Compiler and Runtime Analysis for Efficient Communication in Data Intensive Applications*

Renato Ferreira*  Gagan Agrawal†  Joel Saltz*
*Department of Computer Science
University of Maryland, College Park MD 20742
{renato,saltz}@cs.umd.edu
†Department of Computer and Information Sciences
University of Delaware, Newark DE 19716
agrawal@cis.udel.edu

Abstract
Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We are developing a compiler that processes data intensive applications written in a dialect of Java and compiles them for efficient execution on distributed memory parallel machines.

In this paper, we focus on the problem of generating correct and efficient communication for data intensive applications. We present static analysis techniques for 1) extracting a global reduction function from a data parallel loop, and 2) determining if a subscript function is monotonic. We also present a runtime technique for reducing the volume of communication during the global reduction phase.

We have experimented with two data intensive applications to evaluate the efficacy of our techniques. Our results show that 1) our techniques for extracting global reduction functions and establishing monotonicity of subscript functions can successfully handle these applications, 2) significant reduction in communication volume and execution times is achieved through our runtime analysis technique, 3) runtime communication analysis is critical for achieving speedups on parallel configurations.

1 Introduction
Analysis and processing of very large multidimensional scientific datasets (i.e. where data items are associated with points in a multidimensional attribute space) is an important component of science and engineering. Examples of these datasets include raw and processed sensor data from satellites, output from hydrodynamics and chemical transport simulations, and archives of medical images. These datasets are also very large, for example, in medical imaging, the size of a single digitized composite slide image at high power from a light microscope is over 7GB (uncompressed), and a single large hospital can process thousands of slides per day.

The processing typically carried out over multidimensional datasets in these and related scientific domains share important common characteristics. Access to data items is described by a range query, namely a multidimensional bounding box in the underlying multidimensional space of the dataset. The basic computation consists of 1) mapping the coordinates of the retrieved input items to the corresponding output items, and 2) aggregating, in some way, all the retrieved input items mapped to the same output data items. The computation of a particular output element is a reduction operation.

We have been developing compiler support for allowing high-level, yet efficient, programming of data intensive computations on multidimensional datasets [2, 13, 12]. We use a dialect of Java for expressing this class of computations, which includes data parallel extensions for specifying collection of objects, a parallel for loop, and a reduction interface. Our compiler extensively uses the existing runtime system Active Data Repository (ADR) [5, 6] for optimizing the resource usage during the execution of data intensive applications. ADR integrates storage, retrieval, and processing of multidimensional datasets on a distributed memory parallel machine. The runtime system, our language design, and compilation techniques particularly exploit the commonalities between the data processing applications that we stated earlier. We target a distributed memory parallel configuration, like a cluster of workstations, for execution of data intensive computations.

In compiling any class of applications on a distributed memory parallel configuration, communication generation and optimization is an important challenge. In this paper, we focus on compiler and runtime analysis required for correct, as well as efficient interprocessor communication for the class of data intensive applications we are targeting. As compared to the general communication analysis problem handled by the data parallel compilers [4, 9, 31], the communication problem we handle is less general in certain ways, and harder and more challenging in certain other ways. In the applications we are targeting, the communication that the compiler needs to generate and optimize is restricted to a global reduction between the processors at the end of each data intensive loop. However, the communication problem we handle is different from the previous distributed memory compilation work in two important ways. First, be-
cause each processor processes large disk-resident datasets, the volume of the communication during the global reduction phase can be very large. Second, use of an object oriented data parallel language makes communication analysis harder.

We present three compiler and runtime analysis techniques in this paper. They are:

- A static analysis technique for extracting the global reduction function from the original data parallel loop.
- A static analysis technique for determining if subscripts functions used for accessing input and output collections are monotonic.
- A runtime analysis technique for reducing the volume of communication during the global reduction phase in sparse data intensive applications.

To evaluate our technique, we have developed a prototype compiler based upon the Titanium infrastructure from Berkeley [29]. Our experiences in compiling these applications and our experimental results show the following:

- Though the techniques we have developed for extracting global reduction functions and establishing monotonicity of subscript functions have a number of limitations, they could successfully handle our example applications.
- Significant reduction in communication volume, as well as execution times is achieved through our runtime communication analysis technique. The reduction in communication volume is up to 30% on 8 node version of multi-grid virtual microscope, and up to 15% on 8 node version of satellite data processing application. The reduction in execution time is up to 33% on 8 node version of multi-grid virtual microscope, and up to 21% on 16 node version of satellite.

The rest of the paper is organized as follows. In Section 2, we give an overview of the applications we target, present the language features and high-level abstractions our compiler supports using satellite data processing as an example, and then give an overview of the compilation techniques we use. Static analysis techniques for extracting global reduction functions and determining monotonicity of subscript functions are presented in Section 3. Runtime analysis technique for reducing volume of communication is presented in Section 4. Experimental results are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 Overview

In this section, we give an overview of the class of the data intensive applications we are targeting. We describe the data parallel language dialect we use for this class of applications. Finally, we give an overview of the execution strategy used to handle disk-resident datasets.

2.1 Data Intensive Applications

We now describe some of the scientific domains which involve applications that process large datasets. Then, we describe some of the common characteristics of these applications.

Satellite data processing: Earth scientists study the earth by processing remotely-sensed data continuously acquired from satellite-based sensors, since a significant amount of earth science research is devoted to developing correlations between sensor radiometry and various properties of the surface of the earth [6]. A typical analysis processes satellite data for ten days to a year and generates one or more composite images of the area under study. Generating a composite image requires projection of the globe onto a two dimensional grid; each pixel in the composite image is computed by selecting the “best” sensor value that maps to the associated grid point.

