Resource Allocation for Multiple Concurrent In-network Stream-processing Applications

Anne Benoit[1], Henri Casanova[2], Veronika Rehn-Sonigo[1], and Yves Robert[1]

[1] LIP, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
{Anne.Benoit,Veronika.Sonigo,Yves.Robert}@ens-lyon.fr
[2] University of Hawai'i at Manoa, 1680 East-West Road, Honolulu, HI 96822, USA
henric@hawaii.edu

Abstract. This paper investigates the operator mapping problem for in-network stream-processing applications. In-network stream-processing is the application of one or more trees of operators, in steady-state, to data that are continuously updated at different locations in the network. The goal is to generate final results at a desired rate. Different operator trees may share common subtrees, so that intermediate results could be reused in different applications. This work provides complexity results for different instances of the basic problem and proposes several polynomial-time heuristics. Quantitative comparison of the heuristics in simulation demonstrates the importance of mapping operators to appropriate processors, and allows us to identify a heuristic that achieves good results in practice.

1 Introduction

We consider applications structured as trees of operators, where leaves correspond to basic data objects distributed in a network. Each internal node in the tree denotes the aggregation and combination of the data from its children, which in turn generates new data that is used by the node's parent. The computation is complete when all operators have been applied up to the root node, thereby producing a final result. We consider the scenario in which the basic data objects are constantly being updated, meaning that the tree of operators must be applied continuously. The goal is to produce final results at some desired rate.

The above problem is called *stream processing* [4] and arises in several domains. One such domain is the acquisition and refinement of data from a set of sensors [21, 16]. For instance, [21] outlines a video surveillance application in which the sensors are cameras located at different locations over a geographical area. Another example arises in the area of network monitoring [10, 22]. In this case routers produce streams of data pertaining to forwarded packets. More generally, stream processing can be seen as the execution of one of more "continuous queries" in the relational database sense of the term (e.g., a tree of join and select operators). Many authors have studied the execution of continuous queries on data streams [3, 7, 20, 14].

In practice, the execution of the operators must be distributed over the network. In some cases the servers that produce the basic objects may not have the computational capability to apply all operators. Besides, objects must be combined across devices, thus requiring network communication. Sending all basic objects to a central compute server often proves unscalable due to network bottlenecks, or due to the central server not providing sufficient computational power. The alternative is to distribute the execution by mapping each node in the operator tree to one or more servers in the network, including servers that produce and update basic objects and/or servers that are only used for applying operators. One then talks of *in-network stream-processing*. Several in-network stream-processing systems have been developed [9, 8, 17, 22, 15]. These systems all face the same question: where should operators be mapped in the network?

In this paper we study the operator-mapping problem for *multiple concurrent in-network stream-processing applications*. The problem for a single application was studied in [19] for an ad-hoc objective function that trades off application delay and network bandwidth consumption. In a recent paper [6] we have studied a more general objective function, enforcing the constraint that the rate at which final results are produced, or *throughput*, is above a given threshold. This corresponds to a Quality of Service (QoS) requirement of the application that should be met while using as few resources as possible. In this paper we extend the work in [6]

in two ways. First, we study a "non-constructive" scenario, i.e., we are given a set of compute and network elements, and we attempt to use as few resources as possible. Instead, in [6], we studied a "constructive" scenario in which resources could be purchased and the objective was to spend as little money as possible. Second, while in [6] we studied the case of a single application, in this paper we focus on multiple concurrent applications that contend for the servers, each with its own QoS requirement. In this case, higher performance and reduced resource consumption is possible by reusing common sub-expression between operator trees when applications share basic objects [18]. We consider target platforms that are either fully homogeneous, or with a homogeneous network but heterogeneous servers, or fully heterogeneous. Our contributions are: (i) we formalize operator mapping problems for multiple in-network stream-processing applications and give their complexity; (ii) we propose heuristics to solve the problems and evaluate them in simulation.

2 Framework

2.1 Application Model

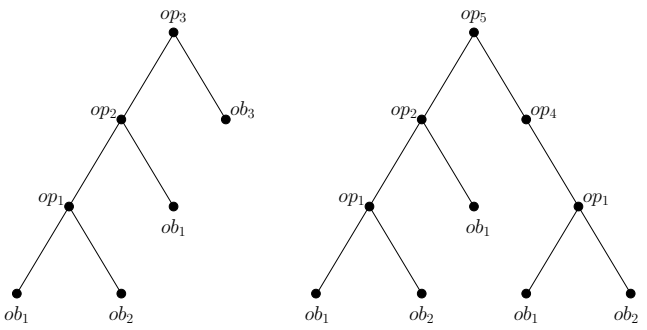


Fig. 1. Sample applications structured as binary trees of operators.

We consider $\mathcal{K}$ applications, each needing to perform several operations organized as a binary tree (see Fig. 1). Operators are taken from the set $\mathcal{OP} = \{op_1, op_2, \dots\}$, and operations are initially performed on basic objects from the set $\mathcal{OB} = \{ob_1, ob_2, \dots\}$. These basic objects are made available and continuously updated at given locations in a distributed network. Operators higher in the tree rely on previously computed intermediate results, and they may also require to download basic objects periodically.

