Using General Grid Tools and Compiler Technology for Distributed Data Mining: Preliminary Report*

Wei Du  Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{duw,agrawal}@cis.ohio-state.edu

Abstract

With the wide availability of distributed datasets over the internet, data mining tasks need to be implemented for grid environments. In this paper, we explore the use of a grid application development framework for implementing distributed data mining algorithms. Though a lot of effort has been put in developing distributed data mining algorithms, the use of general grid tools for implementing them has not been investigated. We also describe our preliminary ideas towards using compiler technology for supporting a higher-level interface for developing data mining implementations.

1 Introduction

Distributed Data Mining (DDM) is the process of analyzing geographically dispersed large datasets for extracting novel and interesting patterns or models [12]. Gene and tissue banks, comprising bioinformatics datasets, are popular examples of publically available and geographically distributed datasets. Scientific datasets like those in a gene bank contain critical information whose analysis can lead to important scientific advances.

A user who wants to mine a distributed dataset can first download all the data from their respective host sites, and then apply existing approaches for centralized analysis. However, there are several reasons why this naive approach is usually not feasible. First, the user’s workstations may not have enough storage capacity to store all data. Second, sufficient computational power may not be available at a single site. Finally, network bandwidth limitations may imply a long delay in downloading all data.

To solve these problems, many DDM systems have been developed. These systems could be classified into two categories: the client-server (CS) model and the agent-based model [13]. In the CS model, the user requests are fed into the data mining

---

*This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant ACR-0130437
server which collects data from different locations and brings them into the server, and
the mining server houses the mining algorithms [15]. The advantage of this model is
that DDM server has powerful computational resources which will facilitate resource
intensive data mining tasks. However, this model has a high communication overhead
as it involves transfers of huge volumes of data. In the agent-based model, a variety of
agents coordinate and communicate with each other to perform the various tasks asso-
ciated with distributed analysis [11, 12, 14, 16, 6]. Agent-based technologies address
the scalability problem and can save the cost of transmitting data by moving code to
remote sites where data is available. However, the absence of dedicated data mining
servers and the lack of control over available computational resources at remote sites
are the key limitations of this approach. A common short-coming of the systems using
either of these approaches is that they are built specifically for distributed data mining.

This paper reports our preliminary work in a project that takes a different approach
towards distributed data mining. We believe that general grid application development
tools and techniques [8] should be used for developing mining implementations that
analyze distributed datasets. By exploiting general purpose grid tools, we can leverage
on the research conducted by a large number of grid projects.

We use DataCutter, which is a general grid middleware for data intensive problems
and is based upon filter-stream programming model [2]. In this paper, we present a
case study of developing k-nearest neighbor search using this framework. Our experi-
ments show that filter granularity selection and filter placement are important issues in
implementing data mining algorithms with DataCutter.

We are also developing language and compiler support to offer a high-level inter-
face for DataCutter. A data parallel dialect of Java is proposed which is suitable for
expressing a variety of data mining tasks. This paper also describes our initial ideas for
translating a program written in data parallel Java to a filter-stream program that could
be run on DataCutter.

The rest of this paper is organized as follows. In Section 2, we give an overview of
DataCutter and explain why it is suitable for distributed data mining. Section 3 presents
our case study of implementing k-nearest neighbor search using DataCutter. Our pro-
posed language and compiler framework are presented in Section 4. We conclude in
Section 5.

2 DataCutter: A Middleware for Data Intensive Com-
putations in a Grid Environment

DataCutter [1, 2] is a middleware framework for developing data intensive applications
in a distributed environment. It targets distributed, heterogeneous environments by al-
lowing decomposition of application-specific data processing operations into a set of
interacting processes. In DataCutter, data intensive applications are represented as a set
of filters. A filter is a user-defined object with methods to carry out application-specific
processing on data. The interface for filters consists of an initialization function (init),
a processing function (process), and a finalization function (finalize). Data exchange
between any two filters is described via streams. A stream is a communication ab-
straction that allows fixed size untyped data buffer to be transported from one filter to
The process of manually restructuring an application using the filter-stream programming model is referred to as decomposing the application. When the application is decomposed into a set of filters, the runtime mapping of filters onto various hosts in a grid environment is called placement. DataCutter provides means for specifying filters, streams and placement. In DataCutter, filters are location-independent, because stream names are used to specify filter to filter connectivity rather than endpoint location on a specific host. This allows the placement of filters on hosts controlled by the runtime system so as to minimize processing, network and data copying overheads.

