Parallel Computing 37 (2011) 331–348

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Resource allocation for multiple concurrent in-network stream-processing applications

Anne Benoit [a,*], Henri Casanova [b], Veronika Rehn-Sonigo [c], Yves Robert [a]

[a] LIP, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
[b] University of Hawai'i at Manoa, 1680 East-West Road, Honolulu, HI 96822, USA
[c] LIG, ENSIMAG, 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France

ARTICLE INFO

Article history:
Available online 10 October 2010

Keywords:
In-network stream-processing
Trees of operators
Multiple concurrent applications
Operator mapping
Polynomial heuristics

ABSTRACT

This work investigates the operator mapping problem for in-network stream-processing. In a stream-processing application, a tree of operators is applied, in steady-state mode, to datasets that are continuously updated at different locations in the network. The goal is to generate updated final results at a desired rate. In in-network stream-processing, dataset updates and operator computations are performed by servers distributed in a network. We consider the problem of mapping operators to these servers in the case of multiple concurrent stream-processing applications. In this case, different operator trees corresponding to different applications may share common subtrees, so that intermediate results can be reused by different applications. This work provides complexity results for different versions of the operator mapping problem, which can be formulated as integer linear programs. Several polynomial-time heuristics are proposed for a particularly relevant version of the problem, which is NP-hard. These heuristics are compared and evaluated via simulation. The results demonstrate the importance of mapping the operators to appropriate processors, and the importance of sharing common sub-trees across operator trees.

1. Introduction

We consider applications structured as trees of operators, where leaves correspond to basic data objects distributed across a network. Each internal node in the tree denotes the aggregation and combination of the data from its children via the application of an operator, 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 updated final results at some desired rate.

The above problem is called *stream processing* [5] and arises in several domains. One such domain is the acquisition and refinement of data from a set of sensors [28,22,7]. For instance, [28] 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 [14,29,12]. 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 [4,20,9,26,19].

* Corresponding author.
E-mail addresses: Anne.Benoit@ens-lyon.fr (A. Benoit), henric@hawaii.edu (H. Casanova), veronika.sonigo@imag.fr (V. Rehn-Sonigo), Yves.Robert@ens-lyon.fr (Y. Robert).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.09.005

In practice, the execution of the operators must be distributed over the network. In some cases the servers that produce the basic objects do not have the computational capabilities necessary for applying all operators. Besides, objects must be combined using operators, thus requiring network communication unless all objects are generated on the same server. Sending all basic objects to a central 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 [3,11,17,10,23,29,8,21].

These systems all face the same question: where should operators be mapped in the network? The operator mapping problem for a single application was studied in [25] for an ad hoc objective function that trades off application delay and network bandwidth consumption. In a recent paper [6], we also studied the single application problem but proposed a more general objective function, enforcing that the rate at which final results are produced, or *throughput*, be 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. Also, we focused on a "constructive" scenario in which resources could be purchased with the objective being to spend as little money as possible.

In this paper, we extend our previous work [6] in two ways. First, we focus on multiple concurrent applications that contend for the servers and the network, each with its own QoS requirement. Second, 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. The constructive scenario is arguably more relevant to the case of a single application for which one purchases resources dedicated to the application. Instead, executing multiple concurrent applications is more typical of an environment in which existing resources are already deployed and must be shared among competing applications. This motivates the use of a non-constructive scenario in this paper.

With multiple concurrent applications, it is possible to gain higher performance and reduced resource consumption by reusing common sub-trees between operator trees when applications share basic objects. Based on this observation, which was made originally in [24], we exploit sub-tree reuse while developing heuristics. More precisely, our main contributions in this work are the following:

- We formalize several versions of the operator mapping problems for multiple in-network stream-processing applications and give their complexity.
- We propose heuristics for solving one particularly relevant version of the operator mapping problem.
- We evaluate our heuristics through an extensive set of simulations; we identify one heuristic that leads to promising results; and we provide general guidelines for heuristic design.