Analysis of Microscopy Data: The Virtual Microscope [11] is an application to support the need to interactively view and process digitized data arising from tissue specimens. The raw data for such a system is captured by digitally scanning collections of full microscope slides under high power. The virtual microscope application emulates the usual behavior of a physical microscope including continuously moving the stage and changing magnification and focus.

Data intensive applications in these and related scientific areas share many common characteristics. The basic computation consists of (1) mapping the coordinates of the retrieved input items to the corresponding output items, and (2) aggregating, in some way, all the retrieved input items mapped to the same output data items. The computation of a particular output element is a reduction operation, i.e. the correctness of the output usually does not depend on the order in which the input data items are aggregated.

2.2 Language Features and Example Application

In this subsection, we present our high-level abstractions for facilitating rapid development of applications that process disk resident datasets. We use the satellite data processing application as an example [7]. We first describe the nature of the datasets captured by the satellites orbiting the earth, then describe the typical processing on these, and finally explain the high-level abstractions and data parallel language constructs we support for performing such processing.

A satellite orbiting the earth collects data as a sequence of blocks. The satellites contain sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band. As the satellite orbits the earth, the sensors sweep the surface building scan lines of 408 measurements each. Each block consists of 204 half scan lines, i.e., it is a 204 × 204 array with 5 short integers per element. Latitude, longitude, and time is also stored within the block for each measure.

The typical computation on this satellite data is as follows. A portion of earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the earth within the specified area, all available pixels within that time-period are scanned and the best value is determined. An example criteria for finding the best value is cloudiness on that day, with the least cloudy image being the best. The best pixel over each point within the area is used to produce a composite image. This composite image is used by researchers to study a number of properties, like deforestation over time, pollution over different areas, etc [22].

There are two sources of sparsity and irregularity in the dataset and computation. First, the pixels captured by the satellite can be viewed as comprising a sparse three dimen-


```java
public class SatData {
  SatOrigData data;
  void SatData(RectDomain[1] InputDomain) {
    data = new SatOrigData(InputDomain);
  }
  void getdata(Point[3] q) {
    Point[1] time = (q.get1());
    Point[2] p = (q.get1(), q.get2());
    return data(time).getdata(p);
  }
}
public class SatelliteApp {
  Point[1] minpt = ...
  Point[1] maxpt = ...
  RectDomain[1] InputDomain = [minpt : maxpt];
  SatData InputData = new Satdata(InputDomain);
  public static void main(int[] args) {
    Point[2] lowend = (args[2], args[4]);
    Point[2] highend = (args[3], args[5]);
    Point[3] low = (args[0], args[2], args[4]);
    Point[3] high = (args[1], args[3], args[5]);
    Image[2d] OutImage = new OutputData[OutputDomain];
    foreach (Point[3] q in AbsDomain) {
      Point[2] p = (q.get1(), q.get2());
      if (pixel val = satdata.getdata(q))
        OutImage(p).Accumulate(val);
    }
}
```

### Figure 1: A Satellite Data-Processing Code

The class `SatData` is the interface to the input dataset visible to the programmer writing the main execution code. Through its access function `getdata`, this class gives the view that a 3-dimensional grid of pixels is available.

The main processing function takes 6 command line arguments as the input. The first two specify a time range over which the processing is performed. The next four are the latitudes and longitudes for the two end-points of the rectangular output desired. We need to iterate over all the blocks within the time range, examine all pixels which fall into the output region, and then perform the reduction operation (i.e., choosing the best pixel).

We specify this computation in a data parallel language as follows. We consider an abstract 3-dimensional rectangular grid, with time, latitude, and longitude as the three axes. This grid is abstract, because pixels actually exist for only a small fraction of all the points in this grid. However, the high-level code just iterates over this grid in the `foreach` loop. For each point `q` in the grid, which is a `(time, lat, long)` tuple, we examine if the block `SatData[time]` has any pixel. If such a pixel exists, it is used for performing a reduction operation on the object `Output[(lat, long)]`.

We have used a number of data parallel constructs in our languages. These have also been used in object-oriented parallel systems like Titanium [29], HPC++ [3], and Con-
foreach ($r \in \mathcal{R}$) {
  $O_1[S_L(r)] = F_1(O_1[S_L(r)], I_1[S_R_1(r)], \ldots, I_n[S_R_n(r)])$
  
  \[ \cdots \]
  $O_m[S_L(r)] = F_m(O_m[S_L(r)], I_1[S_R_1(r)], \ldots, I_n[S_R_n(r)])$
}

Figure 2: Canonical Form of Loop

Current Aggregates [10, 22], and are not unique to our approach. These constructs are:

- A rect domain 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.
- The foreach loop, which iterates over objects in a rect domain, and has the property that the order of iterations does not influence the result of the associated computations.
- We use a Java interface called Redwritten. Any object of any class implementing this interface acts as a reduction variable [16]. 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.

2.3 Execution Strategy Overview

This subsection gives an overview of the compilation and execution strategy we use.

The main challenge in executing a data intensive loop comes from the fact that the amount of data accessed in the loop exceeds the main memory. While the virtual memory support can be used for correct execution, it leads to very poor performance. Therefore, it is compiler’s responsibility to perform memory management, i.e., determine which portions of output and input collections are in the main memory during a particular stage of the computation.

Based upon our experiences from data intensive applications and developing runtime support for them [7, 6], the basic code execution scheme we use is as follows. The output data structure is divided into tiles, such that each tile fits into the main memory. The input dataset is read disk block at a time. This is because the disks provide the highest bandwidth and incur the lowest overhead while accessing all data from a single disk block. Once an input disk block is brought into main memory, all iterations of the loop which read from this disk block and update an element from the current tile are performed. A tile from the output data structure is never allocated more than once, but a particular disk block may be read to contribute to multiple output tiles.

