Load Balanced and Communication Efficient Partitioning Strategies for Parallel Data Cube Generation[*]

Z. Chen [†] F. Dehne [‡] S. Hambrusch [§] A. Rau-Chaplin [¶]

October 18, 1999

Abstract

In this paper we present a general framework for the efficient parallelization of existing data cube construction algorithms. We present two different partitioning strategies, one for top-down and one for bottom-up cube algorithms. Both strategies balance the loads assigned to the processors and minimize communication overhead. Subcube computations are carried out using existing sequential, external memory data cube algorithms. The bottom-up partitioning strategy balances the number of single column external sorts made by each processor. The top-down strategy partitions a weighted tree in which weights reflect algorithm specific cost measures like estimated view sizes. Communication overhead is minimized by avoiding irregular and data-driven communication patterns, sending few large messages instead of many small ones, and overlapping communication and computation.

Both partitioning approaches are architecture independent. They can be implemented on any parallel machines composed of processors connected via an interconnection fabric. Processors are assumed to have standard-size local memory and access to parallel disks. The interconnection fabric can be an interconnection network or shared memory.

Experimental results presented support the claim that our partitioning strategies generate load balanced subcube problems requiring only small communication overhead. We measure the quality of a partitioning with respect to the sizes of the subproblems generated, the sequential time needed by each processor to perform the subcube computation, and the size of the output generated by each processor. We compare our results to optimal partitionings and to optimal parallel time.

[*]Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

[†]Faculty of Computer Science, Dalhousie University, Halifax, Canada B3J 2X4, *zchen@cs.dal.ca*

[‡]School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, *dehne@scs.carleton.ca*

[§]Dept. of Computer Science, Purdue University, West Lafayette, IN 47907, USA, *seh@cs.purdue.edu*

[¶]Faculty of Computer Science, Dalhousie University, Halifax, Canada B3J 2X4, *arc@cs.dal.ca*

1 Introduction

Data cube queries represent an important class of On-Line Analytical Processing (OLAP) queries in decision support systems. The precomputation of the different views of a data cube (i.e., the forming of aggregates for every combination of GROUP BY attributes) is critical to improving the response time of the queries [17]. Numerous solutions for generating the data cube have been proposed. These solutions differ in whether they are tailored towards sparse or dense relations [3, 19, 24, 25, 32], whether the relation is stored in main memory or external memory [24], and whether the approach allows efficient parallelizations [14, 15]. Solutions within each such category can also differ considerably. For example, top-down data cube computations for dense relations based on sorting have different characteristics from those based on hashing.

To meet the need for improved performance, to effectively handle the increase in data sizes and the potential for geographically distributed data, parallel and distributed solutions for generating the data cube are needed. *In this paper we present a general framework for the efficient parallelization of existing data cube construction algorithms.* We present load balanced and communication efficient partitioning strategies which generate a subcube computation for every processor. Subcube computations are then carried out using existing sequential, external memory data cube algorithms.

The heart of this paper are two partitioning strategies, one for top-down and one for bottom-up data cube construction algorithms. Balancing the load assigned to different processors and minimizing the communication overhead are two core problems in achieving high performance on parallel systems. Good load balancing approaches generally make use of application specific characteristics. Our partitioning strategies assign loads to processors by using metrics known to be crucial to the performance of data cube algorithms. The bottom-up partitioning strategy balances the number of single column external sorts made by each processor. The top-down strategy partitions a weighted tree in which weights reflect algorithm specific cost measures like estimated view sizes. Our partitioning strategies minimize the communication overhead by (i) avoiding irregular and data-driven communication patterns, (ii) sending few large messages instead of a many small ones, and (iii) overlapping communication and computation.

Our partitioning approaches are architecture independent. We present them using the current industrial standard parallel machine model. They can be implemented on any parallel machines composed of processors connected via an interconnection fabric. Processors are assumed to have standard-size local memory and access to parallel disks. The interconnection fabric can be any interconnection network or shared memory. Let R be the input relation (i.e., a d-dimensional raw data set) having size N. We assume that R is too large to fit into the local main memory of a single processor. When processors have their own local disks, the distribution of input R among the processors has significant impact on the performance. For example, the partitioning of R used in [14, 15] drives the communication requirements for data movement and communication-intensive aggregate operations. Note that the data cube generated (i.e., the output) is generally significantly larger than R (e.g., [24] indicates that the size of the data cube can be 200 times larger). When processors have local disks, we assume that R is stored once on the locals disks of each processor. When disks are shared, R is stored once overall. This assumption is realistic considering the size of the output generated and the disk sizes of existing parallel machines. It allows us to optimize communication arising from the problem itself and not from the constraints of the input. Communication in our algorithms is regular, can overlap with computation, and processors do not issue sends until they are able to send large messages. In fact, our partitioning strategies yield parallel data cube construction algorithms whose communication consists of a single data movement phase in which the final results are distributed over all parallel disks. In order to ensure maximum IO bandwidth for subsequent parallel applications accessing individual group-bys, for example for visualization, each group-by is stored in striped format over all disks of the parallel system.

Our experimental results show that our partitionings generate load balanced subproblems with small communication overhead. We measure the quality of a partitioning with respect to the sizes of the subproblems generated, the sequential time needed by each processor to perform the subcube computation, and the size of the output generated by each processor. We compare our results to an optimal partitioning (i.e., a partioning in which every processor generates the same amount of output data) and to optimal parallel time (i.e., the time corresponding to the optimal speedup). From these results we conclude that our partitioning approaches produce very efficient parallel data cube generation methods.

The paper is organized as follows. Section 2 describes the parallel models underlying our partitioning approaches. Section 3 discusses the input and the output of our algorithms. Section 4 presents our partitioning approach for parallel bottom-up data cube generation and Section 5 presents our method for parallel top-down data cube generation. Section 7 discusses the performance analysis of our partitioning approaches.

2 Parallel Computing Models

In parallel processing, considerable importance is given to the study of the various parallel models since available parallel processing architectures show much more variety than standard sequential machines. Current commercially available architectures are available in two basic types: message passing architectures and shared memory architectures (including CC-NUMA) [21]. In addition, parallel systems consisting of geographically distributed computing clusters connected over non-uniform bandwidth links have emerged as cost effective parallel systems. Such systems can consist of message passing as well as shared memory clusters. Hence, it is important to develop approaches and strategies which can work in either environments as well as for combinations of different architectures. Our partitioning solutions for parallel data cube construction are load balanced and communication efficient in message passing and shared memory environment. The following two subsections briefly outline the parallel processing models considered in this paper.

2.1 Message Passing Architectures

Various parallel models of computation have been designed to support parallel algorithm design for message passing architectures. We are using the *Coarse Grained Multicomputer* (CGM) model of parallel computation, a common model for algorithm and software development [4, 7, 16, 20, 27]. A CGM is comprised of a set of p processors $P_1, \ldots, P_p$ with local memory of size m per processor and an arbitrary communication network (or shared memory). The term "coarse grained" refers to the fact that we assume that local memory is of "significant" size. A typical definition of "significant", also used in this paper, is that $m \geq p$. To simplify exposition, we shall assume that p is a power of 2. Our algorithms presented in this paper can be easily generalized to handle arbitrary values of p.

