ELSEVIER

Future Generation Computer Systems 18 (2002) 501–523

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

Armada: a parallel I/O framework for computational grids☆

Ron Oldfield*, David Kotz[1]

Department of Computer Science, Dartmouth College, 6211 Sudikoff Laboratory, Hanover, NH 03755-3510, USA

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. One of the great challenges for this environment is to provide efficient access to data that is distributed across remote data servers in a grid. In this paper, we describe our solution, a framework we call *armada*. The framework allows applications and dataset providers to flexibly compose graphs of processing modules that describe the distribution, application interfaces, and processing required of the dataset before computation. The armada runtime system then restructures the graph, and places the processing modules at appropriate hosts to reduce network traffic. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Framework; Parallel I/O; Computational grids; Data grids

1. Introduction

A recent trend in high-performance computing is to build “computational grids” that tie together heterogeneous, geographically distributed computers and storage systems. Applications for these environments use high-speed networks to logically assemble resources into a virtual supercomputer. The trend toward grid computing derives from many factors [1]. Among them are the technological advances in networking, computing, and distributed computing software; the increase in the number of *demand-driven* applications—applications that have infrequent requirements for large computing resources (e.g., a medical diagnosis system during surgery or a seismic simulation after an earthquake); the utilization of idle compute resources; and the increased need to share data that was computed or gathered at geographically distributed locations, e.g., weather prediction simulations that use data gathered from many satellites and stored at locations around the world.

A particularly challenging class of grid applications is the class of data-intensive grid applications. A data-intensive grid application typically requires access to large (terabyte–petabyte size) remote datasets, and has significant computational requirements that may require high-performance supercomputers. In addition, the data is often stored in “raw” formats and requires significant preprocessing or filtering before the computation can take place. For example, seismic data, used to extract images of the subsurface, requires a variety of processing steps to filter and transform data before computation [2]. Data-intensive applications also exist in climate modeling [3,4] physics and astronomy [5], biology and chemistry [6,7], visualization [8–10], and many others.

☆ Expanded version of a talk presented at First IEEE/ACM International Symposium on Cluster Computing and the Grid (Brisbane, Australia, May 2001).

* Corresponding author. Tel.: +1-603-646-1639;
fax: +1-603-646-1672;
URL: http://www.cs.dartmouth.edu/∼raoldfi.
E-mail addresses: raoldfi@cs.dartmouth.edu (R. Oldfield),
dfk@cs.dartmouth.edu (D. Kotz).

[1] Tel.: +1-603-646-1439; fax: +1-603-646-1672;
http://www.cs.dartmouth.edu/∼raoldfi.

PII: S0167-739X(01)00076-0

In this paper, we present the armada framework for building I/O-access paths for data-intensive grid applications. Armada's goal is to allow grid applications to access datasets that are distributed across a computational grid, and in particular to allow the application programmer and the dataset provider to design and deploy a flexible network of application-specific and dataset-specific functionality across the grid.

1.1. Flexibility

Computation grids are large and diverse. Data-intensive applications have a wide variety of needs and data-access patterns. It is, therefore, critical to allow applications the flexibility to construct application-specific interfaces, caching policies, data distribution layouts, and so forth.

Studies of data-intensive scientific applications demonstrate the performance benefits of using application-level interfaces that enable advanced parallel-I/O techniques like collective I/O, prefetching, and data sieving [11–14]. A system based on a flexible and powerful low-level interface encourages the development of application-specific libraries that provide the interface and features that benefit applications [15]. For example, tailoring the prefetching and caching policies to match the application's access patterns can reduce latency and avoid unnecessary data requests [16–18], and matching data-distribution policies to the application's access patterns can optimize parallel access to distributed disks [19,20].

1.2. Remote processing of application code

A typical data-intensive grid application consists of a set of *client* processes and a set of remote *data servers*. The traditional approach is for data servers to run system software, perhaps the server side of a parallel file system, and for client processors to run all application-specific code. Even Galley [15], which encourages layers of libraries that provide application-specific functionality, contains all the application code on the client processors. Since the network is typically the bottleneck of data-intensive grid applications, the placement of data-processing functionality is critical. A key goal of armada is to allow application code to execute on or near the data servers, when that helps to reduce network traffic.

There are many good reasons to allow application-specific functionality to run on processors other than the clients. Application-specific data-distribution patterns, perhaps to support disk-directed I/O [21], need to execute on the data servers. If the distribution of data across clients or across disks is dependent on the value of the data, moving that function to the data server can halve network traffic [22]. Processors near the data servers can filter data in an application-specific way, passing only the necessary data on to the clients, saving network bandwidth and client memory [10,22–24]. Processors near the data servers can exchange blocks without passing the data through clients, e.g., to rearrange blocks between disks during a copy or permutation operation. Format conversion, compression, and decompression are also possible. In short, there are many ways to optimize memory and disk activity, reduce network latency, and reduce disk and network traffic, by moving application-specific code closer to the data servers.

1.3. Overview of the armada framework

Using the armada framework, grid applications access remote datasets by sending data requests through a graph of distributed application objects. The graph is called an *armada* and the objects are called *ships*. We expect most applications to access data through existing armadas constructed by a dataset provider; however, it is also possible for the application to extend existing armadas with application-specific functionality or to construct entire armadas from scratch. The armada encodes the programmer's interface, data layout, caching and prefetching policies, interfaces to heterogeneous data servers, and most other functionality provided by an I/O system. The application code sees an armada as an object that provides access to a specific type of data through a high-level interface.

Fig. 1 illustrates a parallel application using an armada to access a distributed dataset. This armada includes an application-interface ship, a preprocessing ship "Op" that manipulates the data, a distribution ship that defines how the data is distributed among data segments, and low-level storage ships that provide access to the data segments stored on data servers.

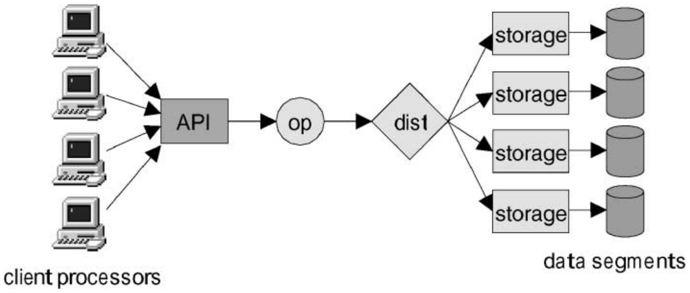


Fig. 1. An armada for a parallel application accessing a distributed dataset. The disk-shaped "data segments" may be as simple as a file in a UNIX system, or as complex as a database.

The armada's clients call methods of the API ship (perhaps through a collective interface). This ship generates *data requests* that flow through the graph toward the storage. Along the way, the ships may transform or split the requests. Each storage ship transfers data either by reading from the data server and pushing the data back through the graph, or by pulling data from the clients through the graph and then writing to the data server. As we describe below, the armada system optimizes this graph, transports data over a subset of this control-flow graph, replicates ships to provide parallel data paths, and carefully places ships onto the grid to limit network traffic.

Notice that the armada system is not a parallel file system, nor does the system itself store any data. The set of *data segments* that make up a *dataset* are each stored in conventional data servers, as files, as databases, or the like. In a computational grid it is important to be able to work with an existing, heterogeneous set of data servers, and (often) with existing datasets stored by those servers in their native formats. Implementations of the armada system provide a variety of *storage ships* that integrate existing resources into an armada.

In the remainder of this paper, we discuss related projects and the design of the armada framework. In the next section, we present related work. In Section 3, we describe the process of constructing an armada, including the design of ships, the arrangement of ships in a graph, and the deployment and management of the armada. In Sections 4 and 5, we describe the implementation of a prototype system and the initial performance results of the prototype. We summarize in Section 6.

2. Related work

Various groups within the research community as well as the commercial sector are investigating issues related to I/O for computational grids. Here, we discuss the systems most related to armada. In particular we discuss I/O systems designed to be flexible, systems that optimize data paths to improve I/O performance, and systems that support remote execution of application code.

2.1. Flexible I/O systems

Much of the design of the armada system is inspired by stackable file systems [25–29]. Stackable file systems provide flexibility by representing files as a collection of application-specific "stackable" building blocks. This flexibility enables the file system to provide only the necessary functionality to suit the application, allowing good performance for a wide variety of applications. Two of the most recent stackable file systems are the hurricane file system (HFS) and wrapFS.

HFS [28] is a stackable file system designed for tightly connected shared-memory machines. They showed that for a several file access patterns, HFS can provide the full I/O bandwidth of parallel disks to parallel applications.

WrapFS [29] is a file system template in which kernel-level modules, with a common vnode interface, are stacked to form complex file systems. They demonstrate flexibility by implementing four separate file systems using wrapFS, including one that hashes files into smaller directories for fast lookups and one that automatically encrypts file data and file names. In a later paper [30], they describe FiST, a language used to describe a stackable file system. The FiST compiler generates the necessary vnode modules used by the system.

Another interesting system that follows the stackable file system philosophy is the parallel storage-and-processing server (PS^2) [31], from École Polytechnique Fédérale de Lausanne. PS^2 is a data-flow system that uses the computer-aided parallelization tool (CAP) [32] to express the parallel behavior of the I/O intensive applications at a high-level. 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, the actors provide application-specific data distribution, prefetching, or filtering.

The armada framework extends the stackable file system concept to a grid environment. A data provider, using armada, can describe complex arrangements and distributions of data with a graph of mobile objects. The application can layer other objects, perhaps to cache, prefetch, or provide a matching interface, on top of the graph constructed by the data provider.

2.2. Systems that optimize data paths to improve I/O

One goal of the armada system is to improve I/O performance by restructuring the data-flow paths between the storage server and the clients. The dynamic QUery OBject (dQUOB) system [33] also has this goal. dQUOB is a runtime system for managing and optimizing large data streams. Users request remote data with SQL-like queries, and computations are performed on the data stream between remote data servers to the client. The computation is encapsulated in a piece of compiled code called a "quoblet". The quoblet is embedded into the data streams in arbitrary points (client, data server, or intermediate points) and are primarily used to filter and/or transform the data. The dQUOB system applies several optimizations to the data stream to improve performance. 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.

