Communication and Memory Optimal Parallel Data Cube Construction

Ruoming Jin Karthik Vaidyanathan Ge Yang Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{jinr,vaidyana,yangg,agrawal}@cse.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 in the aggregation tree 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. Finally, in the cases when memory may be a bottleneck, we describe how tiling can help scale sequential and parallel data cube construction.

Experimental results from implementation of our algorithms on a cluster of workstations show the effectiveness of our algorithms and validate our theoretical results.

Keywords: Data Warehouses, OLAP, Parallel Algorithms, Communication Analysis

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 [9]. 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_m^n m-dimensional arrays, for $0 \leq m \leq n$. Developing sequential algorithms for constructing data cubes is a well-studied

problem [16, 21, 19]. 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 [4, 7, 8].

This paper focuses on a number of algorithmic issues in parallel (and sequential) data cube construction. To motivate the issues we address, we discuss the problem of data cube construction in more details below.

1.1 Data Cube Construction

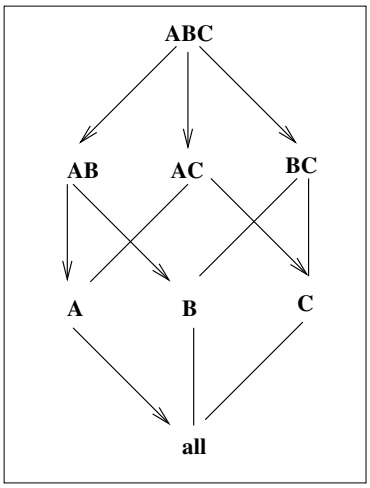

Figure 1: Lattice for data cube construction. Edges with arrows show the minimal spanning tree when $|A| \leq |B| \leq |C|$

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 list the four major issues that arise in data cube construction, using the example above.

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 refered 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.

1.2 Summary of Contributions

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 present a tiling based approach for further scaling sequential and parallel data cube construction, in the cases when the available main memory is not sufficient for holding intermediate results.
- We have implemented our parallel algorithm on a cluster of workstations. We present experimental results that validate our theoretical results on partitioning. We show the our algorithm achieves high parallel efficiency in most cases, with the only exception being sparse, high-dimensional datasets with small dimension sizes.

The rest of the paper is organized as follows. We briefly summarize the existing efforts in this area 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. In Section 6, we describe the use of tiling to scale data cube construction when memory requirements exceed available memory. Experimental results for evaluating our algorithms are presented in Section 7. We conclude in Section 8.

2 Related Work

Since Jim Gray [9] proposed the data cube operator, techniques for data cube construction have been extensively studied for both relational databases [16, 15] and multi-dimensional datasets [21, 19]. Our work belongs to the latter group. Zhao *et. al* [21] 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, our approach involves the use of aggregation tree to bound the total memory requirements, without requiring frequent writing to the disks. In addition, we have focused on parallelization, including ordering of dimensions and partitioning to minimize communication volume. Tam [19] uses MNST (Minimum Number Spanning Tree) to reduce computing cost, with ideas some-what similar to our prefix tree. However, this method also requires frequent writing 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* [7, 8] developed a framework Parsimony, which included support for data storage and data partitioning of multi-dimensional arrays, as well as algorithms for aggregations on dense and sparse chunks, as required for computing datacube and related operations in parallel. They did not suggest a particular algorithm for parallel data cube construction, instead, developed basic functionality required for implementation of any algorithm. In comparison, our work includes new data structures and algorithms, which have associated concrete results on minimizing memory requirements, communication volume, and partitioning.

Recently, Dehne *et. al* [4] 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, our work focuses on the shared-nothing model, which we believe is also more commonly used in practice. Their effort does not consider the memory requirements issue either.

There have also been extensive research on partial materialization of a data cube [14, 13, 11]. Although our current research has concentrated on complete data cube construction, we believe that the techniques we will present 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.

Similarly, many recent efforts have focused on making cube construction process more efficient, either by creating a compressed structure in main memory, computing a compressed cube, or both. An excellent summarization of this work is available from Feng *et al.* [5]. Some of the prominent efforts are as follows. The Bottom-Up Computation (BUC) approach especially exploits the sparsity of the data [1]. Han *et al.* have designed a data structure called the H-Tree, in which the input dataset is compressed by prefix sharing [10]. Sismanis *et al.* use CUBEtree to compress the full cube in memory by utilizing prefix sharing and suffix coalescing [17]. The Range CUBE approach exploits data correlation to reduce both the computation time and the output I/O time [5]. Our parallelization work currently only considers the base algorithms for data cube construction, where neither the input or output is compressed.

Many aspects of parallel data warehouses have been researched. Garcia-Molina *et al.* [6] and Datta *et al.* [3] initially made the case for supporting data warehouses on parallel environments. Stohr *et al.* considered the problem of data allocation in relational data warehouses which are based on star schema and reside on shared disk parallel systems [18]. In comparison, our focus is on multi-dimensional datasets, and shared nothing systems.

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 earlier. 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].

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

Definition 1 *$L(n)$ is a power set 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 $(r, s) \in E$.*

A prefix tree $P(n)$ is a spanning tree of the power set 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:*

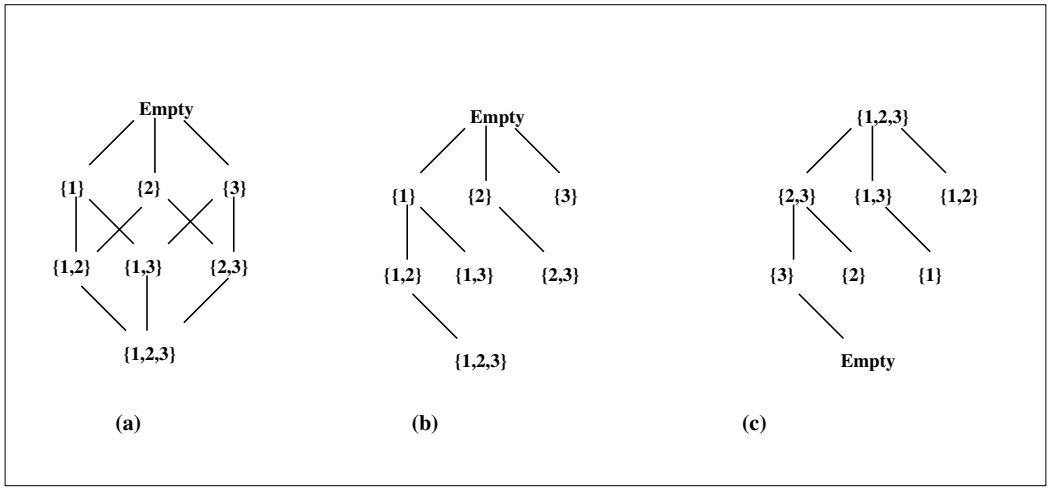

Figure 2: Power Set Lattice (a), Prefix Tree (b), and Aggregation Tree (c) for n = 3

