Resource Allocation Strategies for In-Network Stream Processing

Anne Benoit  Henri Casanova*  Veronika Rehn-Sonigo
Yves Robert

GRAAL team, LIP
École Normale Supérieure de Lyon
France

*University of Hawai‘i at Manoa
USA

Workshop Passau
December 9th, 2008
Introduction and motivation

Operator-mapping problem for in-network stream processing

- Applications structured as trees of operators
- Execution in steady-state
- Multiple data objects are continually updated at various locations on a network
- Constructive scenario: create the platform you need

Applications?

- Processing of data in a sensor network
- Video surveillance
- Continuous queries on distributed relational databases
- Network monitoring
Rule of the game

- **Goal**: Minimize the cost for platform creation.
Rule of the game

Goal

Minimize the cost for platform creation.
Rule of the game

Goal

Minimize the cost for platform creation.
Major contributions

**Theory**  Definition operator-placement problem  
Problem complexity  
Linear programming formulation  

**Practice**  Polynomial heuristics  
Experiments to compare heuristics and evaluate their performance
Major contributions

Theory  Definition operator-placement problem
Problem complexity
Linear programming formulation

Practice Polynomial heuristics
Experiments to compare heuristics and evaluate their performance
Outline

1. Models
2. Complexity Results
3. Linear programming formulation
4. Heuristics
5. Experiments
6. Conclusion
The Application Model

- \( \mathcal{N} \) set of operators
- \( \mathcal{O} = \{ o_1, o_2, o_3, \ldots \} \) basic objects
- \( n_1, n_2, n_3, \ldots \) internal nodes: operator computations
- Computation of operator \( n_i \):
  - \( w_i \) operations, \( \delta_i \) size of output
- \( \rho \) application throughput

Object \( o_k \)
- \( \delta_k \) size of \( o_k \)
- \( f_k \) download frequency
- \( rate_k = \delta_k \times f_k \) bandwidth consumption
The Platform Model

- $S$ servers (given)
- $P$ processors (to be purchased)
- fully connected graph (i.e. a clique)
- $B_{s_i}$, $B_{p_u}$ network cards
- $b_{s_l,u}$ bandwidth link $S_l$ to $P_u$
- $b_{p_u,v}$ (i.e. $b_{p_v,u}$) bidirectional, bandwidth shared link $P_u$ and $P_v$
- $s_u$ compute speed $P_u \in P$
- $c_{ost_u}$ cost $P_u \in P$
The Platform Model

- $S$ servers (given)
- $\mathcal{P}$ processors (to be purchased)
- fully connected graph (i.e. a clique)

- $B_{si}, B_{pu}$ network cards
- $bs_{i,u}$ bandwidth link $S_i$ to $P_u$
- $bp_{u,v} (= bp_{v,u})$ bidirectional, bandwidth shared link $P_u$ and $P_v$
- $s_u$ compute speed $P_u \in \mathcal{P}$
- $\text{cost}_u$ cost $P_u \in \mathcal{P}$
The Platform Model

- $S$ servers (given)
- $P$ processors (to be purchased)
- fully connected graph (i.e. a clique)

- $Bs_i$, $Bp_u$ network cards
- $bs_{i,u}$ bandwidth link $S_i$ to $P_u$
- $bp_{u,v} (=bp_{v,u})$ bidirectional, bandwidth shared link $P_u$ and $P_v$
- $s_u$ compute speed $P_u \in P$
- $cost_u$ cost $P_u \in P$
The Communication Model

**Full-overlap, bounded multi-port model:** resource $R_u$ can be involved in computing, sending data, and receiving data simultaneously.

**multi-port** $R$ can send/receive data simultaneously on multiple network links.

**bounded** Total transfer rate of data sent/received by resource $R$ is bounded by its network card bandwidth ($B_{s_l}$ for server $S_l$, or $B_{p_u}$ for processor $P_u$).
The Mapping Model

Objective

Purchase processors to create a platform of \textit{minimal cost} that is able to achieve the desired application throughput.

- Each processor is in charge of one or several operators
- Operator \( n_i \) mapped on processor \( P_u \)
- Processor computes for the \( t \)-th final result
- \( n_i \) sends to its parent (if any) intermediate results for the \( (t - 1) \)-th final result
- Receives data from its non-leaf children (if any) for computing the \( (t + 1) \)-th final result
The Mapping Model

Objective

Purchase processors to create a platform of minimal cost that is able to achieve the desired application throughput.