For an operator op_p we define $objects(p)$ as the index set of the basic objects in $\mathcal{OB}$ that are needed for the computation of op_p, if any; and $operators(p)$ as the index set of operators in $\mathcal{OP}$ whose intermediate results are needed for the computation of op_p, if any. Since trees are binary we have $|objects(p)|+|operators(p)| \leq 2$. An application is fully defined by the operator at the root of its tree. For instance, in Fig. 1, we have one application rooted at op_3 and another rooted at op_5. Operator op_1 downloads objects ob_1 and ob_2, while operator op_2 downloads only object ob_1 and requires an intermediate result from operator op_1.

The tree structure of application k is defined with a set of labeled nodes. The i^{th} internal node in the tree of application k is denoted as $n_i^{(k)}$, its associated operator is denoted as $op(n_i^{(k)})$, and the set of basic objects required by this operator is denoted as $ob(n_i^{(k)})$. Node $n_1^{(k)}$ is the root node. Let $op_p = op(n_i^{(k)})$ be the operator associated to node $n_i^{(k)}$. Then node $n_i^{(k)}$ has $|operators(p)|$ child nodes, denoted as $n_{2i}^{(k)}, n_{2i+1}^{(k)}$ if they exist. Finally, the parent of a node $n_i^{(k)}$, for $i > 1$, is the node of index $\lfloor i/2 \rfloor$ in the same tree.

The applications must be executed so that they produce final results, where each result is generated by executing the whole operator tree once, at a target rate. We call this rate the application *throughput*, $\rho^{(k)}$, specified as a QoS requirement for each application. Each operator in the tree of the k^{th} application must compute (intermediate) results at a rate at least as high as $\rho^{(k)}$. Conceptually, operator op_p executes two concurrent threads in steady-state. The *first thread* periodically downloads (or continuously stream) the

most recent copies of the basic objects in $objects(p)$, if any. Each download has a bandwidth cost based on QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). Basic object ob_j has size d_j (in bytes) and needs to be downloaded by the processors that use it for application k with frequency $f_j^{(k)}$. This download consumes an amount of bandwidth equal to $rate_j^{(k)} = d_j \times f_j^{(k)}$ on each involved network link and network card. If a processor requires object ob_j for several applications with different update frequencies, it downloads the object only once at the maximum required frequency $rate_j = \max_k\{rate_j^{(k)}\}$. The *second thread* receives intermediate results computed by $operators(p)$, if any, and performs computation using basic objects it is continuously downloading and/or data received from other operators. The operator produces some output, which is either an intermediate result or the final application result. The computation of operator op_p (to evaluate the operator once) requires w_p operations, and produces an output of size δ_p.

2.2 Platform Model

The distributed network is a fully connected graph (i.e., a clique) interconnecting a set of processors $\mathcal{P}$. Operators are mapped onto these processors. Some processors also hold and update basic objects. Processor $P_u \in \mathcal{P}$ is interconnected to the network via a network card with maximum bandwidth B_u. The network link between two distinct processors P_u and P_v is bidirectional and has bandwidth $b_{u,v} (= b_{v,u})$, shared by communications in both directions. Processor $P_u \in \mathcal{P}$ has compute speed s_u. Processors that only provide basic objects and cannot compute are simply given compute speed 0. Resources operate under the full-overlap, bounded multi-port model [12]: Processor P_u can simultaneously compute, send, and receive data. With the "multi-port" assumption, each processor can send/receive data simultaneously on multiple network links. The "bounded" assumption enforces that the total transfer rate of data sent/received by processor P_u is bounded by its network card bandwidth, B_u.

2.3 Mapping Model and Constraints

The objective is to map internal nodes of application trees onto processors. If only one node is mapped to processor P_u, while P_u computes for the t-th final result it sends to its parent (if any) intermediate results for the $(t-1)$-th final result and it receives data from its children (if any) for computing the $(t+1)$-th final result. All three activities are concurrent (see Section 2.2). If several nodes are mapped to P_u the same overlap happens, but possibly on different result instances. (An operator may be applied for computing the t_1-th final result while another is being applied for computing the t_2-th). When several nodes with the same operator are mapped to the same processor, the operator is computed only once but must satisfy the most stringent application QoS requirements.

A basic object can be duplicated, and thus available and updated at multiple processors. We assume that such duplication is achieved in some application-specific manner (e.g., via a distributed database that enforces sufficient data consistency). In this case, a processor can choose among multiple data sources for a basic object (or perform a local access if the basic object is available locally.) Conversely, if two nodes require the same basic object and are mapped to different processors, they must both download the object (and incur the corresponding network overheads.)