A common methodology for the design of parallel algorithms is to design algorithms which alternate between local computation and global communication rounds. Each com-

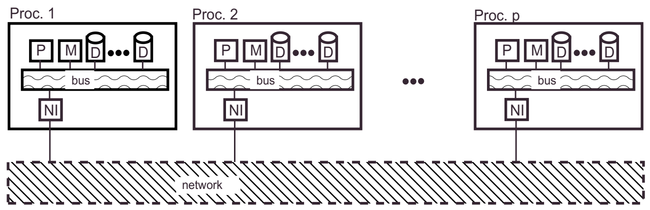


Figure 1: An EM-CGM With Disks Attached To The Individual Processors. (P = processor, M = memory, D = disk, NI = network interface)

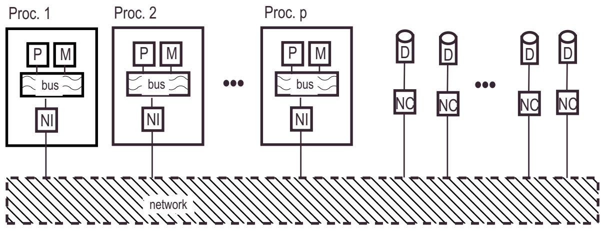


Figure 2: An EM-CGM With Independent Disks. (P = processor, M = memory, D = disk, NC = network controller)

munication round consists of executing an h-relation; in an h-relation each processor sends $O(m)$ data and receives $O(m)$ data. A general requirement is that data sent from a given processor to another processor in one communication round is packed into a single message. This is motivated by the fact that sending one large messages is more efficient than sending a number of small messages. Note that, in the BSP model [28], a computation/communication round is equivalent to a superstep with $L = mg$ (plus the above "packing" and "coarse grained" requirement), where g corresponds to the time the router needs to deliver packets.

Finding an efficient algorithm in the coarse grained multicomputer model is equivalent to minimizing the number of communication rounds as well as the total local computation time. This considers all parameters discussed above that are affecting the final observed speedup, and it requires no assumption on g. It has been shown that minimizing the number of supersteps also leads to improved portability across different parallel architectures [9, 11, 28]. We point out that actual implementations will often relax the view of an algorithm as consisting of a communication followed by a computation rounds and overlap communication and computation to achieve better performance. Our algorithm for generating the data cube have the same characteristic and the description given will be in the form of communication and computation rounds. However, our implementation overlaps communication and computation as much as possible.

Next, we discuss how to handle external memory in this model. The CGM model with multiple disks is referred to as *EM-CGM* [5, 6, 8]. It is a multi-processor version of Vitter's *Parallel Disk Model* [29, 30, 31]. For the purpose of data cube construction, we will distinguish between two versions. The first version, referred to as **EM-CGM with local disks**, is an EM-CGM with multiple disks attached to each processor as shown in Figure 1. Each of the p processors has one or more disks attached to its local bus. A disk is accessible only by its local processor. The second version, referred to as **EM-CGM**

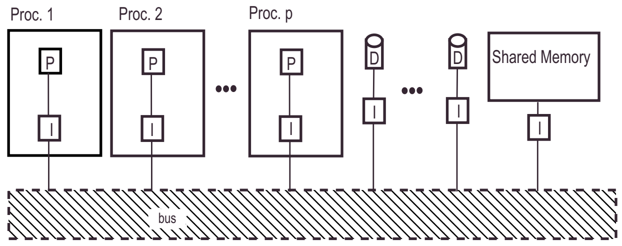


Figure 3: A Shared Memory Architecture. (P = processor, D = disk, I = interface)

with independent disks, is shown in Figure 2. Here, each disk is directly attached to the network via its own network controller. In this model, all processors can access the entire disk array.

The EM-CGM with local disks is usually less expensive to build because its processors are essentially standard, workstation type, machines. Examples include the *IBM SP2* and *SGI Power Challenge*. A EM-CGM with independent disks requires a network controller for each disk which has, in most cases, considerable processing power. The disk array usually implements a parallel file system, for example one with disk striping, where data transfer between the processors and the disk array is automatically parallelized. An example of a EM-CGM with independent disks is the *Cray T3E*. Due to its lower cost, the EM-CGM with local disks occurs much more frequent in practice. This model also covers the case of parallel computing on a network of workstations. However, such machines require that the application distributes the data over the disks in such a way that IO can be performed with maximal parallel IO bandwidth. We have studied such schemes for "general purpose" CGM virtual memory simulations [5, 6, 8]. In this paper, we will study this problem for the particular case of computing the data cube.

2.2 Shared Memory And CC-NUMA Architectures

Another group of today's parallel machines is based on shared memory architectures, either implemented in hardware or simulated at system level (CC-NUMA). Examples include SGI parallel workstations, SUN multiprocessor servers, and multiprocessor Pentium boards. An outline of a shared memory architecture is shown in Figure 3. All processors have shared access to one, single, global memory. The available hard disks are also shared among the processors with, in most cases, automatic disk striping. For smaller numbers of processors, shared memory access is usually provided through a fast bus. For larger numbers of processors, it is usually necessary to provide some system level mechanism which simulates shared memory access via a communication network using, e.g., sophisticated cashing mechanisms (CC-NUMA).

3 Parallel Data Cube Construction: Data Input and Output

We now briefly discuss how the input (i.e., the d-dimensional data set R of size N) and the output (i.e., the group-bys comprising the data cube) will be stored by our algorithms. When building the data cube in parallel, we are not only scaling up the available processing power but also the memory bandwidth and, in most cases, the size of the available memory.

In fact, parallelism is often a very economical way to increase the available memory address space and memory bandwidth as well as the available IO bandwidth.

For architectures with shared disks we assume that the d-dimensional data set R of size N is stored on the shared disks and that the output (i.e., the 2^d group-bys) are written to the shared disks as well.

For architectures with local disks, data set R can be stored in a number of different ways. Consider an EM-CGM architecture with local disks consisting of p processors, with each processor having main memory of size m, $mp \ll N$. We consider two possible input forms:

1. Partition R into p parts and store each part on a separate processor.
2. Have every processor P_i store a copy of the entire set R on its local disk(s), $1 \leq i \leq p$.

Recall that the data cube (i.e., the output) is much larger than the input data set R. As indicated in [24], the size of the data cube can be as much as 200 times the size of the raw data set. In practice, the size of the output can be assumed to be more than p times the size of the input and, hence, it can be assumed that there is enough disk space available at each processor to store the entire raw data set. Our algorithms will thus assume the second form of input; i.e., every processor stores its own input data on its disk(s). If the data were distributed among the p processors and we would decide against making local copies, an algorithm would encounter considerable additional communication costs. This type of partitioning of the data used in the parallel data cube algorithms described in [14, 15] results in considerable communication to move the data and to perform communication-intensive aggregate operations. However, as discussed in Section 1, this communication cost comes from the choice of data representation and is not inherent to the generation of the data cube.

The output generated by our algorithm (i.e., the 2^d group-bys) is stored on the EM-CGM's disks. A group-by is generated by exactly one processor and it is stored exactly once in the system. Each group-by is stored in striped format over all disks of all p processors. This will allow for maximum IO bandwidth when the data cubes are accessed later for applications like visualization. The display of an individual group-by can either be performed through a standard graphics workstation attached to the network of the EM-CGM or through a graphics workstation with a parallel frame buffer which allows for the different processors to write into this frame buffer independently and in parallel. In both cases, it is crucial that the group-by to be displayed is not on one single disk but striped over all disks to allow maximal IO bandwidth. All of our algorithms generate output in exactly this format.

4 Parallel Bottom-Up Data Cube Construction

In many data cube applications, the underlying data set R is sparse; i.e., N is much smaller than the number of possible values in the given d-dimensional space. Bottom-up data cube construction methods aim at computing the data cube for such cases. Bottom-up methods like *PartitionCube/MemoryCube* [24] and *BUC* [3] calculate the group-bys in an order which emphasizes the reuse of previously computed sort orders and in-memory sorts through data locality. If the data has previously been sorted by attribute A then, creating an AB sort order does not require a complete resorting. A local resorting of *A-blocks* (blocks of consecutive elements that have the same attribute A) can be used instead. The sorting of such A-blocks can often be performed in local memory and, hence, instead of another external memory sort, the AB order can be created in one single scan through the disk. Bottom-up methods [3, 24] attempt to break the problem into a sequence of single attribute sorts which share prefixes of attributes and can be performed in local memory with a single disk scan. The total computation time of these methods is dominated by the number of such disk scans.