- Each processor is in charge of one or several operators
- Operator $n_i$ mapped on processor $P_u$
- Processor computes for the $t$-th final result
- $n_i$ sends to its parent (if any) intermediate results for the $(t - 1)$-th final result
- Receives data from its non-leaf children (if any) for computing the $(t + 1)$-th final result
Constraints

- **Computation capability** $P_u$: $\forall P_u \in \mathcal{P}, \quad \sum_{i \in \bar{a}(u)} \rho \cdot \frac{w_i}{s_u} \leq 1$

- **Bandwidth capacity** $P_u$:
  $\forall P_u \in \mathcal{P}, \quad \sum_{(k, l) \in \text{download}(u)} \text{rate}_k + \sum_{j \in \text{Child}(\bar{a}(u)) \setminus \bar{a}(u)} \rho \cdot \delta_j + \sum_{j \in \text{Parent}(\bar{a}(u)) \setminus \bar{a}(u)} \sum_{i \in \text{Child}(j) \cap \bar{a}(u)} \rho \cdot \delta_i \leq Bp_u$

- **Bandwidth capacity** $S_l$:
  $\forall S_l \in \mathcal{S}, \quad \sum_{P_u \in \mathcal{P}} \sum_{(k, l) \in \text{download}(u)} \text{rate}_k \leq Bs_l$

- **Link bandwidth** $S_l \leftrightarrow P_u$:
  $\forall P_u \in \mathcal{P}, \forall S_l \in \mathcal{S}, \quad \sum_{(k, l) \in \text{download}(u)} \text{rate}_k \leq bs_{l,u}$

- **Link bandwidth** $P_u \leftrightarrow P_v$:
  $\forall P_u, P_v \in \mathcal{P}, \quad \sum_{j \in \text{Child}(\bar{a}(u)) \cap \bar{a}(v)} \rho \cdot \delta_j + \sum_{j \in \text{Parent}(\bar{a}(u)) \cap \bar{a}(v)} \sum_{i \in \text{Child}(j) \cap \bar{a}(u)} \rho \cdot \delta_i \leq bp_{u,v}$

$\bar{a}(u)$ denotes the index set of operators mapped on $P_u$.
Assess the complexity of the simplest instance of the problem: Everything is homogeneous

Mapping problem
LDT-Operator-Mapping-Hom
- fully homogeneous left-deep tree application
- fully homogeneous set of servers
- fully homogeneous set of processors
The problem LDT-Operator-Mapping-Hom consists in minimizing the number of processors used in the application execution. $K$ is the prescribed throughput that should not be violated. LDT-Operator-Mapping-Hom-Dec is the associated decision problem: given a number of processors $N$, is there a mapping that achieves throughput $K$?

Theorem

LDT-Operator-Mapping-Hom-Dec is NP-complete.

Reduction from 3-Partition.
Definition

The problem LDT-Operator-Mapping-Hom consists in minimizing the number of processors used in the application execution. $K$ is the prescribed throughput that should not be violated.

$LDT$-Operator-Mapping-Hom-Dec is the associated decision problem: given a number of processors $N$, is there a mapping that achieves throughput $K$?

Theorem

$LDT$-Operator-Mapping-Hom-Dec is NP-complete.

Reduction from 3-Partition.
Introduction

Models

Complexity

LP

Heuristics

Experiments

Conclusion

Outline

1 Models

2 Complexity Results

3 Linear programming formulation

4 Heuristics

5 Experiments

6 Conclusion

Veronika.Sonigo@ens-lyon.fr

December 2008
Integer Linear Programming

- **Integer LP** to solve Operator-Mapping-LAN:
  \[ bp_{u,v} = bp, \quad bs_{l,u} = bs_{l}, \quad Bs_{l} = Bs, \quad Bp_u \neq Bp, \text{ and } s_u \neq s \]

- Many integer variables: no **efficient** algorithm to solve
- Approach limited to small problem instances
- **Absolute performance of the heuristics for such instances**
$C$ classes of processors:
each $c \in C$ has parameters $s_c$, $Bp_c$ and $cost_c$

