A| \times |B|$

We now revisit the four issues we had listed earlier in Section 1. We use the above example to further illustrate these issues.

**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 $|B| \leq |C|$, 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.

**Communication Volume:** Consider the computation of AB, AC, and BC from ABC. Suppose we assume that the dataset will be partitioning along a single dimension. Then, the communication volume required when the dataset is partitioned along the dimensions A, B, or C is $|B| \times |C|$, $|A| \times |C|$, and $|A| \times |B|$, respectively. If $|A| \leq |B| \leq |C|$, then the minimal communication volume is achieved by partitioning along the dimension C.

High communication volume can easily limit parallel performance. It is important to minimize communication volume for the entire data cube construction process, possibly by considering partitioning along multiple dimensions.

3. SPANNING TREES FOR CUBE CONSTRUCTION

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 had 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 [1].

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 \( E \) denote 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 had 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) \( \phi \) 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 \( n = 3 \).

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, as we prove below.

**Theorem 1.** Consider an original \( n \) dimensional array
D_1, D_2, \ldots, D_n where the size of the dimension D_i is |D_i|. The total memory requirement for holding the results in data cube construction using the algorithm in Figure 3 are bounded by

\[ \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) \]

**Proof:** Let A(n) be the aggregation tree used for data cube construction. Let P'(n) be the corresponding prefix tree. A snapshot of the aggregation tree comprises nodes that have been computed and have not yet been written to the disks. In other words, it includes all arrays that need to be held in main memory. Let A'(n) be the snapshot of the aggregation tree any given time and let P'(n) be the corresponding snapshot of the prefix tree.

A snapshot of the prefix tree is shown in Figure 4. All possible snapshots during data cube construction are either captured by this figure, for a choice of y_1, y_2, \ldots, y_m, where 1 \leq m \leq n and 1 \leq y_1 < y_2 < \ldots < y_{m-1} < y_m = n, or a subset of a snapshot captured by this figure.

Consider a node \{y_1, y_2, \ldots, y_i, y_i + k\} in the prefix tree. Then the corresponding node in the aggregation tree is \{x_1, x_2, \ldots, x_{n-(i+1)}\}, where x_j \neq y_1, y_2, \ldots, y_i, y_i + k. The memory requirement for this node in the aggregation tree is

\[ \prod_{j=1, j \neq y_1, \ldots, y_i, y_i + k}^{n} |D_j| \]

The total memory requirements for holding the results, (i.e. not including the initial n-dimensional array) for any snapshot captured in Figure 4 will be

\[ \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) + \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i, j \neq y_1}^{n} |D_j| \right) + \ldots + \sum_{i=m-1}^{n} \left( \prod_{j=1, j \neq i, j \neq y_1, \ldots, y_{m-1}}^{n} |D_j| \right) \]

The above quantity is less than or equal to

\[ \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) \]

The above bound is an important property of the aggregation tree. It further turns out that no other spanning tree results in lower memory requirements, as long as the algorithm does maximal cache and memory reuse, and does not write-back portions of the resulting arrays to the disks.

**Theorem 2.** The memory requirements for holding the results during data cube construction using any spanning tree and algorithm are at least

\[ \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) \]

provided that the algorithm does maximal cache and memory reuse and does not write-back portions of the computed arrays to the disks.

**Proof:** To ensure maximal cache and memory reuse, the algorithm must compute all first level nodes in the data cube lattice from the root node simultaneously. The root node in the data cube lattice, \{1, 2, \ldots, n\} has n children, which can be denoted by c_1, c_2, \ldots, c_n, where, c_i = \{j|j = 1, 2, \ldots, n, j \neq i\}. The memory requirements for holding the n corresponding arrays are

\[ \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) \]

In practice, data cube construction algorithms cannot always hold all elements of computed arrays in the main memory at any given time. For example, the factor

\[ M = \sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} |D_j| \right) \]

can exceed the available main memory. In prior work on data cube construction, two approaches have been proposed for such cases. In the first approach, an element of an array is written back to the
disks as soon as the element’s final value has been computed and is not required for further computations [13]. The second approach is based upon tiling [12]. Consider $m$ arrays that are computed from the same parent. These $m$ arrays are divided into tiles, such that each tile fits in the main memory. Tiles are allocated and computed one at a time.

An obvious question is, “what is the significance of aggregation tree when the factor $M$ exceeds the available main memory?” By having a bound on the total memory requirements, the aggregation tree minimizes the number of tiles that are required, therefore, minimizing the total I/O traffic. More detailed examination of tiling with aggregation tree is beyond the scope of this paper.

