Parallelizing The Data Cube [*†]

Frank Dehne [‡] **Todd Eavis** [§] **Susanne Hambrusch** [¶]
Andrew Rau-Chaplin [‖]

Keywords: OLAP, data cube, parallel processing, partitioning, load balancing.

[*]A preliminary version of this paper has been published in the proceedings of the 8th International Conference on Database Theory (ICDT 2001), London, UK, January 2001.

[†]Research partially supported by the Natural Sciences and Engineering Research Council of Canada and the National Science Foundation (Grant 9988339-CCR).

[‡]School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, frank@dehne.net, www.dehne.net

[§]Faculty of Computer Science, Dalhousie University, Halifax, Canada B3H 1W5 eavis@cs.dal.ca

[¶]Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA, seh@cs.purdue.edu

[‖]Faculty of Computer Science, Dalhousie University, Halifax, Canada B3H 1W5 arc@cs.dal.ca, www.cs.dal.ca/~arc

Abstract

This paper presents a general methodology for the *efficient parallelization of existing data cube construction algorithms*. We describe two different partitioning strategies, one for top-down and one for bottom-up cube algorithms. Both partitioning strategies assign subcubes to individual processors in such a way that the loads assigned to the processors are balanced. Our methods reduce inter-processor communication overhead by partitioning the load in advance instead of computing each individual group-by in parallel as is done in previous parallel approaches. In fact, after the initial load distribution phase, each processor can compute its assigned subcube without any communication with the other processors. Our methods enable code reuse by permitting the use of existing sequential (external memory) data cube algorithms for the subcube computations on each processor. This supports the transfer of optimized sequential data cube code to a parallel setting.

The bottom-up partitioning strategy balances the number of single attribute external memory sorts made by each processor. The top-down strategy partitions a weighted tree in which weights reflect algorithm specific cost measures like estimated group-by sizes. Both partitioning approaches can be implemented on any *shared disk* type parallel machine composed of p processors connected via an interconnection fabric and with access to a shared parallel disk array.

We have implemented our parallel *top-down* data cube construction method in C++ with the MPI message passing library for communication and the LEDA library for the required graph algorithms. We tested our code on an eight processor cluster, using a variety of different data sets with a range of sizes, dimensions, density, and skew. The tests show that our partitioning strategies generate a close to optimal load balance between processors. The actual run times observed show an optimal speedup of p.

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 group-bys 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 [16]. Numerous solutions for generating the data cube have been proposed. One of the main differences between the many solutions is whether they are aimed at sparse or dense relations [4, 17, 21, 22, 28]. Solutions within a 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 and to effectively handle the increase in data sizes, parallel 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.

Balancing the load assigned to different processors and minimizing the communication overhead are the core problems in achieving high performance on parallel systems. At the heart of this paper are two partitioning strategies, one for top-down and one for bottom-up data cube construction algorithms. 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 [1, 4, 22]. The bottom-up partitioning strategy balances the number of single attribute external sorts made by each processor [4]. The top-down strategy partitions a weighted tree in which weights reflect algorithm specific cost measures such as estimated group-by sizes [1, 22].

The advantages of our load balancing methods compared to the previously published parallel data cube construction methods [13, 14] are:

- Our methods reduce inter-processor communication overhead by partitioning the load in advance instead of computing each individual group-by in parallel (as

proposed in [13, 14]). In fact, after our load distribution phase, each processor can compute its assigned subcube without any inter-processor communication.

- Our methods maximize code reuse from existing sequential data cube implementations by using existing sequential data cube algorithms for the subcube computations on each processor. This supports the transfer of optimized sequential data cube code to the parallel setting.

Our partitioning approaches are designed for standard, *shared disk* type, parallel machines: p processors connected via an interconnection fabric where the processors have standard-size local memories and access to a shared disk array. We have implemented and tested our parallel *top-down* data cube construction method. We implemented sequential pipesort [1] in C++, and our parallel top-down data cube construction method (Section 4) in C++ with MPI [2]. We tested our code on an eight processor cluster, using a variety of different data sets with a range of sizes, dimensions, density, and skew. The tests show that our partitioning strategies generate a close to optimal load balance between processors. The actual run times observed show an optimal speedup of p.

The paper is organized as follows. Section 2 describes the parallel machine model underlying our partitioning approaches as well as the input and the output configuration for our algorithms. Section 3 presents our partitioning approach for parallel bottom-up data cube generation and Section 4 outlines our method for parallel top-down data cube generation. In Section 5 we indicate how our top-down cube parallelization can be easily modified to obtain an efficient parallelization of the ArrayCube method [28]. Section 6 presents the performance analysis of our parallel top-down partitioning approach. Section 7 concludes the paper and discusses possible extensions of our methods.

