Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism *

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

Renato Ferreira
Departamento de Ciencia da Computacao
Universidade Federal de Minas Gerais
Brasil
renato@dcc.ufmg.br

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

ABSTRACT

The emergence of grid and a new class of *data-driven* applications is making a new form of parallelism desirable, which we refer to as *coarse-grained pipelined* parallelism. This paper reports on a compilation system developed to exploit this form of parallelism. We use a dialect of Java that exposes both pipelined and data parallelism to the compiler. Our compiler is responsible for selecting a set of candidate *filter boundaries*, determining the volume of communication required if a particular boundary is chosen, performing the decomposition, and generating code. We have developed a one-pass algorithm for determining the required communication between consecutive filters. We have developed a cost model for estimating the execution time for a given decomposition, and a dynamic programming algorithm for performing the decomposition. Detailed evaluation of our current compiler using four data-driven applications demonstrate the feasibility of our approach.

1. INTRODUCTION

This paper focuses on language and compiler support for a relative new form of parallelism, which we refer to as *coarse-grained pipelined* parallelism. Here, the processing associated with an application is carried out in several stages. These stages are executed on a pipeline of computing units, each of which handles the intermediate results obtained from the previous stage. Typically, the first stage in this pipeline is the unit where the input data is available, and the last stage is where the final results are viewed.

Two recent trends are making this form of parallelism feasible and desirable. The first trend is the emergence of grid computing. A grid environment facilitates better sharing of data and compute resources. Particularly, the availability of data repositories and access to data collection instruments and sensors is creating a new scenario for execution of many applications.

The second trend is the emergence of a new class of data-driven or data-intensive applications. This class includes scientific data analysis, data mining, data visualization, and image analysis. These applications have so far not received much attention in the system software community. These applications are typically both compute and data intensive, and require fast or even interactive response time. Therefore, these applications are a suitable target for parallelization.

A coarse-grained pipelined execution model is a good match for execution of such data-driven applications in scenarios where the data is available on a repository or a data collection site on the internet, and the final results are required on a user's desktop. It is usually not possible to perform all analysis at the site hosting such a shared data repository or a data collection instrument. Similarly, networking and storage limitations make it impossible to download all data at a single site before processing. Thus, the application needs to be broken into a process or stage that executes on the site hosting or collecting the data, one or more stages that executes on clusters or SMP machines, and a final stage that executes on the user's local machine.

A number of challenges arise in developing data-driven applications executing in such an environment. The computation associated with an application needs to be decomposed into stages, paying attention to both the amount of computation in each of the stages and the volume of communication between them. Moreover, the decomposition decisions are dependent on the environment in which the application will be executed. Another problem is that the code for each of the processing stages needs to be written to iterate over a packet or buffer of data that is received from the previous stage. Finally, we need to exploit shared or distributed memory parallelism which may be available at one or more of the stages.

Thus, we believe that high-level language and compiler support for exploiting coarse-grained pipelined execution of data-driven applications is clearly needed. We are not aware of any previous work in this direction. Recently, some work has been done on runtime support and scheduling for pipelined parallelism [9, 39, 44], but not at the language/compiler level.

This paper describes and evaluates a compilation system we have been developing for exploiting coarse-grained pipelined parallelism for data-driven applications. Our language dialect exposes both the pipelined and data parallelism. Our work is based upon the observation that data-driven applications from many domains, including scientific data analysis, data mining, visualization, and image analysis, frequently involve generalized reductions. Thus, much of the

*This work was supported by NSF grant ACR-9982087, NSF CAREER award ACR-9733520, and NSF grant ACR-0130437. The equipment used for the experiments reported here was purchased under the grant EIA-9986052.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'03 November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011 ...$5.00.

processing associated with different data elements is independent. Such applications are, therefore, not only suitable for execution in a pipelined fashion, but it is also feasible to exploit such parallelism through a compiler.

For code execution, we use a user-level middleware called DataCutter [11, 9]. DataCutter has been developed at the University of Maryland and the Ohio State University. DataCutter exports an interface for specifying processing as a set of coordinating *filters*. A filter can interact with other filters only through a set of *input streams* and a set of *output streams*. Datacutter's support for creating *transparent* copies allows exploitation of shared memory and distributed memory parallelism.

We have carried out a detailed evaluation of our current compiler using four data-driven applications. Two of these applications implement algorithms for *isosurface rendering*, which is one of the common visualization tasks. The two algorithms are based on z-buffer and active pixels, respectively. Another applications is a key data mining kernel, k-nearest neighbor search, and the last application is *virtual microscope*, which processes digitized images.

The summary of the results obtained from our experiments is as follows. We compared compiler decomposed versions against versions that use pipelined parallelism in a *default* fashion, i.e., copy all data from source nodes to compute nodes, and perform all processing there. The compiler decomposed version was faster by between 10% and 30% for isosurface cases, by nearly 150% for k-nearest neighbors, and by nearly 40% for virtual microscope. The speedups obtained from increasing the width of the pipeline were near-linear, with the exception of a small query on virtual microscope. For k-nearest neighbors and virtual microscope, we also compared the compiler generated versions with manual implementation. For k-nearest neighbors, there was no significant difference in performance. For virtual microscope, the compiler generated version was slower by between 10% and 50%.

The rest of the paper is organized as follows. Background information on our target class of applications and the runtime system DataCutter is given in Section 2. Section 3 then sketches our language dialect. In Section 4, we focus on the key compiler analysis phases. A number of issues related to code generation are briefly discussed in Section 5. Section 6 details the experimental results we have obtained. We compare our work with related research issues in Section 7, and conclude in Section 8.

2. BACKGROUND

In this section, we discuss the characteristics of our target class of applications. We then describe an existing runtime system for which our compiler generates code.

2.1 Target Data-Driven Applications

Our focus is on a variety of data-driven applications, arising in domains like scientific data analysis, data mining, visualization, and image analysis, and spanning both scientific and commercial interests.

Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. In many applications, because of computational and storage requirements, and to ensure fault tolerance and high availability, datasets are stored on storage systems distributed across a wide area network, creating a science data grid [35, 34]. The types of datasets that a data grid can store includes, among others, simulations of time-dependent phenomena that periodically generate snapshots of their state [14, 30, 40, 36, 31], archives of raw and processed remote sensing data [33, 27, 32], and archives of medical images [3, 15]. A variety of analysis and processing can be performed on these scientific datasets and images.

Business decisions today are increasingly driven by analysis of data. Various data mining and On Line Analytical Processing (OLAP) techniques are used in decision support systems. Grid technologies are driving the creation of *virtual organizations*, which comprise collections of institutions or entities sharing and analyzing information [17]. These virtual organizations can be expected to use a variety of business decision support functionalities that have been commonly used in centralized organizations. Thus, we expect that data mining and OLAP will be common operations performed in a grid environment.

Two obvious ways of executing an application that processes datasets available in grid-based data repositories are, downloading all data at the user's local machine, or performing all computations at sites hosting data repositories. However, neither of these two options is feasible in most situations. Sufficient storage space and/or network bandwidth is not likely to be available to download all data. At the same time, a node hosting a data repository is not likely to offer sufficient computing power to execute the entire application.

Therefore, a natural option for executing these applications is to use a pipeline of computing resources, where the site hosting the data repository is the first stage and the user's local machine where the results are required is the final stage. Typically, one or more clusters and/or SMP machines serve as the intermediate stage(s).

Mapping of our target class of applications to the pipelined model is facilitated by an important observation. Our study of a variety of scientific and commercial data intensive applications shows that *generalized reduction operations* are very common in the processing structure. This observation applies across a large number of scientific data intensive applications [14, 30, 36, 31, 33, 27, 32], data mining algorithms [25] including association mining, clustering, and decision tree construction, OLAP applications involving algebraic and distributed aggregations [19], and key visualization algorithms such as the ones for isosurface rendering [29]. Processing for generalized reductions consist of three main steps: (1) Retrieving data items of interest, (2) Applying application-specific transformation operations on the retrieved input items, and, (3) Mapping the input items to output items and aggregating, in some application specific way, all the input items that map to the same output data item. Most importantly, aggregation operations involve *commutative* and *associative* operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated. For simplicity, we can refer to the first two steps as *local processing*, and the third step as *global combination*.

