High-Performance I/O for Computational Grid Applications

Ron Oldfield and David Kotz

{raoldfi,dfk}@cs.dartmouth.edu.

Department of Computer Science, Dartmouth College

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

Computational Grids

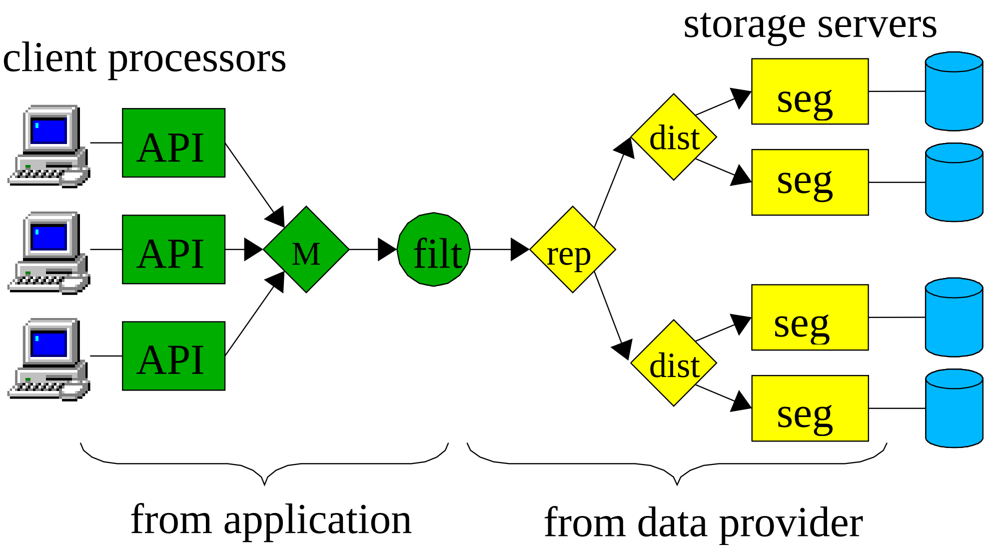
Networks of geographically distributed heterogeneous systems and devices

Data-intensive grid applications

- Access large remote datasets (terabyte–petabyte)
- Datasets often need pre/post-processesing
- Often computationally intensive
- Examples
  - Climate modeling
  - Astronomy
  - Computational Biology
  - High-energy physics

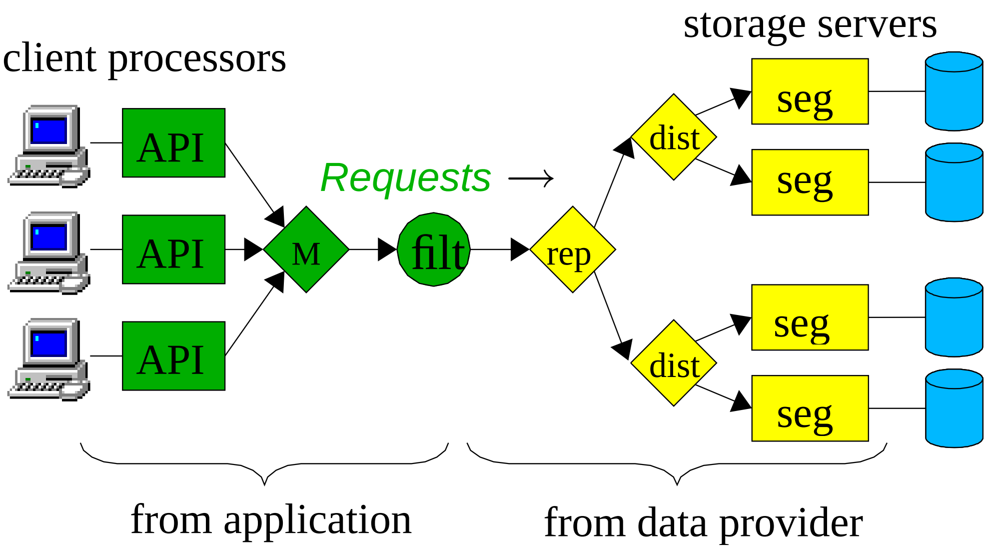
The Armada Framework

- 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)



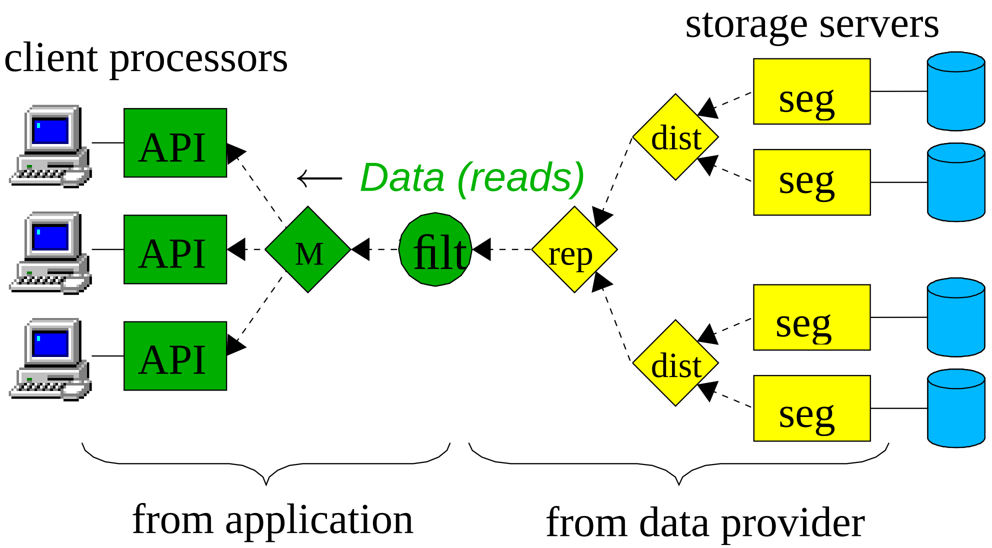
The Armada Framework

- 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)



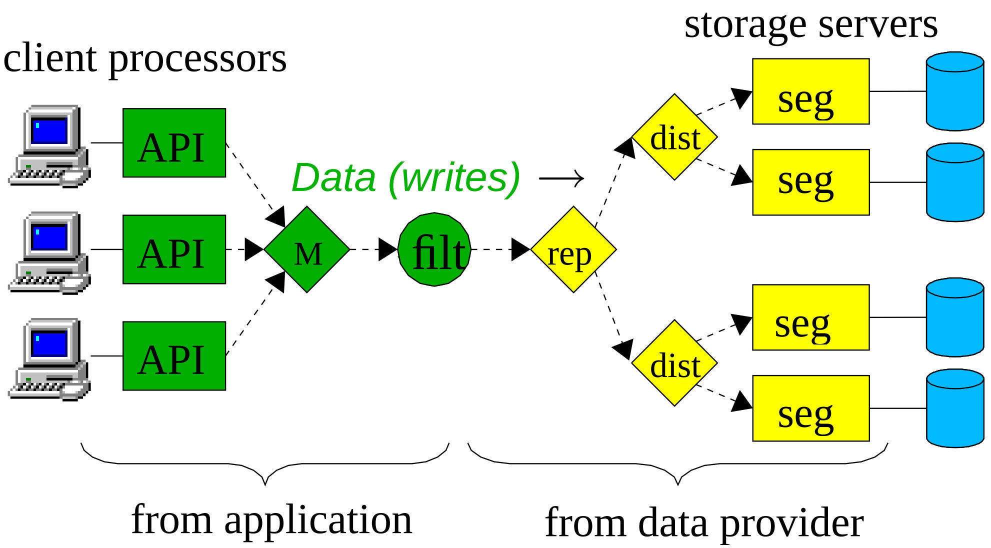
The Armada Framework

- 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...

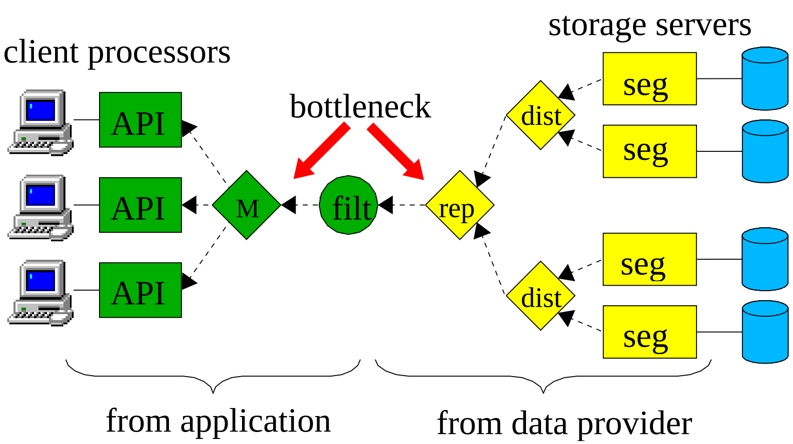
- build a graph for parallel access to a group of legacy files,
- present many similar data sets through a standard interface, and
- provide transparent access to derived “virtual” data– either cached or calculated as needed.

Restructuring

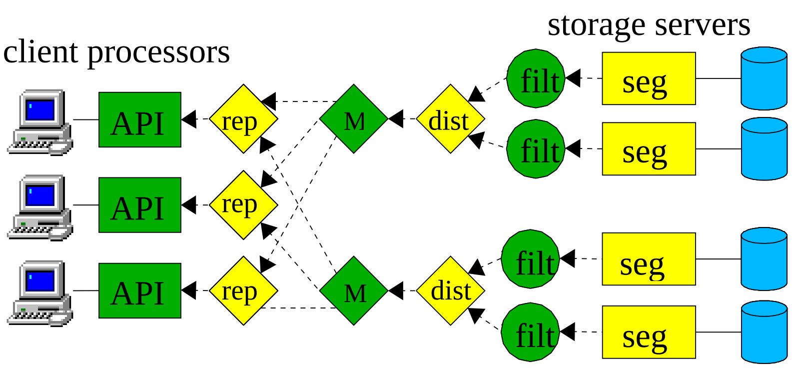
Problems with the example application:

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

Original graph



Restructured graph

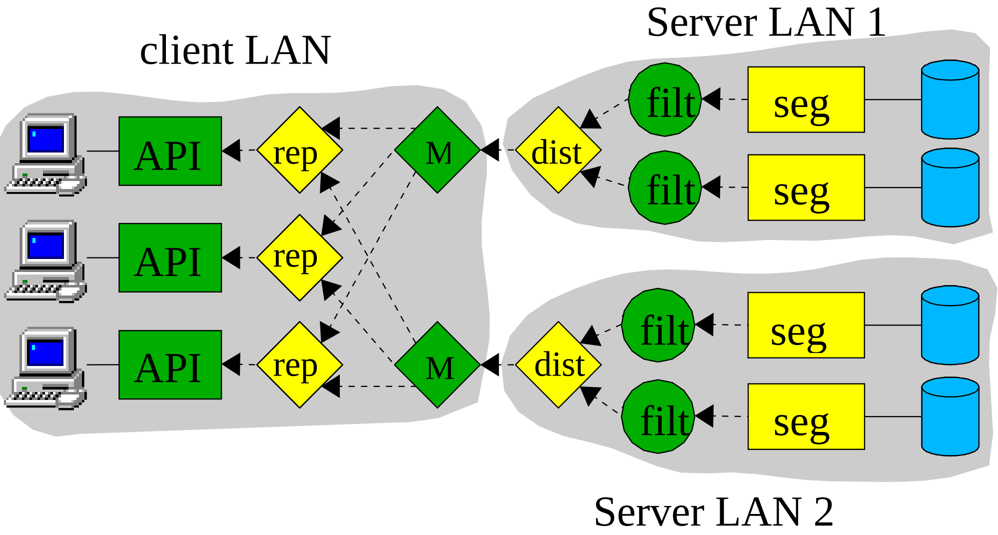


Armada restructures original graph to improve data flow.

Placement

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.

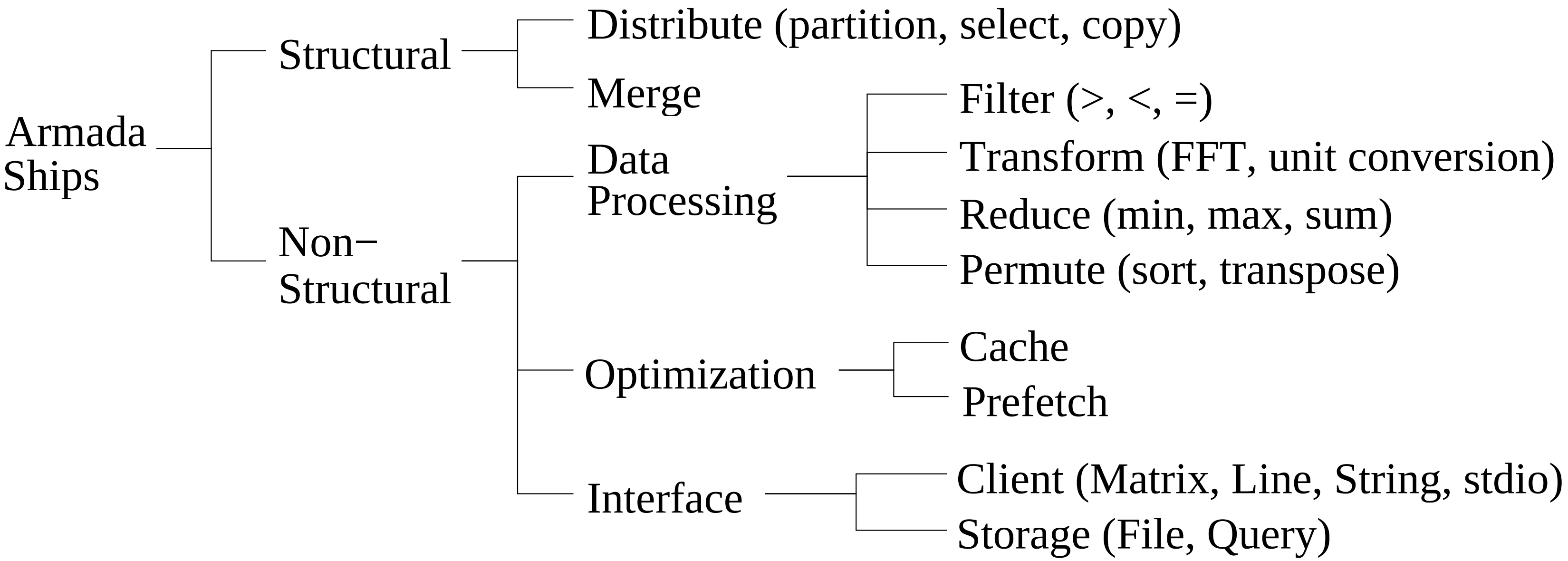


Talk Outline

- *Introduction*
- Framework details
  - Ships
  - Graph Representation
- Restructuring graphs to improve data flow
- Partitioning graphs and placing ships
- Experiments
- Conclusion

Ships

Armada includes a rich set of extensible ship classes.