$P_{c,u}$: $u$-th processor of class $c$

Boolean variables

- $x_{i,c,u}$: 1 if $n_i$ is mapped on $P_{c,u}$
- $d_{c,u,k,l}$: 1 if $P_{c,u}$ downloads $o_k$ from $S_l$
- $y_{i,c,u,i',c',u'}$: 1 if $n_i$ is mapped on $P_{c,u}$, $n_{i'}$ is mapped on $P_{c',u'}$, and $n_i$ is the parent of $n_{i'}$ in the application tree.
- $used_{c,u}$: 1 if $P_{c,u}$ is used in the final mapping
Linear Program: Constraints

Allocation constraints (Proc - Operator)

- $\forall i \sum_{c,u} x_{i,c,u} = 1$
- $\forall c, u, k, l \ d_{c,u,k,l} \leq \text{obj}(k, l)$
- $\forall c, u, k, l \ d_{c,u,k,l} \leq \sum_i x_{i,c,u} \cdot \text{leaf}(i, k)$
- $\forall i, k, c, u \ 1 \geq \sum_l d_{c,u,k,l} \geq x_{i,c,u} \cdot \text{leaf}(i, k)$

Parent constraints

- $y_{i,c,u,i',c',u'} \leq \text{par}(i, j)$; $y_{i,c,u,i',c',u'} \leq x_{i,c,u}$;
- $y_{i,c,u,i',c',u'} \leq x_{i',c',u'}$;
- $y_{i,c,u,i',c',u'} \geq \text{par}(i, j) \cdot (x_{i,c,u} + x_{i',c',u'} - 1)$
Linear Program: Constraints

Processor usage constraints

- $\forall c, u \ used_{c,u} \leq \sum_i x_{i,c,u}$
- $\forall c, u, i \ used_{c,u} \geq x_{i,c,u}$

Throughput constraints

- $\forall c, u \ \sum_i x_{i,c,u} \cdot \rho \frac{w_i}{s_c} \leq 1$
- $\forall c, u \ \sum_{k,l} d_{c,u,k,l} \cdot rate_k + \sum_{i,i'} (c',u') \neq (c,u) \ y_{i,c,u,i',c',u'} \cdot \rho \cdot \delta_i + \sum_{i,i'} (c',u') \neq (c,u) \ y_{i',c',u',i,c,u} \cdot \rho \cdot \delta_i \leq Bp_c$
- $\forall l \ \sum_{c,u,k} d_{c,u,k,l} \cdot rate_k \leq Bs_l$
- $\forall l, c, u \ \sum_k d_{c,u,k,l} \cdot rate_k \leq bs_l$
- $\forall c, u, c', u' \text{ with } (c, u) \neq (c', u') \ \sum_{i,i'} y_{i,c,u,i',c',u'} \cdot \rho \cdot \delta_i + \sum_{i,i'} y_{i',c',u',i,c,u} \cdot \rho \cdot \delta_i \leq bp$
Objective function.
We aim at minimizing the cost of used processors, thus the objective function is

$$\min \left( \sum_{c,u} used_{c,u} \cdot cost_c \right).$$
Outline

1. Models
2. Complexity Results
3. Linear programming formulation
4. Heuristics
5. Experiments
6. Conclusion
Heuristics

Operator-Mapping-LAN

General approach to the operator-placement problem

- **Step 1**: which operators on which processor?
- **Step 2**: object download decisions
- **Step 3**: downgrade the processors as much as possible

![Diagram](attachment:image.png)
Heuristics

Operator-Mapping-LAN

General approach to the operator-placement problem

Step1: which operators on which processor?

Step2: object download decisions

Step3: downgrade the processors as much as possible

Step 1
Heuristics

Operator-Mapping-LAN

General approach to the operator-placement problem

**Step 1**: which operators on which processor?

**Step 2**: object download decisions

**Step 3**: downgrade the processors as much as possible

Step 1
Heuristics

Operator-Mapping-LAN

General approach to the operator-placement problem

**Step1:** which operators on which processor?

**Step2:** object download decisions

**Step3:** downgrade the processors as much as possible

Step 2
Heuristics

Operator-Mapping-LAN

General approach to the operator-placement problem

**Step1**: which operators on which processor?

**Step2**: object download decisions

**Step3**: downgrade the processors as much as possible
Random

While there are some unassigned operators
- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
  If no such proc try to group with children or father
While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
If no such proc try to group with children or father
Random

