



Improving Data Access for Computational Grid Applications

RON OLDFIELD

Scalable Computing Systems, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1110

DAVID KOTZ

Department of Computer Science, Dartmouth College, 6211 Sudikoff Laboratory, Hanover, NH 03755

Abstract. High-performance computing increasingly occurs on “computational grids” composed of heterogeneous and geographically distributed systems of computers, networks, and storage devices that collectively act as a single “virtual” computer. A key challenge in this environment is to provide efficient access to data distributed across remote data servers. Our parallel I/O framework, called Armada, allows application and data-set providers to flexibly compose graphs of processing modules that describe the distribution, application interfaces, and processing required of the dataset before computation. Although the framework provides a simple programming model for the application programmer and the data-set provider, the resulting graph may contain bottlenecks that prevent efficient data access. In this paper, we present an algorithm used to restructure Armada graphs that distributes computation and data flow to improve performance in the context of a wide-area computational grid.

1. Introduction

An exciting trend in high-performance computing is the development of geographically-distributed networks of heterogeneous systems and devices, known as computational grids [15]. Grid applications use high-speed networks to logically assemble collections of resources such as scientific instruments, supercomputers, databases, and so forth. One important challenge facing grid computing is efficient I/O for data-intensive grid applications. Data-intensive grid applications are particularly challenging because they require access to large (terabyte-petabyte) remote data sets and often have computational requirements that can only be met by high-performance supercomputers. In addition, data is often stored in “raw” formats and requires significant preprocessing or filtering before the computation can take place. Such applications exist in seismic processing, climate modeling, physics, astronomy, biology, chemistry, and visualization [22].

The Armada framework for parallel I/O [23] provides a solution for data-intensive applications in which the application programmer and the data set provider deploy a network of application-specific and data-set-specific functionality across the grid. Using the Armada framework, grid applications access remote data by sending data requests through a graph of distributed application objects called *ships*. Data is pushed toward the client for reads or pulled toward the servers for writes. (Currently, a graph can only be used for reading or writing, but not both. Since workload studies [20] indicate the vast majority of scientific applications do not simultaneously read and write the same file, this restriction is not a series limitation.)

Armada is not a parallel file system, nor does the system itself store any data. The set of *data segments* that make up a *data set* are each stored in conventional data servers, as files, as databases, or the like. Indeed, a data set provider can build a graph on top of legacy files and data bases, present a network of similar data sets through a standard interface, or provide transparent access to derived virtual data sets—either cached or calculated as needed. The graph encodes most functionality provided by the I/O system, including the programmer’s interface, data layout, caching and prefetching policies, and interfaces to heterogeneous data servers. A graph, once deployed on a computational grid, appears to the application as an object providing access to a specific type of data through a high-level interface.

While it is possible for an application to construct an entire graph from scratch, we believe a typical graph will consist of two distinct portions: a portion that describes the layout of the data (usually from a data provider), and an application-specific portion that describes the interface and preprocessing required by the application. Figure 1, for example, shows a graph for an application with read access to a replicated and distributed data set. The portion from the data provider describes the layout of two replicas, each stored as a distributed file. The application prepends a graph to the data provider’s portion that includes a filter and three interface ships (one for each client processor).

Two issues prevent the efficient mapping of the resulting graph from figure 1 to a computational grid. First, the connection between the application’s filter and the data-provider’s replica selector forms a bottleneck. Second, the configuration of the graph restricts the placement alternatives for the filter. For example, if in figure 1 the network between the two replicas is slow, or the network between the replicas and the clients is slow, there is no placement of the filter that allows us to significantly reduce the amount of data transferred over a

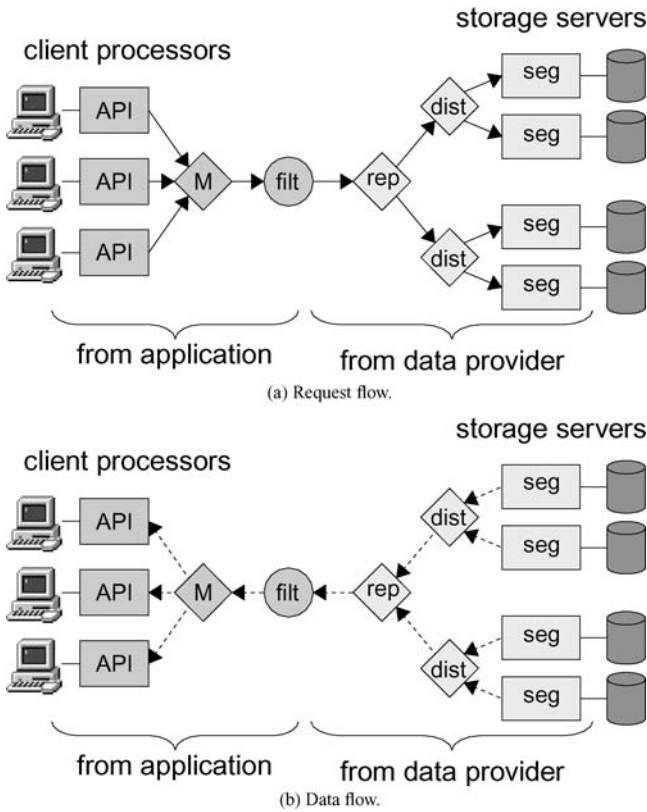


Figure 1. The figures show the Armada graph for an application with read-only access to a replicated and distributed data set. (a) shows the request-flow, (b) shows the data-flow.

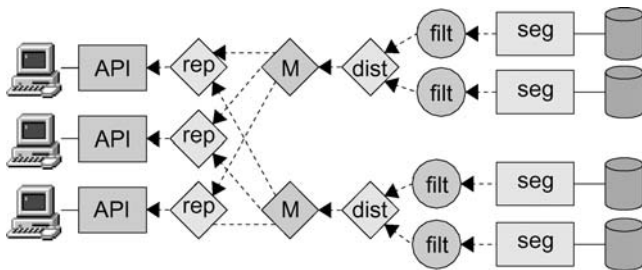


Figure 2. The graph from figure 1 restructured to allow the filters to execute close to the data servers.

slow network. Figure 2 shows a restructured graph that mixes the application’s portion of the graph and the data-provider’s portion of the graph. The new graph has a parallelized version of the replica-select ship to match the number of clients, and it has a parallelized version of the filter to match the number of data servers. The result provides end-to-end parallelism and allows the filters to be placed near (in terms of connectivity) to the data servers, thus reducing the amount of data transferred over a potentially slow network. Armada can automatically restructure graphs in this way.

In previous papers, we describe the Armada framework [21, 23]. In this paper, we present a formal description of the components that make up an Armada graph, and we describe and analyze an algorithm for restructuring a graph based on

programmer-specified properties of the individual ships. We then present performance results demonstrating the value of restructuring on three applications.

2. Related work

The two primary features of our restructuring algorithm are the automatic parallelization of ships and the reordering of ships to improve data-flow performance. This section describes a few projects with related goals.

2.1. Automatic parallelization of user code

Since parallel computation became popular in the 1980’s, many projects have tried to make the task of programming parallel applications easier for the developer [5,8]. One approach attempts to automatically convert sequential programs into parallel programs. A Carnegie Mellon University web page provides a comprehensive reading list on parallel programming languages and systems that support automatic parallelization. Here we discuss three systems of particular relevance to Armada because of their focus on parallel processing of I/O streams.

The Parallel Storage-and-Processing Server (PS²) [18], from École Polytechnique Fédérale de Lausanne, is designed specifically for I/O-intensive applications. It uses the Computer-Aided Parallelization tool (CAP) [16] to express the parallel behavior of the application. The CAP system constructs a data-flow computation graph with “actors” as nodes of the graph. The actors are computational units that provide application-specific functionality. For I/O-intensive applications, actors provide application-specific data distribution, prefetching, or filtering. CAP parallelizes a task by inserting *split* actors to create and distribute data and *merge* actors to gather and synchronize processed data after computation.

A project at Duke University is investigating extensions to the streaming computation model that map computation to active disks [32]. They specify primitive computations as “functors” that perform simple computations as a side-effect of I/O access. Functors have bounded compute and memory requirements. The application exposes functors and their compute costs to the system and the system decides (based on analytic models) a mapping of functors to active disks in a way that balances load. They are primarily interested in applications that implement I/O-efficient external memory algorithms discussed in [4,30].

DataCutter [7], developed at the University of Maryland, is middleware used to explore and analyze scientific datasets stored on archival storage systems across a wide-area network. DataCutter provides a query-based interface with support for accessing subsets of datasets and for performing user-defined transformations of large data sets in archival storage. The processing structure is composed of a set of processes called “filters” that typically execute close to the data source.

Spencer et al. describes how DataCutter can replicate filters to distribute computation inside a pipelined data-flow graph [29]. The scheme uses an analytic model, based on measured performance, to decide how many additional filters to create, then it distributes data elements to filters using either a round robin, weighted round robin, or demand-driven distribution policy. The system then merges results from the distributed filters and forwards the result to the next stage in the data-flow graph.

The systems described above all use a common algorithmic technique known as *partition-and-merge*. In this technique, the system first divides the problem into subtasks that execute in parallel, then it merges results in a single synchronous step. The limitation of this approach is the synchronization point (a bottleneck) where the data merges back together. Armada takes a different approach. The application portion of the Armada graph describes the layout of data to parallel compute nodes with the assumption of a single data source. The data provider describes its layout of the data to parallel storage servers with the assumption of a single compute node for the application. The Armada system combines the two graphs and restructures to provide end-to-end parallelism in an attempt to remove synchronization points.

2.2. Operation ordering to improve data flow

Another feature of the Armada system is the ability to re-order operations in the graph to improve data flow to or from the client. This approach extends the work of projects in the distributed database community that have investigated data-flow optimization for many years [3,13]. Two recent systems of particular interest are the dynamic QUery OBject (dQUOB) system, and River.

The dynamic QUery OBject (dQUOB) system [27] is a runtime system for managing and optimizing large data streams in which the users request remote data with SQL-like queries, and computations are performed on the data stream between the remote data servers and the client. After the dQUOB compiler converts an SQL query to a “query tree,” the tree is evaluated at runtime to determine which portions of the query tree apply the most filtering. The tree is then restructured to move the high-filtering portions closer to the data.

River [2,3] is a data-flow programming environment for database query processing applications. River is specifically designed for clusters of computers with heterogeneous performance characteristics. The goal of the River runtime system is to adapt to “performance faults”—portions of the system that perform poorly— by dynamically adjusting the transfer of data through the data-flow graph. River uses two constructs to build applications: a distributed queue that deals with performance faults by consumers, and graduated declustering that deals with performance faults of producers. A distributed queue pushes data through the data-flow graph at a rate proportional to the rate of consumption and adapts to changes in consumption rates. Graduated declustering deals with producer performance faults by reading from replicated

producers. Although River is designed specifically for query processing, the authors briefly discuss how one might adapt scientific applications to work in their framework [2].

Much of the success of distributed query processing systems can be attributed to the *relational model*. Relational operators have well-defined properties that are ideal for parallel processing execution [12] and easy to describe mathematically. Armada presents a more general approach that allows the reordering of arbitrary functions (not just query processing functions) based on programmer-supplied properties that help the system to decide commutativity between adjacent functions.

3. Background

We begin with a description of Armada’s ships, providing details about the flow of requests and data through the ships, the types of ships available, and programmer-assigned properties. Then we describe the use of series-parallel trees to represent the structure of an Armada graph.

3.1. Armada ships

An Armada graph is a set of interconnected ships that form two directed acyclic graphs (DAGs): one for the flow of requests (*request graph*) and one for the flow of data (*data graph*). As described above, ships provide nearly all essential I/O functionality except storage. From an operational perspective, requests flow (in a pipelined manner) from the client processors, through the ships in the request graph, to the data servers. Data flows back to the client, through the ships in the data graph, for reads and flows toward the data servers for writes. In a typical read-only scenario, an application would process only portions of the data at a time, thus alternating between sending requests and getting data back. From a theoretical perspective, for the purposes of our algorithm, we treat a ship as if the application’s entire *sequence* of requests arrives at once, and we reason about a ship’s effect on the sequence of requests or on the sequence of data passing through.

In this context, a *sequence* is an ordered collection of elements (either requests or data) written $X = \langle x_1, x_2, \dots, x_k \rangle$. Since a single ship may process and generate multiple sequences, we use the notation $S^n = \{S_1, S_2, \dots, S_n\}$ to describe the set of *sequences* S_1, S_2, \dots, S_n . The processing of sequences of requests or data as it passes through a ship is called a *mapping*, which has the following notation:

- $S \overset{n}{\underset{r}{\rightarrow}} T^m$ denotes the *request mapping* of ship A from n sequences in S^n (input) to m sequences in T^m (output).
- $D \overset{n}{\underset{d}{\rightarrow}} E^m$ denotes ship A ’s *data mapping* (for writes) from n sequences in D^n to m sequences in E^m . For reading, $D \overset{n}{\overset{d}{\leftarrow}} E^m$ denotes the data mapping from m sequences in E^m to n sequences in D^n .