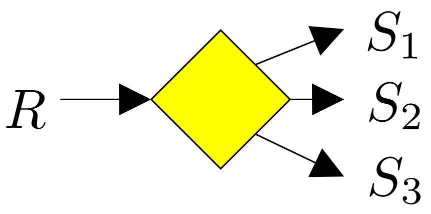
- Armada Ships
  - Structural
    - Distribute (partition, select, copy)
    - Merge
  - Non−Structural
    - Data Processing
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - Optimization
      - Cache
      - Prefetch
    - Interface
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

Ships

Armada includes a rich set of extensible ship classes.

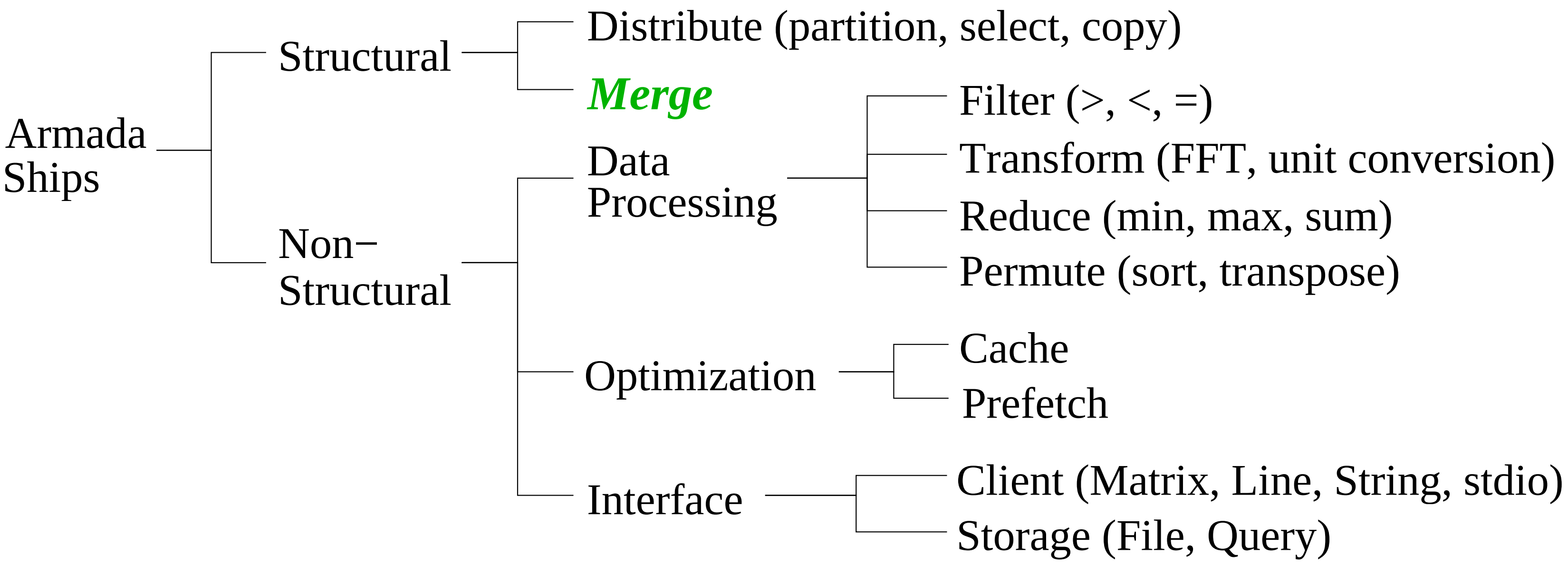
- Armada Ships
  - Structural
    - *Distribute (partition, select, copy)*
    - Merge
  - Non−Structural
    - Data Processing
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - Optimization
      - Cache
      - Prefetch
    - Interface
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

Distribute ships partition requests or data to multiple output streams.



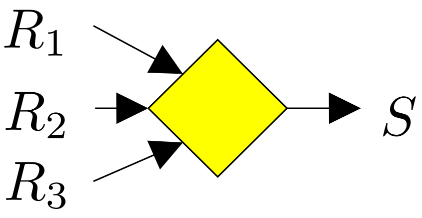
Ships

Armada includes a rich set of extensible ship classes.



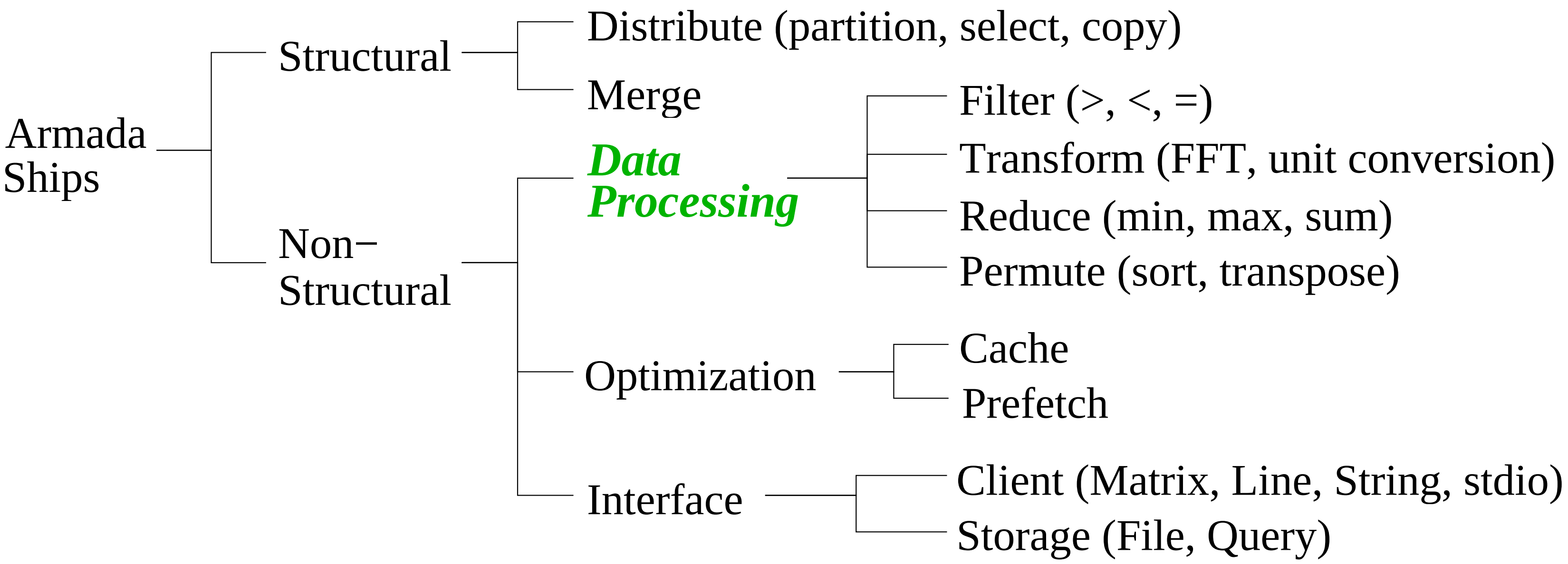
- Armada Ships
  - Structural
    - Distribute (partition, select, copy)
    - *Merge*
  - Non−Structural
    - Data Processing
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - Optimization
      - Cache
      - Prefetch
    - Interface
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

Merge ships interleave requests or data from multiple input streams.



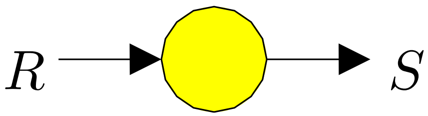
Ships

Armada includes a rich set of extensible ship classes.



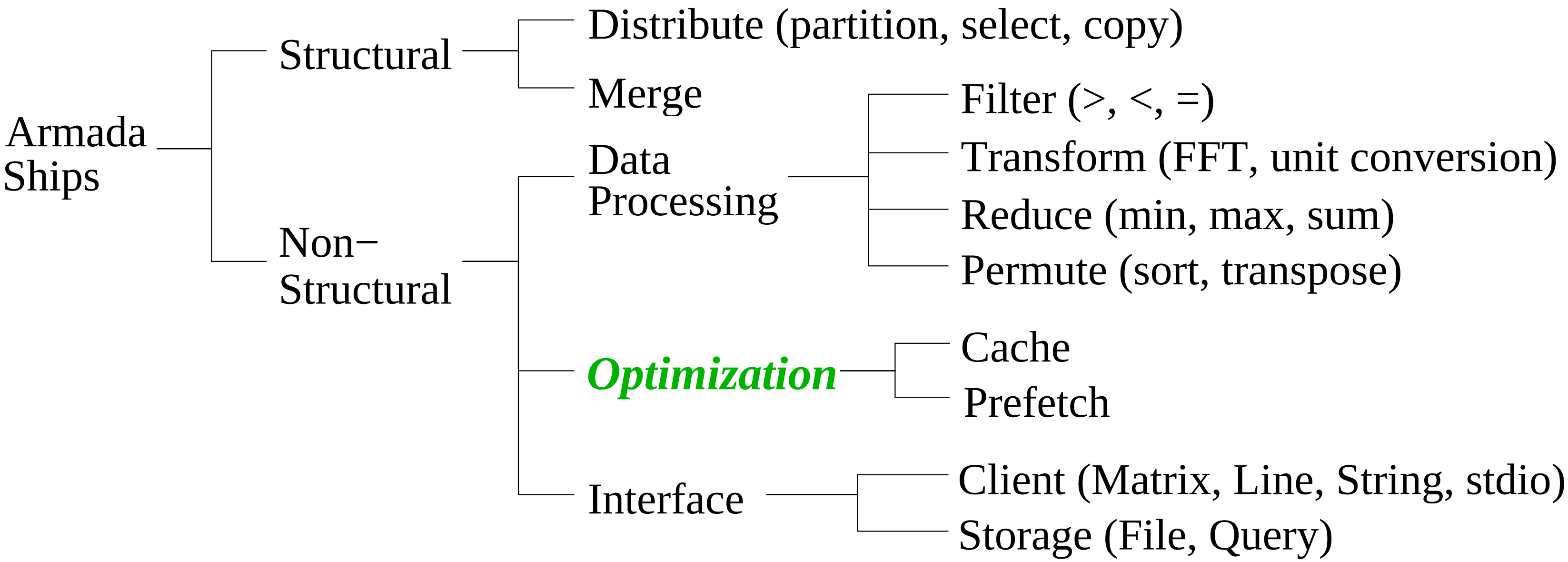
- Armada Ships
  - Structural
    - Distribute (partition, select, copy)
    - Merge
  - Non−Structural
    - ***Data Processing***
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - Optimization
      - Cache
      - Prefetch
    - Interface
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

Data-processing ships manipulate data, either individually, or in groups as it passes through the ship.



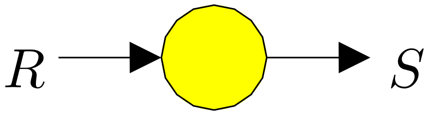
Ships

Armada includes a rich set of extensible ship classes.



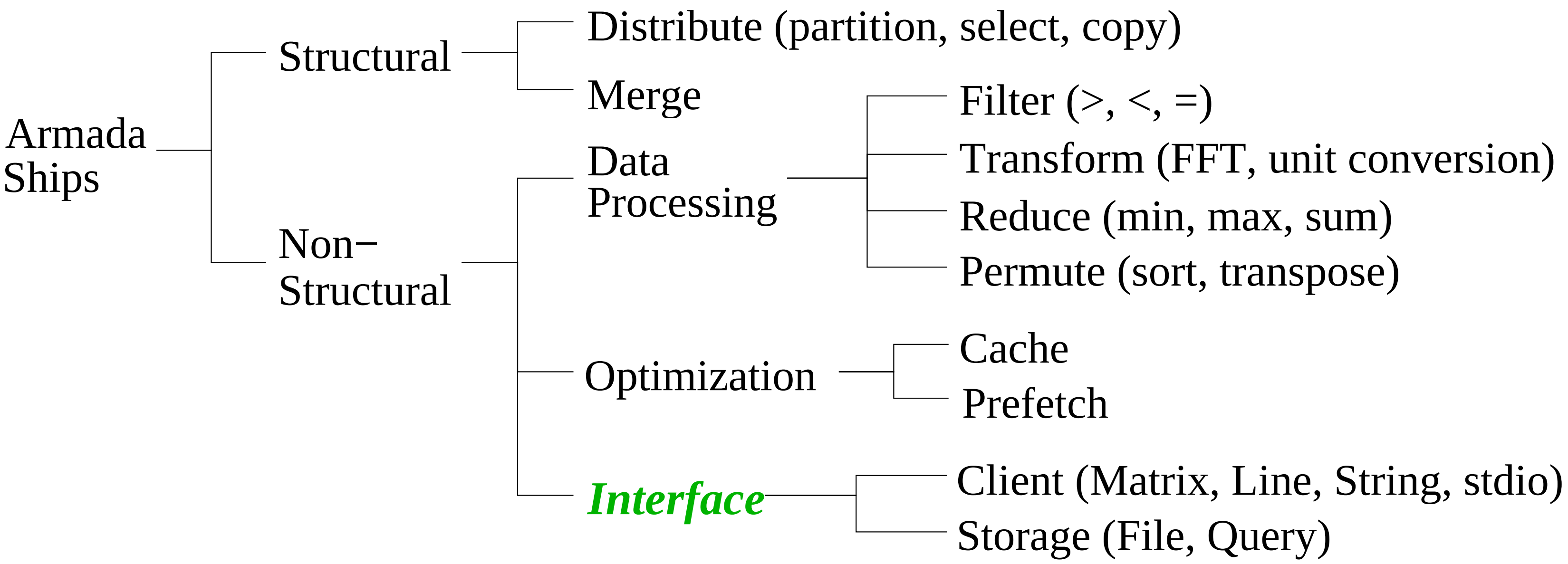
- Armada Ships
  - Structural
    - Distribute (partition, select, copy)
    - Merge
  - Non−Structural
    - Data Processing
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - *Optimization*
      - Cache
      - Prefetch
    - Interface
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

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



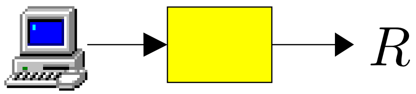
Ships

Armada includes a rich set of extensible ship classes.