While there are some unassigned operators
- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
If no such proc try to group with children or father
Random

While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
- If no such proc try to group with children or father
While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
  If no such proc try to group with children or father
Random

While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
If no such proc try to group with children or father
While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
  If no such proc try to group with children or father
While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
  If no such proc try to group with children or father
Random

While there are some unassigned operators

- Pick randomly one unassigned operator
- Purchase the cheapest possible processor
  If no such proc try to group with children or father
Comp-Greedy

**Idea:** most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,
- Purchase the most expensive processor available
- Assign the most computationally demanding unassigned operator
- If not possible group with father or children
- If some capacity is left on the processor, assign other operators
Comp-Greedy

Idea: most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,
  • Purchase the most expensive processor available
  • Assign the most computationally demanding unassigned operator
  • If not possible group with father or children
  • If some capacity is left on the processor, assign other operators
**Comp-Greedy**

**Idea:** most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,
- Purchase the most expensive processor available
- Assign the most computationally demanding unassigned operator
- If not possible group with father or children
- If some capacity is left on the processor, assign other operators
Comp-Greedy

**Idea:** most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,
- Purchase the most expensive processor available
- Assign the most computationally demanding unassigned operator
- If not possible group with father or children
- If some capacity is left on the processor, assign other operators
Comp-Greedy

**Idea:** most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,

- Purchase the most expensive processor available
- Assign the most computationally demanding unassigned operator
- If not possible group with father or children
- If some capacity is left on the processor, assign other operators
Comp-Greedy

Idea: most computationally demanding operators first
Sort operators in non-increasing order of $w_i$

While there are unassigned operators,
- Purchase the most expensive processor available
- Assign the most computationally demanding unassigned operator
- If not possible group with father or children
- If some capacity is left on the processor, assign other operators
**Comm-Greedy**

**Idea:** Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i. Both operators were unassigned: purchase processor and map;
  - ii. One operator is already assigned: accommodate the other operator as well;
  - iii. Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
Comm-Greedy

Idea: Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i. Both operators were unassigned: purchases processor and map;
  - ii. One operator is already assigned: accommodate the other operator as well;
  - iii. Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
Comm-Greedy

Idea: Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i. Both operators were unassigned: purchases processor and map;
  - ii. One operator is already assigned: accommodate the other operator as well;
  - iii. Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
Comm-Greedy

**Idea:** Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i) Both operators were unassigned: purchases processor and map;
  - ii) One operator is already assigned: accommodate the other operator as well;
  - iii) Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
Comm-Greedy

**Idea:** Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i. Both operators were unassigned: purchases processor and map;
  - ii. One operator is already assigned: accommodate the other operator as well;
  - iii. Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
**Comm-Greedy**

**Idea:** Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i. Both operators were unassigned: purchase processor and map;
  - ii. One operator is already assigned: accommodate the other operator as well;
  - iii. Both operators are already assigned: accommodate both operators on one processor and sell the other processor;
Comm-Greedy

Idea: Group operators to reduce communication costs.

- Pick the two operators that have the largest communication requirements
- Group them
- Assign them to the same processor
  - i) Both operators were unassigned: purchases processor and map;
  - ii) One operator is already assigned: accommodate the other operator as well;
  - iii) Both operators are already assigned: accommodate both operators on one processor and sell the other processor;

\[ \begin{align*}
& n_1 & n_2 & n_3 \\
& n_4 & n_5 & n_6 \\
& n_7 & n_8 & n_9 \\
\end{align*} \]
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort al-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first al-operators, then other operators as much as possible.
Object-Greedy

**Idea:** Group operators that need the same basic objects.

- Sort al-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor.
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first al-operators, then other operators as much as possible.
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort all-operators by non-increasing order of maximum \( r_{a,j} \) values of their required basic objects (and \( w_i \) in case of equality).
- Purchase the most expensive processor.
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first all-operators, then other operators as much as possible.
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort al-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first al-operators, then other operators as much as possible.
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort all-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first all-operators, then other operators as much as possible.
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort all-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor.
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first all-operators, then other operators as much as possible.
Object-Greedy

Idea: Group operators that need the same basic objects.