Our goal is to generalize this approach. The armada framework is flexible enough to allow graphs of generic ships (not just filters using an SQL interface) to be restructured to reduce network traffic. In some cases, we may dynamically replicate ships to distribute network and processing load across the network.

2.3. Systems with support for remote execution of application code

In addition to dQUOB, there are several software systems that support processing near the data server to either reduce network traffic or distribute computational load. Two such systems from the University of Maryland are MOCHA (middleware based on code shipping architecture) [34] and DataCutter [35]. Each provide capability to move filtering code close to the data, but the decision about where to place the code is made by the application. In situations where many applications compete for the server's resources, placing the filtering code on the server may actually slow performance rather than improve it. In such cases, the application may not be able to make the correct choice about where to place the filtering code.

Abacus [36] and Coign [37] are systems that automate placement of functions based on communication profiling. Coign uses historical profiling data while Abacus profiles during execution to adapt placement decisions to runtime conditions. While our current prototype places objects manually, our goal is to use an approach similar to that of Abacus and possibly extend their approach by deciding placement based on communication, CPU, and memory requirements of the various components.

3. Building an armada

There are several steps to build an armada of ships. The application programmer selects ships from a library of ship classes, or writes her own ship classes. She then writes a "blueprint" to describe the armada. When the application runs, it presents the blueprint to the armada system, which validates the blueprint and deploys the ships to hosts in the grid. The armada runtime system monitors and manages the armada throughout the execution, potentially restructuring and re-deploying the armada to improve

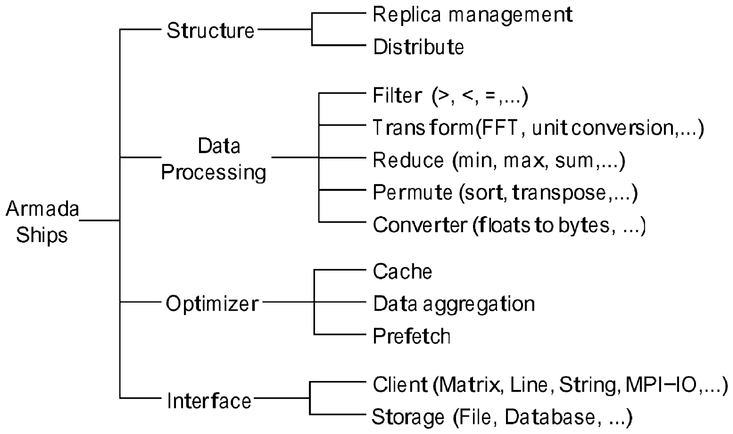

Armada Ships
Structure
Replica management
Distribute
Data Processing
Filter (>, <, =,...)
Transform(FFT, unit conversion,...)
Reduce (min, max, sum,...)
Permute (sort, transpose,...)
Converter (floats to bytes, ...)
Optimizer
Cache
Data aggregation
Prefetch
Interface
Client (Matrix, Line, String, MPI-IO,...)
Storage (File, Database, ...)

Fig. 2. Hierarchy of armada ships.

network performance. We consider each of these steps in turn.

3.1. Ships

The task of the ship builder is to decompose the desired functionality into ships. The class hierarchy in Fig. 2 illustrates the ship classes in a typical armada implementation.

Structural ships manage distributed datasets, through distribution or replication.

Distribution ships manage datasets composed of many data segments. To clients, the dataset is a single large unit, and the distribution ship maps between the dataset and the data in the segments. When writing, the ship decides to which segment(s) the data should be written. When reading, the ship identifies the correct segment(s) to read the desired data. The distribution may be dependent only on the “position” of the data in the dataset, such as a ship that stripes blocks of a dataset across data segments. Or, the distribution may depend on the value of the data, using a hash function or a key to map the request into the appropriate segments.

Replica ships manage replicated datasets. The replica ship decides to which replica to forward the data request, depending on current load and capacity conditions in the grid.

Developers can nest structural ships on top on one another, as in Fig. 3. This application uses both a replica-management ship (labeled D1 in the figure) and two striping distribution ships (labeled D2) to read data from a replicated and distributed dataset. When a request arrives a the replica management ship, it selects one of the two files from which to read the data and forwards the request to the corresponding distribution ship. The distribution ship then forwards to request to the proper storage server based on a simple striping algorithm.

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 Fig. 2 identifies five types of data-processing ships. A *filter* ship outputs a subset of its input elements; e.g., to select observations from a given region of a spatial dataset. A *transform* ship changes the content of individual data elements; e.g., 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; e.g., to sum all of the elements. A *permute* ship rearranges the elements in a collection; e.g., to sort or transpose a dataset. Finally, *conversion ships* cast one data type to another to allow two ships that expect different types to connect. For example, a library may contain a generic structural ship that distributes fixed-sized blocks of byte data to a set of storage ships. The application, however, wishes to view the data as an array of floating-point values. This requires a type-conversion ship between

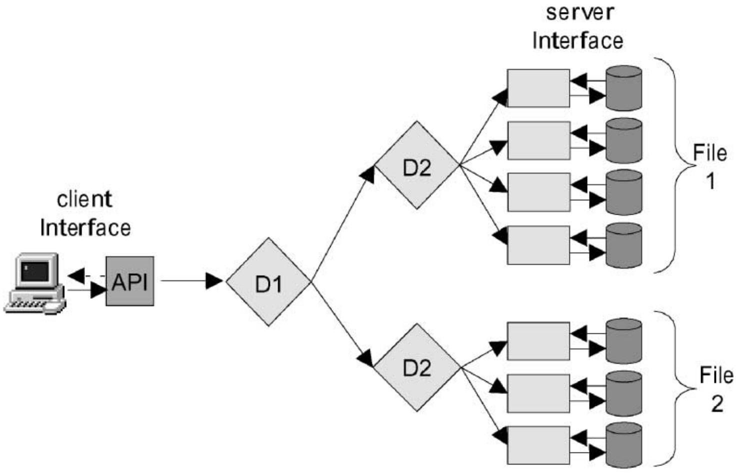


Fig. 3. A graph for an application that uses nested distribution ships to access a replicated dataset.

the client application and the block-distribution ship. The conversion ship converts both requests and data to the appropriate type, e.g., converting a block of bytes into a chunk of floating-point elements in the array.

Optimization ships improve I/O performance through techniques like caching, prefetching, and data aggregation. Data aggregation, near the client nodes for writing and near the I/O nodes for reading, increases the size of network transfers and thus reduces the number of requests that travel through the network.

There are two kinds of *interface ships*: on the client side, to present the application with a convenient interface to its data, and on the server side, to link an armada to specific storage resources (such as files and databases) in the grid.

Client-interface ships convert an application’s method calls to a set of data requests into the rest of the armada ships. We expect library programmers to develop client-interface ships that match the semantics of a particular class of applications, such as computational chemistry applications [14]. Other libraries include interfaces that support collective I/O and data sieving [38]. Others may support out-of-core data-parallel programming [39]. Still others may want to provide a conventional Unix interface to allow legacy software to transparently access grid datasets.

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, e.g., a file ship to store a data segment in a UNIX file, or a database ship to store a data segment in a relational database.

Interface ships exist as the endpoints of an armada. Because they provide direct access to either the application or the storage device, they are not mobile. Throughout the life of the application, they remain on the same host.

3.2. Blueprints

The second stage of building an armada involves arranging the application’s choice of ships into a graph. The meta-data used to describe the graph is called a *blueprint*. The blueprint specifies the interconnection of the ships, and information about each ship (location of its implementation, hints about placement of the ship, etc.).

We constrain graph descriptions to series–parallel form; in such a graph, a node represents a ship (base case) or a subgraph. Subgraphs take two forms: a sequential subgraph (shown in Fig. 4a) that represents a series of connected nodes and a parallel subgraph (Fig. 4b) that consists of a set of simultaneous nodes, connected to a source on the left and a sink on the right. By expressing the arrangement of ships in series–parallel form, we can describe the graph with a directed series–parallel tree (“SP tree”).

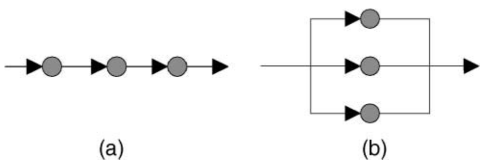


Fig. 4. The types of subgraphs in a series–parallel digraph. (a) This figure shows a sequential subgraph and (b) a parallel subgraph.

Fig. 5 illustrates this conversion. Such a tree is syntactically easy to describe (we use XML) and easy to manipulate internally.

Unlike a file system, the armada system does not provide specific storage or a namespace for blueprints. Blueprints can be stored on conventional file systems, web servers, databases, or any other location accessible to the application; one typical approach would be to store blueprints in files on a Unix workstation.

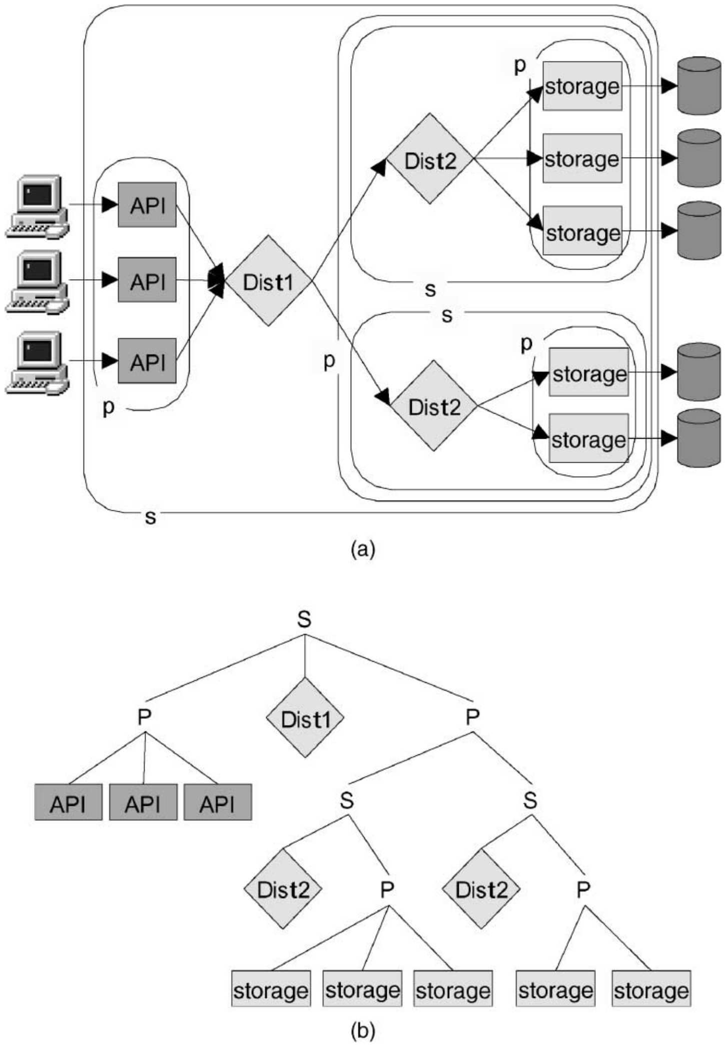


Fig. 5. (a) This figure shows the armada blueprint broken into series–parallel vertices and (b) the series–parallel tree representation of the blueprint.