In this section we describe a load balanced and communication efficient partitioning of the group-by computations into p independent subproblems. Each subproblem corresponds to an external-memory bottom-up group-by computation. It is solved by one processor using an existing external-memory, bottom-up cube algorithm like the ones described in [3, 24]. A load balanced partitioning is achieved by distributing the number of sorts made over the entire data equally among the p processors. Since the final sizes of the groups-bys are not known, the number of sorts are used as a heuristic measure in the partitioning. As will be discussed in Section 7, this measure (number of sorts) achieves a good distribution of the load. We note that for the algorithms in the later sections we use a metric estimating the size of the group-bys to guide the partitioning. Our partitioning approach for bottom-up cube generation is communication efficient. Communication is overlapped with computation, the communication patterns executed are regular, and communication only takes place when all p processors have their buffers filled. In the following we describe our solution with a message-passing architecture in mind. However, the approach is also load balanced and communication efficient in a shared-memory environment.

Let A_1, ..., A_d be the attributes of the data cube such that $|A_1| \geq |A_2| \geq ... \geq |A_d|$ where $|A_i|$ is the number of different possible values for attribute A_i. As observed in [24], the set of all groups-bys of the data cube can be partitioned into those that contain A_1 and those that do not contain A_1. In our partitioning approach, the groups-bys containing A_1 will be sorted by A_1. We indicate this by saying that they contain A_1 as a *prefix*. The group-bys not containing A_1 (i.e., A_1 is projected out) contain A_1 as a *postfix*. We then recurse with the same scheme on the remaining attributes. We shall utilize this property to partition the computation of all group-bys into independent subproblems computing group-bys. The load between subproblems will be balanced and they will have overlapping sort sequences in the same way as for the sequential methods. In the following we give the details of the partitioning.

Let x, y, z be sequences of attributes representing sort orders and let A be an arbitrary single attribute. We introduce the following definition of sets of attribute sequences representing sort orders (and their respective group-bys):

$$B_1(x, A, z) = \{x, xA\} \quad (1)$$

$$B_i(x, Ay, z) = B_{i-1}(xA, y, z) \cup B_{i-1}(x, y, Az), 2 \leq i \leq \log p + 1 \quad (2)$$

The entire data cube construction corresponds to the set $B_d(\emptyset, A_1 \ldots A_d, \emptyset)$ of sort orders and respective group-bys. We refer to d as the *rank* of $B_d(\emptyset, A_1 \ldots A_d, \emptyset)$, $d = \log p + 1$. The set $B_d(\emptyset, A_1 \ldots A_d, \emptyset)$ is the union of two subsets of rank $d-1$: $B_{d-1}(A_1, A_2 \ldots A_d, \emptyset)$ and $B_{d-1}(\emptyset, A_2 \ldots A_d, A_1)$. These, in turn, are the union of four subsets of rank $d-2$. Figure 5 illustrates this process for $d = 8$ and $p = 4$. A complete example for a 4-dimensional data cube with attributes A, B, C, D is shown in Figure 4.

As already stated, for the sake of simplifying the discussion, we assume that p is a power of 2. Consider the $2p$ B-sets of rank $d - \log_2(p) - 1$. Let $\beta = (B^1, B^2, \ldots B^{2p})$ be these $2p$

$B_4(\emptyset, ABCD, \emptyset)$	$B_3(\emptyset, BCD, A)$	$B_2(\emptyset, CD, BA)$	$B_1(\emptyset, D, CBA) = \{\emptyset, D\}$
			$B_1(C, D, BA) = \{C, CD\}$
		$B_2(B, CD, A)$	$B_1(B, D, CA) = \{B, BD\}$
			$B_1(BC, D, A) = \{BC, BCD\}$
	$B_3(A, BCD, \emptyset)$	$B_2(A, CD, B)$	$B_1(A, D, CB) = \{A, AD\}$
			$B_1(AC, D, B) = \{AC, ACD\}$
		$B_2(AB, CD, \emptyset)$	$B_1(AB, D, C) = \{AB, ABD\}$
			$B_1(ABC, D, \emptyset) = \{ABC, ABCD\}$

Figure 4: Partitioning for a 4-Dimensional Data Cube with Attributes A, B, C, D.

sets in the order defined by Equation (2). Define

$$\begin{aligned}\text{Shuffle}(\beta) &= < B^1 \cup B^{2p}, B^2 \cup B^{2p-1}, B^3 \cup B^{2p-2}, \ldots, B^p \cup B^{p+1} > \\ &= < \Gamma_1, \ldots, \Gamma_p >\end{aligned}$$

We assign set $\Gamma_i = B^i \cup B^{2p-i+1}$ to processor P_i, $1 \leq i \leq p$. Observe that from the construction of all group-bys in each Γ_i it follows that every processor performs the same number of single attribute sorts.

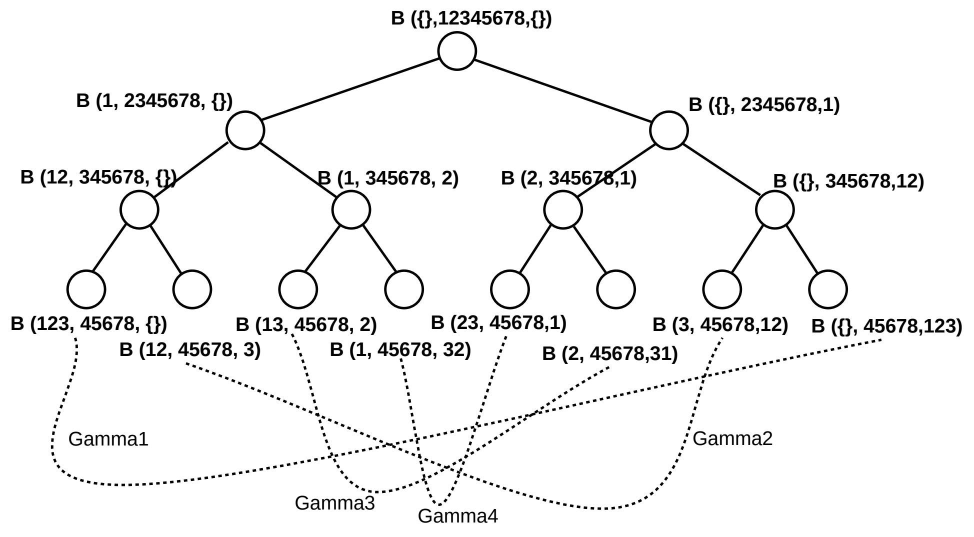


Figure 5: The B and Γ sets for $d = 8$ and $p = 4$.

Algorithm 1 Parallel Bottom-Up Cube Construction (outline).
Architecture: An EM-CGM with p processors, m local main memory per processor, and parallel disks associated with each processor.
Input: The d-dimensional raw data set R; a copy of R is stored on the external memory (i.e., the disks) of each processor.
Output: The data cube with each group-by distributed (in striped format) over the disks of all processors.

Each processor P_i, $1 \leq i \leq p$, performs the following steps, independently and in parallel:

(1) Calculate Γ_i using Equations (1) and (2).

(2) Compute all group-bys in subset Γ_i using a sequential, external-memory bottom-up cube construction method. Distribute partial results of each group-by to disks as results become available, overlapping communication and computation. Details of how buffers are filled and the type of communication performed are described below.

— End of Algorithm —

To complete the description of the algorithm we discuss the communication details of Step 2. Each processor P_i, $1 \leq i \leq p$, creates an output stream of disk blocks which need to be written to disks. For an EM-CGM model with independent disks the writing to disks is straightforward: processors send the filled blocks to the disk array which stores them in striped format. For an EM-CGM with local disks, the disk operations are more involved. If processors write blocks of data to the local disks of other processors whenever data becomes available, the communication pattern is likely to be irregular and data-dependent. Such patterns can lead to poor performance due to hot spots and increased congestion. We avoid this problem by delaying communication until every processor has generated "enough" data to be sent to disks. Having enough data will allow us to use a regular communication pattern. This will now be explained in more detail.