We use an allocation function, a, to denote the mapping of the nodes onto the processors in $\mathcal{P}$: $a(k,i) = u$ if node $n_i^{(k)}$ is mapped to processor P_u. Conversely, $\bar{a}(u)$ is the index set of nodes mapped on P_u: $\bar{a}(u) = \{(k,i) \mid a(k,i) = u\}$. Also, we denote by $a_{op}(u)$ the index set of operators mapped on P_u: $a_{op}(u) = \{p \mid \exists (k,i) \in \bar{a}(u) \;\; op_p = op(n_i^{(k)})\}$. We introduce the following notations:

- $Ch(u) = \{(p,v,k)\}$ is the set of (operator, processor, application) tuples such that processor P_u needs to receive an intermediate result computed by operator op_p, which is mapped to processor P_v, at rate $\rho^{(k)}$; operators op_p are children of $a_{op}(u)$ in the operator tree.
- $Par(u) = \{(p,v,k)\}$ is the set of (operator, processor, application) tuples such that P_u needs to send to P_v an intermediate result computed by operator op_p at rate $\rho^{(k)}$; $p \in a_{op}(u)$ and the sending is done to the parents of op_p in the operator tree.

– $Do(u) = \{(j, v, k)\}$ is the set of (object, processor, application) tuples where P_u downloads object ob_j from processor P_v at rate $\rho^{(k)}$.

Given these notations, we can express constraints for the application throughput: each processor must compute and communicate fast enough to respect the prescribed throughput of each application with nodes allocated to it. The computation constraint is expressed in Eq. 1. Note that each operator is computed only once at the maximum required throughput.

$$\forall P_u \in \mathcal{P} \quad \sum_{p \in a_{op}(u)} \left(\max_{(k,i) \in \bar{a}(u) \mid op(n_i^{(k)}) = op_p} \left(\rho^{(k)} \right) \frac{w_p}{s_u} \right) \leq 1 \,. \tag{1}$$

Communication occurs only when child and parent nodes are mapped on different processors. An operator computing for several applications may send/receive results to/from different processors. If the parent/child nodes corresponding to the different applications are mapped onto the same processor, the communication is done only once, at the most constrained throughput. In expressions below $v \neq u$ since we neglect intra-processor-communications. The first part of Eq. 2 expresses constraints for receiving and the second part for sending:

$$\forall P_u \in \mathcal{P} \quad \sum_{(p,v,k) \in Ch(u)} \left(\rho^{(k)} \frac{\delta_p}{b_{v,u}} \right) \leq 1 \,; \quad \forall P_u \in \mathcal{P} \quad \sum_{(p,v,k) \in Par(u)} \left(\rho^{(k)} \frac{\delta_p}{b_{u,v}} \right) \leq 1 \,. \tag{2}$$

P_u must have enough bandwidth capacity to perform all its basic object downloads, to support downloads of the basic objects it may hold, and also to perform all communication with other processors, all at the required rates. This is expressed in Eq. 3. The first term corresponds to basic object downloads; the second term corresponds to download of basic objects from other processors; the third term corresponds to inter-node communications when a node is assigned to P_u and its parent node is assigned to another processor; and the last term corresponds to inter-node communications when a node is assigned to P_u and some of its children nodes are assigned to another processor.

$$\forall P_u \in \mathcal{P} \quad \sum_{(j,v,k) \in Do(u)} rate_j^{(k)} + \sum_{P_v \in \mathcal{P}} \sum_{(j,u,k) \in Do(v)} rate_j^{(k)} + \sum_{(p,v,k) \in Ch(u)} \delta_p \rho^{(k)} + \sum_{(p,v,k) \in Par(u)} \delta_p \rho^{(k)} \leq B_u \tag{3}$$

Finally, the link between processor P_u and processor P_v must have enough bandwidth capacity to support all possible communications between the nodes mapped on both processors, as well as the object downloads between these processors. Eq. 4 is similar to Eq. 3, but it considers two specific processors:

$$\forall P_u, P_v \in \mathcal{P} \quad \sum_{(j,v,k) \in Do(u)} rate_j^{(k)} + \sum_{(j,u,k) \in Do(v)} rate_j^{(k)} + \sum_{(p,v,k) \in Ch(u)} \delta_p \rho^{(k)} + \sum_{(p,v,k) \in Par(u)} \delta_p \rho^{(k)} \leq b_{u,v} \tag{4}$$

2.4 Optimization Problems

The goal is to achieve a prescribed throughput for each application while minimizing a cost function. Several relevant problems can be envisioned. Proc-Nb minimizes the number of used processors; Proc-Power minimizes the compute capacity and/or the network card capacity of used processors (e.g., a linear function of both criteria); BW-Sum minimizes the sum of the used bandwidth capacities; and BW-Max minimizes the maximum percentage of bandwidth used on all links. Different platform types may be considered depending on resource heterogeneity. We consider the fully homogeneous case ($s_u = s$, $B_u = B$ and $b_{u,v} = b$), which we term Hom. The case in which network links can have various bandwidths is termed Het.