The common structure described above has an important implication. Different steps involved in local processing can be performed independently on different sections of data. This enables exploitation of pipelined parallelism, as well as shared memory or distributed memory parallelism available at an intermediate stage of the pipeline.

2.2 DataCutter Runtime System

Our compiler targets a user-level middleware called DataCutter [11, 9]. Datacutter has been developed at the University of Maryland and the Ohio State University. DataCutter supports processing over distributed, heterogeneous environments by allowing decomposition of application-specific data processing operations into a set of interacting filters. By generating code for the *filter-stream* interface that DataCutter supports, our compiler avoids dealing with many low-level tasks, like process initiation and interprocess communication. Moreover, DataCutter's support for *transparent copies* of filters allows flexible shared memory or distributed

memory parallelization.

In the filter-stream programming model, an application is represented as a set of interacting components, referred to as *filters*. Data exchange between any two filters is described via *streams*. The interface for filters consists of an initialization function (*init*), a processing function (*process*), and a finalization function (*finalize*). A *stream* is an abstraction used for all filter communication, and specifies how filters are logically connected. All transfers to and from streams are through a provided buffer abstraction. A buffer represents a contiguous memory region containing useful data. Streams transfer data in fixed size buffers. Filter operations progress as a sequence of cycles, with each cycle handling a single application-defined *unit-of-work*. A work cycle starts when the filtering service calls the filter *init* function, which is where any required resources such as memory or disk scratch space are pre-allocated. Next the *process* function is called to continually read data arriving on the input streams in buffers from the sending filters. The *finalize* function is called after all processing is finished for the current unit-of-work, to allow release of allocated resources such as scratch space.

DataCutter also provides support for *transparent copies*. Transparent copies allow a finer level of parallelism via multiple copies of a single filter. The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. When the logical producer or logical consumer is transparently copied, the system decides for each producer which copy to send a stream buffer to. Schemes like round-robin allocation are used to achieve load balancing.

3. LANGUAGE DIALECT

We use a Java-like language, with a set of constructs to expose both the pipelined and data parallelism to the compiler. This section describes the set of constructs we use. To explain this set of constructs, we use isosurface construction with z-buffers as an example. The outline of the code using this example is shown in Figure 1.

Isosurface rendering is a key visualization problem [29]. The input to an isosurface construction algorithm is a three-dimensional grid, a scalar isosurface value, and a two-dimensional viewing screen with an angle associated with it. A scalar value is associated with every point in the grid. Typically, it denotes certain physical property such as density or pressure. The goal is to view a surface, as seen from the given viewing angle, which captures the points in the grid where the scalar value matches the given isosurface value. Such a surface is referred to as the *isosurface*.

The following steps are used. The grid is processed as a set of cubes, where a scalar value is associated with each corner point. If the isosurface value is greater or lower than the values at each of the eight end points of the cube, it is assumed that the isosurface does not pass through this cube. Otherwise, a set of triangles is generated to approximate the surface passing through the cube.

Next, each such triangle is transformed from the original grid coordinates to the viewing coordinates. Then, it is projected to a two-dimensional image plane (the screen) and clipped to the screen boundaries. Subsequently, the triangles (polygons after the above processing) are accumulated onto a z-buffer. A z-buffer stores a color and distance associated with each pixel or point in the 2-dimensional viewing screen. As polygons are accumulated onto a z-buffer, the color associated with the least distance is chosen.

Overall, this algorithm is well suited for execution in a pipelined fashion. The processing associated with different cubes and triangles is independent. Moreover, after triangles are extracted from a set of cubes, these triangles can be processed independent of the other cubes. Similarly, after triangles are processed to extract polygons, these polygons can be accumulated independent of other triangles or polygons. The accumulation function is associative and commutative, i.e., the final result obtained does not depend upon the order in which triangles or polygons are processed.

The language constructs we use expose the above facts to the compiler. Our constructs include both data parallel constructs, and a special construct to denote the feasibility of performing pipelined processing. Initially, we review the data parallel constructs.

We borrow two concepts from object-oriented parallel systems like Titanium [45] and HPC++ [12].

- A *Rectdomain* is a collection of objects of the same type. Moreover, each object belonging to such a collection has a *coordinate* associated with it, which belongs to a pre-specified rectilinear section.
- The *foreach* loop iterates over objects in a rectdomain, and has the property that the order of iterations does not influence the result of the computations. We further extend the semantics of foreach to include the possibility of updates to *reduction variables*, as we explain later.

In our isosurface example, one-dimensional Rectdomains are used to denote collections of cubes, triangles, or polygons. The foreach loop is used to imply that the order of processing each cube, triangle, or polygon, does not impact the correctness of the result.

To denote that accumulation onto a z-buffer involves associative and commutative operations, we introduce a Java interface called *Reducinterface*. Any object of any class implementing this interface acts as a *reduction variable* [22]. The semantics of a reduction variable are analogous to those used in version 2.0 of High Performance Fortran (HPF-2) [22] and in HPC++ [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.

The goal of the above three extensions is to give the compiler information about independent collections of objects (with no aliasing between the elements), parallel loops and reduction operations. Though it is possible that advanced compilation techniques may be able to infer some or all of this information automatically, for the purpose of focusing our research effort on issues related to filter decomposition and pipelined execution, we have chosen an experimental framework that explicitly provides this information.

The construct $Pipelined_loop$ is used to capture the processing associated with each *packet*. The processing associated with each packet is assumed to be independent, except for updates to any object that belongs to a class that implements the ReducInterface. The size of each packet can be chosen at runtime. This is expressed through the prefix $runtime_define$ to the variable $num_packets$.

In the code shown in Figure 1, the set of input cubes is divided into $runtime_define_num_packet$ packets. Each such packet is processed independently, and a z-buffer is allocated and updated for each packet. Subsequently, the z-buffers computed from each of the packets are merged together. The z-buffer onto which all intermediate z-buffers are merged is defined to implement the ReducInterface, denoting that this merge is associative and commutative.

Though the above constructs make pipelined and data parallelism known to the compiler, the compiler input code is much simpler in several ways than the filter-stream code that DataCutter requires. First, the boundaries between the filters are not specified in the input code. These can be chosen by the compiler with the knowledge

```
Interface Reducinterface {
   // Any object of any class implementing
   // this interface is a reduction variable
}
class CUBE {
   VERTEX[] cube = new VERTEX[8];
      {* member functions ... *}
}
class TRIANGLE {
   TRI_POINT[] tri;
      {* member functions ... *}
}
class TRI_LIST {
   TRIANGLE[1d] data;
   int tri_cnt;
      {* member functions ... *}
}
class ZBUFFER
{
   Points[2d] data;
   int winx, winy;
      {* member functions ... *}
}
class ZBUFFER_Reduce Extends ZBUFFER
         Implements ReducInterface {
   void Merge_two_zbuffers(ZBUFFER recvzbuf) {
      int i, j;
      for (i=0; i<recvzbuf.winy; i++) {
         for (j=0; j<recvzbuf.winx; j++) {
            if (data[i].row[j].z > recvzbuf.data[i].row[j].z)
               data[i].row[j] = recvzbuf.data[i].row[j];
         }
      }
   }
}
```

```
public class isosurface {
   public static void main(String arg[]) {
      float iso_value;
      RectDomain<1> CubeRange = [min_n:max_n];
      InputCube InputData = new InputCube(CubeRange);
      TRI_LIST tri_list = new TRI_LIST();
      Point<1> p;
      int runtime_define_num_packets;
      int n = (max_n - min_n)/runtime_define_num_packets + 1;
      RectDomain<1> PacketRange = [1:runtime_define_num_packets];
      RectDomain<1> EachPacket = [1:n];

      ZBUFFER_Reduce zbuffer = new ZBUFFER(screen.winx,screen.winy);
      ZBUFFER[1d] zbuffer1 = new ZBUFFER[PacketRange];

      Pipelined_loop(b in PacketRange) {
         zbuffer1[b] = new ZBUFFER(screen.winx,screen.winy);
         foreach (p in EachPacket) {
               {* extract triangles *}
            InputData.getCube(p).ISO_SurfaceTriangles(iso_value, tri_list);
         }

         RectDomain< 1 > TriRange = [1:tri_list.size()];
         foreach (p in TriRange) {
            {* process each triangle *}
            {* and compute an edge list *}
         }
         {* Update a Z-Buffer *}
         edgetable.PZBUF_Process_table(zbuffer1[b], screen);

            {* Merge into a single z-buffer *}
         zbuffer.Merge_two_zbuffers(zbuffer1[b]);
      }
   }
}
```

Figure 1: Outline of Z-buffer Based Iso-surface Extraction Code

of the target execution environment. Second, the input code assumes a global address space. In comparison, the code for each filter that our compiler generates involves iterating over a buffer (a linear set of bytes) that is received from another filter. Third, the programmer does not need to worry about what values will be communicated if a particular boundary is used, instead, this is determined by the compiler.