Because of aggregation tree’s minimally bounded memory requirements while ensuring maximal cache and memory reuse, it appears to be promising for parallel data cube construction also. We examine the use of aggregation tree for parallel data cube construction in the next section.

4. PARALLEL DATA CUBE CONSTRUCTION USING THE AGGREGATION TREE

In this section, we present a parallel algorithm for data cube construction using the aggregation tree. We then develop a closed form expression for the communication volume involved. We also show that the memory requirements for parallel cube construction are also bounded with the use of aggregation tree.

Consider again 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 $\sum_{i=1}^{p} k_i = p$. Each processor is given a unique label $\{i_1, i_2, \ldots, i_n\}$ such that $0 \leq i_l \leq 2^{k_l} - 1$. 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 $i_l$ is given the $l^{th}$ portion along the dimension $D_l$.

A processor with the label $i_l = 0$ is considered one of the lead processors along the dimension $D_l$. There are $2^p/2^{k_l}$ lead processors along the dimension $D_l$. 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.

Parallel algorithm for data cube construction using the aggregation tree is presented in Figure 5.

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 $D_1 D_2 D_3$ and they own to compute partial results for each of $D_1 D_2$, $D_1 D_3$, and $D_2 D_3$.

Next, consider a processor with the label $\{0, i_2, i_3\}$. This processor communicates with the corresponding processor $\{1, i_2, i_3\}$ to compute the final values for the $D_1$ portion of the array $D_1 D_2 D_3$. Similarly, a processor with the label $\{i_1, 0, i_3\}$ communicates with the corresponding processor $\{i_1, 1, i_3\}$ to get the final value for the $D_2$ portion of the array $D_1 D_2 D_3$.

Consider the computation of $D_1$ from $D_1 D_2 D_3$. Only 4 of the 8 processors, i.e., the ones with a label $\{i_1, 0, i_3\}$, perform this computation. These 4 processors process the portion of $D_1 D_2 D_3$ they own to compute partial result for $D_1$. Then, 2 of the processors with the label $\{i_1, 0, 1\}$ communicate with the corresponding processors $\{i_1, 0, 1\}$ to compute the final values for the half portion of the array $D_1$. Computation of $D_2$ and $D_3$ from $D_2 D_3$ proceeds in a similar fashion.

Note that our algorithm sequentializes portions of the computation. However, while computing a data cube when the number of dimensions is not very large, the dominant part of the computation is at the first level. For example, when $n = 4$, the sizes of all dimensions are identical, and the original array is dense, 98% of the computation is at the first level. The computation at the first level is fully parallelized by our algorithm.

An important questions is, “what metric(s) we use to evaluate the parallel algorithm?”. The dominant computation is at the first level, and it is fully parallelized by the algorithm. Our earlier experimental work [12] has shown that communication volume is a critical factor in the performance of parallel data cube construction on distributed memory parallel machines. Therefore, we focus on communication volume as a major metric in analyzing the performance of a parallel data cube construction algorithm.

**Lemma 1.** Consider a node $r = \{y_1, y_2, \ldots, y_n\}$ and its child $s = \{y_1, y_2, \ldots, y_k, m\}$ in the prefix tree, where $1 \leq y_1 < y_2 < \ldots < y_k < m \leq n$. Then, the communication volume in computing the corresponding node $s'$ in the aggregation tree from the node $r'$ is given by

$$\left( \prod_{i=1, j \neq \{y_1, y_2, \ldots, y_k, m\}}^{n} |D_i| \right) \times (2^m - 1)$$

**Theorem 3.** The total communication volume for data cube construction is given by

$$\left( \prod_{i=1}^{n} |D_i| \right) \times \sum_{i=1}^{n} 2^{k_i} - 1 \times \left( \prod_{j=1}^{n-1} (1 + \frac{1}{|D_j|}) \right)$$

We next focus on memory requirements for parallel data cube construction using the aggregation tree. In parallel computation on a distributed memory machine, memory is required for local computations, as well as for temporarily storing the data received from other processors.

In parallel data cube construction, the memory requirements for storing the locally aggregated values depends only upon the spanning tree used and the sizes of the dimensions. The memory requirements for storing the data received from other processors depends upon the implementation. In an extreme case, a processor
can receive a single element from one other processor, add it to the corresponding local element, and then use the same one element buffer for receiving another element, possibly from a different processor. Obviously, such an implementation will be very inefficient because of the high overhead due to the communication and synchronization latencies. However, there is a tradeoff between communication frequency and memory requirements, which is hard to analyze theoretically.

So, to simplify our theoretical analysis, we focus on memory requirements for local aggregations only. We first show that such memory requirements are minimally bounded with the use of aggregation tree.