The rest of this paper is organized as follows. In Section 2 we define our application and platform models, and we formalize several versions of the operator mapping problem. In Section 3 we discuss the computational complexity of our mapping problems. In Section 4 we propose several heuristics, which we evaluate in Section 5. Finally we conclude in Section 6 with a summary of our results and future directions.

2. Framework

We study the operator-mapping problem for *multiple* and *concurrent* in-network stream-processing applications. We first describe the application model and the execution platform. We then detail the mapping model and constraints that should be enforced. Finally, we introduce several relevant optimization problems.

2.1. Application model

We consider $\mathcal{K}$ applications. Each application needs to perform several operations organized as a binary tree of operators (see Fig. 1). Operators are in the set $\mathcal{OP} = \{op_1, op_2, \ldots\}$, and operations are initially performed on basic objects in the set $\mathcal{OB} = \{ob_1, ob_2, \ldots\}$. 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 need to download basic objects periodically. For instance, in Fig. 1, operator op_1 needs to download basic objects ob_1 and ob_2, but operator op_2 downloads only basic object ob_1 and relies on results computed by operator op_1.

Note that an operator can occur multiple times in an operator tree. In our example, operator op_1 occurs twice in application 1 in Fig. 1. Therefore, this operator can be evaluated only once and its result can be used both as an input to op_2 and op_4. This reuse is shown in Fig. 2 with a dashed arrow. An operator can also be shared between operator trees. In our example operator op_2 occurs in both applications. Therefore, one can save the whole computation of the subtree rooted at op_2 and reuse its result for both applications. This is shown with another dashed arrow in Fig. 2. In the extreme, if all operators' results are reused as much as possible, then each operator is only evaluated once. This is the case in Fig. 2 once all tree nodes connected to their parents with dotted edges are removed. Note that result reuse may entail extra network

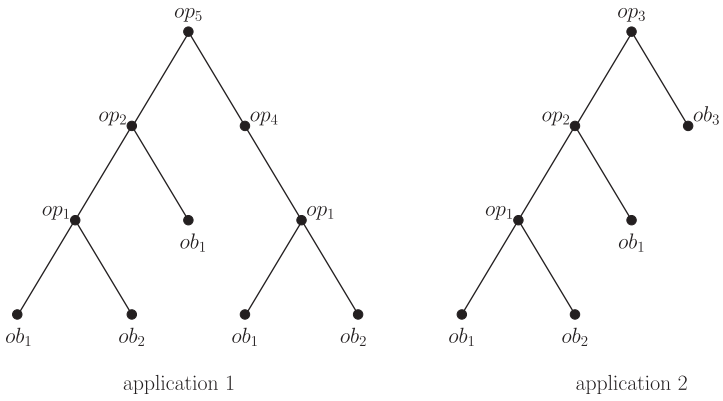


Fig. 1. Two example applications structured as binary trees of operators.

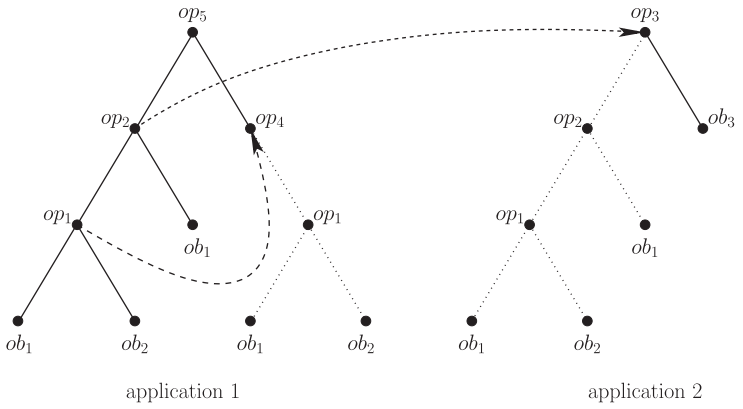


Fig. 2. Example of node reuse. In application 1, the operator op_1 is only computed once, and its result is reused for the computation of op_2 and op_4. The computation of operator op_3 in application *2* reuses the result of op_2 in application 1.

communications since the operators can be executed on different processors. Therefore, due to network bandwidth constraints, it may not be possible to reuse all results.

