Parallelizing XQuery In a Cluster Environment

Xiaogang Li, Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{xgli,agrawal}@cse.ohio-state.edu

ABSTRACT

In this paper, we report on a parallel implementation of XQuery. As XQuery is being used for processing large datasets, and/or for compute-intensive applications, efficiency of XQuery implementations is becoming an important issue. Our work has specifically focused on scientific data processing and data mining applications. Parallelization of this class of XQuery queries involves a number of challenges, which can be summarized as follows: 1) Developing efficient and correct data distribution techniques for semi-structured datasets. 2) Parallelization of generalized reduction functions found in scientific data processing and data mining applications, which particularly requires the synthesis of global reduction functions. 3) Translation of XQuery processing to an imperative language like C/C++, which may be needed for invoking efficient parallel communication libraries.

In this paper, we report our solutions towards the above problems. By implementing the techniques in a compiler and generating code based on a C++ SAX parser and the Message Passing Interface (MPI), we are able to achieve efficient parallel execution on a cluster of machines.

1. INTRODUCTION

In this paper, we report on a parallel implementation of XQuery. As XQuery is being used for processing large datasets, and/or for compute-intensive applications, efficiency of XQuery implementations is becoming an important issue. Distributing the dataset on a set of computers in a cluster, and parallelizing the processing of XQuery can be an approach for speeding up the execution. Though parallelization of database queries has been widely studied in the past [7, 19], we are not aware of any existing work on parallelization of XQuery or XML query processing.

Our work on parallelization of XQuery has focused on scientific data processing and data mining applications. The motivation for this is as follows. XML has rapidly gained acceptance in industry and in a variety of scientific and engineering areas. For example, many well known scientific data formats, such as HDF5 [10] and NetCDF [20], now provide XML interfaces for data exchange. Techniques for supporting XML as a logical view on flat-file datasets have also been developed [14]. As XML is also beginning to play an important role in grid computing area, particularly through its use in SOAP [3], its popularity in scientific computing areas is expected to continue or accelerate.

To query and process data from scientific experiments or simulations that is stored physically or logically as XML, XQuery designed by W3C [2] can be an ideal language. This is because of its declarative nature and powerful features. XQuery is a high-level language like SQL, but it also supports more advanced and complex features such as types and recursive functions. Scientific data processing and data mining applications often target large datasets, and involve significant computations as well. Thus, efficiency of query processing becomes extremely important for this class of applications.

Parallelization of this class of XQuery queries involves a number of challenges, which can be summarized as follows: 1) Developing efficient and correct data distribution techniques for semi-structured datasets. 2) Parallelization of generalized reduction functions found in scientific data processing and data mining applications, which particularly requires the synthesis of global reduction functions. 3) Translation of XQuery processing to an imperative language like C/C++, which may be needed for invoking efficient parallel communication libraries.

In this paper, we report our solutions towards the above problems. By implementing the techniques in a compiler and generating code based on a C++ SAX parser and the Message Passing Interface (MPI), we are able to achieve efficient parallel execution on a cluster of machines.

2. SYSTEM OVERVIEW AND DATA DISTRIBUTION

Our target environment is a shared-nothing architecture, i.e., cluster of machines, each with an attached disk. The overall processing involves a local reduction phase and a global combination phase. During local reduction phase, each node processes its local XML document in a streaming fashion based on a SAX XML parser, and local output elements are updated. In the global combination phase, local output elements from each node are broadcast to all other nodes using MPI calls, and are merged to generate the final output.

To achieve efficient parallel execution and local balance on shared-nothing architectures, an important issue is distributing the XML data evenly among all processing nodes. The method we use is based upon the technique Marian and Simeon used for optimizing the memory in a serial environment [16]. Assume there are \( N \) computers in the cluster. Our distribution algorithm will generate \( N \) XML documents from the original document, each of which contains (approximately) the same number of elements.

Specifically, we first statically analyze the actual paths that will be referred to in the queries we are likely to execute on this dataset. To perform the data distribution, we use the set of actual paths to filter the original document. Any node that is not on these paths is not included in the final documents. We use the common prefix among the set of paths as the distribution criteria. For example, the common path prefix for the Satellite processing application is /data/pixel. We distribute the pixel nodes evenly into \( N \) documents, while the parent nodes are duplicated in each document.

