Efficient I/O for Computational Grid Applications

Ron Oldfield

PhD. Thesis Defense
Department of Computer Science, Dartmouth College

May 15, 2003

Committee: David Kotz (chair), Thomas Cormen, Robert Gray, and David Womble
Computational Grids

Networks of geographically distributed heterogeneous systems and devices

*Data-intensive* scientific applications

- Access large remote datasets (terabyte–petabyte)
- Datasets often need pre/post-processing
- Often computationally intensive
- Examples
  - Climate modeling
  - Astronomy
  - Computational biology
  - High-energy physics
The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Requests cause pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application-programmer (pre/post-processing)
The Armada Framework

- Application deploys a graph of distributed objects (ships)
- Requests cause pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application-programmer (pre/post-processing)
The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Requests cause pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application-programmer (pre/post-processing)
The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Requests cause pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application-programmer (pre/post-processing)
Armada

Armada is not a data storage system. *Armada is not a parallel file system.*

The *data segments* that make up a *data set* are stored in conventional data servers as files, databases, or the like.

The Armada graph encodes most functionality provided by the I/O system:

- programmers interface,
- data layout,
- caching and prefetching policies,
- interfaces to heterogeneous data servers.
Armada can...

With Armada, one can...

- build a graph for parallel access to a group of legacy files,
- present many similar data sets through a standard interface, and
- provide transparent access to derived “virtual” data—either cached or calculated as needed.
Restructuring

Problems with the example application:

- Potential bottlenecks in composed graph
- original graph restricts placement alternatives for filter

Armada restructures original graph to improve data flow.
After restructuring:

1. Armada deploys ships to appropriate administrative domains to optimize data flow, then

2. domain-level resource manager decides placement of individual ships.
After restructuring:

1. Armada deploys ships to appropriate administrative domains to optimize data flow, then

2. domain-level resource manager decides placement of individual ships.
Talk Outline

- Introduction
- Framework details
  - Ships
  - Graph Representation
- Restructuring graphs to improve data flow
- Partitioning graphs and placing ships
- Experiments
- Conclusion
Armada includes a rich set of extensible ship classes.

- **Structural**
  - Distribute (partition, select, copy)
  - Merge

- **Non-Structural**
  - Data Processing
    - Filter (>, <, =)
    - Transform (FFT, unit conversion)
    - Reduce (min, max, sum)
    - Permute (sort, transpose)
  - Optimization
    - Cache
    - Prefetch
  - Interface
    - Client (Matrix, Line, String, stdio)
    - Storage (File, Query)
Armada includes a rich set of extensible ship classes.

*Distribute* ships partition requests or data to multiple output streams.
Armada includes a rich set of extensible ship classes.

*Merge* ships interleave requests or data from multiple input streams.
Armada includes a rich set of extensible ship classes.

**Data-processing** ships manipulate data, either individually, or in groups as it passes through the ship.
Armada includes a rich set of extensible ship classes.

Optimization ships improve I/O performance through latency-reduction techniques like caching and prefetching.
Armada includes a rich set of extensible ship classes.

**Client-interface** ships convert method calls to a set of requests for data.

**Storage-interface** ships access storage devices to process requests.
Properties of Ships

Properties of ships are

- used by restructuring and placement algorithms
- assigned by the programmer
- encoded in the ship’s definition

Properties identify whether a ship

- is data- or request-equivalent
- increases or decreases data flow,
- is parallelizable
Request and Data Equivalent Ships

A sequence $A$ is *equivalent* to sequence $B$ (denoted $A \equiv B$) if $B$ is a permutation of $A$, or if $B$ is a set of subsequences that partition $A$.

Examples:

\[
\begin{align*}
\{1, 2, 3, 4, 5\} & \equiv \{2, 3, 5, 1, 4\} \\
\{1, 2, 3, 4, 5\} & \equiv \{\{2, 3\}, \{1, 4, 5\}\} \\
\{1, 2, 3, 4, 5\} & \equiv \{\{2, 3\}, \{1, 5, 4\}\}
\end{align*}
\]

In other words, order does not matter.
A sequence $A$ is *equivalent* to sequence $B$ (denoted $A \equiv B$) if $B$ is a permutation of $A$, or if $B$ is a set of subsequences that partition $A$.

