Compiling Data Intensive Applications with Spatial Coordinates *

Renato Ferreira* Gagan Agrawal† Ruoming Jin† 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,jrm}@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 which processes data intensive applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines.
In this paper, we focus on data intensive applications with two important properties: 1) data elements have *spatial coordinates* associated with them and the distribution of the data is not regular with respect to these coordinates, and 2) the application processes only a subset of the available data on the basis of spatial coordinates. These applications arise in many domains like satellite data-processing and medical imaging. We present a general compilation and execution strategy for this class of applications which achieves high locality in disk accesses. We then present a technique for hoisting conditionals which further improves efficiency in execution of such compiled codes.
Our preliminary experimental results show that the performance from our proposed execution strategy is nearly two orders of magnitude better than a naive strategy. Further, up to 30% improvement in performance is observed by applying the technique for hoisting conditionals.

1 Introduction

Analysis and processing of very large multi-dimensional scientific datasets (i.e. where data items are associated with points in a multidimensional attribute space) is an important component of science and engineering. An increasing number of applications make use of very large multidimensional datasets. Examples of such datasets include raw and processed sensor data from satellites [23], output from hydrodynamics and chemical transport simulations [19], and archives of medical images [1].

We are developing a compiler which processes data intensive applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines [2, 3, 14]. Our chosen dialect of Java includes data-parallel extensions for specifying collection of objects, a parallel for loop, and reduction variables. We extract a set of functions (including subscript functions used for accessing left-hand-side and right-hand-side object collections, aggregation functions, and range functions) from the given data intensive loop by using the technique of interprocedural program slicing. Data partitioning, retrieval, and

* This work was supported by NSF grant ACR-9982087 awarded jointly to University of Maryland and University of Delaware. Authors Agrawal and Jin were also supported by NSF CAREER award ACI-9733520.

processing is performed by utilizing an existing runtime system called *Active Data Repository* [8–10].

Data intensive computations from a number of domains share two important charactestics. First, the input data elements have *spatial coordinates* associated with them. For example, the pixels in the satellite data processing application have latitude and longitude values associated with them [23]. The pixels in a multi-resolution virtual microscope image have the x and y coordinates of the image associated with them [1]. Moreover, the actual layout of the data is not regular in terms of the spatial coordinates. Second, the application processes only a subset of the available data, on the basis of spatial coordinates. For example, in the satellite data processing application, only the pixels within a bounding box specified by latitudes and longitudes may be processed. In the virtual microscope, again, only the pixels within a rectangular region may be processed.

In this paper, we present a compilation and execution strategy for this class of data intensive applications. In our execution strategy, the right-hand-side data is read one disk block at a time. Once this data is brought into the memory, corresponding iterations are performed. A compiler determined mapping from right-hand-side elements to iteration number and left-hand-side element is used for this purpose. The resulting code has a very high locality in disk accesses, but also has extra computation and evaluation of conditionals in every iteration. We present an analysis framework for code motion of conditionals which further improves efficiency in execution of such codes.

Our preliminary experimental results show that the performance from our proposed execution strategy is nearly two orders of magnitude better than a naive strategy.

The rest of the paper is organized as follows. The dialect of Java we target and an example of data intensive application with spatial coordinates is presented in Section 2. Basic compiler technique and the loop execution model are presented in Section 3. The technique for code motion of conditionals 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 A Data Intensive Application with Spatial Coordinates

2.1 Data-Parallel Constructs

We borrow two concepts from object-oriented parallel systems like Titanium [28], HPC++ [6], and Concurrent Aggregates [12, 25].

- *Domains* and *Rectdomains* are collections of objects of the same type. *Rectdomains* have a stricter definition, in the sense that each object belonging to such a collection has a *coordinate* associated with it that belongs to a pre-specified rectilinear section.
- The *foreach* loop, which iterates over objects in a domain or rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.

We introduce a Java interface called *Reducinterface*. 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.