Assume that for each processor the output stream representing group-by data is stored in a *block buffer*. A block buffer can hold up to $\Theta(m)$ data. When the block buffer is full, it is made available for communication. For this purpose the block buffer is logically partitioned into p consecutive blocks. Whenever every processor has generated one full block buffer, a single global communication operation is invoked. In this operation we route the j-th block in the block buffer at processor P_i to processor P_j, for all $1 \leq i, j \leq p$. This operation corresponds to one call of the *MPI_Alltoallv*(...) operation in the industrial standard parallel programming interface *MPI* [18, 22]. In parallel programming terminology, the communication corresponds to performing an *h-relation* (i.e., a permutation of the block buffer data). Except for possibly the last communication round, a processor will always contribute a full block buffer (or none at all). Each processor, upon receiving blocks from a communication round, will write these onto its local disk, thereby creating a striped format for each group-by. The operation is illustrated in Figure 6. In a worst-case scenario, it is potentially possible that a number of processors have filled their block buffers while others have not. This can interfere with the computational process. To solve this problem we propose to use two block buffers so that a processor with one full block buffer can continue filling a second one while the first one waits for the communication to start. Given the properties stated below, one can expect that processors will complete block buffers at similar times, thus allowing a full overlap of communication and computation. This concludes the description of the algorithm.

Before discussing some of the properties of our partitioning approach crucial to its load balanced and communication efficient characteristics, we point out that Algorithm 1 assumes $p \leq 2^{d-1}$. If a parallel algorithm is needed for larger values of p, the partitioning strategy needs to be augmented. Such an augmentation could, for example, be a partitioning strategy based on the number of data items for a particular attribute. This would be applied after partitioning based on the number of attributes has been done. Since the range $p \in \{2^0 \ldots 2^{d-1}\}$ covers current needs with respect to machine and dimension sizes, we do not further discuss such augmentations in this paper.

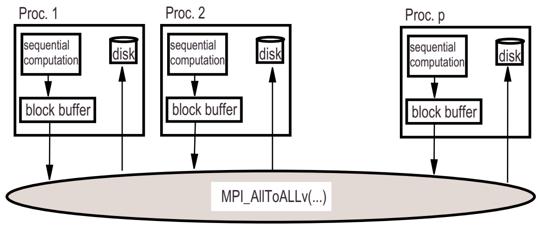


Figure 6: Illustration of Striped Disk Output for an EM-CGM With Local Disks.

The following four properties are the basis of our argument that the partitioning approach is load balanced and communication efficient. A more detailed discussion will be given in the full version of the paper.

Property 1 *Algorithm 1 exhibits the following properties:*

(a) The computation of each group-by is assigned to a unique processor.

(b) The calculation of set Γ_i of group-bys, assigned to processor P_i, requires the same number of single attribute sorts for all $1 \leq i \leq p$.

(c) The sorts performed at processor P_i share prefixes of attributes as in [3, 24] and can be performed in local memory with a single disk scan in the same manner as in [3, 24].

(d) The algorithm requires no communication except for the final distribution of results. The communication is regular and distributes the group-bys over the entire disk array in striped data format.

We observe that for shared memory architectures, Algorithm 1 can be applied in exactly the same way, using disk writes as for the EM-CGM with independent disks. Observation 1 applies analogously, except for Property (d) since in shared memory architectures all memory accesses are uniform.

5 Parallel Top-Down Data Cube Construction

Top-down approaches for computing the data cube, like the sequential *PipeSort*, *Pipe Hash*, and *Overlap* methods [1, 10, 25], use more detailed views to compute less detailed ones that contain a subset of the attributes of the former. They apply to data sets which are not sparse and where the number of data items in a group-by can shrink considerably as the number of attributes decreases (data reduction). A group-by is called a child of some parent group-by if the child can be computed from the parent by aggregating some of its attributes. This induces a partial ordering of the views, called the *lattice*. An example of a 4-dimensional lattice is shown in Figure 7, where A, B, C, and D are the four different attributes. The *PipeSort*, *PipeHash*, and *Overlap* methods select a spanning tree T of the lattice, rooted at the view containing all attributes. *PipeSort* considers two cases of parent-child relationships. If the ordered attributes of the child are a prefix of the ordered attributes of the parent; e.g., ABCD $\rightarrow$ ABC, then a simple scan is sufficient to create the child from the parent. Otherwise, a sort is required to create the child. *PipeSort* seeks

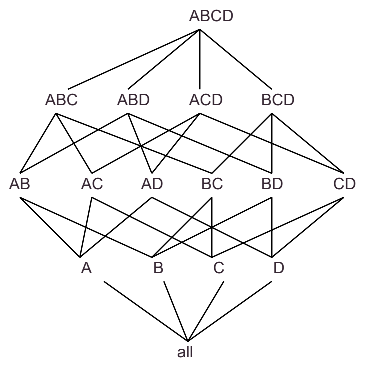


Figure 7: A 4-Dimensional Lattice.

to minimize the total computation cost by computing minimum cost matchings between successive layers of the lattice. *PipeHash* uses hash tables instead of the sorting. *Overlap* attempts to reduce sort time by utilizing the fact that overlapping sort orders do not always require a complete new sort. For example, the ABC group-by has A partitions that can be sorted independently on C to produce the AC sort order. This may allow to perform these independent sorts in memory rather than using external memory sort.

We next outline a partitioning approach which generates p independent subproblems, each of which can be solved by one processor using an existing external-memory top-down cube algorithm. The main goals of the partitioning are:

- balance the load assigned to processors (i.e., balance the sizes of the subproblems),
- minimize computation overhead (i.e., the increase in total computation compared to the sequential solution), and
- minimize communication overhead (i.e., the potential slowdown through additional communication between the processors).

The first step of our algorithm determines a spanning tree T of the lattice by using one of the existing approaches like *PipeSort*, *PipeHash*, and *Overlap*, respectively. To balance the load between the different processors we next perform a storage estimation to determine approximate sizes of the views in T. This can be done, for example, by using methods described in [12] and [26]. We now work with a weighted tree. The most crucial part of our solution is the partitioning of the tree. The partitioning of T into subtrees induces a partitioning of the data cube problem into p subproblems (subsets of group-bys). Determining an optimal partitioning of the weighted tree is easily shown to be an NP-complete problem (by making, for example, a reduction using a p processor scheduling problem). Since the weights of the tree represent estimates, a heuristic approach which generates p subproblems with "some control" over the sizes of the subproblems holds the most promise. While we want the sizes of the p subproblems balanced, we also want to minimize the number of subtrees assigned to a processor. Every subtree may require a scanning of the entire data set R and thus too many subtrees can result in poor IO performance. The solution we develop balances these two considerations.

Our heuristics makes use of a related partitioning problem on trees for which efficient algorithms exist, the *min-max tree k-partitioning problem* [2].

Definition 1 *Min-max tree k-partitioning. Given a tree T with n vertices and a positive weight assigned to each vertex, delete k edges in the tree such that the largest total weight of a resulting subtree is minimized.*

The min-max tree k-partitioning problem has been studied in [2, 13, 23], and an $O(n)$ time algorithm has been presented in [13]. It is not hard to see that a min-max k-partitioning of T does not necessarily compute a partitioning into subtrees of equal size. To achieve a better distribution of the load we apply an over partitioning strategy: instead of partitioning the tree T into p subtrees, we partition it into $s \times p$ subtrees, where s is an integer, $s \geq 1$. We generate $s \times p$ subtrees using a min-max tree k-partitioning algorithm. Then, we use a "packing strategy" to determine which subtrees belong to which processors, assigning s subtrees to every processor. The most successful packing strategies are likely to be based on considering the subtrees by weights and using a heuristic to minimize the differences in weights. The performance results presented in Section 7 assign s subtrees as described in Algorithm 2.

