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, 9]. 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) [4, 5] 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,

*This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant CCR-9808522.

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. This problem is different in certain ways from the general communication analysis problem handled by the data parallel compilers [3, 7, 23]. In the applications we are targeting, the communication restricted to a global reduction between the processors. However, because each processor accesses large disk-resident datasets, the volume of the communication can be very large. Also, the use of an object oriented data parallel language makes communication analysis harder.

We present three compiler and runtime analysis techniques in this paper. The first is a static compiler analysis technique for extracting the global reduction function from the original loop. The second is a technique for determining the monotonicity of subscript functions. Finally, we present a runtime analysis technique for reducing the communication volume during the global reduction.

To evaluate our technique, we have developed a prototype compiler based upon the Titanium infrastructure from Berkeley [21]. Our experiences in compiling these applications and our experimental results have shown that our compiler techniques could successfully handle our example applications, despite their limitations. We have also shown that a very substantial reduction in communication volume could be achieved by our runtime analysis technique, resulting in a significant overall reduction in execution times.

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 present more details about the class of applications we are targeting. We also describe briefly our compiler front-end and the execution strategy used to handle the applications.

2.1 Data Intensive Applications

The applications we target arise from several domains of science and engineering.

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 [5]. 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 [8] 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 [6]. 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 rectangular portion of earth is specified through latitudes and longitudes of end points. A time range is also specified. For any point on the earth within the specified area, all available pixels within that time-period are scanned and one output value is computed, which is used to produce a composite image of the planet. This image is used by re-

```
Interface Reducinterface {
   {* Any object of a class implementing *}
   {* this interface is a reduction variable *}
}
public class pixel {
   short bands[5] ;
   short geo[2] ;
}
class block {
   short time ;
   pixel bands[204*204] ;
   pixel getData(Point[2] p) {
      {* Search for the (lat, long) on geo data *}
      {* Return the pixel if it exists *}
      {* else return null *}
   }
}
{* Low-level Data Layout *}
class SatOrigData {
   block[1d] data ;
   void SatOrigData(RectDomain[1] InputDomain) {
      data = new block[InputDomain] ;
   }
   pixel getData(Point[3] q) {
      Point[1] time = (q.get(0));
      Point[2] p = (q.get(1), q.get(2));
      return data[time].getData(p) ;
   }
}
public class OutputData
            implements Reducinterface {
   int value ;
   void Accumulate(pixel input ) {
      {* Aggregate value of input *}
      {* pixel into output *}
   }
}
```

```
{* High-level Data Layout *}
public class SatData {
   SatOrigData data;
   void SatData(RectDomain[1] InputDomain) {
      data = new SatOrigData(InputDomain) ;
   }
   pixel getData(Point[3] q) {
      return data.getData(q) ;
   }
}
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]);
      Rectdomain[2] OutputDomain = [lowend : highend];
      Point[3] low = (args[0], args[2], args[4]);
      Point[3] high = (args[1], args[3], args[5]);
      Rectdomain[3] AbsDomain = [low : high];
      Image[2d] OutImage = new OutputData[OutputDomain];

      foreach (Point[3] q in AbsDomain) {
         Point[2] p = (q.get(1), q.get(2));
         if (pixel val = satdata.getData(q))
            OutImage[p].Accumulate(val);
      }
   }
}
```

Figure 1. A Satellite Data-Processing Code

searchers to study a number of properties, like deforestation over time, pollution over different areas, etc [18].

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 dimensional array, where time, latitude, and longitude are the three dimensions. Pixels for several, but not all, time values are available for any given latitude and longitude. The second source of irregularity in the dataset comes because the earth is spherical, whereas the satellite sees the area of earth it is above as a rectangular grid. Thus, the translation from the rectangular area that the satellite has captured in a given band to latitudes and longitudes is not straight forward.

In Figure 1, we show the essential structure associated with the satellite data processing application [6].

The class `block` represents the data captured in each time-unit by the satellite. This class has one function (`getData`) that takes a (latitude, longitude) pair and sees if there is any pixel in the given block for that location. If so, it returns that pixel. The class `SatOrigData` stores the data as a one dimensional array of blocks.

Classes `block` and `SatOrigData` are not visible to the programmer writing the processing code. The goal is to provide a simplified view of the dataset to the application programmers, thereby easing the development of correct, but not necessarily efficient, data processing application. The compiler translating the code obviously has the access to the source code of these classes, which enables it to generate efficient low-level 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.

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 rely on three types of data parallel constructs.

