Coarse Grained Parallel On-Line Analytical Processing (OLAP) For Data Mining

Frank Dehne[1], Todd Eavis[2], and Andrew Rau-Chaplin[2]

[1] Carleton University, Ottawa, Canada,
frank@dehne.net,
WWW home page: http://www.dehne.net
[2] Dalhousie University, Halifax, Canada,
eavis@cs.dal.ca, arc@cs.dal.ca,
WWW home page: http://www.cs.dal.ca/~arc

Abstract. We study the applicability of coarse grained parallel computing model (CGM) to on-line analytical processing (OLAP) for data mining. We present a general framework for the CGM which allows for the efficient parallelization of existing data cube construction algorithms for OLAP. Experimental data indicate that our approach yield optimal speedup, even when run on a simple processor cluster connected via a standard switch.

1 Introduction

During recent years, there has been tremendous growth in the data warehousing market. Despite the sophistication and maturity of conventional database technologies, the ever-increasing size of corporate databases, coupled with the emergence of the new global Internet "database", suggests that new computing models may soon be required to fully support many crucial data management tasks. In particular, the exploitation of parallel algorithms and architectures holds considerable promise, given their inherent capacity for both concurrent computation and data access.

Data warehouses can be described as *decision support systems* in that they allow users to assess the evolution of an organization in terms of a number of key data attributes or dimensions. Typically, these attributes are extracted from various operational sources (relational or otherwise), then cleaned and normalized before being loaded into a relational store. By exploiting multi-dimensional views of the underlying data warehouse, users can "drill down" or "roll up" on hierarchies, "slice and dice" particular attributes, or perform various statistical operations such as ranking and forecasting. This approach is referred to as Online Analytical Processing or OLAP.

2 Coarse Grained Parallel OLAP

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 [10]. The resulting data structures can then be used to dramatically accelerate visualization and query tasks associated with large information sets. 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 [3, 11, 13, 15, 17]. 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 CGM model ([5]) of parallel computation which allows 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, 3, 15]. The bottom-up partitioning strategy balances the number of single attribute external sorts made by each processor [3]. The top-down strategy partitions a weighted tree in which weights reflect algorithm specific cost measures such as estimated group-by sizes [1, 15].

The advantages of our load balancing methods compared to the previously published parallel data cube construction methods [8, 9] are as follows. 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 [8, 9]). 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.

The following is a high-level outline of our coarse grained parallel top-down data cube construction method:

1. Construct a lattice for all 2^d views.
2. Estimate the size of each of the views in the lattice.
3. To determine the cost of using a given view to directly compute its children, use its estimated size to calculate (a) the cost of scanning the view and (b) the cost of sorting it.

4. Using the bipartite matching technique presented in [14], reduce the lattice to a spanning tree that identifies the appropriate set of prefix-ordered sort paths.
5. Partition the tree into $s \times p$ sub-trees (s = oversampling ratio).
6. Distribute the sub-trees over the p compute nodes.
7. On each node, use the sequential Pipesort algorithm to build the set of local views.

In the following, we provide a more detailed description of the implementation. We first describe the code base and supporting libraries.

The Code Base

In addition to MPI, we chose to employ a C++ platform in order to more efficiently support the growth of the project. With the expansion of the code base and the involvement of a number of independent developers, several of whom were in geographically distinct locations, it was important to employ an object-oriented language that allowed for data protection and class inheritance. One notable exception to the OOP model, however, was that the more familiar C interface to MPI functions was used.

The LEDA Libraries

A full implementation of our parallel datacube algorithms was very labour intensive. We chose to employ third-party software libraries so that we could focus our own efforts on the new research. After a review of existing packages, we selected the LEDA libraries because of the rich collection of fundamental data structures (including linked lists, hash tables, arrays, and graphs), the extensive implementation of supporting algorithms, and the C++ code base [12]. Although there is a slight learning curve associated with LEDA, the package has proven to be both efficient and reliable.

The OOP Framework

Having incorporated the LEDA libraries into our C++ code base, we were able to implement the lattice structure as a LEDA graph, thus allowing us to draw upon a large number of built-in graph support methods. In this case, we have sub-classed the graph template to permit the construction of algorithm-specific structures for node and edge objects. As such, a robust implementation base has been established; additional algorithms can be "plugged in" to the framework simply by sub-classing the lattice template and (a) over-riding or adding methods and (b) defining the new node and edge objects that should be used as template parameters.

In the current implementation, the base lattice has been sub-classed so as to augment the graph for the sort-based optimization. For each view/node, we estimate its construction cost in two formats: as a linear scan of its parent and as a complete resorting of its parent. Since these cost assessments depend upon accurate estimates of the sizes of the views themselves, the inclusion of a view estimator is required.