3.3. Deployment

An armada's ship objects do not exist until the blueprint is used to deploy the armada. An application obtains or creates a blueprint for the desired dataset, optionally extends the blueprint to add ships to the client side of the graph, and submits it to the armada runtime system. The runtime system validates the blueprint and then instantiates a *ship manager* object to deploy, monitor, and manage all ships in the application. The ship manager decides onto which hosts to place the ships, based on the graph, blueprint hints about the ships, and information about current network load and topology. The primary goal is to reduce network traffic. The ship manager also monitors the hosts' resource consumption and periodically uses that information to optimize the graph if necessary and revise the ships' placement. If the ship manager has access to historical data about the application, it may choose to optimize the graph before the initial placement as well. The placement of ships will be based on the results of an analytic cost/benefit model similar to that employed by the Abacus system [36].

The focus of this paper is the armada framework. We leave the details of our placement algorithms for a later paper.

3.4. Control-flow and data-flow

The application accesses data by sending requests from the client to the data servers along the *control-flow graph*—a graph defined by the blueprint and constructed during deployment. The requests sent along the control-flow graph carry with them a reference to the ship that generated the request. When another ship services that request, the result is sent back to the requesting ship along an edge of the *data-flow graph*. Unlike the control-flow graph, the ships that service requests define edges of the data-flow graph at runtime. Write requests are special because they carry an additional reference to the ship that contains the data. A ship servicing a write request collects the data from the source by sending a new read request directly to the source, along an edge of the data-flow graph. The result (containing the data) is then returned along the same path.

Fig. 6 shows the control-flow graph and data-flow graph for an application with access to a federated dataset. A blueprint from the data provider describes a dataset spanning two sites, with the data at each site distributed, for parallel access. The application extends the blueprint with a processing ship "Op", and a collective client-interface. The solid lines of the graph represent the control-flow graph, defined by the blueprint. Dashed lines represent the data-flow graph, established at runtime.

Data has to flow through ships that manipulate the data, but need not flow through most structural ships. The distribution ships in Fig. 6, e.g., only forward requests and do not process data. This approach allows armada to define complex structures but to avoid touching and transporting data more than necessary.

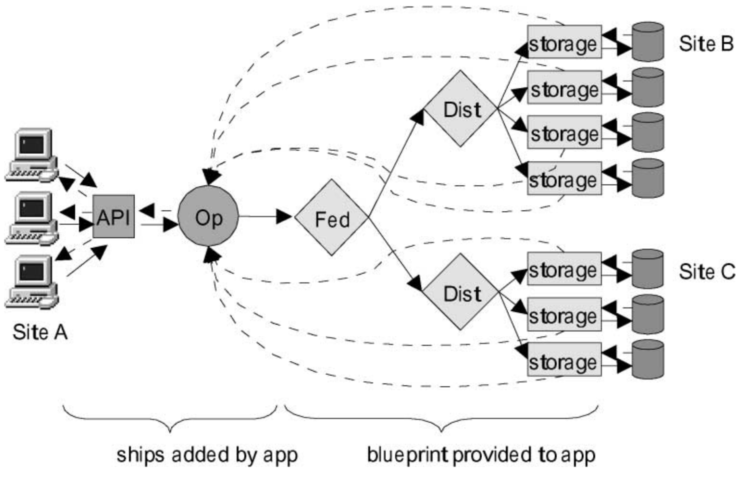


Fig. 6. An armada for a parallel application at site A using a federated dataset located at sites B and C.

3.5. Blueprint optimization

A significant contribution of our work is the optimization of blueprints to reduce the load on the network. In a typical situation, there may be a slow or congested network connecting the site hosting the application and the site hosting data segments. In an armada used for reading a dataset, e.g., moving data-processing ships that reduce the data-flow closer (in terms of network connectivity) to the data can dramatically reduce the amount of data that travels through the "slow" portion of the network.

Consider, again, the application in Fig. 6. Assume that the application-specific processing ship "Op" is a data-reducing filter, and consider the task of choosing a host for that filter. If we place the filter in site A, we do not reduce inter-site traffic. If we place the filter in site B or site C, we do not gain much if the connectivity between sites B and C is slow. If, however, we replace the single filter with an equivalent set of filter ships that

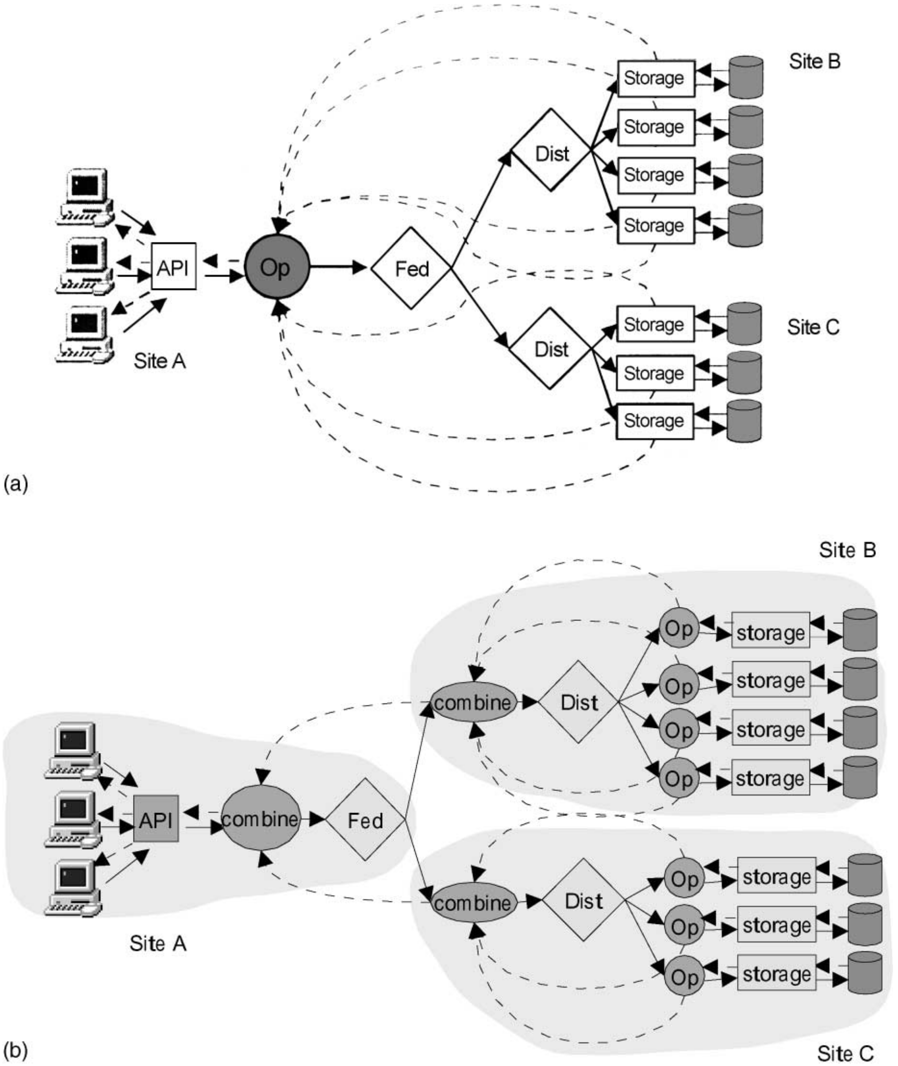


Fig. 7. Original (a) and optimized (b) armadas for an application spanning three domains.

can be pushed past the distribution ships closer to the data servers, plus a merge ship to combine the partial results, we can reduce network traffic by performing a large portion of the filtering near the storage servers. Fig. 7 illustrates the placement for the original and modified graphs.

There are two primary requirements that allow a data-processing ship to be defined in this way: the operator used by the data-processing ship must be recursive and commutative. A processing ship is recursive if it can be defined as a tree of ships, where nodes of the tree combine results from the leaves, which consist of the original processing ship operating on segments of the data. A processing ship is commutative if the order of operations on the data is independent. This allows the operator to move past distribution ships that use arbitrary distribution schemes. There are in fact many operators that are both recursive and commutative, e.g., sum, min, max, sort, and select based on value. Processing ships that operate on a single data element at a time, such as select based on value, do not require a "combine" ship, making it even easier to optimize the data-flow graph.

Our optimizer cannot determine automatically whether a processing ship is recursive or commutative. Therefore, it is responsibility of the ship's designer to document the properties of the ship, and to encode the ships used for the optimization in the blueprint description for the ship.

4. Implementation

In this section, we discuss the implementation of armada. A quick overview: armada ships are Java objects instantiated on hosts in the grid. Each host has a persistent server to host ships, called a *harbor*. Each harbor monitors and manages the resources used by each ship, and controls access to system resources through a capability-based mechanism [40,41].

We chose Java for several reasons: it provides a "sandbox" [42] for executing untrusted client code, it is reasonably efficient now that just-in-time compilers are available, it is increasingly popular among HPC programmers, it has convenient mechanisms for remote execution and communication (RMI), and it can interface to application code in other languages through the Java native interface (JNI). Only the ships and harbors need to be written in Java; client code could be in C, C++, or any other language that interfaces with Java.

