# Using Tiling to Scale Parallel Data Cube Construction

Ruoming Jin Karthik Vaidyanathan Ge Yang Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{jinr,vaidyana,yangg,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. Also, for both sequential and parallel data cube construction, effectively using the main memory is an important challenge.

In our prior work, we have developed parallel algorithms for this problem. In this paper, we show how sequential and parallel data cube construction algorithms can be further scaled to handle larger problems, when the memory requirements could be a constraint. This is done by *tiling* the input and output arrays on each node. We address the challenges in using tiling while still maintaining the other desired properties of a data cube construction algorithm, which are, using minimal parents, and achieving maximal cache and memory reuse. We present a parallel algorithm that combines tiling with interprocessor communication. We present theoretical analysis showing how tiling should be performed to minimize the overhead.

Our experimental results show the following. First, tiling helps in scaling data cube construction in both sequential and parallel environments. Second, choosing tiling parameters as per our theoretical results does result in better performance.

## 1 Introduction

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

Computing multiple related group-bys and aggregates is one of the core operations in OLAP applications [1]. Jim Gray has proposed the *cube* operator, which computes group-by aggregations over all possible subsets of the specified dimensions [6]. When datasets are stored as (possibly sparse) arrays, data cube construction involves computing aggregates for all values across all possible subsets of dimensions. If the original (or *initial*) dataset is an n-dimensional array, the data cube includes $C^n_m$ m-dimensional arrays, for $0 \leq m \leq n$. Developing sequential algorithms for constructing data cubes is a well-studied problem [9, 8, 10, 11].

Data cube construction is a compute and data intensive problem. Therefore, it is natural to use parallel computers for data cube construction. Recently, many parallel algorithms for data cube construction have been proposed [3, 4, 5, 7].

Both the input and output datasets for data cube construction can be extremely large. The problem of scaling data cube construction for large input datasets has been addressed by the early work in this area [11]. The basic idea to is to divide the input dataset into *chunks*, read one chunk at a time, and update corresponding output values. However, only limited effort has been put on scaling data cube construction for large output datasets. In practice, data cubes are constructed starting from an array that is both sparse and high-dimensional. This can result in output sizes that are larger than the input size. For example, the first level of data cube construction starting from a (potentially sparse) $n^m$ dataset results in $m$ dense arrays of size $n^{m-1}$. Depending upon the relative values of $n$, $m$, and the sparsity of the input array, the output at the first level can easily be larger than the size of the input array.

This paper develops a general approach for scaling data cube construction. We tile input and output arrays and only one tile of an output array is typically allocated and updated at any given time. Using tiling, however, involves a number of challenges. In this paper, we present algorithms and analysis addressing the following:

- How do we incorporate tiling of output arrays in a sequential data cube construction, while preserving other desirable properties, i.e., using minimal parents and maximizing cache and memory reuse ?
- Given a memory constraint, how do we tile to both meet the memory constraint and minimize the overhead associated with tiling, i.e., the cost of rereading and rewriting tiles ?
- How do we develop a parallel algorithm that combines tiling with interprocessor communication, while minimizing both communication volume and tiling overhead ?

We have implemented and evaluated our algorithms. The main observations from our experiments are as follows. First, tiling helps scaling data cube construction in both sequential and parallel environment. Second, choosing tiling parameters as per our theoretical results does result in better performance.

The rest of the paper is organized as follows. We further discuss the data cube construction problem in Section 2. A data-structure we introduced in our prior work, the aggregation tree, is reviewed in Section 3. Parallel data cube construction algorithm (without tiling) is presented in Section 4. Tiling based sequential and parallel algorithm are introduced in Section 5. Analysis for minimizing tiling overheads is described in Section 6. Our detailed experimental study is presented in Section 7. We compare our work with related efforts in Section 8 and conclude in Section 9.