**Theorem 4.** Consider an original \( n \) dimensional array \( D_1, D_2, \ldots, D_n \) where the size of the dimension \( D_i \) is \( |D_i| \) and is partitioned among \( 2^p \) processors. When data cube construction is done using \( 2^p \) processors, where \( p = \sum_{i=1}^{n} k_i \), the memory requirements on any processor for holding the results in data cube construction using any spanning tree and algorithm are at least

\[
\frac{\prod_{i=1}^{n} |D_i|}{2^p} \times \left( \sum_{i=1}^{n} 2^{k_i} \right)
\]

provided that the algorithm does maximal cache and memory reuse and does not write-back portions of the computed arrays to the disks.

5. **Optimality Properties and Partitioning**

As we had stated earlier, an aggregation tree is parameterized with the ordering of dimensions. In computing data cube starting from an \( n \) dimensional array, \( n! \) instantiations of the aggregation tree are possible.

In this section, we prove an important result, which is that the same ordering of dimensions minimizes both the communication volume and the computation cost. The latter also means that all nodes in the data cube lattice are computed from minimal parents.

**Theorem 6.** Among all instantiations of the aggregation tree, minimal communication volume is achieved by the instantiation where \( |D_1| \geq |D_2| \geq \ldots \geq |D_n| \).

**Theorem 7.** Using aggregation tree ensures that all arrays are computed from their minimal parents if \( |D_1| \geq |D_2| \geq \ldots \geq |D_n| \).

The next issue we focus on is partitioning of the original dataset between the processors. The expression for communication volume we derived in the previous section is dependent on the partitioning of the original array between the processors, i.e., the values of \( k_i \), \( i = 1, \ldots, n \). Given \( 2^p \) processors and an original array with \( n \) dimensions, there are a total of \( n^p P_n \) distinct ways of partitioning the array between processors. In general, it is not feasible to evaluate the communication costs associated with each of these partitions. We have developed an \( O(p) \) time algorithm for choosing the values of \( k_i \), \( i = 1, \ldots, n \), \( \sum_{i=1}^{n} k_i = p \) to minimize the total communication volume. Later, we will present a detailed proof that our algorithm does minimize the total communication volume.

Recall that the expression for communication volume we derived is

\[
\left( \prod_{i=1}^{n} |D_i| \right) \times \left( \sum_{i=1}^{n} 2^{k_i} \right) \times \left( \prod_{j=1}^{i-1} \left( 1 + \frac{1}{|D_j|} \right) \right)
\]

This can be restated as

\[
\left( \prod_{i=1}^{n} |D_i| \right) \times \left( \sum_{i=1}^{n} 2^{k_i} \right) \times \left( \prod_{j=1}^{i-1} \left( 1 + \frac{1}{|D_j|} \right) \right) =
\]

\[
\sum_{i=1}^{n} \frac{1}{|D_i|} \times \left( \prod_{j=1}^{i-1} \left( 1 + \frac{1}{|D_j|} \right) \right)
\]

Our goal is to choose the values of \( k_i \) for a set of given values of \( |D_i| \), \( i = 1, \ldots, n \). Therefore, we state the communication volume as

\[
c_0 \times \left( \sum_{i=1}^{n} 2^{k_i} \times X_i \right) - d_0
\]

where,

\[
X_i = \frac{1}{|D_i|} \times \left( \prod_{j=1}^{i-1} \left( 1 + \frac{1}{|D_j|} \right) \right)
\]

and the values of \( c_0 \) and \( d_0 \) do not impact the choices of \( k_i \).

The algorithm is presented in Figure 6. Initially, \( k_1 \), for all values of \( i \), are initialized to 0. In each iteration of the algorithm, we find the \( X_i \) with the minimal value, increment the corresponding \( k_i \) by 1, and replace \( X_i \) with \( 2 \times X_i \).

```plaintext
Partition(n, p, X1, X2, ..., Xn)
{
    Initialize k1 = k2 = ... = kn = 0
    While (p > 0) {
        Let \( X_i = \min(X1, X2, ..., Xn) \)
        \( k_i = k_i + 1 \)
        \( X_i = 2 \times X_i \)
        p = p - 1
    }
}
```

**Figure 6:** Partitioning Different Dimensions to Minimize Communication Volume

**Theorem 8.** Partitioning done using the algorithm in Figure 6 minimizes the interprocessor communication volume.

6. **Experimental Results**

We have conducted a series of experiments with two major goals. First, we show that our algorithm achieves high speedups, even though portion of the computation is sequentialized. Second, we show that the versions with partitioning that minimizes communication volume does achieve significantly better performance than versions with other partitioning choices.