4. COMPILER ANALYSIS

This section focuses on various phases of analysis performed by our compiler. We give an overview of the compilation problem, describe our algorithm for communication analysis, state our cost model, and then describe an algorithm for filter decomposition.

4.1 Overview of the Compilation Problem

We focus on a single loop that iterates over a set of packets. Within this loop, there may be several foreach loops, including one or more loops that update a reduction object. We have a pipeline of available computing resources. The first element in this pipeline is always the data repository or data source, i.e., the computing unit where data is available. The last element in this pipeline is where the final results are desired. We denote the computing units in the pipeline by $C_1, \ldots, C_m$. The connection between units C_i and C_{i+1} is denoted by L_i.

Given such a loop over a set of packets and a computing environment described above, we want to compute a mapping between the computation in the loop to the computing units in our pipeline. Stated in terms of our target runtime system DataCutter, we want to decompose the computation into a set of m filters, and place one filter on each unit in the pipeline.

Several problems need to be addressed here. First, we need to select a set of *candidate* points within the loop, each of which can serve as a filter boundary. Second, we have to choose $m-1$ filter boundaries among the candidate filter boundaries. The compiler may have to extract information from the program to make the above decision. Finally, given the $m-1$ filter boundaries, we need to generate code for each of the m filters.

We currently consider three types of candidate filter boundaries: 1) Start and end of a foreach loop, 2) A conditional statement, either inside or outside a foreach loop, and 3) Start and end of a function call within a foreach loop. We also assume that any loop that is not a foreach loop must be completely inside a single filter. In our target class of applications, such loops typically did not enclose a significant amount of computation, and therefore, this restriction did not limit us in anyway.

We represent the set of candidate filters boundaries and the code between those by a *candidate filter boundary graph*. The nodes in this graph are the candidate filter boundaries, with the exception of a *start node* that pre-dominates all other nodes, and an *end node* that post-dominates all other nodes. An edge in this graph con-

nects two candidate filter boundaries that are *adjacent*, i.e., control in the original code can flow from the first boundary to the second boundary, without reaching any other candidate filter boundary. If there are candidate filter boundaries within a foreach loop, we perform loop fission and create separate foreach loops. This ensures that there are no candidate boundaries inside a foreach loop. With such loop fission, and because we require any other kind of loop to be completely enclosed inside a filter, the candidate filter boundary graph is always acyclic. A *flow path* in this graph is defined to be any path from the start node to the end node.

Given any flow path in our graph, we want to select $m-1$ filter boundaries. Our objective in selecting these $m-1$ filter boundaries is to minimize the total execution time for the pipeline. For this purpose, the compiler needs to extract the set of values that will be communicated if a particular candidate filter boundary is selected. The same information is also required for code generation. Analysis for this is presented in the next subsection.

Note that in our current work, we only perform *static* decomposition of the problem, i.e., the mappings of the tasks to the computing units is not changed during the execution. While this is reasonable for our current set of target applications, it could limit performance in some cases. Considering dynamic decomposition of the problem will be a topic for our future work.

4.2 Required Communication Analysis

We now present a one-pass algorithm for determining the required communication across all candidate filter boundaries.

Consider two consecutive candidate filter boundaries, f_1 and f_2, and let b be the code between these. We use the following terms. $Gen(b)$ denotes the set of values that have been defined in the code section b and are still live at the end of b. $Cons(b)$ is the set of values that are used in b, and are not defined in b. $ReqComm(f_1)$ and $ReqComm(f_2)$ denote the communication required at potential filter boundaries f_1 and f_2, respectively. In our model, all communication occurs only between two consecutive filters. Then, we have

$$ReqComm(f_1) = ReqComm(f_2) - Gen(b) + Cons(b)$$

At the end of the last filter in the candidate filter boundary graph, $ReqComm$ is initialized to be the null set. Thus, using the above equation, and computed values of $Gen(b)$ and $Cons(b)$ for code segment between each pair of consecutive filter boundaries, we can determine $ReqComm$ for each candidate filter boundary using just a single pass over the candidate filter boundary graph.

After computing these terms, we select only a subset of the filter boundaries. An important observation is that the term $ReqComm(f_1)$ computed as stated above still holds correct even if a filter boundary is not inserted at f_2. To explain this, we consider a series of three consecutive candidate filters in our graph, f_1, f_2, and f_3, with code segments between them denoted as b_1 and b_2. $ReqComm(f_2)$ is computed using $ReqComm(f_3)$ and the code segment b_2. Subsequently, $ReqComm(f_1)$ is computed using $ReqComm(f_2)$ and the code segment b_1.

Now, suppose, after our analysis, we insert filter boundaries only at f_1 and f_3, i.e., the code segments b_1 and b_2 are put in the same filter. The computed value of $ReqComm(f_1)$ is still correct. This is because any value required for correct execution for the code in the segment b_2 is either generated inside the code segment b_1, or in code segment before f_1. In the former case, it does not need to be communicated any more, and in the latter case, it is captured in $ReqComm(f_1)$ as computed originally.

Computing $ReqComm$ sets requires the terms Gen and $Cons$ for each code segment that falls between two consecutive filter boundaries. We use a simple one pass algorithm that assumes structured control flow. Thus, the entire analysis for determining $ReqComm$ at each filter boundary can be performed using only a single pass over the program. Though our current implementation is in an off-line compiler, the analysis of the type described here is likely to be implemented in Just-In-Time (JIT) compilers. Therefore, the efficiency of analysis is important.

```
Analyze(Code Segment b) {
  Initialize Gen(b) and Cons(b) to null
  for all statements s in b in the reverse order {
    if s is an assignment statement {
      Gen(b) = Gen(b) + LHS(s)
      Cons(b) = (Cons(b) - LHS(s)) + RHS(s)
    }
    else if s is a conditional block {
      Analyze(s)
      Cons(b) = Cons(b) + Cons(s)
    }
    else if s is a loop {
      Analyze(s)
      Insert_Loop_Range(Gen(s))
      Insert_Loop_Range(Cons(s))
      Gen(b) = Gen(b) + Gen(s)
      Cons(b) = (Cons(b) - Gen(s)) + Cons(s)
    }
  }
}
```

Figure 2: Computing Generated and Consumed Sets Using a Single Pass

The analysis performed on a code segment that falls between two consecutive filter boundaries is shown in Figure 2. We view the code segment b as a sequence of statements. A statement can be either an assignment statement, a conditional statement, or a loop. Our algorithm traverses this sequence of statements in the reverse order. Initially, both $Gen(b)$ and $Cons(b)$ are initialized to be null sets.

For an assignment statement s, $LHS(s)$ denotes variables modified by s and $RHS(s)$ denotes all variables used by s. A variable that is given a value in this statement is added to the $Gen(b)$ set and removed from the $Cons(b)$ set. Any variable that is used in this statement is added to the $Cons(b)$ set. We assume that (potentially conservative) alias information is available. In updating the $Gen(b)$ set, we use the *must alias* information, i.e., a variable can only be added to this set if it is definitely defined. In updating the $Cons(b)$ set, we use the *may alias* information, i.e., any variable that could potentially be used is added to this set.

