Exploiting Domain Specific High-level Runtime Support for Parallel Code Generation*

Xiaogang Li Ruoming Jin Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{xgli,jinr,agrawal}@cis.ohio-state.edu

Abstract

Data mining techniques focus on finding novel and useful patterns or models from large datasets. Because of the volume of the data to be analyzed, the amount of computation involved, and the need for rapid or even interactive analysis, data mining applications require the use of parallel machines.

We believe that parallel compilation technology can be used for providing high-level language support for carrying out data mining implementations. Our study of a variety of popular data mining techniques has shown that they can be parallelized in a similar fashion. In our previous work, we have developed a middleware system that exploits this similarity to support distributed memory parallelization and execution on disk-resident datasets. This paper focuses on developing a data parallel language interface for using our middleware's functionality. Compiler techniques for translating this dialect to a middleware specification are presented.

We present a detailed experimental evaluation of our compiler using apriori association mining, k-means clustering, and k-nearest neighbor classifiers. Our experimental results show that: 1) compiler generated parallel data mining codes achieve high speedups in a cluster environment, 2) the performance of compiler generated codes is quite close to the performance of manually written codes, and 3) simple additional optimizations like inlining can further reduce the gap between compiled and manual codes.

1 Introduction

Analysis of large datasets for extracting novel and useful models or patterns, also referred to as *data mining*, has emerged as an important area within the last decade [16]. Because of the volume of data analyzed, the amount of computation involved, and the need for rapid or even interactive response, data mining tasks require the use of parallel machines and careful management of the memory hierarchy.

This paper focuses on the use of compiler technology for offering high-level support for implementing data mining algorithms. In our previous work, we have developed a middleware for rapidly developing data mining implementations [20, 19, 21]. The middleware exploits the similarity between the algorithms to offer a runtime support and a programming interface. In this paper, we present compiler techniques to translate a high-level code to a middleware specification. The particular language that we will use is a data parallel dialect of Java. The main new analysis required is for extracting a global reduction function from a data parallel loop. In addition, a number of other interesting issues arise in handling data mining codes and generating middleware code, and are addressed in our research.

We have implemented our techniques using the Titanium infrastructure from Berkeley [29]. We present experimental results from apriori association mining, k-means clustering, and k-nearest neighbor classifier. We have experimented with disk-resident datasets for each of these codes. Our experimental results show that 1) compiler generated parallel data mining codes achieve high speedups in a cluster environment, 2) the performance of compiler generated codes is quite close to the performance of manually written codes, and 3) simple additional optimizations like inlining can further reduce the gap in performance of compiled and manual codes.

The rest of the paper is organized as follows. The language dialect we use is presented in Section 2. The interface and functionality of our middleware is described in Section 3. Our compiler techniques are presented in Section 4. Experimental evaluation of our prototype compiler is the topic of Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 Data Parallel Language Support

The structure of the various data mining algorithms we have studied, including those for association mining, clustering, nearest neighbor searches, bayesian networks and decision tree construction, can be viewed as comprising

*This work was supported by NSF grant ACR-9982087, NSF CAREER award ACR-9733520, and NSF grant ACR-0130437.

```
Interface Reducinterface {
   {* Any object of any class implementing *}
   {* this interface is a reduction variable *}
}
public class KmPoint implements Disk-resident {
   double x1, x2, x3;
   KmPoint (String buffer) {
      {* constructor for copying to/from a buffer *}
   }
}
public class Kcenter implements Reducface {
   static double [] x1,x2,x3;
   static double[] meanx1, meanx2, meanx3;
   static long[] count;
   Kcenter (String buffer) {
      {* constructor for copying to/from a buffer *}
   }
   void Finalize() {
      for(i=0; i<k; i++) {
         x1[i]=meanx1[i]/count[i];
         x2[i]=meanx2[i]/count[i];
         x3[i]=meanx3[i]/count[i];
      }
   }
   void Assign(KmPoint point,int i,double dis) {
      meanx1[i]+=point.x1;
      meanx2[i]+=point.x2;
      meanx3[i]+=point.x3;
      count[i]+=1;
   }
}
```

```
public class Kmeans {

   public static void main(String[] args) {
      Point< 1 > lowend = .. ;
      Point< 1 > hiend = .. ;
      RectDomain< 1 > InputDomain=[lowend:hiend];
      KmPoint[1d] Input=new KmPoint[InputDomain];

      while(not_converged) {

         foreach (p in InputDomain) {
            min=9.999E+20;
            for (i=0; i < k; i++) {
               int dis = Kcenter.distance(Input[p],i);
               if( dis < min) {
                  min=temp;
                  minindex=i;
               }
            }
            Kcenter.Assign(Input[p],minindex,min);
         }
      Kcenter.Finalize();
      }
   }
}
```