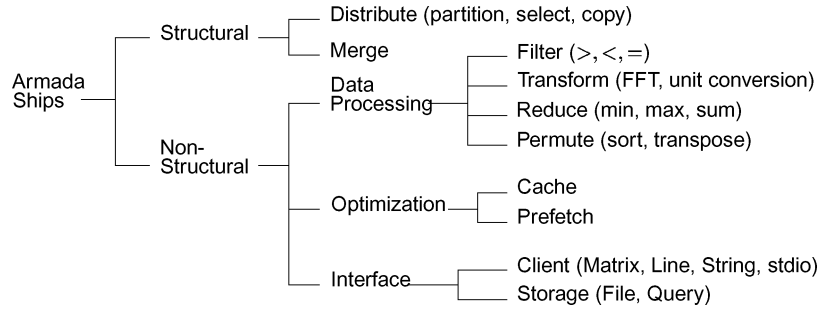


Figure 3. Hierarchy of Armada ships.

3.1.1. Types of ships

Our framework includes a rich set of ship classes (shown in figure 3) divided into two primary categories: structural and non-structural.

Structural ships (illustrated in figure 4) allow one-to-many and many-to-one connections in a graph.

A *Distribute* ship maps the elements of a single request sequence to k sequences. It has request-mapping $R \xrightarrow{1-A} S^k$ and

data-mapping $D \xrightarrow{1-A} E^k$ for writing or $D \xleftarrow{1-A} E^k$ if reading. One could imagine using distribute ships in several ways. For example, a distribute ship could read from a data set partitioned into k pieces. Another example is a *select* ship used to read from distributed replicas of a data set. A select ship could choose a single path to forward all requests, perhaps based on network conditions, or it could partition the requests and send each partition to a separate replica (thus providing parallel access). In yet another instance, a *copy* ship could forward write requests to all output paths so that $R_i = S_i$ and $D_i = E_i$ for all $i = 1, 2, \dots, k$. This ship could be used to update replicated data sets.

A *Merge* ship interleaves k request sequences. It has a request-mapping $R^k \xrightarrow{A} S^1$ and data-mappings $D \xrightarrow{k-A} E^1$ for writing or $D \xleftarrow{k-A} E^1$ for reading. In cases where the ordering of requests is not important, we expect a merge ship to arbitrarily interleave sequences; however, some applications may choose to arrange incoming elements a particular order. For example, an application performing a MERGESORT would take k sorted data sequences as input and output elements

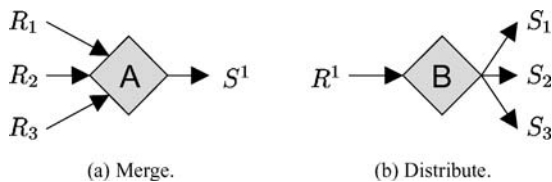


Figure 4. Examples of the two types of structural ships. The merge ship A has request-mapping $R^3 \xrightarrow{A} S^1$. The distribution ship B has request-mapping $R^1 \xrightarrow{B} S^3$.

(based on value) to a single sorted data sequence. Another example is an application receiving write requests for potentially overlapping regions. A POSIX compliant application would have to order requests based on a time stamp embedded in the request [10,11].

Non-structural ships process and generate single sequences of requests and data. A non-structural ship A has request-mapping

$R \xrightarrow{1-A} S^1$ and data-mapping $D \xrightarrow{1-A} E^1$ for writing or $D \xleftarrow{1-A} E^1$ for reading. Figure 3 shows three types of non-structural ships: data-processing ships, optimizing ships, and interface ships.

A *data-processing* ship manipulates data elements, either individually, or in groups, as they pass through the ship. Data-processing ships are likely to be useful for “on-the-fly” preprocessing in scientific applications. Our hierarchy from Figure 3 identifies four types of data-processing ships. A *filter* ship outputs a subsequence of its input; for example, to select interesting observations from a spatial dataset. A *transform* ship changes the content of individual data elements; for example, a Fast Fourier Transform (FFT) ship transforms complex data from time values to frequency values. A *reduction* ship applies a function to a collection of elements and returns a single result; for example, to sum all of the elements. A *permute* ship rearranges the elements in a collection; for example, to sort or transpose a dataset.

Optimization ships improve I/O performance through latency-reduction techniques like caching and prefetching.

Interface ships typically form the right and left end points of an Armada graph. On the left, *client-interface* ships convert an application’s method calls to a set of data requests. We expect library programmers to develop client-interface ships that match the semantics of a particular class of applications, such as computational chemistry applications [19], out-of-core data-parallel programming [9], or a POSIX interface for access by legacy software. On the right, *storage-interface* ships process Armada requests and access low-level data servers to either load or store data based on the request. They are essentially “drivers” for the many available storage systems, for example, a file ship to store a data segment in a UNIX file, or a database ship that “queries” data from a relational database.

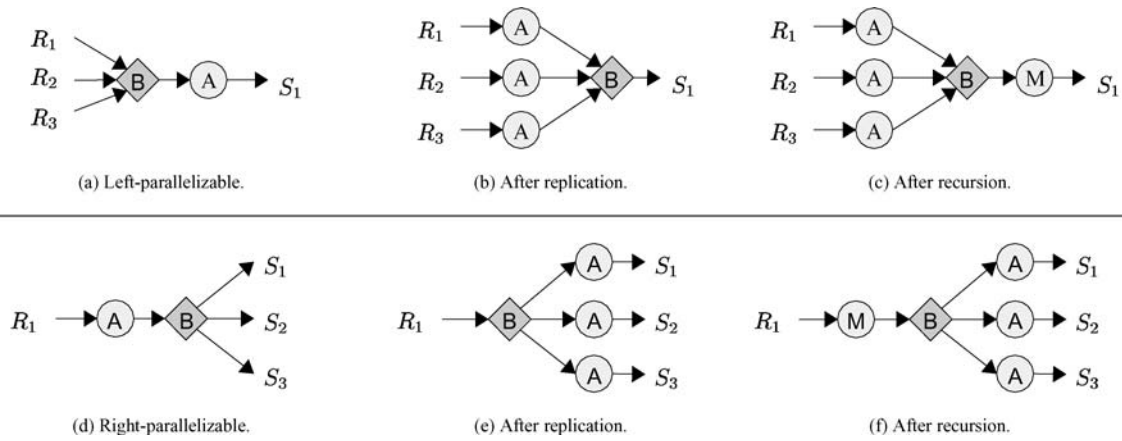


Figure 5. Request-flow for parallelizable ships. Subfigures (b) and (c) show the result of replication and recursion to the left. (e) and (f) show replication and recursion to the right.

3.1.2. Properties of ships

Encoded in a ship’s description are programmer-assigned properties used as *directives* by the restructuring and placement algorithms. These properties provide a simple way for the programmer to describe the capabilities and expected behavior of the ship once deployed on the grid.

Two important properties are *request* and *data equivalence*. A ship that is request-equivalent produces sequences of requests that are *equivalent* to the input. Similarly, a data-equivalent ship produces sequences of data that are equivalent to its input. We declare two sequences S^n and T^m to be *equivalent* (written $S^n \equiv T^m$) if T^m is a permutation of S^n , or if T^m is a set of sequences that partition S^n . For example,

$$\begin{aligned} \{ \langle 1, 2, 3, 4, 5 \rangle \} &\equiv \{ \langle 2, 3, 5, 1, 4 \rangle \} \\ \{ \langle 1, 2, 3, 4, 5 \rangle \} &\equiv \{ \langle 2, 3 \rangle, \langle 1, 4, 5 \rangle \} \\ \{ \langle 1, 2, 3, 4, 5 \rangle \} &\equiv \{ \langle 2, 3 \rangle, \langle 1, 5, 4 \rangle \} \end{aligned}$$

In other words, as long as the elements themselves do not change, order does not matter. We chose to define equivalence in this manner because enforcing a strict ordering involves synchronization that the application may not require. Note also that the equivalence relation is *transitive*, *symmetric*, and *reflexive*.

We expect most structural ships to be both request and data-equivalent, because although they may interleave sequences (as in a merge ship) or partition sequences (as in a distribution ship), the requests or data being operated on do not actually change. For example, a request-equivalent

distribution ship A has request-mapping $S \xrightarrow[r]{1} T^m$, where each T_1, T_2, \dots, T_m is a *subsequence* of S_1 , and a request-equivalent merge ship A has a request-mapping $S \xrightarrow[r]{n} T^1$, where each S_1, S_2, \dots, S_n is a *subsequence* of T_1 .

Another important property describes the expected behavior with respect to data-flow. A ship with the *data-reducer* property reduces the amount of data flowing back

to the client for reads or toward the data servers for writes. For example, a filter is a ship with the data-reducer property. Inversely, a ship with the *data-increaser* property increases the size of the data as if flows through the ship. A ship that caches data is a data increaser for reads and a data reducer for writes.

The *parallelizable* property identifies ships that can transform into a collection of ships that operate on subsequences of requests and data in parallel. For example, the filters in figure 2 are a parallelized version of the filter in figure 1. To have the parallelizable property, the parallelized version must produce output equivalent to the original graph. Ships become parallelized by trading positions with a structural ship (Section 4.1.1 describes when such a “swap” is legal). A ship that is allowed to parallelize toward the clients is *left-parallelizable*. In this case, the parallelized version includes a new ship on every input path of the merge ship. A *right-parallelizable* ship moves to the right of an adjacent distribution ship by placing a new ship on every output path of the distribution ship. There are two types of parallelizable ships (illustrated in figure 5) supported by Armada: *replicable* and *recursive*.

A ship is *replicable* if each parallelized ship is identical to the original. Replicable ships operate on requests and data objects independently. For example, a filter that discards integer data with a negative value, or a distribution ship with a mapping function that operates on individual requests.

A ship with the *recursive* property may “split” into a manager ship and a collection of worker ships that are identical to the original ship. Data-reduction ships that calculate a sum, min, or max are recursive. Merge sort is also a recursive operation.

3.2. Armada graphs

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph. An SP-tree derives from a series-parallel directed acyclic graph (SP-DAG), which is recursively defined as follows [31]:

1. A DAG with a single vertex and no edges is an SP-DAG.
2. If $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are two SP-DAGs, so is the DAG formed by one of the following operations:

- (a) Parallel composition: $G_p = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$
- (b) Series composition: $G_s = \langle V_1 \cup V_2, E_1 \cup E_2 \cup (N_1 \times R_2) \rangle$, where N_1 is the set of sinks of G_1 and R_2 the set of sources of G_2 .

We can represent the composition of an SP-DAG as a *series-parallel tree*, where *s-nodes* represent a series composition, *p-nodes* represent a parallel composition, and *leaves* are the vertices in the original graph.

A *contracted SP-tree* (CSP-tree) is one in which *p*-nodes only have *s*-nodes and leaves as children, and *s*-nodes only have *p*-nodes and leaves as children. It is easy to show that for each SP-tree, there exists a unique CSP-tree, composed by contracting children with the same type as the parent [6].

We chose to use a CSP-tree to describe the composition of an Armada graph because a CSP-tree is syntactically easy to describe (we use XML) and easy to manipulate internally, and it constrains the graph to be an SP-DAG (we show in Section 4.1.1 why this is important). Ships form the vertices of the graph and the leaves of the SP-tree. For example, figure 6(a) shows an annotated version of the graph from figure 1. In figure 6(b), we show the SP-tree representation of the graph.

This section presents, in detail, the ships that make up an Armada graph, the properties that describe the capabilities of the ships, and the graph abstraction used by the Armada

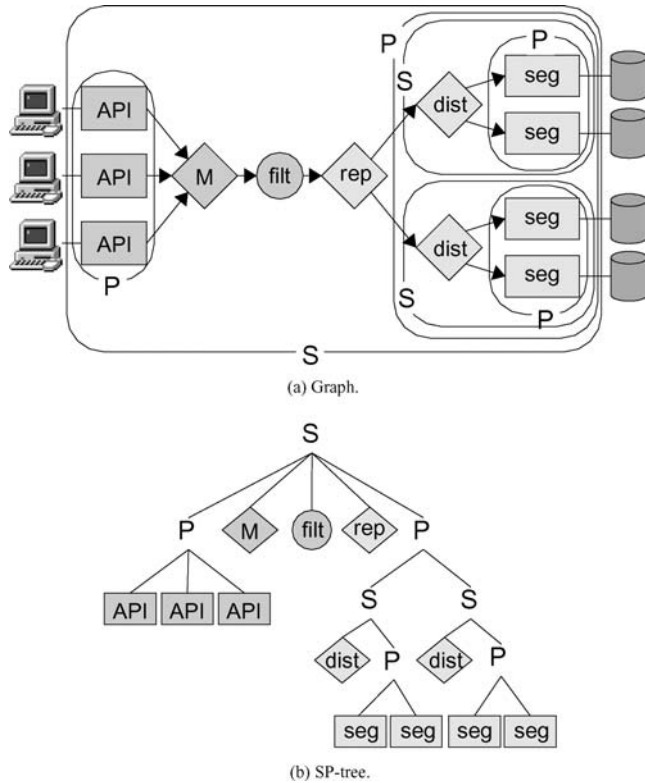


Figure 6. An application graph and its SP-tree representation.

system. With this information, we are ready to describe the algorithm used to restructure an Armada graph.