2.2 Satellite Data Processing Example

In Figure 1, we show the essential structure associated with the satellite data processing application [10, 11]. The satellites generating the datasets contain sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band. As the

```
Interface Reducinterface {
 // Any object of any class implementing
 // this interface is a reduction variable
}
public class pixel {
 short bands[5] ;
 short geo[2] ;
}
public class block {
 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 *}
 }
}
public class SatData {
 public class SatOrigData {
  block[1d] satorigdata ;
  void SatOrigData(RectDomain[1] InputDomain) {
   satorigdata = new block[InputDomain] ;
  }
  pixel getData(Point[3] q) {
   Point[1] time = (q.get(0));
   Point[2] p = (q.get(1), q.get(2));
   return satorigdata[time].getData(p) ;
  }
 }
 void SatData(RectDomain[1] InputDomain) {
  SatOrigData(InputDomain) ;
 }
 pixel getData(Point[3] q) {
  return SatOrigData(q) ;
 }
}
```

```
public class Image
     implements Reducinterface {
 void Accumulate(pixel input ) {
  {* Accumulation function *}
 }
}

public class Satellite {
 Point[1] LoEnd = ...
 Point[1] HiEnd = ...
 SatData satdata ;
 RectDomain[1] InputDomain = [LoEnd : HiEnd];
 satdata.SatData(InputDomain);
 public static void main(int[] args) {
  Point[1] lowtime = (args[0]);
  Point[1] hightime = (args[1]);
  RectDomain[1] TimeDomain = [lowtime : hightime];
  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] Output = new Image[OutputDomain];

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

Fig. 1. A Satellite Data-Processing Code

satellite orbits the earth, the sensors sweep the surface building scan lines of 408 measurements each. Our data file consists of blocks of 204 half scan lines, which means that each block is a 204×204 array with 5 short integers per element. Latitude and longitude is also stored within the disk 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. Typical 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 [23].

The main source of irregularity in this dataset and computation 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.

We next explain the data-parallel Java code representing such computation (Figure 1). The class `block` represents the data captured in each time-unit by the satellite. This class has one function (`getData`) which 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 `SatData` is the interface to the input dataset visible to the programmer writing the main execution loop. Through its access function `getData`, this class gives the view that a 3-dimensional grid of pixels is available. Encapsulated inside this class is the class `SatOrigData`, which stores the data as a 1-dimensional array of `bands`. The constructor and the access function of the class `SatData` invoke the constructor and the access function, respectively of the class `SatOrigData`.

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 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)]`.

The code, as specified above, can lead to very inefficient execution for at least two reasons. First, if the look-up is performed for every point in the abstract grid, it will have a very high over-head. Second, if the order of iterations in the loop is not carefully chosen, the locality can be very poor.

2.3 Common Characteristics

The application we described above has at least two important characteristics which are also found in several other applications, including the multi-resolution virtual microscope [13]. These characteristics are:

- The input data elements have *spatial coordinate* associated with them. For example, the pixels in a multi-resolution virtual microscope image have the x and y coordinates of the image associated with them. Moreover, the actual layout of the data is not regular in terms of the spatial coordinates.

– The application processes only a subset of the available data, on the basis of spatial coordinates. For example, in the multi-resolution virtual microscope, only the pixels within a rectangular area may be processed.

3 Compilation Model for Applications with Spatial Coordinates

This section 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 the experiences from data intensive applications and developing runtime support for them [10, 9], 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 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 the multiple output tiles.

To facilitate the execution of loops in this fashion, our compiler first performs loop fission. For each resulting loop after loop fission, it uses the runtime system called Active Data Repository (ADR) developed at University of Maryland [8, 9] to retrieve data and stage the computation.

3.1 Loop Fission

Consider any loop. 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.

If multiple distinct subscript functions are used to access the right-hand-side (RHS) collections and left-hand-side (LHS) collections and these subscript functions are not known at compile-time, tiling output and managing disk accesses while maintaining high reuse and locality is going to be difficult. Particularly, the current implementation of ADR runtime support requires only one distinct RHS subscript function and only one distinct LHS subscript function. Therefore, we perform *loop fission* to divide the original loop into a set of loops, such that all LHS collections in any new loop are accessed with the same subscript function and all RHS collections are also accessed with the same subscript function.

---

foreach(r $\in \mathcal{R}$) {
$O_1[\mathcal{S}_L(r)] \quad op_1 = \quad \mathcal{A}_1(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)])$
...
$O_m[\mathcal{S}_L(r)] \quad op_m = \quad \mathcal{A}_m(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)])$
}

Fig. 2. A Loop In Canonical Form After Loop Fission

The terminology presented here is illustrated by the example loop in Figure 2. The range (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 $O_1, \ldots, O_m$. Further, we denote the subscript function used for accessing right hand side collections by $\mathcal{S}_R$ and the subscript function used for accessing left hand side collections by $\mathcal{S}_L$.