Figure 1. k-means clustering Expressed in Data Parallel Java

generalized reduction operations [17]. Processing for generalized reductions consists of three main steps: (1) retrieving data items of interest, (2) applying application-specific transformation operations on the retrieved input items, and, (3) mapping the input items to output items and aggregating all the input items that map to the same output data item. Most importantly, aggregation operations involve *commutative* and *associative* operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated. Such a common structure makes data mining algorithms a suitable target for a compilation framework.

We now describe a data parallel dialect of Java that can be used for expressing parallel algorithms for common data mining tasks. Though we propose to use a dialect of Java as the source language for the compiler, the techniques we will be developing will be largely independent of Java and will also be applicable to suitable extensions of other languages, such as C or C++.

We use three main directives in our data parallel dialect. These are for specifying a multi-dimensional collections of objects, a parallel for loop, and a reduction interface. The first two have been commonly used in other object-oriented parallel systems like Titanium [29], HPC++ [6], and Concurrent Aggregates [11]. The concept of reduction interface is, to the best of our knowledge, novel to our approach. The choice of these directives is motivated by the structure of data mining algorithms we described earlier.

Rectdomain: A rectdomain is a collection of objects of the same type such that each object in the collection has a *coordinate* associated with it, and this coordinate belongs to a pre-specified rectilinear section.

Foreach loop: A foreach loop iterates over objects in a rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.

Reduction Interface: Any object of any class implementing the reduction interface acts as a *reduction variable* [17]. The semantics of a reduction variable are analogous to those used in version 2.0 of High Performance Fortran (HPF-

2) [17]. A reduction variable has the property that it can only be updated inside a *foreach* loop by a series of operations that are associative and commutative. Furthermore, the intermediate value of the reduction variable may not be used within the loop, except for self-updates.

Another interface we use is *Disk-resident*. Any class whose objects are either read or written from disks must implement this interface. For any class which implements the reduction interface, or represents objects that are disk-resident, we expect a constructor function that can read the object from a string. In the case of a class that implements the reduction interface, such constructor function is used for facilitating interprocessor communication. Specifically, the code for the constructor function is used for generating code for copying an object to a message buffer and copying a message buffer to an object. Similarly, for any dataset which is either read or written to disks, the constructor function is used to generate code that reads or writes the object.

The data parallel Java code for k-means clustering is shown in Figure 1. k is the number of clusters that need to be computed. An object of the class `KmPoint` represents a three-dimensional point. The variable `Input` represents a one-dimensional array of points, which is the input to the algorithm. In each iteration of the `foreach` loop, one point is processed and the cluster whose center is closest to the point is determined. The function `Assign` accumulates coordinates of all points that are found to be closest to the center of a given cluster. It also increments the count of the number of points that have been found to be closest to the center of a given cluster. The function `Finalize` is called after the `foreach` loop. It determines the new coordinates of the center of a cluster, based upon the points that have been assigned to the cluster. The details of the test for termination condition are not shown here.

3 Middleware Interface and Parallelization Support

Our compiler heavily uses our middleware system for distributed memory parallelization and I/O optimizations in processing disk-resident datasets. The middleware's functionality and interface are described in this section.

The middleware interface exploits the similarity among parallel versions of data mining algorithms. A programmer using the middleware directly needs to write the following functions. Most of these functions can be easily extracted from a sequential version that processes main memory resident datasets.

Subset of Data to be Processed: In many case, only a subset of the available data needs to be analyzed for a given task. For example, while creating associations rules from customer purchase record at a grocery store, we may be interested in processing records obtained in certain months, or for customers in a certain age group, etc.

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system. The reduction object is maintained in the main memory.

Global Reductions: The reduction objects on all processors are combined using a global reduction function. MPI calls for handling the communication are made by the runtime system. However, functions for copying the reduction object to a buffer and from a buffer to the reduction object are expected as part of the middleware interface.

Iterator: Parallel implementations of the applications we target comprise one or more distinct sets of local and global reduction functions, which are be invoked in an iterative fashion. An iterator function contains the loop that invokes local and global reduction functions.