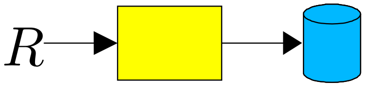


- Armada Ships
  - Structural
    - Distribute (partition, select, copy)
    - Merge
  - Non−Structural
    - Data Processing
      - Filter (>, <, =)
      - Transform (FFT, unit conversion)
      - Reduce (min, max, sum)
      - Permute (sort, transpose)
    - Optimization
      - Cache
      - Prefetch
    - ***Interface***
      - Client (Matrix, Line, String, stdio)
      - Storage (File, Query)

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:

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

In other words, order does not matter.

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.

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

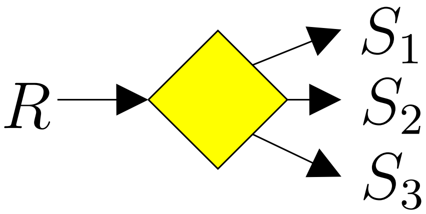
Most structural ships are both request and data-equivalent.

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.

Distribution ships partition requests or data

- S_1, S_2, and S_3 are subsequences of R.
- $R \equiv \{S_1, S_2, S_3\}$

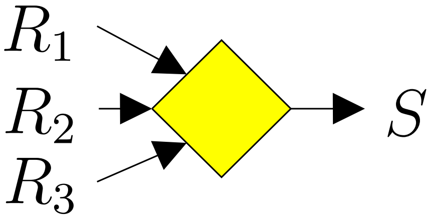


Request and Data Equivalent Ships

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.

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*

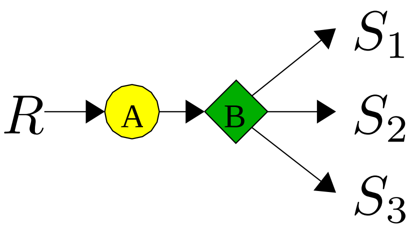
Parallelizable Ships

Parallelizable: a ship that can transform into multiple ships

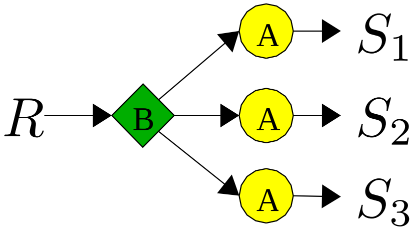
- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

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

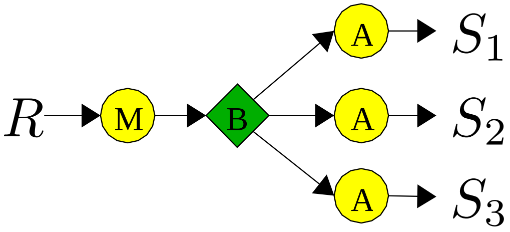
Right-parallelizable



Original



Replicated



Recursed

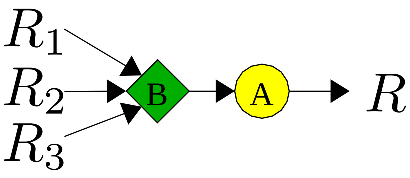
Parallelizable Ships

Parallelizable: a ship that can transform into multiple ships

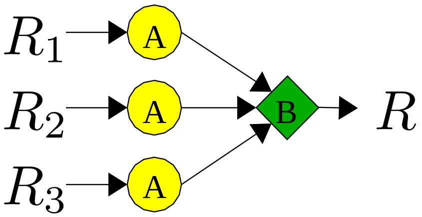
- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

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

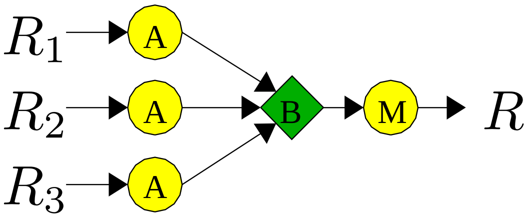
Left-parallelizable



Original



Replicated

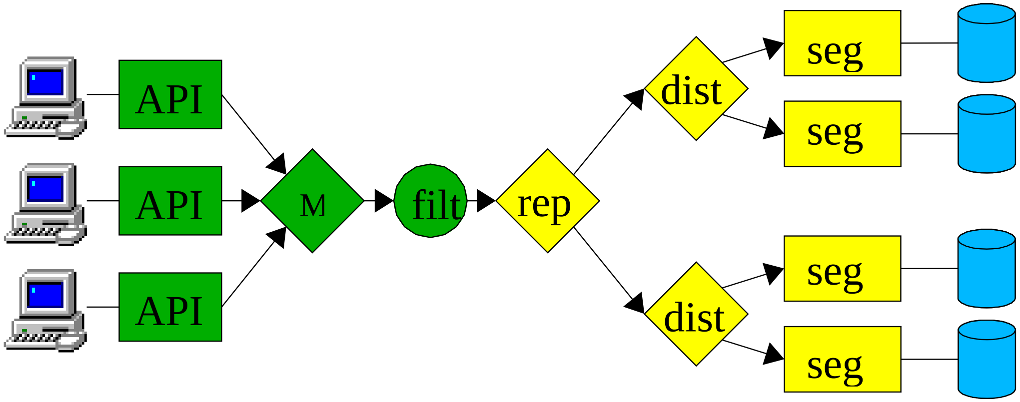


Recursed

Graph Representation

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

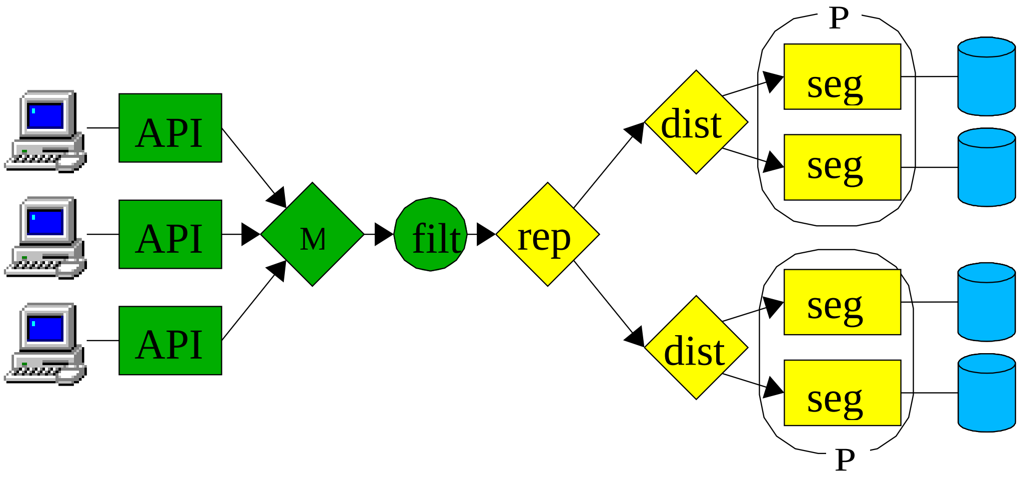
- Syntactically easy to describe (we use XML)
- Easy to manipulate internally