A conditional statement s inside the segment b is handled as follows. First, we independently analyze the set of statements inside the conditional block. The set of variables $Cons(s)$ is added to the set $Cons(b)$. However, the set $Gen(s)$ cannot be added to the set $Gen(b)$, since the statements in the block s are enclosed in a conditional. Note that a variable that is both defined and used in the block s does not get added to the $Cons(b)$ set.

For handling loops, we assume that a loop cannot be executed for zero iterations. We compute $Gen(s)$ and $Cons(s)$ set for the loop body. If the variables in these sets are accessed using a function of the loop index, we replace these variables by rectilinear sections, derived from loop bounds. Then, the set $Gen(s)$ is added to the set $Gen(b)$ and removed from the set $Cons(b)$. The set $Cons(s)$ is added to the set $Cons(b)$.

There are two important aspects of our analysis that are not shown in Figure 2. First, our analysis is applied interprocedurally. The

main issue in doing this is changing variable names from actual parameters to formal parameters and vice-versa. Note that we perform context-sensitive analysis, i.e., a procedure included in multiple code segments is analyzed independently each time.

Second, in storing and manipulating Gen and $Cons$ sets, we use rectilinear sections, whose bounds may only be available symbolically. We also keep track of fields of classes and handle nested classes.

4.3 Cost Model

Once the required communication across any potential filter boundary is known, we want to decompose the loop over a set of packets into a set of filters. The goal in performing this decomposition is minimizing the predicted execution time. We now present the cost model we use for this purpose.

Recall that we have a pipeline of m computing units, denoted by $C_1, \ldots, C_m$ and the connection between units C_i and C_{i+1} is denoted by L_i. Consider any path from beginning to end in the candidate filter boundary graph. Suppose, we have n candidate filter boundaries, denoted by $b_1, \ldots, b_n$. We assume that $n > m-1$, otherwise the problem becomes trivial.

The goal of the filter granularity analysis is to produce a mapping $L_i \rightarrow j,\ 1 \leq i \leq m-1,\ 1 \leq j \leq n$, which denotes that the candidate filter boundary b_j is inserted between the computing units C_i and C_{i+1}.

Given such a filter decomposition, we want to estimate the total execution time of the entire loop over the set of packets. This depends upon the computation performed over the m computing units, as well as the communication performed over $m-1$ links. We assume that each packet is of the same size, each computing unit offers the same performance throughout the computation, and each link offers the same bandwidth. In this case, either the same computing unit or the same link will be the *bottleneck* for each packet, i.e, each packet will spend the most time there.

Thus, the total execution time comprises two components, the time taken by one packet to reach from the start of the pipeline to the end, and the time taken by all but one packet at the bottleneck. Let the loop iterate over N packets, the time spent by one packet at the computing unit C_i be denoted by $T(C_i)$ and the time spent by one packet in the link L_i be denoted by $T(L_i)$. Then, if the computing unit C_k is the bottleneck, the total execution time is

$$(N-1) \times T(C_k) \ + \ \sum_{i=1}^{m} T(C_i) \ + \ \sum_{i=1}^{m-1} T(L_i)$$

Alternatively, if a link L_k is the bottleneck, the total execution time is given by

$$(N-1) \times T(L_k) \ + \ \sum_{i=1}^{m} T(C_i) \ + \ \sum_{i=1}^{m-1} T(L_i)$$

For estimating the computation and communication times, we currently use simple models. Computation time is determined using the number of floating point and integer operations in the code and the processing power available at each computing unit. The communication time is determined using the volume of the data communicated and the bandwidth available.

4.4 Filter Decomposition

To solve the filter decomposition problem optimally, a brute-force approach is to consider all combination of $m-1$ filter boundary placements over n candidates, and choose the one with the lowest estimated execution time. In this approach, there are a total of C_{m-1}^{n+m-1} placements that will need to be evaluated. This term is exponential in the value of m. However, as we will show in this section, dynamic programming can be used to solve the problem efficiently and accurately.

We formulate the problem as follows. We have m computing units in a pipeline $C_1, \ldots, C_m$, which are connected by $m-1$ communication links $L_1, \ldots, L_{m-1}$. We also have n candidate filter boundaries, which divide the entire program into $n+1$ *atomic filters* $f_1, \ldots, f_{n+1}$. Now, we want to make the decision about how to put these $n+1$ atomic filters over the m computing units, such that the total execution time for the pipeline is minimal. To get the final results on the last computing unit C_m, we can either put the last atomic filter f_{n+1} on it, or we can finish all the computations in the first $m-1$ computing units and then communicate the final results to C_m. Similarly, after determining the placement of the filter f_{n+1}, say on C_m, we can either put filter f_n on C_m, or we can finish all the work from f_1 to f_n before the computation reaches C_m, then forward the results of f_n to C_m.

Let $T[i,j]$ denote the minimum cost of doing computations up to f_i on computing units $C_1, \ldots, C_j$, while the results of f_i are on C_j. That is to say, we insert the candidate filter boundary b_i between computing unit C_j and C_{j+1}. Thus, the lowest cost of completing all $n+1$ filters on m computing units would be $T[n+1, m]$.

We can define $T[i,j]$ recursively as follows. The atomic filter f_i can either be placed on C_j while all computations up to f_{i-1} are completed before it, or it can be finished on previous $j-1$ computing units and the results be forwarded to C_j. So, we have

$$T[i,j] = min \left\{ \begin{array}{l} T[i-1,j] + Cost_comp(P(C_j), Task(f_i)) \\ T[i,j-1] + Cost_comm(B(L_{j-1}), Vol(f_i)) \end{array} \right.$$

Here, the function *Cost_comp* takes the power of a computing unit and the computation task of a filter as inputs, and returns the time of execution. The function *Cost_comm* takes bandwidth of a communication link and the communication volume as parameters, and outputs the time of communication. For the computing unit C_i, the computing power is denoted by $P(C_i)$, and for the link L_i, the available network bandwidth is denoted by $B(L_i)$.

Our algorithm is presented in Figure 3. In order to calculate $T[n+1, m]$, the algorithm needs to fill in all cells in the $(n+2) \times (m+1)$ matrix. Since it takes $O(1)$ time to compute each cell, the total execution time of the algorithm is $O(mn)$.

Even though we use a $(n+2) \times (m+1)$ matrix to record $T[i,j]$ in the algorithm, we only need to consider the values in $T[i-1,j]$ and $T[i,j-1]$ for computing $T[i,j]$. Also, we can store back $T[i,j]$ in the cells used by $T[i-1,j]$ after $T[i,j]$ is computed. Thus, the algorithm only has a space complexity of $O(m)$.

5. CODE GENERATION ISSUES

This section focuses on some of the code generation issues that were addressed in our compiler implementation.

After the set of boundaries has been selected, the compiler needs to generate code for each filter. In our model, each filter has one input stream, with the exception of the filter that reads from the data source itself. Also, each filter has one output stream, with the exception of the filter that generates the final results. The following two challenges arise. First, we need to decide how the data communicated from one filter to another is packed. Second, we need to generate code for each filter that receives and unpacks from the input stream, performs the computations, and then packs data for the output stream.

As described in Section 4.2, our communication analysis phase generates the $ReqComm$ sets, which denote the communication required between each pair of consecutive filters. Let the code inside a filter f comprise of code segments $b_1, \ldots, b_k$ between

```
Decompose {
  Inputs:
    A sequence of n+1 candidate filters f_1,...,f_{n+1} which are devided by n filter boundaries b_1,...,b_n
    A sequence of m computing units C_1,... C_m with computing powers P(C_1),...,P(C_m)
    A sequence of m-1 network links L_1,...,L_{m-1} with bandwidths B(L_1),...,B(L_{m-1})
  Goal:
    T[n+1,m]

  {* init *}
  for j <- 0 to m
    T[0,j] = 0
  for i <- 0 to n+1
    T[i,0] = ∞
  T[1,1] = Cost_comp(P(C_1),Task(f_1))

  {* compute T[1,j], j = 2,...,m *}
  for j <- 2 to m
    T[1,j] = min{T[1,j-1] + Cost_comm(B(L_{j-1}),Vol(f_1)), T[0,j] + Cost_comp(P(C_j),Task(f_1))}

  {* compute T[i,j], i = 2,...,n+1,  j = 1,...,m *}
  for i <- 2 to n+1
    for j <- 1 to m
      T[i,j] = min{T[i,j-1] + Cost_comm(B(L_{j-1}),Vol(f_i)), T[i-1,j] + Cost_comp(P(C_j),Task(f_i))}
}
```

Figure 3: Dynamic Programming Algorithm for Filter Decomposition

consecutive candidate filter boundaries. By processing the sets $Cons(b_1), \ldots, Cons(b_k)$, we can compute the set $Cons(f)$, which denotes the values that are consumed inside the filter f.