Algorithm 2 Tree-partition(T, s, p).
Architecture: A single, sequential processor with m main memory and external memory (i.e., disks).
Input: A spanning tree T of the lattice with positive weights assigned to the nodes (representing the cost to build each node from it's ancestor in T). An integer parameters s (oversampling ratio) and p (number of processors).
Output: A partitioning of T into p subsets $\Sigma_1, \ldots, \Sigma_p$ of s subtrees each.

```
(1) Compute a min-max tree $s \times p$ -partitioning of $T$ into $s \times p$ subtrees $T_1, \ldots, T_{s \times p}$.
(2) Distribute subtrees $T_1, \ldots, T_{s \times p}$ among the $p$ subsets $\Sigma_1, \ldots, \Sigma_p$, $s$ subtrees per subset,
    as follows:
    (2a) Create $s \times p$ sets of trees named $\Upsilon_i$, $1 \leq i \leq sp$, where initially $\Upsilon_i = \{T_i\}$. The
         weight of $\Upsilon_i$ is defined as the total weight of the trees in $\Upsilon_i$.
    (2b) For $j = 1$ to $s - 1$
         • Sort the $\Upsilon$-sets by weight, in increasing order. W.l.o.g., let $\Upsilon_1, \ldots, \Upsilon_{sp-(j-1)p}$
           be the resulting sequence.
         • Set $\Upsilon_i := \Upsilon_i \cup \Upsilon_{sp-(j-1)p-i+1}$, $1 \leq i \leq p$.
         • Remove $\Upsilon_{sp-(j-1)p-i+1}$, $1 \leq i \leq p$.
    (2c) Set $\Sigma_i = \Upsilon_i$, $1 \leq i \leq p$.
— End of Algorithm —
```

The above tree partition algorithm is embedded into our parallel top-down data cube construction algorithm as follows.

Algorithm 3 Parallel Top-Down Cube Construction (outline).
Architecture: An EM-CGM with p processors, m local main memory per processor, and parallel disks associated with each processor.
Input: The d-dimensional raw data set R of size N; R is stored on external memory (i.e., the disks).
Output: The data cube with each group-by distributed (in striped format) over the disks of all processors.

Each processor P_i, $1 \leq i \leq p$, performs the following steps independently and in parallel:

(1) Compute the spanning tree T of the lattice as required by a sequential top-down cube construction method (e.g., *PipeSort*, *PipeHash*, or *Overlap*).

(2) Apply the storage estimation method in [26] and [12] to determine the approximate sizes of all group-bys in T. Compute the weight of each node of T; i.e., the cost to build each node from it's ancestor in T.

(3) Execute Algorithm *Tree-partition(T, s, p)* as shown above, creating p sets $\Sigma_1, \ldots, \Sigma_p$. Each set Σ_i contains s subtrees of T.

(4) Compute all group-bys in subset Σ_i using the sequential top-down cube construction method chosen in Step 1. Distribute partial results of each group-by to disks (in striped format) as results become available, overlapping communication and computation.

— End of Algorithm —

Our performance results described in Section 7 show that an over partitioning with $s = 2$ or 3 achieves very good results with respect to balancing the loads assigned to the processors. This is an encouraging result since a small value of s is crucial for optimizing IO performance. Recall that each subtree may require the processor to scan the entire input R. Consult Section 7 for more details.

The details of Step 4 are analogous to Step 2 in Algorithm 1. For an EM-CGM with local disks, each processor P_i, $1 \leq i \leq p$, computes the group-bys in Σ_i using a sequential top-down cube construction method (e.g. *PipeSort*, *PipeHash*, or *Overlap*). This creates an output stream of disk blocks stored in a block buffer. When the block buffer is full, it is partitioned into p consecutive blocks. An h-relation operation is executed to send the j-th block in the block buffer of P_i to processor P_j, $1 \leq i, j \leq p$. This operation corresponds to one call of the $MPI\_Alltoallv(...)$ operation in the industrial standard parallel programming language MPI [22]. We point out that the algorithm can be applied in exactly the same way to the EM-CGM with independent disks and to shared memory architectures with uniform memory accesses.

6 Parallel Array-Based Data Cube Construction

Our methods in Section 5 can be modified to obtain an efficient parallelization of the *ArrayCube* method presented in [32]. The *ArrayCube* method is aimed at dense data cubes and structures the raw data set in a d-dimensional array stored on disk as a sequence of "*chunks*". Chunking is a way to divide the d-dimensional array into small size d-dimensional chunks where each chunk is a portion containing a data set that fits into a disk block. When a fixed sequence of such chunks is stored on disk, the calculation of each group-by requires a certain amount of buffer space [32]. The *ArrayCube* method calculates a minimum memory spanning tree of group-bys, *MMST*, which is a spanning tree of the lattice such that the total amount of buffer space required is minimized. The total number of disk scans required for the computation of all group-bys is the total amount of buffer space required divided by the memory space available.

The following is a modified version of Algorithm 3 which efficiently parallelizes the *ArrayCube* method by partitioning the MMST. A more complete discussion of where the two algorithms differ will appear in the full version of the paper.

```
Algorithm 4 Parallel ArrayCube Construction (outline).
Architecture: An EM-CGM with $p$ processors, $m$ local main memory per processor, and
parallel disks associated with each processor.
Input: The $d$-dimensional raw data set $R$ of size $N$ stored in the external memory of each
processor.
Output: The data cube with each group-by distributed over all disks (in striped format).
    Each processor $P_i$, $1 \leq i \leq p$, performs the following steps independently and in
    parallel:
      (1) Compute the minimum memory spanning tree $T$ of the lattice as required by the
          ArrayCube method.
      (2) Compute the weight of each node of $T$, i.e. the buffer size required to build each
          node.
      (3) Execute Algorithm Tree-partition($T$, $s$, $p$) using the weights computed in Step 2,
          creating $p$ sets $\Sigma_1, \ldots, \Sigma_p$ of $s$ subtrees of $T$, each.
      (4) Compute all group-bys in subset $\Sigma_i$ using the sequential top-down cube construction
          method chosen in Step 1. Distribute each group-by over all disks (in striped
          format) in the same way as discussed for Algorithm 3.
— End of Algorithm —
```

7 Performance Analysis

In this section we present our experimental results for the two partitioning approaches. The experimental evaluation provides answers to the following questions:

- How balanced are the sizes of the generated subproblems?
- How balanced are the sequential subcube computations carried out by each processor?
- How balanced are the output sizes generated by each processor?

The next two sections answer these questions for the top-down and bottom-up partitionings. The results demonstrate that our partitioning approaches produce very efficient parallel data cube generation methods.

7.1 Bottom-Up Cube Partitioning

Figures 8 and 9 illustrate the performance of our parallel bottom-up data cube construction method. The experiments were performed on a Sun Ultra Sparc 30 with 64 Mbytes of memory running under SunOS 5.6. The results are based on the generous provision of an executable for the *Cube Computation Software Version 2.0 beta 2* (an implementation of [24]) provided to us by Dr. K. Ross of Columbia University. All graphs are based on results for seven dimensional input data sets with $N = 200,000$. The input data was generated using the data generator provided with Ross's Cube Computation Software. Five runs were used to generate each data point.