In the remainder of this section, we detail the ship and harbor implementations, and discuss how we use the extensible markup language (XML) for armada blueprints.

4.1. Ships

Ships are the foundation of the armada framework. They provide all of functionality of the I/O system, including distribution, replica management, data-processing, and so forth. The ship class, the base class for all other ships, provides mechanisms that manage connections between ships, and provide methods that allow the communication of requests and data along those connections. In our current implementation, ships communicate using a combination of RMI and TCP sockets. We use RMI for administrative tasks, such as establishing a connection, sending and receiving profiling information, or shutting down a ship, and we use persistent TCP sockets to communicate requests and data between ships.

An application or library developer extends the ship class to implement application-specific functionality. Extended classes communicate through the "sendRequest" and "sendResult" methods. These methods use the underlying communication protocol to send data to a given destination ship. Abstracting the details of the communication protocol from the library developer allows us to change the protocol without changes to ship libraries.

The ship class manages control connections (for requests) and data connections (for results) separately. In each case, the ship class manages existing connections with a hash table that maps a destination ship's id to a TCP socket. If a connection does not exist, the ship class creates a new connection by calling the "connect" method. The connection persists, and is thus reused, until the application exits, or the connection is explicitly closed by the application.

4.2. Harbors

Each host has a persistent server to host ships, called a *harbor*. The harbor class provides methods (executed through RMI) to “anchor” armada ships to the harbor. The anchor methods instantiate a ship object within the harbor, and thus require a class name, an array of string arguments for the constructor, and an optional class loader. The ShipManager (see Section 3.3) calls the anchor method when deploying ships according to the blueprint.

```
public interface Harbor extends Remote{
   Ship anchor(String className, String[ ] args)
     throws RemoteException;
   Ship anchor(String className, String[ ] args,
        ClassLoader loader)
     throws RemoteException;
 }
```

The harbor is layered on top of the host’s operating system to provide a secure execution environment that allows ships to access the CPU, memory, network, and storage resources of the host machine. The essential components of the harbor include a *security manager* and a *resource manager*.

The security manager is responsible for providing a secure execution environment for application ships. Before installing an untrusted application ship on a harbor, the security manager authenticates the code for the ship and user wishing to install the ship. and authorizes use of the host resources based on the user identity and the security policies of the host. After identifying the user, the security manager installs the ship inside a protected domain, known as a “sandbox ” [42]. Once inside the sandbox, access to resources outside the domain are strictly controlled resource manager. The resource manager provides “capability-based” access [40,41] to system resources (the CPU, network, storage, and memory) outside of the protected domain of the sandbox, and it monitors per-ship and overall usage of the resources available on the host. Although Java provides protection against unauthorized memory accesses, it does not allow object references to be revoked by the system. Capabilities provide the ship with a revocable “ticket” that enables cross-domain access to a resource while the ticket is valid. This scheme prevents abuse of system resources by allowing the harbor to invalidate a ticket if the ship violates a resource usage policy or exceeds a consumption limit assigned to the capability.

The resource manager also monitors and publishes information about the resource consumption of the individual ships and the system as a whole. The harbor requires this information to enforce resource consumption policies, but we make the information available so external programs (like the ship manager) can use it for their own needs.

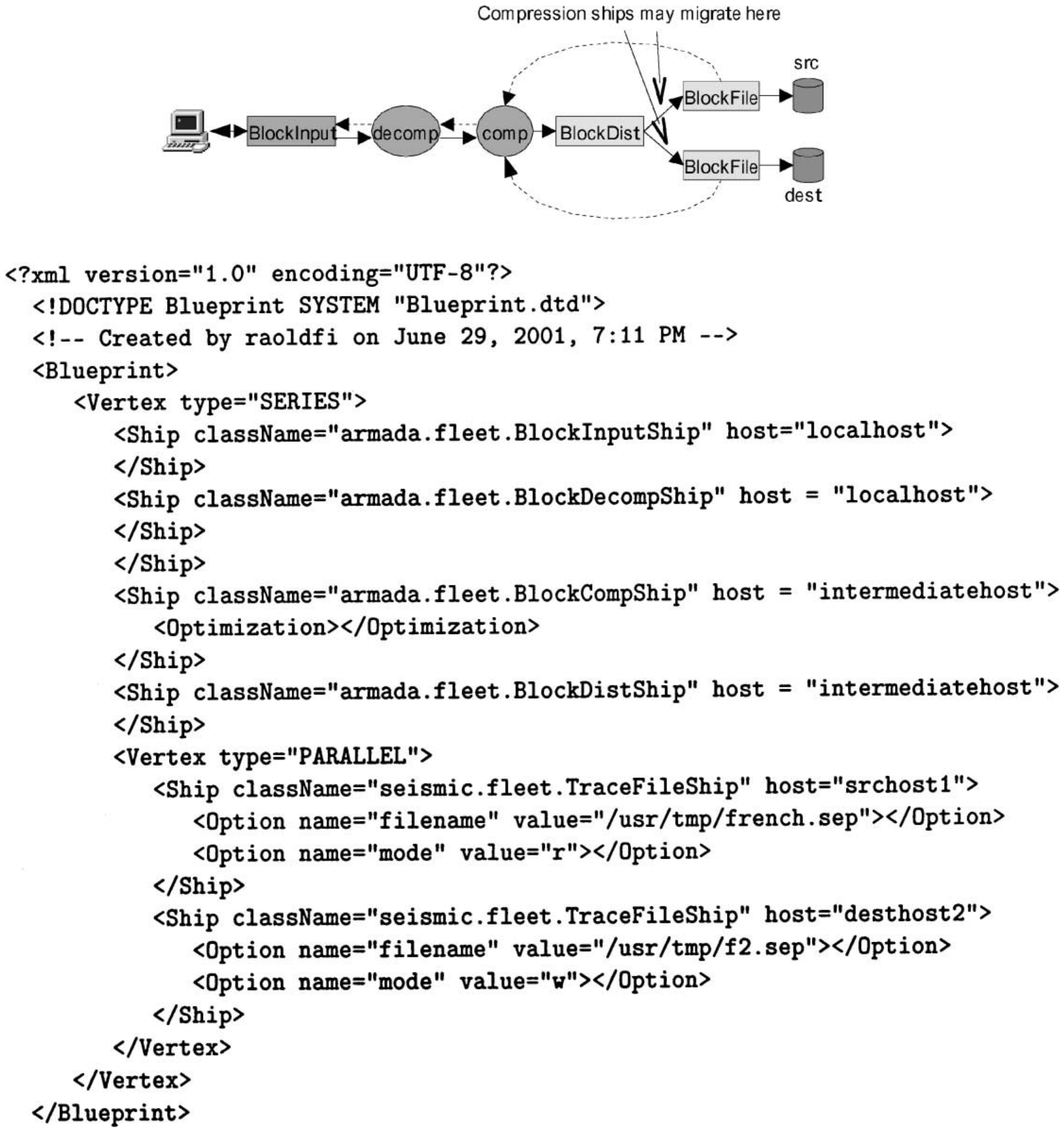
4.3. XML blueprints

We use the XML to encode the blueprints used to describe an armada. We chose XML because XML’s hierarchical structure fits our SP tree needs well and there are existing class libraries (e.g., Java API for XML parsing (JAXP)) that provide mechanisms for creating and manipulating XML documents.

As specified in Section 3.2, blueprints use series–parallel trees to represent ship graphs. A vertex in the SP tree is either a representation of a ship, a series of connected vertices, or a set of parallel vertices. We define the structure of the SP-tree blueprint with the XML document type definition (DTD) shown in Fig. 8. The ship element contains attributes for the classname and the initial host where the ship is to be installed. Additional options (an array of name–value pairs represented as an array of string objects) can be passed into the constructor of the ship by adding Option elements to the ship’s description. Finally, if a processing ship is optimizable (see Section 3.5), the ship designer inserts an optimization element into the ship’s XML description that specifies a set of ships to be used by the blueprint optimizer. If a processing ship operates on individual elements, and thus does not require a ship to combine

```
<?xml version='1.0' encoding='UTF-8'?>
  <!ELEMENT Blueprint (Vertex | Ship)>
  <!ELEMENT Vertex (Vertex | Ship)*>
  <!ATTLIST Vertex
     type (SERIES|PARALLEL) #REQUIRED>
  <!ELEMENT Ship (Option*, Optimization?)>
  <!ATTLIST Ship
     className CDATA #REQUIRED
     host CDATA #REQUIRED >
  <!ELEMENT Option EMPTY>
  <!ATTLIST Option
     name CDATA #IMPLIED
     value CDATA #IMPLIED>
  <!ELEMENT Optimization (Ship)*>
```

Fig. 8. DTD for an armada blueprint.



```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE Blueprint SYSTEM "Blueprint.dtd">
  <!-- Created by raoldfi on June 29, 2001, 7:11 PM -->
  <Blueprint>
     <Vertex type="SERIES">
        <Ship className="armada.fleet.BlockInputShip" host="localhost">
        </Ship>
        <Ship className="armada.fleet.BlockDecompShip" host = "localhost">
        </Ship>
        </Ship>
        <Ship className="armada.fleet.BlockCompShip" host = "intermediatehost">
           <Optimization></Optimization>
        </Ship>
        <Ship className="armada.fleet.BlockDistShip" host = "intermediatehost">
        </Ship>
        <Vertex type="PARALLEL">
           <Ship className="seismic.fleet.TraceFileShip" host="srchost1">
              <Option name="filename" value="/usr/tmp/french.sep"></Option>
              <Option name="mode" value="r"></Option>
           </Ship>
           <Ship className="seismic.fleet.TraceFileShip" host="desthost2">
              <Option name="filename" value="/usr/tmp/f2.sep"></Option>
              <Option name="mode" value="w"></Option>
           </Ship>
        </Vertex>
     </Vertex>
  </Blueprint>
```

Fig. 9. XML blueprint of a simple distributed file.

sub-results, the designer inserts an empty optimization element.

Fig. 9 shows an XML blueprint for a simple distributed file with a compression and de-compression ship added by the application. The application wishes to compress blocks of data before transmission through a slow network. In this application, the compression ship is optimizable. Because the compression ship operates on individual blocks and does not require another ship to combine results, thus, the optimize element is empty.