The middleware support for distributed memory parallelization is relatively simple because of the structure of the applications it targets. After data has been distributed between different nodes, each node can execute local reduction functions on data items it owns. After each invocation of local reduction function, local copies of reduction objects on each node are broadcasted to all other nodes, and local copies of reduction objects from all other nodes are received on each node. This communication is facilitated by the middleware. After the communication phase, global reduction function is invoked on each node. The result of global reduction is then broadcasted to all processors.

Our middleware's support for I/O optimizations for processing disk-resident datasets is based upon an earlier system called Active Data Repository (ADR), which was developed for scientific data intensive applications [9, 8].

4 Compiler Techniques

We now present compilation techniques for translating a data mining code written in the data parallel dialect to a middleware specification.

In generating the code for the middleware interface, the compiler needs to: 1) Generate an *Iterator* function that invokes local and global reduction functions. 2) For each data parallel loop that updates a reduction object, generate a) the specification of the subset of the data processed in that loop, b) the local reduction function, c) the global reduction function, and d) functions for copying reduction object to and from message buffers.

Conceptually, the most difficult problem is extracting the global reduction function from the body of the data parallel loop. Here, we describe our solution towards this problem.

{* *Data Dependence Analysis* *}
Let O be the reduction object and let e be input element processed
For each statement that updates a data member of O:
  Let the statement update $O.x$
  Classify the value assigned to $O.x$ as:
    (a) A function of loop constants and data members of O
    (b) A function $g(e)$, $g()$ does not depend upon O
    (c) An express of the form $O.x \;\; op \;\; g(e)$,
        op is an associative and commutative operator, and $g()$ does not depend upon O
    (d) Any other expression
Let S be the set of statement that update any data member of O
If any statement in S is of type (d), *exit()*

{* *Control Dependence Analysis* *}
Identify all predicates that any statement in S is control dependent on
For each such predicate
  Classify into:
    (I) a loop constant predicate
    (II) predicate that is not a loop constant

{* *Code Generation* *}
For every statement s of type (c)
  Replace the statement by $O.x \;=\; O.x \;\; op \;\; O1.x$
  Remove any control flow of type (II)
  Insert a loop around s that ranges over all fields that x can represent
For every statement s of type (b)
  Replace the statement by $O.x \;=\; O1.x'$ where x' ranges over all fields where $g(e)$ can be assigned
For every statement s of type (a) and (b)
  If s is control dependent upon a field l of the input
    If an expression $h(l)$ is assigned to $O.x$
      replace l in control predicate by $h^{-1}(O1.x')$, where x' ranges over all fields that x can represent
    Else
      Remove the conditional
Let $S(r)$ denote the set of statements in S that are data
      or control dependent upon a $O1.x'$, where x' has a range r
If a statement belongs to two distinct sets $S(r)$ and $S(t)$, *exit()*
If a statement belonging to $S(r)$ is dependent upon a statement in $S(t)$, $r \neq t$, *exit()*
For each such set $S(r)$
  Insert a loop in which $O1.x'$ iterates over r around all statements in $S(r)$
Remove all statements that no statement in S is data or control dependent on

Figure 2. Algorithm for Synthesizing Global Reduction Function from the Body of Data Parallel Loop

4.1 Global Reduction Analysis

Consider the reduction objects O and $O1$ computed by two processors after their local reduction phase. We need a function f to update O with the values computed in $O1$, i.e., to perform the computation $O \;=\; f(O, O1)$.

The algorithm we have developed is presented in Figure 2. Our algorithm can handle significantly more complicated reduction functions than the previous work in this area [5, 14, 15, 24]. To explain the different cases that the algorithm handles, we use three examples, from k-means clustering, apriori association mining, and k-nearest neighbor search, which are shown in Figures 1, 3, and 4, respectively.

The algorithm is divided into three phases. The first phase is data dependence analysis, the second phase is control dependence analysis, and the final phase is code generation. Any expression or predicate whose value remains unchanged across iterations of the data parallel loop is considered a *loop constant* expression or predicate. Note that the value of such an expression can be different over different invocations of the loop.

```
int cnt = t.cnt;
while (−−cnt >= 0) {
   i = t.iids[j] - p.offs;
   if (i < 0) continue;
   if (i >= p.size) return ;
   if ((offset= t.cnt - p.countvector) >= p.upper_used) {
      vect[offset+i] += 1;
   }
   if ( p.chcnt <= 0) continue;
   i += p.offs - p.ID(tree.children[0]);
   if ((i < 0) || (i >= p.chcnt)) continue;
   if (p.children[i] != null)
      LocalReduc( t , p.children[i]);
}
```

```
GlobalReduc() {
   for(i = 0; i < ncands ; i++)
      vect[i] += buf.vect[i];
}
```