All combinations could be studied, but we will see that Proc-Power on a Hom platform is equivalent to Proc-Nb. Proc-Nb makes more sense in this setting, while Proc-Power is more relevant for Het platforms. Both types of platforms are considered for the BW-Sum and BW-Max problems.

3 Complexity

Problem Proc-Nb is NP-complete in the strong sense even for a simple case: a Hom platform and a single application ($|\mathcal{K}| = 1$), that is structured as a left-deep tree [13], in which all operators take the same amount

of time to compute and produce results of size 0, and in which all basic objects have the same size. We refer the reader to [6] for the proof. It turns out that the same proof holds for Proc-Power on a Hom platform.

The BW-Max problem is NP-hard because downloading objects with different rates on two processors is the same as the NP-hard 2-Partition problem [11]. Here is a sketch of the straightforward proof for a single application. Consider an application in which all operators produce zero-size results, and in which each basic object is used only by one operator. Consider three processors, with one of them holding all basic objects but unable to compute any operator. The two remaining processors are able to compute all the operators, and they are connected to the first one with identical network links. Such an instance can be easily constructed. The goal is to partition the set of operators in two subsets so that the bandwidth consumption on the two network links is as equal as possible. This is exactly the 2-Partition problem.

The BW-Sum problem can be reduced to the NP-hard Knapsack problem [11]. Here is a proof sketch for a single application. Consider the same application as for the proof of the NP-hardness of BW-Max above. Consider two identical processors, A and B, with A holding all basic objects. Not all operators can be executed on A and a subset of them need to be executed on B. Such an instance can be easily constructed. The problem is then to determine the subset of operators that should be executed on A. This subset should satisfy the constraint that the computational capacity of A is not exceeded, while maximizing the bandwidth cost of the basic objects associated to the operators in the subset. This is exactly the Knapsack problem.

All above problems can be solved via linear programming (see [5] for Integer Linear Program formulations). However, they cannot be solved in polynomial time (unless P=NP).

4 Heuristics

In this section we propose polynomial heuristics[3] for solving the Proc-Power problem when considering only the compute capacities of used processors. We propose 6 *heuristics* to map application nodes to processors. Each heuristic can use one of 4 generic *processor selection strategies* to select which processor a node should be mapped to, for a total of (potentially) $6 \times 4 = 24$ heuristics.

We consider two processor selection strategies, each with a *blocking* and a *non-blocking* version. *Blocking* means that once chosen for a given operator op_1, a processor cannot be used later for another operator op_2 unless op_2 is a relative (i.e., father or child) of op_1. *Non-blocking* heuristics impose no such restriction. We obtain four strategies *(S1) Select the fastest processor (blocking)*; *(S2) Select the processor with the fastest network card (blocking)*; *(S3) Select the fastest processor (non-blocking)*; and *(S4) Select the processor with the fastest network card*. Note that the processor and network card speeds used for the selection are computed while accounting for operators that may have already been mapped to servers.

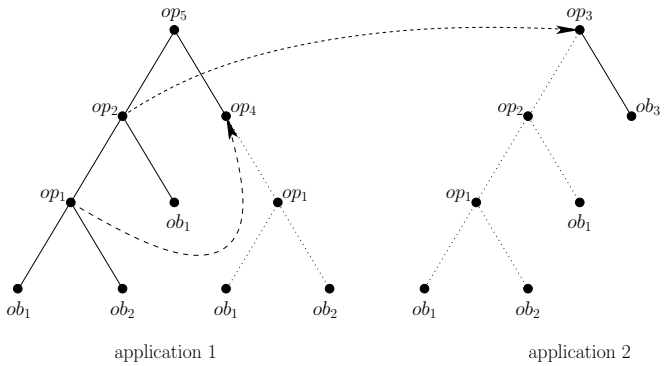


Fig. 2. Example for the reuse of nodes. op_1 is only computed once and its result is reused for the computation of op_2 and op_4. op_3 uses the result of op_2 in application 1 for its computation.

[3] To ensure the reproducibility of our results, the code for all heuristics is available on the web [2].

All our heuristics, except for the first one below, attempt to re-use results from common operator sub-trees across applications. For this purpose they try to add additional communications as shown in Fig. 2 on an example. The processor that computes the leftmost op_1 in application 1 sends its result not only to the processor that computes op_2, but also to the processor that computes op_4. The operator op_1 on the right of application 1 no longer has to be computed. In the same way, we save the whole computation of the subtree rooted at op_2 in application 2 when we add the communication between op_2 in application 1 and op_3 in application 2. We consider the following 6 heuristics:

• **(H1) RandomNoReuse** – H1 randomly picks the next node to map. If the node's parent is already mapped, H1 tries to map the node to the same processors. If unsuccessful, it makes similar attempts with the node's children. If unsuccessful, then H1 choose a new processor according to the selected processor selection strategy. If unsuccessful, then H1 fails.