4. Restructuring an Armada graph

As mentioned in Section 1, the graph resulting from the concatenation of the data-provider's portion and the application's portion may not provide good performance. We can improve performance by restructuring an Armada graph to increase parallelism and to arrange processing ships so as to minimize traffic on the slowest network links. In this section, we describe such an algorithm.

Armada restructures a graph by *swapping* (exchanging the position of) adjacent ships. It increases parallelism by swapping parallelizable ships with structural ships. It reduces network traffic by moving data-reducing ships toward the data-source and data-increasing ships toward the data destination.

The restructuring algorithm manipulates an Armada graph by restructuring its SP-tree. Recall that an SP-tree node is either a series node, parallel node, or leaf node that corresponds to a ship. The process of restructuring an SP-tree requires two tasks: initializing the tree, and recursively traversing the tree swapping series-connected nodes as needed.

The INITIALIZE procedure (Algorithm 1) verifies that root node N is indeed an SP-tree, compresses the tree to form a CSP-tree (if necessary), and initializes internal nodes (series and parallel nodes) to *dirty*. The dirty flag identifies nodes that need to be recursively restructured.

The RESTRUCTURE algorithm (Algorithm 2) traverses the CSP-tree in a depth-first manner, revisiting subtrees when nec-

-
- 1: verify N is an SP-tree
 - 2: compress N (as necessary) to make N a CSP-tree
 - 3: mark all internal nodes of N dirty
-

Algorithm 1. INITIALIZE (N).

-
- 1: **if** (N is clean or a leaf) **then** {base case}
 - 2: return
 - 3: **end if**
 - 4: **if** (N is a parallel node) **then**
 - 5: **for all** $child \in$ children of N **do**
 - 6: RESTRUCTURE($child$)
 - 7: **end for**
 - 8: **else** { N is a series node}
 - 9: $New \leftarrow$ new series node
 - 10: **for all** $child \in$ children of N **do** {from left to right}
 - 11: append $child$ to New
 - 12: SLIDELEFT($child$) {slide $child$ left to proper position}
 - 13: **end for**
 - 14: $N \leftarrow New$ { N is now restructured}
 - 15: **end if**
 - 16: mark N clean
-

Algorithm 2. RESTRUCTURE (N).

```

1:  $A \leftarrow$  node to the left of  $B$  {else null}
2: if ( $B$  is a parallel node) then { $A$  may now be able to slide left}
3:   SLIDELEFT( $A$ )
4:   return
5: end if
6: while ( $A \neq$  null) do {iterate left until we reach the end}
7:   if (COMMUTATIVE( $A, B$ ) and BENEFICIAL( $A, B$ )) then
8:      $L \leftarrow$  node to the left of  $A$  {else null}
9:      $R \leftarrow$  node to the right of  $B$  {else null}
10:    if ( $A$  and  $B$  are non-structural ships) then {case 1}
11:      reverse the order of  $A$  and  $B$  in the series node
12:       $A \leftarrow L$  {iterate left}
13:    else if ( $A$  is right-parallelizable and non-structural,  $B$  is a distribute ship, and  $R$  is a parallel node) then {case 2}
14:      remove  $A$  from the series
15:      PARALLELIZERIGHT( $A, R$ ) { $R$  is a parallel node}
16:      if ( $A$  is recursive) then
17:        create a new manager ship  $M$ 
18:        place  $M$  to the left of  $B$ 
19:         $B \leftarrow M$  {see if  $M$  needs to slide left– it cannot move right}
20:      end if
21:       $A \leftarrow$  node to the left of  $B$  {the old  $A$  has been removed}
22:    else if ( $L$  is a parallel node,  $A$  is a merge ship, and  $B$  is left-parallelizable and non-structural) then {case 3}
23:      remove  $B$  from the series
24:      PARALLELIZELEFT( $B, L$ ) { $L$  is a parallel node}
25:      if ( $B$  is recursive) then
26:        create a new manager ship  $M$ 
27:        place  $M$  to the right of  $A$ 
28:        SLIDERIGHT( $M$ ) {see if  $M$  needs to move right– it cannot move left}
29:      end if
30:      return {everything else is already sorted}
31:    else if ( $L$  is a parallel node,  $A$  is a right-parallelizable merge ship,  $B$  is a left-parallelizable distribute ship, and  $R$  is a parallel node) then {case 4}
32:      remove  $A$ 
33:      remove  $B$ 
34:      PARALLELIZERIGHT( $A, R$ ) { $R$  is a parallel node}
35:      PARALLELIZELEFT( $B, L$ ) { $L$  is a parallel node}
36:      return {everything else is already sorted}
37:    else {not a swappable configuration of nodes}
38:      return {everything else is already sorted}
39:    end if
40:  else
41:    return {not commutative and beneficial}
42:  end if
43: end while

```

Algorithm 3. SLIDELEFT (B).

essary. The base case occurs if N is a leaf or if N is marked clean (meaning it has already been restructured). If N is a parallel node, the algorithm recursively calls RESTRUCTURE on each child of N . For series nodes, RESTRUCTURE individually removes and properly aligns each child of N onto a new series node (labeled *New* in Algorithm 2) in by iteratively “sliding” the child (by swapping adjacent nodes where necessary) from right-to-left using the SLIDELEFT algorithm (Algorithm 3). After properly aligning each child, RESTRUCTURE sets N to point to the new node and marks N clean. The process of restructuring series-connected nodes forms the core of our restructuring algorithm.

4.1. Restructuring series-connected nodes

As mentioned above, the RESTRUCTURE procedure restructures a series node by iteratively “sliding” each

child into the correct position of a previously restructured series node using the SLIDELEFT procedure (Algorithm 3). SLIDELEFT attempts to “swap” the designated node with the node to its left. The algorithm decides whether or not to swap two series-connected nodes (labeled A and B) based on three conditions: the graph formed by swapping A and B must produce an SP-tree, A and B must be *commutative*, and swapping A and B must *benefit* the application. We describe each of these conditions in order.

4.1.1. A swap must produce an SP-tree

Since our algorithm relies heavily on the structure of series-parallel DAGs, our first condition requires that a graph formed by swapping two series-connected ships in a series-parallel DAG is also a series-parallel DAG. Since we use an SP-tree to represent an SP-DAG, we satisfy this condition by only allowing a swap when the configuration of A and B in

Table 1

Possible configurations of series-connected ships in an SP-tree. The cells identify whether swapping is allowed (subject to the commutativity requirement).

Ship A	Ship B		
	non-struct	distrib	distrib- <i>p</i>
non-struct	yes	not SP	yes
merge	not SP	not SP	not SP
<i>p</i> -merge	yes	not SP	yes

the SP-tree matches one of four configurations guaranteed to produce another SP-tree, if *A* and *B* are swapped.

Table 1 lists all possible SP-tree configurations of two series-connected ships and identifies configurations that are

allowed to swap. The rows represent the left side (ship *A*) of a connection and the columns represent the right side (ship *B*). The three types of ships are non-structural, merge, and distribution. Since a merge ship takes multiple inputs it must either follow a parallel node (labeled *p*-merge in Table 1), or it must be the first ship in a series node (labeled *merge*); thus, it cannot be on the right side of a series connection. Similarly, a distribution ship does not appear on the left side because it must either be connected to a parallel node (labeled *distrib-p*), or it must be the last ship in a series node. Here are the four configurations in which a swap is allowed.

1. Two non-structural ships connected in series (figure 7(a)): Exchanging the positions of two non-structural ships does not effect the structure of the SP-tree.
2. A series connection of a right-parallelizable non-structural ship *A*, a distribution ship *B*, and a parallel node