A *request-equivalent* ship produces request sequence equivalent to its input.

A *data-equivalent* ship produces data sequence equivalent to its input.

*Most structural ships are both request and data-equivalent.*
A sequence $A$ is \textit{equivalent} to sequence $B$ (denoted $A \equiv B$) if $B$ is a permutation of $A$, or if $B$ is a set of subsequences that partition $A$.

Distribution ships partition requests or data

- $S_1$, $S_2$, and $S_3$ are subsequences of $R$.
- $R \equiv \{S_1, S_2, S_3\}$
A sequence $A$ is *equivalent* to sequence $B$ (denoted $A \equiv B$) if $B$ is a permutation of $A$, or if $B$ is a set of subsequences that partition $A$.

Merge ships interleave requests or data

- $R_1$, $R_2$, and $R_3$ are subsequences of $S$.
- \{$R_1, R_2, R_3$\} $\equiv S$
Ships that Change Data Flow

*Data-reducer:* a ship that decreases the data flow

- filter
- compress
- reduce (min, max, sum)

*Data-increaser:* a ship that increases the data flow

- cache
- decompress
Parallelizable Ships

**Parallelizable**: a ship that can transform into multiple ships

- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable*, *recursive*
**Parallelizable Ships**

**Parallelizable**: a ship that can transform into multiple ships

- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable*, *recursive*

Right-parallelizable

- Original
- Replicated
- Recursed
Parallelizable Ships

**Parallelizable**: a ship that can transform into multiple ships

- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable, recursive*

Left-parallelizable

![Diagram showing original, replicated, and recursed versions of a parallelizable ship]

Armada – p.13
We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Representation

We use a \textit{series-parallel tree} (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
Graph Restructuring

Goals:

- remove bottlenecks (increase parallelism)
- allow effective placement of ships

We restructure by *swapping* adjacent ships in the SP-tree

- increase parallelism by swapping *parallelizable* ships with *structural* ships
- reduce network traffic on slow links by
  - moving *data-reducing* ships toward data source,
  - moving *data-increasing* ships toward data dest
The Restruct Algorithm

The Restruct algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked dirty).

1. if $N$ is a leaf or clean (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) Restruct each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. $child \leftarrow$ remove leftmost child of $N$
      ii. append $child$ to $S$
      iii. Slide $child$ left
   (c) $N \leftarrow S$
4. mark $N$ clean
The Restruct Algorithm

The Restruct algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if $N$ is a leaf or *clean* (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) Restruct each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. $child \leftarrow$ remove leftmost child of $N$
      ii. append $child$ to $S$
      iii. Slide $child$ left
   (c) $N \leftarrow S$
4. mark $N$ clean
The Restruct Algorithm

The \textsc{Restruct} algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked \textit{dirty}).

1. if $N$ is a leaf or \texttt{clean} (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) \textsc{Restruct} each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
       i. $child \leftarrow$ remove leftmost child of $N$
       ii. append $child$ to $S$
       iii. \texttt{Slide} $child$ left
   (c) $N \leftarrow S$
4. mark $N$ \texttt{clean}
The Restruct Algorithm

The Restruct algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked dirty).

1. if $N$ is a leaf or clean (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) Restruct each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
       i. child $\leftarrow$ remove leftmost child of $N$
       ii. append child to $S$
       iii. Slide child left
   (c) $N \leftarrow S$
4. mark $N$ clean
The Restruct Algorithm

The **Restruct** algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if $N$ is a leaf or *clean* (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) **Restruct** each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. `child ← remove leftmost child of $N$
      ii. append `child to $S$
      iii. `Slide `child left
   (c) $N ← S$
4. mark $N$ *clean*
The Restruct Algorithm