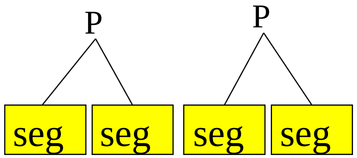


Graph Representation

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

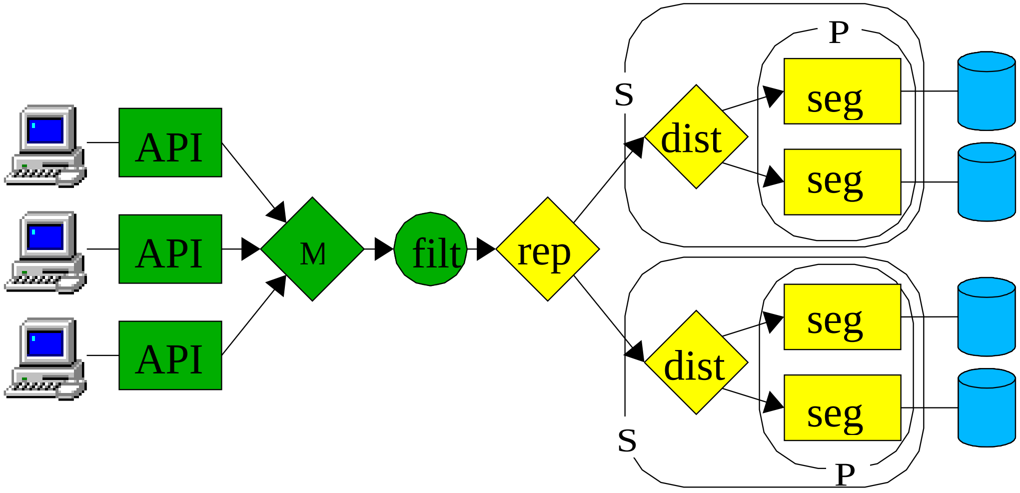


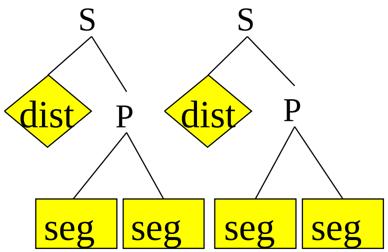


Graph Representation

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

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

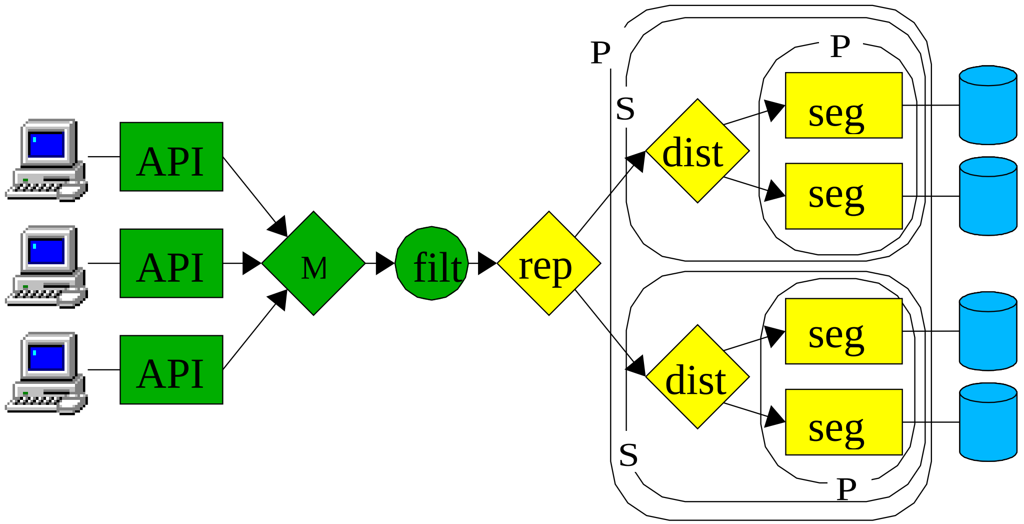


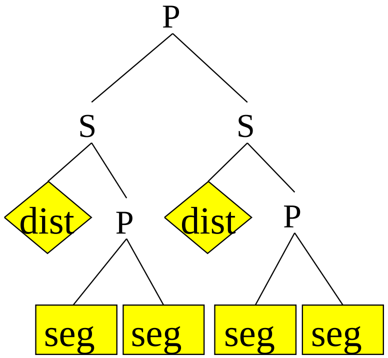


Graph Representation

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

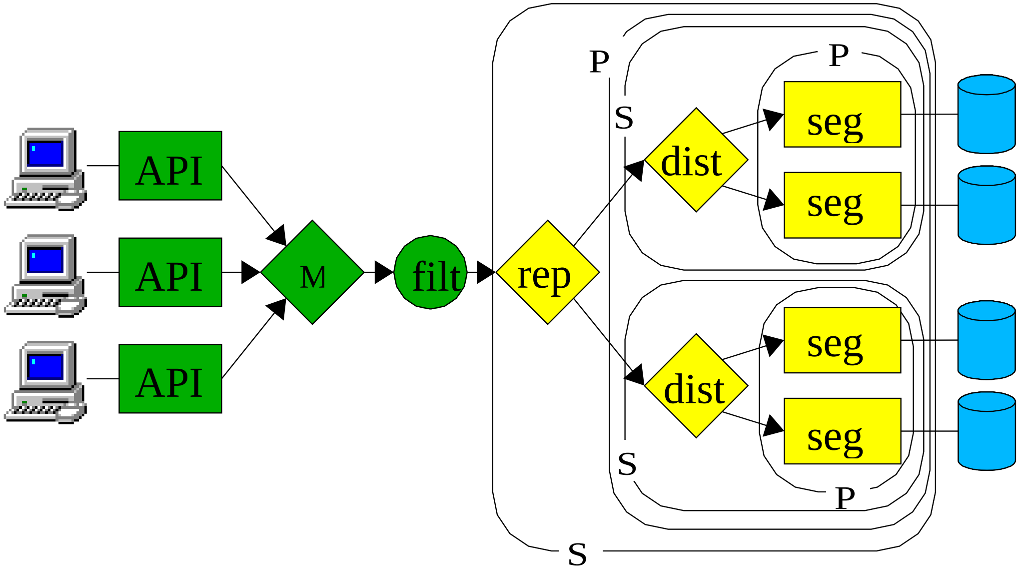


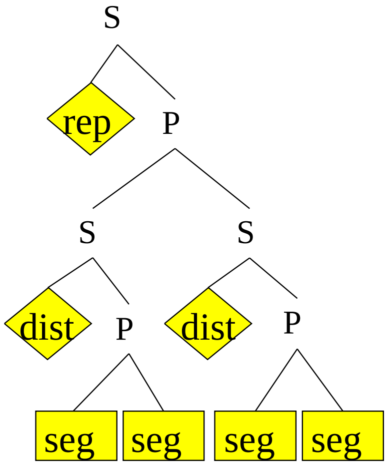


Graph Representation

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

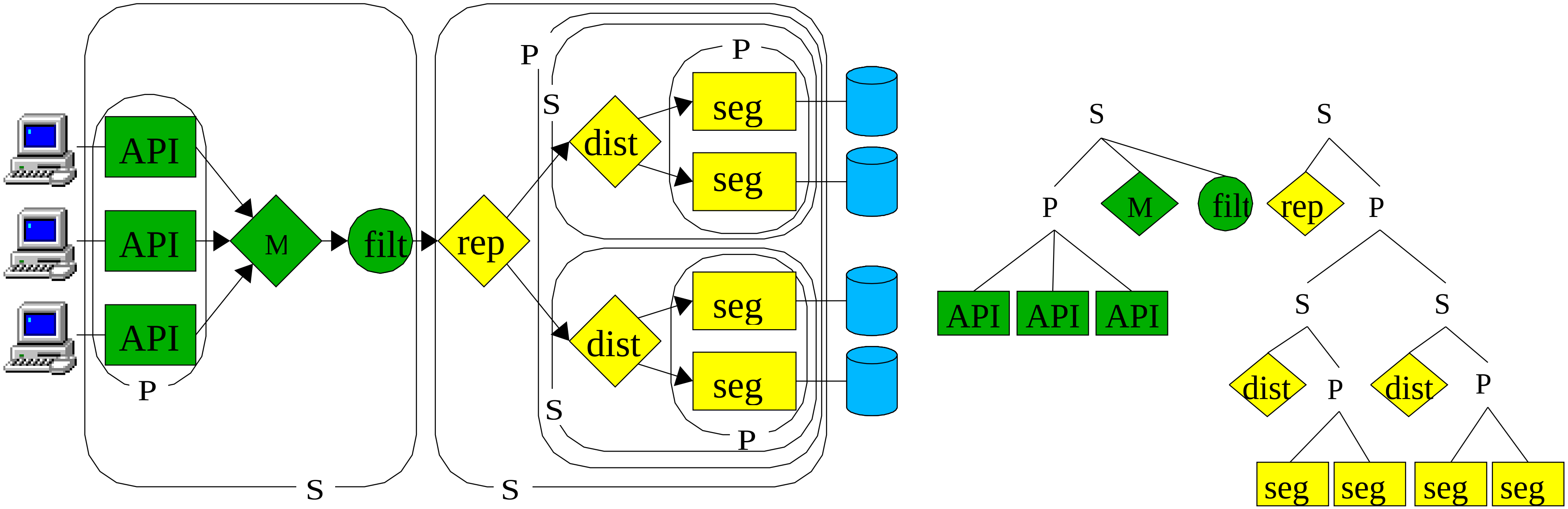




Graph Representation

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

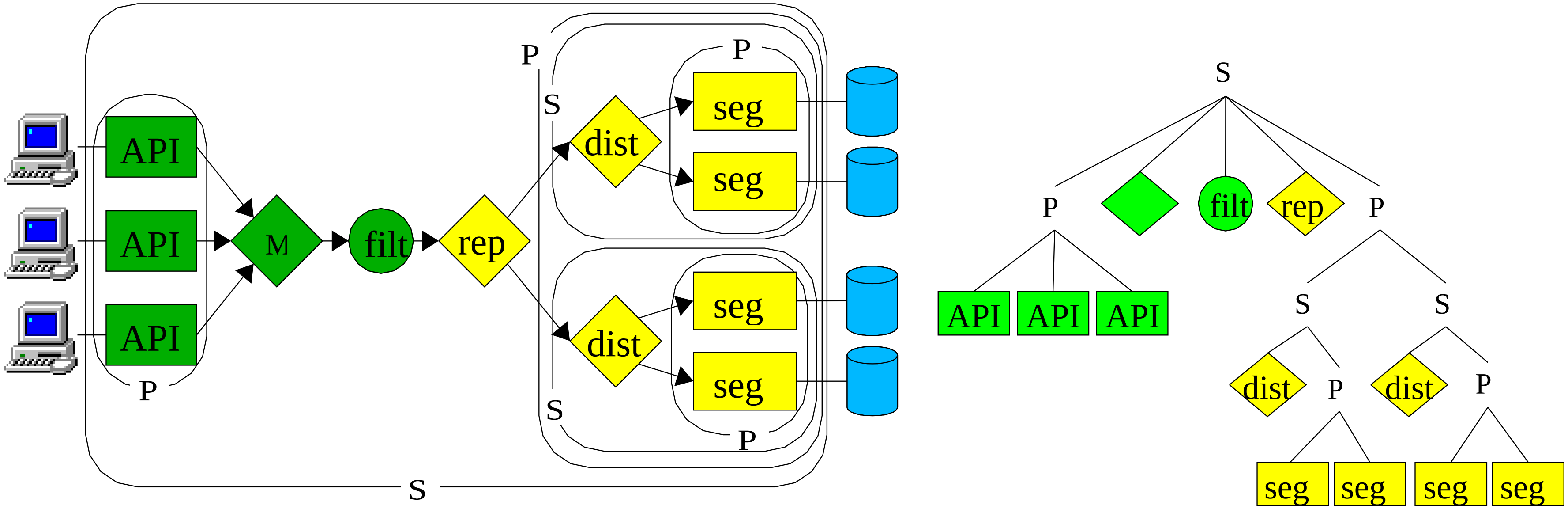
- Syntactically easy to describe (we use XML)
- Easy to manipulate internally



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*).

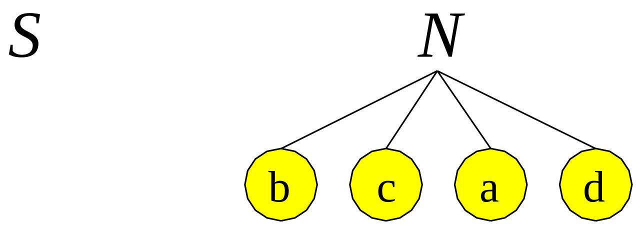
```
1. if $N$ is a leaf or $clean$ (base case)
    (a) return
2. else if $N$ is a parallel node
    (a) RESTRUCT each child of $N$
3. else if $N$ is a series node
    (a) create a new series node $S$
    (b) while $N$ has children
        i. $child \leftarrow$remove leftmost child of $N$
        ii. 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 $N$ $clean$
```

Append b to S



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 $N$ $clean$
```

Append b to S

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 $N$ $clean$
```

Append c to S

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 $N$ $clean$
```

Append c to S

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 $N$ $clean$
```

Slide c left

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 $N$ $clean$
```

Append a to S

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 $N$ $clean$
```

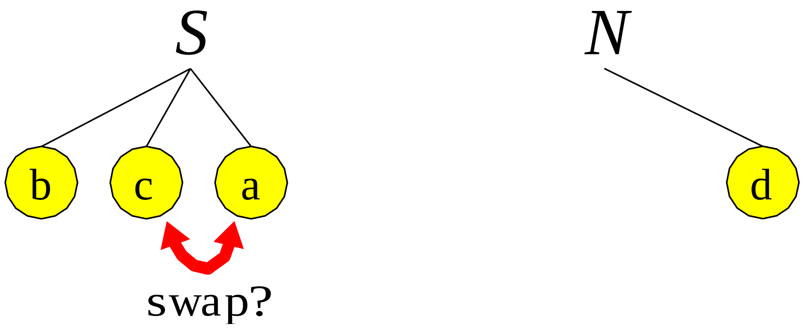
Append a to S

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 $N$ $clean$
```

Slide a left

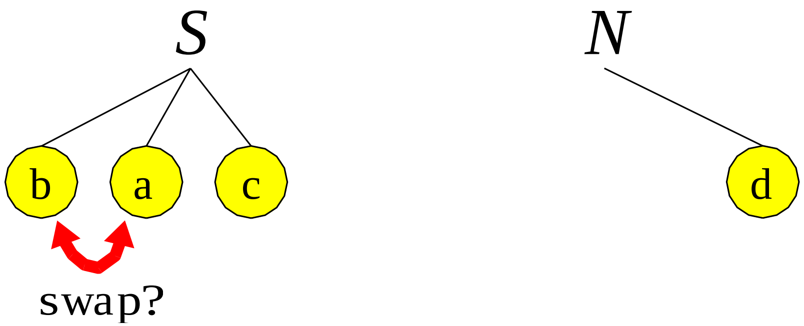


The Restruct Algorithm

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 $N$ $clean$
```

Slide a left



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 $N$ $clean$
```

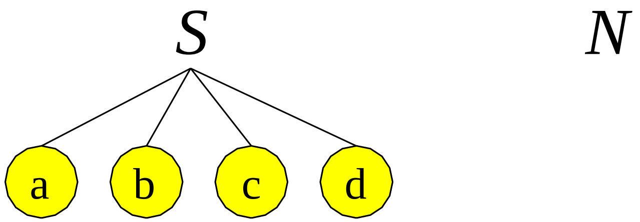
Append d to S

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 $N$ $clean$
```

Append d to S

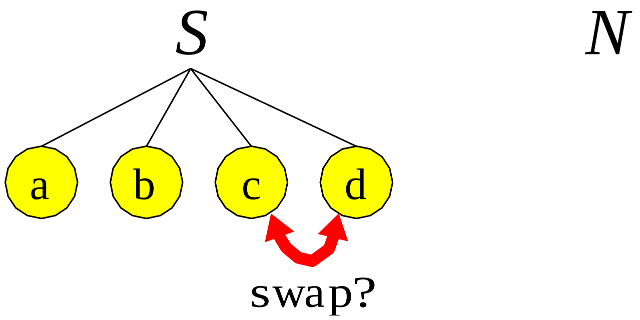


The Restruct Algorithm

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 $N$ $clean$
```

Slide d left

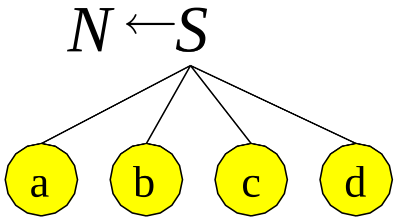


The Restruct Algorithm

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 $N$ $clean$
```

Assign S to N



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).

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).