Our experiments have been performed on a cluster with 16 Sun Microsystems 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 node have 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 multi-dimensional array can be stored in a dense format or a sparse format [13]. 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 [13]. 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.

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 are presented in Figure 7. 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 are three dimensional, two dimensional, and one dimensional partitions, respectively. Results from these three options are presented in Figure 7. The sequential execution times were 22.5, 12.4, and 8.6 seconds, with sparsity levels of 25%, 10%, and 5%, respectively.

Our results from Section 5 suggest that when $|D_1| = |D_2| = |D_3| = |D_4|$, partitioning more dimensions reduces the communication volume. Our results from Figure 7 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.

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, our 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 performed on different nodes is, therefore, different. So, this could be another factor behind the observed difference in execution times. 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 8 shows experimental results on 8 processors. Again, the problem can be partitioned in three ways and we have implemented all three. The sequential execution times are 321, 154, and 97 seconds, for 25%, 10%, and 5% cases, respectively.

The experimental results again validate our theoretical result that three dimensional partition is better than two dimensional or one dimensional. The version with two dimensional partition is slower by 8%, 15% and 16% with sparsity levels of 25%, 10%, and 5%. The version with one dimensional partition is slower by 30%, 42%, and 51% over the three cases. The speedups of the three dimensional version are 6.39, 5.31, and 4.52, with sparsity levels of 25%, 10%, and 5%.
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 partition. 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 9.

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 \( k_1 = k_2 = k_3 = k_4 = 2 \), 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.

7. RELATED WORK

Since Jim Gray [5] proposed the data cube operator, techniques for data cube construction have been extensively studied for both relational databases [10, 9] and multi-dimensional datasets [13, 11]. Our work belongs to the latter group. Zhao et. al [13] use MMST (Minimum Memory Spanning Tree) with optimal dimension order to reduce memory requirements in sequential data cube construction. However, their method requires frequent write operation to the disks. In comparison, we have used the aggregation tree to bound the total memory requirements, without requiring frequent write to the disks. In addition, we have focused on parallelization, including ordering of dimensions and partitioning to minimize communication volume. Tam [11] uses MNST (Minimum Number Spanning Tree) to reduce computing cost, with ideas somewhat similar to our prefix tree. However, this method also requires frequent write back to disks. Neither Zhao’s nor Tam’s approaches have been parallelized, and we believe that they will be difficult to parallelize because of the need for frequent writing to the disks.

Goil et. al [3, 4] did the initial work on parallelizing data cube construction starting from multi-dimensional arrays. In comparison, our work includes concrete results on minimizing memory requirements, communication volume, and partitioning. Recently, Dehne et. al [2] 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. There effort does not consider the memory requirements issue either.

The work reported here is also very different from our earlier publication on data cube construction [12]. In that effort, we described implementation of a 3-dimensional case using a cluster middleware. The use of aggregation tree and theoretical results on memory requirements, communication volume and partitioning are novel to this paper.

Recently, there have been extensive research on partial materialization of a data cube [8, 7, 6]. Although our current research has concentrated on complete data cube construction, we believe that the results we have obtained here could form the basis for work on partial data cube construction. In the future, we will like to apply our results on bounded memory requirements and communication volume to partial materialization.

8. CONCLUSIONS

In this paper, we have addressed a number of algorithmic and theoretic results for sequential and parallel data cube construction. For sequential data cube construction, we have developed a data structure called aggregation tree. If the data cube is constructed using a right-to-left depth-first traversal of the tree, the total memory requirements are minimally bounded. As compared to the existing work in this area, our approach achieves a memory bound without requiring frequent writing back to the disks. This, we believe, makes our approach more practical and also suitable for parallelization.

We have presented a number of results for parallel data cube construction. First, we have presented an aggregation tree based algorithm for parallel data cube construction. Again, we have shown that memory requirements are minimally bounded. We have also developed a closed form expression for total communication volume in data cube construction. We have shown that the same ordering of dimensions minimizes both the communication volume as well as computation. Finally, we have presented an algorithm with \( O(p) \) time complexity for optimally partitioning the input array on \( 2^p \) processors, with the goal of minimizing the communication requirement. There is very limited prior work on parallel cube construction on a shared-nothing architectures, and this earlier work did not establish any theoretical bounds.

We have obtained experimental results from an implementation of our parallel algorithm on a cluster of workstations. These results establish that 1) our parallel algorithm is practical and achieves good parallel efficiency, and 2) the partitioning choice that minimizes communication volume does result in significantly better performance than other partitioning choices.

9. REFERENCES


