Advanced Compiler and Runtime Support for 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 play an increasingly important role in many domains of scientific research. However, high-level language and compiler support for developing such applications have been so far lacking.**

We are developing compiler and runtime techniques necessary to generate efficient implementations of such applications from a high-level programming language. Through advanced compiler analysis, our system is able to extract necessary information from the source code. This information is used to generate an executable that makes extensive use of a runtime system called Active Data Repository (ADR).

In this paper, we present advanced techniques that can enhance the performance of compiler generated data intensive codes. We show how characterization of access patterns into *dense* and *sparse* and choosing appropriate execution strategy for each leads to better performance. We also show how *demand-driven* output space allocation on each processor, managed through a *sparse* data-structure, can reduce the number of tiles that need to be allocated on each processor and enhances performance.

Keywords— **data parallel compilers, data intensive applications, tiling, cluster of workstations.**

I. Introduction

More powerful computers make it possible for scientists and engineers to model and simulate physical phenomena in increasingly greater detail. Associated with that, the low cost of storage makes it possible to store more and more of these simulation datasets as well as increasing amounts of data collected from a variety of sensors attached to devices such as satellites or microscopes. Analysis and processing of these very large datasets are becoming very important components in many fields of science and engineering. Examples of such large datasets include collections of raw and processed sensor data from satellites [21], output of long running simulations of time dependent phenomena that periodically generate snapshots of their state [18], and archives of medical images [2].

A large fraction of the applications that make use of such datasets share several important characteristics. First, the dataset is usually multi-dimensional which means that it represents attributes associated with points in some multi-dimensional coordinate system. Both the input and the output are often disk-resident. Applications may use only a subset of all the data available in input and output. Access to data items is described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional space. Only the data items whose associated coordinates fall within the *query box* are retrieved. The processing performed by such applications is also highly stylized. It basically consists of retrieving the input elements for the region of interest, mapping these input points into output points and aggregating the values of input that map to the same output.

One very simple example of such application is the Virtual Microscope [2], [13]. Large quantities of microscope slides are kept digitized at high resolution in some storage device. Users want to browse some slides looking for interesting features on the images. The browsing happens in low resolution and the users switch to higher resolutions as they believe they see an interesting feature. What the system is doing is aggregating the portion of the stored high resolution grid the user is looking at into a low resolution output grid. Naturally, the user for such a system can be another program that is automatically analyzing the datasets. Other typical applications are satellite data processing for earth science applications [1], [11], [24] and simulation systems for water contamination studies [18], airplane wake turbulence [19] or flame sweeping through a volume [23].

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 [4], [3], [14]. For our prototype implementation, we have chosen a dialect of Java with data-parallel extensions based on Titanium [25]. The prototype compiler makes extensive use of the existing runtime system ADR (*Active Data Repository*) [7], [8], [10] for optimizing the resource usage during execution of data intensive applications. ADR integrates storage, retrieval and processing

*This research was supported by NSF grant ACR-9982087 awarded jointly to University of Maryland and University of Delaware. Author Agrawal was also supported by NSF CAREER award ACI-9733520 and NSF grant CCR-9808522.

of multi-dimensional datasets on a distributed memory machine. While a number of applications have been developed using ADR's low-level API and high performance has been demonstrated [9], [24], developing applications in this style requires detailed knowledge of the design of ADR. In comparison, our proposed data-parallel extensions to Java enable programming of data intensive applications at a much higher level. It becomes the responsibility of the compiler to utilize the services of ADR for memory management, data retrieval and scheduling of processes.

We have previously worked on techniques to extract the necessary information from data parallel Java programs [4], [3], [14]. 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.

In this paper, we present advanced techniques that can enhance the performance of compiler generated data intensive codes. Two specific techniques we present are:

Performing Local Reductions: We show how characterization of access patterns into *dense* and *sparse* and choosing appropriate execution strategy for each leads to better performance.

Tiling Output Space: We show how *demand-driven* output space allocation on each processor, managed through a *sparse* data-structure, can reduce the number of tiles that need to be allocated on each processor and leads to better performance.

The rest of this paper is organized as follows. The Section II describes briefly the programming language used. In section III we show how characterization of access patterns into *dense* and *sparse* and choosing appropriate execution strategy for each leads to better performance. In Section IV, we show how *demand-driven* output space allocation on each processor, managed through a *sparse* data-structure, can reduce the number of tiles that need to be allocated on each processor and lead to better performance. We compare our work with related work in Section V and conclude in section VI.