- 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.
- The *foreach* loop, which iterates over objects in a rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.
- We use a Java interface called *Reducinterface*. Any object of any class implementing this interface acts as a *reduction variable* [12]. 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

Based upon our experiences from data intensive applications and developing runtime support for them [6, 5], 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. We do it to avoid causing page faults during loop execution, which would severely degrade the performance. The input dataset is read one 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.

foreach(r $\in \mathcal{R}$) {
$O_1[\mathcal{S}_L(r)] = \mathcal{F}_1(O_1[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
...
$O_m[\mathcal{S}_L(r)] = \mathcal{F}_m(O_m[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
}

Figure 2. Canonical Form of Loop

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 loop 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 LHS collections are accessed using a single subscript function, $\mathcal{S}_L$. In the iteration r of this loop, the value of output element $O_i[\mathcal{S}_L(r)]$ is updated using the function $\mathcal{F}_i$. More specifically, the function has the form:

$$O_i[\mathcal{S}_L(r)] = O_i[\mathcal{S}_L(r)] \; op \; g_1(I_1[\mathcal{S}_{R1}(r)]) \; op \; g_2(I_2[\mathcal{S}_{R2}(r)]) \; op \; \ldots \; op \; g_n(I_n[\mathcal{S}_{Rn}(r)])$$

where, *op* is a associative and commutative operator, and the function g_i only uses $I_i[\mathcal{S}_{Ri}(r)]$ and scalar values in the program.

2.3.2 Loop Planning

In executing these data intensive loops, a number of decisions need to 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. Our approach is to execute each iteration 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.

```
For each LHS strip S_l:
  Execute on each Processor P_j:
    Allocate and initialize strip S_l for O_1, ..., O_m
    For each RHS collection I_i
      For each disk block in L_ijl
        For each element e in the block
          I = Iters(e)
          For each i ∈ I
            If (i ∈ R) ∧ (S_L(i) ∈ S_l)
              Update values of O_1[S_L(r)], ..., O_m[S_L(r)]
  Perform global reduction to finalize the values for S_l
```

Figure 3. Basic Loop Execution Strategy

Another issue is the choice of tiling strategy for LHS collections. Our approach so far has been to query the runtime system to determine the available memory that can be allocated on a given processor. As mentioned earlier, the LHS is divided in tiles (or *strips*) $\{S_1, S_2, \ldots, S_r\}$ that will each fit in the available memory.

After that, we have to determine 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_{ijl}.

2.3.3 Loop Execution Strategy

Our basic loop execution strategy is shown in Figure 3. In a separate paper [10], we also describe variations of this strategy to match characteristics of certain applications.

Details of the compiler support for operations performed locally on each processor are presented in a separate paper [10]. In this paper, we focus on the last statement in our strategy, which is the global reduction stage.

3 Compiler Analysis Techniques

In this section, we present two static analysis techniques. The first one is for extracting a global reduction function from the original data parallel loop. Such a function is required for correct processing of the loop on a distributed memory machine. The second technique is for determining if the subscript 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 the technique for extracting the global reduction function.

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

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

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

$$O_i[\mathcal{S}_L(r)] = f_i(O_i[\mathcal{S}_L(r)], O_i'[\mathcal{S}_L(r)])$$

to perform global reduction, using the collections O_i and O_i' computed on individual processors after the local reduction phase.