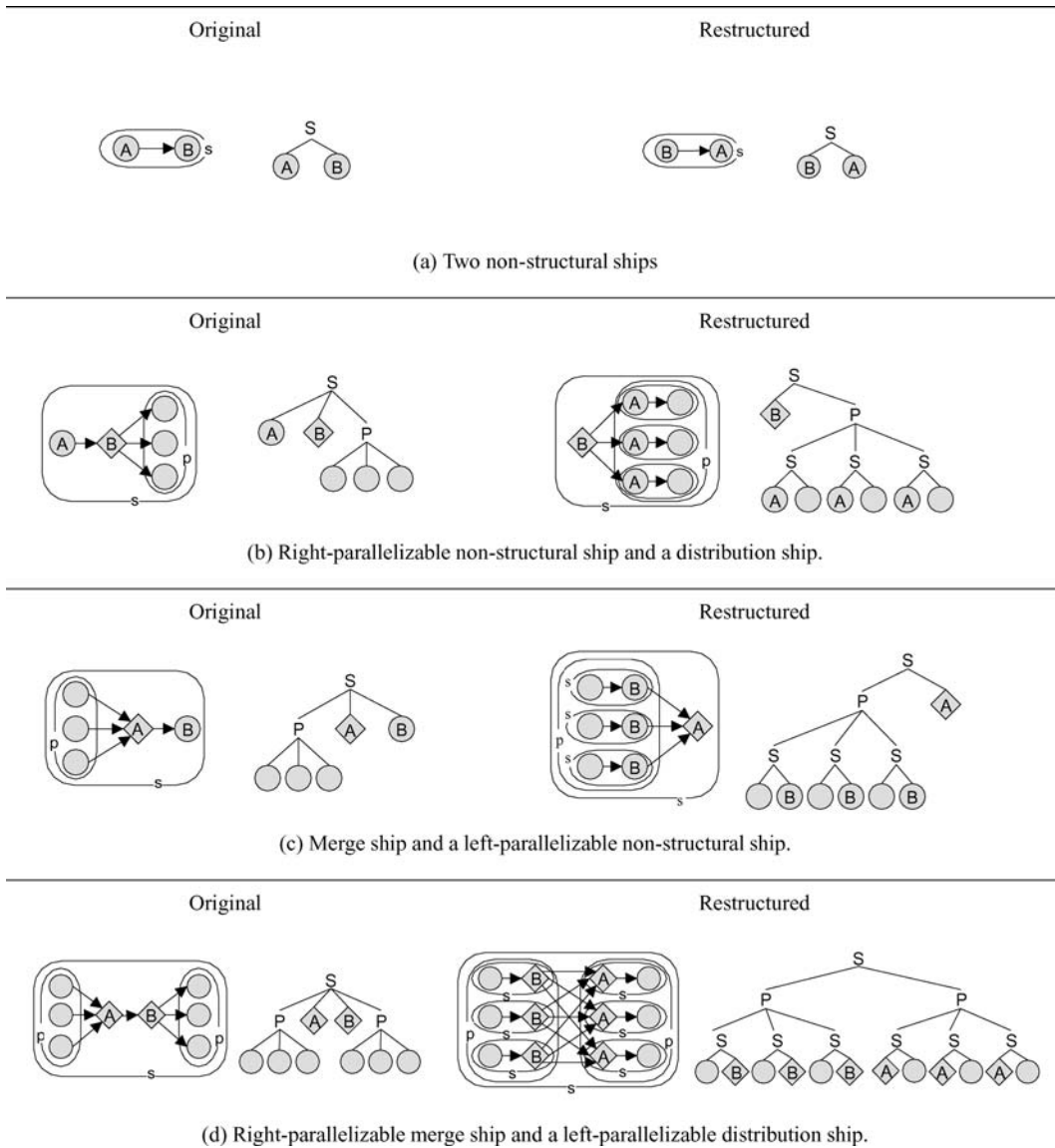


Figure 7. Four cases for swapping neighboring ships.

```

1:  $B \leftarrow$  node to the right of  $A$  {else null}
2: if ( $A$  is a parallel node) then { $B$  may now be able to slide right}
3:   SLIDERIGHT( $B$ )
4:   return
5: end if
6: while ( $A \neq null$  and  $B \neq null$ ) do {iterate right until we reach the end}
7:   if (COMMUTATIVE( $A, B$ ) and BENEFICIAL( $A, B$ )) then
8:      $L \leftarrow$  node to the left of  $A$  {else null}
9:      $R \leftarrow$  node to the right of  $B$  {else null}
10:    if ( $A$  and  $B$  are non-structural ships) then {case 1}
11:      reverse the order of  $A$  and  $B$  in the series node
12:       $B \leftarrow R$  {iterate right}
13:    else if ( $A$  is right-parallelizable and non-structural,  $B$  is a distribute ship, and  $R$  is a parallel node) then {case 2}
14:      remove  $A$  from the series
15:      PARALLELIZERIGHT( $A, R$ ) { $R$  is a parallel node}
16:      if ( $A$  is recursive) then
17:        create a new manager ship  $M$ 
18:        place  $M$  to the left of  $B$ 
19:        SLIDELEFT( $M$ ) {see if  $M$  needs to move left– it cannot move right}
20:      end if
21:      return {everything else is already sorted}
22:    else if ( $L$  is a parallel node,  $A$  is a merge ship, and  $B$  is left-parallelizable and non-structural) then {case 3}
23:      remove  $B$  from the series
24:      PARALLELIZELEFT( $B, L$ ) { $L$  is a parallel node}
25:      if ( $B$  is recursive) then
26:        create a new manager ship  $M$ 
27:        place  $M$  to the right of  $A$ 
28:         $A \leftarrow M$  {see if  $M$  needs to move right– it cannot move left}
29:      end if
30:       $B \leftarrow$  nodes to the right of  $A$  {iterate right}
31:    else if ( $L$  is a parallel node,  $A$  is a right-parallelizable merge ship,  $B$  is a left-parallelizable distribute ship, and  $R$  is a parallel node) then {case 4}
32:      remove  $A$ 
33:      remove  $B$ 
34:      PARALLELIZERIGHT( $A, R$ ) { $R$  is a parallel node}
35:      PARALLELIZELEFT( $B, L$ ) { $L$  is a parallel node}
36:      return {everything else is already sorted}
37:    else {not a swappable configuration of series nodes}
38:      return {everything else is already sorted}
39:    end if
40:  else
41:    return {not commutative and beneficial}
42:  end if
43: end while

```

Algorithm 4. SLIDERIGHT (A).

(Figure 7(b)): We swap A and B by prepending parallelized versions of A to the children of the parallel node. If a child of the parallel node is a leaf, we replace the leaf with a new series node that has A and the leaf as its children.

3. A series connection of a parallel node, a merge ship A , and a left-parallelizable non-structural ship B (Figure 7(c)): We swap A and B by appending parallelized versions of B to each child of the parallel node. If a child of the parallel node is a leaf, we replace the leaf with a new series node that has the leaf followed by A as its children.
4. A series connection of a parallel node, a right-parallelizable merge ship A , a left-parallelizable distribution ship B , and another parallel node (Figure 7(d)): The result is an all-to-all connection between the parallelized

versions of B and the parallelized versions of A . We append each parallelized version of B to the children of the left-most parallel node, and we prepend each parallelized version of A to the children of the right-most parallel node.

The SLIDELEFT and SLIDERIGHT procedures (Algorithms 3 and 4) implement a swap for the four allowed configurations. Lines 11–12 implement case 1, lines 14–21 implement case 2, lines 23–30 implement case 3, and lines 32–36 implement case 4. SLIDELEFT moves a node from right to left, and SLIDERIGHT moves the node from left to right.

The additional helper routines PARALLELIZELEFT and PARALLELIZERIGHT (Algorithms 5 and 6) perform the operations necessary to create parallelized versions of the ships and place them in the correct branch of the SP-tree.

```

1:  $n \leftarrow$  the number of children of  $P$  { $P$  is a parallel node}
2: parallelize  $B$  creating  $B_1, B_2, \dots, B_n$ , each marked as dirty
3: for ( $i = 1 \dots n$ ) do
4:    $C_i \leftarrow i$ 'th child of  $P$ 
5:   if ( $C_i$  is a series node) then
6:     append  $B_i$  to  $C_i$ 
7:     SLIDERIGHT( $B_i$ ) {slide  $B_i$  to the proper position in  $C_i$ }
8:   else { $C_i$  must be a leaf}
9:      $C_i \leftarrow$  a new series node with the original  $C_i$  followed by  $B_i$ 
10:    RESTRUCTURE( $C_i$ ) {restructure the new series node}
11:   end if
12: end for

```

Algorithm 5. PARALLELIZELEFT (P, B).

```

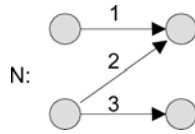
1:  $n \leftarrow$  the number of children of  $P$  { $P$  is a parallel node}
2: parallelize  $A$  creating  $A_1, A_2, \dots, A_n$ 
3: for ( $i = 1 \dots n$ ) do {prepend each  $A$  to the children of  $P$ }
4:    $C_i \leftarrow i$ 'th child of  $P$ 
5:   if ( $C_i$  is a series node) then
6:     prepend  $A_i$  to  $C_i$ 
7:     SLIDERIGHT( $A_i$ ) {slide  $A_i$  to the proper location in  $C_i$ }
8:   else { $C_i$  must be a leaf}
9:      $C_i \leftarrow$  a new series node with  $A_i$  followed by the original  $C_i$ 
10:    RESTRUCTURE( $C_i$ ) {restructure the new series node}
11:   end if
12: end for

```

Algorithm 6. PARALLELIZERIGHT (A, P).

4.1.2. Configurations that do not produce an SP-tree

Table 1 shows five configurations that do not produce series-parallel DAGs after a swap. These configurations occur either when a distribution ship on the right is not series-connected to a parallel node, or when a merge ship on the left does not follow a parallel node. We show that these configurations do not produce series-parallel DAGs after a swap by using the *forbidden subgraph characterization* of Valdes et al. [31]. They proved that a DAG is series-parallel if and only if its transitive closure does not contain the graph N (shown in figure 8) as an *induced subgraph*. An induced

Figure 8. The forbidden subgraph N .

subgraph is obtained by deleting some vertices and all edges incident with deleted vertices.

Figure 9(a) shows an example of the case when a distribution ship B is not series-connected to a parallel node. In this particular example, the left-most ship A is non-structural. Figure 9(b) shows a similar example of the second case, where a merge ship does not follow a parallel node. In each example, bold lines indicate the forbidden subgraph N in the restructured portions.

4.1.3. Commutativity of series-connected ships

The second condition for making a swap requires two connected ships to be *request-commutative* and *data-commutative*. Two series-connected ships are request-commutative if the graph created by swapping the two ships generates a sequence of requests equivalent to the requests produced by the original graph (see Section 3.1.2 for our definition of equivalence). Similarly, two series-connected ships

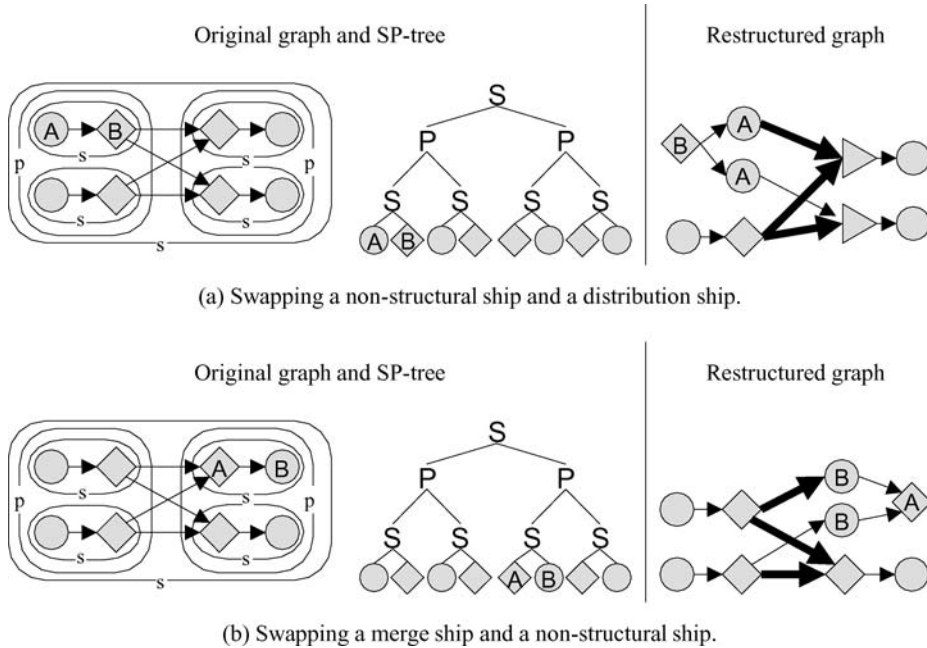


Figure 9. Two configurations of series-connected ships that produce non-series-parallel graphs when swapped. Bold lines highlight the forbidden subgraph N in the restructured portions.

```

1: if ( $A$  or  $B$  is data-equivalent) and ( $A$  or  $B$  is request-equivalent) then
2:   return true
3: else
4:   return false
5: end if

```

Algorithm 7. COMMUTATIVE (A, B).

```

1: if (DIRECTION( $A$ ) > DIRECTION( $B$ )) then
2:   return true
3: else
4:   return false
5: end if

```

Algorithm 8. BENEFICIALSWAP (A, B).

are data-commutative if the graph created by swapping the two ships generates a sequence of data equivalent to the data produced by the original graph. The COMMUTATIVE function (Algorithm 7) uses user-provided ship properties to decide commutativity. It returns *true* if and only if at least one ship is request-equivalent and at least one ship is data-equivalent. Intuitively, COMMUTATIVE returns *true* when the request-mapping function for at least one ship is the *request-identity*, and the data-mapping function for at least one ship is the *data-identity*. We provide a more formal proof of commutativity in [24].

4.1.4. A swap should benefit the application

Although two series-connected ships may be commutative, a swap of the two ships may not improve the performance of the application. The boolean function BENEFICIALSWAP (Algorithm 8) provides this last step in deciding whether to perform a swap of two series-connected ships. The goal is to accurately predict an increase or decrease in the overall application performance resulting from a swap. Our initial heuristic is a greedy approach based on two expectations: increased parallelism leads to improved performance, and moving data-reducing ships closer (in terms of the number of ships) to the data source and data-increasing ships closer to the data destination results in improved performance by reducing the amount of data transferred through slow portions of the network.

BENEFICIALSWAP uses the DIRECTION function (Algorithm 9) to assign a preferred direction to each ship (1 for right, -1 for left, and 0 for no preference). As in our diagrams, we use the convention that storage is on the right and clients are on the left. Based on the expectation that increased parallelism leads to improved performance, the merge ship always prefers to move right and the distribution ship always prefers to move left. If a ship is a *data-reducer* (a property assigned by the programmer) the ship prefers to move toward the data source (right for reads, left for writes). Similarly, ships with the *data-increaser* property prefer to move right for writes and left for reads. The BENEFICIALSWAP function takes as input two connected ships A and B , in which A is currently to the left of B , and

```

1: if ( $A$  is a merge ship) then
2:   return 1 {want to go right to increase parallelism}
3: else if ( $A$  is a distribution ship) then
4:   return  $-1$  {want to go left to increase parallelism}
5: end if
6: if (read-only armada) then
7:   if  $A$  is a data-reducer then
8:     return 1 {want to move closer to the data servers}
9:   else { $A$  is a data-increaser}
10:    return  $-1$  {want to move closer to the clients}
11:   end if
12: else {write-only armada}
13:   if ( $A$  is a data-increaser) then
14:     return 1 {want to move closer to the data servers}
15:   else { $A$  is a data-reducer}
16:     return  $-1$  {want to move closer to the clients}
17:   end if
18: end if
19: return 0

```

Algorithm 9. DIRECTION (A).

returns *true* if the preferred direction of A has a greater value than the preferred direction of B . This can only happen if A wants to go right and B wants to go left, if A wants to go right and B has no preference, or if A has no preference and B wants to go left.

4.2. Algorithm summary

The goal of the restructuring algorithm is to take an SP-tree that represents a distributed graph of application objects, and restructure the tree (adding nodes where necessary) to create a new SP-tree with increased parallelism and an arrangement of data-processing objects that allows a more efficient deployment of processing objects to grid resources. The algorithm works by traversing the tree depth-first and reordering (through a sequence of “swaps”) series-connected nodes in the tree when

- two series-connected nodes are “commutative”,
- swapping two series-connected nodes “benefits” the application, and
- the graph resulting from a swap of two series-connected nodes is an SP-tree.

Note that our solution is not optimal, since there are cases when both ships are commutative and a swap benefits the application (by our definition of beneficial), but swapping the two ships will produce a non-SP-tree. However, we believe these cases to be rare, and the benefits of the keeping the graph series-parallel outweigh the benefit of covering those cases.

4.3. Analysis

The total time to restructure a graph is the sum of the time to initialize the nodes in the graph and the time to recursively traverse and restructure the SP-tree.