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 $|objects(p)| + |operators(p)| \leqslant 2$. An application is fully defined by the operator at the root of its tree. For instance, in Fig. 1, application 1 is rooted at op_5 and application 2 is rooted at op_3.

The tree structure of application k is defined with a set of labeled nodes. The ith 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 $|operators(p)|$ children nodes, denoted as $n_{2i}^{(k)}$ and $n_{2i+1}^{(k)}$ if they exist. Finally, the parent of a node $n_i^{(k)}$, for $i > 1$, is the node with 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 kth application must compute (intermediate) results at a rate at least as high as $\rho^{(k)}$. Conceptually, a processor computing operator op_p executes two concurrent activities in steady-state:

- *Basic object download* – The processor must periodically download (or continuously stream) the most recent copies of the basic objects in $objects(p)$, if any. Each download consumes bandwidth 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 should 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 at the maximum required frequency, leading to bandwidth consumption $rate_j = \max_k\{rate_j^{(k)}\}$.

- *Operator computation* – The processor must receive intermediate results computed by *operators*(p), if any. It then must compute operator op_p using these intermediate results and the basic objects it continuously downloads. The computation of operator op_p (to evaluate the operator once) requires w_p operations, and produces an output of size δ_p. This output, for each operator tree in which it occurs and for which it is not the root, must be sent to its parent operators. Receiving and sending operator results from children operators and to parent operators consumes bandwidth on each involved network link and network card, and must be allocated enough bandwidth to achieve the application's prescribed throughputs.

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. 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 operators require the same basic object and are mapped to different processors, they must both download the object and incur the corresponding network overheads.

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 computing speed s_u, in operations per second. Processors that only provide basic objects and cannot compute are simply given a speed of 0.

Note that we do not need to have a single physical link between every processor pair. Instead, we may have a switch, or even a path composed of several physical links, to interconnect P_u and P_v; in the latter case we would retain the bandwidth of the slowest link in the path for the value of $b_{u,v}$.

Resources operate under the full-overlap, bounded multi-port model [16]: 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. This can be implemented in practice via multiple threads. For instance, a different thread could be used for each basic object download, for each operator result reception, for each operator computation, and for each operator result sending.

2.3. Mapping model and constraints

The objective is to map internal nodes of application trees onto processors. As explained in Section 2.1, if the operator associated to a node requires basic objects, the processor in charge of this internal node must continuously download up-to-date basic objects, which consumes bandwidth on its network card.

If only one node is mapped to processor P_u, while P_u computes part of the tth final result, it sends to its parent (if any) intermediate results needed for the $(t-1)$th final result, and it receives data from its children (if any) for computing part of the $(t+1)$th final result. Recall that all three activities are concurrent (see Section 2.2). If several nodes are mapped to P_u, then the same overlap happens, but possibly on different result instances. An operator may be applied for computing the t_1th final result, while another is being applied for computing the t_2th. 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.

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: $\bar{a}(u) = \{(k,i) | a(k,i) = u\}$.

Also, we denote by $a_{op}(u)$ the index set of operators mapped on P_u:

$$a_{op}(u) = \{p | \exists (k,i) \in \bar{a}(u) \quad \text{with} \quad op_p = op(n_i^{(k)})\}.$$

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 such that P_u downloads object ob_j from processor P_v at rate $\rho^{(k)}$.

Given these notations, we can now express constraints for the application throughput: each processor to which nodes are allocated must compute and communicate fast enough to achieve the prescribed throughput of each application, without exceeding compute and network bandwidth capacities.

The computation constraint is expressed in Eq. (1). Note that each operator is computed only once at the maximum required throughput.

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

Communication occurs between two (child and parent) operators only when the two operators are mapped on different processors. An operator that is computed for several applications may send/receive results to/from different processors. If the parent/child operators corresponding to the different applications are mapped onto the same processor, the communication is done only once, at the most constrained throughput. In all expressions hereafter, $u \neq v$ since we neglect intra-processor communications. We have two types of communication constraints, one for network cards and one for network links.