Initially, we focus on how the code for each filter is generated. Consider the code in the input language that corresponds to this filter. We primarily focus on collections of objects that are read or written in a foreach loop in this code. Let T be a class whose collection is accessed inside a foreach loop. We initially determine all fields of T that are read or written in the code for this filter. Then, we create a new class $_T$ which includes only these data fields. Before executing an iteration of the foreach loop, we allocate an object of type $_T$. Then, any fields of $_T$ that are communicated from the previous filter are unpacked from the received packet and copied to the allocated object.

Next, we discuss how our compiler decides to pack the elements within the set $ReqComm$. Suppose, for example, the class T has two fields, x and y, that are communicated, and each packet includes $count$ objects of type T. In this simple case, there are two ways in which the packet can be arranged. The first will be

$$< count, t_1.x, t_1.y, \ldots, t_{count}.x, t_{count}.y >$$

and is referred to as the *instance-wise* method. The second will be

$$< count, offset1, t_1.x, \ldots, t_{count}.x, t_1.y, \ldots t_{count}.y >$$

and is referred to as the *field-wise* method. The value $offset1$ denotes the offset for getting the first instance of the field y, and is stored for easing the unpacking. For more general examples, where multiple classes, more than two fields, and/or nested classes are involved, we can use a combination of these two basic methods.

If a field is included in the set $ReqComm$ at the boundary of two filters f_1 and f_2, it could be used in f_2, or in any of the filters following f_2. In the former case, the field belongs to the set $Cons(f_2)$, and in the latter case, it does not. If two or more fields are used in f_2, then it is clearly more efficient to pack them in the instance-wise fashion. However, if one field is used in the receiving filter, and another is packed again and sent to the next filter, then the field-wise method turns out to be more efficient. In parts (a) and (b) of Figure 4, we show unpacking code for instance-wise and field-wise methods, respectively.

Thus, for packing fields of objects, we use the following approach. For each filter that has an output stream, we sort the fields of classes by the first filter whose $Cons$ set they belong to. The fields that are used for the first time in the same filter are packed in the instance-wise fashion. For the fields that are used for the first time in different filters, we use the field-wise fashion, sorting by the order in which they are first read.

Figure 4, part (c), shows code for a filter that involves both unpacking and packing. Here, the input includes three fields x, y and z from class T, among which x and z are used by the current filter, thus packed together in the instance-wise fashion. Because the field y is used by later filters, so it is packed separately in the field-wise fashion. After processing, the current filter creates a new data-structure s_list, whose x field is required by one of the following filters. Together with $T.y$, $s_list.x$ is packed as field-wise and forwarded.

6. EXPERIMENTAL RESULTS

The techniques described in this paper have been implemented in a prototype compiler, which is based on the Titanium [45] infrastructure. This section reports the results we obtained from our current compiler. Initially, we describe the applications we used, the versions we created, and the configurations in which the applications were executed. Then, we present the results obtained from each of the applications.

6.1 Applications

We have used four data-driven applications. Two of these applications implement algorithms for *isosurface rendering*, which is one of the common visualization tasks. The third algorithm is a key data mining kernel, k-nearest neighbor search, and the last application is *virtual microscope*, which processes digitized images.

We now describe these applications in more details. The basic isosurface rendering problem was described in Section 3. Z-buffer and active pixels are two of the popular algorithms used for this purpose [29]. The initial steps, i.e. extraction of triangles and transformations on triangles, are identical in these two algorithms.

```
_t = new _T();
Inptr = Inbuf.getPtr();
memcpy(&count, Inptr, sizeof(int));
Inptr += sizeof(int);
for(int i=0; i<count; i++) {
   memcpy(&(_t− >x), Inptr+0+i*(sizeof(t.x)+sizeof(t.y)), sizeof(int));
   Inptr += sizeof(int);
   memcpy(&(_t− >y), Inptr+sizeof(t.x)+i*(sizeof(t.x)+sizeof(t.y)), sizeof(int));
   Inptr += sizeof(int);
   {* some processing on _t *}
}
```

(a)

```
_t = new _T();
Inptr = Inbuf.getPtr();
memcpy(&count, Inptr, sizeof(int));
Inptr += sizeof(int);
for(int i=0; i <count; i++) {
   off = 0;
   memcpy(&(_t− >x), Inptr+off+sizeof(int)+0+i*(sizeof(t.x)), sizeof(int));
   off = 0;
   for (int k=0; k<1; k++) {* order of y is 1 *}
   {
      memcpy(&size, Inptr+off, sizeof(int));
      off += size;
   }
   memcpy(&(_t− >y), Inptr+off+sizeof(int)+0+i*(sizeof(t.y)), sizeof(int));
   {* some processing on _t *}
}
```

(b)

```
_t = new _T();
Inptr = Inbuf.getPtr();
memcpy(&count, Inptr, sizeof(int));
Inptr += sizeof(int);
{* unpacking *}
for(int i=0; i<count; i++) {
   off = 0;
   memcpy(&(_t− >x), Inptr+off+sizeof(int)+0+i*(sizeof(t.x)+
         sizeof(t.z)), sizeof(int));
   off = 0;
   memcpy(&(_t− >z), Inptr+off+sizeof(int)+sizeof(t.x)+
         i*(sizeof(t.x)+sizeof(t.z)), sizeof(int));
   {* some processing on _t and transform it to s *}
   {* put s in s_list *}
}
{* packing *}
Outptr = Outbuf.getPtr();
memcpy(Outptr, &count, sizeof(int));
Outptr += sizeof(int);
head = Outptr;
Outptr+=sizeof(int); {* save for size info *}
for(int i=0; i<count; i++) {
   memcpy(Outptr, &(s_list− >get(i)− >x), sizeof(int));
   Outptr += sizeof(int);
}
size=Outptr-head;
memcpy(head, &size, sizeof(int));
head = Outptr;
Outptr+=sizeof(int); {* save for size info *}
off = 0;
for (int k=0; k < 1; k++) {
   memcpy(&size, Inptr+off, sizeof(int));
   off += size;
}
for(int i=0; i < count; i++) {
   memcpy(Outptr, Inptr+off+sizeof(int)+0+i*(sizeof(t.y)), sizeof(int));
   Outptr += sizeof(int);
}
size=Outptr-head;
memcpy(head, &size, sizeof(int));
```

(c)

Figure 4: Examples of Unpacking: Instance-wise (a) and Field-wise (b), and Full Generated Code (c)

However, they differ significantly in how the triangles or polygons are projected to a screen. In the z-buffer approach, an image is created corresponding to a set of triangles in a packet or on a processor. Then, these z-buffers or images are merged to create the final image. The active pixel algorithm removes some of the computational and memory inefficiencies associated with the z-buffer algorithm. Essentially, it uses a sparse representation of the dense z-buffer, and avoids allocating, initializing, or communicating a full z-buffer.

We implemented these two algorithms using our language dialect. In our discussion, they are referred to as `z-buffer` and `active-pixel`, respectively. The code in our input language dialect was nearly 1400 lines for each of these.

Our third application is k-nearest neighbor search, referred to as `knn`. It is one of the basic data mining problems [21]. Here, the training samples are described by an n-dimensional numeric space. Given a new point, the goal is to find the k training samples that are closest to the new point.

The final application we used is virtual microscope [15], denoted by `vmscope`. A virtual microscope is designed to interactively view and process digitized data arising from tissue specimens. The input data is a high-resolution digitized image. This application emulates the usual behavior of a physical microscope, including continuously moving the stage and changing magnification or resolution.

The code in our language dialect for both `knn` and `vmscope` was relatively small, i.e., under 200 lines.