5. Experimental results

In this section, we measure the performance of two applications that use armada: a remote file copy application, and the I/O portion of a seismic imaging

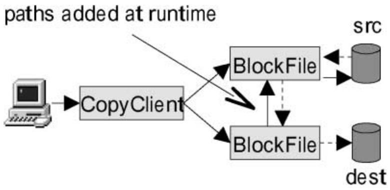


Fig. 10. An armada for a simple remote file copy application. The client application sends a file transfer request to the destination host, which then requests data from the source.

application. It is important to note that although we have a developing implementation of the harbor class that uses JKernel for capability-based access, we use a version without this additional security for our experiments. The JKernel version requires a large amount of tuning, and some debugging, before it will be ready for large-scale experiments.

5.1. Remote file copy

We constructed a remote copy application (shown in Fig. 10) that transfers data from one data server to another. The armada consists of two *BlockFile* server-interface ships that read or write 32KB blocks from the local file system, and a *CopyClient* application-interface ship that initiates a transfer between the two hosts. The CopyClient ship sends a write request to the destination BlockFile ship that identifies the source ship as the data source. When the destination ship receives the write request, it generates read requests and sends them to the source ship. The source ship then returns the data to the destination ship, and the destination ship writes the data to the disk as the blocks arrive. Although this application is unusual because data-flows through the two BlockFile ships without traveling through the client, configuring this type of application is easy and totally transparent to the BlockFile objects.

This test measures how well armada compares with existing remote copy applications for two-party and third-party copies. In a two-party copy, one of the data servers is the host that initiated the transfer. In a third-party transfer, a client initiates a transfer between two separate data servers. This test provides a baseline benchmark for the raw performance of armada, and for the potential of remote ship placement to improve performance.

We used three workstations[2] connected with 100MB Ethernet to perform our tests. Each workstation contains a single disk that achieved a measured disk bandwidth (for sequential accesses) of 7.1Mbps for reads and 11Mbps for writes. We measured a network bandwidth of 66Mbps, with latency measured at 160 μs between each machine.

For the two-party tests, we measured the time to transfer a 126MB file using scp2 (secure copy), rsync (with ssh2), sftp2, nfs, and armada. To avoid additional overhead applied by ssh, we disabled data encryption and compression for all tools that use ssh. Since rsync and sftp do not have support for third-party transfers, we were unable to include them in the second test. Fig. 11 shows timing results (averaged over 10 tests) from the five applications.

The results show that we are competitive with traditional file transfer mechanisms, and in fact outperform nfs and scp for third-party file transfers. For third-party transfers, NFS moves data across the network twice—from source to client, then from client to the destination. We believe scp is moving data across the network twice, but are uncertain about why it is so slow. armada was able to avoid transferring data through the client.

5.2. Seismic imaging

Our second example demonstrates how an armada can be restructured to improve performance for a seismic imaging application. 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 dataset), and data-intensive. A seismic dataset can be large, sometimes more than a terabyte in size, and is often stored as a collection of files. For example, the SEG/EAEG synthetic seismic dataset (SSD) [43,44] is a multi-terabyte synthetically generated dataset consisting of several thousand files. A 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. Each file contains data from a different source position. We refer to the data collected by a

[2] The workstations are PCs running RedHat Linx 7.0. Each have a Pentium II, 500 MHz processor with 256MB of RAM.

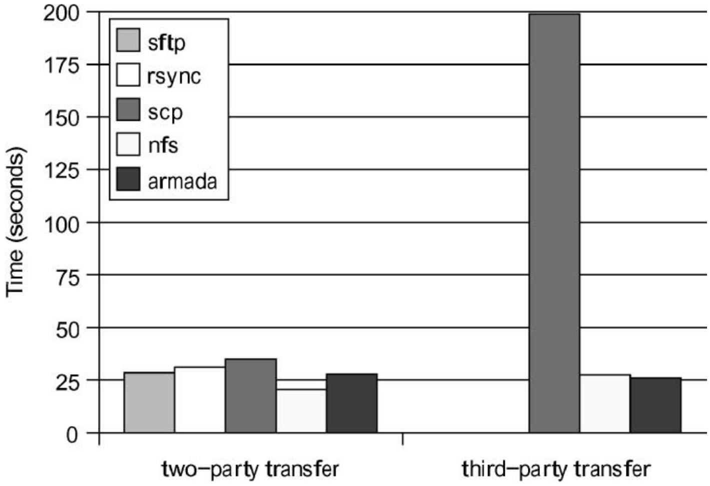


Fig. 11. Measured time to copy a 126MB file between two remote hosts.

single receiver as a "trace", and the file associated with a single source position as a "shot file".

Post-stack migration [2] is a technique that significantly reduces the amount of processing by "stacking" (i.e., summing) traces from each shot file before the computation phase. If the result of the post-stack computation shows promise, the scientist may perform the more computationally intensive "pre-stack" method that calculates an image for each shot file before combining the results.

The post-stack imaging application is ideal for demonstrating the potential of the armada system—it requires access to large datasets, it requires application-specific preprocessing, and datasets are often stored remotely. The goals of our experiments are to investigate the effect of placement of application ships and the benefit of restructuring armadas to move filtering ships close to the data source.

Using armada, we developed an application to emulate the input phase of a "post-stack" seismic imaging application. Our application accesses data from a collection of eight shot files. Each shot file contains 2000 traces (32KB/trace). The shot files that make up the dataset are a subset of the SEG/EAEG SSD, which uses the enhanced stanford exploration project (SEP) format [45]. Note that our dataset is much smaller than a real dataset that may contain thousands of shot files, each with over a thousand traces per file. We have several reasons for using a smaller dataset. First, we do not have the physical resources to store terabytes of data for our experiments, and our prototype is not yet developed enough to deploy to a real grid test-bed where we could access such a dataset. Second, a smaller dataset is sufficient to demonstrate the effectiveness of the armada approach.

Our experiments take place on an 8-processor SGI Origin,[3] and four PCs.[4] All machines are connected with 100MB Ethernet and achieve a measured communication bandwidth of 66Mbps between any two machines, with latencies ranging from 160 μs to 2 ms. The SGI Origin has eight disks that can be accessed in parallel. Each disk achieves a measured bandwidth of 67Mbps for writes and 82Mbps for reads.

We use the five machines to emulate a wide-area network that consists of three separate domains: two domains that contain the data, and a client domain. The data domains consist of four processors each from the Origin machine, and the client domain contains at least one of the Linux machines to host the application. We emulate a wide-area network by forcing communications between domains to pass through a

[3] The Origin is an 8-processor SMP that uses R10000 186 MHz processors. It has 4 GB main memory and uses the IRIX 6.5 operating system.

[4] The PCs consist of one 2-processor (each 1 GHz Pentium Pros) SMP machine with 500MB of main memory, and three single processor (500 MHz Pentium IIs) machines. All run RedHat Linux Version 7.0.

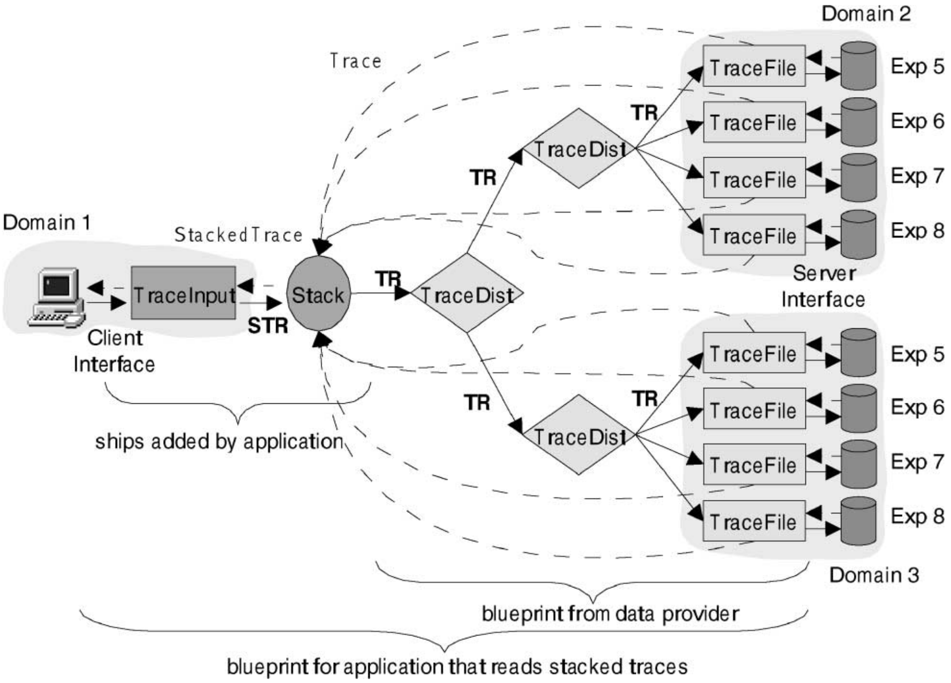


Fig. 12. Blueprint for an application that reads stacked traces.

single Linux machine that uses NistNet [46] to control bandwidth and network latency. We use the remaining Linux machines to host armada ships between the host and the data servers. These machines can exist in any of the three domains.

Fig. 12 shows an armada blueprint for the application. The data provider presents the collection of files as an armada of ships that expects a *TraceRequest* object as input and returns a *Trace* object. A TraceRequest identifies a seismic trace based on the shot number, and the index that identifies the trace within the experiment. The armada consists of three layers of ships: a trace-distribution ship (*TraceDist*) that distributes trace requests to the correct domain, another layer of trace-distribution ships that send trace requests to the correct storage server, and storage interface ships (*TraceFile*) that retrieve trace data stored in SEP format, from the local disk of a data server.

The application extends the armada from the data provider by first prepending a *Stack* operator ship and a *TraceInput* ship. The Stack ship takes as input a *StackedTraceRequest* that identifies an array of traces. It converts a single StackedTraceRequest into an array of TraceRequests and forwards the TraceRequests to the first distribution ship from the data provider. The Stack ship returns a *StackedTrace* object that represents the vector sum of the data from each trace. In front of the Stack ship is the TraceInput client-interface ship that provides a simple interface for requesting stacked traces.