Initialization includes verifying that the root node is a SP-tree, compressing it to form a CSP-tree, and initializing internal nodes of the tree to dirty. A paper from Valdes et al. [31] describes a linear-time algorithm for recognizing series-parallel DAGs and trees, so recognizing a SP-tree requires running the Valdes algorithm. To convert the SP-tree to a CSP-tree, we traverse the tree depth-first and combine nodes where the children and the parent are of the same type. We also visit each internal node once to initialize them to *dirty*. Since each task is linear in time, with respect to the number of nodes in the SP-tree, the initialization step has a running time of $O(n)$, where n is the number of nodes in the original SP-tree.

We include a detailed analysis of graph restructuring in [24]. To summarize, our analysis has two parts: first, we calculate the size of the largest possible tree (based on the number of leaves in the initial tree) and the number of operations used by PARALLELIZELEFT and PARALLELIZERIGHT to generate the tree, then we calculate the number of operations (based on the number of nodes in the largest possible tree) used by RESTRUCTURE, SLIDELEFT, and SLIDERIGHT to decide whether or not to swap series-connected nodes. An upper bound on the total cost of restructuring is the sum of the number operations use to generate the largest possible tree, and the number of operations used to traverse and decide which nodes to swap. Finally, we accumulate the results to show that the restructuring algorithm requires $O(v^3)$ operations, where v is the number of leaves in the initial SP-tree, that is, the number of ships in the original graph.

5. Placement of Armada ships

Ships that make up the Armada graph execute on processors near the client, processors near the data, or intermediate network processors. An effective placement (especially for ships that increase or decrease data flow) has a significant impact on the overall performance of the application.

Our approach is to treat placement as a hierarchical graph-partitioning problem. We first partition the graph into administrative domains in an attempt to minimize data transferred between domains. Then we partition ships in each domain to processors provided by domain-level resource managers. Our implementation makes use of a graph-partitioning software library called Chaco [17]. Further details of the placement algorithm are beyond the scope of this document, but can be found in [24].

6. Evaluation

We evaluate the benefits of restructuring by measuring performance of three applications using Armada: a repre-

sentative application, an application for third-party file transfers and data permutation, and a seismic processing application. For each application, we present a brief description followed by performance results. For details on the design and implementation of the various applications, see [24].

6.1. The network testbed

We used the Emulab network testbed [33] at the University of Utah for all results in this section. Emulab provides an emulated network environment by allocating local nodes in a cluster and connecting them through a *Virtual LAN* (VLAN) that restricts traffic to the subnet defined by the user. Each node allocated by the user functions as either an application node, a simulated router, or a traffic generator. To enforce control over bandwidth, latency, and packet loss, Emulab routes controlled network traffic through automatically allocated additional processors that use Dummynet [28] (a network emulator). The Dummynet nodes act as an Ethernet bridge between nodes in the virtual network and are transparent to experimental traffic.

6.2. Representative application

In Section 1, we introduced a representative application (see figure 10(a)) that reads data from a replicated and distributed data set. The portion from the data provider describes the layout of two replicas, each stored as a distributed file. The application prepends a graph to the data provider's portion that includes a filter and three interface ships (one for each client

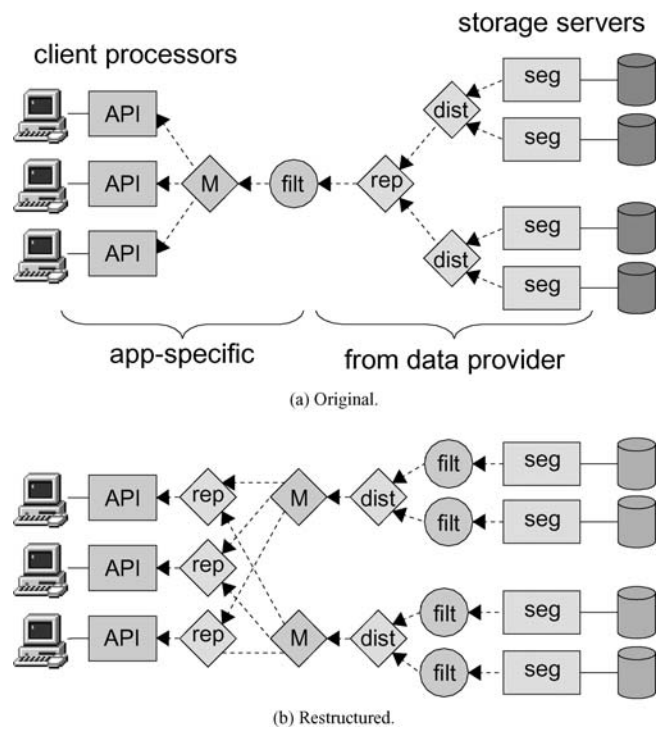


Figure 10. A representative application for reading a replicated and distributed dataset. (a) shows the original graph, (b) shows the restructured graph.

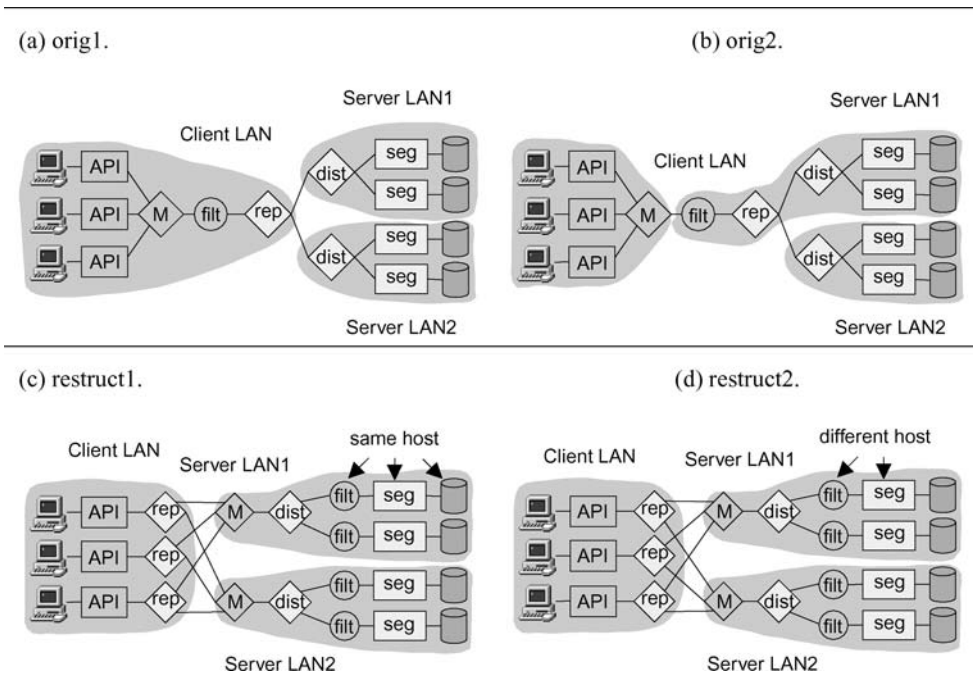


Figure 11. Four configurations of the filtering application.

processor). In Section 4, we again used the representative application to illustrate the process of graph restructuring. We show the resulting restructured graph in Figure 10(b). In this section, we present performance results for an implementation of the representative application.

6.2.1. Experiments

We demonstrate the performance benefits of graph restructuring by measuring the execution time of four different configurations of the representative application. For each configuration, the network topology consists of a wide-area network (WAN) connecting three local-area networks (LANs).

Figure 11 illustrates the four configurations that use manual placements of Armada ships to processors. The “blobs” in the figure encompass ships deployed to a single LAN. The first configuration (labeled *orig1*) uses the original graph and places the filter on the same LAN as the clients. In the second configuration (*orig2*) we use the original graph, but place the filter on the same LAN as one of the remote files. The third (*restruct1*) and fourth (*restruct2*) configurations use the restructured graph. In *restruct1*, each adjacent filter and segment ship share a host. In *restruct2*, we place each filter on a host in the same LAN different than that of its adjacent segment ship. Except for *restruct1*, each ship is installed on a separate processor within its LAN.

The area between the blobs represents the WAN. Each LAN is connected to the WAN by a single router, which has a link to each of the other two LANs. We illustrate these links in figure 12. Regardless of the number of graph edges crossing the WAN, each WAN link has limited capacity. In configuration *orig2*, note that the client LAN uses only one of its WAN

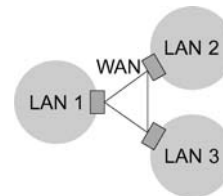


Figure 12. Topology of network illustrating links between LANs.

connections, so the client/server cut has half the bandwidth of the other configurations.

We configured each LAN to have five 850 MHz Intel Pentium III processors connected by a 100 Mbps switched network with a network latency of 0.15 msec. The three LANs were connected by a triangle of three WAN links, each with equal bandwidth, and latency 2.0 msec. We varied the WAN link bandwidths to understand the performance of the application under different network conditions. For *orig1*, *restruct1*, and *restruct2*, we varied the WAN link bandwidths from 1 to 50 Mbps, so that the bandwidth available between client and servers varied from 2 to 100 Mbps. Since *orig2* used only one client/server WAN link, we generously varied its WAN link bandwidths from 2 to 100 Mbps.

6.2.2. Results and discussion

Figures 13 and 14 show timing and throughput measurements for each configuration, as the total client/server WAN bandwidth varied. In this particular application, the filter removed exactly fifty percent of the data. In the throughput plot (bottom), we also show the optimal throughputs for *orig1* (lower solid line) and the others (upper solid line).

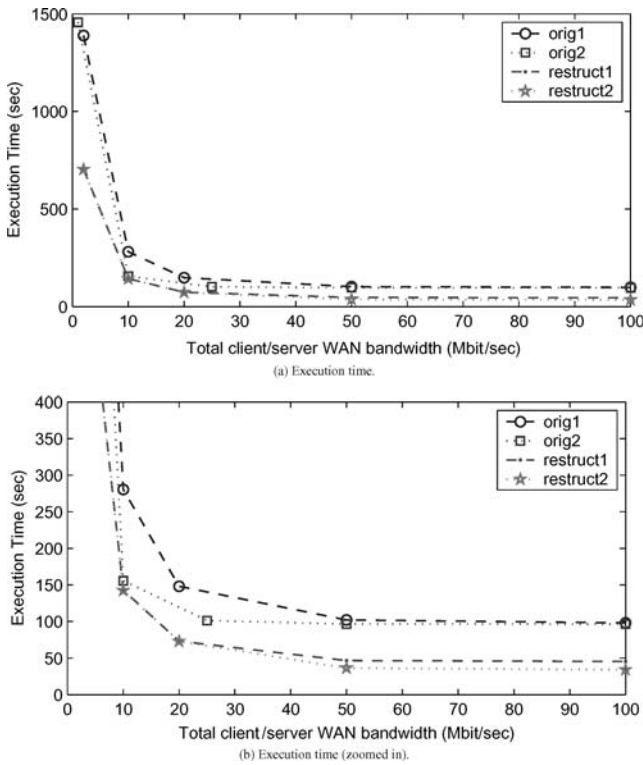


Figure 13. Execution time for the representative application with latency = 2 ms and bandwidth ranging from 0–100 Mbps. (a) shows the full range of the timings, (b) shows the range between 0 and 400 seconds. Each point is the mean of four independent trials.

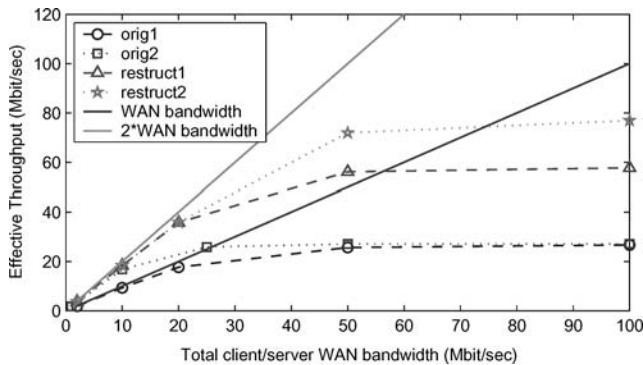


Figure 14. Measured effective throughput for the representative application with latency = 2 ms and bandwidth ranging from 0–100 Mbps. Each point is the mean of four independent trials.

For bandwidths below 30 Mbps, the network was the bottleneck for all configurations. Placement of the filtering code on the server side of the WAN allowed a near-doubling in performance over *orig1*, due to the filter’s halving of the WAN traffic (*orig2* only matched the restructured graphs because it had double the WAN link bandwidth). When the client/server WAN bandwidth was above 30 Mbps, computation associated with Java serialization and the filter code became the bottleneck. The restructured graph’s distribution of the filter across four processors provided a significant performance gain over the original graph.