Figure 3. Loop Body (left) and Global Reduction Function (right) for Apriori

```
dis = Input_pt.distance(test_x1,test_x2,test_x3);
kbuffer.Insert(Input_pt, dis);

Insert(kPoint kpoint, double dis) {
   i = k - 1 ;
   while( (dis < distance[i]) &&(i >= 0) ) {
      if(i > 0) {
         x1[i] = x1[i-1];
         x2[i] = x2[i-1];
         x3[i] = x3[i-1];
         distance[i] = distance[i-1];
      }
      i = i - 1;
   }
   if (i < k - 1) {
      x1[i+1] = kpoint.x1;
      x2[i+1] = kpoint.x2;
      x3[i+1] = kpoint.x3;
      distance[i+1] = dis;
   }
}
```

```
GlobalReduc() {
   for(j = 0; j < k; j++) {
      i = k - 1;
      while ((buf.distance[j] < distance[i] )
            && (i >= 0)) {
         if(i > 0) {
            x1[i] = x1[i - 1];
            x2[i] = x2[i - 1];
            x3[i] = x3[i - 1];
            distance[i] = distance[i - 1];
         }
         i = i - 1;
      }
      if(i < k-1) {
         x1[i + 1] = buf.x1[j];
         x2[i + 1] = buf.x2[j];
         x3[i + 1] = buf.x3[j];
         distance[i + 1] = buf.distance[j];
      }
   }
}
```

Figure 4. Loop Body (left) and Global Reduction Function (right) for k-nearest neighbors

Consider the body of the loop that processes an element e and updates a reduction object O. Consider any statement in this code that updates a data member of O. If this statement includes any temporary variables that are defined in the loop body itself, we perform forward substitution and replace these temporary variables. After such forward substitution, the update to the data member can be classified as being one of the following: a) assignment to the value of another data member of the reduction object, or an expression involving one or more other data members, and loop constants, b) assignment to $g(e)$, where the g does not involve any members of the reduction object O, c) update using a commutative and associative operator. *op*, such that the data member $O.x$ is updated as $O.x = O.x \; op \; g(e)$, where the function g does not involve any members of the reduction object O, or d) any other expression.

Updates of type (a) can arise in a function like average, where sum, count, and average are the three data members of the reduction object. After computing sum and count, average is computed by dividing sum by count, which is an expression involving other data members. An example of a local reduction function with an update of type (b) is k-nearest neighbors (Figure 4). The reduction object in this code comprises the coordinates of k-nearest neighbors to the given point. As a new point is processed, it may be inserted as one of the nearest neighbors. Updates of type (c) arise frequently in loop bodies of codes that perform reduction computations. Examples include updates to sum and count fields in a function like average, and updates in k-means clustering and apriori association mining (Figure 1 and 3, respectively).

After data dependence analysis, the set of statements which update a data member of the reduction object is denoted by S. The next phase of the algorithm classifies control dependence to any statement in the set S. We consider two types of control dependence, loop constant and non-loop constant.

The final and the major part of the algorithm is actual code generation. Statements in the set S are treated differently based upon both the type of assignment and control dependence.

Consider any statement with an assignment of type (c). We replace the statement $O.x = O.x \ op \ g(e)$ by a statement of form $O.x = O.x \ op \ O1.x$, i.e. the values aggregated are part of the two reduction objects are combined using the same associative and commutative operator. As an example, look at the assignments to `meanx1[i]`, `meanx2[i]`, `meanx3[i]`, and `count[i]` as part of the function `Assign` in Figure 1. In this case, simple symbolic analysis can determine that i can represent values between 0 and $k - 1$. In other words, $O.x$ can represent one of many data members of the reduction object. When processing an element e, the actual data member that are incremented depends upon the element. However, in combining two local reduction objects, we simply need to add corresponding `meanx1[i]`, `meanx2[i]`, `meanx3[i]`, and `count[i]`. Therefore, we insert a loop that iterates over the range of i. We also remove any control predicates that are not loop constant. If there is any loop constant control predicate, it needs to be maintained in the global reduction code. An example of such a predicate will be a conditional for determining whether to compute maximum or minimums during a particular invocation of the loop.