Figure 8 compares optimal CPU time to the CPU time required to solve the largest subproblem produced by Algorithm 1. As optimal CPU time we use the time of the bottom-up algorithm in [24] on the entire data set divided by p, the number of processors. The

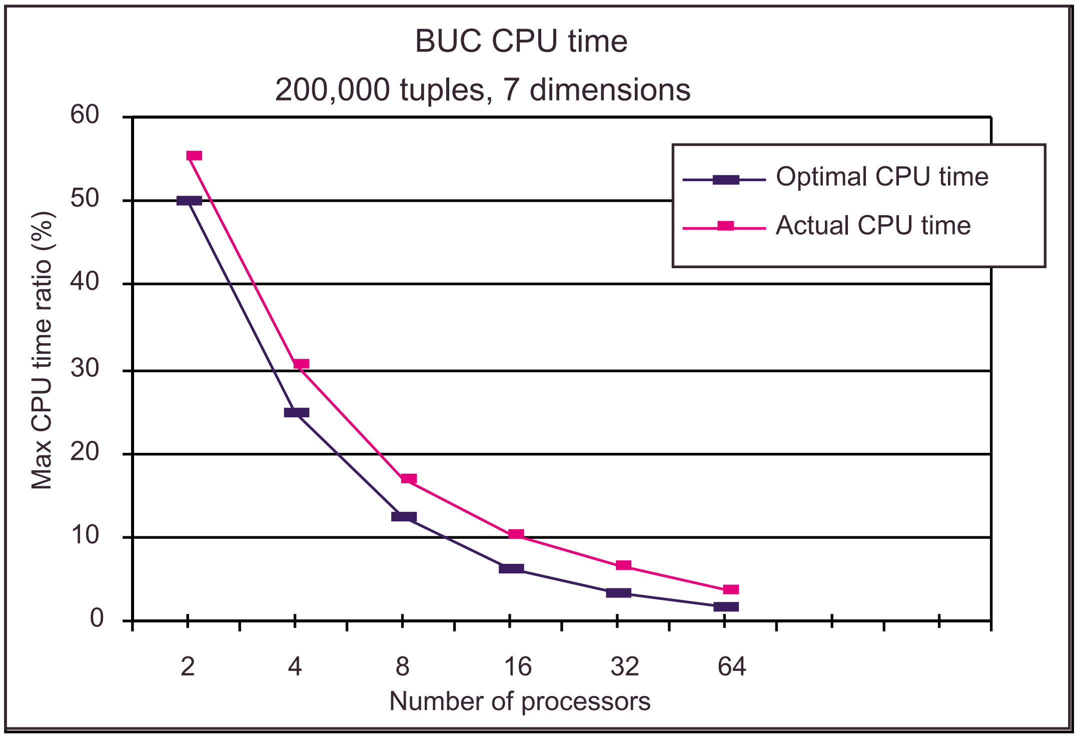


Figure 8: Experimental Results for Bottom-Up Method: Maximum CPU time.

plot shows these CPU times as the number of processors increases from 2 to 64. Figure 8 shows that the optimal and actual CPU time curves track each other closely. It shows that partitioning the number of sorts evenly among all processors produces a good distribution of the computational load.

Our experiments reported in Figure 8 show that for seven dimensional data and values of p up to 2^4, Algorithm 1 generates a solution that is within a factor of two from optimal with respect to CPU time. The relationship between 7 and 2^4 is not a coincidence. In general, we observed that for d-dimensional data and processors up to 2^{d-3}, the CPU time is within a factor of 2 from the optimal time.

The CPU times given in Figure 8 account for the computation of the data cubes, but not for the time required to output them to disks. In Figure 9, we measure the maximum size of the output generated by a processor. The optimal output size is the total output size divided by p; i.e., the situation when every processor generates the same amount of data. Figure 9 compares optimal output size to the output size of the largest subproblem produced by the load balancing scheme of Algorithm 1. The figure shows these two quantities as the number of processors increases from 2 to 64. The optimal and actual output size curves track each other closely. Figure 9 shows that for $d = 7$ and $p \leq 2^4$, the results are within a factor of 2 from optimal with respect to output size. In general, for a d-dimensional data set, we observed a load within a factor of 2 of the optimal load as long as $p \leq 2^{d-3}$. For example, for $d = 7$, as long as at least $2^3 = 8$ cubes are allocated to each processor, Algorithm 1 balances CPU time and output size effectively.

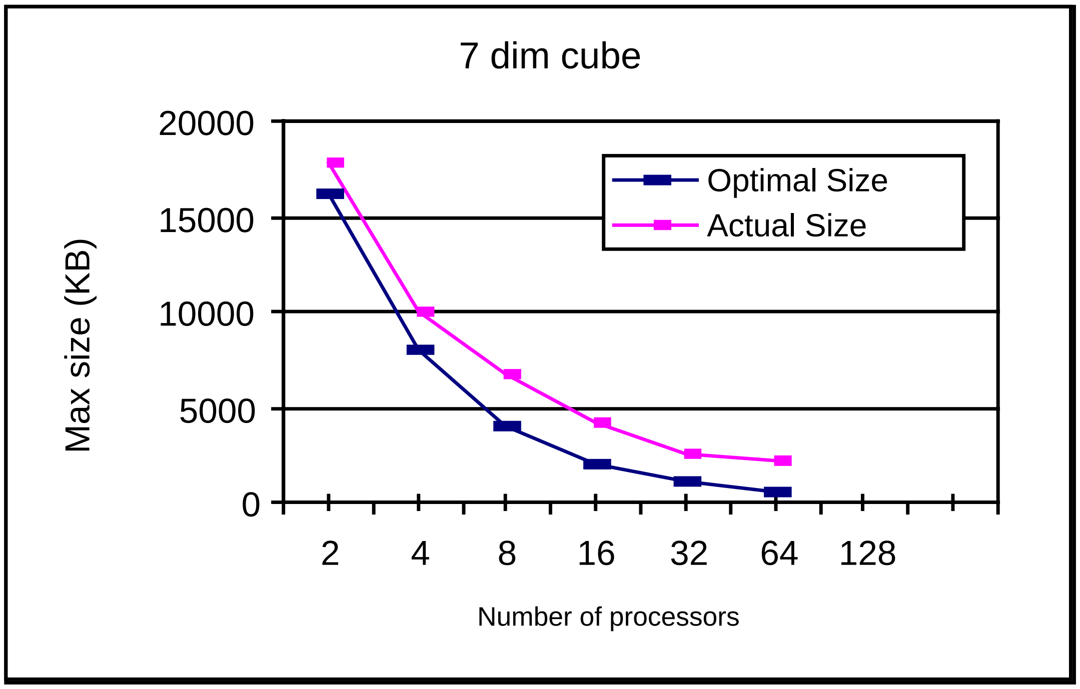


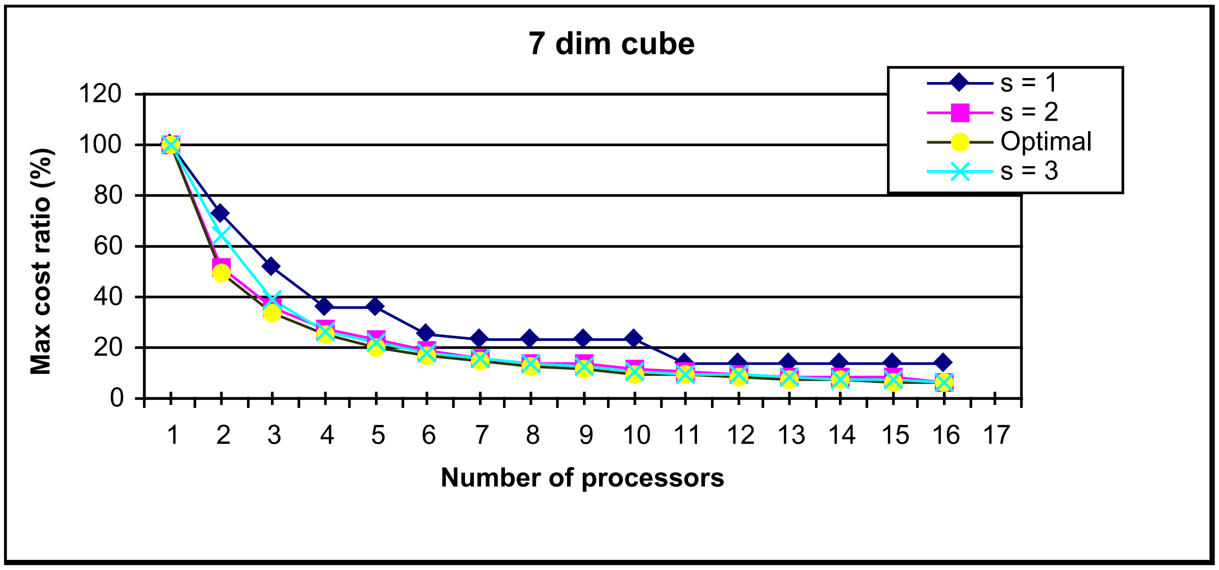
Figure 9: Experimental Results for Bottom-Up Method: Maximum output size

7.2 Top-Down Cube Partitioning

Figures 10, 11, and 12 illustrate the performance of the parallel top-down data cube partitioning algorithm described in Section 5. Recall that Algorithm 3 partitions a weighted subtree T of the lattice, generated by a sequential top-down algorithm, into $s \times p$ subtrees. These subtrees are then allocated to p processors. The weight of a node in the tree reflects the cost of the sequential cube algorithm as well as disk accesses. An optimal partitioning is one in which all subtrees have equal weight. However, such a partitioning may not even exist.