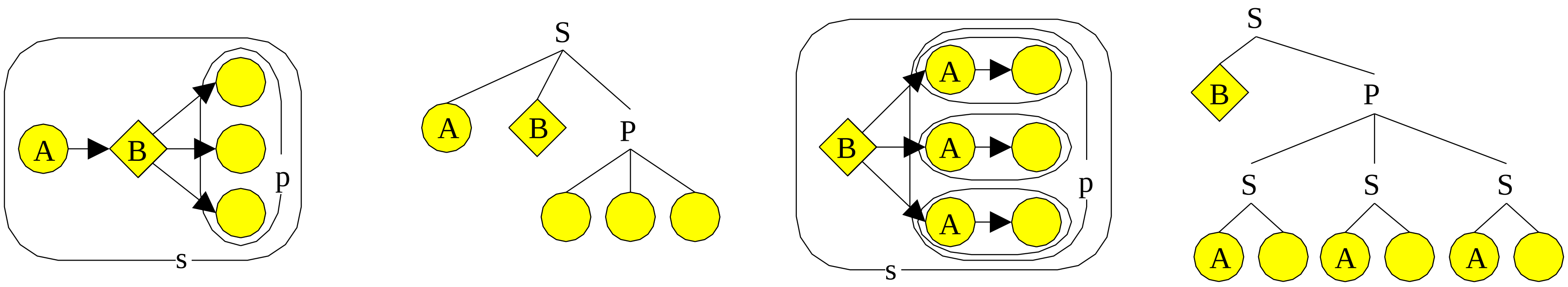
(A) Non-structural, (B) Non-structural

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



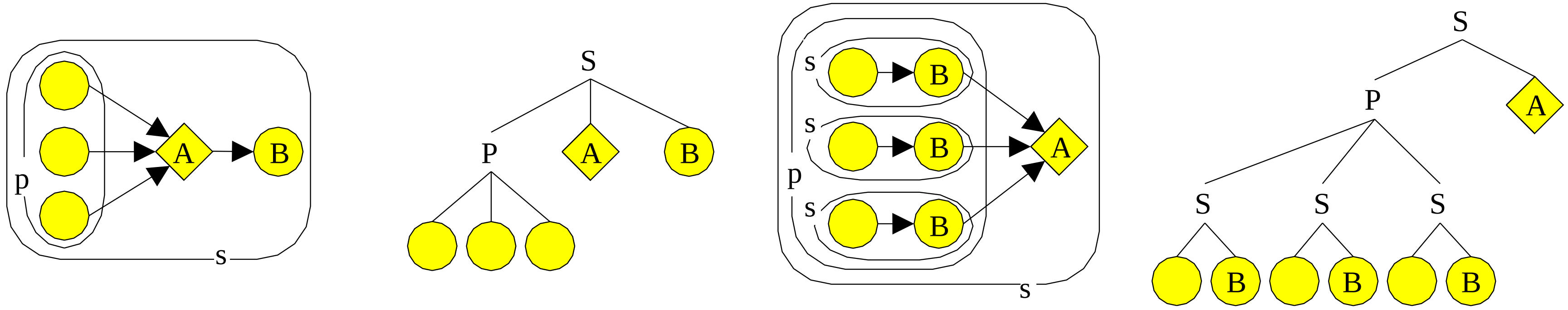
PARALLELIZE right

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



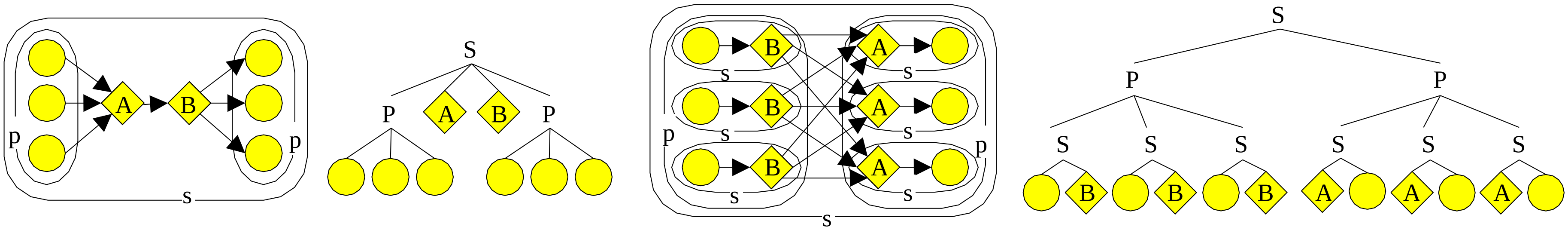
PARALLELIZE left

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

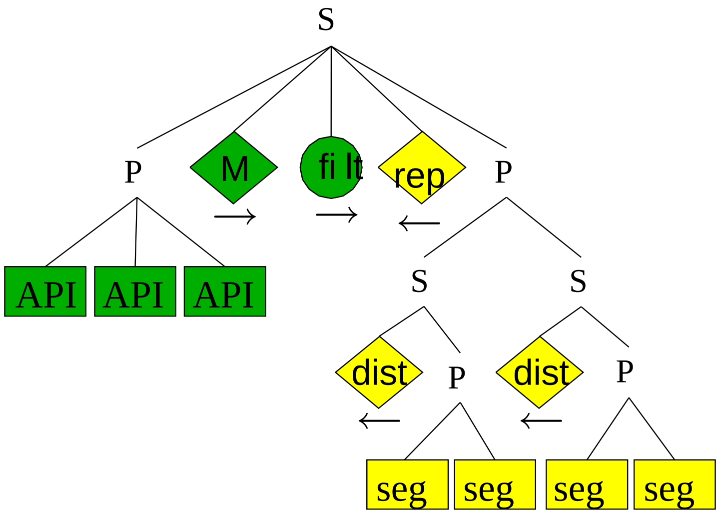
Beneficial Swap

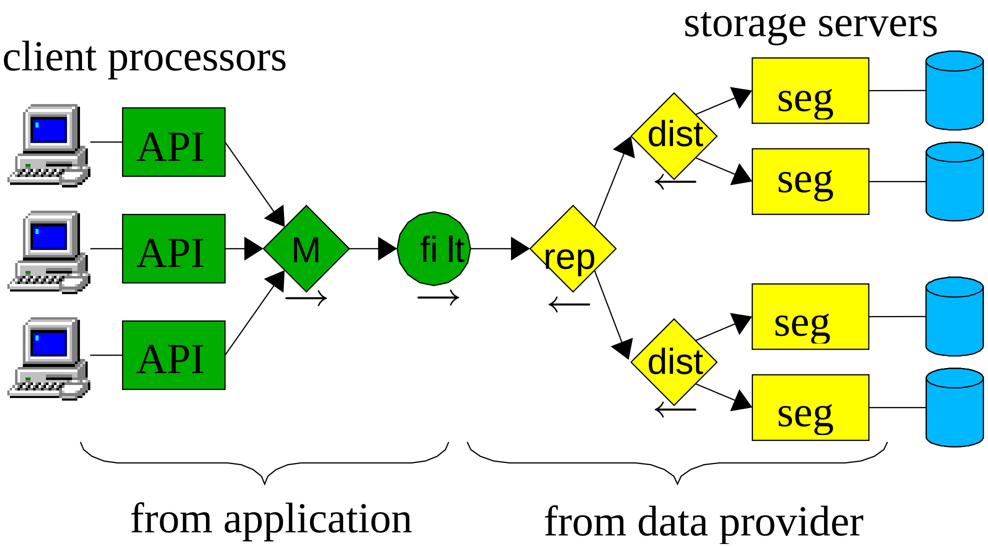
A swap is deemed *beneficial* if it increases parallelism, moves a data-reducing ship closer to the data source, or moves a data-increasing ship closer to data destination.

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

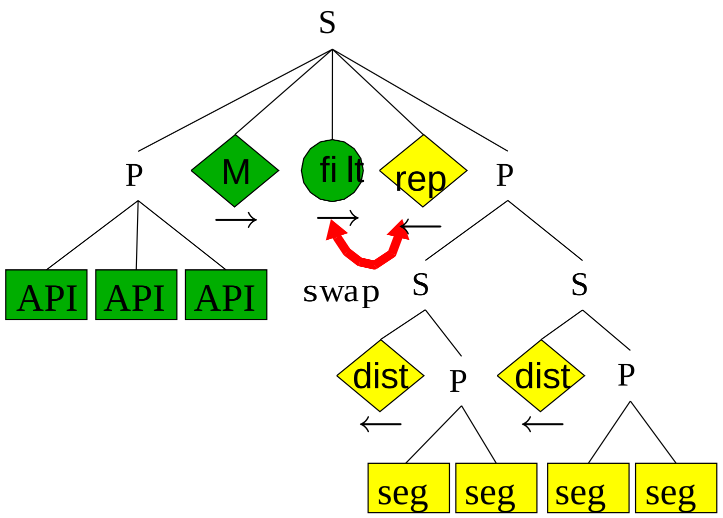
1. Assign a preferred direction to each ship (1 for right, -1 for left, or 0)
    - Merge ships prefer to go right (increase parallelism)
    - Distribution ships prefer to go left (increase parallelism)
    - Data-reducing ships prefer to swap toward the data destination
    - Data-increasing ships prefer to swap toward the data source
2. return $true$ if preferred direction of A is greater than preferred direction of B
3. else return $false$