Given a point r in the range for the loop, elements $\mathcal{S}_L(r)$ of the LHS collections are updated using one or more of the values $I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]$, and other scalar values in the program. We denote by $\mathcal{A}_i$ the function used for updating LHS collection O_i.

Consider any element of a RHS or LHS collection. Its abstract address is referred to as its *l-value* and its actual value is referred to as its *r-value*.

3.2 Extracting Information

Our compiler extracts the following information from a given data-parallel loop after loop fission.

1. We extract the range $\mathcal{R}$ of the loop by examining the domain over which the loop iterates.

2. We extract the accumulation functions used to update the LHS collections in the loop. For extracting the function $\mathcal{A}_i$, we look at the statement in the loop where the LHS collection O_i is modified. We use interprocedural program slicing [26], with this program point and the value of the element modified as the slicing criterion.

3. For a given element of the RHS collection (with its l-value and r-value), we determine the iteration(s) of the loop in which it can be accessed. Consider the example code in Figure 1. Consider a pixel with the l-value $< t, num >$, i.e., it is the num^{th} pixel in the `SatData[t]` block. Suppose its r-value is $< c1, c2, c3, c4, c5, lat, long >$. From the code, it can be determined that this element will be and can only be accessed in the iteration $< t, lat, long >$.

Formally, we denote it as

$$IterVal(e \; = \; (< t, num >; < c1, c2, c3, c4, c5, lat, long >)) \; = < t, lat, long >$$

While this information can be extracted easily from the loop in Figure 1, computing such information from a loop is a hard problem in general and a subject of future research.

4. For a given element of the RHS collection (with its l-value and r-value), we determine the l-value of the LHS element which is updated using its value. For example, for the loop in Figure 1, for a RHS element with l-value $< t, num >$, and r-value $< c1, c2, c3, c4, c5, lat, long >$, the l-value of the LHS element updated using this is $< lat, long >$.

Formally, we denote it as

$$OutVal(e \; = \; (< t, num >; < c1, c2, c3, c4, c5, lat, long >)) \; = < lat, long >$$

This can be extracted by composing the subscript function for the LHS collections with the function $IterVal$.

3.3 Storing Spatial Information

To facilitate decisions about which disk blocks have elements that can contribute to a particular tile, the system stores additional information about each disk block in the *meta-data* associated with the dataset. Consider a disk block b which contains a number of elements. Explicitly storing the spatial coordinates associated with each of the elements as part of the meta-data will require very large additional storage and is clearly not practical. Instead, the range of the spatial coordinates of the elements in a disk block is described by a *bounding box*.

A bounding box for a disk block is the minimal rectilinear section (described by the coordinates of the two extreme end-points) such that the spatial coordinates of each element in the disk block falls within this rectilinear section.

Such bounding boxes can be computed and stored with the meta-data during a preprocessing phase when the data is distributed between the processors and disks.

3.4 Loop Planning

The following decisions need to made during the loop planning phase:

- The size of LHS collection required on each processor and how it is tiled.
- The set of disk blocks from the RHS collection which need to read for each tile on each processor.

The static declarations on the LHS collection can be used to decide the total size of the output required. Not all elements of this LHS space need to be updated on all processors. However, in the absence of any other analysis, we can simply replicate the LHS collections on all processors and perform global reduction after local reductions on all processors have been completed.

The memory requirements of the replicated LHS space is typically higher than the available memory on each processor. Therefore, we need to divide the replicated LHS buffer into chunks that can be allocated on each processor's main memory. We have so far used only a very simple strip mining 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 of that size. Formally, we divide the LHS domain into a set of smaller domains (called *strips*) $\{S_1, S_2, \ldots, S_r\}$. Since each of the LHS collection of objects in the loop is accessed through the same subscript function, same strip mining is used for each of them.

We next decide which disk blocks 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.

On a processor j and for a given RHS collection I_i, the bounding box of spatial coordinates for the disk block b_{ijk} is denoted by $BB(b_{ijk})$. On a given processor j and for a given LHS strip l, the set of disk blocks which need to read is denoted by L_{jl}. These sets are computed as follows:

$$L_{jl} \;=\; \{k \,|\, (BB(b_{ijk}) \cap S_l) \;\neq\; \phi\}$$

3.5 Loop Execution

The generic loop execution strategy is shown in Figure 3. The LHS tiles are allocated one at a time. For each LHS tile S_l, the RHS disk blocks from the set Ljl are read successively. For each element e from a disk block, we need to determine:

1. If this element is accessed in one of the iterations of the loop. If so, we need to know which iteration it is.
2. The LHS element that this RHS element will contribute to, and if this LHS element belongs to the tile currently being processed.

We use the function $IterVal(e)$ computed earlier to map the RHS element to the iteration number and the function $OutVal(e)$ to map a RHS element to the LHS element.

Though the above execution sequence achieves very high locality in disk accesses, it performs considerably higher computation than the original loop. This is because the mapping from the element e to the iteration number and the LHS element needs to be evaluated and intersected