A similar situation arises in apriori association mining (Figure 3). The element or transaction analyzed by the loop body is a list of items. A prefix tree maintains sets of items that can occur with a high frequency. The list of transactions is matched against these sets of items and the count associated with a set is incremented when a match is found. The loop body involves complicated control flow. However, the only update to a reduction element occurs in the statement `vect[offset+i] += 1`. The global reduction code, therefore, simply iterates over the different elements of the array `vect` and adds the corresponding elements.

Next, we consider a statement of type (b). As a concrete example, consider the assignment of `kpoint.x1` to `x1[i+1]` in the loop body shown in Figure 4. Again, simple symbolic analysis can determine that the expression $i+1$ can range between 0 and $k - 1$. We replace this assignment by the assignment `x1[i + 1] = buf.x1[j]`, where j ranges between 0 and $k - 1$. Assignments to `x2`, `x3`, and `distance` are processed similarly.

Note that if the loop body includes a statement of type (a), we leave it unchanged.

For statement of both type (a) and (b), we further consider control dependence on non-loop constants. Suppose any statement of type (a) or (b) is control dependent upon a predicate that involves a field l of the input element. We further check if a function $h(l)$ involving this field is assigned to a data member of the reduction object. An example of such control dependence arises in the k-nearest neighbor code. The variable `dis` is computed from the input element and is assigned to the data member `distance`. In such cases, we replace the occurrence of l in the control predicate by $h^{-1}(O1.x')$, where x' ranges over all fields to which $h(l)$ can be assigned. In our example, we replace `dis` by `buf.distance[j]`, with j ranging from 0 to $k - 1$.

If a statement of type (a) or (b) is control dependent upon a field l that is not assigned to the reduction object, we remove such conditional. An example will be if a field of the input element determines whether the data is valid and needs to be processed further. Such a conditional is not required in combining results from local reductions.

In the k-nearest neighbor example, we get several expressions that involve a variable j that ranges from 0 to $k - 1$. Moreover, the statements involving `buf.x1[j1]`, `buf.x2[j]`, and `buf.x3[j]` are control dependent upon the predicate involving `buf.distance[j]` . For a range r, we compute the set $S(r)$ that includes the statements that are data or control dependent upon a $O1.x'$, where x' has the range r. In generating the code, we insert a single loop that iterates over the range r around these statements.

Two potential complications can arise that our algorithm cannot currently handle. First, a statement belonging to a set $S(r)$ may be dependent upon a statement belonging to the set $S(t)$, $r \neq t$. Second, a single statement may belong to two distinct sets. Our algorithm cannot handle either of these cases. Such cases, however, did not arise in any codes we examined.

5 Experimental Results

We have implemented a prototype compiler incorporating the techniques we have described in this paper. We used the Titanium front-end developed at Berkeley [29]. In this section, we evaluate the compiler using three important data mining algorithms, k-means clustering, apriori association mining, and k-nearest neighbor classifier.

The manually programmed versions we used for our experiments were previously used to evaluate the middleware [20, 19]. The experiments have been conducted on a cluster of workstations. We used 8 Sun Microsystem Ultra Enterprise 450's, with 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each node has a 4 GB system disk and a 18 GB data disk. The data disks are Seagate-ST318275LC with 7200 rotations per minute and 6.9 milli-second seek time. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8.

5.1 Results from k-means Clustering

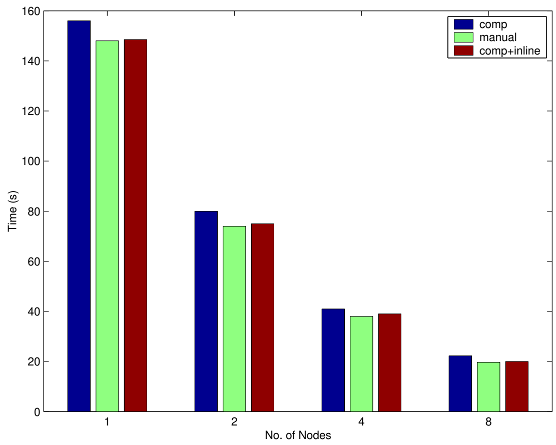

Figure 5. Performance of k-means clustering: k = 3, 1 GB Dataset

The first data mining algorithm we focus on is k-means clustering. The algorithm is parameterized with the value of k (i.e., the number of clusters computed) and the size of the dataset. We used 3 and 100 as the values of k and two datasets, with the size of 1 GB and 2 GB, respectively.