Probabilistic View Size Estimator

A number of inexpensive algorithms have been proposed for view size estimation [16]. The simplest merely entails using the product of the cardinalities of each dimension to place an upper bound on the size of each cuboid. A slightly more sophisticated technique computes a partial datacube on a randomly selected sample of the input set. The result is then "scaled up" to the appropriate size. although both approaches can give reasonable results on small, uniformly distributed datasets, they are not as reliable on real world data warehouses. Consequently, the use of probabilistic estimators that rely upon a single pass of the dataset have been suggested. As described in [16], our implementation builds upon the counting algorithm of Flajolet and Martin [7]. Essentially, we concatenate the d dimension fields into bit-vectors of length L and then hash the vectors into the range $0 \ldots 2^L - 1$. The algorithm then uses a probabilistic technique to count the number of distinct records (or hash values) that are likely to exist in the input set. To improve estimation accuracy, we employ a universal hashing function [4] to compute k hash functions, that in turn allows us to average the estimates across k counting vectors.

The probabilistic estimator was fairly accurate, producing estimation error in the range of 5–6 % with 256 hash functions. However, its running time on large problems was disappointing. The main problem is that, despite an asymptotic bound of $O(n * 2^d)$, the constants hidden inside the inner computing loops are enormous (i.e, greater than one million!). For the small problems described in previous papers, this is not an issue. In high dimension space, it is intractable. The running time of the estimator extends into weeks or even months. Considerable effort was expended in trying to optimize the algorithm. All of the more expensive LEDA structures (strings, arrays, lists, etc.) were replaced with efficient C-style data types. Despite a factor of 30 improvement in running time, the algorithm remained far too slow. We also experimented with the GNU-MP (multi-precision) libraries in an attempt to capitalize on more efficient operations for arbitrary length bit strings. Unfortunately, the resulting estimation phase was still many times slower than the construction of the views themselves. At this point, it seems unlikely that the Flajolet and Martin estimator is viable in high dimension space.

A simple view estimator

We needed a fast estimator that could be employed even in high dimension environments. We chose to use the technique that bounds view size as the product of dimension cardinalities. We also improved upon the basic estimate by exploiting the fact that a child view can be no bigger than the smallest of its potential parents. The estimated size for a node is the minimum of 1) the product of the cardinalities of the node's dimensions, and 2) the estimated size of the node's smallest parent. However, additional optimizations that incorporate intermediate results are not possible since a parallel implementation prevents us from sequentially passing estimates up and down the spanning tree. Section 3 discusses the results obtained using this version of the view estimator.

Computing the Edge Costs

As previously noted, the values produced by the estimator only represent the sizes of each view, not the final edge costs that are actually placed into the lattice. Instead, the algorithm uses the view estimate to calculate the potential cost of scanning and sorting any given cuboid. An appropriate metric must be experimentally developed for every architecture upon which the datacube algorithm is run. For example, on our own cluster, an in-memory multi-dimensional sort is represented as $(d + 2)/3) * (n \log n)$, where d is the current level in the lattice. At present, we are working on a module that will be used to automate this process so that appropriate parameters can be provided without manually testing every architecture.

Constructing the spanning tree

Once the lattice has been augmented with the appropriate costs, we apply a weighted bipartite matching algorithm that finds an appropriate set of sort paths within the lattice (as per [14]). Working bottom-up, matching is performed on each pair of contiguous levels in order to identify the most efficient distribution of sort and scan orders that can be used to join level i to level $i-1$. The matching algorithm itself was provided by LEDA and required only minor modification for inclusion in our design.

Min-Max Partitioning

As soon as the bipartite matching algorithm has been executed, we partition the lattice into a set of k sub-trees using the min-max algorithm proposed by Becker, Schach and Perl [2]. The original algorithm is modified slightly since it was designed to work on a graph whose costs were assigned to the nodes, rather than the edges. Furthermore, a *false root* with zero cost must be added since the algorithm iterates until the root partition is no longer the smallest sub-tree. The min-max algorithm performs nicely in practice and, in conjunction with the over-sampling factor mentioned earlier, produces a well balanced collection of sub-trees (see [6] for a more complete analysis).

Once min-max terminates, the k sub-trees are collected into p sets by iteratively combining the largest and smallest trees (with respect to construction cost). Next, each sub-tree is partitioned into a set of distinct prefix-ordered sort paths, and then packaged and distributed to the individual network nodes.

The local processor decompresses the package into its composite sort paths and performs a pipesort on each pipeline in its assigned workload. No further communication with the root node is required from this point onward.

Local Pipesorts

Pipesort consists of two phases. In the first round, the root node in the list is sorted in a given multi-dimensional order. In phase two, we perform a linear pass through the sorted set, aggregating the most detailed records into new records that correspond to the granularity level of each cuboid in the sort path. As the new records are produced, they are written directly to disk. For example, if we sort the data in the order *ABCD*, we will subsequently create the *ABCD*, *ABC*, *AB*, and *A* views as we traverse the sorted set.

Although we originally exploited LEDA's array sorting mechanism to sort the root node in memory, we have since re-written the sort using the C library routines in order to to maximize performance. At present, all input sorting is performed in main memory. In the future, we expect to incorporate robust external memory sorting algorithms into the project.

3 Performance Evaluation

We now discuss the performance of our goarse grained parallel data cube implementation. As parallel hardware platform, we used a cluster consisting of a front-end machine and eight processors. The front-end machine is used to partition the lattice 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 1 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 data cube construction method, 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 1 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 1 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 2 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 $s = 2$.