(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 power set 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

```
Construct_Cube(D_1, D_2, ..., D_n)
{
  Evaluate({D_1, D_2, ..., 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

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*

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

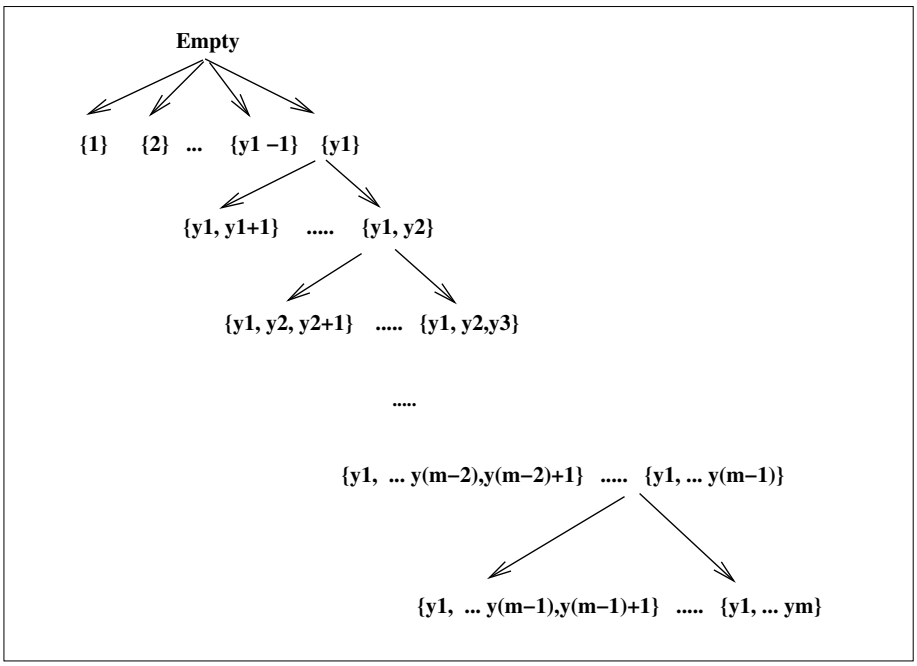

Figure 4: A Snapshot of the Prefix Tree

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 are 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

$$\prod_{j=1, j \neq y_1, y_2, \ldots, y_i, y_i+k}^{n} |D_j|$$

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

$$\sum_{i=i}^{y_1} \Big(\prod_{j=1, j \neq i}^{n} |D_j| \Big) + \sum_{i=y_1}^{y_2} \Big(\prod_{j=1, j \neq i, j \neq y_1}^{n} |D_j| \Big) + \ldots + \sum_{i=y_{m-1}}^{n} \Big(\prod_{j=1, j \neq i, j \neq y_1, y_2, \ldots, y_{m-1}}^{n} |D_j| \Big)$$

The above quantity is less than or equal to

$$\sum_{i=i}^{n}(\prod_{j=1,j\neq i}^{n} |D_j|)$$

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*

$$\sum_{i=1}^{n} (\prod_{j=1,j\neq i}^{n} |D_j|)$$

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

$$\sum_{i=1}^{n} (\prod_{j=1,j\neq i}^{n} |D_j|)$$

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

$$M = \sum_{i=1}^{n} (\prod_{j=1,j\neq i}^{n} |D_j|)$$

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 [21]. The second approach is based upon *tiling* [20]. 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 discussed in Section 6.

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

```
Construct_Cube(D_1, D_2, ..., D_n)
{
  Evaluate({D_1, D_2, ..., 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, ..., D_im}
    If the processor is the lead processor along D_i1, ..., 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 5: 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 $\sum_{i=1}^{n} k_i = p$. Each processor is given a unique label $\{l_1, l_2, \ldots, l_n\}$ such that $0 \leq l_i \leq 2^{k_i} - 1$. 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.

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. Figure 6 illustrates such partitioning of a three dimensional array. Initially, all 8 processors process the

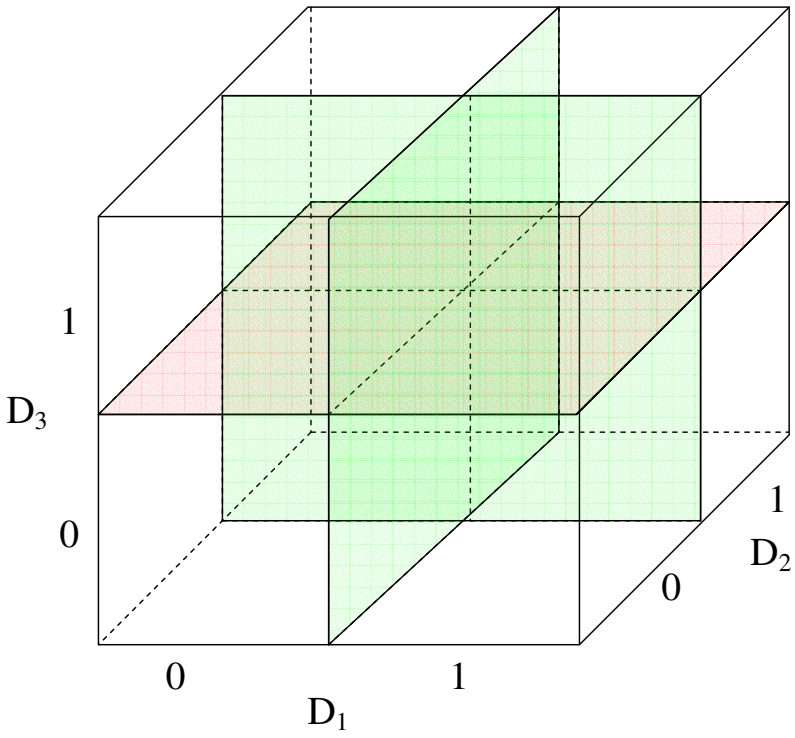

Figure 6: Partitioning of a Three Dimensional Array

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. Figure 7 shows the aggregations and communication steps involved in the algorithm.

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 is 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 [20] 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.