• **(H2) Random** – H2 also picks a node randomly but attempts to reuse sub-trees across applications. If the node's operator has not already been mapped, possibly for another application, H2 defaults to H1. Otherwise if the node's operator has already been mapped somewhere else in the forest, H2 tries to add a communication from the already mapped operator to the father of the current node to reuse the common result. In this case, H2 marks the whole subtree (rooted at the operator) as mapped. Otherwise, H2 chooses a new processor according to the selected processor selection strategy. If unsuccessful, then H2 fails.

• **(H3) TopDownBFS** – H3 performs a breadth-first-search (BFS) traversal of all application trees, using an artificial node at which all application trees are rooted. For each node, H2 checks whether its operator has not been mapped yet and whether its father's has. In this case, H3 tries to map the operator on the same processor as its father, and in case of success continues the BFS traversal. If the node's operator has already been mapped, H3 tries to add a communication link between the mapped operator and the node's father: the mapped operator sends its result not only to its father but also to the node's father. If none of these two conditions holds, or if the mapping was not possible, H3 picks a processor according to the the processor selection strategy. If the mapping is successful, the BFS traversal continues, otherwise H3 fails.

• **(H4) TopDownDFS** – H4 uses the same mechanism as H3, but with a depth-first-search (DFS).

• **(H5) BottomUpBFS** – Like H3, H5 performs a BFS traversal of the application trees. For each node, H5 verifies whether it's operator has already been mapped. In this case a communication link is added (if possible), connecting the mapped operator and the node's father. If the operator is not yet mapped and if it has children, H5 tries to map the operator to one of its children's processors. If unsuccessful, or if the operator is at the bottom of a tree, H5 tries to map the operator onto a new processor chosen according to the processor selection strategy. If the mapping is successful, the BFS traversal continues, otherwise H5 fails.

• **(H6) BottomUpDFS** – H6 is similar to H5, but uses a DFS traversal. This adds complexity as more cases need to be considered. For each node H6 checks if its operator has already been mapped and none of its children has. In this case H6 goes up in the tree until it reaches the last node n_1 such that there exists a node n_2 somewhere else in the forest whose operator is already mapped, and such that $op(n_1) = op(n_2)$. In this case H6 tries to add a communication between n_2 and the n_1's father to share a sub-tree. If the children have already been mapped H6 simply tries to map the operator to one of the children's processors. If this is not possible, or if the additional communication was not possible, or again if the operator has not been mapped anywhere in the forest, H6 tries to map the operator onto a new processor chosen according to the processor selection strategy. Otherwise H6 fails.

5 Experimental Results

We have conducted several experiments to assess the performance of the different heuristics described in Section 4. In particular, we are interested in the impact of node reuse on the number of solutions found by the heuristics. Except for Experiment 1, all application trees are fixed to a size of at most 50 operators, and except for Experiment 5, we consider 5 concurrent applications. The leaves in the tree correspond to basic objects, and each basic object is chosen randomly among 10 different types. The size d of each object type is also chosen randomly and varies between 3MB and 13MB. The download frequencies of objects for each application, f, as well as the application throughput, ρ, are chosen randomly such that $0 < f \leq 1$ and $1 \leq \rho \leq 2$. The operands of operators are also chosen randomly. In all experiments, except Experiment 4,

the computation amount w_i for an operator lies between 0.5MFlop/sec and 1.5MFlop/sec, and the output size of each operator δ_i is randomly chosen between 0.5MB and 1.5MB.