6.2 Experiment Design

Our overall goal was to demonstrate that compiler generated pipelined code is efficient, and can effectively use a pipeline of computing units.

In the long run, we expect that pipelined parallelism can be exploited in wide-area networks. However, this is going to require high bandwidth networks and certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [38]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. We assumed that input data is available on only a subset of the nodes, and final results are required or viewed on a single node. The cluster we used had 700 MHz Pentium machines connected through Myrinet LANai 7.0.

We used three different pipeline configurations. In the first configuration, the data is available at a single node, another node is available for computations, and the results are required at yet another node. This configuration is designated as the `1-1-1` config-

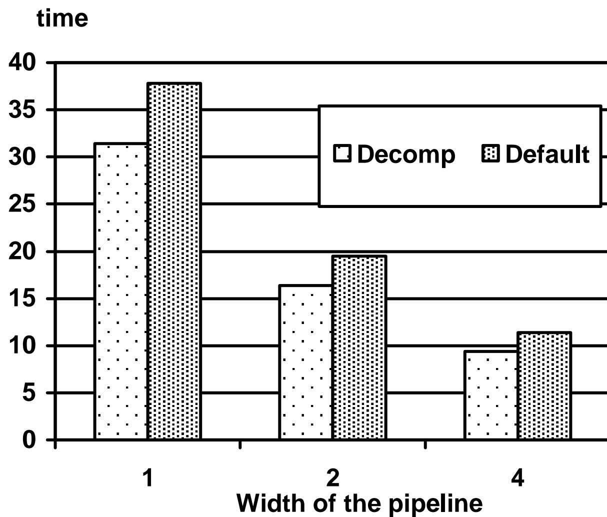

Figure 5: Results from Z-Buffer Based Isosurface Rendering, *small* dataset

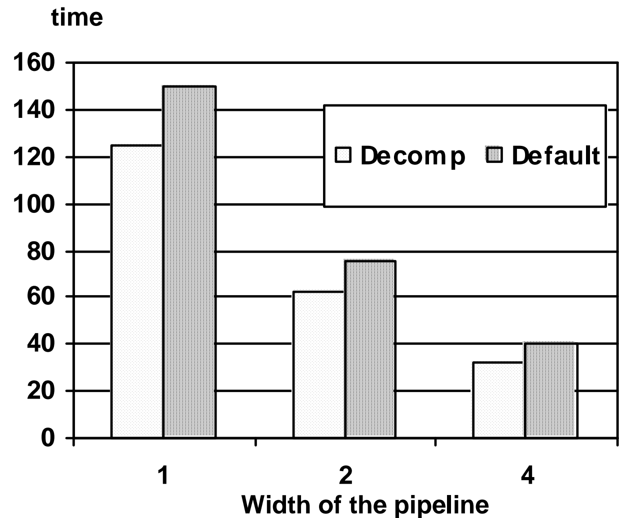

Figure 6: Results from Z-Buffer Based Isosurface Rendering, *large* dataset

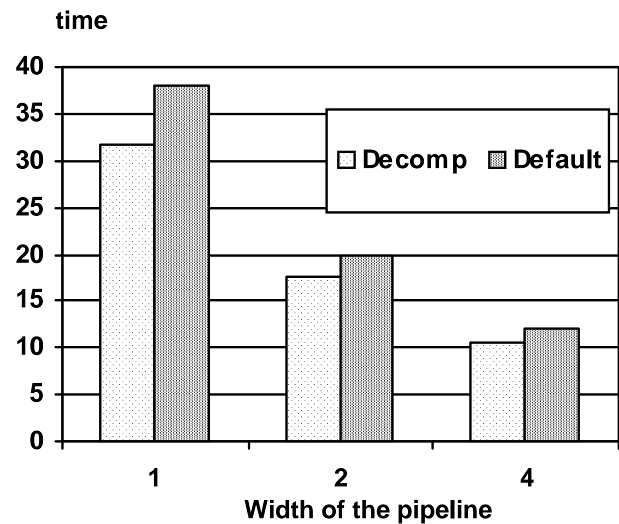

Figure 7: Results from Active Pixel Based Isosurface Rendering, *small* dataset

uration. In the second configuration, data is available at 2 nodes in the cluster, another 2 nodes are available for computations, and the final results are required at another node. This configuration is designated as the `2-2-1` configuration. In the third configuration, referred to as the `4-4-1` configuration, data is available at 4 nodes, another 4 nodes are available for computations, and the final results are required at another node. These three configurations are also referred to as configurations with pipeline widths of 1, 2, and 4, respectively.

To evaluate our compiler, we created two or three versions for each of our applications. One version is denoted as `Default`, and reflects a simple way of using pipelined parallelism. Here, the nodes hosting the data only read the data and transmit it to the compute nodes. The final results are generated on these compute nodes, and then copied on the node where they are viewed. Our experiments contrasted these versions with versions in which more intelligent decomposition is performed by the compiler. Such latter versions are referred to as `Decomp`. Typically, these version performed more computations on either or both of the data source nodes and the node where the results were required, and reduced the volume of data that had to be communicated between the nodes. For `knn` and `vmscope`, we also compared the performance of compiler generated code with the manually written DataCutter codes, where similar decomposition was performed. For these two applications, the compiler and manual versions are referred to as `Decomp-Comp` and `Decomp-Manual`, respectively.

6.3 Isosurface Algorithms

We now describe the results we obtained from the two isosurface algorithms, `z-buffer` and `active-pixels`.

We used two datasets to evaluate both of these algorithms. These datasets were generated by an environmental simulator ParSSim [4] and were previously used by Beynon *et al.* in the context of DataCutter [10]. These datasets comprised grid data for 10 time-steps, and were 1.5 GB and 6 GB, respectively. The two datasets are referred to as `small` and `large` datasets. We report experiments on processing a single time-step, the data corresponding to which is 150 MB and 600 MB, respectively.

As we stated previously, we created two versions, `Default` and `Decomp`, for both the algorithms. We did not have access to

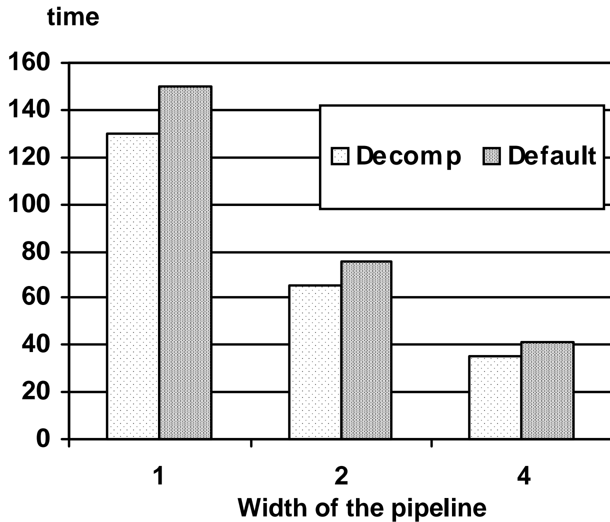

Figure 8: Results from Active Pixel Based Isosurface Rendering, *large* dataset

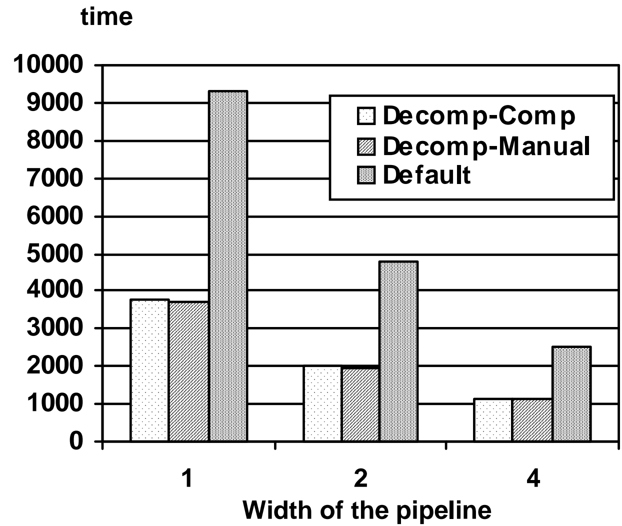

Figure 9: Results from k-nearest neighbors, k = 3

comparable manual versions. The `Default` version reads data in packets, each of which comprises a set of cubes, from the nodes hosting data and copies them to compute nodes. Compute nodes receive data in packets, and perform all processing. In the `Decomp` version, the compiler places some preprocessing of the cubes at the nodes where the data is. In particular, a loop that checks whether all corners of a cube have an isosurface value greater or lower than the specified isosurface value is placed at data nodes. Besides reducing the amount of computation that needs to be performed on compute nodes, this reduces the volume of data that needs to be copied from data nodes to compute nodes.