We used two configurations of the armada (Fig. 13). The first configuration places the stacking code in the client domain. This configuration represents the traditional approach where most of the application-specific functionality exists on the client. In the second configuration, we restructured the graph and replace the Stack ship with a tree of three types of ships: a *Merge* ship, that combines results from the two data domains; a *Group* ship, placed in each data domain, to combine individual TraceRequests into StackedTraceRequests; and another Stack ship following each Group ship, to stack the subset of traces stored in that domain. Like the Stack ship, the Merge ship takes as input Staked-TraceRequests and converts them to TraceRequests, allowing the Group and Stack ships to migrate toward the data servers without modifying the input types

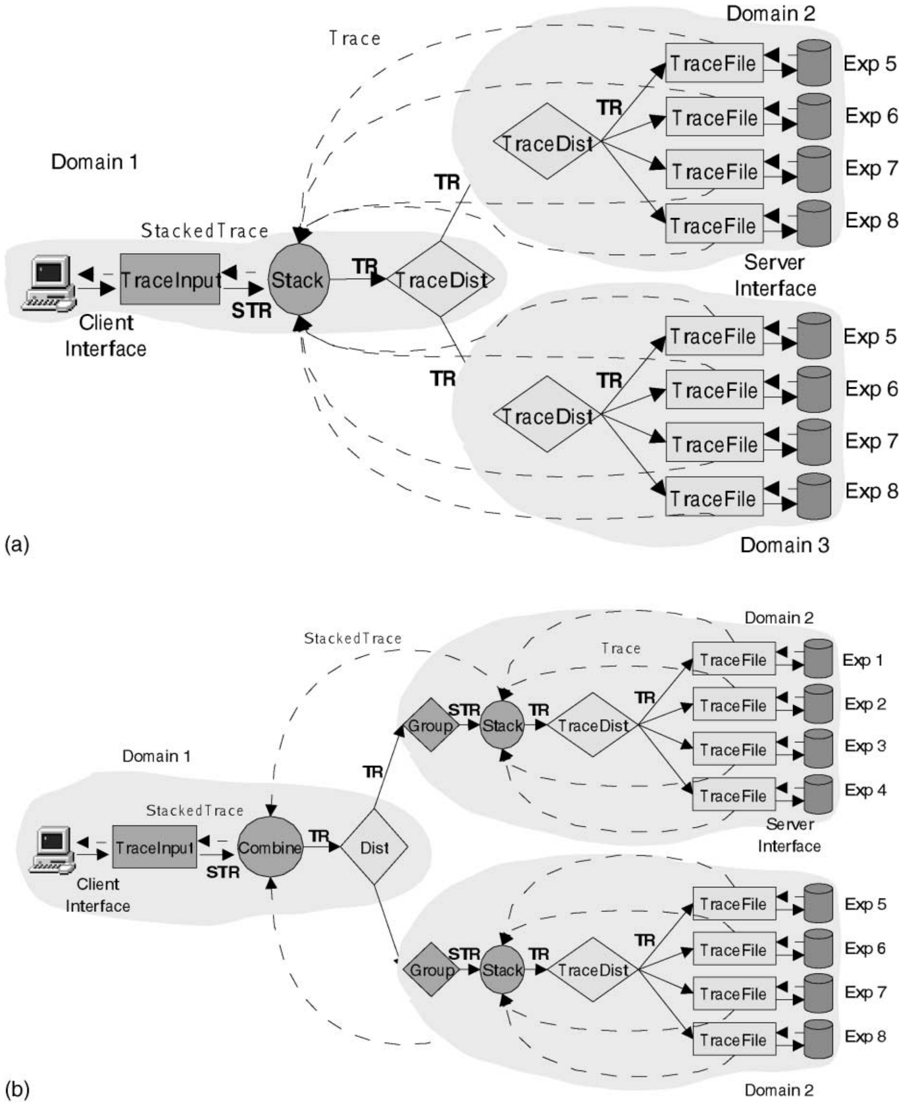


Fig. 13. Two configurations of the seismic application: (a) unoptimized armada; (b) optimized armada.

of the distribution ships. The programmer that writes the code for the Stack ship also codes the Merge and Group ships, and she identifies the ships to use for optimization in the blueprint (see Section 3.5).

The plots in Fig. 14 show running times of the two armadas as the available bandwidth between domains increase. We experimented with three different inter-domain latencies: (a) 2 ms, equivalent to a local-area network, (b) 100 ms, equivalent to a cross-country connection, and (c) 200 ms, for domains located on different continents. Fig. 15 shows the same data, unoptimized in (a) and optimized in (b), allowing easy comparison across the three latencies.

The results show, as we expected, that the second configuration outperforms the first configuration, particularly in situations with limited bandwidth or high latency. In both configurations, the application appears to be bandwidth-limited for bandwidths below 2Mbps

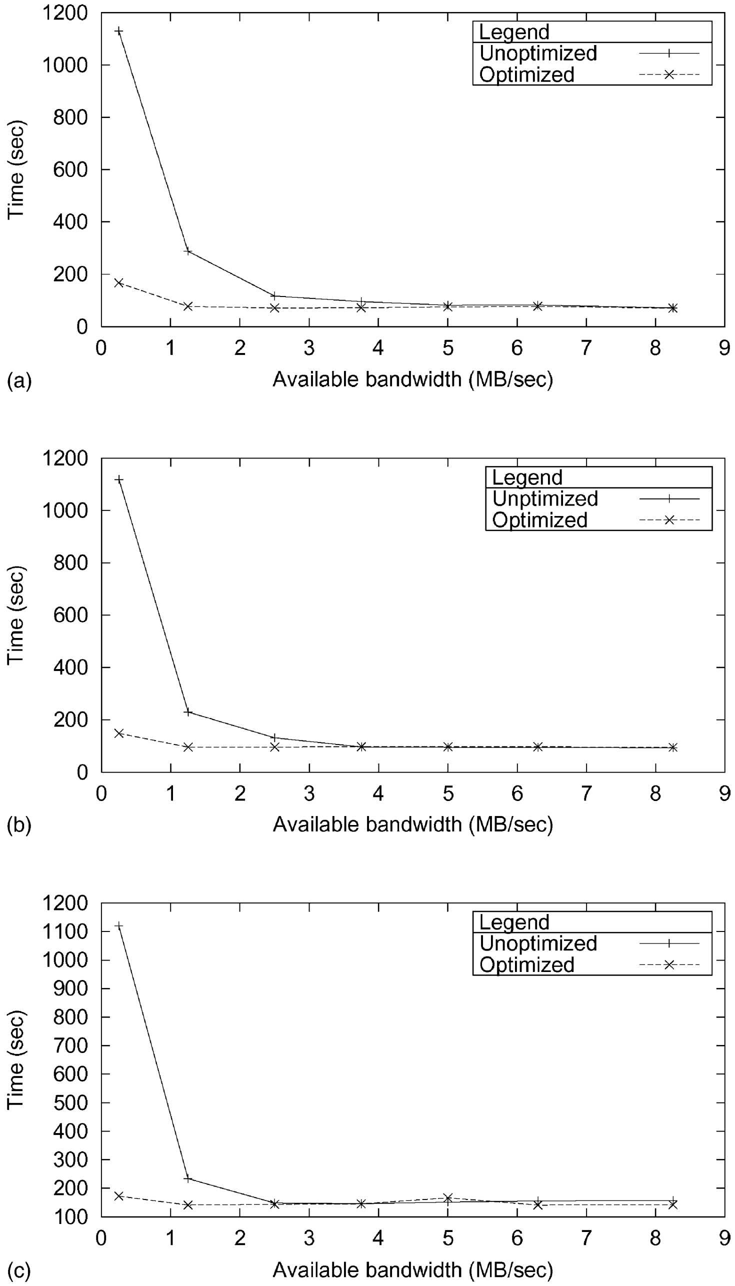


Fig. 14. Comparison of the running time of the optimized and the unoptimized armada for the seismic application that reads eight shot files (approximately 32MB each): (a) latency = 2 ms; (b) latency = 100 ms; (c) latency = 200 ms.

and latency limited elsewhere. We clearly show this effect in Fig. 16, which shows the throughput of the two armadas (measured as the ratio of the total number of bytes read from disk and the running time). The straight line in the figure shows the available bandwidth between the data domains and the client domain. The optimized application achieves a higher (perceived) throughput than the available bandwidth

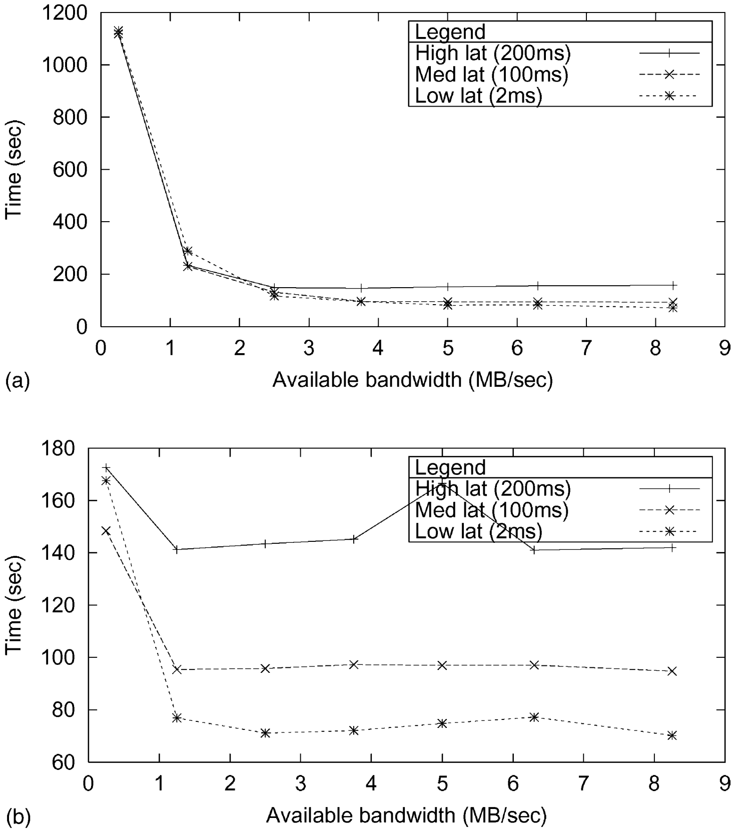


Fig. 15. Plots that show how latency effects the two armadas: (a) unoptimized armada; (b) optimized armada.

because it filters three-fourths of the total bytes in the data domain.