## 2 Data Cube Construction

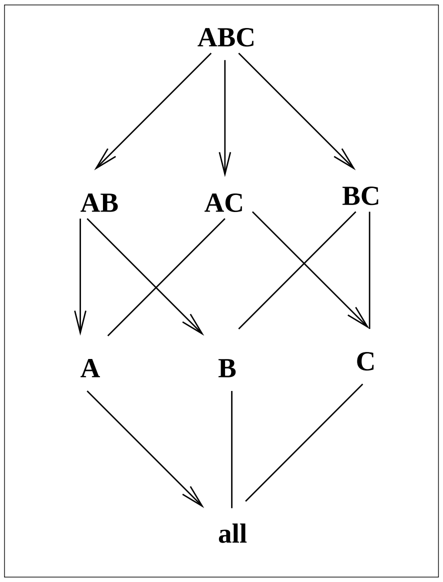

Figure 1: Lattice for data cube construction. Edges with arrows show the minimal spanning tree when $|A| \leq |B| \leq |C|$

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

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

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

To provide fast response to the users, a data warehouse computes aggregated values for all combinations of values. If the original dataset is $n$ dimensional, this implies computing and storing ${}^nC_m$ m-dimensional arrays, for $0 \leq m \leq n$. ${}^nC_m$ 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 $|A| \times |B|$.

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

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

**Using minimal parents:** In our example, the arrays AB, BC, and AC need to be computed from ABC, by aggregating values along the dimensions C, A, and B, respectively. However, the array A can be computed from either AB or AC, by aggregating along dimensions B or C. Because $|B| \leq |C|$, 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.

Another important issue is as follows. To ensure maximal cache and memory resue, we need to update AB, AC, and BC, simultaneously. However, if sufficient memory is not available to hold these three arrays in memory at the same time, we need to allocate and update portions of these arrays. Managing this can be quite challenging.

Some of these issues are addressed by a new data structure, aggregation tree which we introduce in the next section.

## 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 described in the previous section. Aggregation tree has the property that it bounds the total memory requirements for the data cube construction process.

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

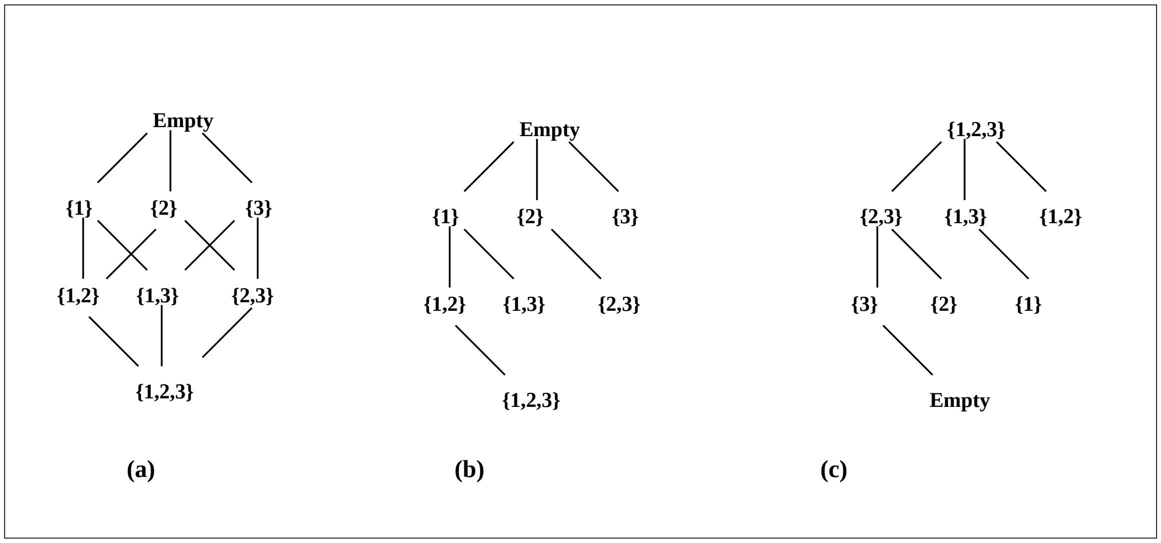

Figure 2: Prefix Lattice (a), Prefix Tree (b), and Aggregation Tree (c) for n = 3

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

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

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

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

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

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

*(a) $\phi$ is the root of the tree.*

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

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

Figure 3: Sequential Data Cube Construction Using the Aggregation Tree

(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 [7].

As we had stated in the previous section, one challenging situation arises when sufficient memory is not available to hold AB, AC, and BC in memory at any given time. To handle this, we need to allocate and update portions (or *tiles*) of these arrays at any given time. Since the aggregation tree based approach minimally bounds the memory requirements without tiling, we believe it gives us a good basis for developing tiling approach.