P_u's network card 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 communications with other processors, all at the required rates. This is expressed in Eq. (2). The first term corresponds to basic object downloads; the second term corresponds to downloads 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.

$$\forall P_u \in \mathcal{P}, \quad \sum_{(j,v,k) \in Do(u)} rate_j^{(k)} + \sum_{P_v \in \mathcal{P}} \sum_{(j,u,k) \in Do(v)} rate_j^{(k)} + \sum_{(p,v,k) \in Ch(u)} \delta_p \rho^{(k)} + \sum_{(p,v,k) \in Par(u)} \delta_p \rho^{(k)} \leqslant B_u. \tag{2}$$

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. This is expressed in Eq. (3), which is similar to Eq. (2) but considers two specific processors:

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

2.4. Optimization problems

The goal is to achieve the prescribed throughput for each application while minimizing a cost function. Several relevant optimization problems are defined:

- Proc-Nb: minimize the number of processors in use;
- Proc-Power: minimize the compute capacity and/or the network card capacity of processors processor in use (e.g., a linear function of both criteria);
- BW-Sum: minimize the sum of the used bandwidth capacities;
- BW-Max: minimize the maximum used bandwidth capacity over all links.

We have developed our problem formulations with the assumption of a fully heterogeneous platform. However, one can consider simpler cases, including a fully homogeneous platform ($s_u = s$, $B_u = B$, and $b_{u,v} = b$), in which case Proc-Power is equivalent to Proc-Nb.

3. Complexity results

In this section, we investigate the complexity of the four optimization problems introduced in Section 2.4.

- Problem Proc-Nb is NP-complete in the strong sense even for a simple case: a fully homogeneous platform and a single application ($|\mathcal{K}| = 1$), structured as a left-deep tree [18], 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.
- The previous result also holds for Proc-Power since it is equivalent to Proc-Nb on a fully homogeneous platform.
- The BW-Max problem is NP-hard because downloading objects at different rates on two processors is the same as the NP-hard 2-Partition problem [15]. Here is a sketch of the straightforward proof for a single application. Consider an application in which all operators have the same computational cost and produce zero-size results, and in which each basic object is used by only one operator. All basic objects are of different sizes. Consider three processors, with one of them holding all basic objects but unable to compute any operator (its compute speed is 0). The two remaining processors are identical and must each compute half of the operators to meet the application's throughput requirements. These two processors are connected to the first one with identical network links. Such an instance can be easily constructed. The goal is then to partition the set of operators into two subsets so that the bandwidth consumption on the two network links is equal. In other words, one must partition the set of basic objects into two sets, so that the sum of all basic object sizes in the set is the same for both sets. This is exactly the 2-Partition problem.

- The BW-Sum problem can be reduced to the NP-hard Knapsack problem [15]. Here is a sketch of the proof 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. To satisfy the application's throughput requirement, not all operators can be executed on A. Consequently, some 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.

We conclude that, unless P = NP, the operator mapping problems cannot be solved in polynomial time. It turns out that integer linear program formulations can be given for all these problems, and we refer the reader to [27] for details. A typical use for such formulations is to relax integer variables to be rational, thereby making it possible to solve the relaxed program in polynomial time and obtaining a bound of the optimal solution of the non-relaxed program. Unfortunately, in the case of our operator mapping problems, the linear program formulations are enormous: for k applications each with n nodes, q operators, and p processors, these linear programs have $O(kp^2(q^2 + n^2))$ variables and $O(kp^2q^2n^2)$ constraints. The efficient commercial linear solver CPLEX [13] is unable to provide solutions for our linear programs within reasonable amounts of time unless small and non-representative values for k, n, q, and p are used. In fact, in most cases CPLEX cannot even load the linear program description file. As a result, in the rest of this paper, we focus on the development of polynomial-time heuristics without comparison to a bound on optimal.