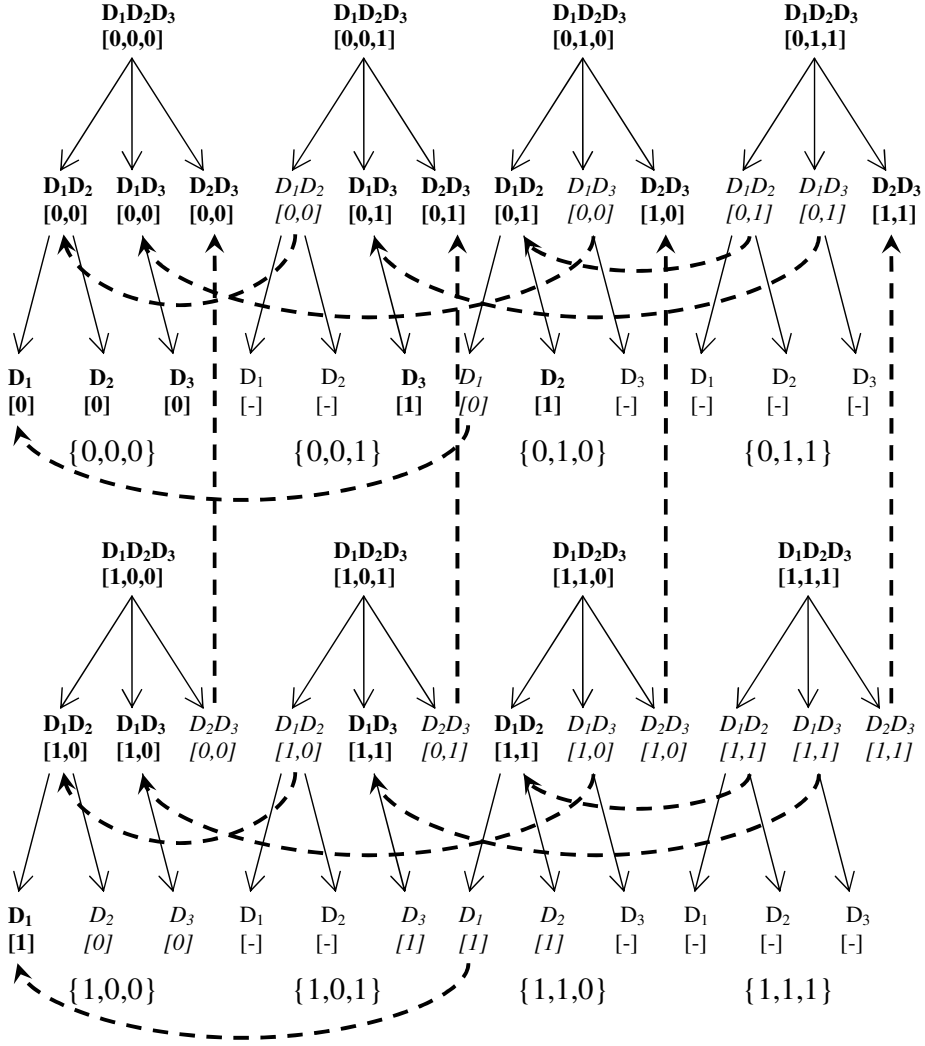

Figure 7: Aggregation and Communication Steps in Parallel Data Cube Construction (Dotted Arrows Denote Interprocessor Communication)

Lemma 1 *Consider a node* $r = \{y_1, y_2, \ldots, y_k\}$ *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*

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

Proof: The total size of the array at the node s' is the product of the sizes along each of its dimensions, which is given by $\prod_{i=1, i \neq y_1, y_2, \ldots, y_k, m}^{n} |D_i|$.

Let there be a total of N processors participating in the computation of the node s'. Among these,

$N/2^{k_m}$ will be the lead processors that will communicate with other processors. Each lead processor will compute a fraction $\frac{2^{k_m}}{N}$ of the array s'. The volume of data that each lead node will receive from other nodes will be

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

.

Since there are $N/2^{k_m}$ lead nodes receiving the data, the total volume of communication will be

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

.

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

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

Proof: Consider the level l in the aggregation tree. There are ${}^nC_l$ nodes at this level. We denote the total communication volume for computing these ${}^nC_l$ nodes by $Comm(l)$. Using the previous lemma, we have

$$Comm(l) = \sum_{1 \leq y_1 < y_2 < \ldots y_l} (\prod_{i=1, i \neq y_1, y_2, \ldots, y_l}^{n} |D_i|) \times (2^{k_{y_l}} - 1)$$

This can also be written as

$$Comm(l) = \prod_{i=1}^{n} |D_i| \times (\sum_{1 \leq y_1 < y_2 < \ldots y_l} \frac{1}{|D_{y_1}| \times |D_{y_2}| \times \ldots \times |D_{y_l}|} \times (2^{k_{y_l}} - 1))$$

The total communication volume for the entire data cube construction process is

$$\sum_{l=1}^{n} Comm(l)$$

Using the expression for $Comm(l)$ and with some mathematical manipulation, we get the above result.

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^{k_i} 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 the algorithm in Figure 5 are bounded by*

$$\frac{\prod_{i=1}^{n} |D_i|}{2^p} \times (\sum_{i=1}^{n} \frac{2^{k_i}}{|D_i|})$$

Proof: The proof is quite similar to the proof of Theorem 1. The details are presented in a related technical report [12].

Theorem 5 *The memory requirements on any processor for holding the results during parallel data cube construction using any spanning tree and algorithm are at least*

$$\frac{\prod_{i=1}^{n} |D_i|}{2^p} \times (\sum_{i=1}^{n} \frac{2^{k_i}}{|D_i|})$$

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

Proof: Again, the proof is similar to the proof of Theorem 2 and the details are presented in a technical report [12].

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 $|D_1| \geq |D_2| \geq \ldots \geq |D_n|$.*

Proof: The communication volume required for data cube construction can also be stated as

$$c_0 \times (\sum_{i=1}^{n} \frac{c_i}{|D_i|} \times (\prod_{j=1}^{i-1} (1 + \frac{1}{|D_j|})))$$

Consider an ordering of the dimensions $D_1, D_2, \ldots, D_n$, where $p < q$ and $|D_p| < |D_q|$. Let $C1$ be the communication cost associated with this ordering of the dimensions. Consider another ordering of the dimensions,

$D_1, D_2, \ldots, D_{p-1}, D_q, D_{p+1}, \ldots, D_{q-1}, D_p, \ldots, D_n$. Let $C2$ be the cost associated with this ordering.

We will show that $C1 - C2 > 0$. We ignore the constant term c_0 in our arguments. So, we have

$$C1 = \sum_{i=1}^{n} \frac{c_i}{|D_i|} \times (\prod_{j=1}^{i-1}(1 + \frac{1}{|D_j|}))$$

and

$$C2 = \sum_{i=1}^{p-1} \frac{c_i}{|D_i|} \times (\prod_{j=1}^{i-1}(1 + \frac{1}{|D_j|})) + \sum_{i=q+1}^{n} \frac{c_i}{|D_i|} \times (\prod_{j=1}^{i-1}(1 + \frac{1}{|D_j|})) + \frac{c_q}{|D_q|} \times \prod_{j=1}^{p-1}(1 + \frac{1}{|D_j|}) +$$

$$\frac{c_p}{|D_p|} \times \prod_{j=1, j\neq p}^{q-1}(1 + \frac{1}{|D_j|}) \times (1 + \frac{1}{|D_q|}) + \sum_{i=p+1}^{q-1} \frac{c_i}{|D_i|} \times (\prod_{j=1, j\neq p}^{i-1}(1 + \frac{1}{|D_j|})) \times (1 + \frac{1}{|D_q|})$$

With some mathematical manipulation, it can be shown that

$$C1 - C2 = d \times (\frac{1}{|D_p|} - \frac{1}{|D_q|})$$

where d is a positive constant. Since $|D_p| < |D_q|$, we have $C1 - C2 > 0$. This shows that ordering the dimensions in descending order of the sizes minimizes the communication volume.

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

Proof: Consider a node in the prefix lattice defined in Section 3. Let this node be $\{y_1, y_2, \ldots, y_k\}$, where $1 \leq y_1 < y_2 < \ldots < y_k \leq n$.