II. Data Parallel Dialect of Java

We now describe our current solution towards providing high-level language support for data intensive computations.

We borrow two concepts from object-oriented parallel systems like Titanium [25], HPC++ [5] and Concurrent Aggregates [12].

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

We introduce a Java interface called *Reducinterface*. Any object of any class implementing this interface acts as a *reduction variable* [15]. The semantics of a reduction variable is analogous to that used in version 2.0 of High Performance Fortran (HPF-2) [15] and in HPC++ [5]. 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.

Figure 1 outlines an example code with our chosen extensions. This code shows the essential computation in a virtual microscope application [13]. A large digital image is stored in disks. This image can be thought of as a two dimensional array or collection of objects. Each element in this collection denotes a pixel in the image. The interactive user supplies two important pieces of information. The first is a bounding box within this two dimensional box, this implies the area within the original image that the user is interested in scanning. The first 4 arguments provided to the main are integers and together, they specify the *points* `lowend` and `hiend`, the two extreme corners of the rectangular bounding box. The second information provided by the user is the subsampling factor, an integer denoted by `subsamp`, which tells the granularity at which the user is interested in viewing the image. A `querybox` is created using specified points `lowend` and `hiend`. Each pixel in the original image which falls within the `querybox` is read and then used to increment the value of the corresponding output pixel.

III. Execution Models for Local Reductions

In this section, we show characterization of access patterns into *dense* and *sparse* and choosing appropriate execution strategy for each leads to better performance.

Initially, we present a general execution strategy, which is applicable to both dense and sparse codes. While it produces very efficient execution for codes with sparse access patterns, it also incurs significant overheads for dense codes. Later, we introduce a modification of this strategy, based upon the monotonicity of the subscript functions, which is very efficient for dense access patterns.

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.

```
Interface Reducinterface {
 // Any object of any class implementing
 // this interface is a reduction variable
}
public class VMPixel {
 char[] colors;
 void Initialize() {
  colors[0] = 0 ;
  colors[1] = 0 ;
  colors[2] = 0 ;
 }
 void Accum(VMPixel Apixel, int avgf) {
  colors[0] += Apixel.colors[0]/avgf ;
  colors[1] += Apixel.colors[1]/avgf ;
  colors[2] += Apixel.colors[2]/avgf ;
 }
}
public class VMPixelOut extends VMPixel
        implements Reducinterface;
public class VMScope {
 static int Xdimen = ... ;
 static int Ydimen = ... ;
 static Point[2] lowpoint = [0,0];
 static Point[2] hipoint = [Xdimen-1,Ydimen-1];
 static RectDomain[2] VMSlide = [lowpoint : hipoint];
 static VMPixel[2d] VScope = new VMPixel[VMSlide];
 public static void main(String[] args) {
  Point[2] lowend = [args[0],args[1]];
  Point[2] hiend = [args[2],args[3 ]];
  int subsamp = args[4];
  RectDomain[2] Outputdomain = [[0,0]:(hiend -
          lowend)/subsamp];
  VMPixelOut[2d] Output = new VMPixelOut[Outputdomain] ;
  RectDomain[2] querybox ;
  Point[2] p;
  foreach(p in Outputdomain) {
   Output[p].Initialize();
  }
  querybox = [lowend : hiend] ;
  foreach(p in querybox) {
   Point[2] q = (p - lowend)/subsamp ;
   Output[q].Accum(VScope[p],subsamp*subsamp) ;
  }
 }
}
```

Fig. 1. Example Code

Based upon the experiences from data intensive applications and developing runtime support for them [10], [8], 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 [7], [8] to retrieve data and stage the computation.

A. 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.

$$
\begin{aligned}
&\text{foreach}(r \in \mathcal{R}) \{ \\
&\quad O_1[\mathcal{S}_L(r)] \quad op_1 = \quad \mathcal{A}_1(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]) \\
&\quad \ldots \\
&\quad O_m[\mathcal{S}_L(r)] \quad op_n = \quad \mathcal{A}_m(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]) \\
&\}
\end{aligned}
$$

Fig. 2. ADR's canonical loop

After loop fission, we focus on an individual loop at a time. We introduce some notation about this loop which is used for presenting our solution. 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 $I_1[S_R(r)], \ldots, I_n[S_R(r)]$, 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*.

B. 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 with this program point and the value of the element modified as the slicing criterion.

3. For a given element e 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. This is denoted by $IterVal(e)$.

4. For a given element e 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. This is denoted by $OutVal(e)$.

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

C. 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 endpoints) 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.