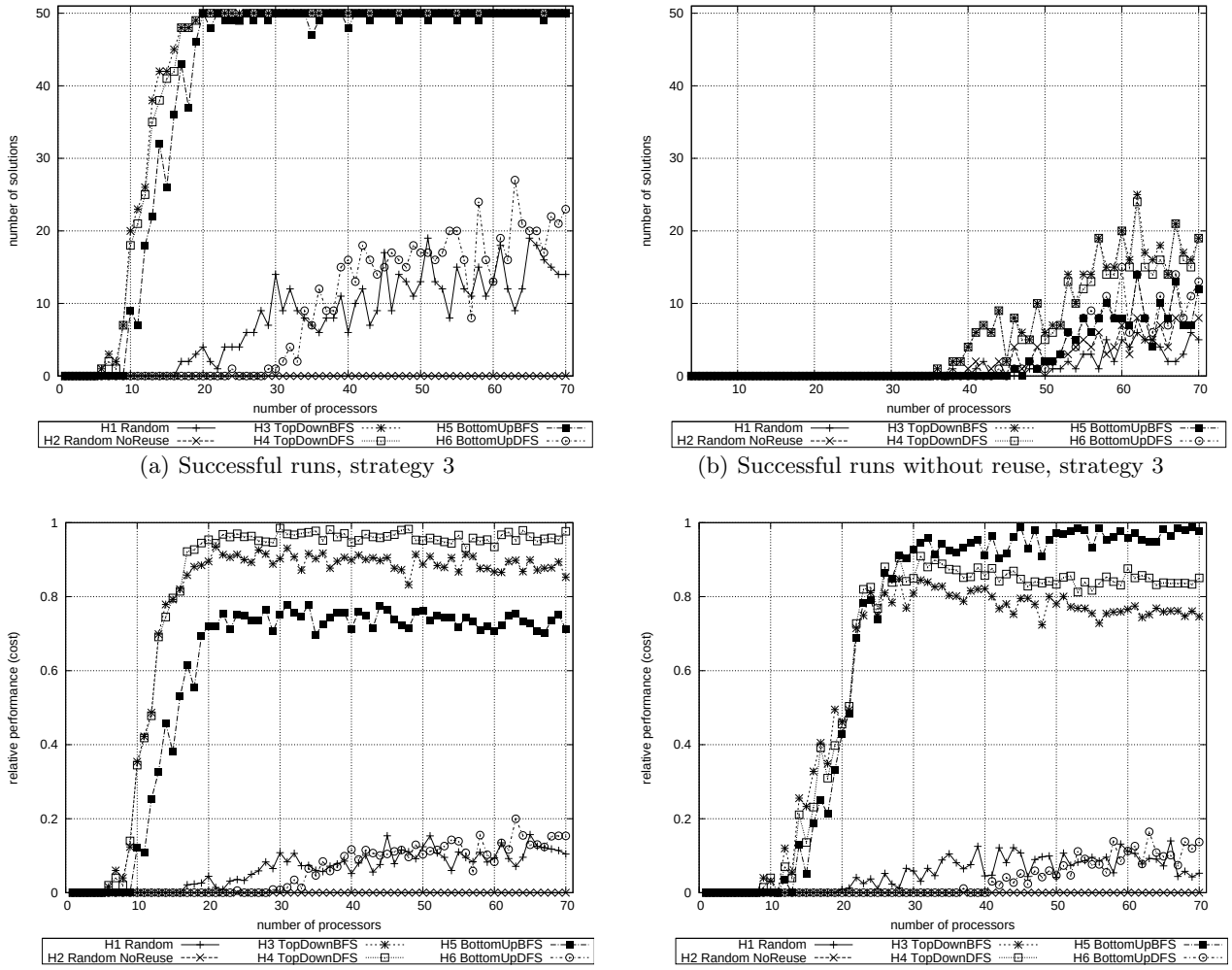
Throughout most of our experiments we use the following platform configuration (variants will be mentioned explicitly when needed.) We dispose of 30 processors. Each processor is equipped with a network card of bandwidth between 50MB and 180MB. We use the same range for processor compute speed : 50MIPS to 180MIPS. Processors are interconnected via heterogeneous communication links, whose bandwidths are between 60MB/s and 100MB/s. The 10 different types of objects are randomly distributed over the processors. Execution time and communication time are scaled units, thus execution time is the ratio between computation amount and processor speed, while communication time is the ratio between object size (or output size) and link bandwidth.

We study the relative performance of each heuristic compared to the best solution found by any heuristic. This allows to compare the cost, in amount of resources used, of the different heuristics. The relative performance for heuristic h is obtained by: $\frac{1}{|runs|}\sum_{r=1}^{|runs|} a_h(r)$, where $a_h(r) = 0$ if heuristic h fails in run r and $a_h(r) = \frac{cost_{best}(r)}{cost_h(r)}$. $cost_{best}(r)$ is the best solution cost returned over all heuristics for run r, and $cost_h(r)$ is the cost in the solution returned by heuristic h. The number of runs is fixed to 50 in all experiments. The complete set of results is available on the web [1].

Experiment 1: Number of Processors – We vary the number of processors from 1 to 70. Figure 3(a) shows the number of successes of the different heuristics using strategy S3. Between 1 and 20 processors, the number of solutions steeply increases for H4, H3 and H5 and for higher numbers of processors all three heuristics find solutions for most of the 50 runs. H6 finds solutions when more than 30 processors are available. H2 already finds solutions when only 20 processors are available, but for the runs with more than 30 processors, it finds fewer solutions than H6. H1 does not find any solution. To summarize, H3 finds the most solutions, shortly followed by H4 and H5. Comparing the success rates of the different selection strategies, all heuristics find the most solutions using strategy S3, followed by strategy S4, strategy S2, and finally strategy S1. But the differences are small. More interesting is the relative performance of the heuristics using the different processor selection strategies in comparison to the number of solutions. Figure 3(c) shows the relative performance using strategy S3. Comparing with Fig. 3(d), we conclude that for the same number of successful runs, the performances of the heuristics significantly differ according to the processor selection strategy. Using strategy S3 (and also strategies S2 and S4), H4 performs better than H3, which performs better than H5. However, H5 outperforms both TopDown heuristics when strategy S1 is used. The performance of H6 and of the random ones mirrors exactly the number of successful runs. As for the heuristics without reuse of common subtrees, we see that they do not find results until at least 35 processors are available (with S3) or even 60 (with S2). Independently of the processor selection strategy, both TopDown heuristics outperform all other heuristics in success and performance, but the results are poor (see Fig. 3(b)).