2 Parallel Computing Model

We use the standard *shared disk* parallel machine model. That is, we assume p processors connected via an interconnection fabric where processors have typical workstation size local memories and concurrent access to a shared disk array. For the purpose of

parallel algorithm design, we use the *Coarse Grained Multicomputer* (CGM) model [5, 8, 15, 18, 24]. More precisely, we use the *EM-CGM* model [6, 7, 9] which is a multi-processor version of Vitter's *Parallel Disk Model* [25, 26, 27].

For our parallel data cube construction methods we assume that the d-dimensional input data set R of size N is stored on the shared disk array. The output, i.e. the group-bys comprising the data cube, will be written to the shared disk array. For the choice of output file format, it is important to consider the way in which the data cube will be used in subsequent applications. For example, if we assume that a visualization application will require fast access to individual group-bys then we may want to store each group-by in striped format over the entire disk array.

3 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 *BUC* [4] and *PartitionCube* [part of [21]] 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 [4, 21] 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. As outlined in [4, 21], the total computation time of these methods is dominated by the number of such *single attribute sorts*.

In this section we describe a partitioning of the group-by computations into p independent subproblems. Our goal is to balance the number of single attribute sorts required to solve each subproblem and to ensure that each subproblem has overlapping sort sequences in the same way as for the sequential methods (thereby

avoiding additional work).

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 [21], 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 our partitioning method.

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, where d is the dimension of the the data cube. We refer to i as the *rank* of $B_i(\ldots)$. 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 $d-2$. A complete example for a 4-dimensional data cube with attributes A, B, C, D is shown in Figure 1.

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 $2p$ 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

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

Figure 1: Partitioning For A 4-Dimensional Data Cube With Attributes A, B, C, D.

construction of all group-bys in each Γ_i it follows that every processor performs the same number of single attribute sorts.

Algorithm 1 Parallel Bottom-Up Cube Construction.

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

(1) Calculate Γ_i as described above.

(2) Compute all group-bys in Γ_i using a sequential (external-memory) bottom-up cube construction method.

— End of Algorithm —

Algorithm 1 can easily be generalized to values of p which are not powers of 2. We also note that Algorithm 1 requires $p \leq 2^{d-1}$. This is usually the case in practice. However, 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.

The following four properties summarize the main features of Algorithm 1 that make it load balanced and communication efficient:

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

- The calculation of the group-bys in Γ_i, assigned to processor P_i, requires the same number of single attribute sorts for all $1 \leq i \leq p$.
- The sorts performed at processor P_i share prefixes of attributes in the same way as in [4, 21] and can be performed with disk scans in the same manner as in [4, 21].
- The algorithm requires no inter-processor communication.

4 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, 22], use more detailed group-bys to compute less detailed ones that contain a subset of the attributes of the former. They apply to data sets 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 group-bys, called the *lattice*. An example of a 4-dimensional lattice is shown in Figure 2, 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 group-by 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 to minimize the total computation cost by computing minimum cost matchings between successive layers of the lattice. *PipeHash* uses hash tables instead of 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 permit independent sorts in memory rather than always using external memory sort.

Next, we outline a partitioning approach which generates p independent subproblems, each of which can be solved by one processor using an existing external-memory

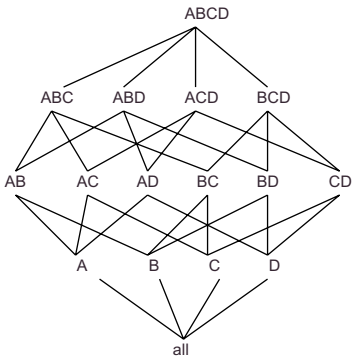

Figure 2: A 4-Dimensional Lattice.

top-down cube algorithm. 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 group-bys in T. This can be done, for example, by using methods described in [11] and [23]. 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 to processor scheduling). 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 I/O 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* [3] defined as follows: 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 [3, 12, 20]. These methods assume that the weights are fixed. Note that, our partitioning problem on

T is different in that, as we cut a subtree T' out of T, an additional cost is introduced because the group-by associated with the root of T' must now be computed from scratch through a separate sort. Hence, when cutting T' out of T, the weight of the root of T' has to be increased accordingly. We have adapted the algorithm in [3] to account for the changes of weights required. This algorithm is based on a *pebble shifting* scheme where k pebbles are shifted down the tree, from the root towards the leaves, determining the cuts to be made. In our adapted version, as cuts are made, the cost for the parent of the new partition is adjusted to reflect the cost of the additional sort. Its original cost is saved in a hash table for possible future use since cuts can be moved many times before reaching their final position. In the remainder, we shall refer to this method as the modified min-max tree k-partitioning.