Experimental results from 1 GB dataset and $k = 3$ are presented in Figure 5. The first two versions shown in the figure are compiler generated (`comp`) and manually programmed (`manual`). The relative speedups obtained by the compiler generated version are 1.95, 3.81, and 7.0 on 2, 4, and 8 nodes, respectively. The difference between the compiler and manual versions is less than 10% on 1, 2, and 4 nodes, and 13% on 8 nodes.

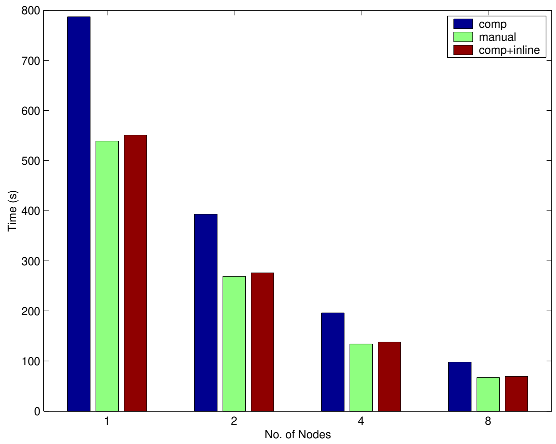

Figure 6. Performance of k-means clustering: k = 100, 1 GB Dataset

We carefully analyzed the reasons for performance differences between the compiler and manual versions. Almost all of the difference comes because the *distance* function invoked inside the inner loop is inlined in the manual version and not in the compiler version. Though such inlining is commonly performed by machine-level compilers, it is was not performed by the `g++` compiler that was used. We created another version, `comp+inline`, in which call to this function is manually inlined. The difference between `manual` and `comp+inline` versions is within 2%.

Figure 6 shows experimental results from the same dataset, but with $k = 100$. The *distance* function is invoked more frequently with the larger value of k. Therefore, there is a greater difference in the performance of `comp` and `manual` versions. After applying inlining manually, the difference reduces to within 3%. Because there is more computation between two phases of communication, the speedups are higher. All three versions have perfect linear relative speedups on 2, 4, and 8 nodes.

Figure 7 shows experimental results from the 2 GB dataset with $k = 100$. The speedups and relative performance of the three versions are almost identical to the ones seen with 1 GB dataset. A comparison of the results from 1 GB and 2 GB datasets shows that as the dataset size is increased, the execution times increase in a linear fashion. As the dataset does not fit in the main memory, the execution times do not increase in a super-linear fashion.

5.2 Results from Apriori Association Mining

The second data mining algorithm we used for evaluating the compiler is apriori association mining.

Figure 8 presents experimental results from a 3 GB dataset. Support level of .25% and confidence level of 90% were used in our experiments. For this dataset and these pa-

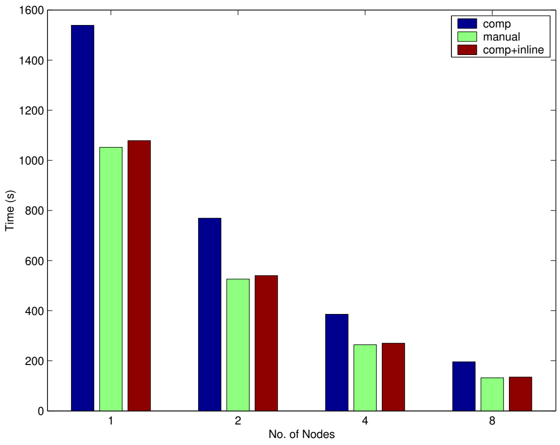

Figure 7. Performance of k-means clustering: k = 100, 2 GB Dataset

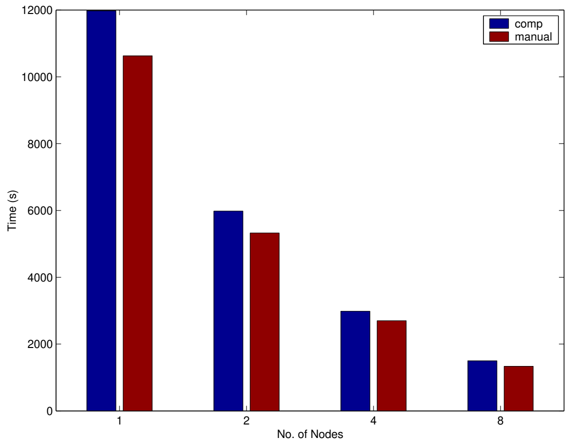

Figure 8. Performance of Apriori Association Mining: 3 GB Dataset, 7 iterations

rameters, 7 iterations were required to obtain final results. Figure 8 shows execution times for 7 iterations. On 8 nodes, `comp` and `manual` versions have relative speedups of 7.98 and 7.97, respectively. The difference in performance is between 10% and 13% in all cases.