To understand why the application is latency-limited for high bandwidths, we measured the computational rate of each computer (to estimate the stack operation), and the time required to serialize and de-serialize a single trace. Then we used those measurements to construct a rough analytic model of the data-flow path for the optimized armada. In the optimized armada, data passes through four ships on its way to the client: TraceFile, Stack, Merge, and TraceInput. Our model measures the time to transfer a single trace between each of these ships. We show the equations used in our model below. The variables α_{des}, α_{ser}, and α_{comm} are the latencies (in seconds) associated with de-serializing a trace, serializing a trace, and initiating a network transfer. β_{disk} and β_{comm} are the bandwidths (in bytes/s) available for reading data from the disk and sending data through the network. The variable *flops*, is the measured number of floating points per second a computer can calculate. The variable k represents the size (in bytes) of a trace.

The approximate time to transfer a single trace from a TraceFile to a Stack ship is

$$t = k\frac{1}{\beta_{\mathrm{disk}}} + \alpha_{\mathrm{des}} + \alpha_{\mathrm{ser}} + \alpha_{\mathrm{comm}} + k\frac{8}{\beta_{\mathrm{comm}}}. \qquad (1)$$

We divide the communication bandwidth by 8 because the network is shared by eight other FileShips. The time to transfer data from the Stack ship to the Merge ship is

$$t = \alpha_{\mathrm{des}} + k\frac{1}{\mathrm{flops}} + \alpha_{\mathrm{ser}} + \alpha_{\mathrm{comm}} + k\frac{2}{\beta_{\mathrm{comm}}}. \qquad (2)$$

The equation calculating the time from the Merge ship to the TraceInput ship is the same as Eq. (2).

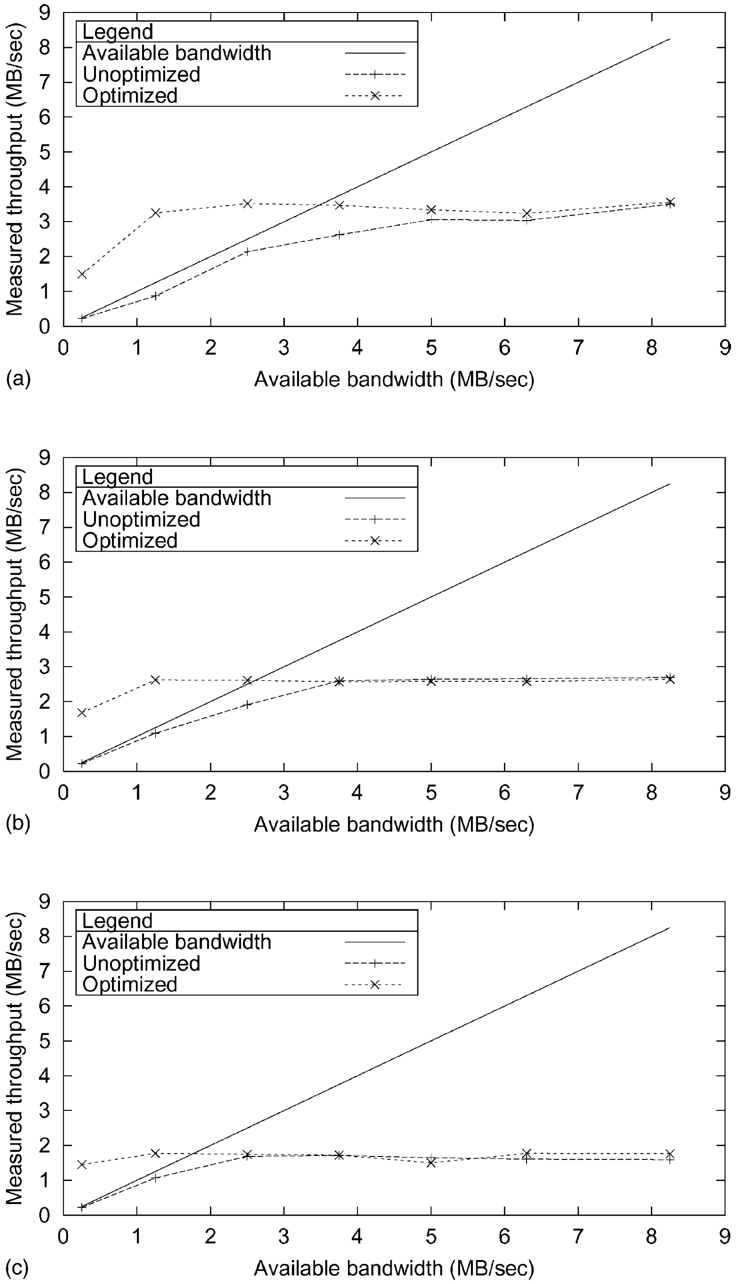


Fig. 16. Performance relative to available bandwidth: (a) latency = 2 ms; (b) latency = 100 ms; (c) latency = 200 ms.

The approximate throughput is *k*/*t* for the path from the TraceFile to the Stack ship, 4*k*/*t* for transfer from the Stack ship to the Merge ship, and 8*k*/*t* from the Merge ship to the TraceInput ship. We multiply the throughputs to account for the amount of data removed by each filter. Table 1 shows the measured values for each variable, and in Fig. 17, we plot the estimated throughput to the application

Table 1
Measured latencies and bandwidths used by the optimized armada

	TraceFile to Stack Origin to Fast PC	Stack to Merge	Merge to TraceInput
k	32KB	32KB	32KB
α_{des}	7 ms	800 μs	800 μs
α_{ser}	10 ms	1.4 ms	1.4 ms
α_{comm}	2 ms	{2,100,200} ms	160 μs
β_{comm}	8.25Mbps	{0.2, 0.4, ..., 8}Mbps	8.25Mbps
β_{disk}	83Mbps	–	–
Flops	13 Mflop/s	70 Mflop/s	70 Mflop/s

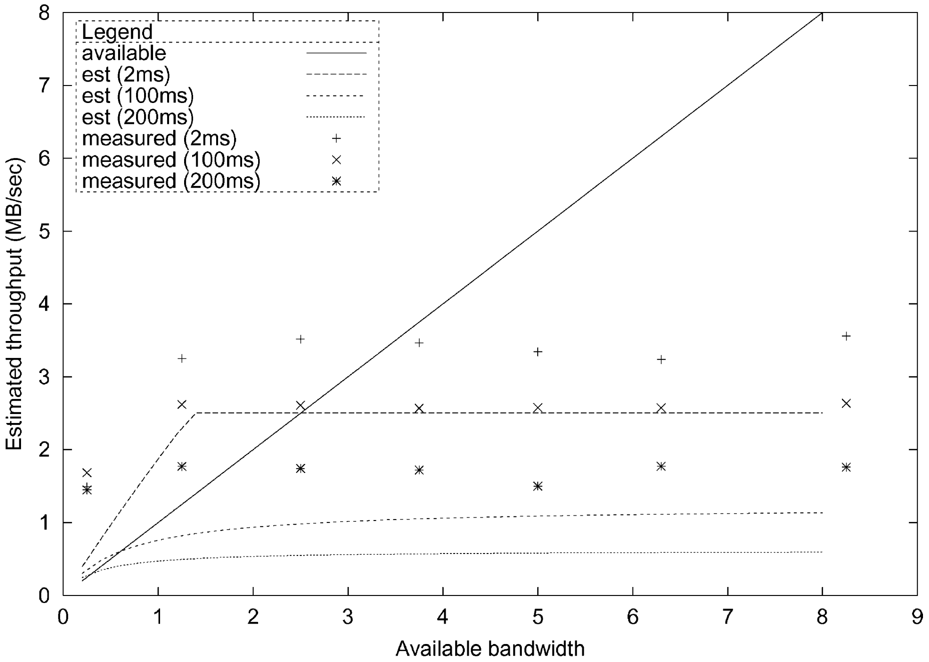


Fig. 17. Approximation of the running time for the optimized armada.

as the minimum of the three calculated throughputs.

We notice that the approximate application bandwidth is close to the measured bandwidth of the application. The application achieves slightly higher performance because we pipeline data requests, hiding some of the latency costs. The analysis indicates that the primary factor limiting performance of our application is Java serialization.

It is clear from the results that further tuning is necessary. For the current prototype, we were more concerned with finishing the implementation than improving performance.

6. Summary

In this paper, we describe the design of a framework that supports data-intensive parallel applications in a wide-area computational network, or “grid”. Our framework, armada, allows applications and data providers to create datasets that span the grid. Applications combine flexible modules called “ships” into series–parallel graphs called “armadas”. These armadas provide a path for the application to access the data. Typically, the armada is a combination of a graph designed by the dataset owner and extensions provided by the application. The armada thus encodes the structure of the data, the application's interface, and intermediate processing and filtering.

Our framework includes a rich hierarchy of ship classes, a "blueprint" mechanism to specify the structure of an armada graph, execution environments called "harbors" that securely host ships, and algorithms that optimize blueprints and deploy ships onto the network of harbors.

The preliminary results presented here demonstrate the value of the armada approach. The ability to flexibly combine modular ships into an access graph leads directly to the ability to adjust the placement of functionality within the grid. It is well known that placement can improve performance by reducing network load. Our placement and graph-optimization schemes enable applications-specific code to migrate deep into potentially complex data distributions to filter data close to the source. Since data-intensive grid applications are typically limited by the network, we expect armada's approach will lead to better overall application performance.

Our work on the armada system is far from complete. Our priority is to complete the implementation by including capability-based mechanisms on the harbors; developing efficient placement algorithms that consider memory and CPU rates, as well as network capacity; and performance tuning. Our plans also include a further study of algorithms for optimizing the graphs described by blueprints, and protocols that can improve the communication efficiency of data-flowing through an armada.

Acknowledgements

Work funded by Sandia National Laboratories under contract DOE-AV6184.

References

[1] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, Los Altos, CA, 1998.

[2] O. Yilmaz, Seismic Data Processing: Investigations in Geophysics, Vol. 2, Society of Exploration Geophysicists, Tulsa, Oklahoma, 1990, p. 74170.