However, even a perfect min-max k-partitioning does not necessarily result in a partitioning of T into subtrees of *equal size*, and nor does it address tradeoffs arising from the number of subtrees assigned to a processor. We use tree-partitioning as an initial step for our partitioning. 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, $s \geq 1$. Then, we use a *"packing heuristic"* to determine which subtrees belong to which processors, assigning s subtrees to every processor. Our packing heuristic considers the weights of the subtrees and pairs subtrees by weights to control the number of subtrees. It consists of s matching phases in which the p largest subtrees (or groups of subtrees) and the p smallest subtrees (or groups of subtrees) are matched up. Details are described in Step 2b of Algorithm 2.

Algorithm 2 Sequential Tree-partition(T, s, p).

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). 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 modified 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 Υ_i, $1 \leq i \leq sp$, where initially $\Upsilon_i = \{T_i\}$. The weight of Υ_i is defined as the total weight of the trees in Υ_i.

(2b) For $j = 1$ to $s - 1$

- Sort the Υ-sets by weight, in increasing order. W.l.o.g., let Υ_1 , ... , $\Upsilon_{sp-(j-1)p}$ 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. Our method provides a framework for parallelizing any sequential top-down data cube algorithm. An outline of our approach is given in the following Algorithm 3.

Algorithm 3 Parallel Top-Down Cube Construction.

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

(1) Apply the storage estimation method in [23] and [11] to determine the approximate sizes of all group-bys in T.

(2) Select a sequential top-down cube construction method (e.g., *PipeSort*, *Pipe Hash*, or *Overlap*) and compute the spanning tree T of the lattice as used by this method. Compute the weight of each node of T: the estimated 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 Σ_1, ..., Σ_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.

— End of Algorithm —

Our performance results described in Section 6 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 important result since a small value of s is crucial for optimizing performance.

5 Parallel Array-Based Data Cube Construction

Our method in Section 4 can be easily modified to obtain an efficient parallelization of the *ArrayCube* method presented in [28]. 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 [28]. 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 *ArrayCube* method can now be parallelized by simply applying Algorithm 3 with T being the *MMST*. More details will be given in the full version of this paper.

6 Experimental Performance Analysis

We have implemented and tested our parallel *top-down* data cube construction method presented in Section 4. We implemented sequential pipesort [1] in C++, and our parallel top-down data cube construction method (Section 4) in C++ with MPI [2]. Most of the required graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [19]. Still, the implementation took one person year of full time work. We chose to implement our parallel *top-down* data cube construction method rather than our parallel *bottom-up* data cube construction method because the former has more tunable parameters that we wish to explore. As parallel hardware platform, we use a cluster consisting of a front-end machine and eight processors. The front-end machine is used to partition the lattice