2.3.1 Loop Preprocessing

To facilitate the execution of loops in this fashion, our compiler first performs an initial preprocessing of the loop. In the process, it may replace an initial loop with a sequence of loops, each of which conforms to a canonical form.

Consider any data intensive parallel loop in the dialect of Java described earlier in this paper. For the purpose of our discussion, collections of objects whose elements are modified in the loop are referred to as left hand side or LHS collections, and the collections whose elements are only read in the loop are considered as right hand side or RHS collections.

The canonical form we support is shown in Figure 2. The domain over which the iterator iterates is denoted by $\mathcal{R}$. Let there be $n$ RHS collection of objects read in this loop, which are denoted by $I_1, \ldots, I_n$. Similarly, let the LHS collections written in the loop be denoted by $O_1, \ldots, O_m$. All RHS collections are accessed using a single subscript function, $S_L$. In the iteration $r$ of this loop, the value of output element $O_i[S_L(r)]$ is updated using the function $F_i$. This is a commutative and associative function and uses one or more of the values $O_i[S_L(r)], I_1[S_R_1(r)], \ldots, I_n[S_R_n(r)]$, and other scalar values in the program. More specifically, the function has the form:

$O_i[S_L(r)] = O_i[S_L(r)] \circ g_1(I_1[S_R_1(r)]) \circ g_2(I_2[S_R_2(r)]) \cdots \cdots \circ g_n(I_n[S_R_n(r)])$

where, $\circ$ is a associative and commutative operator, and the function $g_i$ only uses $I_i[S_R_i(r)]$ and scalar values in the program.

The two main restrictions we impose in this definition of a canonical loop are:

- All the LHS collections are accessed using the same subscript function. This restriction allows us to tile all LHS collections in an identical fashion.

- The updates to the LHS elements are performed using associative and commutative functions only. With this restriction, the elements from different input collections that contribute to the value of an output element need not be brought into the memory at the same time.

Starting from a general data parallel loop, loop fission can be performed to obtain a series of loops, such that each of them conforms to this canonical form [13]. Applying such loop fission may involve introducing temporary collections. However, for the applications we experimented with, there were only one input and one output collection.

2.3.2 Loop Planning

In executing the data intensive loops we are targeting, a number of decisions need to be made before execution of the iterations of the loop. These decisions are made during the loop planning phase and are described here. For many of these decisions, we have chosen a simple strategy, and more sophisticated treatment is a topic for future research.

One of the issues in executing any loop in parallel is work or iteration partitioning, i.e., deciding which iterations are performed on each processor. The work distribution strategy we use is that each iteration is performed on the owner of the element read in that iteration. As a result, no communication is required for the RHS elements. The motivation behind this is that the input collections are usually much larger than the output collections for the class of applications.

The next important issue is the choice of tiling strategy for LHS collections. We have so far used a very simple strategy. We query the run-time system to determine the available memory that can be allocated on a given processor. Then, we divide the LHS space into blocks that will fit in that memory. Formally, we divide the LHS range into a set of smaller ranges (called strips) $\{S_1, S_2, \ldots, S_t\}$. Since each of the LHS collection of objects in the loop is accessed
For each LHS strip $S_l$:
- Execute on each Processor $P_j$;
- Allocate and initialize strip $S_l$ for $O_1, \ldots, O_m$;
- For each RHS collection $I_i$:
  - For each disk block in $I_{ij}$:
    - For each element $e$ in the block
      - $T = \text{Iers}(e)$
    - For each $i \in T$
      - if $i \in \mathcal{R}$ \& $(S_l(i) \in S_l)$
      - Update values of $O_1[S_l(r)], \ldots, O_m[S_l(r)]$
- Perform global reduction to finalize the values for $S_l$

Figure 3: Basic Loop Execution Strategy

through the same subscript function, the same strip mining is used for each of them.

After the tiling decision, the next important compilation problem is determining the set of disk blocks that need to be read for performing the updates on a given tile. A LHS tile is allocated only once. If elements of a particular disk block are required for updating multiple tiles, this disk block is read more than once. The compiler uses static declarations in program to extract an expression that is applied to the meta-data associated with each disk block. For the purpose of describing our execution strategy, we assume that for the RHS collection $I_i$ on the given processor $j$, and for the LHS strip $l$, the set of disk blocks that need to be read is denoted by $L_{ij}$.

2.3.3 Loop Execution Strategy

Our basic loop execution strategy is shown in Figure 3. In a separate paper [14], we also describe variations of this strategy to match characteristics of certain applications.

The LHS tiles are allocated one at a time. For each LHS tile $S_l$, and for each RHS collection $I_i$, the RHS disk blocks from the set $L_{ij}$ are read successively. For the disk block read in memory, we need to determine the set of iterations in the loop that use each element in this block. For an element $e$, the set of iterations is denoted by $\text{Iers}(e)$. For each iteration $i$ belonging to the set $\text{Iers}(e)$, we further check if it belongs to the loop range $\mathcal{R}$, and if the element of the LHS collection updated in that loop iteration, $S_l(i)$, belongs to the tile currently being processed ($S_l$). After checking for these conditions, we update the elements $O_1[S_l(r)], \ldots, O_m[S_l(r)]$ using the element $e$.

All processors participate in a global reduction and communication phase after processing each tile. Details of compiler and runtime analysis required for correct and efficient communication during this phase are presented in the next two sections.

3 Compiler Analysis Techniques

In this section, we present two static analysis techniques that enable correct and efficient interprocessor communication for the class of data intensive applications we are targeting. These two techniques are for:

- Extracting a global reduction function from the original data parallel loop. Such a global reduction function is required for correct processing of the loop on a distributed memory machine.
- Determining if subscripts functions used for accessing input and output collections are monotonic. This information is exploited by a runtime technique described in Section 4 to reduce the volume of communication.