4. Heuristics

We have described four versions of the operator mapping problem. For the rest of this paper we focus on the Proc-Power problem, i.e., on minimizing the CPU power of enrolled resources. While minimizing the bandwidth is interesting as well, we feel that Proc-Power is particularly relevant as its objective is tightly related to minimizing energy consumption, which is a popular goal for large-scale, long-running distributed applications.

Since the Proc-Power problem is NP-hard, we propose polynomial-time heuristics to find sub-optimal but reasonable solutions. To ensure the reproducibility of our results, the code for all heuristics is available on the web [2].

Our algorithms are based on six *heuristics* to map application nodes to processors. Each heuristic can use one of four generic *processor selection strategies* to select the processor to which a node should be mapped. Overall we have potentially $6 \times 4 = 24$ algorithms.

Let us first describe the four processor selection strategies. We consider two main processor selection strategies, each with a *blocking* and a *non-blocking* version. *Blocking* means that once chosen for a given operator op_i, a processor cannot be used later for another operator op_j unless op_j is a direct relative (i.e., parent or child) of op_i. *Non-blocking* heuristics impose no such restriction. The four strategies are as follows:

(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 (non-blocking).

In the above, processor and network card speeds are computed while accounting for nodes and, thus, operators, that may have already been mapped to servers.

We describe below our node mapping heuristics. Except for H1, all these heuristics attempt to reuse results from common operator sub-trees across applications.

(H1) **RandomNoReuse** – H1 randomly picks the next node, n, to map to a processor. If n's parent is already mapped to a processor, H1 tries to map n to the same processor. If unsuccessful, H1 makes similar attempts with each of n's children. If unsuccessful, then H1 chooses a new processor according to the processor selection strategy in use. If no feasible processor can be found, then H1 fails.
(H2) **Random** – H2 also randomly picks the next node, n, but attempts to reuse results from operator sub-trees common across applications. If n's operator has not already been mapped to a processor, possibly for another application, then H2 works similarly to H1. Otherwise if n's operator has already been mapped to a processor, then H2 tries to add a communication from the already mapped operator to n's parent node in an attempt to reuse the common result. In this case, H2 marks the whole subtree rooted at node n as mapped. Otherwise, H2 chooses a new processor according to the processor selection strategy in use. If no feasible processor can be found, 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, n, H3 checks whether its operator has not been mapped to a processor yet and whether its parent's operator has been mapped to a processor. In this case, H3 tries to map n on that same processor, and, in case of success, continues the BFS traversal. If n's operator has already been mapped, H3 tries

to add a communication between the mapped operator and n's parent: the operator sends its result not only to its parent but also to n's parent. If none of these two conditions holds, or if the mapping was not feasible, H3 picks a processor according to the processor selection strategy in use. If no feasible processor can be found, then H3 fails.

(H4) **TopDownDFS** – H4 is similar to H3 but uses a depth-first-search (DFS) instead of a BFS.

(H5) **BottomUpBFS** – Like H3, H5 performs a BFS traversal of the application trees. For each node, n, H5 checks whether its operator has already been mapped to a processor. In this case a communication is added (if possible), going from the mapped operator to n's parent. If the operator is not yet mapped to a processor and if it has children, H5 tries to map the operator to one of its children's processors. If unsuccessful, or if n is at the bottom of a tree, H5 tries to map it onto a new processor chosen according to the processor selection strategy in use. If a feasible processor is found then 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, n, H6 checks if its operator has already been mapped and none of its children have. In this case H6 goes up in the tree until it reaches the last node n_1 such that there exists another node n_2 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 n_1's parent to reuse results from a common sub-tree. More precisely n_2's processor has to send n_2's result not only to n_2's parent but also to n_1's parent. If n's children have already been mapped to processors, H6 simply tries to map n to one of these processors. If this is not possible, or if the additional required communication is not feasible, or again if the operator has not been mapped to any processor, then H6 tries to map the operator onto a new processor chosen according to the processor selection strategy in use. If no feasible processor can be found, then H6 fails.