With the original graph, the *orig2* placement was faster than *orig1* only because its WAN links were twice as fast. When computation was the bottleneck, *orig2* and *orig1* had equivalent performance. With the restructured graph, *restruct2* was equivalent to *restruct1* at low WAN bandwidths, but was faster at high WAN bandwidths because the filter in *restruct1* shares a processor with its adjacent segment ship.

The overhead of computation for this simple application was unexpectedly high. Although restructuring helps by distributing some of the computation, further performance tuning of the Armada system would also improve performance. These improvements, however, would not change the nature of the results. That is, we could perhaps raise the lines on the throughput plots in figure 14, but the shape of the plots would remain the same.

6.3. File transfer and permutation

The second application copies fixed-sized blocks of data from one distributed file to another, compressing data before transfer across a WAN, decompressing data after transfer across a WAN, and permuting data as necessary to match the distribution scheme of the output file. Figure 15 shows the original and restructured Armada graphs for this application.

For block I/O, we use a storage-interface ship (labeled *file* in figure 15) that reads or writes blocks of byte data. We use two different structural ships to describe the layout of the data. For the data source, we use a stripe reader (labeled *srd*). For the data destination, we use a stripe writer (labeled *swr*). Although the structural ships could implement any type of block-distribution scheme, we chose standard block striping because of the simple implementation. In both cases, the structural ships convert global block indices to local block indices for their respective block I/O ships. The compression (*cmp*) and decompression (*dec*) ships are processing ships that use

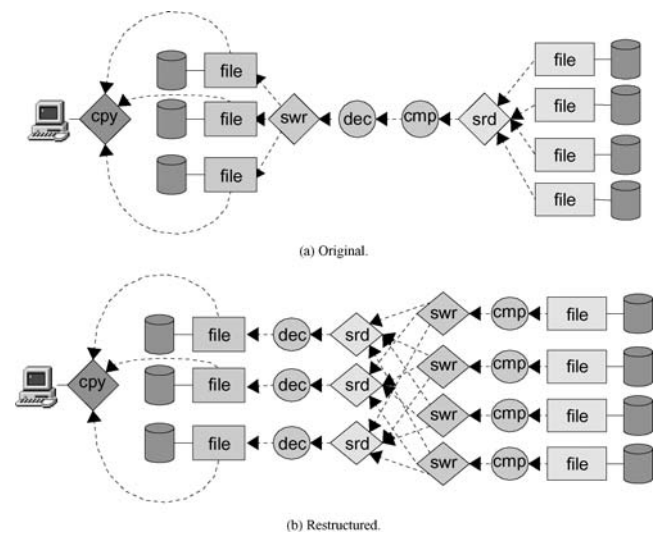


Figure 15. Original and restructured Armada graphs for a distributed file copy application.

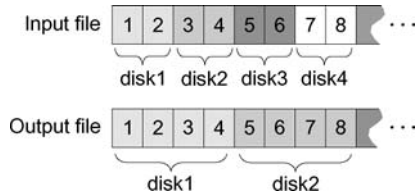


Figure 16. Logical layout of data for the distributed file copy application. The figures show the first eight blocks.

gzip compression on individual data blocks to reduce data transferred across the WAN.

Driving the application is a single client-interface ship (labeled *cpy*) that executes on the client processor. It sends the initial transfer request to the I/O ships on the left, which then forward the requests through the Armada graph to the I/O ships on the right. The *cpy* ship then waits for acknowledgement that the transfer completed from the I/O ships on the left. Since no block-data actually travels through the client, the copy application provides third-party transfer capabilities, an important feature for the management of remote datasets.

6.3.1. Experiments

To investigate the performance of the copy operation, we created experiments for two types of applications. The first application (the baseline test) copies a single 600 Mbyte file (using Armada) across a WAN. The second application copies a distributed file to another distributed file (with a different layout of the data) across a WAN. Figure 16 shows the logical block layout of the first eight blocks for the two distributed files. The input file is a distributed version of the original 600 Mbyte file, striped to six I/O servers with a striping unit of two. The application copies the input to a file distributed to four I/O servers using a striping unit of four.

We configured a virtual network (shown in figure 17) with 30 processors from two LANs. The client executes on a processor in *lan0*. The source file exists on six I/O servers on *lan1*, the destination file exists on four I/O servers on *lan0*. The remaining processors are available as potential hosts for Armada ships.

6.3.2. Results and discussion

Figures 18 and 19 show plots of the execution time and effective throughput for the single-file copy, the original Armada graph, and the restructured Armada graph for distributed file copy. For these plots, we assigned each Armada ship to a separate host. We calculated results of the remote transfers for WAN bandwidths ranging between 1 Mbps and 100 Mbps and network latency fixed at 2 ms.

We expected better results from the restructured graph (especially for low bandwidths), because of the compression; however, after further analysis, we noticed that we were getting less than 10% compression from the compression ships. The reason was the choice of dataset. Since we also ran experiments for the seismic application (Section 6.4) we reused the

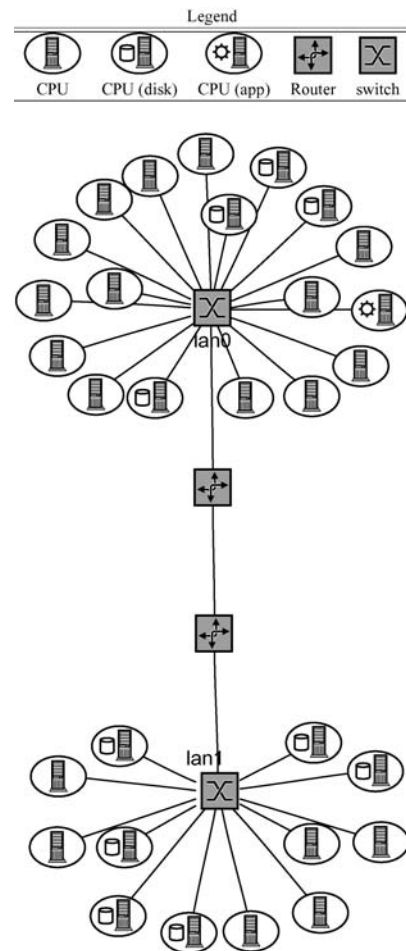


Figure 17. Network topology for the distributed file copy application.

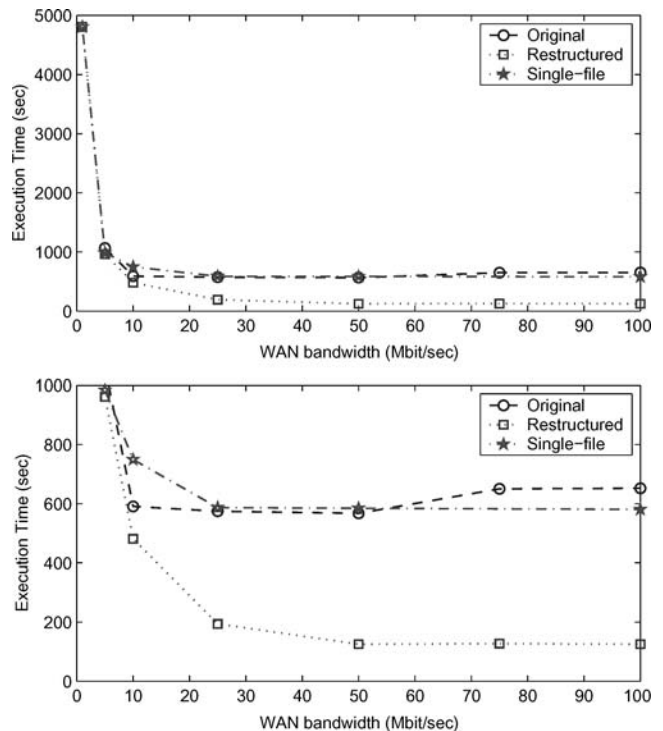


Figure 18. Execution time of file copy applications.

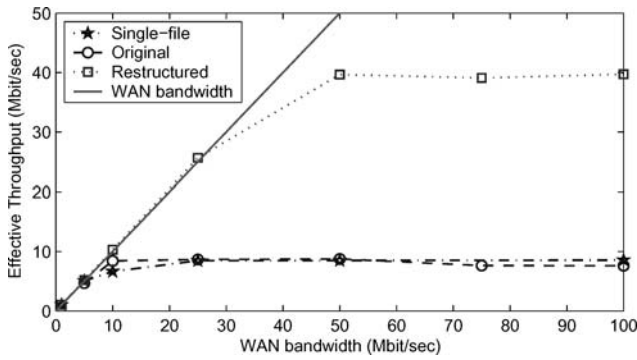


Figure 19. Effective throughput of the file copy applications.

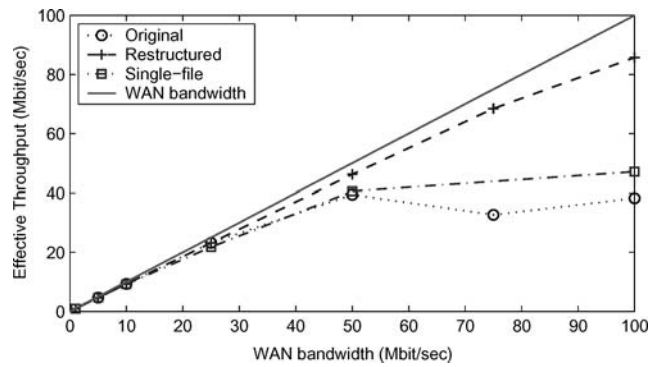


Figure 21. Effective throughput of the distributed file copy application without data compression.

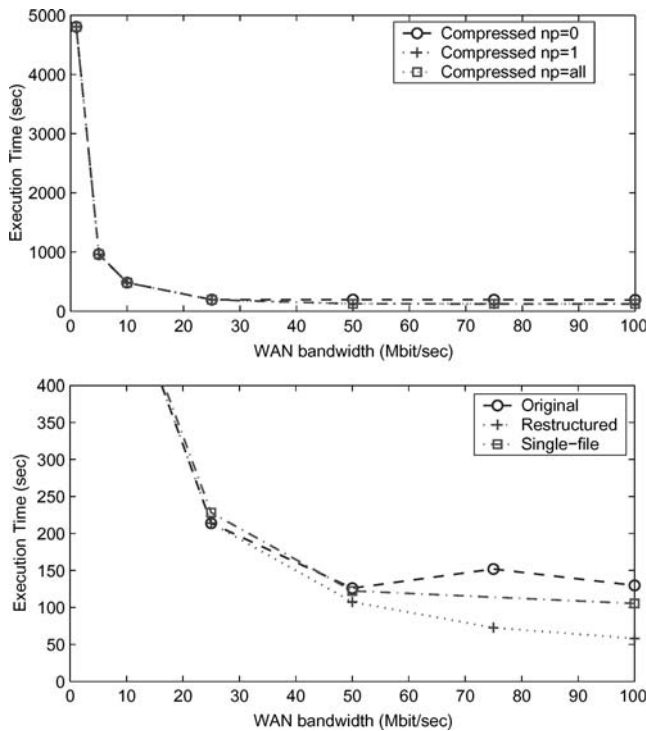


Figure 20. Execution time of file copy applications without data compression.

seismic data for the copy experiments. Unfortunately, seismic data does not compress well with gzip compression.

Despite the poor compression, we see significant improvement of the restructured Armada graph over the original graph. This is primarily due to parallelizing the compression and decompression ships. The original graph becomes compute bound (at around 10 Mbps), since all data converges at the compression ship. The restructured graph becomes compute bound near 30 Mbps and becomes completely limited by the decompression ships near 50 Mbps.

Figures 20 and 21 show plots of a similar application in which we remove the compression and decompression ships. We see that without the overhead of the compression, the restructured graph transfers data near the speed of the network. Even without compression, there is overhead associated with serialization of objects as they pass through the graph. The

restructured graph adds enough parallelism to keep pace with the network, but the single file and the original graph become bound by Java serialization at around 50 Mbps.

6.4. Remote seismic imaging

The goal of seismic imaging is to identify sub-surface geological structures that may contain oil. Seismic imaging is both computationally intensive (often requiring months to process a single data set), and data intensive. A seismic data set can be large, sometimes more than a terabyte in size, and is stored as a collection of files. Each file consists of recorded pressure waves, gathered by a set of receivers distributed across the surface, and generated by a single acoustic source, also located on the surface. The dataset consists of data collected from the same receivers for thousands of different source positions. We refer to the data collected by a single receiver for a single source as a “trace,” and the file associated with a single source position as a “shot file”. Figure 22 shows a 2D slice of a propagating acoustic wave from a single source (demonstrating the acquisition of data), and the calculated image of the SEG/EAGE overthrust model [1].

Post-stack migration [34] is a technique that significantly reduces the amount of processing by “stacking” (i.e., summing) co-located traces from each shot file before the computation phase. Post-stack migration is ideal for demonstrating the effectiveness of Armada: it requires efficient access to large (potentially remote) datasets that require significant preprocessing.