We believe that DataCutter is well suited for a typical distributed data mining algorithm. The two primary steps for a typical DDM algorithm are:

- Computation on data available at each site, including reading the data instances, processing each data instance, performing a local reduction and generating partial results. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed.

- Combining the results obtained from local processing at each data source. This phase is referred to as global reduction.

In a grid environment, data is available on geographically distributed sites. Further, the processing needs to be performed in a heterogeneous environment. The ability to decompose the computation into interacting but location independent filters is important for taking full advantage of grid resources. Thus, we conclude that DataCutter’s filter-stream based programming model is especially suitable for distributed data mining algorithms.

### 3 A Case Study

To validate our belief that distributed data mining implementations can be conveniently developed using DataCutter, we have implemented a distributed version of k-nearest neighbor classifier.

The problem is quite simple: given a 3-dimension range \( R = \{x_1, y_1, z_1, x_2, y_2, z_2\} \), and a point \( w = (a, b, c) \), we want to find the nearest \( k \) neighbors of \( w \) within range \( R \), where \( k \) is a parameter given to the algorithm.

The basic idea for finding k-nearest neighbors in a sequential environment is as follows. For each point \( X = (x, y, z) \) in the file, if \( X \) is in the range \( R \), we compute the distance from \( X \) to \( w \) by \( dist = \sqrt{(x-a)^2 + (y-b)^2 + (z-c)^2} \), and find the points with the k-lowest values of distance.

In the filter-stream approach, the problem is decomposed into two filters for local reduction: one for getting all points in the range \( R \), referred to as \( \text{range} \_\text{query} \), and the other for selecting the k-nearest neighbors, called \( \text{select} \). Since data files are distributed at different sites, one more filter is needed for performing global reduction, referred to as \( \text{combine} \). Communication between filters are implemented via streams.
Filter range_query outputs all the points of the local data file which lies in R to filter select via stream R-S, while filter select communicate with combine through S-C stream by sending the local nearest k neighbors to the global reduction filter combine. Finally, combine will decide the globally nearest k neighbors. The following figure shows the filters and their communication streams.

![Image of filters and streams]

Several experiments were performed based on different filter placement and filter configuration.

### 3.1 Filter Placement

For evaluating the effect of different placement policies, the following two versions of the algorithm are tested.

1. **Place filter range_query and select on different sites:** As described above, data files are distributed on two sites, say maia and bombur. Each site has about 250,000 3-dimension points, and filter range_query is placed on each of them. While filter select is placed on another set of sites, say oin and thorin. The filter combine for global reduction is placed separately on taygeta. Figure 2 shows this configuration.

![Image of version 1 placement]

2. **Place filter range_query and select on the same site:** In this placement scheme, filter range_query and select are placed on the same site as shown in Figure 3.

### 3.2 Filter Configuration

In addition, we also found that the granularity of filters can influence the performance of the algorithm, which is investigated by the following two configurations.
1. Combine filter \textit{range query} and select: Here, the first two filters are combined as one and placed on the site where data files are stored (Figure 4).

2. Combine filter select and combine: In this version, filter select and combine are programmed as one filter and put on the remote site from the one where data is stored (Figure 5).

3.3 Results

The running time is given in Table 1 for the above four versions of k-nearest neighbors algorithm where \(k=8\) and \(k=20\). It’s easy to understand why version-3 performs the best and version-2 is better than the remaining two versions. All the data falling in the range \(R\) need to be transmitted from filter \textit{range query} to select via a stream. While those two filters are put together on the same site, communication cost over the network are sharply reduced, especially for larger \(k\). Furthermore, if their functions are to be accomplished by only one filter, the cost for copying from one buffer to another between filters is also saved, which gives the best performance over the four versions.