Again, we analyzed the reasons for difference in the performance. The compiler generated version performed extra copying of the input data, whereas the manual version analyzed data directly from the read buffer. To further analyze these differences, we experimented with a smaller dataset. Figures 9 and 10 show execution times of compiler and manual versions for the first and the first two iterations, respectively.

In apriori, the first iteration is involves very little computation and is I/O bound. Thus, we expect that the extra cost of copying will make a more significant difference in the execution time of first iteration. Figure 9 validates this conjecture. `comp` version is slower by almost 25% for the first iteration. The second iteration involves a higher level of computation. The execution times for the first two iterations combined, reported in Figure 10, show only a 5% difference in the overall performance of the two versions.

5.3 Results from k-nearest Neighbor Classifier

The last data mining algorithm we use is k-nearest neighbor classifier. Figure 11 presents experimental results from a 1 GB dataset with $k = 100$.

Since k-nearest neighbor is a relatively simple code, there is very little difference between the compiler generated and manual versions. `comp` version is consistently slower than the `manual` version, but by at most 5%.

6 Related Work

To the best of our knowledge, our goal of developing compiler support for data mining algorithms is a unique one. We are not aware of any existing work or project with such a direction.

Our work can be considered as developing an out-of-core Java compiler. Compiler optimizations for out-of-core data-structures have been considered by several projects [7, 22, 23, 26, 27]. These projects have concentrated on stencil computations written in Fortran. Our work is different in considering a different applications class with very different communication and data access patterns, a different language, and targeting an application-class specific middleware as the compiler output.

In earlier work of Agrawal with Ferreira and Saltz, compiler techniques for supporting scientific data intensive applications written in the same dialect of Java were presented [13, 14]. In this earlier effort, a different runtime system and a different class of applications were targeted. Particularly, data mining codes involve more complicated reductions. Therefore, the global reduction analysis we have presented in this paper is much more sophisticated than the algorithm reported earlier for the same problem [14]. Some preliminary ideas towards developing compiler support for data mining were described in an earlier workshop paper [2].

Our proposed compiler work is also different from the various distributed memory compilation [1, 3, 10, 18, 28, 30] projects. We are performing parallelization of generalized reductions over disk-resident datasets, which has not been targeted by any of these projects. Several recent projects have explored the use of Java for numerical and high-performance computing [4, 12, 25, 29]. Our work is distinct in focusing on a different class of applications and performing distributed memory parallelization.

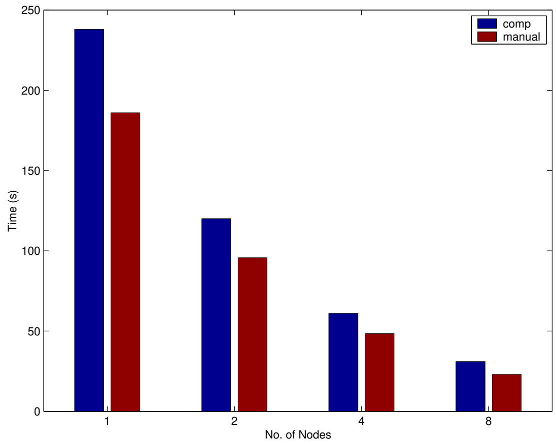

Figure 9. Performance of Apriori Association Mining: 1 GB Dataset, 1 iteration

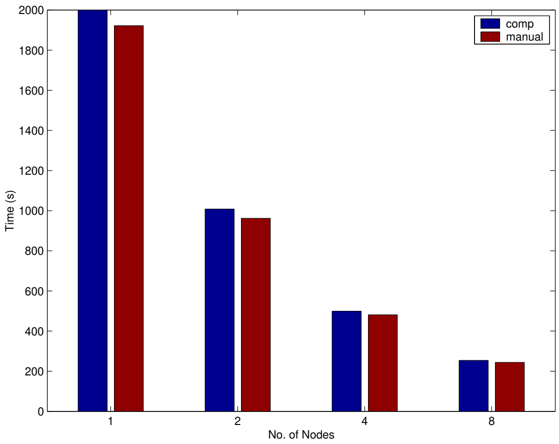

Figure 10. Performance of Apriori Association Mining: 1 GB Dataset, 2 iterations

7 Conclusions

In this paper, we have described and evaluated a compiler for distributed memory parallelization of data mining codes that execute on disk-resident datasets. We have used a data parallel dialect of Java to express this class of applications. Our compiler heavily uses a middleware that we had developed in our earlier work for the same class of applications.