For these experiments, we modified a seismic imaging application called Salvo [25,26] to use an Armada graph to read a seismic dataset. Figure 23 shows the original and restructured Armada graphs for the seismic application. The graph from the data provider consists of replicatable structural ships (labeled *Sdst*) that direct requests toward the datasets, and a storage interface ship (labeled *File*) that reads seismic traces from a Unix file. The application-specific portion consists of two processing ships (*Stk* and *FFT*) to perform the stack and the FFT, a structural ship (*Tdst*) to direct traces to the proper compute node, and a client-interface ship (*API*) that converts method calls to I/O requests.

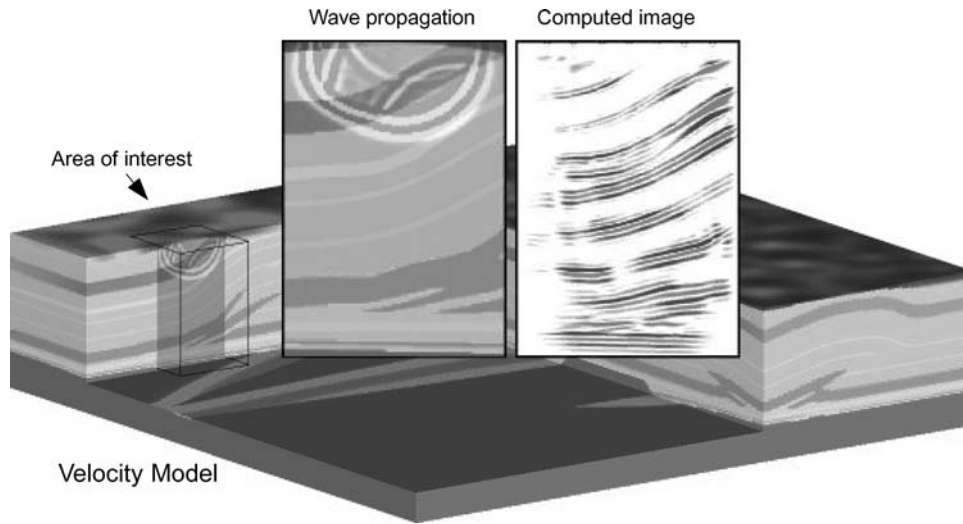


Figure 22. Seismic imaging codes calculate a 3D image of the sub-surface by processing recorded pressure waves collected at the surface. Here we show cross-sections of the SEG/EAGE overthrust model illustrating the propagation of acoustic waves (top-left), and the image computed by a seismic imaging code (top-right).

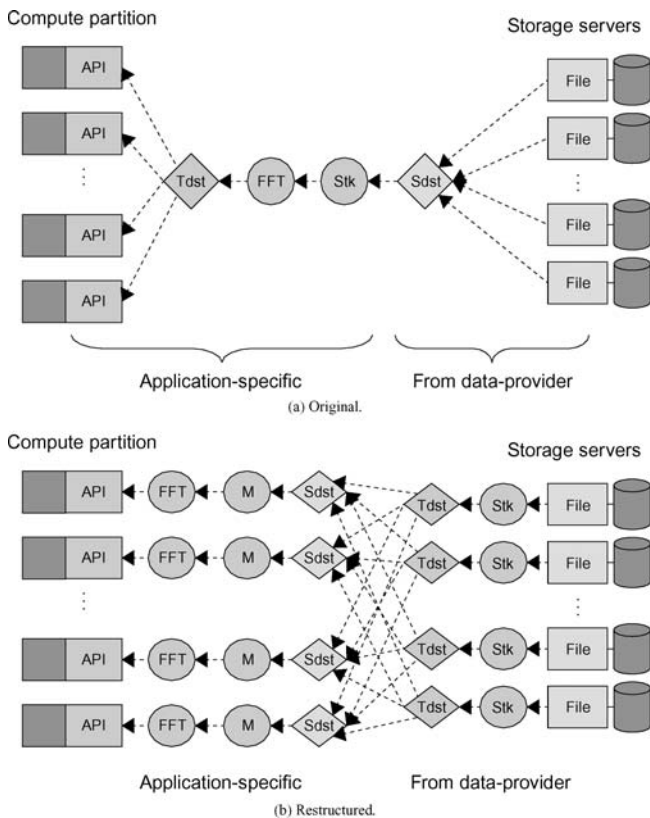


Figure 23. The original and restructured Armada graphs for reading a seismic data set.

6.4.1. Experiments

We ran experiments measuring the performance of our implementation of the trace-input phase of Salvo. Figure 24 shows the Emulab virtual network used for our experiments. The Salvo application used six processors in the first LAN (*lan0*). These processors represent compute nodes that access a remote data set distributed across five I/O servers in *lan1*

and five I/O servers in *lan2*. The remaining processors are available as potential hosts for Armada ships.

The dataset consists of 10 shot files, each with roughly 25 Mbytes of trace data (111×111 traces with 500 samples per trace). Note that our data set is much smaller than a real data set that may contain thousands of shot files. The main reason for choosing a smaller dataset is that we do not have the physical resources to store terabytes of data for our experiments. For this experiment, however, the smaller data set is sufficient to demonstrate the effectiveness of our approach.

Figure 25 shows the original and restructured Armada graphs (illustrating the partitioning to domains assigned by the placement algorithm) for our experiments. In these experiments, we assume there are enough processors so that every Armada ship has its own host. Notice that since we distributed the data set to two domains, we added an additional *Sdst* ship to first direct requests to the appropriate domain-level *Sdst* ship.

6.4.2. Results and discussion

The first set of plots (figures 26 and 27) compare the performance of the original and restructured Armada graphs for a fixed latency (2 ms) as WAN bandwidth increases from 1 Mbps to 100 Mbps.

For the restructured graph, we see a dramatic barrier at 5 Mbps where the application changes from being network bound to being compute bound. Since an FFT is a fairly compute-intensive operation, our first guess placed the bottleneck at the ships performing the FFTs; however, after further analysis, we discovered the bottleneck to be stacking the data on the data-server LANs. The FFT is, in fact, very fast (we used a Java-enabled version of the *fftw* library [14]). The stack operator, has to deserialize and perform a vector sum for each incoming trace, but the FFT only works on the

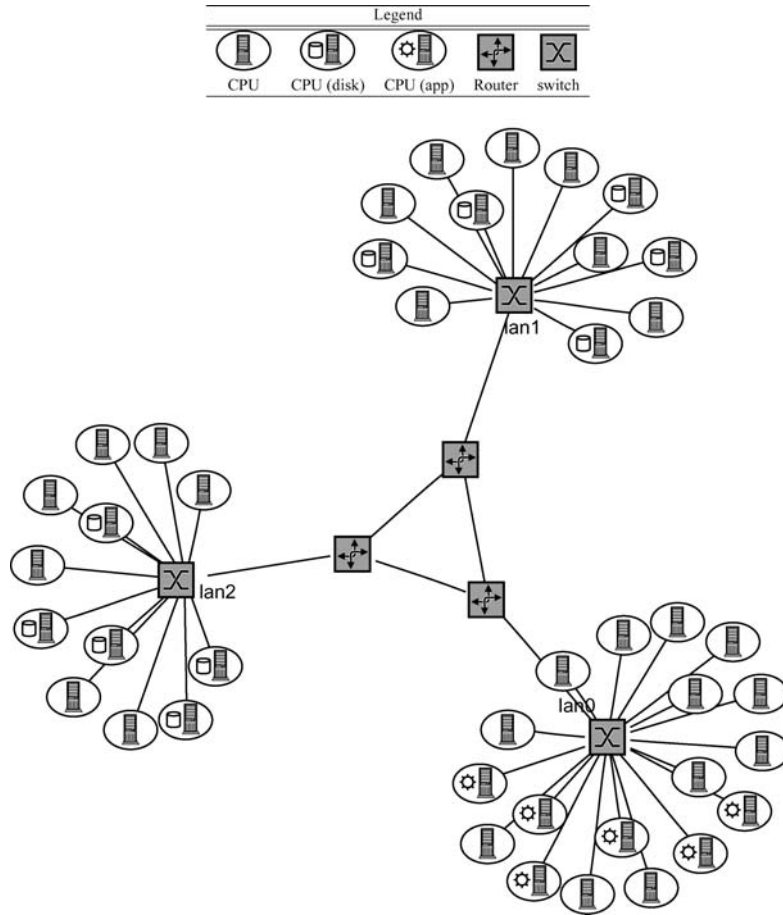


Figure 24. Virtual network topology used to measure the performance of the trace-input phase of Salvo. The Salvo application used six processors on *lan0* to access data distributed to five disks on *lan1* and five disks on *lan2*. The remaining processors on each LAN are available as hosts for Armada ships.

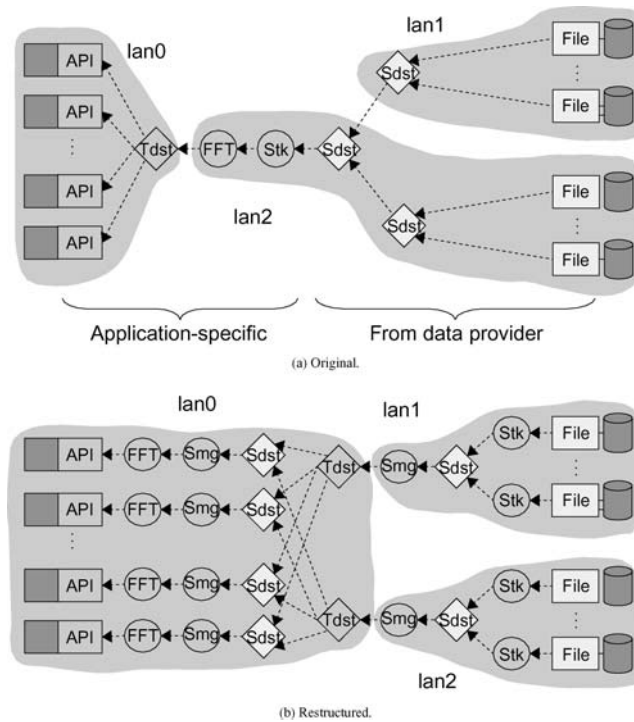


Figure 25. Original and restructured graphs (illustrating LAN placement) for reading seismic trace data.

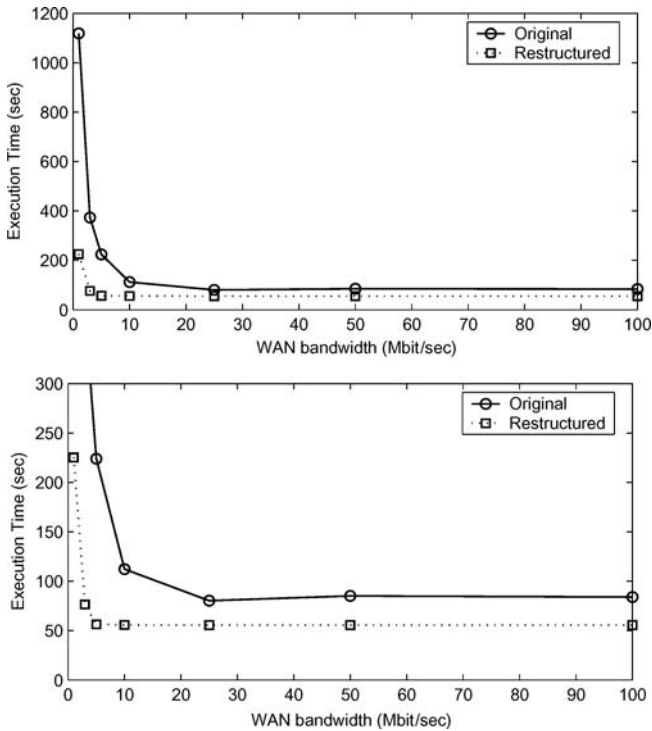


Figure 26. Execution time for the original and restructured Armada graphs for reading seismic trace data. Latency is fixed at 2 ms, bandwidths range from 0–100 Mbps. (a) shows the full range of the timings, (b) shows the range between 0 and 300 seconds. Each data point is the mean of five independent trials.

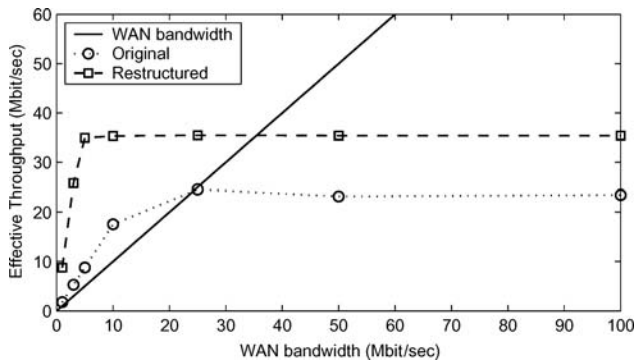


Figure 27. Effective throughput for the original and restructured Armada graphs for reading seismic trace data.

resulting value. Also, the stack ship acts as a synchronization point for the application, since co-located traces from all inputs must arrive before passing the result on to the FFT ship.