This node has k parents in the prefix lattice, which are denoted by $p(j),\ j = 1, \ldots, k$, where

$$p(j) = \{y_i | i = 1, \ldots, k, i \neq j\}$$

.

The computation cost in using the parent $p(j)$ is

$$\frac{\prod_{i=1}^{n} |D_i|}{\prod_{i=1, i\neq j}^{k} |D_{y_i}|}$$

This expression is minimized if $|D_{y_j}|$ has the smallest value among $|D_{y_i}|, i = 1, \ldots, k$. In using prefix/aggregation tree, we always use $p(k)$ for evaluating this node. If $|D_1| \geq |D_2| \geq \ldots \geq |D_n|$, then $|D_{y_k}|$ has the smallest value among $|D_{y_i}|, i = 1, \ldots, k$.

We prove the *only if* part using contradiction. Suppose we have a dimension ordering $D_1, D_2, \ldots, D_n$, where $p < q$ and $|D_p| < |D_q|$. Consider the node $\{D_1, D_2, \ldots, D_q\}$ in the prefix lattice. Using prefix/aggregation tree, this node will computed from $\{D_1, D_2, \ldots, D_{q-1}\}$. However, since $|D_p| < |D_q|$, $\{D_1, D_2, \ldots, D_{p-1}, D_{p+1}, \ldots, D_q\}$ will be a lower cost parent.

The next issue we focus on it 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 ${}^{n+p}C_n$ 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

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

This can be restated as

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

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

$$c_0 \times (\sum_{i=1}^{n} 2^{k_i} \times X_i) - d_0$$

where,

$$X_i = \frac{1}{|D_i|} \times (\prod_{j=1}^{i-1} (1 + \frac{1}{|D_j|}))$$

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

The algorithm is presented in Figure 8. Initially, k_i, 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 $2 \times X_i$.

```
Partition(n, p, X_1, X_2, ..., X_n)
{
  Initialize k_1 = k_2 = ... = k_n = 0
  While (p > 0) {
    Let X_i = min(X_1, X_2, ..., X_n)
    k_i = k_i + 1
    X_i = 2 × X_i
    p = p - 1
  }
}
```

Figure 8: Partitioning Different Dimensions to Minimize Communication Volume

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

The proof of this theorem is presented in an associated technical report [12].

6 Tiling-Based Approach for Scaling Data Cube Construction

The sequential and parallel algorithm we have presented so far assume that sufficient memory is available to store all arrays at the first level in memory. In general, this assumption may not hold true. In this section, we present sequential and parallel algorithms that use tiling to scale data cube construction.

6.1 Sequential Tiling-Based Algorithm

Let the initial multidimensional array from which a data cube is constructed be denoted by $D_1 D_2 \ldots D_n$. We tile this array, dividing each dimension D_i into t_i tiles, creating a total of $\prod_{i=1}^{n} t_i$ tiles. Suppose we are computing a partial or complete data cube using a given aggregation tree. Consider any node N of the tree $D_{x_1} \ldots D_{x_{m-1}}$, where $1 \leq x_i \leq n$ and $m < n$. Let the parent of this node in the tree be $D_{x'_1} \ldots D_{x'_m}$, where

$$\{x'_1, \ldots x'_m\} = \{y\} \cup \{x_1, \ldots, x_{m-1}\}$$

Thus, the node N is computed from its parent by aggregating along the dimension y.

The array $D_{x_1} \ldots D_{x_{m-1}}$ computed at the node N comprises $t_{x_1} \times \ldots t_{x_{m-1}}$ tiles. For scaling the computations of views, we can separately read and write these portions from and to disks. A particular tile of this array is denoted by a tuple $< p_{x_1}, \ldots p_{x_{m-1}} >$, where $1 \leq p_{x_i} \leq t_{x_i}$.

Dividing each array into tiles adds a new complexity to the process of computing these arrays. A given tile $< p_{x_1}, \ldots, p_{x_{m-1}} >$ of the node N is computed using t_y different tiles of its parent. This is because the dimension y, which is aggregated along to compute N from its parent, is divided into t_y tiles. Since the different tiles comprising the parent array of N can be allocated in the memory only one at a time, a tile of the node N may have to be computed in t_y phases. In each of these phases, one tile of the parent of N is processed and the corresponding elements in N are updated.

Note that a node can have multiple children in the tree. To ensure high memory and cache reuse, when a tile of an array is brought into memory, we update the corresponding tiles of all children of that node. Since these children are computed by aggregating along different dimensions, it is not possible to read all tiles that are used to compute one tile of a child node consecutively. As a result, a tile of a node being computed may have to be written and reread from the disks as it is computed from multiple tiles of its parent node.

To facilitate correct computations using tiling, we associate a table with each node of the tree. For the node N described above, this table is an array with $m - 1$ dimensions,

$N.Table[1 \ldots t_{x_1}, 1 \ldots t_{x_2}, \ldots, 1 \ldots t_{x_{m-1}}]$. An element $Table(< p_{x_1}, \ldots, p_{x_{m-1}} >)$ has a value between 0 and t_y and denotes the status of the tile $< p_{x_1}, \ldots, p_{x_{m-1}} >$. A value of 0 means that this tile is currently uninitialized. A value i, $0 < i \leq t_y$ means that the elements of this tile have been updated using i tiles of the parent node. If the value is t_y, then the elements in this tile have received their final values. In this case, we say that the tile is *expandable*, because it can now be used for starting the computation of its children nodes.

```
Construct_Views(D_1 D_2 ... D_n)
{
  Foreach tile T of this node
    Expand_tile(D_1 ... D_n, T)
}

Expand_tile(Node N, Tile T)
{
  Foreach child C of N in the tree {
    T' = Maptile(N, T, C)
    C.Table(T')++
    If C.Table(T') == 1
      Allocate and initialize the tile T'
    Else
      Read the tile T' from disk if required
  }
  Foreach chunk of the tile T {
    Read the chunk
    Foreach child C of N
      Perform aggregation operations on the tile Maptile(N, T, C)
  }
  Foreach child C of N {
    T' = Maptile(N, T, C)
    If (C.Table(T') == Reduc_tiles(C))
      Expand_tile(C, T')
    Else
      Write-back the tile T' to disk if required
  }
  If N is not root
    Write-back T to disk
}
```

Figure 9: A Tiling-Based Algorithm for Constructing Data Cube

The tiling-based algorithm is presented in Figure 9. We assume that the original array is indexed in such a way that each tile can be retrieved easily. In the algorithm, $Maptile(N, T, C)$ is the tile of C which can be updated using the tile T of N, where N is a given node, T is a tile of this node and C is a child of this node. $Reduc\_tiles(N)$ is the number of tiles of the parent of N along the dimension that is aggregated to compute N, where N is a non-root node.

The function *Expand_tile* takes a tile and a node of the tree, and computes or updates the appropriate portions of the descendants of the tree. Given a node N and a tile T, we find the tiles of the children of N that can be updated using the function $Maptile(N, T, C)$. We then use the $Table$ data structure to

determine the status of the tiles of children. If they have not yet been initialized, we allocate space and initialize them. If they have been updated previously, they may have to be read from the disks. Once a chunk corresponding to a parent node is brought into memory and cache, all children are updated together.