We have evaluated our compiler using three popular data mining algorithms, apriori association mining, k-means clustering, and k-nearest neighbors classifiers. We used disk-resident datasets for each of these three codes. Our experimental results show that 1) the compiler generated parallel data mining codes achieve high speedups in a cluster environment, 2) the performance of compiler generated codes is quite close to the performance of manually written codes, and 3) simple additional optimizations like inlining can further reduce the gap between compiled and manual codes.

References

[1] Vikram Adve and John Mellor-Crummy. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[2] Gagan Agrawal, Ruoming Jin, and Xiaogang Li. Middleware and compiler support for scalable data mining. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[3] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[4] B. L. Blount and S. Chatterjee. An evaluation of java for numerical computing. In *Computing in Object-Oriented Parallel Environments: Proceedings, Second International Symposium, ISCOPE 98*, pages 35–46, 1998.

[5] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoelflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[6] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[7] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[8] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.

[9] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.

[10] Siddhartha Chatterjee, John R. Gilbert, Fred J.E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*,

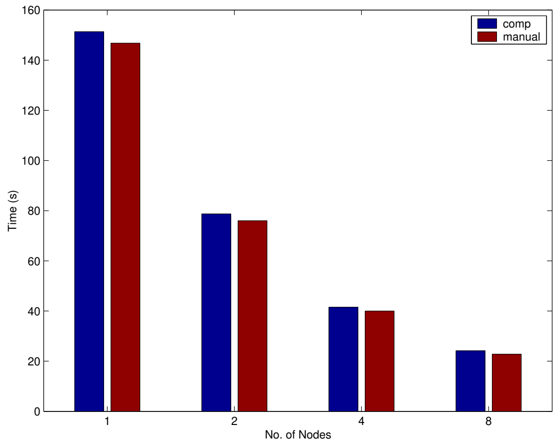

Figure 11. Performance of k-nearest neighbor classifier: 1 GB Dataset, k = 100

pages 149–158, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[11] A.A. Chien and W.J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196. ACM Press, March 1990.

[12] M. Cierniak and W. Li. Just-in-time optimizations for high-performance Java programs. *Concurrency Practice and Experience*, 9(11):1063–73, November 1997.

[13] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive computations. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[14] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiler and runtime analysis for efficient communication in data intensive applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[15] M. W. Hall, S. Amarsinghe, B. R. Murphy, S. Liao, and Monica Lam. Detecting Course-Grain Parallelism using an Interprocedural Parallelizing Compiler. In *Proceedings Supercomputing '95*, December 1995.

[16] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[17] High Performance Fortran Forum. Hpf language specification, version 2.0. Available from http://www.crpc.rice.edu/HPFF/versions/hpf2/files/hpf-v20.ps.gz, January 1997.

[18] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[19] Ruoming Jin and Gagan Agrawal. An efficient implementation of apriori association mining on cluster of smps. In *Proceedings of the workshop on High Performance Data Mining, held with IPDPS 2001*, April 2001.

[20] Ruoming Jin and Gagan Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[21] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.

[22] M. Kandemir, A. Choudhary, and A. Choudhary. Compiler optimizations for i/o intensive computations. In *Proceedings of International Conference on Parallel Processing*, September 1999.

[23] M. Kandemir, A. Choudhary, J. Ramanujam, and M. A.. Kandaswamy. A unified framework for optimizing locality, parallelism, and comunication in out-of-core computations. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):648–662, 2000.

[24] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, April 1998.

[25] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, and Richard D. Lawrence. Parallel data mining in Java. Technical Report RC 21326, IBM T. J. Watson Research Center, November 1998.

[26] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and plementation (OSDI '96)*, Nov 1996.

[27] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[28] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[29] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Libit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 9(11), November 1998.

[30] Hans P. Zima and Barbara Mary Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993. In Special Section on Languages and Compilers for Parallel Machines.