```
For each LHS strip S_l:
    Execute on each Processor P_j:
        Allocate and initialize strip S_l for O_1, ..., O_m
        Foreach k ∈ L_jl
            Read blocks b_ijk, i = 1, ..., n from disks
            Foreach element e in b_ijk
                i = IterVal(e)
                o = OutVal(e)
                If (i ∈ R) ∧ (o ∈ S_l)
                    Evaluate functions A_1, ..., A_m
    Global reduction to finalize the values for S_l
```

Fig. 3. Loop Execution on Each Processor

with the range and tile in every iteration. In the next section, we describe a technique for hoisting conditional statements which eliminates almost all of the additional computation associated with the code shown in Figure 3.

The accumulation function obtained using this strategy for the data-parallel code presented in Figure 1 is shown in Figure 4. The procedure in this figure shows the computation performed after reading each disk block. Three conditional statements are executed in each iteration. The first conditional checks if the iteration corresponding to this input element belongs to the range of the original loop. The second conditional checks if the output element corresponding to the input element belongs to the current tile. The third conditional comes from the slice of the aggregation function $\mathcal{A}$.

4 Code Motion for Conditional Statements

The code execution shown in Figure 4 leads to inefficient execution because of the costs associated with evaluating the conditional statements. In this section, we present a technique which eliminates redundant conditional statements and merges the conditionals with loop headers or other conditionals wherever possible.

Program Representation We consider only structured control flow, with `if` statements and `loops`. Within each control level, the *definitions* and *uses* of variables are linked together with *def-use* links. Since we are looking at *def-use* links within a single control level, each use of a variable is linked with at most one definition.

Candidates for Code Motion The candidates for code motion in our framework are if statements. One common restriction in code hoisting frameworks like Partial Redundancy Elimination (PRE) [20] and the existing work on code hoisting for conditionals [5, 22] is that syntactically different expressions which may have the same value are considered different candidates. We remove this restriction by following *def-use* links and considering multiple *views* of the expressions in conditionals and loops.

To motivate this, consider two conditional statements, one of which is enclosed in another. Let the outer condition be `x > 2` and let the inner condition be `y > 3`. Syntactically, these are different expressions and therefore, it appears that both of them must be evaluated. However, by seeing the definitions of `x` and `y` that reach these conditional statements, we may be able to relate them. Suppose that `x` is defined as `x = z -3` and `y` is defined as `y = z - 2`. By substituting the definitions of `x` and `y` in the expressions, the conditions become `z - 3 > 2` and `z - 2 > 3`, which are identical.

```
void Accumulate(ADR_Box QBox, ADR_Box Tile, block td) {

  ADR_Point opt(2);
  ADR_Region org;
  InputValue val;
  val = td.getFirst();
  while (val != NULL) {

    {* Intersection with the range *}
    if ((QBox.getHigh()[0] >= args[0]) && (val.y >= args[1]) && (val.x >= args[2])&&
        (QBox.getLow()[0] <= args[3]) && (val.y <= args[4]) && (val.x <= args[5])) {

      {* Check if the output point maps to the current tile *}
      if ((val.y - args[0] >= tile[0].getLow()[0]) && (val.x - args[1] >= tile[0].getLow()[1]) &&
          (val.y - args[0] <= tile[0].getHigh()[0]) && (val.x-args[1] <= tile[0].getHigh()[1])) {