Overall, our experience in implementing k-nearest neighbor search using DataCutter has shown that filter granularity and filter placement are critical for achieving
4 Language and Compiler Framework

This section reports on our preliminary ideas towards developing a high-level language support for distributed data mining implementations. We will use compiler technology for automatically decomposing the program into a set of filters.

4.1 Data Parallel Java

We propose to use a data parallel dialect of Java that we had previously used for the same class of applications on a cluster environment [7]. In this dialect of Java, a programmer can specify a computation assuming that all data is available in a flat memory and all computations are done on a single processor. Our previous compiler effort has shown that our chosen extensions give the compiler critical information about the properties and structure of the computation in our target class of applications.

We now describe the extensions and limitations on Java that our dialect uses. We borrow two concepts from object-oriented parallel systems like Titanium [18], HPC++ [3] and Concurrent Aggregates [4].

- **Domains** and **Rectdomains** are collections of objects of the same type. **Rectdomains** have a stricter definition, in the sense that each object belonging to such a collection has a *coordinate* associated with it that belongs to a pre-specified rectilinear section of the domain.

<table>
<thead>
<tr>
<th></th>
<th>version-1</th>
<th>version-2</th>
<th>version-3</th>
<th>version-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K = 8$</td>
<td>83366</td>
<td>70655</td>
<td>58796</td>
<td>87987</td>
</tr>
<tr>
<td>$K = 20$</td>
<td>1000548</td>
<td>874385</td>
<td>122045</td>
<td>1137217</td>
</tr>
</tbody>
</table>

Table 1: Running time for 4 versions of k-nearest neighbors algorithm, where $k=8$ and $k=20$. 

high-performance.
Interface Reducinterface {
  /* Any object of any class implementing */
  /* this interface is a reduction variable */
}
public class KmPoint {
  double x1, x2, x3;
}
public class Kcenter implements Reducface {
  static double[] x1, x2, x3;
  static double[] meanx1, meanx2, meanx3;
  static long[] count;
  static double MES;
  void Finalize() {
    for (i=0; i<5; 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;
    MES+=dis;
  }
}

Figure 6: Parallel k-means clustering in Data Parallel Java

- The foreach loop, which iterates over objects in a domain or rectdomain, and has the property that the order of iterations does not influence the result of the associated computations. We further extend the semantics of foreach to include the possibility of updates to reduction variables, as we explain later.

We have introduced a Java interface called Reducinterface. Any object of any class implementing this interface acts as a reduction variable [10]. The semantics of a reduction variable are analogous to those used in version 2.0 of High Performance Fortran (HPF-2) [10] and in HPC++ [3]. 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. The goals of these extensions is to give the compiler information about independent collections of objects (with no aliasing between the elements), parallel loops and reduction operations.

The outline of the data parallel Java code for k-means clustering is shown in Figure 6. There are several advantages associated with specifying the analysis and processing over datasets in this fashion. The above model assumes that all data is available at a single site. It also assumes that the data is available in arrays of object references, and is not in persistent storage. It is the responsibility of the compiler to locate individual
elements of the arrays from disks.

4.2 Proposed Compilation System

In our on-going research on translating the language interface we described to filter-stream based programs that can be executed on DataCutter, the main challenges that we are addressing are:

- The program must be able to adapt to the distribution of input datasets, including the case when all data required for processing is available at a single site, and the cases when data may be distributed across a number of sites.

- Only very limited computing cycles and memory may be available at a shared data repository. The generated program must include very low-cost filters that can operate on the site(s) hosting the data. These filters should still try to minimize the volume of data that should be transferred from such sites to other computing sites.

- Filter granularity can have a significant impact on performance. It is important to decompose the program into sufficient filters to exploit the variety of computing resources available. At the same time, very fine-grained filters result in unnecessary data transfers, which can severely degrade performance.

- The filters and the runtime system managing the filters must have the ability to adapt to runtime changes in available computing, storage, and network resources.

- The code generation phase must correctly insert interactions between filters and streams.

We present a formulation of the filter placement problem, and give an overview of our approach. Consider any data parallel loop. As we discussed while describing our dialect of Java, the only loop-carried dependencies possible are on data members of an object implementing a reduction interface. Further, these dependencies can only arise in self-updates using associative and commutative operations. Therefore, a data parallel loop can be executed independently on sites having portions of datasets, and a global combination function can be applied later to obtain the final results.