- Sort all-operators by non-increasing order of maximum $r_{\text{rate}_j}$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor.
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first all-operators, then other operators as much as possible.
Object-Greedy

**Idea:** Group operators that need the same basic objects.

- Sort all-operators by non-increasing order of maximum $rate_j$ values of their required basic objects (and $w_i$ in case of equality).
- Purchase the most expensive processor.
- Assign the first such operators to it. If not possible group the operator with one of its unassigned parent or child operators. Otherwise fail.
- Fill the processor greedily: first all-operators, then other operators as much as possible.
Subtree-Bottom-Up

- Purchase as many most expensive processors as there are al-operators
- Assign each al-operator to a distinct processor
- While there are unassigned operators
  - Merge the operators with their father on a single machine (bottom-up)
  - Buy new processors if required
  - Sell back unused processors
Subtree-Bottom-Up

- Purchase as many most expensive processors as there are al-operators.
- Assign each al-operator to a distinct processor.
- While there are unassigned operators:
  - Merge the operators with their father on a single machine (bottom-up).
  - Buy new processors if required.
  - Sell back unused processors.
Subtree-Bottom-Up

- Purchase as many most expensive processors as there are al-operators.
- Assign each al-operator to a distinct processor.
- While there are unassigned operators:
  - Merge the operators with their father on a single machine (bottom-up).
  - Buy new processors if required.
  - Sell back unused processors.
Subtree-Bottom-Up

- Purchase as many most expensive processors as there are al-operators
- Assign each al-operator to a distinct processor
- While there are unassigned operators
  - Merge the operators with their father on a single machine (bottom-up)
  - Buy new processors if required
  - Sell back unused processors
Subtree-Bottom-Up

- Purchase as many most expensive processors as there are al-operators
- Assign each al-operator to a distinct processor
- While there are unassigned operators
  - Merge the operators with their father on a single machine (bottom-up)
  - Buy new processors if required
  - Sell back unused processors
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity.
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need.
- While there are unassigned operators:
  - Purchase the most expensive processor.
  - Assign the first al-operator in the list.
  - Assign other al-operators that require the same basic objects.
  - Assign non-al-operators.
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: **popularity**
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators

Veronika.Sonigo@ens-lyon.fr December 2008
Object-Grouping

For each basic object, count how many operators need this basic object: popularity

Sort al-operators by non-increasing sum of the popularities of the basic objects they need

While there are unassigned operators

- Purchase the most expensive processor
- Assign the first al-operator in the list
- Assign other al-operators that require the same basic objects
- Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: **popularity**
- Sort al-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first al-operator in the list
  - Assign other al-operators that require the same basic objects
  - Assign non al-operators
Object-Grouping

- For each basic object, count how many operators need this basic object: popularity
- Sort all-operators by non-increasing sum of the popularities of the basic objects they need
- While there are unassigned operators
  - Purchase the most expensive processor
  - Assign the first all-operator in the list
  - Assign other all-operators that require the same basic objects
  - Assign non all-operators
Object-Availability

**Idea:** Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $av_k$ of servers handling object $o_k$ is calculated.
- Al-operators in turn are treated in increasing order of $av_k$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

**Idea:** Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $a_{v_k}$ of servers handling object $o_k$ is calculated.
- Al-operators in turn are treated in increasing order of $a_{v_k}$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

**Idea:** Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $av_k$ of servers handling object $o_k$ is calculated.
- AI-operators in turn are treated in increasing order of $av_k$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

Idea: Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $a_v^k$ of servers handling object $o_k$ is calculated.
- Al-operators in turn are treated in increasing order of $a_v^k$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

**Idea:** Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $av_k$ of servers handling object $o_k$ is calculated.
- Al-operators in turn are treated in increasing order of $av_k$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

**Idea:** Take into account the distribution of basic objects on the servers.

- For each object \( k \) the number \( av_k \) of servers handling object \( o_k \) is calculated.
- Al-operators in turn are treated in increasing order of \( av_k \) of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object \( k \) as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Object-Availability

Idea: Take into account the distribution of basic objects on the servers.