We next check if the corresponding tile of a child node has been completely updated (i.e. if $(C.Table(T') == Reduc\_tiles(C))$). If so, we expand its children before writing it back to the disks.

Thus, our algorithm ensures that once a tile is in memory, we update all its children simultaneously, and further expand upon the children if possible. In the process, however, a tile of child node may have to be written back and read multiple times. We prefer to ensure high memory and disk reuse of the parent tiles to some possible extent for two important reasons. First, the sizes of arrays decrease as we go down the tree, so it is preferable to write back and read lower level nodes in the tree. Second, if the original array is partitioned along only a few dimensions, *Reduc_tiles* will have the value of one for many nodes in the tree. In this case, the node being computed will not need to be written back and read multiple times.

6.2 Using Tiling in Parallel Data Cube Construction

While applying our tiling-based algorithm to parallel construction of data cubes, we should note that we have two kinds of partitions of a node in the aggregation tree. The first is due to the data distribution among multiple processors. Since each processor has a portion of the original array, interprocessor communication is needed to get final values of this node. The second is due to tiling. The portion on each processor is divided into several tiles and the final values can not be obtained until all tiles of the node are aggregated.

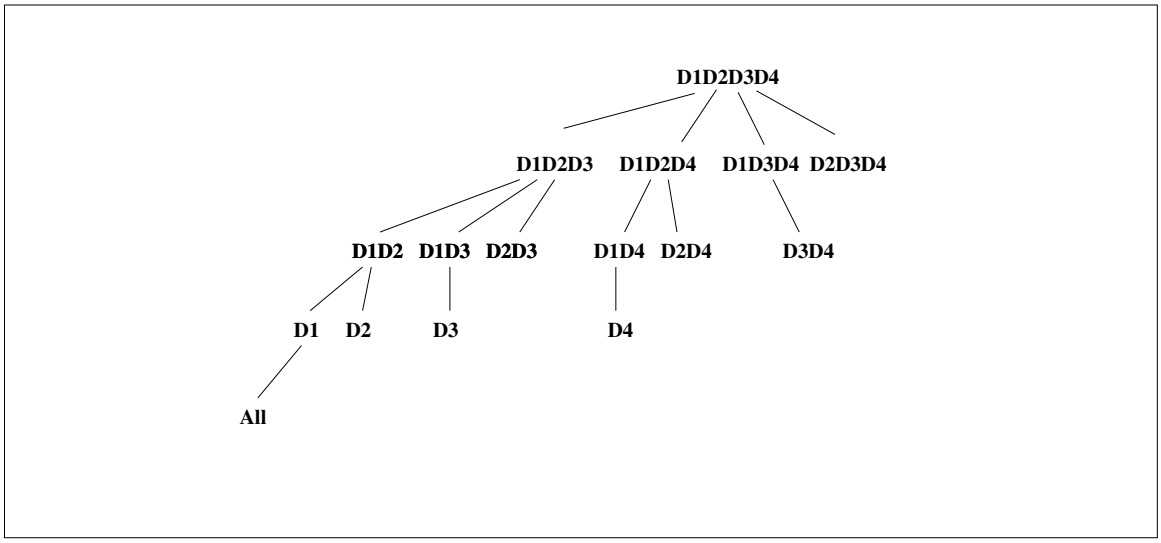

Figure 10: Aggregation Tree for Four-dimensional Data Cube Construction

The existence of these two different kinds of partitions adds complexity in deciding whether or not a node is ready for computing its children. For example, consider constructing a data cube from an original four-dimensional array $D_1D_2D_3D_4$ on 8 processors. The aggregation tree is shown in Figure 10. Using the terminology used in the previous section, we do a three-dimensional partition for the original array, which means dimensions D_2, D_3, D_4 are partitioned along two processors. Then on each processor, we divide the $\frac{1}{8^{th}}$ portion of the array on this processor into 4 tiles by tiling along dimensions D_3 and D_4 in half respectively. According to the tiling-based algorithm we introduced above, after computing each tile of the node $D_1D_3D_4$, the tile becomes expandable since $D_1D_3D_4$ is obtained by aggregating along dimension D_2 and D_2 is not tiled. But it is not the case since D_2 is partitioned along two processors and we have to do the interprocessor communication to get the final values of $D_1D_3D_4$ before we can compute its child. Therefore, we can not apply the tiling-based algorithm directly to parallel data cube construction on multiple processors.

A solution to this problem is to apply tiling-based algorithm only to the children of the root node in the aggregation tree. For the computation of other nodes, we follow a similar process as we had presented in the previous section. Considering that the dominant part of computation is at the first level for multidimensional data cube construction, we believe that tiling all nodes at the first level can reduce memory requirements. For simplicity, we use Level One Parallel Algorithm for the computation of lower levels nodes in the aggregation tree. The complete algorithm is shown in Figure 11. In this algorithm, $C.T'$ stands for a tile of values of child C. Other notations have the same meaning as in Figures 9 and 5.

Compared with the sequential tiling-based algorithm in Figure 9, we apply the sequential tiling-based algorithm only to the children of the root node. In addition, we do not expand the node even when $C.Table(T') == Reduc\_tiles(C)$. (Actually, we do not check whether $C.Table(T') == Reduc\_tiles(C)$ at all.) We write back every $T' = Maptile(D_1D_2 \ldots D_n, T, C)$ to the disk and after all tiles are processed, each child has t_{total}/t_y tiles of values, where y is the dimension along which the child is computed by aggregating its parent, t_y is the number of tiles of dimension y and t_{total} is the total number of tiles of the original array.

As we have mentioned earlier, we can not get the final values of children of the root node until we do interprocessor communication. Therefore, we follow a similar procedure as in Level One Parallel Algorithm to finalize each tile of values of the children. The difference is that we first do the required interprocessor communication to get the final values of the child, and then we aggregate its children. Note that we do not use optimized Level One Parallel Algorithm since memory requirement is our key consideration here.

We use the same example we mentioned at the beginning of this section to describe how this algorithm

```
Construct_Cube(D_1 D_2 ... D_n)
{
  Foreach tile T of the root node D_1 D_2 ... D_n on each processor{
    Foreach child C of D_1 D_2 ... D_n in the tree {
      T' = Maptile(D_1 D_2 ... D_n, T, C)
      C.Table(T')++
      If C.Table(T') == 1
        Allocate and initialize the tile T'
      Else
        Read the tile T' from disk if required
    }
    Foreach chunk of the tile T {
      Read the chunk
      Foreach child C of D_1 D_2 ... D_n
        Perform aggregation operations on the tile Maptile(D_1 D_2 ... D_n, T, C)
    }
    Write-back the tile Maptile(D_1 D_2 ... D_n, T, C) to disk if required
  }
  Foreach child C of D_1 D_2 ... D_n from right to left
    Foreach tile T' of C {
      Read C.T' from disk if required
      Evaluate(C.T') on each processor
    }
}

Evaluate(C)
{
  Let C' = X - C = {D_i1, ..., D_im}
  If the processor is the lead processor along D_i1, ..., D_im
    Communicate with other processors to finalize portion of C if required
    If C has no children
      Write-back the portion to disk if required
    Else
      Locally aggregate all children of C
      Foreach child r from right to left
        Evaluate(r)
  Write-back C to disk if required
}
```

Figure 11: A Tiling-Based Algorithm for Parallel Data Cube Construction

works for parallel data cube construction. We consider three-dimensional partition of the original array, which means dimensions D_2, D_3, D_4 are partitioned along two processors. Then on each processor, we divide the $\frac{1}{8^{th}}$ portion of array on this processor into 4 tiles by tiling along dimensions D_3 and D_4 in half respectively.