The `Restruct` algorithm traverses the SP-tree (depth-first) from node \( N \), revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if \( N \) is a leaf or *clean* (base case)
   (a) return
2. else if \( N \) is a parallel node
   (a) `Restruct` each child of \( N \)
3. else if \( N \) is a series node
   (a) create a new series node \( S \)
   (b) while \( N \) has children
      i.  `child` ← remove leftmost child of \( N \)
      ii. `append child` to \( S \)
      iii. `Slide child` left
   (c) \( N \) ← \( S \)
4. mark \( N \) *clean*
The **Restruct** Algorithm

The **Restruct** algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. **if** $N$ **is a leaf or clean** (base case)
   (a) return

2. **else if** $N$ **is a parallel node**
   (a) **Restruct** each child of $N$

3. **else if** $N$ **is a series node**
   (a) create a new series node $S$
   (b) while $N$ has children
      i. child $\leftarrow$ remove leftmost child of $N$
      ii. append child to $S$
      iii. **Slide** child left
   (c) $N \leftarrow S$

4. **mark** $N$ clean
The **Restruct** Algorithm

The **Restruct** algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if $N$ is a leaf or *clean* (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) **Restruct** each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. $child \leftarrow$ remove leftmost child of $N$
      ii. append $child$ to $S$
      iii. **Slide** $child$ left
   (c) $N \leftarrow S$
4. mark $N$ *clean*
The Restruct Algorithm

The `RESTRUCT` algorithm traverses the SP-tree (depth-first) from node \( N \), revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if \( N \) is a leaf or clean (base case)
   (a) return
2. else if \( N \) is a parallel node
   (a) `RESTRUCT` each child of \( N \)
3. else if \( N \) is a series node
   (a) create a new series node \( S \)
   (b) while \( N \) has children
      i. \( \text{child} \leftarrow \text{remove leftmost child of } N \)
      ii. append \( \text{child} \) to \( S \)
      iii. \( \text{SLIDE} \) \( \text{child} \) left
   (c) \( N \leftarrow S \)
4. mark \( N \) clean
The Restruct Algorithm

The \textsc{Restruct} algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked \textit{dirty}).

1. if $N$ is a leaf or \textit{clean} (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) \textsc{Restruct} each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. \textit{child} $\leftarrow$ remove leftmost child of $N$
      ii. append \textit{child} to $S$
      iii. \textsc{Slide} \textit{child} left
   (c) $N \leftarrow S$
4. mark $N$ \textit{clean}
The Restruct Algorithm

The **Restruct** algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if $N$ is a leaf or *clean* (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) **Restruct** each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. `child ← remove leftmost child of $N$
      ii. `append `child to $S$
      iii. `Slide `child left
   (c) $N ← S$
4. mark $N$ *clean*
The Restruct Algorithm

The **Restruct** algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if $N$ is a leaf or *clean* (base case)
   (a) return
2. else if $N$ is a parallel node
   (a) **Restruct** each child of $N$
3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
      i. `child ←` remove leftmost child of $N$
      ii. `append child to S`
      iii. `Slide child left`
   (c) $N ← S$
4. mark $N$ *clean*
The Restruct Algorithm

The `RESTRUCT` algorithm traverses the SP-tree (depth-first) from node \( N \), revisiting when necessary (all series and parallel nodes are initially marked *dirty*).

1. if \( N \) is a leaf or clean (base case)
   (a) return
2. else if \( N \) is a parallel node
   (a) `RESTRUCT` each child of \( N \)
3. else if \( N \) is a series node
   (a) create a new series node \( S \)
   (b) while \( N \) has children
       i. \( child \leftarrow \) remove leftmost child of \( N \)
       ii. append \( child \) to \( S \)
       iii. `SLIDE` \( child \) left
   (c) \( N \leftarrow S \)
4. mark \( N \) clean
The Restruct Algorithm

The \texttt{Restruct} algorithm traverses the SP-tree (depth-first) from node $N$, revisiting when necessary (all series and parallel nodes are initially marked \textit{dirty}).