[3] J. Demmel, M.Y. Ivory, S.L. Smith, Modeling and identifying bottlenecks in EOSDIS, in: Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, San Jose, CA, 1996, pp. 300–308.

[4] P. Lyster, K. Ekers, J. Guo, M. Harber, D. Lamich, J. Larson, R. Lucchesi, R. Rood, S. Schubert, W. Sawyer, M. Sienkiewicz, A. da Silva, J. Stobie, L. Takacs, R. Todling, J. Zero, Parallel computing at the NASA data assimilation office (DAO), in: Proceedings of the SC97 on High Performance Networking and Computing, IEEE Computer Society Press, San Jose, CA, 1997.

[5] W. Greiman, W.E. Johnston, C. McParland, D. Olson, B. Tierney, C. Tull, High-speed distributed data handling for HENP, in: Proc. Int. Conf. on Computing in High Energy Physics, Berlin, Germany, 1997, http://www-rnc.pbp.gov/computing/ldrd_fy97/henpdata.htm

[6] S. Young, G.G.Y. Fan, D. Hessler, S. Lamont, Implementing a collaborator for microscopic digital anatomy, Int. J. Supercomput. Appl. High Perform. Comput. 10 (2) (1996) 170–181.

[7] Y. Wang, F. D. Carlo, I. Foster, J. Insley, C. Kesselman, P. Lane, G. von Laszewski, D. Mancini, I. McNulty, M.-H. Su, B. Tieman, A quasi-realtime X-ray microtomography system at the advanced photon source, in: Proceedings of the SPIE99, Vol. 3772, 1999, pp. 318–327.

[8] J. Leigh, A.E. Johnson, T.A. DeFanti, S. Bailey, R. Gromman, A methodology for supporting collaborative exploratory analysis of massive data sets in tele-immersive environments, in: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Redondo Beach, CA, 1999, pp. 62–69.

[9] L.A. Freitag, R.M. Loy, Adaptive, multiresolution visualization of large data sets using a distributed memory octree, in: Proceedings of the SC99 on High Performance Networking and Computing, ACM Press/IEEE Computer Society Press, Portland, OR, 1999.

[10] E. Franke, M. Magee, Reducing data distribution bottlenecks by employing data visualization filters, in: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Redondo Beach, CA, 1999, pp. 255–262.

[11] D. Mackay, G. Mahinthakumar, E. D'Azevedo, A study of I/O in a parallel finite element groundwater transport code, Int. J. High Perform. Comput. Appl. 12 (3) (1998) 307–319.

[12] R.A. Oldfield, D.E. Womble, C.C. Ober, Efficient parallel I/O in seismic imaging, Int. J. High Perform. Comput. Appl. 12 (3) (1998) 333–344.

[13] E. Smirni, D. Reed, Lessons from characterizing the input/output behavior of parallel scientific applications, Perform. Eval. Int. J. 33 (1) (1998) 27–44 http://vibes.cs.uiuc.edu/Publications/Papers/PerfEval98.ps.gz

[14] J. Nieplocha, I. Foster, R. Kendall, ChemIO: high-performance parallel I/O for computational chemistry applications, Int. J. High Perform. Comput. Appl. 12 (3) (1998) 345–363.

[15] N. Nieuwejaar, D. Kotz, The Galley parallel file system, Parallel Comput. 23 (4) (1997) 447–476.

[16] A.S. Grimshaw, J. Prem, High performance parallel file objects, in: Proceedings of the Sixth Annual Distributed-memory Computer Conference, 1991, pp. 720–723.

[17] D. Kotz, C.S. Ellis, Practical prefetching techniques for multiprocessor file systems, J. Distrib. Parallel Databases 1 (1) (1993) 33–51.

[18] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka, Informed prefetching and caching, in: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, ACM Press, Copper Mountain, CO, 1995, pp. 79–95.

[19] T.H. Cormen, D. Kotz, Integrating theory and practice in parallel file systems, in: Proceedings of the 1993 DAGS/PC Symposium, Dartmouth Institute for Advanced Graduate Studies, Hanover, NH, 1993, revised as Dartmouth PCS-TR93-188 on 9/20/1994, pp. 64–74.

[20] D. Womble, D. Greenberg, S. Wheat, R. Riesen, Beyond core: making parallel computer I/O practical, in: Proceedings of the 1993 DAGS/PC Symposium, Dartmouth Institute for Advanced Graduate Studies, Hanover, NH, 1993, pp. 56–63.

[21] D. Kotz, Disk-directed I/O for MIMD multiprocessors, ACM Trans. Comput. Syst. 15 (1) (1997) 41–74.

[22] D. Kotz, Expanding the potential for disk-directed I/O, in: Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, San Antonio, TX, 1995, pp. 490–495.

[23] A. J. Borr, F. Putzolu, High performance SQL through low-level system integration, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, Chicago, IL, 1988, pp. 342–349.

[24] T. Kurc, C. Chang, R. Ferreira, A. Sussman, Querying very large multi-dimensional datasets in ADR, in: Proceedings of the SC99 on High Performance Networking and Computing, ACM Press/IEEE Computer Society Press, Portland, OR, 1999.

[25] J.S. Heidemann, G.J. Popek, File-system development with stackable layers, ACM Trans. Comput. Syst. 12 (1) (1994) 58–89.

[26] D. Rosenthal, Requirements for a stacking Vnode/VFS interface, Technical Report SD-01-02-N014, Unix International, 1992.

[27] G.C. Skinner, T. Wong, Stacking vnodes: a progress report, in: Proceedings of the 1993 Summer USENIX Technical Conference, USENIX Association, 1993, pp. 161–174.

[28] O. Krieger, M. Stumm, HFS: A performance-oriented flexible file system based on building-block compositions, ACM Trans. Comput. Syst. 15 (3) (1997) 286–321.

[29] E. Zadok, I. Badulescu, A. Shender, Extending file systems using stackable templates, in: Proceedings of the 1999 Annual USENIX Technical Conference, USENIX Association, 1999, pp. 57–70.

[30] E. Zadok, J. Nieh, FiST: a language for stackable file systems, in: Proceedings of the 2000 Annual USENIX Technical Conference, USENIX Association, 2000, pp. 55–70.

[31] V. Messerli, Tools for parallel I/O and compute intensive applications, Ph.D. Thesis, École Polytechnique Fédérale de Lausanne, 1999, p. 1915.

[32] B.A. Gennart, M. Mazzariol, V. Messerli, R.D. Hersch, Synthesizing parallel imaging applications using the CAP Computer Aided Parallelization Tool, in: Proc. IS&T SPIE 10th Ann. Symp. on Electronic Imaging. Storage and Retrieval for Image and Video Databases VI, pp. 446–458.

[33] B. Plale, K. Schwan, dQUOB: managing large data flows by dynamic embedded queries, in: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing, 2000.

[34] M. Rodríguez-Martínez, N. Roussopoulos, MOCHA: a self-extensible database middleware system for distributed data sources, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, TX, 2000.

[35] M.D. Beynon, R. Ferreira, T. Kurc, A. Sussman, J. Saltz, DataCutter: Middleware for filtering very large scientific datasets on archival storage systems, in: Proceedings of the 2000 Mass Storage Systems Conference, IEEE Computer Society Press, College Park, MD, 2000, pp. 119–133.

[36] K. Amiri, D. Petrou, G.R. Ganger, G.A. Gibson, Dynamic function placement for data-intensive cluster computing, in: Proceedings of the 2000 Annual USENIX Technical Conference, USENIX Association, 2000, pp. 307–322.

[37] G. Hunt, M. Scott, The Coign automatic distributed partitioning system, in: Proceedings of the 1999 Symposium on Operating Systems Design and Implementation, USENIX Association, San Diego, CA, 1999, pp. 45–56.

[38] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, 1999, pp. 182–189.

[39] A. Colvin, T.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), 1998, pp. 23–33.

[40] J. Saltzer, M. Schroeder, The protection of information in computer systems, IEEE 63 (9) (1975) 1278–1308.

[41] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, T. von Eicken, Implementing multiple protection domains in Java, in: Proceedings of the 1998 Annual USENIX Technical Conference, New Orleans, LA, 1998.

[42] R. Wahbe, S. Lucco, T.E. Anderson, S.L. Graham, Efficient software-based fault isolation, in: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, ACM Press, Asheville, NC, 1993, pp. 203–216.

[43] R.A. Oldfield, B. D. Semeraro, J.P. VanDyke, Parallel acoustic wave propagation and generation of a seismic dataset, in: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Fransisco, CA, 1995, pp. 243–244.

[44] Gas and Oil National Information Infrastructure (GONII)#, The Synthetic Seismic Dataset. http://www.llnl.gov/gonii/ssd.html.

[45] SEG/EAEG 3-D Modeling Committee, SEP Format Description for Numerical Data. http://www.seg.org/research/3Dmodel/SEPformat.html.

[46] National Institute of Standards and Technology, NistNet Home page. http://snad.ncsl.nist.gov/itg/46/.

Ron Oldfield is a graduate student in the Computer Science Department at Dartmouth College. He received his BSc 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. For GONII, he implemented a parallel 3D finite-difference acoustic wave propagation code that was used to generate a large synthetic seismic dataset. He also worked on the R&D 100 award winning project “Salvo”, a project to develop a 3D finite-difference prestack-depth migration algorithm for massively parallel architectures. His work with the Salvo focused on minimizing the effect of the massive I/O requirements associated with seismic processing. At Dartmouth, his research targets parallel file systems and parallel I/O for the grid. He is also an active member of the Remote Data Access group of the Global Grid Forum.

David Kotz is an Associate Professor of Computer Science at Dartmouth College, Hanover, NH. He received his MS and PhD degrees in Computer Science from Duke University in 1989 and 1991, respectively. He received the AB degree in Computer Science and Physics from Dartmouth College, Hanover, NH, in 1986. He rejoined Dartmouth College in 1991 and was promoted with tenure to Associate Professor in 1997. His research interests include mobile agents, parallel and distributed operating systems, multiprocessor file systems, and computer ethics. He is a member of the ACM, IEEE Computer Society, and USENIX associations, and of Computer Professionals for Social Responsibility.