D. Sparse Execution Strategy

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.

```
For each LHS strip T_l:
    Execute on each Processor P_j:
        Allocate and initialize strip T_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 ∈ T_l)
                    Evaluate functions A_1, ..., A_m
        Global reduction to finalize the values for T_l
```

Fig. 3. *Sparse* Loop Execution on Each Processor

- 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.

Different strategies can be used for breaking up the replicated space into tiles which fit in the main memory. These techniques are addressed in Section IV. For the purpose of discussion in this section, we say that LHS domain is divided into a set of smaller domains (called *strips*) $\{T_1, T_2, \ldots, T_r\}$. 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.

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:

$$L_{jl} = \{k \mid (BB(b_{ijk}) \cap T_l) \neq \phi\}$$

The generic loop execution strategy is shown in Figure 3. The LHS tiles are allocated one at a time. For each LHS tile T_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.

E. Dense Execution Strategy

If there is more regularity in the data within a disk block, and if there are updates to the output collection associated

```
For each output tile T_i:
    Execute on each Processor P_j:
        Allocate and initialize T_i for O_1, ..., O_m
        Foreach k ∈ L_ji
            Read blocks b_ijk, i = 1, ..., n from disks
            Let R = OD(b_ijk) ∩ T_i
            Foreach element e of R
                    Evaluate functions A_1, ..., A_m
        Global reduction to finalize the values for T_i
```

Fig. 4. *Dense* Loop Execution on Each Processor

with each iteration of the loop, we can improve upon the execution strategy presented in Figure 3. Instead of computing the iteration of the loop corresponding to each element, we can iterate over the domain of input elements that intersect with the range of the loop.

For each disk block b_{ijk}, we compute the domain $D(b_{ijk})$, which denotes the subset of the domain $\mathcal{S}_R \cdot \mathcal{R}$ which is resident on the disk block b_{ijk}. If the subscript functions are monotonic and invertible, we apply the function $\mathcal{S}_L \cdot \mathcal{S}_R^{-1}$ to each $D(b_{ijk})$ to obtain the corresponding domain in the LHS region. These output domains that each disk block can contribute towards are denoted as $OD(b_{ijk})$. Formally, we compute the sets L_{jl}, for each processor j and each output tile l, such that

$$L_{jl} = \{j \mid \forall\, i\, (OD(b_{ijk}) \cap T_l) \neq \phi\}$$

If the subscript functions are not monotonic and invertible, we can compute the sets L_{jl} by using spatial bounding boxes described earlier.

The loop execution sequence under this scenario is shown in Figure 4.

F. Experimental Results

We conducted experiments to demonstrate that the appropriate choice of execution strategy makes significant difference in the performance of compiler generated code. We run the experiments on a cluster of 400MHz Pentium II computers with 256 MB of main memory and 18 GB of local disk.

We experimented with two applications. The first application is `sat` and abstracts the essential steps in any application processing satellite data [10], [11]. The dataset for this application involves spatial coordinates. Accesses to the dataset are irregular and a large fraction of the iterations do not result in updates to any element of an output collection. The second application used is `vmscope` and closely matches the code shown in Figure 1. This application involves traversing large microscope images and generating an output image for a select region at a user specified magnification. It represents a class of application with regular input data, in which the data contained on the disk block represents all the points within the bounding box for the block. This is a dense code with subscript functions that are monotonic and invertible.

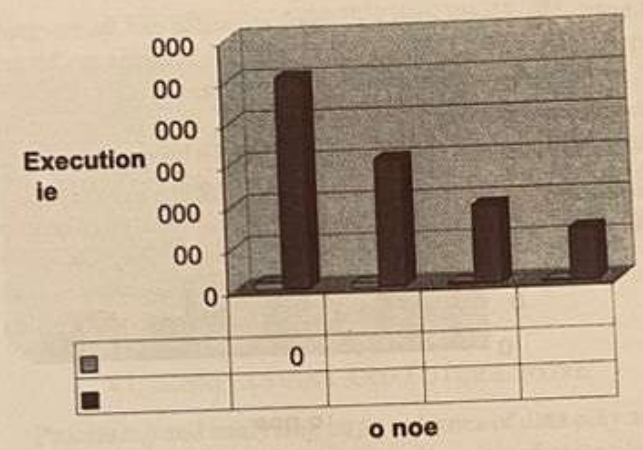

Fig. 5. Comparing `sat-dense` and `sat-sparse`

We create two versions for each of the codes. `vmscope-dense` and `vmscope-sparse` are versions of the virtual microscope with *dense* and *sparse* execution strategies, respectively. `sat-dense` and `sat-sparse` are the versions of the satellite data processing application with *dense* and *sparse* execution strategies, respectively.