## 4 Parallel Algorithm Without Tiling

In the previous section, we introduced the aggregation tree based sequential data cube construction algorithm. In this section, we develop and analyze a parallel version of this algorithm. In the next section, we show how tiling can be applied to both sequential and parallel algorithms.

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

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

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

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

Figure 4: Parallel Data Cube Construction Without Tiling

verify that there are $2^p$ unique labels. A processor with the label $l_i$ is given the $l_i^{th}$ portion along the dimension $D_i$.

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

The parallel algorithm is presented in Figure 4.

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

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

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

An important question is, how does the initial distribution of the data impact the performance. Here, we analyze the communication volume as a function of data distribution.

**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 $1 \le y_1 < y_2 < \ldots < y_k < m \le n$. Then, the communication volume in computing the corresponding node $s'$ in the aggregation tree from the node $r'$ is given by*

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

**Theorem 1** *The total communication volume for data cube construction is given by*

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

The above expression for communication volume 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.

The expression for communication volume can be restated as

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

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

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

where,

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

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

The algorithm is presented in Figure 5. 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 $2 \times X_i$.

**Theorem 2** *Partitioning done using the algorithm in Figure 5 minimizes the interprocessor communication volume.*

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

Figure 5: Partitioning Different Dimensions to Minimize Communication Volume

## 5 Tiling-Based Approach for Scale 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.

### 5.1 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.

The tiling-based algorithm is presented in Figure 6. 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

```
Construct_Views($D_1 D_2 \ldots D_n$)
{
  *Foreach* tile $T$ of this node
    Expand_tile($D_1 \ldots D_n$, $T$)
}

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 6: A Tiling-Based Algorithm for Constructing Data Cube

be written back and read multiple times.

## 5.2 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.

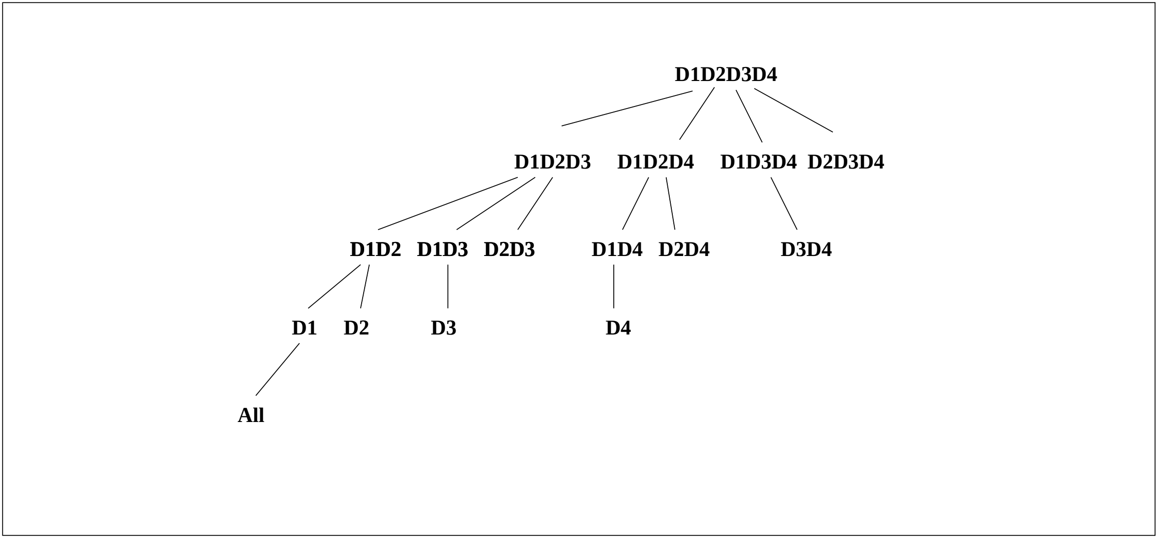

Figure 7: Aggregation Tree for Four-dimensional Data Cube Construction

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 7. 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 8. In this algorithm, $C.T'$ stands for a tile of values of child $C$. Other notations have the same meaning as in Figures 6 and 4.

Compared with the sequential tiling-based algorithm in Figure 6, 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 $T' = Maptile(D_1D_2 \dots D_n, T, C)$ 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 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 $\frac{1}{8th}$ 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