        {* Now the accumulation function slice which includes the conditional*}
        if ((QBox.getHigh()[0] >= args[0]) && (val.y >= args[1]) && (val.x >= args[2]) &&
            (QBox.getLow()[0] <= args[3]) && (val.y <= args[4]) && (val.x <= args[5])) {
          opt[0] = val.x - args[1] ;
          opt[1] = val.y - args[0] ;
          Output.Accumulate(val.ndvi);
        }
      }
    }
  val = td.getNext();
  }
}
```

Fig. 4. Processing Code for Each Disk Block for the Satellite Application

We define a view of a candidate for code motion as follows. Starting from the conditional, we do a forward substitution of the definition of zero or more variables occurring in the conditional. This process may be repeated if new variables are introduced in the expression after forward substitution is performed. By performing every distinct subset of the set of the all possible forward substitutions, a distinct view of the candidate is obtained. Since we are considering *def-use* within a single control level, there is at most one reaching definition of a use of a variable. This significantly simplifies the forward substitution process.

Views of a loop header are created in a similar fashion. Forward substitution is not done for any variable, including the induction variable, which may be modified in the loop.

Phase I: Downward Propagation In the first phase, we propagate *dominating constraints* down the levels, and eliminate any conditional which may be redundant. Consider any loop header or conditional statement. The range of the loop or the condition imposes a constraint for values of variables or expression in the control blocks enclosed within. As described previously, we compute the different views of the constraints by performing a different set of forward substitutions. By composing the different views of the loop headers and conditionals statements, we get different views of the *dominating constraints*.

Consider any conditional statement for which the different views of the dominating constraints are available. By comparing the different views of this conditional with the different views of dominating constraints, we determine if this conditional is redundant. A redundant conditional is simple removed and the statements enclosed inside it are merged with the control block in which the conditional statement was initially placed.

Phase II: Upward Propagation After the redundant conditionals have been eliminated, we consider if any of the conditionals can be folded into any of the conditionals or loops enclosing it. The following steps are used in the process. We compute all the views of the conditional which is the candidate for hoisting.

Consider any statement which dominates the conditional. We compute two terms: *anticipability* of the candidate at that statement, and *anticipable views*. The candidate is anticipable at its original location and all views of the candidate computed originally are anticipable views.

The candidate is considered anticipable at the beginning of a statement if it is anticipable at the end of the statement and any assignment made in the statement is not live at the end of the conditional. This reason behind this condition is as follows. A statement can only be folded inside the conditional only if the values computed in it are used inside the conditional only. To compute the set of anticipable views at the beginning of a statement, we consider two cases:

Case 1. If the variable assigned in the statement does not influence the expression inside the conditional, all the views anticipable at the end of the statement are anticipable at the beginning of the statement.

Case 2. Otherwise, let the variable assigned in this statement be v. From the set of views anticipable at the end of the statement, we exclude the views in which the definition of v at this statement is not forward substituted.

Now, consider any conditional or loop which encloses the original candidate for placement, and let this candidate be anticipable at the beginning of the first statement enclosed in the conditional or loop. We compare all the views of this conditional or loop against all anticipable views of the candidate for placement. If either the left-hand-side or the right-hand-side of the expression are identical or separated by a constant, we fold in the candidate into this conditional or loop.

5 Experimental Results

In this section we present results from the experiments we conducted to demonstrate the effectiveness of our execution model. We also present preliminary evidence of the benefits from conditional hoisting optimization. We used a cluster of 400 MHz Pentium II based computers connected by a gigabit switch. Each node has 256 MB of main memory and 18 GB of local disk. We ran experiments using 1, 2, 4 and 8 nodes of the cluster.

The application we use for our experiments closely matches the code presented in Figure 1 and is referred to as `sat` in this section. We generated code for the satellite template the compilation strategy described in this paper. This version is referred to as the `sat-comp` version. We also had access to a version of the code developed by customizing the runtime system Active Data Repository (ADR) by hand. This version is referred to as the `sat-manual` version. We further created two more versions. The version `sat-opt` has the code hoisting optimization applied by hand. The version `sat-naive` is created to measure the performance using a naive compilation strategy. This naive compilation strategy is based upon an execution model we used in our earlier work for the regular codes [14].