3.1 Extracting Global Reduction Function

We now describe a static analysis technique we have developed for extracting the global reduction function from the local reduction function or the data intensive loop itself.

Problem Statement: As we had shown in Figure 2, a given output collection $O_i$ is updated in the loop as follows:

$$O_i[S_L(r)] = \mathcal{F}_i(O_i[S_L(r)], I_1[S_{R1}(r)], \ldots, I_n[S_{Rn}(r)])$$

We want to synthesize a function $f_i$, which can be used in the following form

$$O_i[S_L(r)] = f_i(O_i[S_L(r)], O'_i[S_L(r)])$$

to perform global reduction, using the collections $O_i$ and $O'_i$ computed on individual processors after the local reduction phase. The function $f_i$ is also such that the original computation can be stated as

$$O_i[S_L(r)] = f_i(O_i[S_L(r)], \mathcal{F}_i(I_1[S_{R1}(r)], \ldots, I_n[S_{Rn}(r)])$$

This problem is significantly different from the previous work on analysis of reduction statements [15, 21, 30], because we are considering an object-oriented language where several members of an output object may be updated within the same function.

Solution Approach: Our approach is based upon classifying data dependencies and control dependencies of updates to the data members of the LHS objects.

Consider any statement in the local reduction function that updates a data member of the LHS object $O_i[S_L(r)]$. If this statement includes any temporary variables that are defined in the local reduction function itself, we perform forward substitution and replace the temporary variables. After such forward substitution(s), any update to a data member can be classified as being one of the following types:

1. Assignment to a loop constant expression, i.e., an expression whose value is constant within each invocation of the data intensive loop from which the local reduction function is extracted.
2. Assignment to the value of another data member of the LHS object, or an expression involving one or more other data members and loop constants.
3. Update using a commutative and associative function $op$, such that the data member $O_i[S_L(r)] \cdot x$ is updated as $O_i[S_L(r)] \cdot x = O_i[S_L(r)] \cdot x \ op g(\ldots)$
   where the function $g$ does not involve any members of the LHS object $O_i[S_L(r)]$.
4. Update which cannot be classified in any of the previous three groups.

Our compiler can only compile data intensive loops in which every update to a data member of the LHS object in the local reduction function can be classified in the first, second, or third group above. This restriction did not create
any problems for the applications we have looked at so far. The set of statements in the local reduction function that update the data members of the LHS object is denoted by $S$. In general, the statements in the set $S$ can be control dependent upon predicates in the function. We can currently only handle local reduction functions in which statements in the set $S$ are control dependent upon loop constant expressions only. Again, this restriction did not create any problems for the set of applications we examined.

**Code Generation:** In synthesizing the function $f$, we start with the statements in the set $S$. The statements that fall in groups 1 or 2 above are left unchanged. The statements that fall in the group 3 are replaced by the statement of the form

$$O_i[S_i(r)] \cdot x = O_i[S_i(r)] \cdot O_i[S_i'(r)] \cdot x$$

Our code generation is based upon the notion of program slicing technique [28]. The basic definition of a program slice is as follows. Given a slicing criterion $(s, x)$, where $s$ is a program point in the program and $x$ is a variable, the program slice is a subset of statements in the program or function such that these statements, when executed on any input, will produce the same value of the variable $x$ at the program point $s$ as the original program.

Within the original local reduction function, we use the statements in the set $S$ as the slicing criteria, and use program slicing to construct an executable function that will produce the same results (except as modified for the statements in the group 3) for these statements. The use of slicing for code generation naturally handles the possibility that a statement in the set $S$ may be control dependent upon a loop constant expression.

A simple example of the application of our technique is shown in Figure 4. This example is based upon the satellite application we described in Section 2.2. The last statement in the local reduction function is the only statement that updates the value of a data member of the reduction object. This statement is of the type 3) as per our analysis. After replacing the statement to use the same data member of the object old computed on another processor, we construct a program slice. This slice does not include any other statement in the function, so our resulting global reduction function has a single statement.

### 3.2 Monotonicity Analysis on Subscript Functions

In this subsection, we present a static analysis technique for determining if a function is monotonic. Specifically, we are interested in determining if the subscript functions used for accessing LHS and RHS collections are monotonic. Our approach is based upon combining flow analysis with integer programming techniques.

The monotonicity property of subscript functions is exploited by the runtime communication analysis we present in Section 4. This runtime analysis finds the set of disjoint rectangles in the output space that are updated by the input disk blocks owned by a particular processor. The monotonicity property of the subscript function $S_L$ and inverse subscript function $S_R^{-1}$ can be used to map the bounding box corresponding to an input block to an output rectangle, by simply applying the subscript functions to the two end corners of the bounding box. The technique we use for inverting a subscript function is described in a separate paper [14]. If these functions are not monotonic, then they need to be applied to all elements in an input block to the find the set of output elements updated by them. This becomes extremely expensive, and can easily defeat the purpose of the runtime analysis we perform.

We initially present our analysis under two assumptions. First, we assume that there are no loops in the subscript (or inverted subscript) function. Second, for simplicity of presentation of our basic ideas, we consider for each loop and input and output collections that have a single dimension. Let us denote the function under consideration by $S$. Because we have assumed that the foreach loop as well as the collections have a single dimension, this function takes one integer as the input, and returns another integer as the output.

Consider a control flow graph (CFG) representing the function $S$. If the function $S$ calls to other functions, we inline such functions, so that the code in the function can be represented by a single CFG. We enumerate the acyclic paths in the CFG and denote them by $p_1, \ldots, p_n$. We focus on the code along each acyclic path. By performing forward substitution for each temporary value, we can create an expression relating the output of the function with the input to the function and other values in the program. For an input $i$, the output from the function $S$ when the path $p_j$ is taken is denoted by $S_j(i)$.