```
$Construct\_Cube(D_1D_2\ldots D_n)$
{
  Foreach tile $T$ of the root node $D_1D_2\ldots D_n$ on each processor{
    Foreach child $C$ of $D_1D_2\ldots D_n$ in the tree {
      $T' = Maptile(D_1D_2\ldots D_n, T, C)$
      $C.Table(T')$++
      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 $D_1D_2\ldots D_n$
        Perform aggregation operations on the tile $Maptile(D_1D_2\ldots D_n, T, C)$
    }
    Write-back the tile $Maptile(D_1D_2\ldots D_n, T, C)$ to disk if required
  }
  Foreach child $C$ of $D_1D_2\ldots D_n$ 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 $C' = X - C = \{D_{i1}, \ldots, D_{im}\}$
  If the processor is the lead processor along $D_{i1}, \ldots, D_{im}$
    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 8: A Tiling-Based Algorithm for Parallel Data Cube Construction

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.

## 6 Choosing Tiling Parameters

In using tiling, the key issue is how to tile the original array, so that the memory constraint is satisfied and overheads are minimized. This section presents theoretical analysis with the above gaols. For simplicity, this analysis is only presented for sequential tiling based algorithm. In the next section, we experimentally demonstrate that the results apply even for parallel tiling based algorithm.

Consider the original array $D_1 \dots D_n$ where the length of the dimension $D_i$ is $|D_i|$. This dimension is partitioned into $t_i$ tiles. The vector $(t_1, \dots, t_n)$ represents the tiling parameters for the original array.

**Theorem 3** *The total memory requirements for constructing the data cube from an array $D_1 \dots D_n$ using the aggregation tree, the algorithm in Figure 6 with right to left expansion of children, and a tiling vector $(t_1, \dots, t_n)$ is bounded by $\sum_{i=1}^{n} \left( \prod_{j=1, j \neq i}^{n} (|D_j|/t_j) \right)$*

Thus, we want to choose the tiling parameters so that

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

where $M$ is the memory available for storing intermediate results. Our goal is to minimize the *total tiling overhead*, while meeting the above constraint. Total Tiling Overhead is defined as the cost of writing-back and rereading a tile, as it is computed from multiple tiles of its parent. Consider a node $D_{x_1} \dots D_{x_m}$, and let us suppose that it is computed by aggregating its parent along the dimension $y$. This node will have to computed using $t_y$ tiles of its parent, and in the worst case, will have to be written-back and reread $t_y - 1$ times. Thus, the tiling overhead associated with this node will be $(t_y - 1) \times |D_{x_1}| \times |D_{x_m}|$. The total tiling overhead associated with the entire tree is simply the sum of this factor over all nodes in tree, except the root node.

**Theorem 4** *For an array $D_1 \ldots D_n$ that is tiled using a tiling vector $(t_1, \ldots, t_n)$, the total tiling overhead is*

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

For our discussion in the rest of this section, we only focus on tile sizes that are power of two, i.e., $t_i = 2^{q_i}$. In choosing the tile sizes, we have two considerations. First, we must tile sufficiently so that we meet the *memory constraint*. From Theorem 3, we have

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

Restating this, we get

$$(\prod_{i=1}^{n} |D_i|) \times (\sum_{i=1}^{n} \frac{2^{q_i}}{|D_i|}) \leq M \times 2^q$$

where, $q = \sum_{i=1}^{n} q_i$.

Because the dimension sizes are fixed, we rewrite this function as

$$\sum_{i=1}^{n} (f_i \times 2^{q_i}) \leq M \times 2^q$$

The second consideration in choosing tiling parameters is to minimize the total tiling overhead we had defined earlier. Recall that the tiling overhead for the entire tree is

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

We can state it as a function

$$\sum_{i=1}^{n} (g_i \times 2^{q_i})$$

Initially, we describe an algorithm for meeting the memory constraint, while minimizing the number of tiles required (i.e. the term $q$ defined above). The algorithm proceeds as follows. We initialize $q_1, q_2, \ldots, q_n$ to 0. In each step, we find the $i$ such that $f_i \times 2^{q_i}$ is the minimum among all values of $i$. Then, the value of $q_i$ is incremented by one. This process is repeated till the memory constraint is met.

**Theorem 5** *The above algorithm find the minimum value of $q$ that can meet the memory constraint.*

However, our primary goal is to minimize the tiling cost. One possible approach will be to use linear optimization method to minimize the tiling cost function, subject to the memory constraint. We have developed an algorithm whose complexity is linear in the number of dimensions, but does not always find a solution. If this algorithm does not find a solution, linear optimization can always be used as a backup.