1. if $N$ is a leaf or \textit{clean} (base case)
   (a) return

2. else if $N$ is a parallel node
   (a) \texttt{Restruct} each child of $N$

3. else if $N$ is a series node
   (a) create a new series node $S$
   (b) while $N$ has children
       i. \textit{child} $\leftarrow$ remove leftmost child of $N$
       ii. append \textit{child} to $S$
       iii. \texttt{Slide} \textit{child} left
   (c) $N \leftarrow S$

4. mark $N$ \textit{clean}
Swapping Ships

Conditions for swapping two series-connected ships (labeled $A$ and $B$)

- $A$ and $B$ are \textit{commutative} ($A$ or $B$ is request-equivalent and $A$ or $B$ is data-equivalent)
- swapping $A$ and $B$ is \textit{beneficial} to the application (see next slide), and
- the graph resulting from a swap is an SP-DAG (we allow four configurations).
Swapping Ships

Conditions for swapping two series-connected ships (labeled $A$ and $B$)

- $A$ and $B$ are *commutative* ($A$ or $B$ is request-equivalent and $A$ or $B$ is data-equivalent)
- swapping $A$ and $B$ is *beneficial* to the application (see next slide), and
- the graph resulting from a swap is an SP-DAG (we allow four configurations).

(A) Non-structural, (B) Non-structural

\[ A \rightarrow B \quad S \]
\[ S \quad A \rightarrow B \quad S \quad B \rightarrow A \]

\[ S \quad A \rightarrow B \quad S \quad B \rightarrow A \]
Swapping Ships

Conditions for swapping two series-connected ships (labeled $A$ and $B$)

- $A$ and $B$ are **commutative** ($A$ or $B$ is request-equivalent and $A$ or $B$ is data-equivalent)
- swapping $A$ and $B$ is **beneficial** to the application (see next slide), and
- the graph resulting from a swap is an SP-DAG (we allow four configurations).

(A) Non-structural, (B) Distribution, Parallel node

PARALLELIZE right
Swapping Ships

Conditions for swapping two series-connected ships (labeled $A$ and $B$)

- $A$ and $B$ are *commutative* ($A$ or $B$ is request-equivalent and $A$ or $B$ is data-equivalent)
- swapping $A$ and $B$ is *beneficial* to the application (see next slide), and
- the graph resulting from a swap is an SP-DAG (we allow four configurations).

Parallel node, $(A)$ Merge, $(B)$ Non-structural

Parallellize left
Swapping Ships

Conditions for swapping two series-connected ships (labeled $A$ and $B$)

- $A$ and $B$ are *commutative* ($A$ or $B$ is request-equivalent and $A$ or $B$ is data-equivalent)
- swapping $A$ and $B$ is *beneficial* to the application (see next slide), and
- the graph resulting from a swap is an SP-DAG (we allow four configurations).

Parallel node, ($A$) Merge, ($B$) Distrib, Parallel node

PARALLELIZE right and left
A swap is deemed beneficial if it increases parallelism, moves a data-reducing ship closer to the data source, or moves a data-increasing ship closer to data destination.

Algorithm to decide a beneficial swap of adjacent ships $A$ and $B$

1. Assign a preferred direction to each ship (1 for right, $-1$ for left, or 0)
   - Merge ships prefer to go right (increase parallelism)
   - Distribution ships prefer to go left (increase parallelism)
   - Data-reducing ships prefer to swap toward the data source
   - Data-increasing ships prefer to swap toward the data destination

2. return $true$ if preferred direction of $A$ is greater than preferred direction of $B$

3. else return $false$
Restructuring the Example Graph
Restructuring the Example Graph
Restructuring the Example Graph
Restructuring the Example Graph
Restructuring the Example Graph

[Diagram of a graph with nodes labeled S, P, M, rep, dist, and edges indicating relationships between them.]

Armada – p.19
Restructuring the Example Graph
Restructuring the Example Graph
Placement

Hierarchical graph partitioning

1. Partition the ships into $k$ sets (each set represents an administrative domain).
2. Partition the ships within each domain to processors provided by domain-level schedulers.