For the function $S$ to be monotonic along the path $p_j$, one of the following must hold:

$$\forall i \ (S_j(i+1) \geq S_j(i))$$

or

$$\forall i \ (S_j(i+1) \leq S_j(i))$$

Suppose the function $S$ is invoked with a particular parameter. The particular acyclic path taken can depend upon the value of the parameter. Therefore, for establishing monotonicity of the function, we need one of the following to hold:

$$\forall j \forall i \ (S_j(i+1) \geq S_j(i))$$

or

$$\forall j \forall i \ (S_j(i+1) \leq S_j(i))$$

Using the expressions for the functions $S_j$ computed by forward substitution, we can check the above conditions using the integer set manipulation ability of omega calculator [19, 25]. While this calculator obviously cannot answer all queries of the above type, it turned out to be sufficient for our set of applications.

We can usually improve the accuracy of the technique in the presence of control flow by establishing that certain conditionals are independent of the input parameter of the function. Consider a conditional predicate $c$. If the predicate $c$ can be shown independent of the input parameter, then invocation of the function with any parameter $i$ will take the same successor of $c$ in the CFG. This allows us to partition the $n$ paths into $k$ disjoint groups, $P_1, \ldots, P_k$. These groups of paths have the property that if the invocation of the function with a parameter results in execution along a path belonging the group $P_j$, then invocation of the function with any parameter will result in execution along one of the paths belonging to the group $P_j$.

With this analysis of the possible paths of execution, the condition for monotonicity can be restated as:

$$\forall k \forall j \forall p \forall p' \in P_k \ (S_j(i+1) \geq S_j(i))$$

or

$$\forall k \forall j \forall p \forall p' \in P_k \ (S_j(i+1) \leq S_j(i))$$
Deal With Loops: We next discuss how we can perform monotonicity analysis on subscript functions that contain loops. In the set of applications we used, the subscript functions did not have any loops, however, our technique can deal with loops in a limited form. We can only process loops that have the following properties: 1) the loop must have a single entry-point and a single exit-point, 2) the loop does not contain any conditionals, 3) the loop is countable, i.e., the number of times the loop iterates must not depend upon any value computed in the loop, 4) any array accessed in the loop must be accessed using affine subscripts, and must be assigned affine values, and 5) any scalar updated in the loop is also updated using affine values.

In such cases, the updates performed in the loop to any array elements or scalars can be summarized using the loop count and other constant values. The loop can then be replaced by a single basic block. After such treatment of loops, the analysis presented previously can be applied.

Multidimensional Spaces: Our data parallel dialect of Java allows foreach loops and collections over multidimensional spaces. Therefore, a subscript function takes a multidimensional point as the input, and outputs a multidimensional point. In such cases, the runtime analysis we present in Section 4 requires the subscript functions to have the following properties:

- The value along each dimension of the output point is either dependent upon exactly one dimension of the input point, or is a loop constant.
- The value along each dimension of the input point influences the value along at most one dimension of the output point.
- For each dimension of the output point where its value is dependent upon a dimension of the output point, the function relating the input value to the output value is monotonic.

The first two properties can be ascertained using simple dependence analysis in the subscript function. For establishing the third property, the analysis presented for single dimensional cases is applied along each dimension.

4 Runtime Communication Analysis

In this section, we present the runtime techniques used to reduce the communication volume during the global reduction phase. As mentioned earlier in Section 2.3, after processing each tile locally, each processor participates in a global reduction phase to aggregate the values computed on each node.

Let the number of nodes in the system be N. A naive approach would be to divide an output tile into N sections. Each node is responsible for collecting and aggregating all the elements for each such section of the tile. Each node also needs to communicate, to each of the other N - 1 nodes, one section of the tile. If the total output size is M, the total communication volume becomes (N - 1) * M. The volume of communication increases linearly with the number of nodes, and can rapidly become a bottleneck.

We propose a different approach, which exploits the fact that each node on the system may not have data for the entire output tile being processed. Conceptually, each tile is still partitioned into N sections, and each node is responsible for collecting and aggregating all the elements for each such section of the tile. However, instead of communicating the entire sections of the tile to its owner node, a node only sends the set of elements in that portion that it has actually updated. In the best case, all processors may update disjoint set of output elements. So, the total communication volume will be (N - 1) * M/N, a factor that increases asymptotically to M.

The application and the dataset determines how much better we can do with our optimized approach, as compared to the naive approach. We have developed the runtime analysis techniques for supporting our optimized communication scheme in the way that their overhead is very low. So, we ensure that performance is not sacrificed when relatively low reduction in communication volume is achieved by our optimized approach.

There are two main challenges in supporting the optimized communication strategy:

- Determining, in an efficient fashion, the set of elements in the section of the tile that have been updated by each node, and
- Communicating to other nodes, in a compact fashion, what set of elements from the section of the tile are being sent in each message.

The first challenge is addressed using the meta-data associated with each disk block, and the monotonicity analysis described in Section 3.2. As we mentioned in Section 2.3.2, the system determines the list of RHS disk blocks that need to be brought into memory for processing each tile. As part of the meta-data, a bounding box of coordinates associated with elements of the disk-block is stored. If monotonicity property of the subscript function $S_L$ and inverse subscript function $S^{-1}_R$ is established, this input bounding box can be mapped into an output rectangle by simply applying the subscript function to the two end corners.