After processing all 4 tiles of $D_1D_2D_3D_4$, each child of $D_1D_2D_3D_4$ has tiles of values stored on each processor. For instance, $D_2D_3D_4$ and $D_1D_3D_4$ each has 4 tiles of values, $D_1D_2D_4$ and $D_1D_2D_3$ each has 2 tiles of values. We then consider each tile of $D_2D_3D_4$. Since $D_2D_3D_4$ is computed by aggregating along dimension D_1 which is not partitioned, we do not need to do interprocessor communication and each processor already has a tile of final values of $D_2D_3D_4$. $D_2D_3D_4$ also has no child, therefore, it is done and can be written back to the disks.

We now consider the first of the 4 tiles of $D_1D_3D_4$. Since D_2 is partitioned in half, interprocessor communication is needed to get the final values of the tile of $D_1D_3D_4$. The communication process is the same as in the parallel algorithm we presented originally. After final values are obtained on lead processors, we compute D_3D_4 from final values of this tile of $D_1D_3D_4$. Since there is no need to communicate for D_3D_4 and D_3D_4 has no child, we are done with the first tile of $D_1D_3D_4$. For the other three tiles of $D_1D_3D_4$, we follow the same procedure as above.

The computation of each tile of $D_1D_2D_4$ and $D_1D_2D_3$ can be proceeded in a similar fashion, except that we must pay attention to the fact that some offsprings of $D_1D_2D_4$ and $D_1D_2D_3$, such as D_1D_4, D_1D_3, D_1D_2 and D_1, also need interprocessor communication to get final values.

Note that the number of tiles of each child below the first level in the aggregation tree is decided by the number of tiles of its parent. For example, since $D_1D_3D_4$ has 4 tiles of values, D_3D_4 also has 4 tiles of values. In contrast, each offspring of $D_1D_2D_4$ and $D_1D_2D_3$ has only 2 tiles. The number of tiles of children at the first level is determined by t_{total}/t_y, as we have stated earlier.

7 Experimental Results

This section reports on a series of experiments we conducted to evaluate our techniques and algorithms. We had the following three goals in designing our experiments. First, we wanted to see the speedups from our algorithms, across datasets with varying sizes and varying number of dimensions. Second, we wanted to see if the versions with partitioning that minimizes communication volume does achieve better performance than versions with other partitioning choices. Finally, we wanted to see how tiling impacts sequential and parallel scalability.

In constructing data cubes, the initial multi-dimensional array can be stored in a dense format or a sparse format [21]. 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 [21]. 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.

Through-out this paper, our results have been presented assuming that the initial array from which the data cube is constructed is dense. If the initial array is sparse, the memory requirements for storing the results and the communication volume do not change. Therefore, our results on bounded memory requirements and communication volume remain the same even when the initial array is sparse. The only difference comes in the computation cost for computing the first-level results (i.e. the $n-1$ dimensional

arrays). However, since the first-level nodes only have a single parent, our result on dimensional ordering for minimal parents is still applicable.

Since sparse formats are frequently used in data warehouses, most of 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. Note that an array with a numerically lower sparsity value is more sparse than the one with a numerically higher sparsity value. We have experimented with different levels of sparsity in this paper.

7.1 Parallel Scalability and Impact of Partitioning

We initially present and analyze results from a set of relatively small datasets. Later, we present results from larger and higher-dimensional datasets.

7.1.1 Results from Smaller Datasets

These 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. 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.

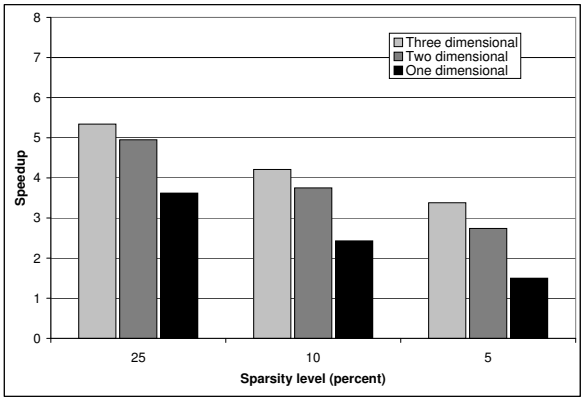

Figure 12: Results on 64^4 datasets, 8 processors

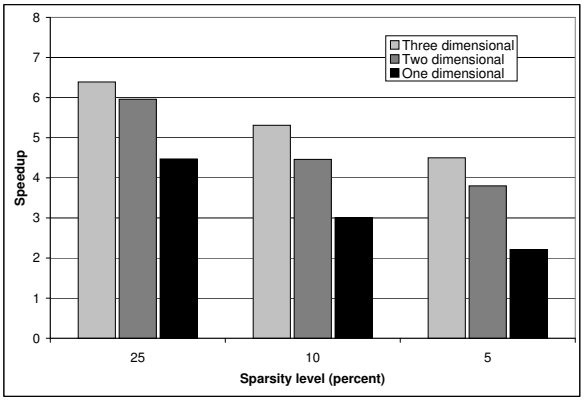

Figure 13: Results on 128^4 datasets, 8 processors

The first set of experimental results are obtained from $64 \times 64 \times 64 \times 64$ datasets. We experimented with three different levels of sparsity, 25%, 10%, and 5%. The results on 8 processors are presented in Figure 12. 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. The sequential execution times were 22.5, 12.4, and 8.6 seconds, with sparsity levels of 25%, 10%, and 5%, respectively.