The data for the satellite application we used is approximately 2.7 gigabytes. This corresponds to part of the data generated over a period of 2 months, and only contains data for bands 1 and 2, out of the 5 available for the particular satellite. The data spawns the entire surface of the planet over that period of time. The processing performed by the application consists of generating a composite image of the earth approximately from latitude 0 to latitude 78 north and from longitude 0 to longitude 78 east over the entire 2 month period. This involves composing over about 1/8 of the available data and represents an area that covers almost all of Europe, northern Africa, the Middle East and almost half of Asia. The output of the application is a 313×313 picture of the surface for the corresponding region.

Figure 5 compares three versions of `sat`: `sat-comp`, `sat-opt` and `sat-manual`. The difference between the execution times of `sat-comp` and `sat-opt` shows the impact of eliminating redundant conditionals and hoisting others. The improvement in the execution time by performing this optimization is consistently between 29% and 31% on 1, 2, 4, and 8 processor configurations. This is a significant improvement considering that a relatively simple optimization is applied.

The `sat-opt` version is the best performance we expect from the compilation technique we have. Comparing the execution times from this version against a hand generated version show us how close the compiler generated version can be to hand customization. The versions `sat-opt` and `sat-manual` are significantly different in terms of the implementation. The hand customized code has been carefully optimized to avoid all unnecessary computations by only traversing the parts of each disk block that are effectively part of the output. The compiler generated code will traverse all the points of the data blocks. However, our proposed optimization is effective in hoisting the conditionals within the loop to the outside, therefore minimizing that extra computation. Our experiments show that after optimizations, the compiler is consistently around 18 to 20% slower than the hand customized version.

Figure 6 shows the execution time if the execution strategy used for regular data intensive applications is applied for this code (the `sat-naive` version). In this strategy, each input disk block is read like the strategy proposed in this paper. However, rather than iterating over the elements and mapping each element to an iteration of the loop, the bounding box of the disk block is mapped into a portion of the iteration space. Then the code is executed for this iteration space.

As can be seen from Figure 6, the performance of this version is very poor and the execution times are almost two orders of magnitudes higher than the other versions. The reason is that

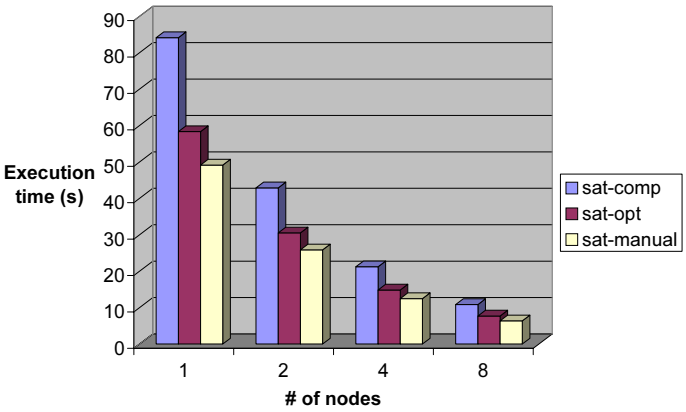

Fig. 5. Satellite: Comparing `sat-comp`, `sat-opt`, and `sat-manual` versions

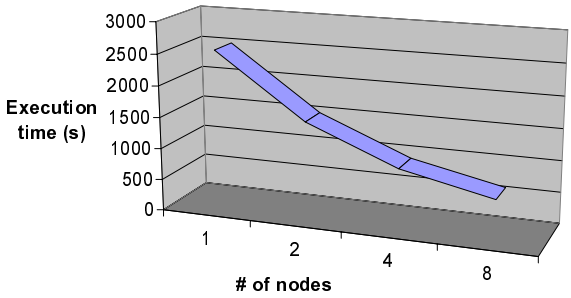

Fig. 6. Performance of the naive implementation of the satellite template

this code iterates over a very large iteration space for each disk block and checks whether or not there is input data for each point in the domain. Due to the nature of the problem, the blocks towards the poles of the planet will spawn a very big area over the globe, which leads to a huge number of iterations. Clearly, the execution strategy for regular codes is not applicable for an application like `sat`.

6 Related Work

The work presented in this paper is part of our continuing work on compiling data intensive applications [14, 3]. Our previous work did not handle applications with sparse accesses and datasets with spatial coordinates. This paper has two main original contributions beyond our previous publications. First, we have presented a new execution strategy for sparse accesses which is significantly different from the execution strategy for dense accesses presented previously [14]. Second, we have presented a technique for code motion of conditionals. The language extensions, loop fission based preprocessing of the original loops, and slicing used as part of our execution strategy are also described in our earlier publications [14].

Our code execution model has several similarities to the *data-centric* locality transformations proposed by Pingali *et al.* [18]. We fetch a data-chunk or shackle from a lower level in the memory hierarchy and perform the iterations from the loop which use elements from this data-chunk. We have focused on applications where no computations may be performed as part of many iterations from the original loop. So, instead of following the same loop pattern and inserting conditionals to see if the data accessed in the iteration belongs to the current data-chunk, we compute a mapping function from elements to iterations and iterate over the data elements. To facilitate this, we simplify the problem by performing loop fission, so that all collections on the right-hand-side are accessed with the same subscript function.