By constructing such rectangles corresponding to each disk block read for processing the given tile, we have a description of the set of elements in the tile that have been updated by each node. However, the complication is that such rectangles overlap with each other. Particularly, while

```
Accumulate(pixel val) {
    int b0 = val.bands[0];
    int b1 = val.bands[1];
    int ndvi = ((b1 - b0) / (b1 + b0) + 1) * 512;
    value = max(value, ndvi);
}

Accumulate(OutputData old) {
    value = max(value, old.value);
}
```

Figure 4: Local Reduction Function (left) and Global Reduction Function (right) for satellite Application.
while (E = Dequeue(Event List) ! = “No more events”) {
  Switch (E.type) {
    Case “Beginning” or “Split”:
      for each R in the work list {
        Switch (Intersection of R.range and E.range) {
          Case “Entirely outside”:
            Do nothing;
          Case “Entirely inside”:
            if E.box ends beyond R.box
              Insert split for E.box after R.box
          Default:
            Output rect for R.range from R.started to E.coord
            Remove R from work list
            Grow E.range to incorporate R.range
            Insert split for longest rect after shortest ends
        }
      }
      Insert E.range into work list if it is not inside range
      of any rectangle in the work list
    case “End”:
      if E.range is in work list
        Output a rect for E.range from R.started to E.coord
  }
}

Figure 5: Sweeping Line Algorithm to Remove Intersections
of Rectangular Regions

communication these elements to other nodes, we need to
describe the set of elements in terms of a set of rectangles
that do not overlap. An algorithm for this purpose is pre-
sented in the next subsection.

4.1 Eliminating Intersections
We present an algorithm that receives as input a collection
of intersecting rectangles. It produces as output another set
of rectangles that comprise the same set of elements, but do
not intersect each other.

The algorithm is based on the notion of a sweeping line.
We have implemented, and present here, a two-dimensional
version of the algorithm, as it turned out to be sufficient
for our set of applications. The algorithm can be extended
to three-dimensional space by sweeping a plane instead of a
line.

The algorithm is presented in Figure 5. A vertical sweep-
ing line (parallel to the y axis) is used in this algorithm.
There are two main data structures used in this algorithm.
The first is the event list. Initially, it stores the beginning
and end x coordinates of each rectangle. When a rectangle
is split, new values may be inserted in the event list.
Each event is explicitly marked as being a beginning event,
end event, or a split event. Because we are using a vertical
sweeping line, when we refer to the range of a rectangle, we
mean the range of its y coordinates.

The second data structure is the work list. At any given
point during the execution of the algorithm, it stores the
rectangles that intersect with the current location of the
sweeping line. Initially, the work list is empty.

The actions taken by the algorithm at each event are de-
scribed in Figure 5. Depending on the event and the current
state of the work list, the system takes an action that may
include inserting another event in the event list, inserting
or removing a rectangle into the work list, or outputting a
rectangle.

In order to illustrate the execution of this algorithm, we
present an example in Figure 6. There are 4 intersecting
rectangles in the picture. At first, after initialization, the
sweeping line is at the leftmost position, as shown in (i).
The first event to be handled is the beginning of rectangle
A. The work list is currently empty and the processing just
inserts the rectangle A in the work list. The next event
on the list, shown in (ii), is the beginning of rectangle B.
When comparing against the other ranges we notice that it
intersects the range for rectangle A. The actions taken are:
1) outputting a rectangle for the fraction of A that is already
traversed, 2) removing the rectangle A from the work list,
3) updating the range for B to include the range of A, 4)
inserting A in the work list after the end of rectangle B, and
5) inserting the rectangle B to the work list.

Next, shown in (iii), rectangle C is located by the sweep-
ing line. It lies entirely within the active range and it ends
before rectangle B. So, no action is taken.

The next event, shown in (iv), is the beginning of D. This
is also within the active range, but it ends after B is over, so
a split event is inserted for D, after the end coordinate of B.
The next event is the end of C. Since the range for C is not
in the work list, no action is taken. Then, the line reaches
the end of the rectangle B, as shown in (v). Since the range
for this rectangle is in the work list, the algorithm outputs
the rectangle. Recall that the range of B was updated earlier
to include the range of A.

The next two event are the splits of A and D, which
were inserted earlier in the execution. For both these events,
the system just inserts the ranges in the work list, without
any changes, because they do not intersect any of the active
ranges. The end of each of these rectangles are then reached,
and the algorithm outputs the remaining portions of each as
separate rectangles. The end of rectangle A event is shown
in (vi).

This algorithm does not guarantee optimality, in terms of
returning a minimal number of rectangles. However, as
we will show in Section 5, the runtime overhead of this al-
gorithm is extremely low.

4.2 Loop Execution
As mentioned in Section 2.3.3, all communication takes place
during the global reduction phase. Each node is responsible
for sending the set of non-overlapping rectangles that inter-
sect with each section of the tile to the owner of that section
of the tile. Each node sends only one message to each other
node. This message includes: 1) one integer containing the
total number of rectangles in the message, 2) 4 integers per
rectangle, describing the coordinates of the rectangle, and
3) all data elements in these rectangles. The first two com-
ponents of the message are referred to as the meta-data
associated with the message.

The total overhead depends on the number of dimensions for
the output collection and the number of rectangles being
processed. For a message containing R D-dimensional rect-
angles it is $4 \times R \times 2 \times D + 1$ bytes. Further reducing the
number of rectangles returned by the intersection elimina-
tion algorithm can help in reducing this overhead. However,
as we discuss in Section 5, this overhead was extremely low
for all our test cases.

After such messages are exchanged, each node processes
each message received by traversing the rectangles contained
in it, and using the meta-data attached at the beginning of
the message to update the associated elements. Finally, the nodes update the section of the tile they own using the global reduction function constructed as described in Section 3.1.

5 Experimental Results

We now present experimental results to demonstrate the effectiveness of our techniques in generating correct and efficient communication for our class of applications. Specifically, we present experimental data to evaluate the following:

• The time taken by the intersection elimination algorithm, and the number of rectangles it takes as inputs and the number of rectangles it outputs.

• The size of meta-data that needs to be sent with optimized messages that are comprised of disjoint rectangular sections.