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 $O_i[\mathcal{S}_L(r)]$. 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 $O_i[\mathcal{S}_L(r)].x$ is updated as
$$O_i[\mathcal{S}_L(r)].x = O_i[\mathcal{S}_L(r)].x \; op \; g(\ldots)$$
where the function g does not involve any members of the LHS object $O_i[\mathcal{S}_L(r)]$.
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.

```
(a) 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);
    }

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

Figure 4. Local Reduction Function (a) and Global Reduction Function (b) for satellite Application

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_i, 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

$$O_i[\mathcal{S}_L(r)].x \;=\; O_i[\mathcal{S}_L(r)].x \; op \; O'_i[\mathcal{S}_L(r)].x$$

Our code generation is based upon the notion of program slicing [20]. Within the original local reduction function, we use the statements in the set S as the slicing criteria, and apply the technique to construct a 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

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 control flow analysis with integer programming techniques.

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 foreach loops and input and output collections that have a single dimension.

Let us denote the function under consideration by $\mathcal{S}$. 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 $\mathcal{S}$. If the function $\mathcal{S}$ contains 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 $\mathcal{S}$ when the path p_j is taken is denoted by $\mathcal{S}_j(i)$.

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

$$\forall i \; (\; \mathcal{S}_j(i+1) \geq \mathcal{S}_j(i) \;)$$

or

$$\forall i \; (\; \mathcal{S}_j(i+1) \leq \mathcal{S}_j(i) \;)$$

Suppose the function $\mathcal{S}$ 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:

$$\forall i \;\; (\;\; \forall j \, \forall l \;\; \mathcal{S}_j(i+1) \geq \mathcal{S}_l(i) \;)$$

or

$$\forall i \;\; (\;\; \forall j \, \forall l \;\; \mathcal{S}_j(i+1) \leq \mathcal{S}_l(i) \;)$$

Using the expressions for the functions $\mathcal{S}_j$ computed by forward substitution, we can check the above conditions using the integer set manipulation ability of omega calculator [15]. 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:

$$\forall k \forall i \; (\; \forall j_{p_j \in P_k} \; \forall l_{p_l \in P_k} \; \mathcal{S}_j(i+1) \; \geq \; \mathcal{S}_l(i) \;)$$

or

$$\forall k \forall i \; (\; \forall j_{p_j \in P_k} \; \forall l_{p_l \in P_k} \; \mathcal{S}_j(i+1) \; \leq \; \mathcal{S}_l(i) \;)$$

Dealing 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 input 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

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.

In this section we present an 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 section 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.

There are two main challenges in supporting the optimized communication strategy. The first one is determining efficiently the set of elements in a tile that have been updated on each node. The other is to communicate only those element across the nodes, without incurring high data overhead.

The first challenge is addressed by 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, the bounding box is stored for each disk-block containing the range of the elements within that block. If the monotonicity for the subscript function $\mathcal{S}_L$ and inverse subscript function $\mathcal{S}_R^{-1}$ 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 a list of such rectangles for all blocks available locally for a particular tile, we know the elements that are updated on each node. Using these rectangles we can create blocks of elements that need to be communicated to other nodes. To avoid communicating the same element more than once, we want first to eliminate all intersections on the list of rectangles. An algorithm for this purpose is presented in the next subsection.

```
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

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 sweeping 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 algorithm is presented in Figure 5. It consists of retrieving the next event from the event list, and performing the actions that are triggered by that event. It goes on until there are no more events.

To illustrate the 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 sweeping 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 algorithm 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 intersect 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 components of the message are referred to as the

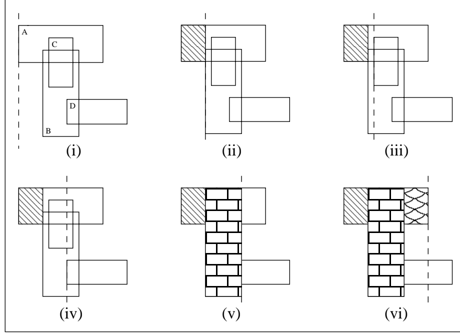

Figure 6. An Example Execution of Intersection Removal Algorithm

meta-data associated with the message.

The total overhead depends on the number of dimensions in the output collection and the number of rectangles being processed. For a message containing R D-dimensional rectangles, it is $4 \times R \times 2 \times D + 1$ bytes. Further reducing the number of rectangles returned by the intersection elimination 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 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 [8]. 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.

	mg-vscope			satellite		
	orig	1-node	8-node	orig	1-node	8-node
medium	803	27	622	2380	9	905
large	3894	66	2814	9433	550	6090

Table 1. Number of Input Rectangles Before and After Eliminating Intersections

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 about 3.3 GB of data. The `medium` query size corresponds to reading 627 MB and generating an output of 400 MB. The `large` query for this application requires reading nearly 3 GB, and generating an output of nearly 1.6 GB. The entire dataset for the `satellite` application contains 2.7 GB. The `medium` query reads 446 MB to generate an output of 50 MB. The `large` query reads nearly 1.7 GB and generating a 400 MB output.

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 3 numbers reported for each application and query pair are 1) the total number of rectangles before applying the algorithm (`orig`), 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

	mgv/med		mgv/large	
2	356.70 MB	(2.05 KB)	1451.69 MB	(7.39 KB)
4	853.21 MB	(7.13 KB)	3422.83 MB	(27.44 KB)
8	1290.92 MB	(16.13 KB)	5212.48 MB	(56.85 KB)
	sat/med		sat/large	
2	47.70 MB	(0.41 KB)	374.24 MB	(25.82 KB)
4	142.99 MB	(15.21 KB)	1074.47 MB	(116.46 KB)
8	331.31 MB	(86.54 KB)	2281.47 MB	(237.30 KB)

Table 2. Communication and Meta-data Sizes for opt Versions

	mgv/med	mgv/large	sat/med	sat/large
2	381.47 MB	1525.88 MB	47.70 MB	381.53 MB
4	1144.41 MB	4577.64 MB	143.11 MB	1144.58 MB
8	2670.29 MB	10681.20 MB	333.92 MB	2670.69 MB

Table 3. Total Communication Volume for full Versions

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. 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 0.03% 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 5%, 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.