**The Graph Partitioning Problem**

Given graph $G(V, E)$ with weighted vertices and weighted edges, partition the vertices into $k$ sets in such a way to balance the sum of the vertices and to minimize the weights of the edge crossings between sets (NP-hard [Garey et al., 1976]).
Partitioning an Armada Graph

*Chaco Graph Partitioning Software* [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

1. Construct model from SP-tree
   (a) Assign edge weights
   (b) Assign vertex weights
2. partition graph (using *CHACO*)
3. for each domain
   (a) request procs from domain
   (b) partition sub-graph
Algorithm for placement of Armada ships

1. Construct model from SP-tree
   (a) Assign edge weights
   (b) Assign vertex weights
2. partition graph (using CHACO)
3. for each domain
   (a) request procs from domain
   (b) partition sub-graph

*Chaco Graph Partitioning Software* [Hendrickson and Leland, SNL]
Partitioning an Armada Graph

*Chaco Graph Partitioning Software* [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

1. Construct model from SP-tree
   (a) Assign edge weights
   (b) Assign vertex weights
2. partition graph (using CHACO)
3. for each domain
   (a) request procs from domain
   (b) partition sub-graph
Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

1. Construct model from SP-tree
   (a) Assign edge weights
   (b) Assign vertex weights
2. partition graph (using CHACO)
3. for each domain
   (a) request procs from domain
   (b) partition sub-graph
Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

1. Construct model from SP-tree
   (a) Assign edge weights
   (b) Assign vertex weights
2. partition graph (using CHACO)
3. for each domain
   (a) request procs from domain
   (b) partition sub-graph
Experiments

Evaluate performance benefit of restructuring and placement

- Representative application
  - Placement considerations
- File copy and permutation
  - Third-party transfers
  - Data permutations
  - Number of processors required by Armada
- Seismic processing
  - C++ interface
  - Recursive filter
  - Latency effects
Representative Application

Examined four configurations of the example application with a filter that removed exactly 50% of the data.

(a) orig1

(b) orig2

(c) restruct1

(d) restruct2
Experiment Setup

The area between the blobs represents the WAN

- each LAN connected to the WAN by single router
- each WAN link has limited capacity

Ran experiments on the Emulab Network Testbed

- Three LANs, each with...
  - Five 850 MHz Pentium III processors
  - 100 Mbps switched network (0.15 msec latency)
- WAN consisted of...
  - Three network links with 2.0 msec latency
  - Bandwidth ranged from 2 to 100 Mbps
Results: Effective Throughput

![Graph showing effective throughput vs. total client/server WAN bandwidth. The graph includes lines for orig1, orig2, restruct1, restruct2, WAN bandwidth, and 2*WAN bandwidth. The x-axis represents the total client/server WAN bandwidth in Mbit/sec, and the y-axis represents effective throughput in Mbit/sec.]
Results: Effective Throughput

![Graph showing effective throughput versus total client/server WAN bandwidth. The graph includes lines for orig1, orig2, restruct1, restruct2, WAN bandwidth, and 2*WAN bandwidth.](Armada -- p.25)
Results: Effective Throughput

Graph showing the effective throughput vs. total client/server WAN bandwidth. The graph includes lines for orig1, orig2, restruct1, and restruct2, as well as markers for WAN bandwidth and 2*WAN bandwidth.
Results: Effective Throughput

- **orig1**
- **orig2**
- **restruct1**
- **restruct2**
- **WAN bandwidth**
- **2*WAN bandwidth**

Total client/server WAN bandwidth (Mbit/sec)

Effective Throughput (Mbit/sec)
Results: Effective Throughput

- Effective Throughput (Mbit/sec)
- Total client/server WAN bandwidth (Mbit/sec)

Legend:
- orig1
- orig2
- restruct1
- restruct2
- WAN bandwidth
- 2*WAN bandwidth
Copy distributed file from lan1 to distributed file on lan0.
File Copy and Permutation

Copy distributed file from lan1 to distributed file on lan0.