• The reduction in communication volume achieved by our runtime communication optimization.

• The reduction in execution time achieved by our runtime communication optimization.

• The overall performance on a parallel configuration, and the speedups achieved after performing communication optimizations.

5.1 Experiment Design

We used our prototype compiler to generate code for two applications. We run our test programs on a cluster of 400 MHz Pentium II based nodes 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 some of our runs, we also used a 16 node configuration.

The first application we use is based on the Virtual Microscope [11]. We implemented a multi-grid version of this application, which processes images captured at different magnifications and generates one high resolution output image. We refer to this application as mg-vscope. The second application we experimented with is a satellite image processing application similar to the one described in Section 2.2. We refer to it as satellite in this section.

These two applications are substantially different, both in terms of the nature of the datasets they handle and the computations they perform. mg-vscope accesses a dense dataset to retrieve data corresponding to a portion of the slide in order to generate the output image. The regularity in the dataset can be exploited to partition the disk blocks between the nodes in a fashion that each node will have data only for a portion of the entire output region. So, this application can benefit more from the runtime communication optimization we have developed. The second application, satellite, accesses a sparse dataset to retrieve data corresponding to a region of the planet and aggregates it over a period of time. If the aggregation is performed over a large time period, we expect that all nodes will have data points for all elements on the output. This makes the runtime communication optimization less profitable for this application.

We implemented three separate versions of each of these applications. The first one implements the naive approach described in Section 4, which has a high communication overhead. We refer to this version as full. The second version incorporates the runtime communication optimization we have presented and is referred to as opt. Finally, as a base line best case, we implemented a version that performs no communication. This version is referred to as none.

For both these applications, we experimented with two different query sizes, referred to as medium and large queries. By query size, we mean the size of the input dataset the application processes during execution. For the mg-vscope application, the dataset contains an image of 29, 238 × 28, 800 pixels collected at 5 different magnification levels, which corresponds to 3.3 GB of data. The medium query size processes a region of 10, 000 × 10, 000 pixels, which corresponds to reading 627 MB and generating an output of 400 MB. The large query for this application generates an image of 20, 000 × 20, 000. This requires reading nearly 3 GB, and generating an output of nearly 1.6 GB. The entire dataset for the satellite application contains data for the entire earth at a resolution of 1/128° of a degree in latitude and longitude, over a period of time that covers nearly 15, 000 time steps. The size of the dataset is 2.7 GB. The medium query traverses a region of 15, 000 × 10, 000 × 10, 000, which involves reading 446 MB to generate an output of 50 MB. The large query corresponds to a region of 6, 000 × 20, 000 × 40, 000 points. This requires reading nearly 1.7 GB and generating a 400 MB output.

5.2 Results

We first consider our intersection elimination algorithm. The number of rectangles before and after applying the algorithm for each test case are shown in Table 1. The numbers reported for each application and query pair are 1) the total number of rectangles before applying the algorithm (orig).
Table 1: Number of Input Rectangles Before and After Eliminating Intersections

<table>
<thead>
<tr>
<th></th>
<th>mg-vscope</th>
<th></th>
<th>satellite</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>1-node</td>
<td>8-node</td>
<td>orig</td>
</tr>
<tr>
<td>medium</td>
<td>803</td>
<td>27</td>
<td>622</td>
<td>2380</td>
</tr>
<tr>
<td>large</td>
<td>3894</td>
<td>60</td>
<td>2814</td>
<td>9143</td>
</tr>
</tbody>
</table>

Table 2: Total Communication Volume and Total Meta-data Size (shown in parenthesis) for opt Versions

<table>
<thead>
<tr>
<th></th>
<th>mgv/med</th>
<th>mgv/large</th>
<th>mgv/med</th>
<th>mgv/large</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>236.70 MB</td>
<td>(2.05 KB)</td>
<td>1451.69 MB</td>
<td>(7.39 KB)</td>
</tr>
<tr>
<td>4</td>
<td>803.21 MB</td>
<td>(7.13 KB)</td>
<td>3422.83 MB</td>
<td>(27.41 KB)</td>
</tr>
<tr>
<td>8</td>
<td>1280.92 MB</td>
<td>(16.13 KB)</td>
<td>5212.48 MB</td>
<td>(56.85 KB)</td>
</tr>
<tr>
<td></td>
<td>sat/med</td>
<td>sat/large</td>
<td>sat/med</td>
<td>sat/large</td>
</tr>
<tr>
<td>2</td>
<td>477.70 MB</td>
<td>(0.41 KB)</td>
<td>3424.14 MB</td>
<td>(25.82 KB)</td>
</tr>
<tr>
<td>4</td>
<td>1249.99 MB</td>
<td>(15.21 KB)</td>
<td>1074.47 MB</td>
<td>(16.46 KB)</td>
</tr>
<tr>
<td>8</td>
<td>331.31 MB</td>
<td>(96.51 KB)</td>
<td>2281.47 MB</td>
<td>(297.30 KB)</td>
</tr>
</tbody>
</table>

2) the number of rectangles after applying the algorithm for the 1 node case (1-node), and 3) the aggregate number of rectangles on 8 nodes after applying the algorithm (8-node).

Two important observations can be made from this table. First, the total number of rectangles that need to be processed by the algorithm is quite large in all cases, so the efficiency of the algorithm is important in keeping the overhead of analysis low. Second, substantial reduction in the number of rectangles is achieved, which means that there is a significant overlap between the rectangles corresponding to different disk blocks.