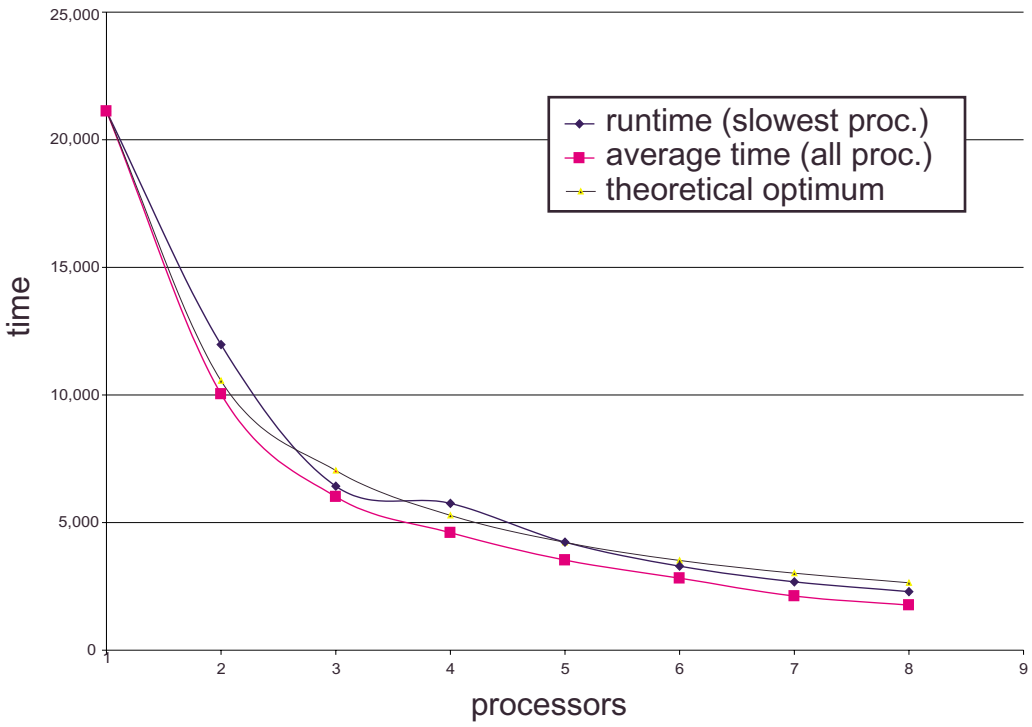

Figure 3: Running Time In Seconds As A Function Of The Number Of Processors. (Fixed Parameters: Data Size = 1,000,000 Rows. Dimensions = 7. Experiments Per Data Point = 5.)

and distribute the work among the other 8 processors. The front-end machine is an IBM Netfinity server with two 9 GB SCSI disks, 512 MB of RAM and a 550-MHZ Pentium processor. The processors are 166 MHZ Pentiums with 2G IDE hard drives and 32 MB of RAM, except for one processor which is a 133 MHZ Pentium. The processors run LINUX and are connected via a 100 Mbit Fast Ethernet switch with full wire speed on all ports. Clearly, this is a very low end, older, hardware platform. However, for our main goal of studying the speedup obtained by our parallel method rather than absolute times, this platform is sufficient. In fact, the speedups measured on this low end cluster are lower bounds for the speedup that our software would achieve on newer and more powerful parallel machines.

Figure 3 shows the running time observed as a function of the number of processors used. For the same data set, we measured the sequential time (sequential pipesort [1]) and the parallel time obtained through our parallel top-down data cube

construction method (Section 4), using an oversampling ratio of $s = 2$. The data set consisted of 1,000,000 records with dimension 7. Our test data values were uniformly distributed over 10 values in each dimension. Figure 3 shows the running times of the algorithm as we increase the number of processors. There are three curves shown. The *runtime* curve shows the time taken by the slowest processor (i.e. the processor that received the largest workload). The second curve shows the *average time* taken by the processors. The time taken by the front-end machine, to partition the lattice and distribute the work among the compute nodes, was insignificant. The *theoretical optimum* curve shown in Figure 3 is the sequential pipesort time divided by the number of processors used.

We observe that the *runtime* obtained by our code and the *theoretical optimum* are essentially identical. That is, for an oversampling ratio of $s = 2$, an optimal speedup of p is observed. (The anomaly in the *runtime* curve at $p = 4$ is due to the slower 133 MHZ Pentium processor.)

Interestingly, the *average time* curve is always below the *theoretical optimum* curve, and even the *runtime* curve is sometimes below the *theoretical optimum* curve. One would have expected that the *runtime* curve would always be above the *theoretical optimum* curve. We believe that this *superlinear speedup* is caused by another effect which benefits our parallel method: improved I/O. When sequential pipesort is applied to a 10 dimensional data set, the lattice is partitioned into pipes of length up to 10. In order to process a pipe of length 10, pipesort needs to write to 10 open files at the same time. It appears that under LINUX, the number of open files can have a considerable impact on performance. For 100,000 records, writing them to 4 files each took 8 seconds on our system. Writing them to 6 files each took 23 seconds, not 12, and writing them to 8 files each took 48 seconds, not 16. This benefits our parallel method, since we partition the lattice first and then apply pipesort to each part. Therefore, the pipes generated in the parallel method are considerably shorter.

Figure 4 shows the running times of our top-down data cube parallelization as we increase the data size from 100,000 to 1,000,000 rows. The main observation is that the parallel *runtime* increases slightly more than linear with respect to the data size which is consistent with the fact that sorting requires time $O(n \log n)$. Figure 4 shows that our parallel top-down data cube construction method scales gracefully