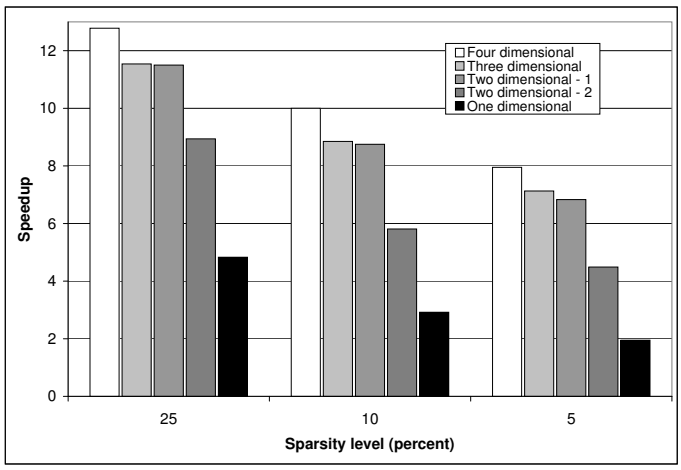

Figure 14: Results on 128^4 datasets, 16 processors

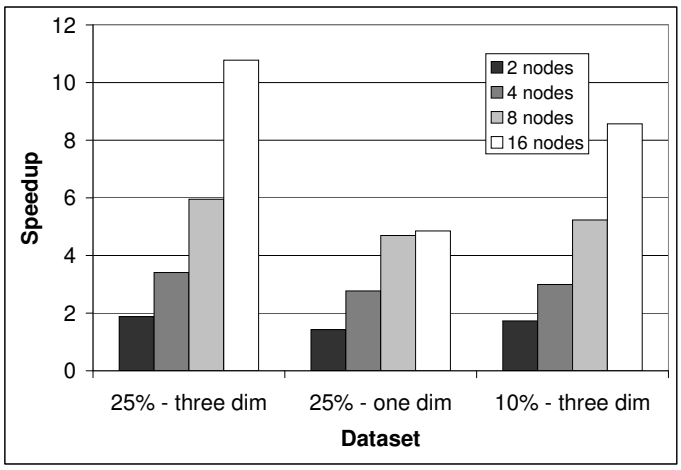

Figure 15: Parallel Scalability on 128^4 datasets, Up to 16 processors

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 12 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.

It should be noted that as the arrays become more sparse, the absolute speedups decrease, but the relative difference between the communication optimal version and the other versions increases. The reason is as follows. For a given problem size, a sparse array involves less computation at the first level of the tree. Because the aggregated (and partially aggregated) arrays are stored in the dense format, the communication volume remains unchanged. Therefore, the more sparse the initial array is, the higher is the ratio between communication and computation.

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 of 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. Furthermore, this component is parallelized on all nodes, and does not have any overheads besides the communication costs. Therefore, we can conclude that: 1) the communication costs are the only significant factor why the speedups are not linear, and 2) the difference in performance seen as a result of the partitioning

choice made is almost all 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 13 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%, respectively. The speedups reported here are higher because of the larger dataset size, 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 14.

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.

Finally, in Figure 15, we show scalability of our algorithm on 1, 2, 4, 8, and 16 processors. We have considered two different partitioning schemes for the 25% sparsity dataset, and one partitioning scheme for the 10% sparsity dataset. The results are consistent with those from other experiments.

7.1.2 Results from Larger and Higher-Dimensional Datasets

We also conducted a series of experiments on larger and higher-dimensional datasets. These experiments were performed on a cluster of 700 MHz Pentium machines. The nodes in the cluster were connected through Myrinet LANai 9.0. The memory on each node is 1GB.

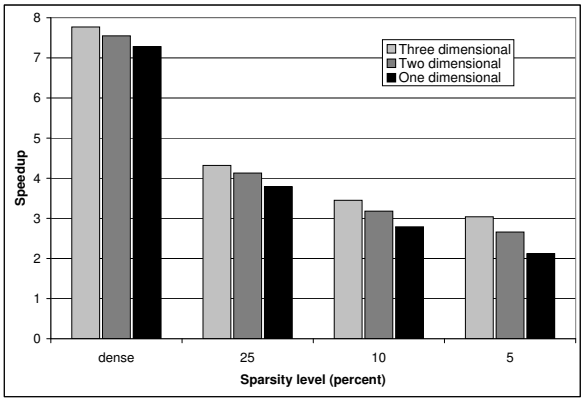

Figure 16: Results on 16^6 datasets, 8 processors

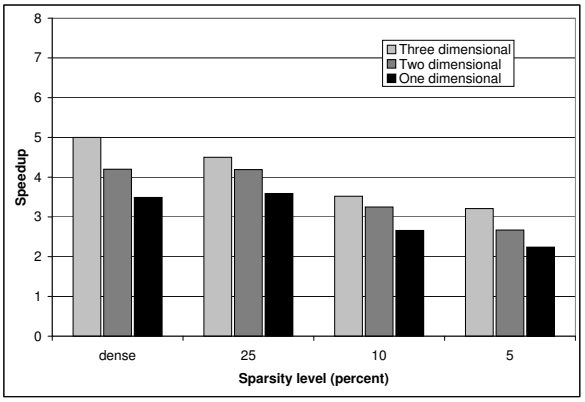

Figure 17: Results on 8^8 datasets, 8 processors

Figure 16 shows the results from 16^6 datasets. We report performance from sparse datasets with sparsity levels of 25%, 10%, and 5%, as well as the dense dataset. We compare three, two, and one dimensional partitioning schemes. The performance trends are similar to what we reported earlier. Three dimensional partitioning schemes, which have the lowest communication volume as per our theoretical results, give the best speedups. One 8 nodes, the speedup on the dense dataset with the three dimensional partitioning scheme is 7.77. The speedups are relatively modest on sparse datasets. The highest speedups on 8 nodes are 4.32, 3.45, 3.04, respectively, with 25%, 10%, and 5% sparsity levels. Smaller size of dimensions, higher number of dimensions, and higher sparsity levels all result in a higher communication to computation ratio and a higher fraction of sequentialized code.

Figure 17 shows the results from 8^8 datasets. The trends are again similar, though a higher number of dimensions and smaller dimension size results in lower speedups.

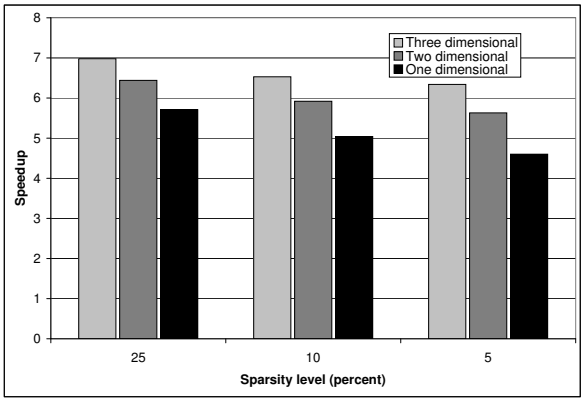

Figure 18: Results on 32^6 datasets, 8 processors

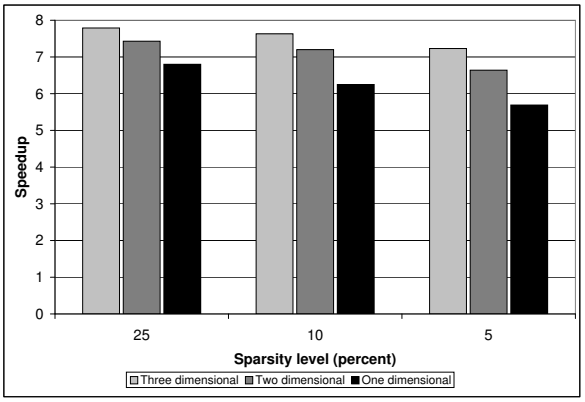

Figure 19: Results on 256^4 datasets, 8 processors

The results from 32^6 datasets are reported in Figure 18. The dense dataset with these dimensions has a size of 4 GB, and could not be executed on our environment. Therefore, we only report results from three sparse datasets. As compared to the results on the 16^6 dataset described earlier, the speedups of

sparse datasets are clearly better. This is because of a lower communication to computation ratio. The relative performance of the versions with different partitioning methods is still the same.