Experiment 2: Number of Applications – In this set of experiments we vary the number of applications, $\mathcal{K}$. As the number of application increases, all heuristics are less successful with strategies S1 and S2 than with strategies S3 and S4, and relative performance is poorer as well. Regardless of the strategy used, both TopDown heuristics show a better relative performance than H5, with the only exception when using strategy S1 with a small number of applications. H6 and both random heuristics perform poorly. For instance, H6 only finds solutions with up to 4 applications. The best strategy seems to be strategy S3 in combination with H3 for more than 10 applications and H4 for less than 10 applications.

Experiment 3: Application Size – When increasing the application sizes, strategy S3 is the most robust. Up to application sizes of 40 operators, the other strategies are competitive, but for bigger applications both TopDown heuristics and H5 achieve the best relative performance and find the most solutions. The success ranking of the three heuristics is the same, independently of the strategy: H3 finds more solutions than H4, which, in turn, finds more solutions than H5. H1 finds solutions for applications with fewer than 20 operators, H5 up to 40 operators, and H2 up to 50 operators, but the number of solutions from the latter is poor. As far as relative performance is concerned, both TopDown heuristics achieve the best results for application sizes larger than 20 using strategy S3. H6 is competitive when using strategy S1 for applications with less than 40 operators. Heuristics that do not reuse common subtrees no longer find results when application



(a) Successful runs, strategy 3

(b) Successful runs without reuse, strategy 3

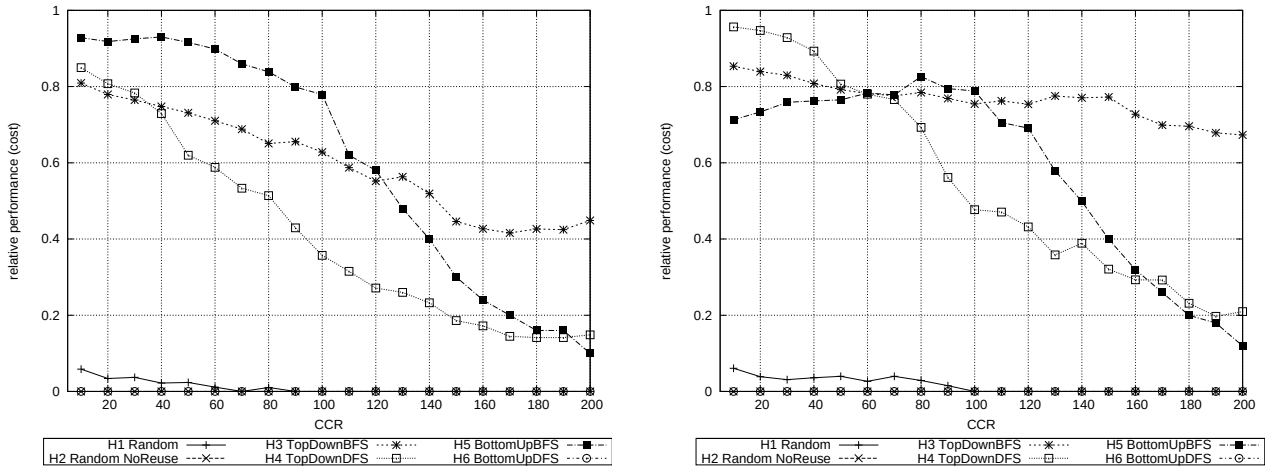
(c) Relative performance, strategy 3

(d) Relative performance, strategy 1

Fig. 3. Experiment 1: Increasing number of processors.

sizes exceed 40 operators. In summary, TopDown heuristics perform best, and the best strategy is one of the two non-blocking ones (S3 or S4).

Experiment 4: Communication-to-Computation Ratio (CCR) – We define CCR as the ratio between the mean amount of communications and the mean amount of computations, where the communications correspond to the output sizes of operators (δ_i) and the computations to the computational volume w_i of the operators. When increasing the CCR, strategies S3 and S4 react very sensitively. H3, H4 and H5 have a 100% success rate for CCR $\leq$ 60, but the success decreases drastically until no solution is found at all for a CCR of 180 (using strategy S2, H3 still finds 32 solutions). H6 is largely outperformed by H2, and H1 fails completely. In this experiment, strategy S2 seems to be the most successful processor selection strategy. H3 achieves the best results, followed by H5 for CCR $<$ 120, and by H4 for CCR $>$ 120. Interestingly, the relative performances of the heuristics using the different strategies do not mirror their success rates. Compare Figs. 4(a) and 4(b): H5 finds fewer solutions using strategy S1 than S2, but its relative performance using strategy S1 and CCR smaller than 80 is better than when using strategy S2. Furthermore, H3 using strategy S1 always finds the most solutions of all heuristics, but its relative performance is only the best when the CCR becomes bigger than 120. Also, H4 finds fewer solutions than H3 and H5 using strategy S2 and CCR= 30, but its relative performance is the best.



(a) Relative performance, strategy 1

(b) Relative performance, strategy 2

Fig. 4. Experiment 4: Communication-Computation Ratio CCR. Relative performance.