The algorithm proceeds as follows. Initially, we use the algorithm described earlier and determine $q_i$, $i = 1, \ldots, n$ and $q = \sum_{i=1}^{n} q_i$, such that $2^q$ is the minimum number of tiles to satisfy the memory constraint.

The next step is to determine $q_i'$, $i = 1, \ldots, n$, to minimize the expression $\sum_{i=1}^{n} (g_i \times 2^{q_i})$, under the constraint $\sum_{i=1}^{n} q_i' = q$. This is done as follows. We initialize $q_1', \ldots, q_n'$ to 0. In each step, we find the $i$ such that $g_i \times 2^{q_i'}$ is the minimum among all values of $i$. Then, the value of $q_i'$ is incremented by one. This process is repeated until $\sum_{i=1}^{n} q_i' = q$.

The above step minimizes the expression $\sum_{i=1}^{n} (g_i \times 2^{q_i})$, under the constraint that $\sum_{i=1}^{n} q_i' = q$. This will give us the optimal solution to minimizing the tiling cost, provided that the values $q_1', \ldots, q_n'$ satisfy the memory constraint. If they do, the algorithm is successful. Otherwise, we say that this algorithm is unsuccessful and we need to use general linear optimization techniques.

**Theorem 6** *The above algorithm, if successful, returns tiling parameters $q_i'$ that minimize the tiling overhead while meeting the memory constraint.*

Note that in the expressions for memory constraint and tiling overhead

$$f_i = \frac{\prod_{j=1}^{n} |D_j|}{|D_i|}$$

and

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

As we will expect dimension sizes to be large integers, the terms $1 + \frac{1}{|D_j|}$ will be small. Thus, it is quite likely that we will get $q_i = q_i'$, and algorithm will terminate successfully.

## 7 Experimental Results

We have implemented the algorithms we have presented in this paper. In this section, we present a series of experimental results evaluating our algorithms and validating our theoretical results. Specifically, we had the following two main goals:

- Evaluating how tiling helps scale sequential and parallel data cube construction and the impact the number of tiles has on execution time.
- Evaluating how the choice of tiling parameters impacts execution time.

### 7.1 Scaling Data Cube Construction

We now present results showing how tiling helps scale data cube construction in both sequential and parallel environments.

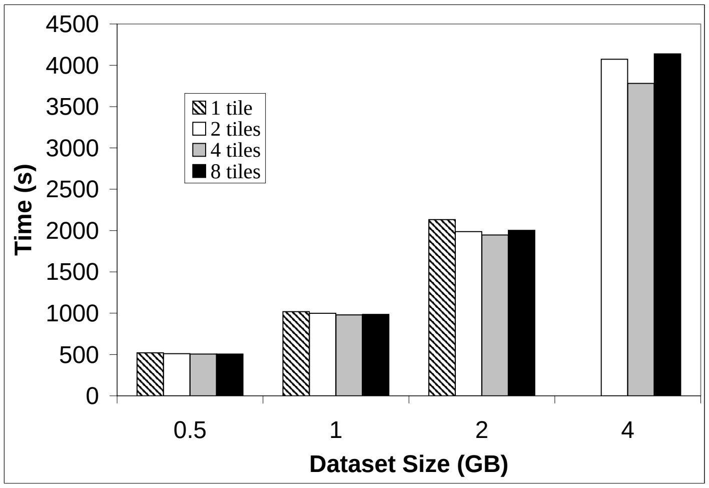

Figure 9: Scaling Sequential Data Cube Construction with Tiling (8 dimensional datasets)

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 9. 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