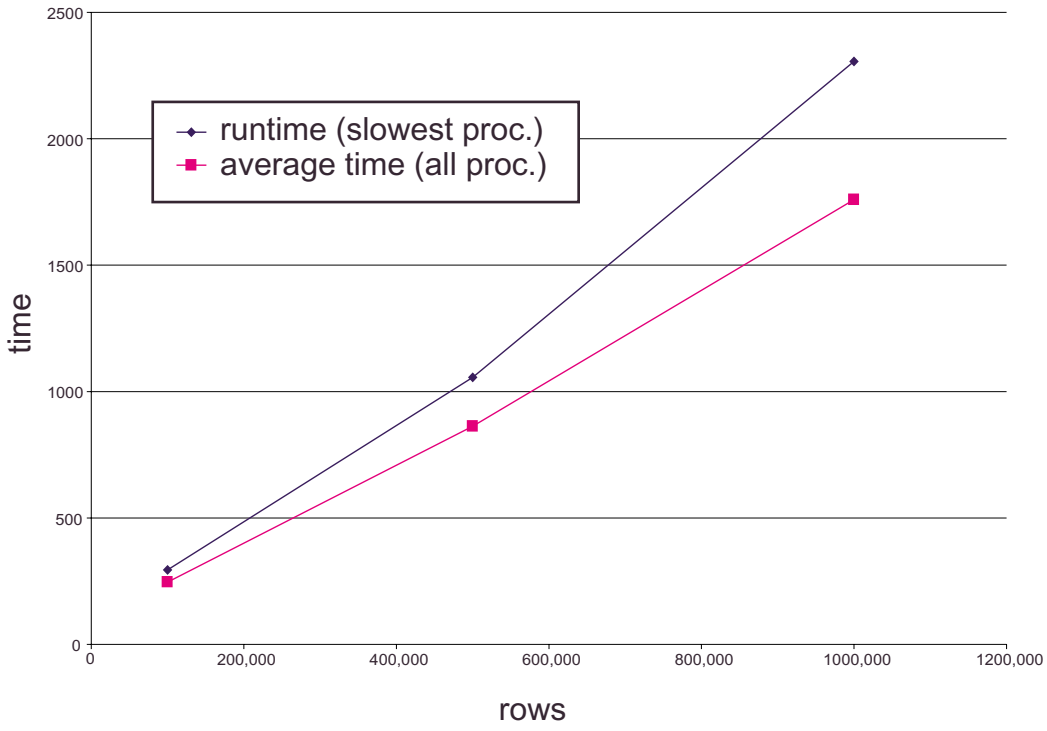

Figure 4: Running Time In Seconds As A Function Of The Data Size. (Fixed Parameters: Number Of Processors = 8. Dimensions = 7. Experiments Per Data Point = 5.)

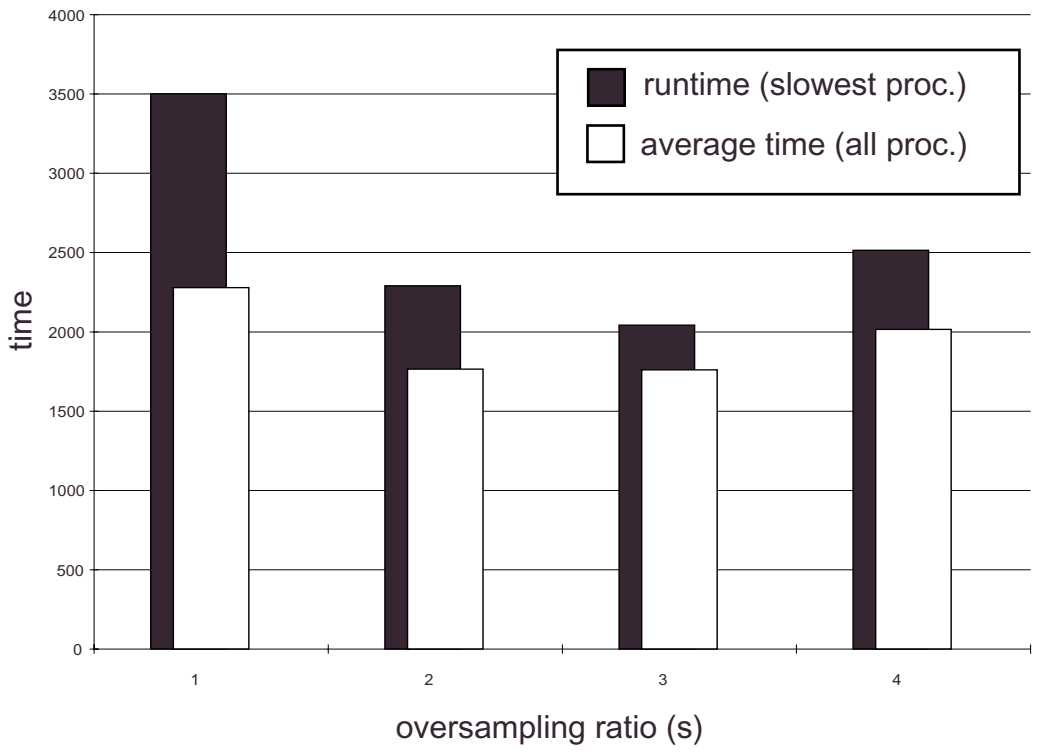

Figure 5: Running Time In Seconds As A Function Of The Oversampling Ratio s. (Fixed Parameters: Data Size = 1,000,000 Rows. Number Of Processors = 8. Dimensions = 7. Experiments Per Data Point = 5.)

with respect to the data size.