We represent the data parallel loop using the Static Single Assignment (SSA) form [5]. This representation has the property that a variable cannot be assigned values in multiple statements. This removes output and anti-dependencies from the body of the loop.

Each assignment or conditional statement in the resulting code, enclosed in the loop body, represents a potential filter. Our filter enumeration phase ends by listing all potential filters. Each such filter is considered an atomic filter.

Suppose two statements in the loop, \( s1 \) and \( s2 \), are placed in different filters, denoted by \( f1 \) and \( f2 \), respectively. Let the statement \( s2 \) be data dependent upon the statement \( s1 \). The filter \( f2 \) does not need to wait for all iterations of \( f1 \) to be finished. Instead, \( f1 \) forwards the data to \( f2 \) after processing each chunk of data. This allows these two filters to simultaneously perform computations at different computing sites. The compiler can choose the unit-of-work to keep a balance between overhead due
to data transfer latencies and opportunity for parallelism. If the loop iterates over a
large dataset, the unit-of-work can be easily chosen to be a small fraction of the total
iteration space of the loop without incurring any significant transfer latency overhead.
Thus, dividing $s_1$ and $s_2$ in separate filters can potentially allow the program to use
additional computing sites. The obvious tradeoff is the cost of data transfer between
the two sites. Note that if each computing site is a multiprocessor, transparent copies
of each filter can be used to exploit parallelism within the loop.

After the filter enumeration phase, we can represent the program as a graph. The
nodes represent atomic filters. The edges represent dependence between the filters.
A filter $f_1$ can be dependent upon the filter $f_2$ in two ways: 1) $f_2$ needs to finish
executing all iterations before $f_1$ can be initiated, or 2) $f_2$ only needs to finish a unit-
of-work before $f_1$ can be initiated. Filters arising from statements that are in different
loops are more likely to, but do not necessarily, have dependence of the first type.
Filters arising from statements that are in the same data-parallel loop have dependence
of the second type.

We associate a cost with each node and edge. A node simply represents the pro-
cessing cost of the filter. An edge represents the data transfer cost. Our goal is to
compose many atomic filters into a single filter, so as to minimize the overall execution
time on a given set of computing resources. Two filters $f_1$ and $f_2$ can be combined
into a single filter if one of them is directly dependent upon other. In this case, any
filter dependent upon either of the two filters becomes dependent on the combined fil-
ter, and the combined filter becomes dependent upon any filter that either of them was
dependent upon. If the dependence between $f_1$ and $f_2$ was of the first type, the total
execution cost of the combined filter is simply the sum of the execution costs of in-
dividual filters. However, if the dependence between the two was of the second type,
the combined execution cost can be either the maximum, or the sum, of the individual
execution costs, depending upon the available computing resources. In either case, the
edge between the two is deleted and there is no longer any cost of data transfer between
the two filters.

Optimally, the process of composing atomic filters into combined filters to mini-
mize the execution time will involve the following: 1) enumerate all possible mappings
of atomic filters into combined filters, 2) for each such mapping, assign the filters to
available computing resources, and find the execution time, and 3) find the composing
with the least execution time.

There is no obvious polynomial time algorithm for this task. Moreover, this prob-
lem is very similar to the thread partitioning problem for multithreaded architectures,
which has been proven to be NP-complete [17]. Therefore, it is clear that the only
feasible approaches for filter granularity selection will be based on heuristics. Again,
a number of heuristics used in thread partitioning seem quite relevant [9, 17]. We are
developing a number of heuristics for filter granularity selection as part of this project,
using the existing work on thread partitioning as our starting point.
5 Summary

This paper has summarized our preliminary work on a new project on distributed data mining. Our project has two novel thrusts. First, we are using a general purpose grid tool for distributed mining implementations. This is in contrast with the existing approaches that rely on specialized systems. The specific tool we are using is DataCutter, which is a grid middleware for data intensive problems. Second, we are using compiler technology to provide a high-level language support on top of this middleware. Our ongoing compiler work aims to translate a data parallel dialect of Java to a filter-stream based program suitable for execution on DataCutter.

References