- For each object $k$ the number $av_k$ of servers handling object $o_k$ is calculated.
- Al-operators in turn are treated in increasing order of $av_k$ of the basic objects they need to download.
- Purchase a most expensive processor
- Assign as many al-operators downloading object $k$ as possible
- Remaining internal operators are assigned in the same mechanism as Comp-Greedy proceeds
Step 2: Server Selection Heuristics

Server-Selection-Random only used for Random
Associate randomly a server to each basic object a processor has to download.

Server-Selection-Intelligent used in combination with all other heuristics
Three loops:

- Assign objects that are held in exclusivity, i.e., objects that have to be downloaded from a specific server.
- Associate as many downloads as possible to servers that provide only one basic object type.
- Assign remaining basic objects that have to be downloaded:
  - Objects are treated in decreasing order of interestedProcs/numPossibleServers
  - Servers are considered in decreasing order of min(remainingBW, linkBW)
Step 2: Server Selection Heuristics

**Server-Selection-Random** only used for Random
Associate randomly a server to each basic object a processor has to download.

**Server-Selection-Intelligent** used in combination with all other heuristics
Three loops:

- Assign objects that are held in exclusivity, i.e., objects that have to be downloaded from a specific server.
- Associate as many downloads as possible to servers that provide only one basic object type.
- Assign remaining basic objects that have to be downloaded:
  - Objects are treated in decreasing order of interestedProcs/numPossibleServers
  - Servers are considered in decreasing order of \( \min(\text{remainingBW}, \text{linkBW}) \)
Outline

1. Models
2. Complexity Results
3. Linear programming formulation
4. Heuristics
5. Experiments
6. Conclusion
Dell prices for configurations of Intel’s latest, high-end, rack-mountable server (PowerEdge R900). Prices of March 2008

Base configuration: $7,548

<table>
<thead>
<tr>
<th>Perf. (GHz)</th>
<th>Cost ($/GHz)</th>
<th>Ratio (GHz/$)</th>
<th>BW (Gbps)</th>
<th>Cost ($/Gbps)</th>
<th>Ratio (Gbps/$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.72</td>
<td>7,548 + 0</td>
<td>1.55 × 10^{-3}</td>
<td>1</td>
<td>7,548 + 0</td>
<td>1.32 × 10^{-4}</td>
</tr>
<tr>
<td>19.20</td>
<td>7,548 + 1,550</td>
<td>1.93 × 10^{-3}</td>
<td>2</td>
<td>7,548 + 399</td>
<td>2.51 × 10^{-4}</td>
</tr>
<tr>
<td>25.60</td>
<td>7,548 + 2,399</td>
<td>2.38 × 10^{-3}</td>
<td>4</td>
<td>7,548 + 1,197</td>
<td>4.57 × 10^{-4}</td>
</tr>
<tr>
<td>38.40</td>
<td>7,548 + 3,949</td>
<td>3.12 × 10^{-3}</td>
<td>10</td>
<td>7,548 + 2,800</td>
<td>9.66 × 10^{-4}</td>
</tr>
<tr>
<td>46.88</td>
<td>7,548 + 5,299</td>
<td>3.43 × 10^{-3}</td>
<td>20</td>
<td>7,548 + 5,999</td>
<td>14.76 × 10^{-4}</td>
</tr>
</tbody>
</table>
Simulation Methodology

- Assess performance of polynomial heuristics
- Random application trees
- 15 object types
  - Small objects: 5-30MB
  - Big objects: 450-530MB
- Download frequency
  - Small frequency: 1/50s
  - High frequency: 1/2s
- $w_n = (\delta_l + \delta_r)^\alpha$, where $\alpha$ is fixed for each run.
- $\delta_n = (\delta_l + \delta_r)$ for all simulations.
- $\rho = 1.0$.
- 6 servers with 10 GB network cards
- 1GB links proc-proc and proc-server
High frequency - small object sizes

Increasing number of operators.
objects: 5-30MB, frequency: 1/2s

\[ \alpha = 0.9 \]

😊 Subtree-Bottom-Up

\[ \alpha = 1.7 \]

😊 Object-Greedy, Comp-Greedy
Increasing $\alpha$.

objects: 5-30MB, frequency: 1/2s

20 operators.

Subtree-Bottom-Up

60 operators.