In Figures 10, 11, and 12, the y-axis represents the ratio of the weight of the largest subproblem to the total weight of the initial problem. Each figure also contains a curve showing the optimal result (i.e. total weight / p) for comparison. Each value shown is the result of five experiments. We consider data sets having between seven and ten dimensions and numbers of processors between 1 and 16. We observe that the curves corresponding to oversampling ratios $s \in \{1, 2, 3\}$ track the optimal curve well. This supports our claim that the use of Min-Max tree k-partitioning in Algorithm 3 is an effective heuristic for generating load balanced subproblems. Even with an oversampling ratio $s = 1$ (i.e. no oversampling) the algorithm produces a load balance that is within a factor two from optimal. With an oversampling ratio of $s = 2$ the algorithm achieves a load balance that is, on average, within 17% from optimal. With an oversampling ratio of $s = 3$ the algorithm achieves a load balance that is, on average, within 14% from optimal.

Table 1 summarizes the average load balancing results for $p \in \{1, \ldots, 16\}$ and $d \in \{7, \ldots, 10\}$. The oversampling ratio s can be viewed as a tuning parameter that can be adjusted to a particular computing environment (machine type, p, parallel implementation) and data cube generation problem (d, N). Figure 12 examines the performance of Algo-



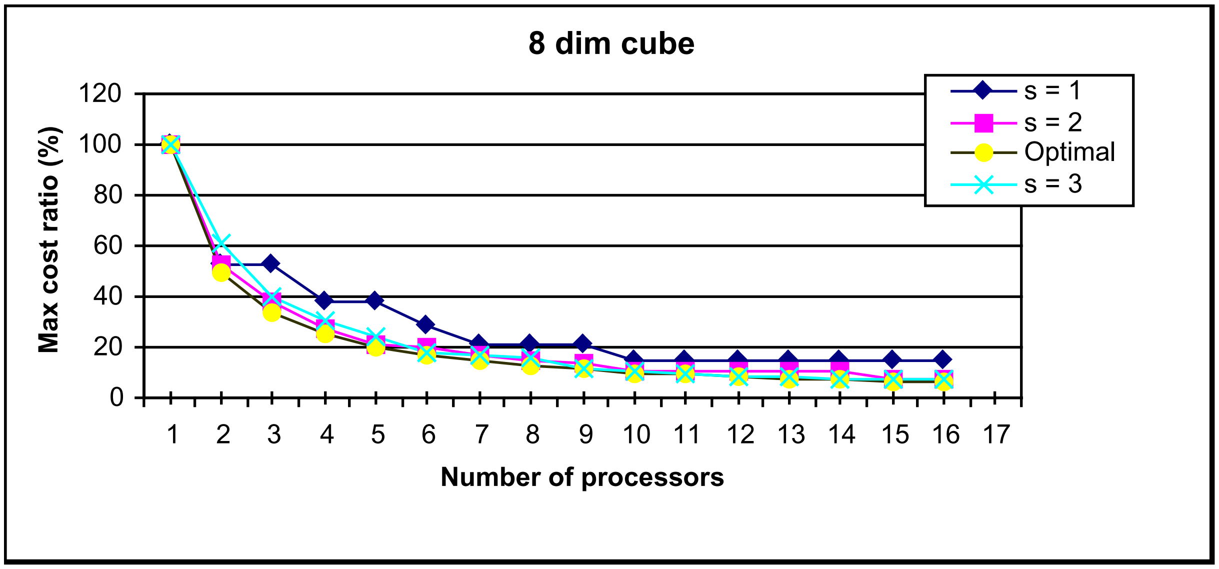
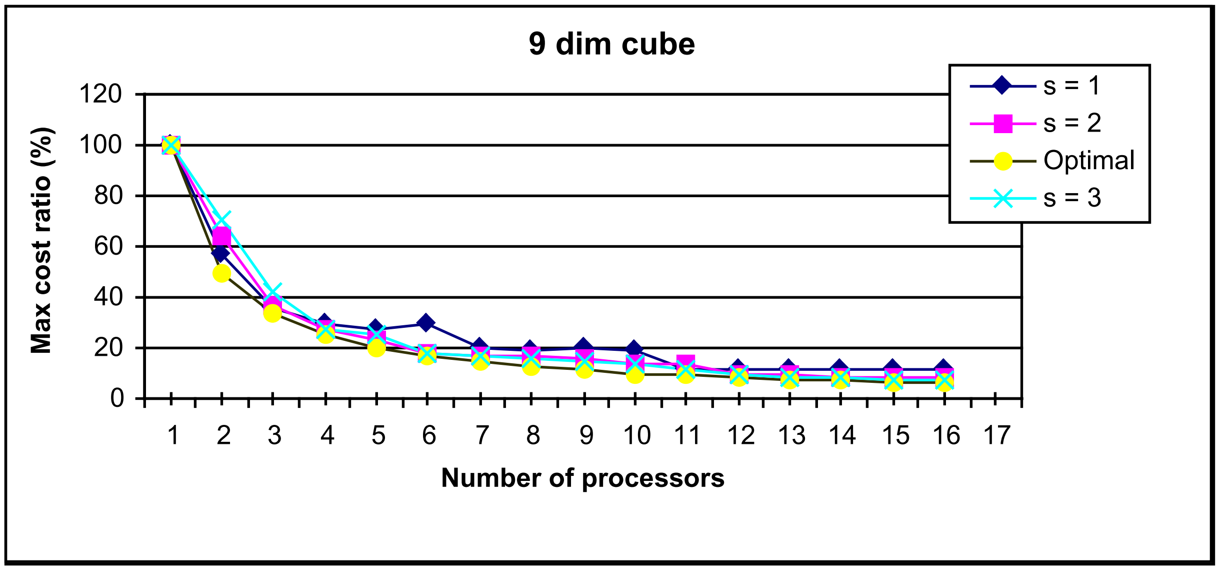


Figure 10: Experimental Results for Top-down Method: d=7 and d=8.



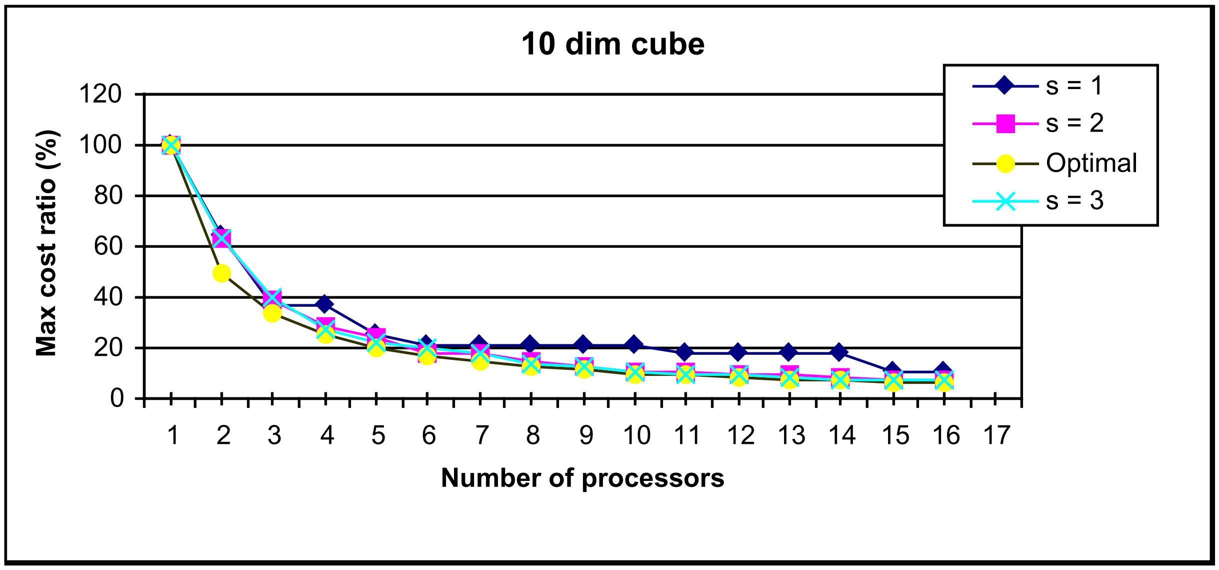


Figure 11: Experimental Results for Top-down Method: d=9 and d=10.