Clearly, our approach has some limitations. We require that multiple occurrence must be defined in the Schema for the nodes corresponding to the common path prefix. Also, similar to [16], we can only handle forward axis. Although preprocessing and distribution may introduce extra costs for query evaluation, we believe such preprocessing...
strategy may still be beneficial for many queries, for the following reasons. 1) By removing unnecessary nodes and distributing useful nodes evenly, the memory requirement used to process the original query can be decreased dramatically. 2) For queries that are compute-bound, the time saved by parallel processing may be much more than the time spent on preprocessing. 3) Finally, one distribution scheme may be shared for multiple similar queries. Thus, the cost of preprocessing and distribution may be amortized over the execution of multiple queries.

3. PARALLELIZATION OF XQUERY

In this section, we discuss parallel code generation of XQuery on shared-nothing or cluster architectures. For our target class of applications, parallelization can be done by dividing data and computation evenly across the nodes, and having each node execute local reduction operations on the data items it owns. After this, each node broadcasts the result of its local reduction to all other nodes, and a global reduction phase is invoked to combine the local results. Depending on the application, we may need to iterate over local and global reduction phases.

As described in the previous section, we process SAX events at the local reduction phase and invoke MPI calls for message passing at the global reduction phase. To support SAX parsing and invoking MPI calls, we must translate the original XQuery code to an imperative language such as C/C++. To facilitate generation of such code, we introduce a representation called Generalized Nested Loops (GNL). This representation helps exploit the imperative nature of XML parsers such as SAX. Though our current implementation has been carried out on top of SAX and MPI, our code generation techniques are more general. For example, besides MPI, we believe GNL can facilitate code generation for using parallel interfaces and middleware such as OpenMP [6], Global Arrays [18], and ADR [5, 4].

3.1 Definition of GNL

**Definition.** A GNL $N$ is a four tuple $(I, E_p, E_c, S)$ where,
1. $I$ is the index variable bound to $N$,
2. $E_p$ is the location path of the corresponding XPath expression,
3. $E_c$ is the predicate expression of $E_p$ (if any), and
4. $S$ is the loop body, which is an ordered sequence of operations performed on any tuple bound to the index variable $I$.

The semantic meaning of a GNL is similar to a foreach loop or iterator, which iterates over tuples specified by the XPath expression. For every element in the target XPath expression $E_p$ filtered with $E_c$, it is bound to the variable $I$, and each statement $s \in S$ is executed according to their order in $S$.

By definition, the tuple stream that a GNL operates on is specified by its path expression $E_p$, and the aggregation operations are specified by the statement sequence in its body. With such a syntax structure, code generation for C++ functions triggered by SAX events and MPI calls becomes easier.

3.2 GNL Formation and Aggregation Rewrite

We use the results from the earlier static analysis for data distribution to translate the original query into GNL representation. Specifically, we use the common prefix of path expressions as $E_p$ of the GNL, since this common prefix specifies the sequence of tuples that are processed in parallel. References to other path expressions are merged into the body of the GNL. For aggregation functions, we move them into the GNL by using function inlining. Internal functions, such as `sum`, `count`, and `average`, can be easily rewritten in an iterative fashion. For example, we can rewrite `sum()` as `tmp = tmp + v;` where `tmp` is a temporary variable and $v$ is the tuple. For a user-defined function, including recursive functions, we apply a previously developed static analysis technique to extract an associative operation from the definition of the function [15]. The basic idea is to examine the syntax tree from leaves and apply tree pattern matching to retrieve the desired sub-tree. Our algorithm can only deal with linear recursive functions.

The GNL generated for Satellite Data Processing is shown in Figure 1. Here, we use `data/pixel` as the path expression for GNL, and rewrite the recursive function into an iterative reduction operation.

3.3 Parallel Code Generation

Consider a XQuery query represented by GNLs. To parallelize structure, we need to generate parallel code for local processing on each node, and a global combination across the nodes. We now describe how we generate code for each of these phases.

3.3.1 Local Reduction

The data instances owned by a processor are read. A local reduction function specifies how, after processing one data instance, a local output element is updated. The result of this processing must be independent of the order in which data instances are processed on each processor.

The GNL generated from a query serves as a convenient intermediate representation for code generation. GNL uses a nested loop structure, which is commonly supported in imperative languages. Since the
SAX parser internally supports streaming traversal, and generates a series of streaming events according to the document order, the explicit traversal defined in a GNL does not appear in the final code.

Specifically, we use the following strategy to evaluate a GNL: 1) For a given GNL $E$ with $E_p = /x/y/z$, when the event $\text{endElement}(E)$ is triggered, the body of $E$ is executed once. 2) For any path expressions $p$ nested inside GNLs $E$, the processing for $p$ is always performed before the processing for $E$.