Figure 5 shows the running time as a function of the oversampling ratio s. We observe that, for our test case, the parallel *runtime* (i.e. the time taken by the slowest processor) is best for $s = 3$. This is due to the following tradeoff. Clearly, the workload balance improves as s increases. However, as the total number of subtrees, $s \times p$, generated in the tree partitioning algorithm increases, we need to perform more sorts for the root nodes of these subtrees. The optimal tradeoff point for our test case is $s = 3$. It is important to note that the oversampling ratio s is a tunable parameter. The best value for s depends on a number of factors. What our experiments show is that $s = 3$ is sufficient for the load balancing. However, as the data set grows in size, the time for the sorts of the root nodes of the subtrees increases more than linear whereas the effect on the imbalance is linear. For substantially larger data sets, e.g. 1G rows, we expect the optimal value for s to be $s = 2$.

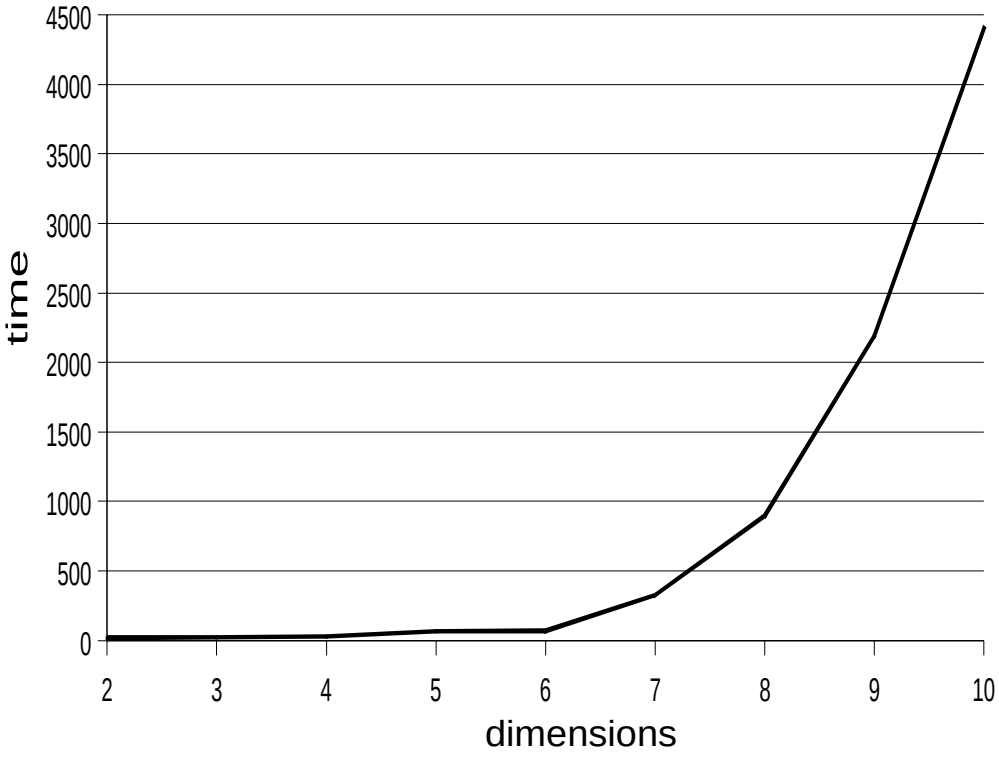

Figure 6: Running Time In Seconds As A Function Of The Number Of Dimensions. (Fixed Parameters: Data Size = 200,000 Rows. Number Of Processors = 8. Experiments Per Data Point = 5.)