dim	$s=1$	$s=2$	$s=3$
7	75%	12%	11%
8	70%	17%	11%
9	46%	25%	20%
10	66%	16%	13%

Table 1: Average deviation of the weight of the largest subproblem from optimal value ($p \in \{1 \ldots 16\}$ and $d \in \{7 \ldots 10\}$).

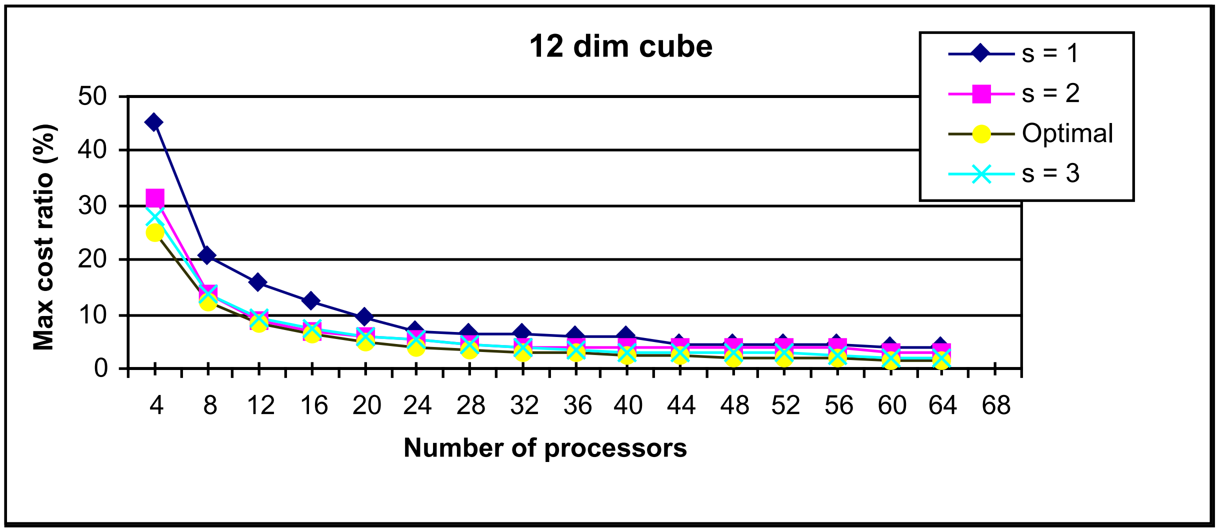


Figure 12: Experimental Results for Top-down Method: d=12, $p \in \{1 \ldots 64\}$.

rithm 3 for larger values of p, $p \in \{1, \ldots, 64\}$. Notice that, even with a relatively large number of processors Algorithm 3 generates load balanced subcube computations. With an oversampling ratio of $s = 3$ the algorithm achieves a load balance that is, on average, within 22% from optimal.

In summary, our experiments showed that our parallel top-down data cube partitioning algorithm performs very well and generates load balanced subproblems with only a small amount of oversampling.

8 Conclusion

We presented two different partitioning strategies, one for top-down and one for bottom-up cube algorithms. Both strategies balance the loads assigned to the processors and minimize communication overhead. Subcube computations are carried out using existing sequential, external memory data cube algorithms. Both partitioning approaches are architecture independent and can be implemented on any parallel machines composed of processors connected via an interconnection fabric. Experimental results indicate that our partitioning strategies produce very efficient parallel data cube generation methods.

References

[1] S. Agarwal, R. Agarwal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Srawagi. On the computation of multi-dimensional aggregates. In *Proc. 22nd VLDB Conf.*, pages 506–521, 1996.

[2] R.I. Becker, Y. Perl, and S.R. Schach. A shifting algorithm for min-max tree partitioning. *J. ACM*, (29):58–67, 1982.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of 1999 ACM SIGMOD Conference on Management of data*, pages 359–370, 1999.

[4] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant. Bulk synchronous parallel computing - A paradigm for transportable software. In *Proc. of the 28th Hawaii International Conference on System Sciences. Vol. 2: Software Technology*, pages 268–275, 1995.

[5] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 106–115, 1997.

[6] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 889–890, 1999.

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *ACM Symp. Computational Geometry*, pages 298–307, 1993.

[8] F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing i/o complexity by simulating coarse grained parallel algorithms. In *Proc. 13th International Parallel Processing Symposium (IPPS'99)*, pages 14–20, 1999.

[9] X. Deng and P. Dymond. Efficient routing and message bounds for optimal parallel algorithms. In *Proc. Int. Parallel Processing Symposium*, 1995.

[10] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R Ramakrishnan. Computation of multidimensional aggregates. Technical Report 1314, University of Wisconsin, Madison, 1996.

[11] L.G. Valiant et. al. *in J. v. Leeuwen (ED.), Handbook of Theoretical Computer Science*, chapter General Purpose Parallel Architectures, pages 943–972. MIT Press/Elsevier, 1990.

[12] P. Flajolet and G.N. Martin. Probablistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[13] G.N. Frederickson. Optimal algorithms for tree partitioning. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 168–177, 1991.

[14] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

[15] S. Goil and A. Choudhary. A parallel scalable infrastructure for OLAP and data mining. In *Proc. International Data Engineering and Applications Symposium (IDEAS'99)*, Montreal, August 1999.

[16] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 1–12, 1996.

[17] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, April 1997.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[19] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):205–216, 1996.

[20] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[21] K. Hwang. *Advanced Computer Architecture, Parallelism, Scalability, Programmability.* McGraw-Hill, New York, 1993.

[22] MPI. Message passing interface standard. http://www-unix.mcs.anl.gov/mpi.

[23] Y. Perl and U. Vishkin. Efficient implementation of a shifting algorithm. *Disc. Appl. Math.*, (12):71–80, 1985.

[24] K.A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 23rd VLDB Conference*, pages 116–125, 1997.

[25] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[26] A. Shukla, P. Deshpende, J.F. Naughton, and K. Ramasamy. Storage estimation for mutlidimensional aggregates in the presence of hierarchies. In *Proc. 22nd VLDB Conference*, pages 522–531, 1996.

[27] J.F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proc. of 3rd Italian Conf. on Algorithms and Complexity (CIAC-97)*, volume LNCS 1203, pages 229–240. Springer, 1997.

[28] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, (33):103–111, 1990.

[29] D.E. Vengroff and J.S. Vitter. I/o-efficient scientific computation using tpie. In *Proc. Goddard Conference on Mass Storage Systems and Technologies*, pages 553–570, 1996.

[30] J.S. Vitter. External memory algorithms. In *Proc. 17th ACM Symp. on Principles of Database Systems (PODS '98)*, pages 119–128, 1998.

[31] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory. i: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[32] Y. Zhao, P.M. Deshpande, and J.F.Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Conf.*, pages 159–170, 1997.