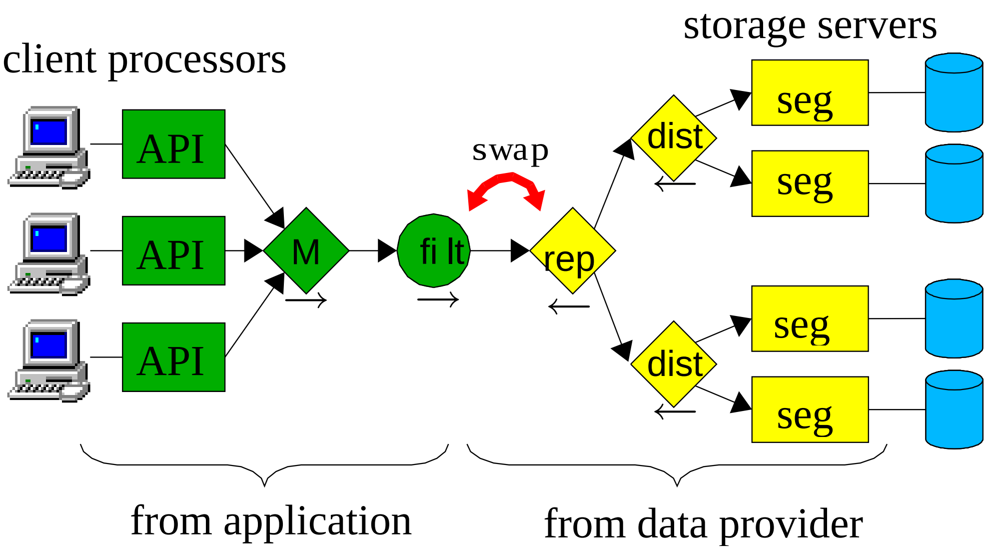
Restructuring the Example Graph



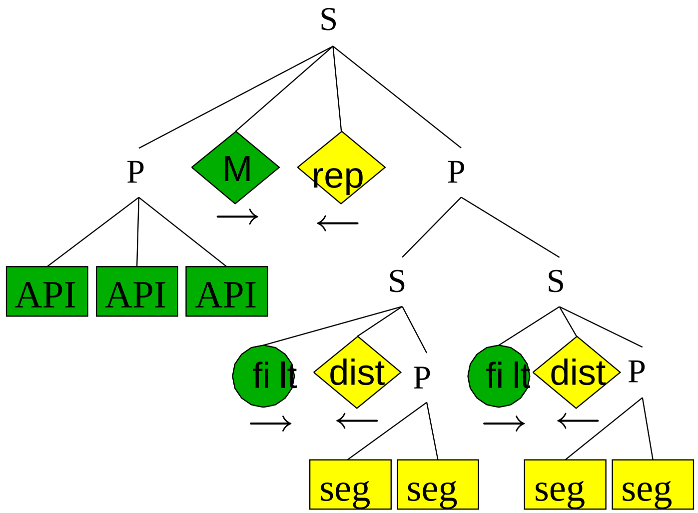


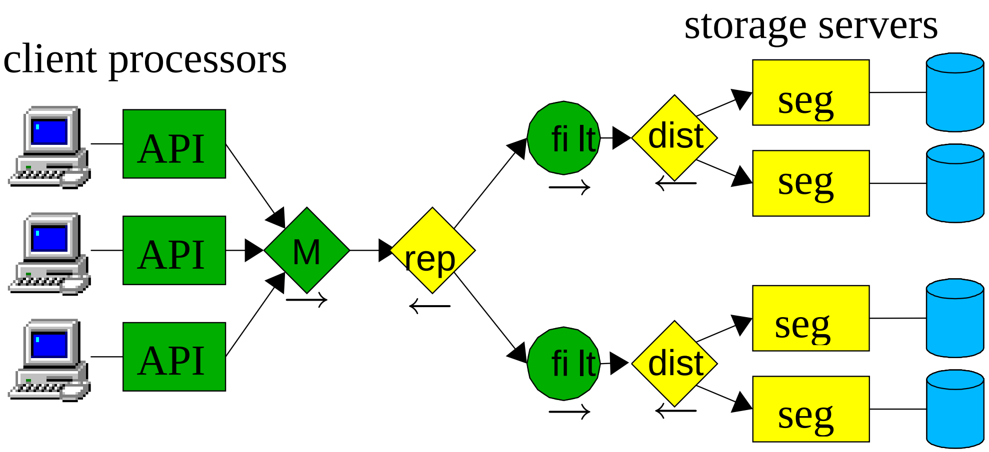
Restructuring the Example Graph



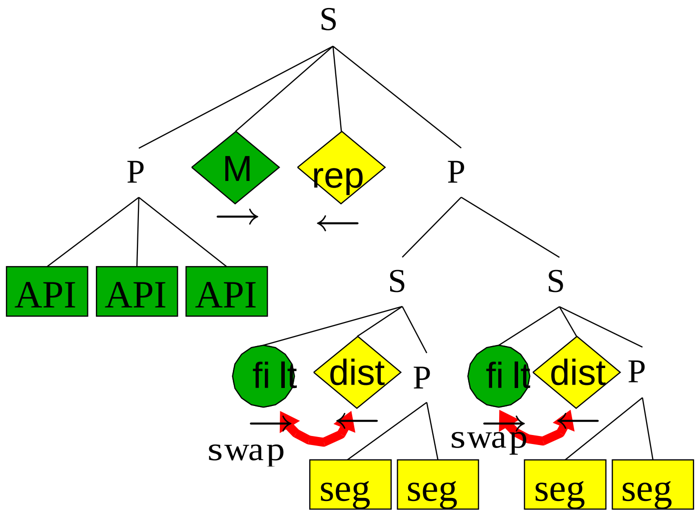


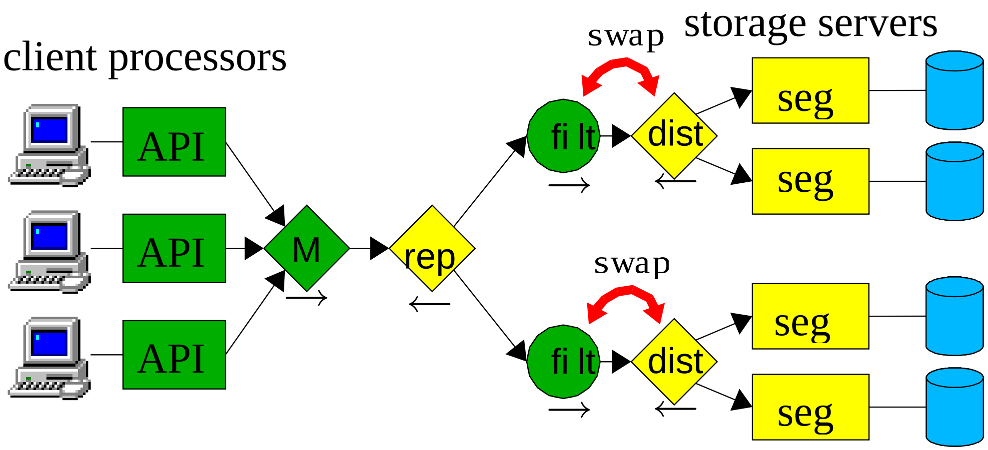
Restructuring the Example Graph



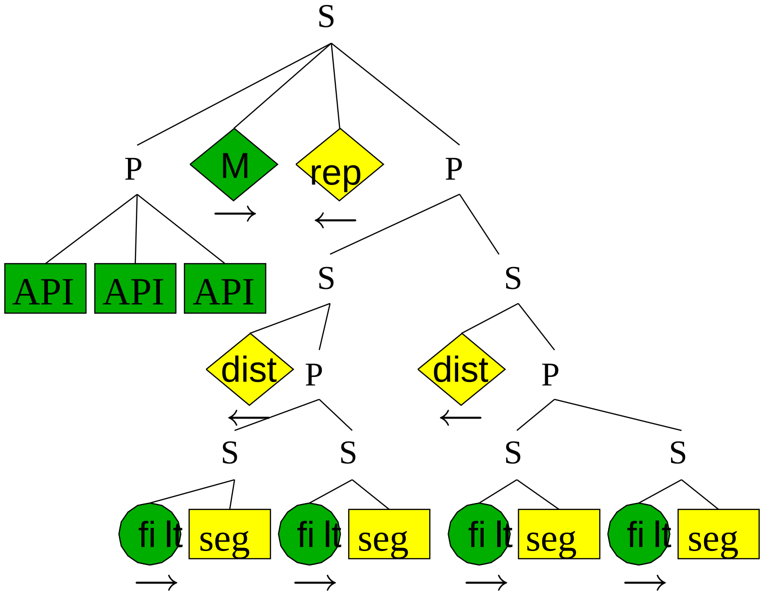


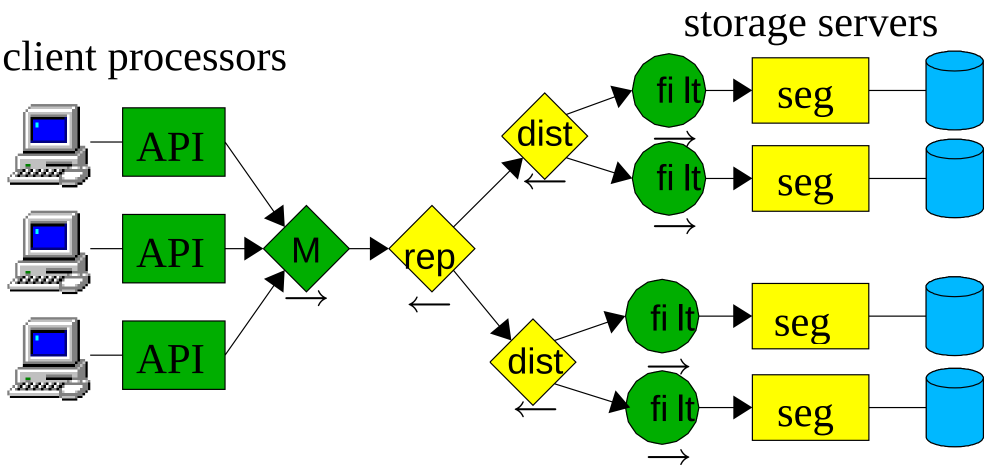
Restructuring the Example Graph



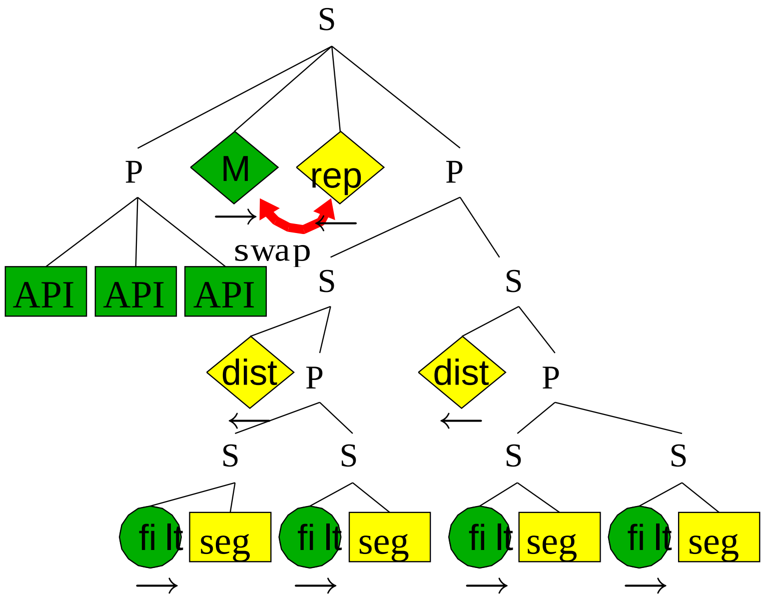


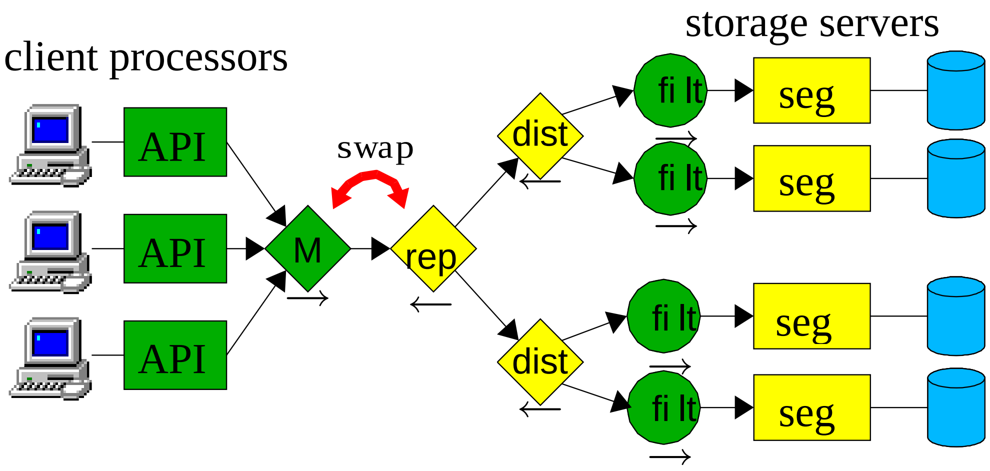
Restructuring the Example Graph



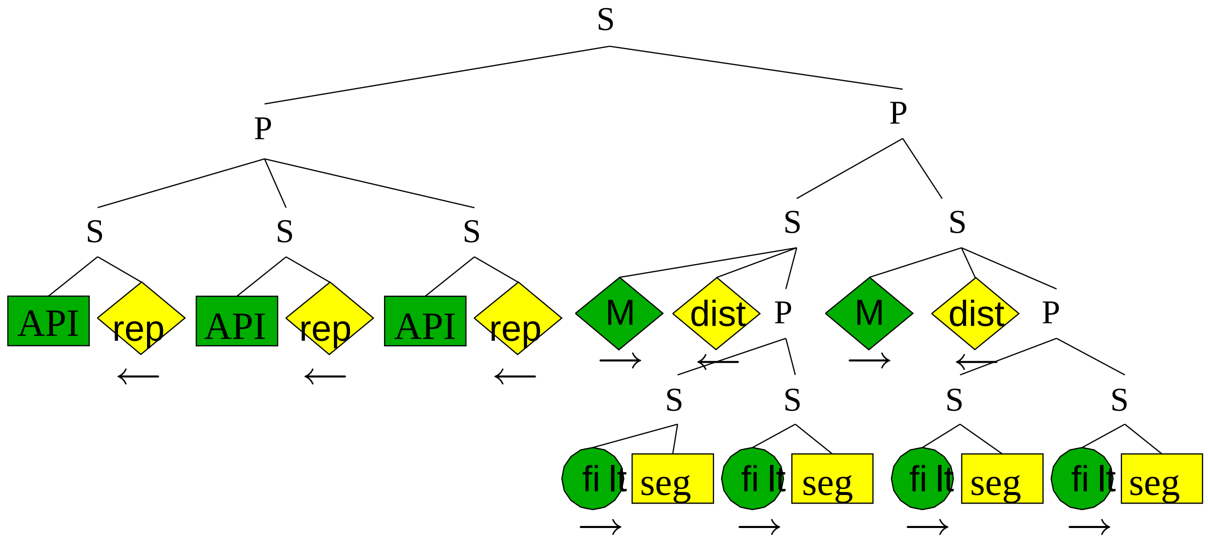


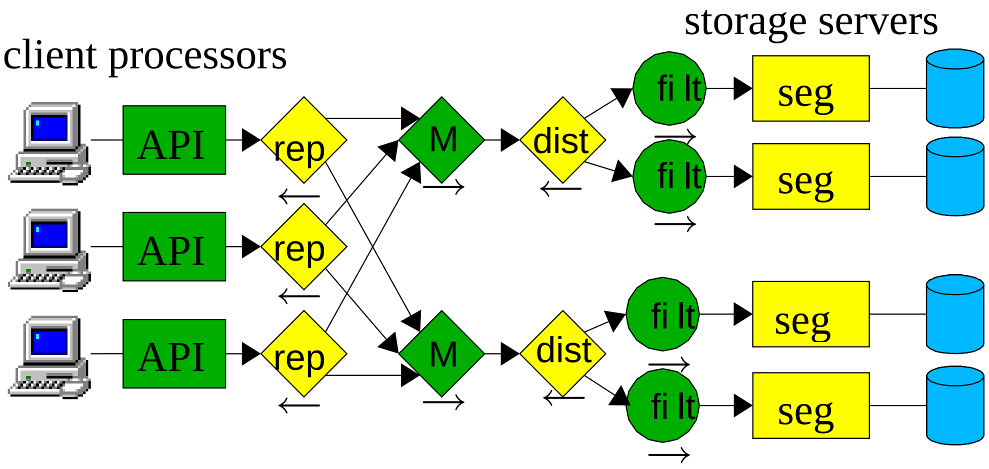
Restructuring the Example Graph





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

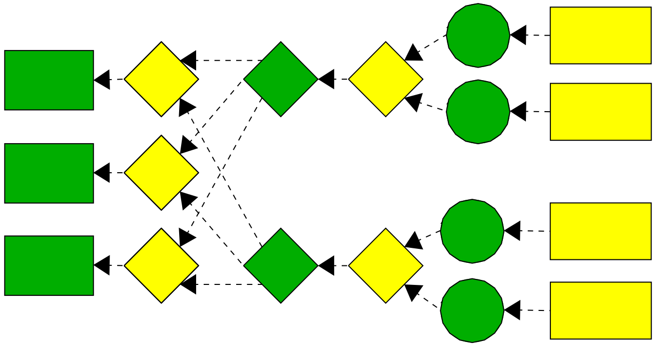
Given graph $G(V, E)$ with weighted vertices and weighted edges, partition the vertices into k sets in such a way to balance the sum of the vertices and to minimize the weights of the edge crossings between sets
(NP-hard [Garey et al., 1976]).

Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

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

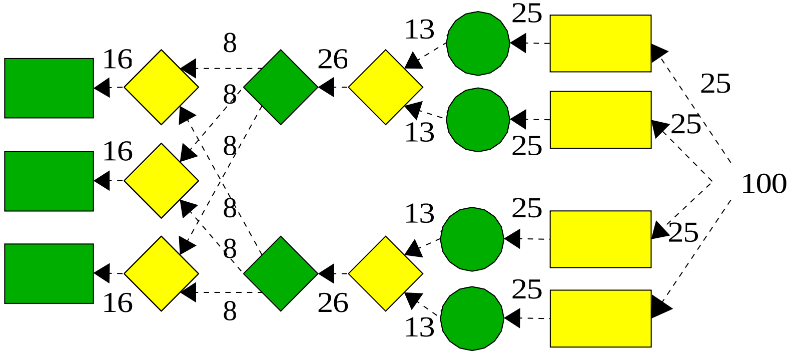


Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

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

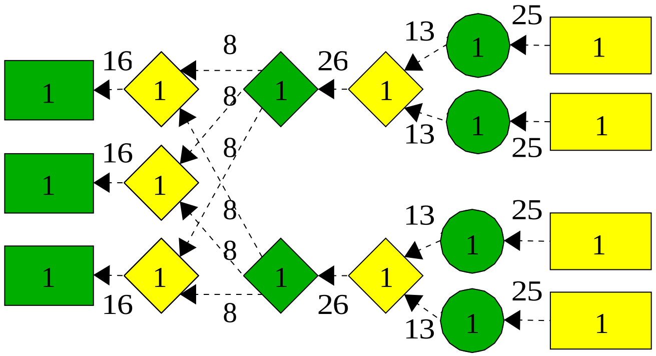


Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

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

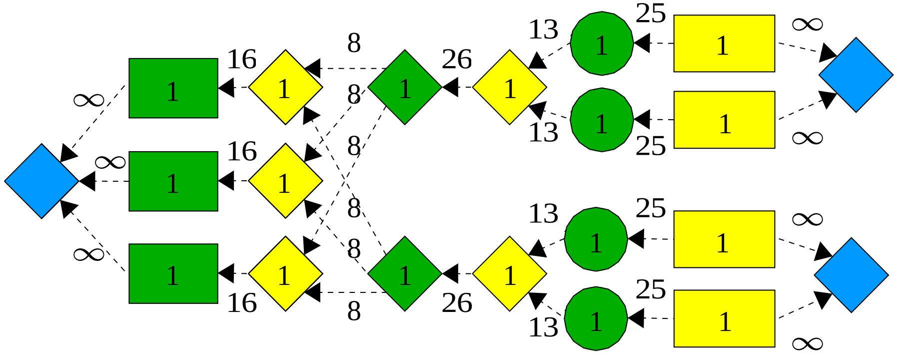


Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

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

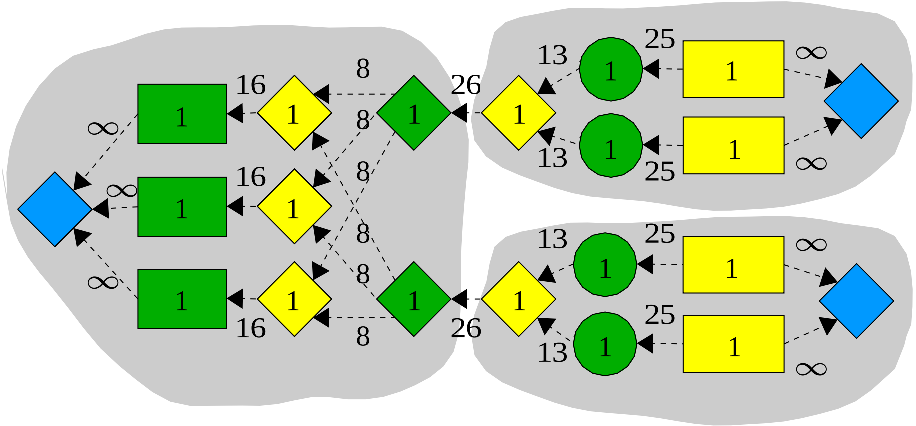


Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

Algorithm for placement of Armada ships

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

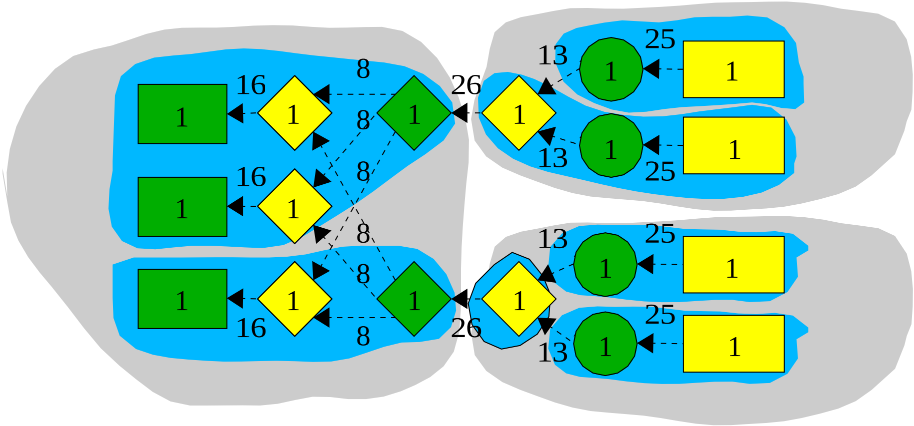


Partitioning an Armada Graph

Chaco Graph Partitioning Software [Hendrickson and Leland, SNL]

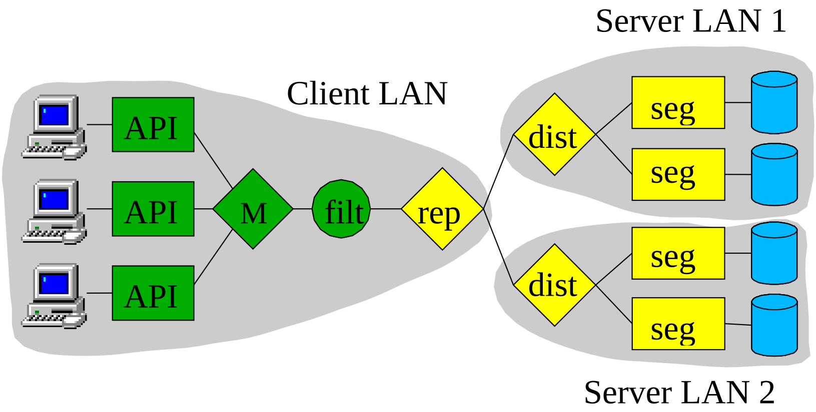
Algorithm for placement of Armada ships

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

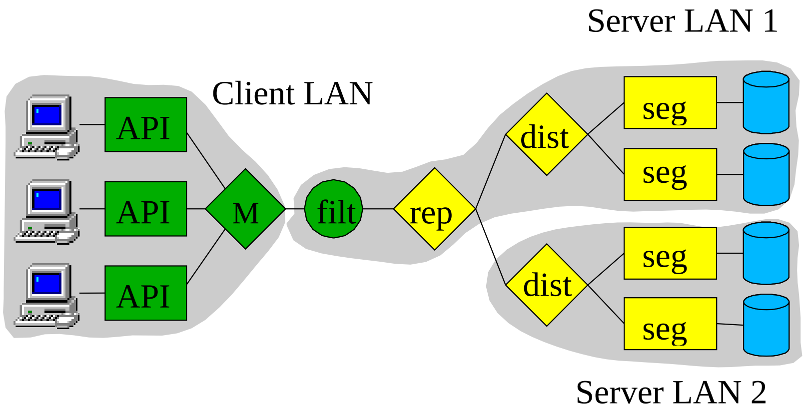


Experiments

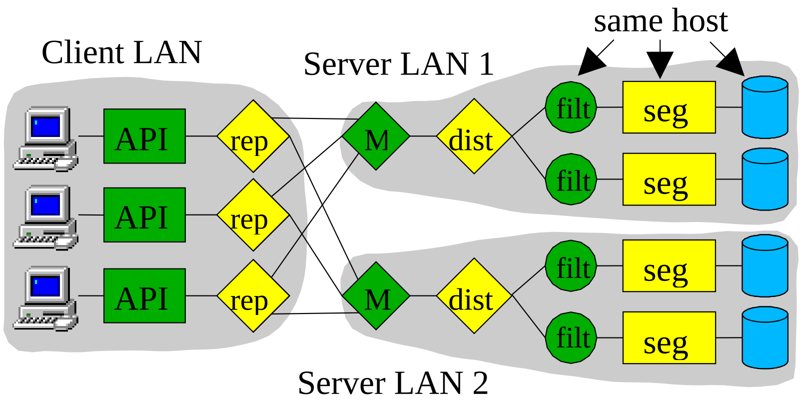
Examined four configurations of the example application with a filter that removed exactly 50% of the data.



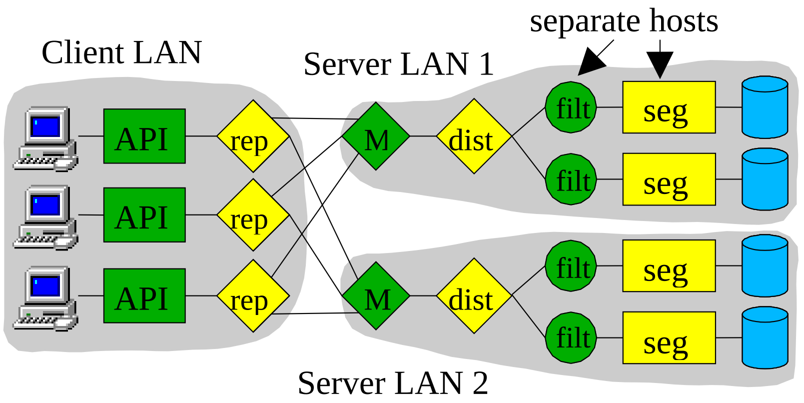
(a) orig1



(b) orig2



(c) restruct1

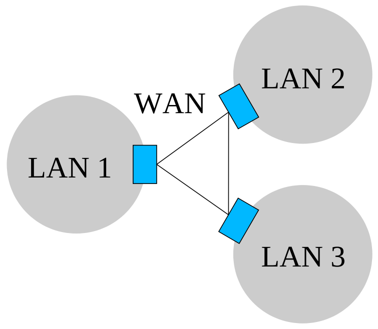


(d) restruct2

Experiment Setup

The area between the blobs represents the WAN

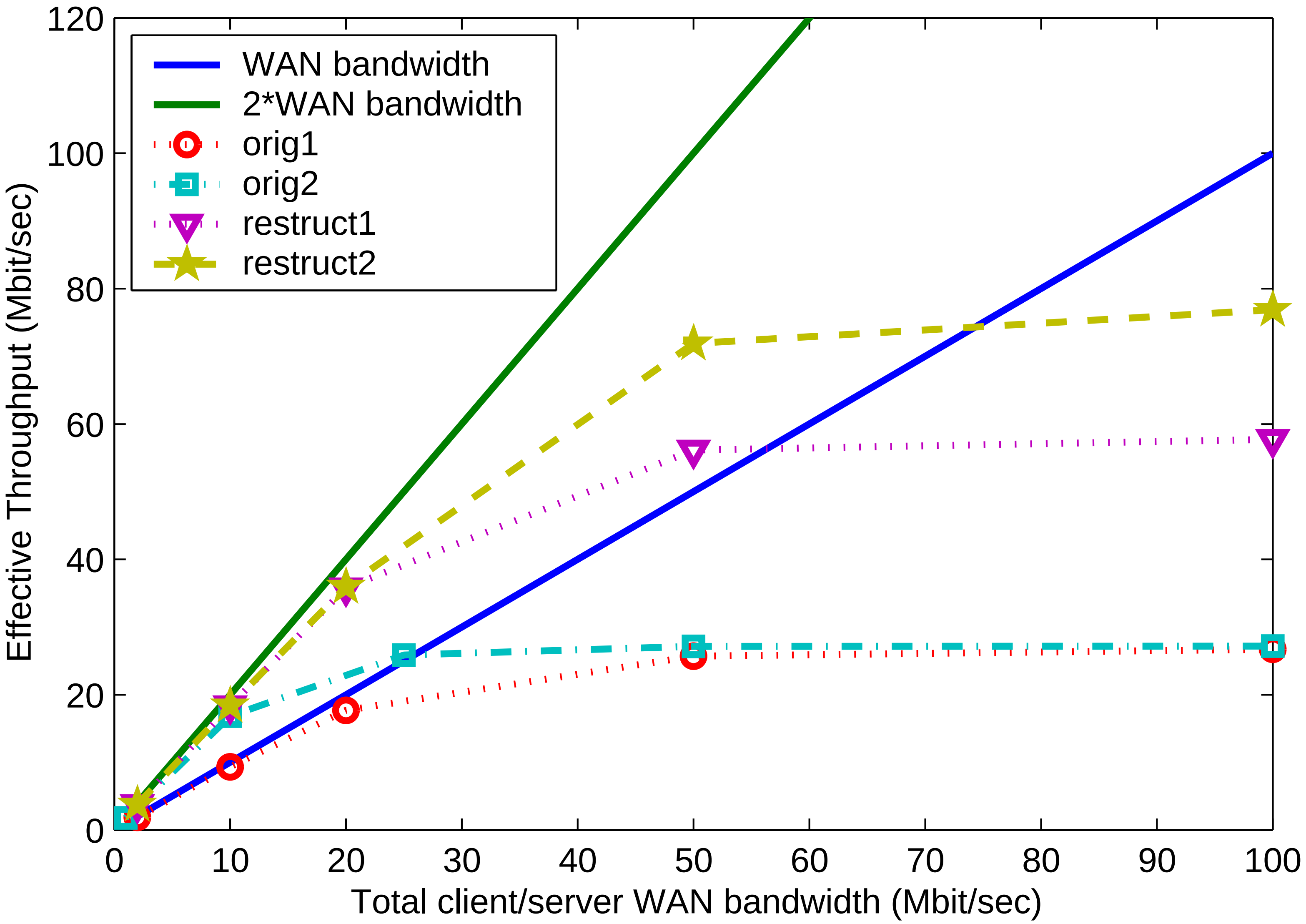
- each LAN connected to the WAN by single router
- each WAN link has limited capacity



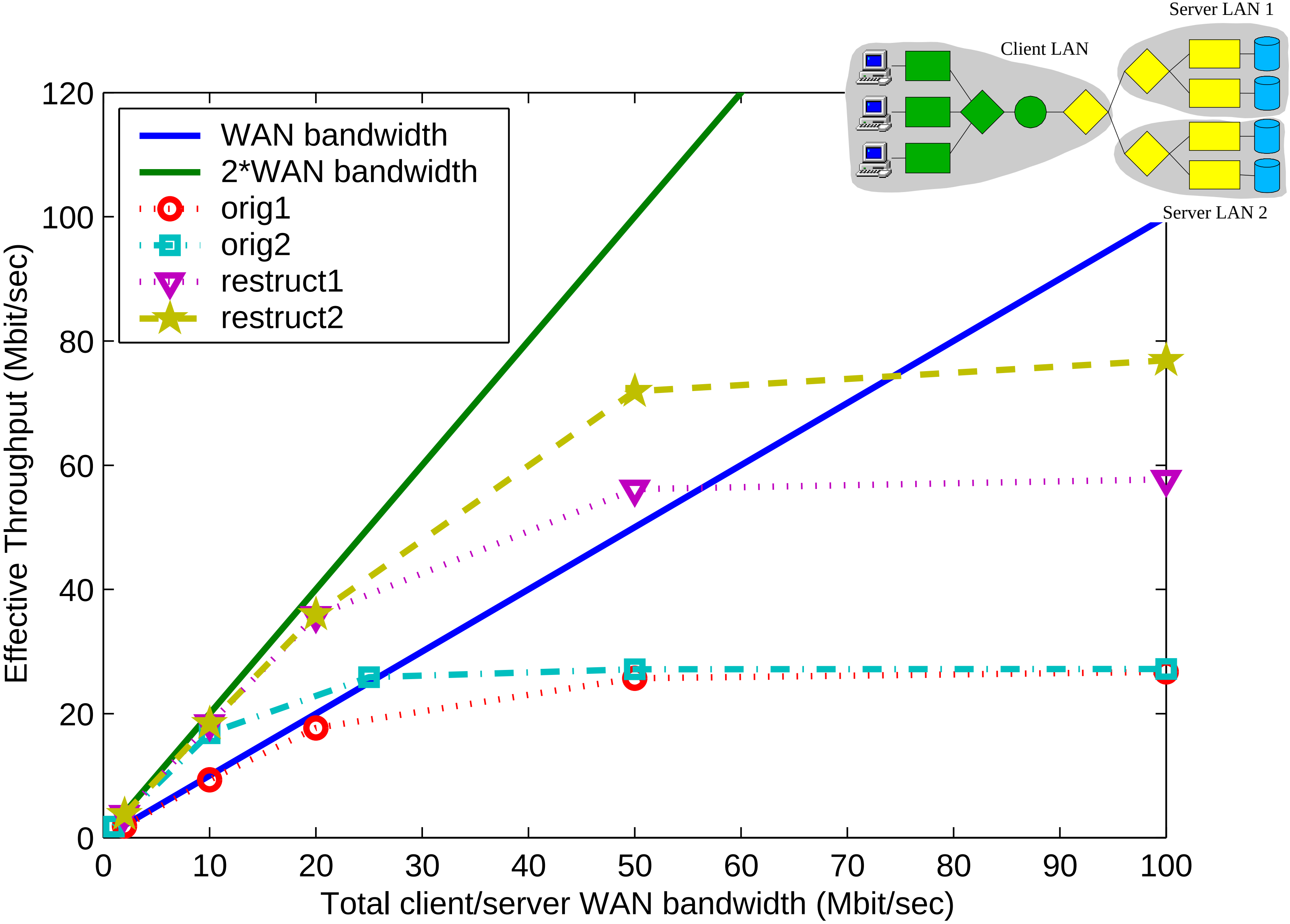
Ran experiments on the Emulab Network Testbed