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 west 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.

The performance comparison for `sat-sparse` and `sat-dense` is shown in Figure 5. The performance of `sat-dense` version is very poor and is nearly two orders of magnitudes worse than the `sat-sparse` version.

We now describe the results from the `vmscope` application. The dataset for this application is a $19,700 \times 15,360$ pixel image from a microscope. The application is generating an image the corresponds to an area of $10,000 \times 10,000$ of that image. The dense execution strategy fits this application very well, whereas, using the sparse execution strategy incurs extra overhead of conditionals and applying subscript functions multiple times. The `vmscope-dense` version outperforms the `vmscope-sparse` version consistently by a factor of 2.5 on 1, 2, 4, and 8 processors.

Our results on these two applications show that it is crucial to choose the correct execution strategy depending upon the

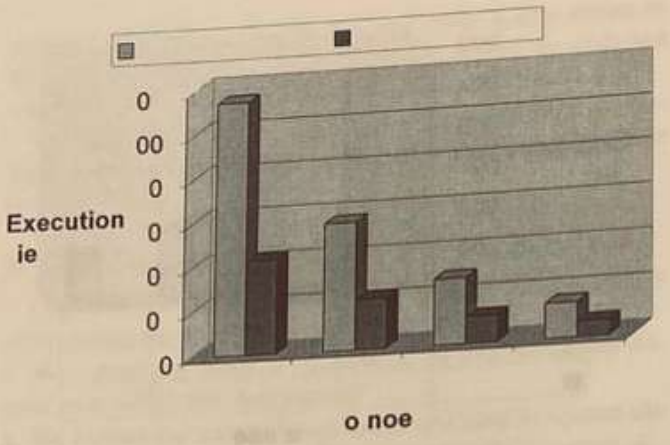

Fig. 6. Comparing vmscope-dense and vmscope-sparse

characteristics of the application.

We believe that a set of heuristics can be used by the compiler to determine appropriate execution strategy. Two main considerations are:

- Are most of the elements in the bounding box of a disk block actually present in the disk block ? This can be estimated by comparing the size represented by the bounding box against the size of data actually resident in the disk block.
- Are there any conditionals inside the body of the loop, which may imply that only a fraction of the iterations result in an actual update to an element of the output collection ?

IV. Advanced Work Partitioning and Memory Management

In the compiler generated code, the output space is replicated on each processor. If this replicated output space does not fit into the main memory of each processor, it needs to divided into tiles. As shown in Figure 3, each processor allocates the same tile and performs the iterations it can perform using the elements from the input blocks it reads. If the same input block contributes to multiple tiles, it needs to be read multiple times.

Consider a simple example. Suppose a program is executed on 8 processors, each with 100 MB of available main memory for output tiles. If the total size of output is 1 GB and the output space is replicated on all processors, then the output space needs to be divided into 10 tiles on each processor. The larger the number of tiles the output space is divided into, the larger is number of disk blocks that may need to be read for multiple tiles. This leads to significant overhead and reduced performance. If the overlap between the output regions that the input disk blocks on different processors update is not large, most of the elements on each tile will not be updated. This also means significant unnecessary overhead for initializing and later communicating the elements.

We propose an alternative strategy. Let the size of local memory available on each processor be M and let the number of processors by p. The main charactestics of our strategy are as follows:

- An *overlap factor* (F) is introduced denoting that elements of the replicated space actually updated by different processors do not usually overlap by more than this factor.
- The size of logical tile allocated on each processor is $(M \times p)/F$.
- The tile is not physically allocated at once. As input blocks are read, the corresponding output rectilinear sections are allocated.

The overlap factor is chosen based upon the charactestics of the application. If the actual overlap between the output space updated on different processors is larger than this factor, virtual memory gets used. This leads to slower but correct execution. Therefore, the choice of overlap factor is crucial for the performance, but not for the correctness.