Figure 5 shows experimental results from `z-buffer`, using the small dataset. We show results on three configurations, `1-1-1`, `2-2-1`, and `4-4-1`. Compiler based decomposition consistently gives 20% improvements on all three configurations. By increasing the width of the pipeline, we get significant speedups. With a width of 2, the compiler decomposed version gives a speedup of 1.92, and with a width of 4, the speedup is 3.34.

Figure 6 shows experimental results from `z-buffer`, using the larger dataset. Again, the performance improvements from the compiler decomposed versions are between 20% and 25%. For the compiler decomposed version, a speedup of 1.99 is obtained with the pipeline width of 2, and a speedup of 3.82 is obtained with the pipeline width of 4.

The results from `active-pixel` algorithm are presented in Figures 7 and 8. We see almost the same trends. The `Decomp` versions outperform the `Default` versions by between 15% and 25%. The speedups obtained from increasing the width of the pipeline are quite close to linear.

6.4 k-nearest Neighbor

For our experiments with `knn`, we used a 108 MB dataset comprising 4.5 million three-dimensional points. We used two test cases, with values of k being 3 and 200.

We used three versions. In the `Default` version, all input data is forwarded in packets to compute nodes. In both `Decomp-Comp` and `Decomp-Manual` versions, the volume of data to be transmitted is reduced. These two versions only differed in how the data received in a packet was iterated on.

The results are presented in Figures 9 and 10. In both the test

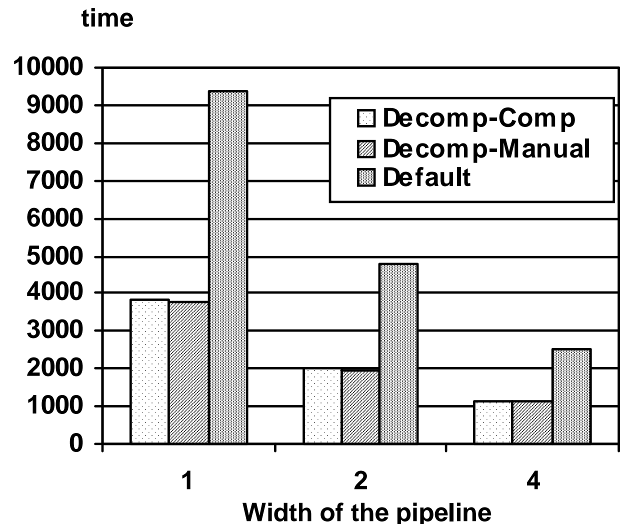

Figure 10: Results from k-nearest neighbors, k = 200

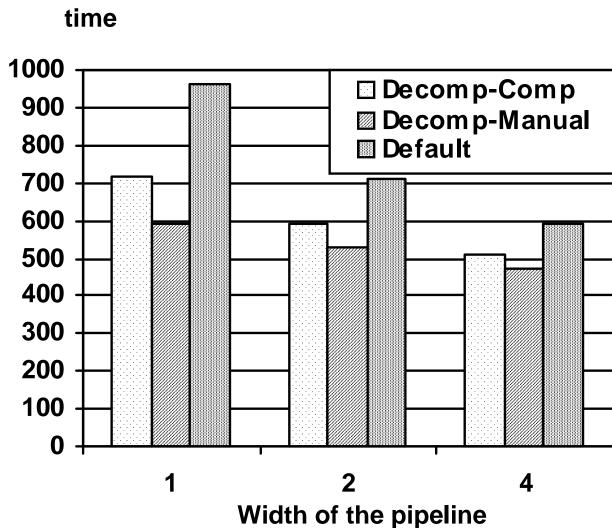

Figure 11: Results from Virtual Microscope, Small Query

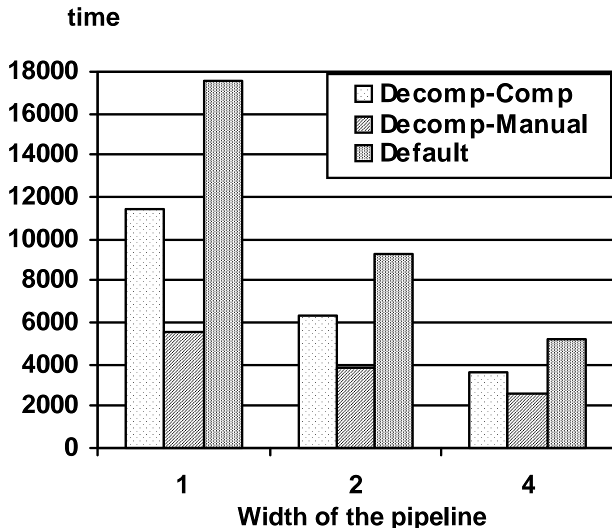

Figure 12: Results from Virtual Microscope, Large Query

cases, the performance difference between `Decomp-Comp` and `Decomp-Manual` versions is insignificant. The `Default` version is consistently slower by almost 150% in all cases. This is because of the large volume of data that is transferred by this version. All versions achieve good speedups as the width of the pipeline is increased. With the `Decomp-Comp` version and $k = 3$, the speedups are 1.89 and 3.38 with pipeline width of 2 and 4, respectively. Similarly, with $k = 200$ on the same version, the speedups are 1.87 and 3.35, respectively.

6.5 Virtual Microscope

The last application we experimented with is `vmscope`. We used a 800 MB image for our experiments. A query specifies a rectangular region and a *subsampling* or resolution factor for the desired output. We used two queries. The first, referred to as the `small` query, required a 512 × 512 output with a subsampling factor of 2. The second, referred to as the `large` query, required a 2048 × 2048 output with a subsampling factor of 4.

Like `knn`, we generated three versions. `Decomp-Manual` and `Decomp-Comp` versions reduced the volume of data that had to be copied from the data sources to compute nodes. These two versions again differed on how the data received in a packet is iterated on.

The results from the `small` and `large` queries are presented in Figures 11 and 12, respectively. For `small` query, the amount of data processed is quite small. It is hard to achieve a good load balance between the different data nodes and the different compute nodes. As a result, the speedups are very limited. With a pipeline width of 1, the `Decomp-Comp` version is slower than the `Decomp-Manual` version by nearly 20%, but is faster than the `Default` version by nearly 40%. The same trend is seen between these versions as the pipeline width is increased, but the relative differences are smaller. This is because load imbalance is the biggest factor contributing to the performance in these cases.

The main difference between compiler generated and manual code is as follows. In this application, only every $subsamp^{th}$ element along each dimension is picked from the original image. In the compiler generated version, a conditional is used for this purpose whereas the manual version simply uses a stride in reading from input buffers. Since the application does not involve a lot of computation, this made a significant difference in the performance.

For `large` query, a good load-balance could be obtained, and therefore, good speedups were obtained. However, because of a larger subsampling factor, the differences between manual and compiler versions are even larger. The compiler decomposed version is consistently faster than the `default` version by nearly 40%.

7. RELATED WORK

We are not aware of any previous work on developing language and compiler support for coarse-grained pipelined parallelism, or for data-driven applications in a distributed environment.

Several efforts have targeted runtime support for coarse-grained pipelined parallelism. The Stampede project [39] has focused on interactive multimedia applications, which have several common characteristics with the applications we have targeted. The support offered is in the form of cluster-wide threads and shared objects. They do not include high-level language or compiler support, but can handle more dynamic applications. Yang *et al.* have developed a scheduler for vision applications which are executed in a pipelined fashion within a cluster [44]. They include support for meeting real-time constraints, but require low-level programming of applications.