Several other researchers have focused on removing fully or partially redundant conditionals from code. Mueller and Whalley have proposed analysis within a single loop-nest [22] and Bodik, Gupta, and Soffa perform demand-driven interprocedural analysis [5]. Our method is more aggressive in the sense we associate the definitions of variables involved in the conditionals and loop headers. This allows us to consider conditionals that are syntactically different. Our method is also more restricted than these previously proposed approaches in the sense that we do not consider partially redundant conditionals and do not restructure the control flow to eliminate more conditionals. Many other researchers have presented techniques to detect the equality or implies relationship between conditionals, which are powerful enough to take care of syntactic differences between expressions [4, 15, 27].

Our work on providing high-level support for data intensive computing can be considered as developing an out-of-core Java compiler. Compiler optimizations for improving I/O accesses have been considered by several projects. The PASSION project at Northwestern University has considered several different optimizations for improving locality in out-of-core applications [7, 17]. Some of these optimizations have also been implemented as part of the Fortran D compilation system's support for out-of-core applications [24]. Mowry *et al.* have shown how a compiler can generate prefetching hints for improving the performance of a virtual memory system [21]. These projects have concentrated on relatively simple stencil computations written in Fortran. Besides the use of an object-oriented language, our work is significantly different in the class of applications we focus on. Our technique for loop execution is particularly targeted towards reduction operations, whereas previous work has concentrated on stencil computations.

7 Conclusions and Future Work

Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We have developed a compiler which processes data intensive

applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines. In this paper, we focus on data intensive applications with two important properties: 1) data elements have *spatial coordinates* associated with them and the distribution of the data is not regular with respect to these coordinates, and 2) the application processes only a subset of the available data on the basis of spatial coordinates. We have presented a general compilation model for this class of applications which achieves high locality in disk accesses. We have also outlined a technique for hoisting conditionals and removing redundant conditionals that further achieves efficiency in execution of such compiled codes.

Our preliminary experimental results show that the performance from our proposed execution strategy is nearly two orders of magnitude better than a naive strategy. Further, up to 30% improvement in performance is observed by applying the technique for hoisting conditionals.

We need to extend our work in several directions. First, we need general techniques for extracting the function `Iterval`. We will also like to modify our code execution strategy to handle codes where such a function cannot be precisely extracted or multiple iterations can access the same right-hand-side element. Limitations of our code execution model also need to be formalized. Finally, we need to make our implementation more robust and experiment with several other applications.

References

1. Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.
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. Gagan Agrawal, Renato Ferriera, and Joel Saltz. Language extensions and compilation techniques for data intensive computations. In *Proceedings of Workshop on Compilers for Parallel Computing*, January 2000.
4. W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, August 1995.
5. Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 146–158. ACM Press, June 1997.
6. Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.
7. R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.
8. 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.
9. 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.
10. 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.

11. Chialin Chang, Alan Sussman, and Joel Saltz. Scheduling in a high performance remote-sensing data server. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.
12. A.A. Chien and W.J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196. ACM Press, March 1990.
13. 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.
14. Renato Ferriera, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive computations. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.
15. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.
16. 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.
17. M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the performance of out-of-core computations. In *Proceedings of International Conference on Parallel Processing*, August 1997.
18. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.
19. Tahsin M. Kurc, Alan Sussman, and Joel Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.
20. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.
21. Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and plementation (OSDI '96)*, Nov 1996.
22. Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.
23. 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*.
24. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.
25. John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 324–340, October 1994.
26. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
27. Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, 1995.
28. 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.