The main challenge in this scheme is the management of the sparse data-structure created on each processor. When the first input disk block is processed, an initial rectangle is allocated corresponding to the bounding box of the input block. For each subsequent input disk block that is read, the bounding box is compared with the existing allocated rectangles on that processor. The following three possibilities are considered:

- If the bounding box read is strictly enclosed inside an existing rectangle, no new allocation needs to be done.
- If the bounding box read does not overlap with any existing rectangle, space corresponding to this rectangle is allocated and elements are initialized and updated.
- If the bounding box read overlaps with an existing rectangle, the existing rectangle is expanded so that bounding box of the new disk block is now enclosed inside it.

A. Experimental Results

We use the vmscope template described in the previous section to evaluate the performance improvements achieved using the *demand-driven* output space allocation scheme. The dataset and experimental configuration is the same as used for the experiments presented in the previous section.

Figure 7 shows the execution time for the two versions of the virtual microscope application. The original version is referred to as vmscope-orig, and version with demand-driven space allocation is referred to as vmscope-opt. The graph shows the time for running each version on 1, 2, 4, and 8 processor configurations.

The total size of the output created by the dataset we used is 300 MB. This is not a very large dataset for evaluating the effect of tiling on a cluster where each node has 256 MB of main memory. We assumed for our experiments that only 16 MB

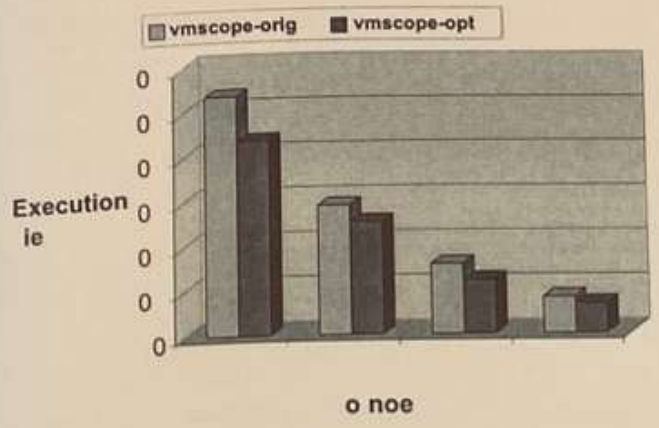

Fig. 7. Performance Improvement Through Demand-Driven Allocation

are available for allocating the output tile and calculated the number of tiles accordingly. We expect to experiment with much larger datasets for the final paper. The overlap factor we assumed was 1.66.

Figure 7 shows significant performance improvements by using the demand-driven space allocation scheme. The percentage improvement in the performance is 19%, 15%, 24%, and 18% on 1, 2, 4, and 8 processors. Clearly, the overhead in managing and allocating a sparse data-structure is much less than the benefits from not initializing entire tiles and fetching a larger number of disk blocks because of the increased number of tiles.

V. Related Work

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 [6], [16]. Some of these optimizations have also been implemented as part of the Fortran D compilation system's support for out-of-core applications [22]. Mowry *et al.* have shown how a compiler can generate prefetching hints for improving the performance of a virtual memory system [20]. 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.

Our code execution models have several similarities to the *data-centric* locality transformations proposed by Pingali *et al.* [17]. 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.

VI. Conclusions and Future Work

Processing and analyzing large volumes of data play an increasingly important role in many domains of scientific research. However, high-level language and compiler support for developing such applications have been so far lacking.

We are developing compiler and runtime techniques necessary to generate efficient implementations of such applications from a high-level programming language. Through advanced compiler analysis, our system is able to extract necessary information from the source code. This information is used to generated an executable that makes extensive use of a runtime system called Active Data Repository (ADR).

In this paper, we have presented advanced techniques that can enhance the performance of compiler generated data intensive codes. We have shown how characterization of access patterns into *dense* and *sparse* and choosing appropriate execution strategy for each leads to better performance. We have also shown how *demand-driven* output space allocation on each processor, managed through a *sparse* data-structure, can reduce the number of tiles that need to be allocated on each processor and enhances performance.

References

[1] Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Michael Beynon, Jeffrey K. Hollingsworth, Joel Saltz, and Alan Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS)*. ACM Press, May 1996.

[2] 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.

[3] Gagan Agrawal, Renato Ferreira, and Joel Saltz. Language extensions and compilation techniques for data intensive computations. In *Proceedings of Workshop on Compilers for Parallel Computing*, January 2000.

[4] Gagan Agrawal, Renato Ferreira, Joel Saltz, and Ruoming J in. High-level programming methodologies for data intensive comput ing. In *Proceedings of the Fifth Workshop on Languages, Compilers, an d Run-time Systems for Scalable Computers*, May 2000.

[5] 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.

[6] 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*