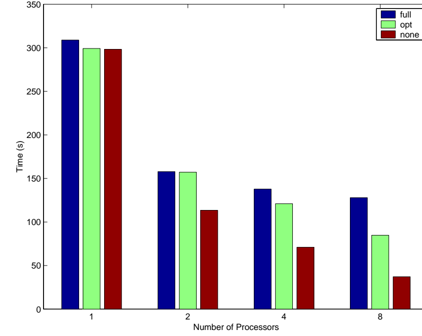

Figure 7. mgvscope Application with medium Query

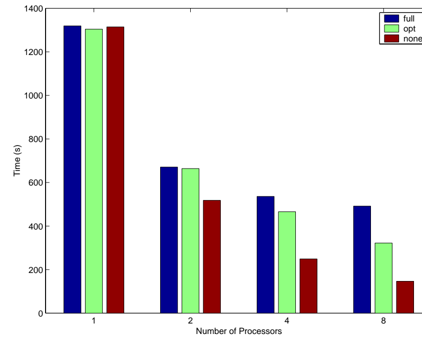

Figure 8. mgvscope Application with large Query

Figure 7 shows the execution times for `mgvscope` 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 3.53, 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 commu-

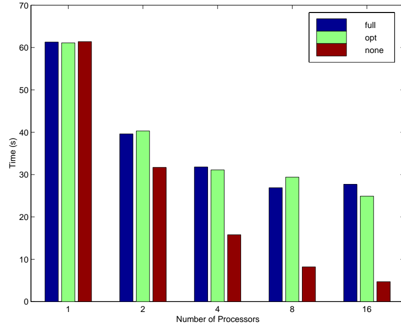

Figure 9. satellite application running medium query

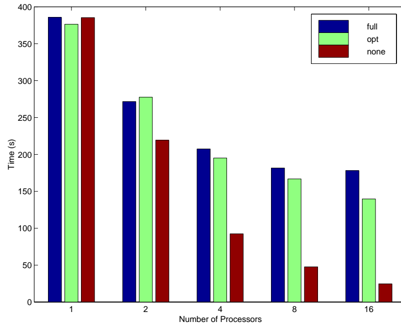

Figure 10. satellite Application with large Query

nicating 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 improvement observed are -2%, 5.9%, 8% and 21.5% for 2, 4, 8, and 16 nodes, respectively. The speedups this time are 1.36, 1.93, 2.26, and 2.69, on 2, 4, 8, and 16 nodes, respectively. Again, the performance gains across nodes and query sizes are proportional to the reduction in communication volume that we reported earlier.

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 [3, 7, 23]. The communication is restricted to the global reduction stage in our target applications. However, because each processor accesses large disk-resident datasets, the volume of the communication can be very large. Also, the use of an object-oriented data parallel language makes communication analysis harder.

Kandemir et al. have focused on analysis and optimizations for out-of-core programs, including inter-processor communication [14]. Besides 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. Our applications, on the other hand, 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 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 [11, 17, 22]. 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 [16, 19], 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, 13].

7 Conclusions

We have been developing a compiler for a data parallel dialect of Java targeting data intensive 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 extracting a global reduction function from a data parallel loop, and for determining the monotonicity of a subscript function. 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) 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

[1] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 186–198. ACM Press, June 1998. ACM SIGPLAN Notices, Vol. 33, No. 5.

[2] Gagan Agrawal, Renato Ferreira, Joel Saltz, and Ruoming Jin. High-level programming methodologies for data intensive computing. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 2000.

[3] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[4] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[5] 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.

[6] 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.

[7] 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)*, pages 149–158, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[8] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., October 1997. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.

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

[10] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiler supported high-level abstractions for sparse disk-resident datasets. Submitted for publication, available at http://www/eecis.udel.edu/ãgrawal/p/ics01_renato.ps, 2001.

[11] H. Han and Chau-Wen Tseng. Improving compiler and runtime support for irregular reductions. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, August 1998.

[12] 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.

[13] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujan, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1251–1297, November 1999.

[14] 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.

[15] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The Omega calculator and library, version 1.1.0. November 1996.

[16] Yuan Lin and David Padua. Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In *In Proceedings of the Conference on Compiler Construction (CC)*, pages 202 – 218, March 2000.

[17] 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.

[18] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. *http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html*.

[19] Madelene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[21] 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.

[22] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.

[23] 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.