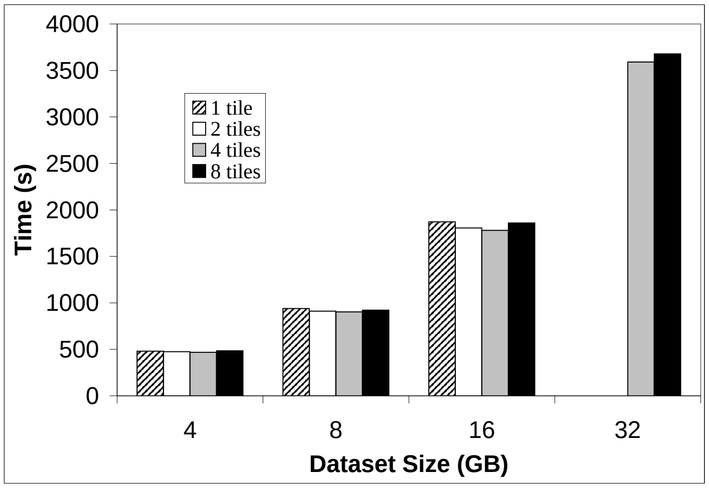

Figure 10: Scaling Parallel Data Cube Construction with Tiling (9 dimensional datasets on 8 nodes)

GB, respectively, similar to the previous experiment. The results are presented in Figure 10 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 9 and 10 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.

## 7.2 Impact of Tiling Parameters

In this sub-section, we focus on the evaluation of the theoretical results on tiling overhead.

Our first experiment was done on a single processor machine. We experimented on a $128 \times 128 \times 128 \times 128$ dataset with two different levels of sparsity 25% and 5%. We assume that the main memory is not sufficient and divide the dataset into 8 tiles. There are three possible tiling parameters, (0 1 1 1), (0 0 1 2) and (0 0 0 3). We refer these three options as three-dimensional, two-dimensional and one- dimensional tiling, respectively. The results are shown in Figure 11. The versions involving tiling are compared with a version that does not involve tiling.

We have two observations from these experimental results. First, even as the sparsity is varied, tiling on multiple dimensions has less tiling overhead, which is consistent with our theoretical results. Second, for the same dataset, less sparsity means more tiling overhead due to the smaller computation volume involved in each tile. In our experiments, compared with the tiling-based algorithm with the best tiling parameter (0 1 1 1), the sequential algorithm without tiling is slowed down by 8.41% and 15.92% on 25% and 5% datasets, respectively.

We also experimented on a non-cubic dataset with size $64 \times 64 \times 128 \times 512$. According to the

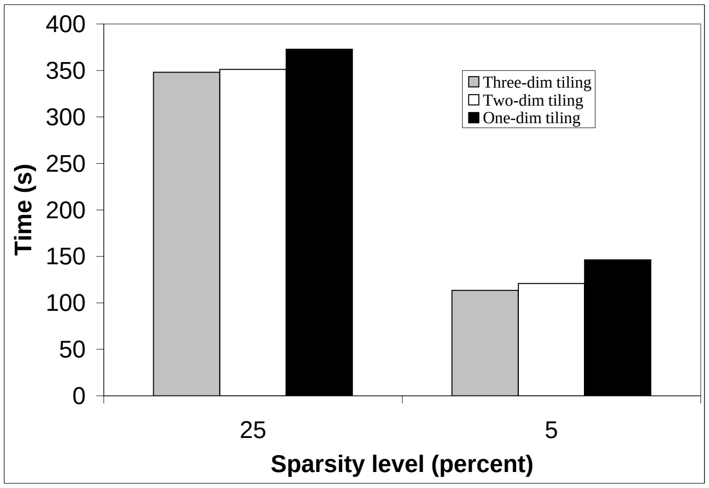

Figure 11: Impact of Tiling Parameters on Tiling Overhead, $128^4$ dataset, 1 Processor

algorithm of choosing tiling parameters, (0 0 1 2) should have the least tiling overhead. Figure 12 shows that experimental results are consistent with the theoretical results.

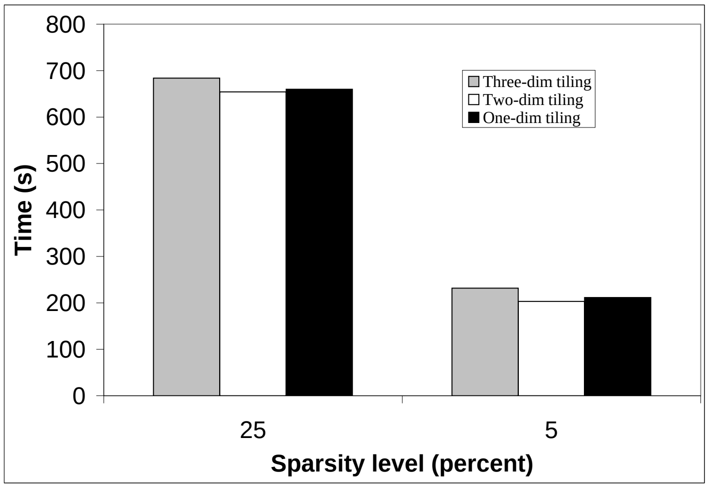

Figure 12: Impact of Tiling Parameters on Tiling Overhead, $64 \times 64 \times 128 \times 512$ dataset, 1 Processor