- 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

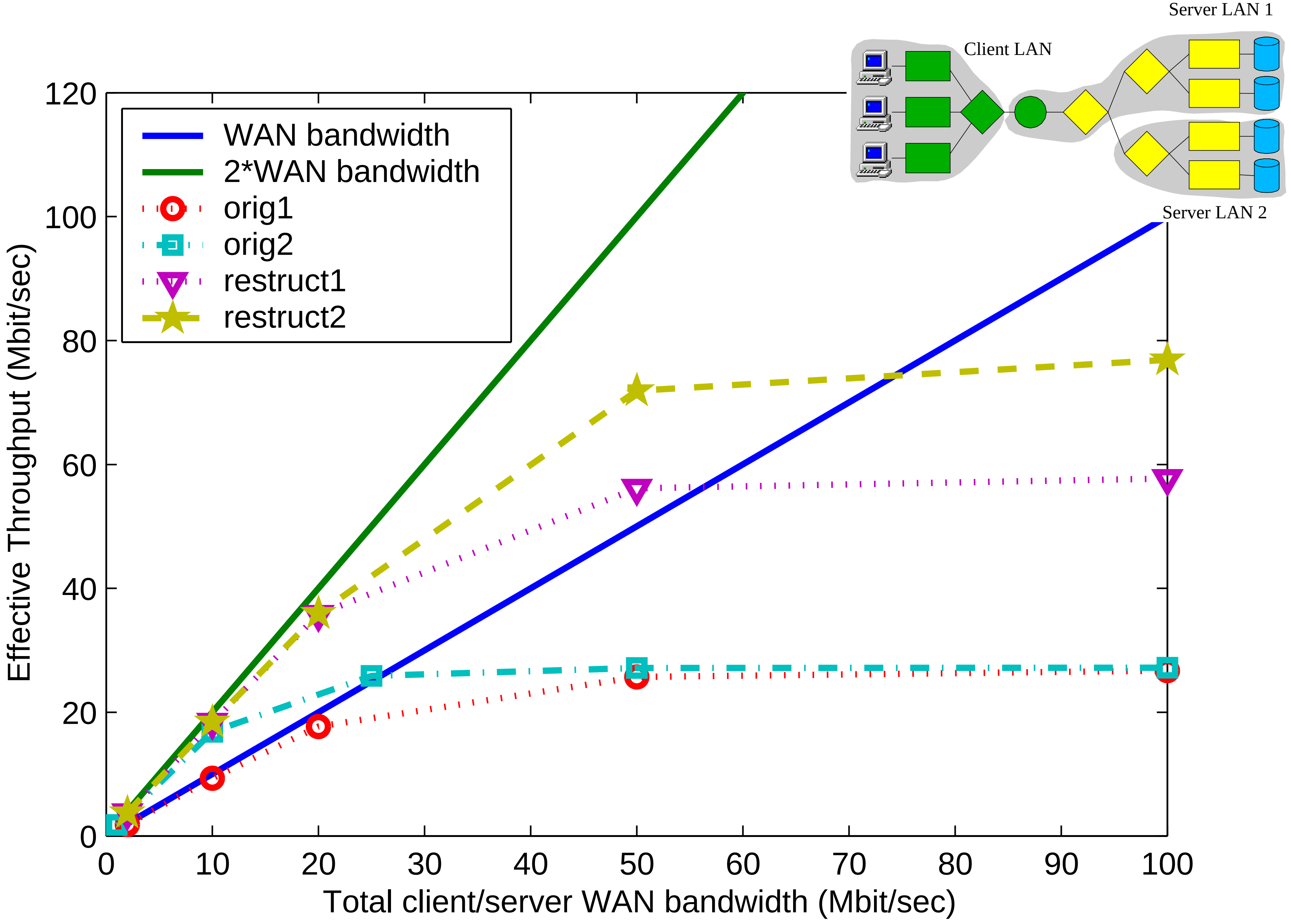
Results: Effective Throughput



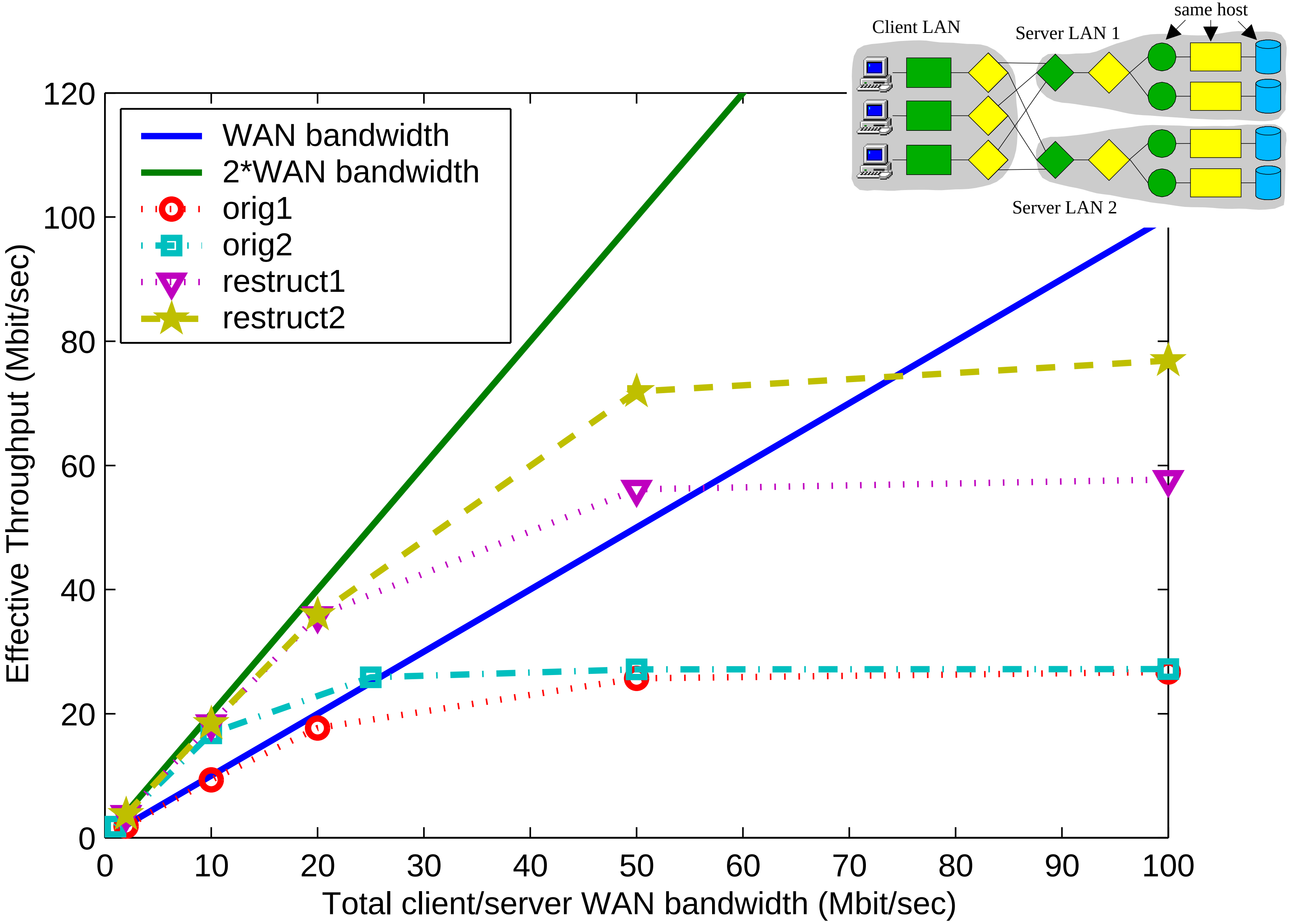
Results: Effective Throughput



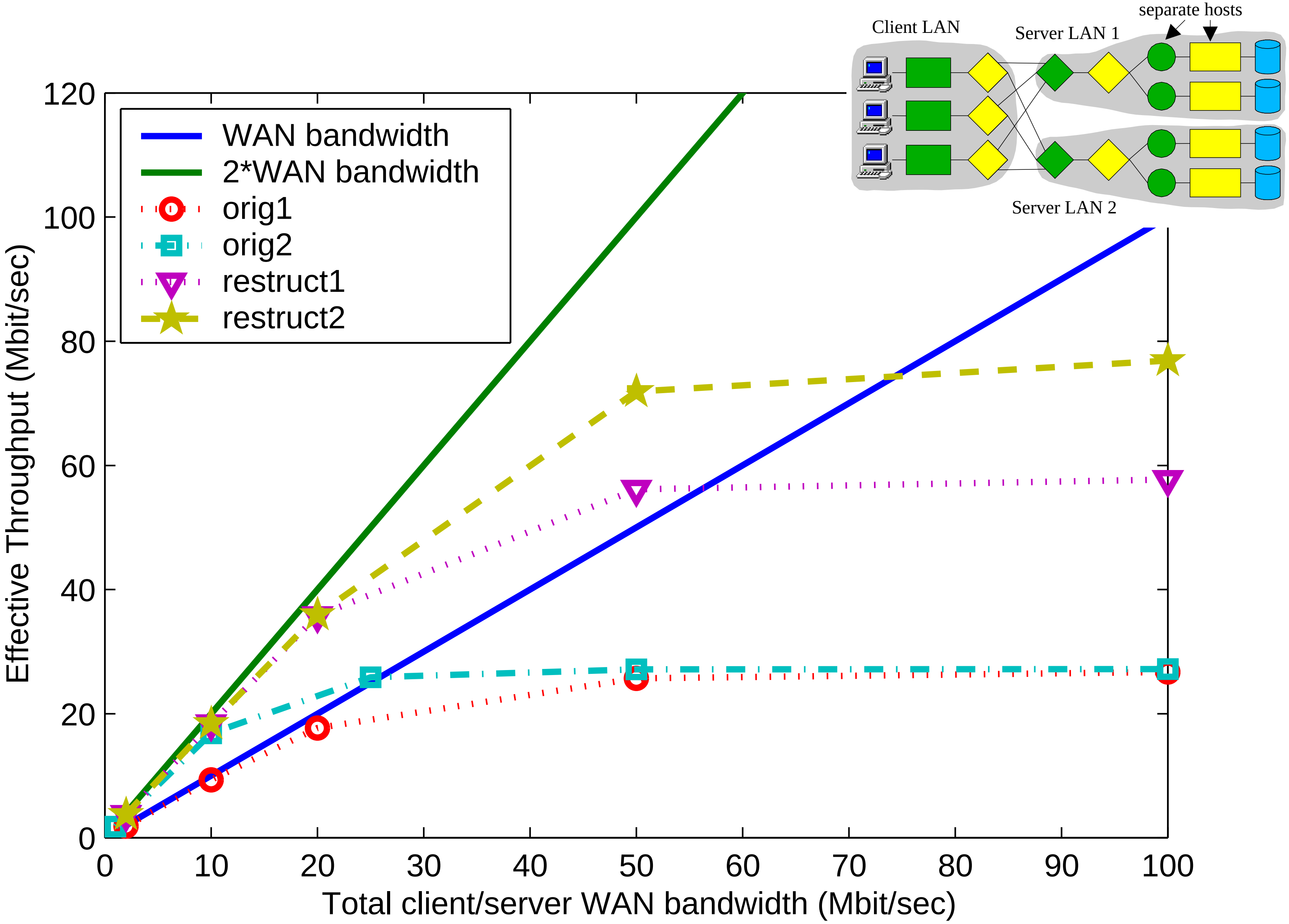
Results: Effective Throughput



Results: Effective Throughput



Results: Effective Throughput



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

Real Applications

- fMRI application (80 TBytes of brain image data)
- Seismic application (3 TBytes of synthetic seismic data)
- Can components be reused between applications?
- How much can performance benefit?

Modifications to BENEFICIAL and COMMUTATIVE

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

- allows data provider to describe complex distributed data sets
- allows the application to describe processing required before computation
- data-flow model provides a “latency-tolerant” approach useful for wide-area computing

Restructuring algorithm

- arranges graph to provide end-to-end parallel I/O
- enables effective placement of data-processing components to reducing network traffic over slow network links

Placement

- hierarchical approach: application-level assignment to domain, domain-level assignment to processors.

Experiments show that restructuring is beneficial in both low and high-bandwidth environments.

The High-Performance I/O for Computational Grid Applications

Ron Oldfield and David Kotz

Department of Computer Science, Dartmouth College

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

Supported by Sandia National Laboratories under contract DOE-AV6184.