The code generated for local reduction of Satellite Processing is shown in Figure 2. Though our goal is to process queries on the fly, certain XML elements may need to buffered. For example, a node may be issued for output after a condition involving its children nodes is evaluated. Clearly, because buffering requires memory, we want to buffer as few elements as possible. We use a filtering and projection technique similar to the one described in [12]. However, our technique is simpler because we do not process blocking operators like $\text{join}$ and sort. Specifically, we buffer a node with its entire subtree if it is used as output. Also, we buffer any leaf node $v$ whose value is referred to in the query. In such a case, if $v$ is not in the subtree of a node used as output, the buffer of $v$ is immediately dispatched when reference of $v$ is finished. Using the GNL representation, we check the location path $E_p$ and $\text{filter}$ $E_v$ of each GNL, and only mark a node for buffering if it is referred to in the body of that GNL.

3.3.2 Global Combination

The local output elements on all processors are combined using a global reduction function. Consider the output of reduction $O$ and $O_1$ computed by two processors after their local reduction phase. We need a function $f$ to update $O$ with the values computed in $O_1$, i.e., to perform the computation $O = f(O, O_1)$, where $f$ defines the same associative and commutative operation as local reduction.

The code generation for the local reduction module was relatively simple, since the operations to be performed locally are explicitly defined within the body of a GNL. Although the global combination function performs the same associative and commutative operation that is defined in a GNL body, we cannot use the body of a GNL simply as the global combination function. This is because the input to a global combination function is copies of output from other nodes, and not the data tuples from the original dataset. Therefore, more sophisticated analysis is required for generating a global reduction function.

Our compiler uses an algorithm for this purpose which we summarize below. The basic idea is to extract a part of the code that impacts the output, either directly or indirectly, from the GNL. This is done by a static analysis technique called program slicing [22]. After a program slice is extracted, certain statements are either removed or modified. These include: 1) Statements that are only control dependent on the input data. Because input data is not visible at the global phase, control dependence on input data tuples can be discarded. 2) Statements that create data dependence on the input data. For the same reason, we can replace each instance of the input data (bound to the GNL variable) with references to remote copies of output from other nodes.

The remaining statements are used in the body of the global combination function. The detailed algorithm is shown in Figure 3. The global reduction function for the Satellite Data Processing application is shown in Figure 4.

4. EXPERIMENTAL RESULTS

We used four different queries for our experiments. The first two queries involve processing of scientific datasets. satellite involves processing of data collected from satellites and creating composite images [5]. vscope is an application to support interactive viewing and processing of digitized data arising from tissue specimens [8]. The other two queries are taken from the XMark benchmark [21]. Specifically, we used the queries 5 and 20 from this set, as they used aggregation operations and matched the processing structure we have targeted. Our experiments were conducted on a cluster of 900 MHz Pentium III machines, connected by a Gigabit Ethernet. Each node has 256 MB of main memory and 18GB of local disk. We ran our experiments on 1, 2, 4, 8 nodes of the cluster.

For the satellite and vscope applications, we used XML datasets with sizes of 200 MB and 400 MB. Since all of the elements in these two applications are traversed to generate the results, projection does not alter the sizes of these two datasets significantly. The output of satellite is a $5 \times 5$ array and the output of vscope is a 16 $\times$ 16 array. Thus, the relative overhead of communication is quite small for these two applications. The results are shown in Figures 5 and 6. These speedups are almost linear in both the cases.

The results from the parallelization of XMark queries are shown in Figures 7 and 8. For these queries, we used the standard data generator with factors 5 and 10, which resulted in datasets of size 500 MB and 1000 MB, respectively. For Query 5, the projection shrinks the two original datasets to 4.3 MB and 8.6 MB, respectively. For the Query 20, the sizes of the datasets after projection are 32 MB and 64 MB. The ratio of communication to local processing is somewhat higher for these queries. We observe good speedups for the Query 20 on both the datasets. Since the sizes of the final datasets for Query 5 are relatively small, communication and other overheads start to dominating on 4
and 8 nodes, which leads to smaller speeds.

5. CONCLUSIONS

In this paper, we have reported on a parallel implementation of XQuery. As XQuery is being used for processing large datasets, and/or for compute-intensive applications, efficiency of XQuery implementation is becoming an important issue. Our work has specifically focused on scientific data processing and data mining applications.

Our current results from scientific data processing and XMark queries have been encouraging. However, many issues remain open with respect to parallelization of XQuery. We need to consider different types of XQuery queries, and investigate the issues involved in parallelizing them.

6. REFERENCES


Figure 5: Parallel Performance: satellite

Figure 6: Parallel Performance: vmscope

Figure 7: Parallel Performance: XMARK Q20

Figure 8: Parallel Performance: XMARK Q5