Next, we focus on the runtime cost for executing the intersection elimination algorithm. For all our test cases and execution on 1, 2, 4, or 8 nodes, the time taken by this algorithm is less than 0.9% of the total execution time. The highest relative cost for the optimization is for satellite application with medium query. On one node, it takes 0.5% of the total execution time and goes down to 0.35%, 0.14% and 0.05%, on 2, 4 and 8 nodes, respectively. For the mg-vscope application, the highest relative cost comes with the large query and on 1 node, and is 0.02% of the total execution time. For medium query of mg-vscope, the ratio on one node is 0.013%. For the large query of satellite, the ratio on one node is 0.62%. In all cases, the ratio decreases as the number of nodes increases.

We next focus on the overhead of meta-data that needs to be sent with optimized messages that are comprised of disjoint rectangular sections. Table 2 presents the total communication volume (in Mega Bytes), i.e., the sum of the sizes of all messages sent by the application during execution for opt versions. Next to that, in parenthesis, the table shows the sum of the sizes of meta-data sent with all messages. The size of meta-data is less than 0.05% of the total communication volume in all cases. The data presented in Table 2 clearly establishes that the overhead of sending meta-data with optimized messages is negligible.

Table 3 shows the total communication volume for the full versions. These numbers can be compared against the total communication volume shown for opt versions in Table 2 to see the reduction in communication volume achieved by our runtime technique. For mg-vscope, with medium query size, the reduction in communication volume is 9%, 25%, and 52% on 2, 4, and 8 processors, respectively. For mg-vscope, with large query size, the reduction in communication volume is 3%, 25%, and 51%, on 2, 4, and 8 processors, respectively. For satellite, with medium query size, the reduction in communication volume is less than 1% in all configurations. With large query size on the same application, the reduction is 2%, 6%, and 15%, on 2, 4, and 8 nodes, respectively.

Figure 7 shows the running times for mg-vscope application running the medium query. The improvements produced by the optimization are 0.4%, 12% and 33.75% for 2, 4, and 8 nodes, respectively. The speedups for the opt version on 2, 4 and 8 processors are 1.9, 2.47, and 3.33, respectively. Figure 8 presents the execution times for mgvscope executing the large query. opt version performs 1%, 13% and 34% better than the full version, on 2, 4 and 8 nodes, respectively. The speedups on this query are 1.96, 2.79, and 4.05 for 2, 4, and 8 nodes, respectively.

The performance gains on 2, 4, and 8 nodes, and on medium and large queries, are proportional to the reduction in communication volume that we reported earlier. As the number of nodes increases, the amount of overlap between the portions written by different nodes decreases. Thus, higher performance gains are obtained by communicating only the portions that have been written by each node.

We next present execution times for the satellite application. Because we expected the runtime communication analysis to be less effective for this application, we also included a 16 node execution. The results for medium query are shown in Figure 9. The improvements are -1.7%, 2.5%, -9.3% and 10.1%, on 2, 4, 8, and 16 nodes, respectively. The speedups are 1.52, 1.96, 2.08, and 2.45 for opt versions on 2, 4, 8, and 16 nodes, respectively. Results for this application on large query are shown in Figure 10. The performance
different because we are considering a different set of applications, and a different language and runtime support infrastructure. The communication analysis in their project is focused on near-neighbor communication, whereas the class of applications we target require global reduction.

Runtime analysis for communication optimization has been extensively used as part of the inspector/executor framework for parallelizing sparse computations on distributed memory machines [24]. The details of the runtime analysis we have presented in Section 4 are very different because we are considering a different application class.

Analysis and optimization of reduction operations is a well studied topic in the compiler literature [15, 21, 30]. Because of the use of an object-oriented language, where reduction operations are performed on complex objects, the compiler analysis we have used is significantly different than the previous work.

Our work on monotonicity analysis is significantly different from the existing work on similar problems [20, 27], because of handling more complex control flow. Integer set manipulation has also been previously used for many specific optimization and code generation problems in parallel compilation [1, 17, 26].

7 Conclusions

Analysis and processing of very large multidimensional scientific datasets is becoming an increasingly important component of science and engineering research. Compiler technology can help accelerate advances in these areas by enabling rapid development of applications that process large datasets. We have been developing a compiler for a data parallel dialect of Java targeting such applications.

In this paper, we have focused on compiler and runtime techniques for enabling correct, as well as efficient communication for this class of applications. We have presented static analysis techniques for 1) extracting a global reduction function from a data parallel loop, and 2) determining if a subscript function is monotonic. We have also presented a runtime technique for reducing the volume of communication during the global reduction phase.

We have experimented with two data intensive applications to evaluate the efficacy of our techniques. Our results show that 1) our techniques for extracting global reduction functions and establishing monotonicity of subscript functions can successfully handle these applications, 2) sig-

---

Figure 8: Comparing full, opt and node Versions of mgyscope Application with large Query

Figure 9: Comparing full, opt and node versions of satellite application running medium query

---

6 Related Work

The communication analysis and optimization problem we have addressed is harder and more challenging in many ways, and less general in other ways, than the general communication analysis problem handled by the distributed memory compilers [4, 9, 31]. In the applications we are targeting, the communication that the compiler needs to generate and optimize is restricted to a global reduction between the processors at the end of each data intensive loop. However, the communication problem we handle is different from the previous distributed memory compilation work in two important ways. First, because each processor processes large disk-resident datasets, the volume of the communication during the global reduction phase can be very large. Second, the use of an object-oriented data parallel language makes communication analysis harder.

Kandemir, Choudhary, and their co-authors have focused on analysis and optimizations for out-of-core programs, including inter-processor communication [18]. Our work is

---

Figure 10: Comparing full, opt and node Versions of satellite Application with large Query
significant reduction in communication volume and execution times is achieved through our runtime analysis technique, 3) runtime communication analysis is critical for achieving speedups on parallel configurations.

References


[22] NASA Goddard Distributed Active Archive Center (DAAC): A Distributed Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data.

http://www.vpr.nasa.gov/MDAC/MDAC_INDEX.html