For the original graph, the link between the *sdst* ship on *lan1* and the *sdst* ship on *lan2* is the bottleneck up to 25 Mbps. For bandwidths greater than 25 Mbps, the stack operator became the bottleneck.

Figures 28 and 29 plot the execution time and effective throughput of the restructured Armada graph for different latencies. The goal is to demonstrate that pipelining large data transfers effectively negates the effect of latency over a wide-area network. As expected, the different experiments

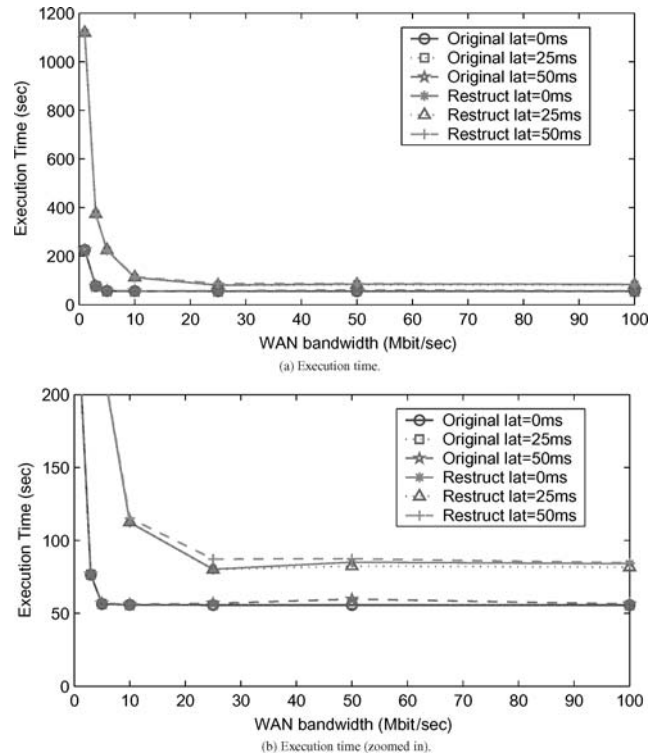


Figure 28. Execution time for seismic application with different WAN latencies.

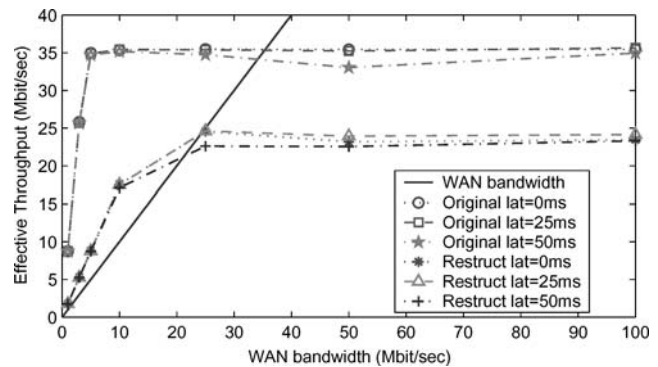


Figure 29. Throughput of seismic application for different WAN latencies.

had nearly identical running times. This feature is a particularly important; although technology limits the amount of available bandwidth, the speed of light makes latency across large geographic distances unavoidable.

7. Conclusion

The trend to develop applications for wide-area computing environments (computational grids) is hindered by several performance-related challenges. Unlike traditional parallel computers, grid applications execute in environments with unavoidable latency, low bandwidth, and unpredictable behavior. In this paper, we describe how to improve data access performance of applications using the Armada I/O framework

by restructuring the Armada graph to increase parallelism and to allow effective placement of data-reduction filters on remote servers. Our approach demonstrates that a flexible design along with careful attention to data-flow performance can lead to efficient I/O for grid applications. Performance results show that Armada does an exceptional job of hiding network latency inherent in grid computing, and that Armada applications perform well in low-bandwidth environments because of an effective placement, but also in high-bandwidth environments because of the end-to-end parallelism provided by restructuring.

Acknowledgments

Special thanks to Jay Lepreau, and the many students and staff who run the Emulab at the University of Utah, for allowing us to use their facility to run our experiments. Thanks also to David Womble and Rob Leland (both from Sandia National Laboratories) for their help with the Chaco graph partitioning software and the Salvo seismic imaging code.

Notes

1. CMU Reading List on Parallel Programming Languages, <http://www-2.cs.cmu.edu/scandal/parallel-lang/reading-list/reading-list.html>
2. A request-identity function returns an “equivalent” sequence of requests, which may not be identical to the original sequence of requests.
3. GNU zip (<http://www.gzip.org>)

References

- [1] F. Aminzadeh, N. Burkhard, L. Nicoletis, F. Rocca and K. Wyatt, SEG/EAGE 3-D modeling project: 2nd update, *The Leading Edge* 13(9) (September 1994).
- [2] Remzi H. Arpaci-Dusseau, Run-time adaptation in River, *ACM Transactions on Computer Systems* 21(1) (2003) 36–86.
- [3] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaf, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick, Cluster I/O with River: Making the fast case common, in: *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, ACM Press, Atlanta, GA, (1999) pp. 10–22.
- [4] James Abello and Jeffrey Scott Vitter, (eds.) *External Memory Algorithms and Visualization*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI (1999).
- [5] Henri E. Bal, A comparative study of five parallel programming languages, *Future Generation Computer Systems* 8 (1992) 121–135.
- [6] Hans Leo Bodlaender and Babette de Fluiter, Parallel algorithms for series parallel graphs, in: *Proc. 4th Eur. Symp. Algorithms*, number 1136 in *Lecture Notes in Computer Science*, Springer-Verlag (1996) pp. 277–289.
- [7] Michael D. Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman and Joel Saltz, DataCutter: Middleware for filtering very large scientific datasets on archival storage systems, in: *Proceedings of the 2000 Mass Storage Systems Conference*, College Park, MD, IEEE Computer Society Press (2000) pp. 119–133.
- [8] Henri E. Bal, Jennifer G. Steiner and Andrew S. Tanenbaum, Programming languages for distributed computing systems, *ACM Computing Surveys* 21(3) (1989) 261–322.
- [9] Alex Colvin and Thomas H. Cormen, ViC*: A compiler for virtual-memory C*, in: *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, IEEE Computer Society Press (1998) pp. 23–33.
- [10] Peter F. Corbett and Dror G. Feitelson, The Vesta parallel file system, *ACM Transactions on Computer Systems* 14(3) (1996) 225–264.
- [11] J. Carretero, F. Pérez, P. de Miguel, F. García and L. Alonso, Prototype POSIX-style parallel file server and report for the CS-2. Technical Report D1.7/1, Universidad Politecnica Madrid, Madrid, Spain, (1993).
- [12] David DeWitt and Jim Gray, Parallel database systems: The future of high-performance database systems, *Communications of the ACM* 35(6) (1992) 85–98.
- [13] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar and M. Muralikrishna, GAMMA—A high performance dataflow database machine, in: *Proceedings of the 12th International Conference on Very Large Data Bases* (1986) pp. 228–237.
- [14] Matteo Frigo and Steven G. Johnson, FFTW: An adaptive software architecture for the FFT, in: *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, vol. 3, IEEE (1998) pp. 1381–1384.
- [15] Ian Foster and Carl Kesselman (eds.) *The Grid: Blueprint for a New Computing Infrastructure* (Morgan Kaufmann Publishers, 1998).
- [16] Benoit A. Gennart, Marc Mazzariol, Vincent Messerli and Roger D. Hersch, Synthesizing parallel imaging applications using the CAP computer-aided parallelization tool, in: *Proceedings of the IS&T/SPIE 10th Annual Symposium on Electronic Imaging, Storage & Retrieval for Image and Video Databases VI*, San Jose, CA, (1998) pp. 446–458.
- [17] Bruce Hendrickson and Robert Leland, The Chaco user’s guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.
- [18] Vincent Messerli, Tools for Parallel I/O and Compute Intensive Applications. PhD thesis, École Polytechnique Fédérale de Lausanne, 1999. Thèse 1915.
- [19] Jarek Nieplocha, Ian Foster and Rick Kendall, ChemIO: High-performance parallel I/O for computational chemistry applications, *The International Journal of High Performance Computing Applications* 12(3) (1998) 345–363.
- [20] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis and Michael Best, File-access characteristics of parallel scientific workloads, *IEEE Transactions on Parallel and Distributed Systems*, 7(10) (1996) 1075–1089.
- [21] Ron Oldfield and David Kotz, Armada: A parallel file system for computational grids, in: *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, IEEE Computer Society Press (2001) pp. 194–201.
- [22] Ron Oldfield and David Kotz, Scientific applications using parallel I/O, in Hai Jin, Toni Cortes, and Rajkumar Buyya (eds.), *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 45, IEEE Computer Society Press and John Wiley & Sons, (2001) pp. 655–666.
- [23] Ron Oldfield and David Kotz, Armada: A parallel I/O framework for computational grids, *Future Generation Computing Systems (FGCS)* 18(4) (2002) 501–523.
- [24] Ron Oldfield. *Efficient I/O for Computational Grid Applications*. PhD thesis, Dept. of Computer Science, Dartmouth College, May 2003. Available as Dartmouth Computer Science Technical Report TR2003-459.

- [25] Curtis Ober, Ron Oldfield, David Womble, L. Romero and Charles Burch, Practical aspects of prestack depth migration with finite differences, in: *Proceedings of the 67th Annual International Meeting of the Society of Exploration Geophysicists*, Dallas Texas, Expanded Abstracts (1997) pp. 1758–1761.
- [26] Ron A. Oldfield, David E. Womble and Curtis C. Ober, Efficient parallel I/O in seismic imaging, *The International Journal of High Performance Computing Applications* 12(3) (1998) 333–344.
- [27] Beth Plale and Karsten Schwan, dQUOB: Managing large data flows by dynamic embedded queries in: *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, Pennsylvania, (2000) pp. 263–270.
- [28] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols, *ACM Computer Communication Review* 27(1) (1997) 31–41.
- [29] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman and J. Saltz, Executing multiple pipelined data analysis operations in the grid, in: *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, Maryland (2002).
- [30] Jeffrey Scott Vitter, External memory algorithms and data structures: dealing with massive data, in Abello and Vitter [4], pages 1–38.
- [31] Jacobo Valdes, Robert E. Tarjan and Eugene L. Lawler, The recognition of series parallel digraphs, *SIAM Journal of Computing* 11(2) (1982) 298–313.
- [32] Rajiv Wickremesinghe, Jeffrey S. Chase and Jeffrey S. Vitter, Distributed computing with load-managed active storage, in: *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, IEEE Computer Society Press (2002) pp. 24–34.
- [33] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb and Abhijeet Joglekar, An integrated experimental environment for distributed systems and networks, in: *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX Association (2002) pp. 255–270.
- [34] Ozdogan Yilmaz, *Seismic Data Processing* (Society of Exploration Geophysics, 1987).



Ron A. Oldfield is a senior member of the technical staff at Sandia National Laboratories in Albuquerque, NM. He received the B.Sc. in computer science from the University of New Mexico in 1993. From 1993 to 1997, he worked in the computational sciences department of Sandia National Laboratories, where he specialized in seismic research and parallel I/O. He was the primary developer for the GONII-SSD (Gas and Oil National Information Infrastructure–Synthetic Seismic Dataset) project and a co-developer for the R&D 100 award winning project “Salvo”, a project to develop a 3D finite-difference prestack-depth migration algorithm for massively parallel architectures. From 1997 to 2003 he attended graduate school at Dartmouth college and received his Ph.D. in June, 2003. In September of 2003, he returned to Sandia to work in the Scalable Computing Systems department. His research interests include parallel and distributed computing, parallel I/O, and mobile computing.

E-mail: raoldfi@sandia.gov



David Kotz is a Professor of Computer Science at Dartmouth College in Hanover NH. After receiving his A.B. in Computer Science and Physics from Dartmouth in 1986, he completed his Ph.D in Computer Science from Duke University in 1991. He returned to Dartmouth to join the faculty in 1991, where he is now Professor of Computer Science, Director of the Center for Mobile Computing, and Executive Director of the Institute for Security Technology Studies. His research interests include context-aware mobile computing, pervasive computing, wireless networks, and intrusion detection. He is a member of the ACM, IEEE Computer Society, and USENIX associations, and of Computer Professionals for Social Responsibility. For more information see <http://www.cs.dartmouth.edu/dfk/>.

E-mail: dfk@cs.dartmouth.edu