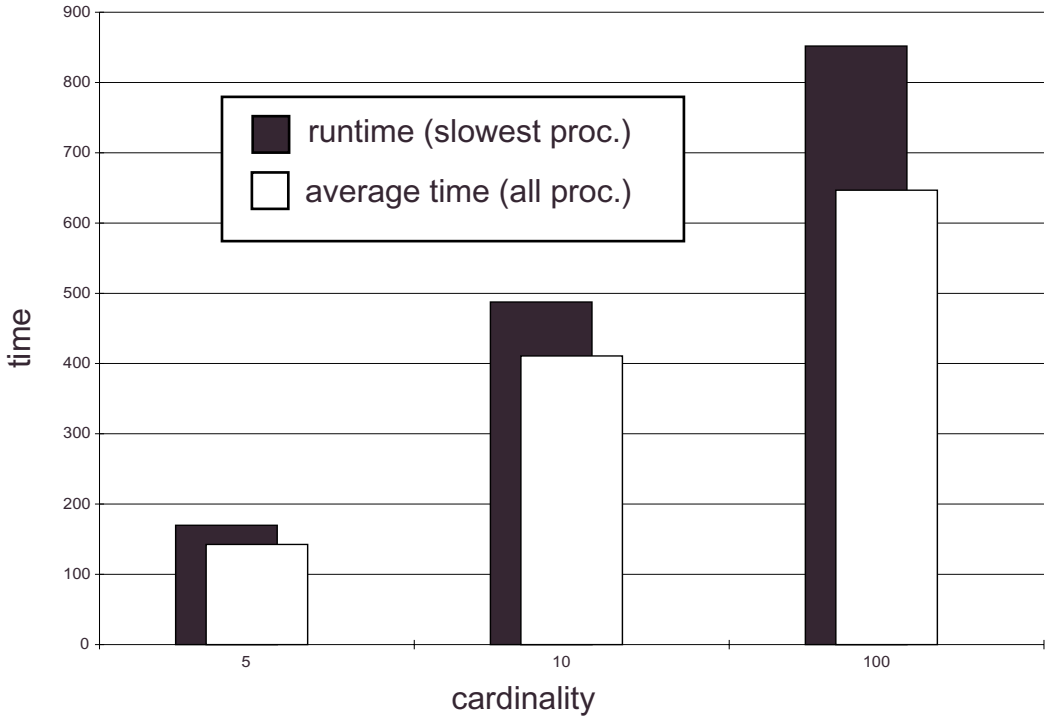

Figure 7: Running Time In Seconds As A Function Of The Cardinality, i.e. Number Of Different Possible Data Values In Each Dimension. (Fixed Parameters: Data Size = 200,000 Rows. Number Of Processors = 8. Dimensions = 7. Experiments Per Data Point = 5.)

Figure 6 shows the running time of our top-down data cube parallelization as we increase the dimension of the data set from 2 to 10. Note that, the number of group-bys that must be computed grows exponentially with respect to the dimension of the data set. In Figure 6, we observe that the parallel running time grows essentially linear with respect to the output size. We also tried our code on very high dimensional data where the size of the output becomes extremely large. For example, we executed our parallel algorithm for a 15-dimensional data set of 10,000 rows, and the resulting data cube was of size more than 1G.

Figure 7 shows the running time of our top-down data cube parallelization as we increase the cardinality in each dimension, that is the number of different possible data values in each dimension. Recall that, top-down pipesort [1] is aimed at dense data cubes. Our experiments were performed for 3 cardinality levels: 5, 10, and 100