Subtree-Bottom-Up, Greedy-Family
Increasing number of operators.

objects: 450-530MB, frequency: 1/2s

\[ \alpha = 0.9 \]

😊 Subtree-Bottom-Up

😊 it fails 2 times

\[ \alpha = 1.1 \]

😊 Subtree-Bottom-Up, Comm-Greedy
High frequency - big object sizes

Increasing $\alpha$.

objects: 450-530MB, frequency: 1/2s

20 operators.

😊 Subtree-Bottom-Up

40 operators.

😊 Subtree-Bottom-Up

😢 Object-Greedy

Veronika.Sonigo@ens-lyon.fr  December 2008
Influence of the frequency - small object sizes

Influence of the frequency on the platform cost, in $, when object sizes are small.

objects: 5-30MB

<table>
<thead>
<tr>
<th>N</th>
<th>low frequency</th>
<th>high frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comm-Greedy</td>
<td>Comm-Greedy</td>
</tr>
<tr>
<td></td>
<td>Subtree-b-up</td>
<td>Subtree-b-up</td>
</tr>
<tr>
<td>115</td>
<td>7947</td>
<td>7548</td>
</tr>
<tr>
<td>116</td>
<td>15495</td>
<td>15096</td>
</tr>
<tr>
<td>117</td>
<td>7947</td>
<td>7548</td>
</tr>
<tr>
<td>118</td>
<td>15495</td>
<td>15096</td>
</tr>
<tr>
<td>119</td>
<td>15495</td>
<td>15096</td>
</tr>
</tbody>
</table>

Identical mapping, but sometimes less powerful network cards.
Influence of the frequency - big object sizes

Comparison between the simulations with big basic objects and different frequencies, $\alpha = 0.9$.

objects: 450-530MB

$\text{frequency} = 1/50\text{s}$.  

$\text{frequency} = 1/2\text{s}$.
Influence of download rates on the solution

Simulation to evaluate the influence of download rates on the solution.

140 operators.

160 operators.

Veronika.Sonigo@ens-lyon.fr December 2008
Influence of object availability

Simulation to evaluate the influence of object availability in relation with frequency on the solution.

Each basic object available only on one server.

Each basic object available on 50% of the servers.
Heuristics vs LP

Simulation to compare the heuristics’ performances to the LP performance on homogeneous platforms.

\[ \alpha = 0.9 \]

😊 Subtree-Bottom-Up

\[ \alpha = 1.1 \]

😊 Subtree-Bottom-Up
Summary of experiments

- All more sophisticated heuristics perform better than Random
- Greedy heuristics perform reasonably well
- **Subtree-bottom-up**
  - Outperforms other heuristics in most situations
  - Produces results very close to the optimal
- Object-Grouping and Object-Availability do not show the desired performance

Classification:
**Subtree-Bottom-Up** > Greedy family > Object aware > Random
Outline

1. Models
2. Complexity Results
3. Linear programming formulation
4. Heuristics
5. Experiments
6. Conclusion
Related work

Babu et al., Liu et al.
Execution of continuous queries on data streams

Chen et al., van Rennesse et al.
In-network stream processing systems
These systems all face the same question: where should operators be mapped in the network?

Pietzuch et al., Srivastava et al.
Operator-mapping problem for in-network stream processing
Resource allocation problem for in-network stream processing

- Formulation of the operator-placement problem
- Focus on a “constructive” scenario:
  minimizing the cost of a platform that satisfies an application throughput requirement
- Complexity analysis: NP-completeness for simplest instance
- Integer linear programming formulation

Practical side

- Polynomial time heuristics
- Simulation: Subtree-bottom-up is almost always better
- Absolute performance via ILP optimal solution: Subtree-bottom-up heuristic almost always produces optimal results.
Conclusion

Resource allocation problem for in-network stream processing

- Formulation of the operator-placement problem
- Focus on a “constructive” scenario: minimizing the cost of a platform that satisfies an application throughput requirement
- Complexity analysis: NP-completeness for simplest instance
- Integer linear programming formulation

Practical side

- Polynomial time heuristics
- Simulation: Subtree-bottom-up is almost always better
- Absolute performance via ILP optimal solution: Subtree-bottom-up heuristic almost always produces optimal results.
Future work

- Multiple applications must be executed simultaneously
  - Throughput must be achieved for each application
  - Reuse of common sub-expression between trees

- Mutable applications
  - Operators can be rearranged based on operator associativity and commutativity rules
  - Ex: relational database applications