Input file

Output file
File Copy and Permutation

Copy distributed file from lan1 to distributed file on lan0.
File Copy and Permutation

Copy distributed file from lan1 to distributed file on lan0.
Results (effective throughput)

![Graph showing the relationship between WAN bandwidth and effective throughput for different file formats: Single-file, Original, and Restructured. The graph compares the effective throughput (Mbit/sec) against the WAN bandwidth (Mbit/sec).]
Results (different placements)

![Graph showing the relationship between WAN bandwidth and Effective Throughput for different compressed np settings. The graph includes lines for Compressed np=0, Compressed np=1, Compressed np=all, and WAN bandwidth.]
Post-Stack Seismic Imaging

Properties of seismic processing

- Compute intensive
- Large (terabyte) data sets
  - Collections of files (> 1K)
  - Each file contains a set of *traces* (recorded pressure waves)
- Preprocessing
  - *Stack* co-located traces
  - FFT time traces
  - Distribute frequencies to compute nodes
Connect with the data provider and describe compute node distribution

// called by all nodes,
// ... node0 gets graph from data provider
// ... constructor decomposes data (3D block decomposition)
TraceDataset dataset(comm, pmesh, providerURL)
Connect with the data provider and describe compute node distribution

// called by all nodes,
// ... node0 gets graph from data provider
// ... constructor decomposes data (3D block decomposition)
TraceDataset dataset(comm, pmesh, providerURL)
Constructing the Armada Graph

Append operators.

// called by node0
dataset.appendOp(new FFTOp());
Constructing the Armada Graph

Append operators.

```
// called by node0
dataset.appendOp(new FFTOp());
dataset.appendOp(new StackOp());
```
Constructing the Armada Graph

Append operators.

```java
// called by node0
dataset.appendOp(new FFTOp());
dataset.appendOp(new StackOp());
```
Restructure and Deploy the Armada graph.

// called by node0
dataset.open();
Restructure and Deploy the Armada graph.

```c
// called by node0
dataset.open();
// ... connect app-specific with data-provider portion
```
Restructure and Deploy the Armada graph.

// called by node0
dataset.open();
// ... connect app-specific with data-provider portion
// ... restructure graph
Restructure and Deploy the Armada graph.

```c
// called by node0
dataset.open();
// ... construct entire Armada graph
// ... restructure graph
// ... assign placement
```
Restructure and Deploy the Armada graph.

```
// called by node0
dataset.open();
// ... construct entire Armada graph
// ... restructure graph
// ... assign placement
// ... deploy
```
Collectively read dataset.

```cpp
// called by all procs
int size = dataset.getLocalSize();
float *data = new float[size];
dataset.read(data);

// do computation ...
```
Experiment Setup

Original

- Compute partition
- Storage servers
  - app-specific from data-provider

Restructured

- Compute partition
- Storage servers
  - From data-provider
Results (effective throughput)
Results (different latencies)
Related Work

Parallel processing of I/O streams

- **PS$^2$** [Messerli, 1999]
  - data-flow model with automatic parallelization
- **DataCutter** [Spencer et al., 2002]
  - component-based, analytic model to decide parallelization

*Armada does not force the whole application into a data-flow model*

*Armada widens data flow for parallel clients and parallel servers*

Operation re-ordering to improve data flow, e.g., in databases

- **dQUOB** [plale et al. 2000]
  - optimize query tree to move high-filtering portions close to data
  - exploit well-defined properties associated with query processing

*Armada provides a more general approach*
Future Work

Other Applications

- fMRI application (time-series analysis of brain data)
- Can components be reused between applications?

Modifications to **BENEFICIAL** and **COMMUTATIVE**

- Non-greedy methods
- Analytic models to approximate benefit

Placement

- incorporate domain-specific information into the partitioner (compute capacity, memory capacity, etc...)
- dynamic re-deployment when network conditions change

Tuning for cluster computing (in addition to the grid)
Summary

The Armada framework

- data provider can describe complex distributed data sets
- application describes processing required before computation
- data-flow model provides a “latency-tolerant” approach

Restructuring algorithm

- arranges graph to provide end-to-end parallel I/O
- enables effective placement of data-processing components

Placement

- domain assignments to minimize data flow.
- host assignments based on administrative domain policies.

Experiments demonstrate good performance in multiple environments.
Efficient I/O for Computational Grid Applications

Ron Oldfield

Department of Computer Science, Dartmouth College

http://www.cs.dartmouth.edu/~dfk/armada/

Supported by Sandia National Laboratories under contract DOE-AV6184.