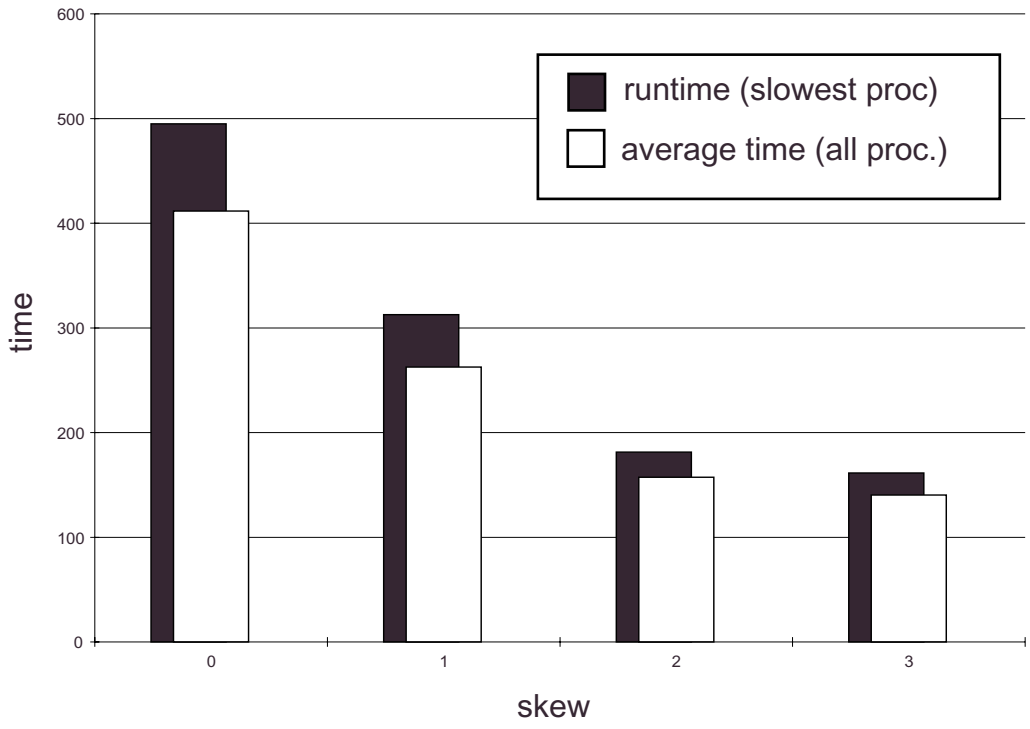

Figure 8: Running Time In Seconds As A Function Of The Skew Of The Data Values In Each Dimension, Based On ZIPF. (Fixed Parameters: Data Size = 200,000 Rows. Number Of Processors = 8. Dimensions = 7. Experiments Per Data Point = 5.)

possible values per dimension. The results shown in Figure 7 confirm our expectation that the method performs better for denser data.

Figure 8 shows the running time of our top-down data cube parallelization for data sets with skewed distribution. We used the standard ZIPF distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 3$. Since data reduction in top-down pipesort [1] increases with skew, the total time observed is expected to decrease with skew which is exactly what we observe in Figure 8. Our main concern regarding our parallelization method was how balanced the partitioning of the tree would be in the presence of skew. The main observation in Figure 8 is that the relative difference between *runtime* (slowest processor) and *average time* does not increase as we increase the skew. This appears to indicate that our partitioning method is robust in the presence of skew.

7 Conclusion

We presented two different, partitioning based, data cube parallelizations for standard shared disk type parallel machines. Our partitioning strategies for bottom-up and top-down data cube parallelization balance the loads assigned to the individual processors, where the loads are measured as defined by the original proponents of the respective sequential methods. Subcube computations are carried out using existing sequential data cube algorithms. Our top-down partitioning strategy can also be easily extended to parallelize the ArrayCube method. Experimental results indicate that our partitioning methods are efficient in practice. Compared to existing parallel data cube methods, our parallelization approach brings a significant reduction in inter-processor communication and has the important practical benefit of enabling the re-use of existing sequential data cube code.

A possible extension of our data cube parallelization methods is to consider a *shared nothing* parallel machine model. If it is possible to store a duplicate of the input data set R on each processor's disk, then our method can be easily adapted for such an architecture. This is clearly not always possible. It does solve most of those cases where the total output size is considerably larger than the input data set; for example *sparse* data cube computations. As reported in [21], the data cube can be several hundred times as large as R. Sufficient total disk space is necessary to store the output (as one single copy distributed over the different disks) and a p times duplication of R may be smaller than the output. Our data cube parallelization method would then partition the problem in the same way as described in Sections 3 and 4, and subcube computations would be assigned to processors in the same way as well. When computing its subcube, each processor would read R from its local disk. For the output, there are two alternatives. Each processor could simply write the subcubes generated to its local disk. This could, however, create a bottleneck if there is, for example, a visualization application following the data cube construction which needs to read a single group-by. In such a case, each group-by should be distributed over all disks, for example in striped format. To obtain such a data distribution, all processors would not write their subcubes directly to their local disks but buffer their output. Whenever the buffers are full, they would be permuted over the network.

In summary we observe that, while our approach is aimed at shared disk parallel machines, its applicability to shared nothing parallel machines depends mainly on the distribution and availability of the input data set R. An interesting open problem is to identify the "ideal" distribution of input R among the p processors when a fixed amount of replication of the input data is allowed (i.e., R can be copied r times, $1 \leq r < p$).

8 Acknowledgements

The authors would like to thank Steven Blimkie, Khoi Manh Nguyen, Thomas Pehle, and Suganthan Sivagnanasundaram for their contributions towards the implementation described in Section 6. The first, second, and fourth author's research was partially supported by the Natural Sciences and Engineering Research Council of Canada. The third author's research was partially supported by the National Science Foundation under Grant 9988339-CCR.

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] Argonne National Laboratory, http://www-unix.mcs.anl.gov/mpi/index.html. *The Message Passing Interface (MPI) standard.*

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Max Planck Institute. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/.

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

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.