4 Conclusion

As data warehouses continue to grow in both size and complexity, so too will the opportunities for researchers and algorithm designers who can provide powerful, cost-effective OLAP solutions. In this paper we have discussed the implementation of a coarse grained parallel algorithm for the construction of a multi-dimensional data model known as the datacube. By exploiting the strengths of existing sequential algorithms, we can pre-compute all cuboids in a load balanced and communication efficient manner. Our experimental results have demonstrated that the technique is viable, even when implemented in a *shared* *nothing* cluster environment. In addition, we have suggested a number of opportunities for future work, including a parallel query model that utilizes packed r-trees. More significantly perhaps, given the relatively paucity of research currently being performed in the area of parallel OLAP, we believe that the ideas we have proposed represent just a fraction of the work that might lead to improved data warehousing solutions.

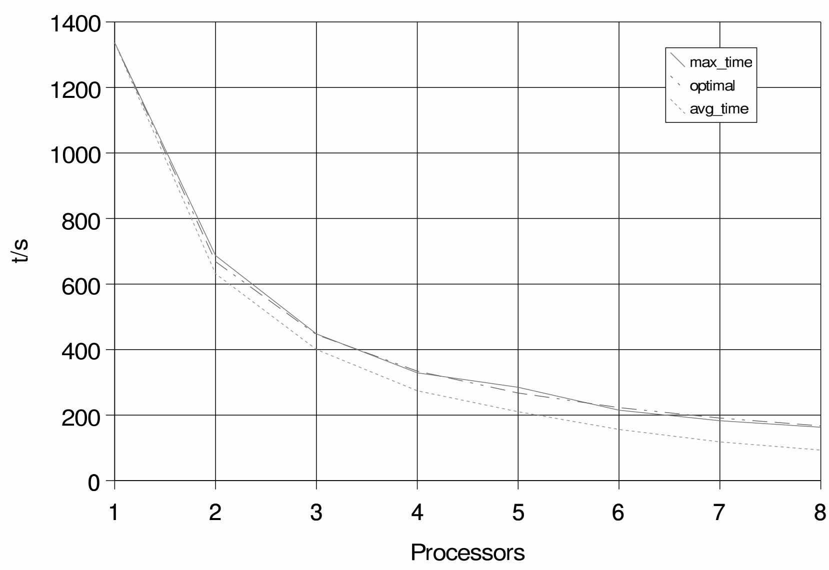

Fig. 1. Running Time In Seconds As A Function Of The Number Of Processors. (Fixed Parameters: Data Size = 1,000,000 Rows. Dimensions = 7. Experiments Per Data Point = 5.)

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. Becker, S. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *Journal of the 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. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1996.
5. F. Dehne. Guest editor's introduction, special issue on "coarse grained parallel algorithms". *Algorithmica*, 24(3/4):173–176, 1999.

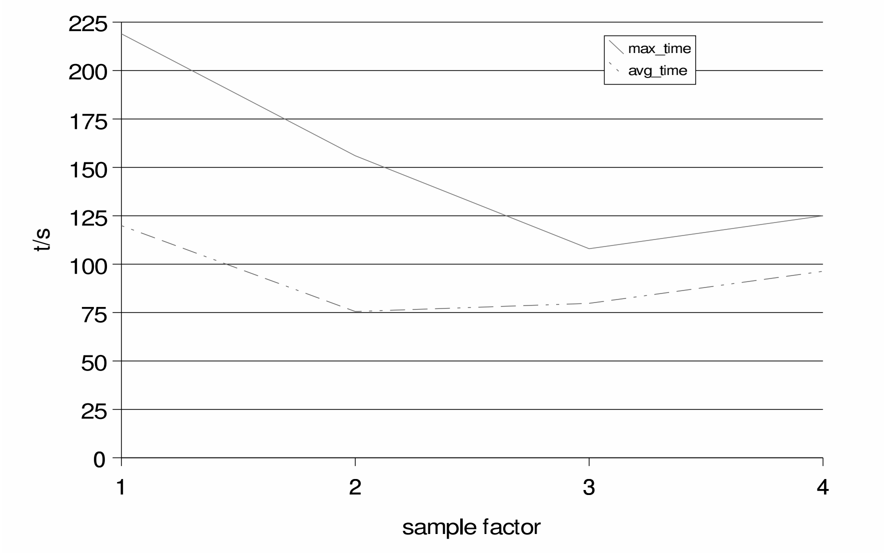

Fig. 2. Running Time In Seconds As A Function Of The Oversampling Ratio s. (Fixed Parameters: Data Size = 1,000,000 Rows. Number Of Processors = 8. Dimensions = 7. Experiments Per Data Point = 5.)

6. F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.
7. P Flajolet and G. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
8. S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.
9. 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.
10. 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.
11. 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.
12. Max Planck Institute. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/.
13. K.A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 23rd VLDB Conference*, pages 116–125, 1997.
14. S. Sarawagi, R. Agrawal, and A.Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.
15. S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

16. A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.
17. 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.