Several other projects offer support which is somewhat similar to our target runtime system, DataCutter. This includes the Remos project from CMU and active streams and related efforts from Georgia Tech [13, 24, 37]. We believe that the techniques we are developing can also be used to offer high-level interface to these systems.

Our work has some similarities with the StreamIt effort at MIT [41]. Though the set of applications that have been experimented with are different, streaming applications have some similarities with the data-driven applications we are targeting. Our language is at a higher-level, but we can only handle applications that involve generalized reductions. Also, the target architecture of the two efforts are quite different.

There is very little compiler work in the context of grid computing. One other project that we are aware of is from Adve *et al.* [1]. They have focused on language and compiler support for adapting applications to resource availability in a distributed environment. In comparison, our work focuses on a very different set of applications. We require fewer extensions in our language support, but do not include the support for modifying the precision of computation in adapting to resources. The GrADS Project focuses on opti-

mizing programs that invoke functions from standard libraries [7]. In comparison, our focus is on exploiting coarse-grained pipelined parallelism that we defined earlier. The grid computing community has generally focused on providing services, protocols, low-level APIs, system-level middleware systems and user-level middleware systems [5, 8, 16, 18, 20, 28, 43]. In the past, several research projects have focused on the use of heterogeneous environments for parallel computing. A representative project is HenCE [6]. Our work is distinct in supporting a high-level language that hides heterogeneity from the programmers.

A significant part of the analysis our compiler performs is *communication analysis*. Though communication analysis has been studied extensively as part of SPMD compilers [2, 23], our analysis is different because of supporting a different form of parallelism. A large body of work exists on compiler support for instruction-level or fine-grained pipelined parallelism [26]. Our work considers a very different target environment. Wang *et al.* have worked on analysis of communication pipelines [42]. The cost model we have used is derived from their work.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed a number of design and algorithmic issues in providing high-level programming support for coarse-grained pipelined parallelism. Our basic premise has been two-fold. First, we believe that the availability of data repositories and data collection instruments on the internet is making this form of parallelism desirable and feasible for a variety of data-driven applications. Second, we believe that language and compiler support will be crucial for exploiting such parallelism, as manual decomposition of the processing is time-consuming and error-prone.

The results from our current compiler have been very encouraging. We have shown that compiler-based decomposition of applications can achieve significantly better performance as compared to what we considered the *default* way of using pipelined parallelism. Note, however, that it is currently not easy to implement such default pipelined parallelism, either. Most often, all data is downloaded on compute nodes, and then processing is performed. This is generally much slower than what we used as the baseline in our experiments. By using the support for transparent copies in our target runtime system DataCutter, we have also achieved good speedups from increasing the width of the pipeline.

In the future, we will like to expand our work in many directions. First, we will like to consider data resident in multiple data repositories. Second, we will like to consider an environment where available compute and communication resources can change at runtime. Generating code that can adapt to such changes is an interesting problem. Our cost models also need to be evaluated further. Automatically choosing the packet size is another issue. We will also like to expand the set of applications, particularly, including applications that do not involve generalized reductions.

Acknowledgements: We are extremely grateful to our shepherd, Larry Carter, who gave us the outline of the dynamic programming algorithm used in Section 4.4. We are also grateful to the members of DataCutter project, including Tahsin Kurc, Umit Catalyurek, Mike Beynon, Alan Sussman, and Joel Saltz, for providing us with DataCutter implementation, manual codes, datasets, and for helping us with our experiments.

9. REFERENCES

[1] Vikram Adve, Vinh Vi Lam, and Brian Ensink. Language and Compiler Support for Adaptive Distributed Applications. In *Proceedings of the SIGPLAN workshop on Optimization of Middleware (OM) and Distributed Systems*, June 2001.

[2] Vikram Adve and John Mellor-Crummy. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[3] 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.

[4] T. Arbogast, S. Bryant, C. Dawson, and M. F. Wheeler. Parssim: The parallel subsurface simulator, single phase. *http://www.ticam.utexas.edu/∼arbogast/parssim*.

[5] D. Arnold, H. Casanova, and J. Dongarra. Innovation of the netsolve grid computing system. *Concurrency Practice and Experience*, 2002.

[6] Adam Beguelin, Jack J. Dongarra, George Al Geist, Robert Manchek, and Keith Moore. HenCE: A Heterogenuous Network Computing Environment. *Scientific Programming*, 3(1):49–60, 1994.

[7] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Keselman, J. Mellor-Crummery, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[8] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS . *IEEE Transactions on Parallel and Distributed Systems (to appear)*, 2003.

[9] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[10] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. A component-based implementation of iso-surface rendering for visualizing large datasets. Technical Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS, May 2001.

[11] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. In *Proceedings of the Conference on Cluster Computing and the Grid (CCGRID)*, pages 56–63. IEEE Computer Society Press, May 2001.

[12] 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.

[13] Fabian E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Active Streams and the Effects of Stream Specialization. In *Poster in Proc. of Tenth International Symposium on High Performance Distributed Computing (HPDC-2001)*. IEEE Computer Society Press, August 2001.

[14] Srinivas Chippada, Clint N. Dawson, Monica L. Martínez, and Mary F. Wheeler. A Godunov-type finite volume method for the system of shallow water equations. *Computer Methods in Applied Mechanics and Engineering (to appear)*, 1997. Also a TICAM Report 96-57, University of Texas, Austin, TX 78712.

[15] 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.

[16] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. MK, 1999.

[17] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 2001.

[18] D. Gannon and A. Grimshaw. Object-based approaches. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 205–236. Morgan Kaufmann, 1999.

[19] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by,

cross-tab, and sub-totals. In *Proceedings of the 1996 International Conference on Data Engineering*, pages 152–159, February 1996.

[20] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 1994.

[21] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[22] 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.

[23] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[24] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[25] Ruoming Jin and Gagan Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[26] S. M. Krishnamurthy. A brief survey on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[27] Land Satellite Thematic Mapper (TM). *http://edcwww.cr.usgs.gov/nsdi/html/landsat_tm/ landsat_tm*.

[28] M. Livny. High throughput resource management. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 311–337. Morgan Kaufmann, 1999.

[29] W. Lorensen and H. Cline. Marching Cubes: A High Resoltion 3D Surface Reconstruction Algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[30] Richard A. Luettich, Johannes J. Westerink, and Norman W. Scheffner. *ADCIRC*: An advanced three-dimensional circulation model for shelves, coasts, and estuaries. Technical Report 1, Department of the Army, U.S. Army Corps of Engineers, Washington, D.C. 20314-1000, December 1991.

[31] Kwan-Liu Ma and Z.C. Zheng. 3D visualization of unsteady 2D airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–31, Oct 1994.

[32] The Moderate Resolution Imaging Spectrometer. *http://ltpwww.gsfc.nasa.gov/MODIS/MODIS.html*.

[33] 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*.

[34] Grid Physics Network. GriPhyN. http://www.griphyn.org.

[35] Ron Oldfield. Summary of existing and developing data grids. White paper, Remote Data Access Group, Global Grid Forum, available from http://www.sdsc.edu/GridForum/RemoteData/Papers/papers.html.

[36] G. Patnaik, K. Kailasnath, and E.S. Oran. Effect of gravity on flame instabilities in premixed gases. *AIAA Journal*, 29(12):2141–8, Dec 1991.

[37] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[38] Teragrid project partners. The TeraGrid: A Primer, September 2002. Available at www.teragrid.org.

[39] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proceedings of the Conference on Principles and Practices of Parallel Programming (PPoPP)*, pages 183–192. ACM Press, May 1999.

[40] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: calculation by a new MHD. *Jounal of Geophysical Research*, 98(A10):17251–62, Oct 1993.

[41] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of Conference on Compiler Construction (CC)*, April 2002.

[42] R. Y. Wang, A. Krishnamurthy, R. P. Martin, T. E. Abderson, and D. E. Culler. Modeling Communication Pipeline Latency. In *Proceedings of the ACM SIGMETRICS Conference*. ACM Press, June 1998.

[43] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 1998.

[44] M. T. Yang, R. Kasturi, and A. Sivasubramaniam. An Automatic Scheduler for Real-Time Vision Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[45] 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.