We also evaluated the performance of applying tiling-based algorithm to parallel data cube construction. The experiment was conducted on a $128 \times 128 \times 128 \times 128$ dataset with two different levels of sparsity 25% and 5%. We assume that after data distribution among processors, we still need to partition the portion of the original array on each processor into 4 tiles. Figure 13 shows that given a data distribution, using tiling parameters obtained through the algorithm of choosing tiling parameters can reduce the tiling overhead. The best case with tiling parameter (1 0 0 1) outperforms the case (0 0 0 2) by 4.62% and 6.14% on 25% and 5% datasets, respectively. Note that the execution times of the two cases with tiling parameters (1 0 0 1) and (0 0 1 1) are quite close to each other. This is because these two cases both involve two-dimensional tiling.

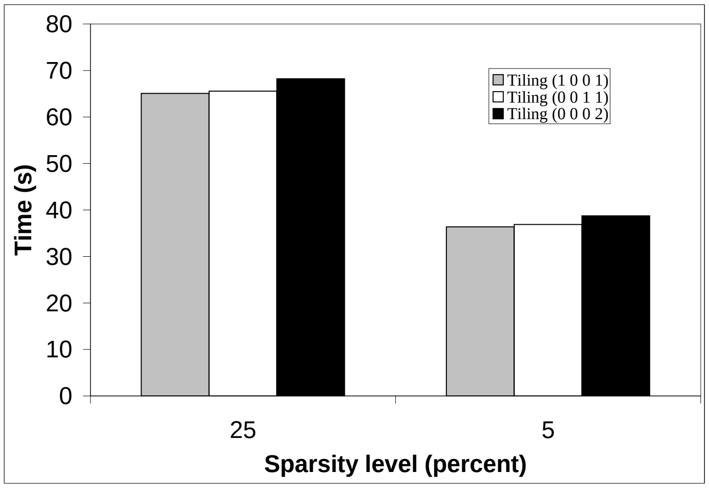

Figure 13: Evaluation of Tiling in a Parallel Environment

## 8 Related Work

Since Jim Gray [6] proposed the data cube operator, techniques for data cube construction have been extensively studied for both relational databases [9, 8] and multi-dimensional datasets [11, 10]. Our work belongs to the latter group. Zhao *et. al* [11] 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 of tiling does not require frequent write operations, and can flexibly work with different amounts of available memory. Tam [10] uses MNST (Minimum Number Spanning Tree) to reduce computing cost, with ideas some-what similar to our prefix tree. Again, this method also requires frequent writing back to disks. Also, the MMST and MNST approaches have not been parallelized so far.

Many researchers have developed parallel algorithms for data cube construction. Goil *et. al* [4, 5] did the initial work on parallelizing data cube construction starting from multidimensional arrays. Recently, Dehne *et. al* [3] have studied the problem of parallelizing data cube. They focus on a *shared-disk* model where all processors access data from a common set of disks. None of these efforts have considered combining tiling and parallelization.

## 9 Conclusions

In this paper, we have shown how sequential and parallel data cube construction algorithms can be further scaled to handle larger problems. This is done by *tiling* the input and output arrays on each node. We have addressed the challenges in using tiling while still maintaining the other desired properties of a data cube construction algorithm, which are, using minimal parents, and achieving maximal cache and memory reuse. We have presented a parallel algorithm that combines tiling with interprocessor communication. We have developed a closed form expression for tiling overhead, and have shown how tiling parameters can be chosen to minimize this overhead.

Our experimental results show the following. First, tiling helps in scaling data cube construction in both sequential and parallel environments. Second, choosing tiling parameters as per our theoretical results does result in better performance.

## References

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

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

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

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

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

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

[7] Ruoming Jin, Ge Yang, Karthik Vaidyanathan, and Gagan Agrawal. Communication and Memory Optimal Parallel Data Cube Construction. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, October 2003.

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

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

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

[11] 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.