Experiment 5: Similarity of Applications – We now use only two applications for each run and the processing platform is smaller, consisting of only 10 processors. We study our heuristics when applications are very similar or very dissimilar. For this purpose we create applications that differ in more and more operators. Strategies S1 and S2 are more sensitive to application differences and we observe the following ranking: strategy S3 > strategy S4 > strategies S1 and S2, which have similar success rates.The ranking of the heuristics within the different strategies is the same: H3 is the most successful, followed by H4, and H5. H6 and H2 are in fourth place, while H1 fails. H3 has the best relative performance using the blocking strategies, whereas with non-blocking strategies H4 achieves the best results, which is important as its success rate is slightly poorer. H5 always ranks third.

Summary of Experiments – Our results show that a random approach for multiple applications is not feasible. Not reusing results from common subtrees dramatically limits the success rate and also the quality of the solution in terms of cost. The TopDown approach turns out to be the best, and in most cases BFS traversal achieves the best result. The BottomUp approach is only competitive using a BFS traversal. The DFS traversal seems unable to reuse results efficiently (it often finds itself with no bandwidth left to perform necessary communications.) Furthermore we see a strong dependency of the processor selection strategy on solution quality. The blocking strategies outperform the non-blocking strategies when the CCR is large. Overall, H3 in combination with strategy S3 proves to be a solid combination.

6 Conclusion

We have studied the operator mapping problem of multiple concurrent in-network stream-processing applications onto a collection of heterogeneous processors. These stream-processing applications come as a set of operator trees, that have to continuously download basic objects at different sites of the network and at the same time have to process this data to produce some final result. We have identified four relevant optimization problems. All are NP-hard but can be formalized as integer linear programs. We have focused on one of these optimization problems, for which we have designed several polynomial-time heuristics. We have evaluated these heuristics in simulation. Our experiments show the importance of node reuse across applications. Reusing nodes leads to an important number of additional solutions, and also the quality of the solutions improves considerably. We conclude that top-down traversal of the application trees is more efficient than bottom-up traversal. In particular, the combination of a top-down traversal with a breadth-first search (i.e., our heuristic H3) achieved good results across the board.

As future work, we could develop heuristics for the other optimization problems defined in Section 2.4. We could also envision a more general cost function $w_{i,u}$ (time required to compute operator i onto processor u), in order to express even more heterogeneity. This would lead to the design of more sophisticated heuristics. Also, we believe it would be interesting to add a storage cost for objects downloaded onto processors, which could lead to new objective functions. Finally, we could address more complicated scenarios with many (conflicting) relevant criteria to consider simultaneously, some related to performance (throughput, response time), some related to fault-tolerance (replicating some computations for more reliability), and some related to environmental costs (resource costs, energy consumption).

References

[1] Diagrams of all experiments. `http://graal.ens-lyon.fr/~vsonigo/code/query-multiapp/diagrams/`.

[2] Source Code for the Heuristics. `http://graal.ens-lyon.fr/~vsonigo/code/query-multiapp/`.

[3] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[4] B. Badcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Intl. Conf. on Very Large Data Bases*, pages 456–467, 2004.

[5] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert. Resource Allocation for Multiple Concurrent In-Network Stream Processing Applications. Research Report 2009-07, LIP, ENS Lyon, France, Feb. 2009.

[6] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert. Resource Allocation Strategies for Constructive In-Network Stream Processing. In *Proceedings of APDCM'09, the 11th Workshop on Advances in Parallel and Distributed Computational Models*. IEEE, 2009.

[7] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of ICDE*, 2002.

[8] L. Chen, K. Reddy, and G. Agrawal. GATES: a grid-based middleware for processing distributed data streams. *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 192–201, 4-6 June 2004.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, January 2003.

[10] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high-performance network monitoring with an SQL interface. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–633, 2002.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[12] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[13] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[14] J. Kräme and B. Seeger. A Temporal Foundation for Continuous Queries over Data streams. In *Proceedings of the Intl. Conf. on Management of Data*, pages 70–82, 2005.

[15] D. Logothetis and K. Yocum. Wide-Scale Data Stream Management. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[16] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 491–502, 2003.

[17] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An Architecture for Internet-scale Sensing Services.

[18] V. Pandit and H. Ji. Efficient in-network evaluation of multiple queries. In *HiPC*, pages 205–216, 2006.

[19] P. Pietzuch, J. Leflie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–60, 2006.

[20] B. Plale and K. Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, 2003.

[21] U. Srivastava, K. Munagala, and J. Widom. Operator Placement for In-Network Stream Query Processing. In *Proceedings of the 24th ACM Intl. Conf. on Principles of Database Systems*, pages 250–258, 2005.

[22] R. van Rennesse, K. Birman, D. Dumitriu, and W. Vogels. Scalable Management and Data Mining Using Astrolabe. In *Proceedings from the First International Workshop on Peer-to-Peer Systems*, pages 280–294, 2002.