Finally, the results from 256^4 datasets are presented in Figure 19. Because of a small number of dimensions and large size of each dimension, the speedups with even the 5% sparse dataset are quite high.

7.2 Impact of Tiling

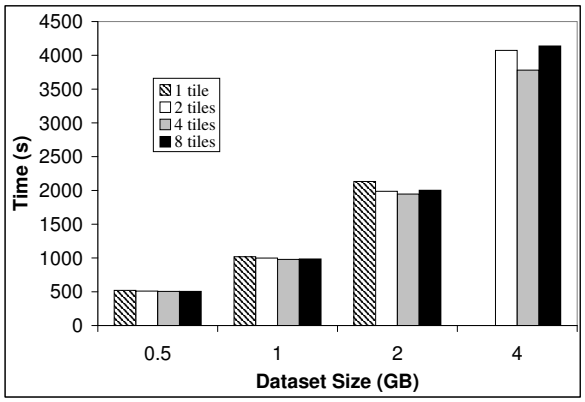

Figure 20: Scaling Sequential Data Cube Construction with Tiling (8 dimensional datasets)

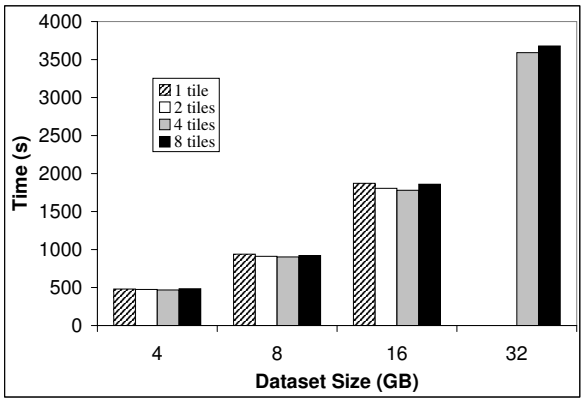

Figure 21: Scaling Parallel Data Cube Construction with Tiling (9 dimensional datasets on 8 nodes)

We conducted several experiments to see the benefits from tiling. We present results showing how tiling helps scale data cube construction in both sequential and parallel environments.

Our experiments for sequential execution were conducted on a machine with 1 GB memory. We used 4 8-dimensional datasets, which were dense arrays with sizes $8^5 \times 16^3$, $8^4 \times 16^4$, $8^3 \times 16^5$, $8^2 \times 16^6$, respectively. As each element requires 4 bytes, the sizes of these datasets are .5, 1, 2, and 4 GB, respectively. Without the use of tiling, the total memory required for the first level of the tree is 416 MB, 768 MB, 1.4 GB, and 2.5 GB, respectively.

The execution time with 1, 2, 4, and 8 tiles for these 4 datasets are presented in Figure 20. For the .5 GB and 1 GB datasets, sufficient memory was available to execute the algorithm without tiling (or using a single tile). The execution time for these datasets remains approximately the same with the use of 1, 2, 4, or 8 tiles. As all data can fit in main memory, the read and write operations for tiles only involve accessing main memory buffers, and therefore, use of large number of tiles does not result in a slow down.

A more interesting trend is noted with the 2 GB dataset. The use of 2 or 4 tiles results in lower execution time than the use of 1 or 8 tiles. With only 1 tile, memory thrashing causes the overhead. With the use of 8 tiles, the high tiling overhead causes the slow down. As the total memory requirements are

large, read and write operations for tiles now require disk accesses. Therefore, the use of larger number of tiles is not desirable.

With the 4 GB dataset, the code cannot even be executed with the use of a single tile. The lowest execution time is seen with the use of 4 tiles. Memory thrashing and tiling overheads are the reasons for slow down with 2 and 8 tiles, respectively. Note, however, because the execution times are dominated by computation, the relative differences are never very large.

We repeated a similar experiment for parallel data cube construction, using a 8 node cluster. We used four 9-dimensional datasets whose size were 4 GB, 8 GB, 16 GB, and 32 GB, respectively. After data partitioning, the size of the array portion on each node was .5 GB, 1 GB, 2 GB, and 4 GB, respectively, similar to the previous experiment. The results are presented in Figure 21 and are similar to the previous set of results. Note that there is some increase in per node memory requirements, because memory is needed for communication buffers. Therefore, with the largest dataset, a minimum of 4 tiles are required to complete execution.

Another observation from Figures 20 and 21 is as follows. As we experiment with larger input datasets, the execution time remains proportional to the amount of computation on each node. Thus, the use of tiling and parallelism helps scale data cube construction.

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 requirements. 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 in most cases, with the exception being sparse, high-dimensional datasets with small dimension sizes, and 2) the partitioning choice that minimizes communication volume does result in significantly better performance than other partitioning choices.

References

[1] Kevin Beyer and Raghu Ramakrishnan. Bottom up computation of sparse and iceberg cubes. In *Proceedings of ACM SIGMOD*, pages 359–370, 1999.

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

[3] A. Datta, B. Moon, and H. Thomas. A Case for Parallelism in Data Warehousing. In *Proceedings of DEXA-98*, 1998.

[4] 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.

[5] Y. Feng, D. Agrawal, A. E. Abbadi, and A. Metwally. Range CUBE: Efficient Cube Computation bu Expoliting Data Correlation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, March 2004.

[6] H. Garcia-Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge. Distributed and Parallel Computing in Data Warehousing (INvited Talk). In *Proceedings of the Conference on Principles of Database Systems (PODS)*, June 1998.

[7] 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.

[8] 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.

[9] 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.

[10] J. Han, J. Pei, G. Dong, and K. Wang. Efficient Computation of Iceberg Cube with Complex Measures. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[11] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD Conference*, Montreal, Canada, June 1996.

[12] Ruoming Jin, Karthik Vaidyanathan, Ge Yang, and Gagan Agrawal. Communication and memory optimal parallel data cube construction. Technical Report OSU-CISRC-9/04-TR50, Department of Computer Science and Engineering, The Ohio State University, September 2004.

[13] Alon Y. Levy. Answering queries using views: A survey. In *http://www.cs.washington.edu/homes/alon/site/files/view-survey.ps, 2000.*

[14] Chen Li, Mayank Bawa, and Jeffrey D. Ullman. Minimizing view sets without losing query-answering power. In *the 8th International Conference on Database Theory (ICDT)*, London, UK, January 2001.

[15] 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.

[16] 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.

[17] Y. Sismanis, A. Deligiannakis, N. Roussopolous, and Y. Kotidis. Dwarf: Shrinking the Petacube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2002.

[18] T. Stohr, H. Martens, and E. Rahm. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *Proceedings of Conference on Very Large DataBases (VLDB)*, August 2000.

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

[20] 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.

[21] 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.