5. Experimental results

In this section we present results obtained via simulation to evaluate and compare the algorithms proposed in Section 4. In particular, we are interested in the impact of node reuse on the number of solutions found by the heuristics. We describe our simulation methodology and results from five experiments.

5.1. Simulation methodology

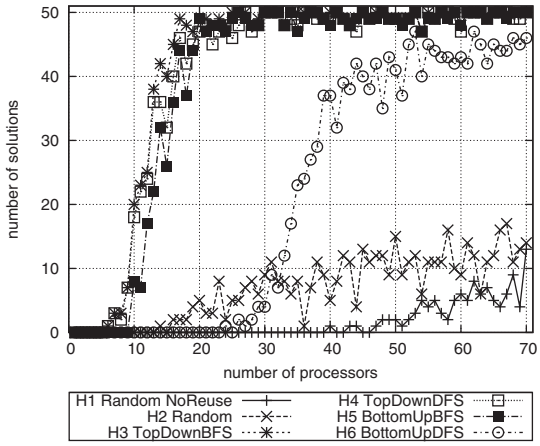
Given the lack of published representative models of query-streaming workloads, we opt for generating many random simulation scenarios using arbitrary but reasonable ranges for the parameters that define the operator mapping problem. In the following, random values are generated using uniform probability distributions. Except when stated otherwise, we consider five concurrent applications, and each tree of operators contains up to 50 operators. The leaves in a tree correspond to basic objects, and each basic object is chosen randomly among 10 different object types, with each type defined by its size. The size of each object type is chosen randomly between 3 and 13 MB. The download frequencies of objects for each application, f, is chosen randomly between 0 and 1. Each application's required throughput, ρ, is chosen randomly between 1 and 2. The operands of operators are also chosen randomly. In all experiments, except Experiment 4, the computation amount of each operator, w_i, lies between 0.5 and 1.5 MFlop, and the output size of each operator, δ_i, is randomly chosen between 0.5 and 1.5 MB.

In most of our experiments we use the following 30-processor platform configuration (variants are mentioned explicitly when needed). Each processor is equipped with a network card of bandwidth between 50 and 180 MB/s and can compute with speed between 50 and 180 MFlop/s. Processors are interconnected via heterogeneous communication links, whose bandwidths are between 60 and 100 MB/s. The 10 different types of objects are randomly distributed over the processors. Recall that computation 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.

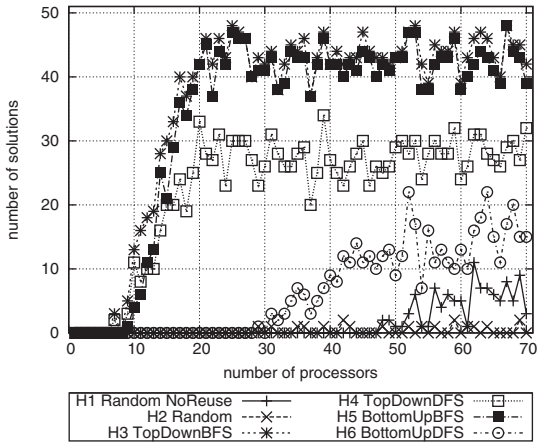
For a given problem instance, we compute the cost achieved by each heuristic. For the Proc-Power problem, this cost is simply the sum of the compute speeds of all used processors. For each heuristic, still for a given problem instance, we compute its achieved cost relative to that achieved by the best-performing heuristic for this instance. We then compute the average relative cost over all problem instances. We use this value to quantify the relative performance of the heuristic. More precisely, the relative performance of heuristic h over a set of R problem instances is $\frac{1}{R}\sum_{r=1}^{R} a_h(r)$, where $a_h(r) = 0$ if heuristic h fails on instance r, and $a_h(r) = \frac{cost_{best}(r)}{cost_h(r)}$ otherwise, where $cost_{best}(r)$ is the best cost over all heuristics for run r and $cost_h(r)$ is the cost of 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].

5.2. Experiment 1: number of processors

In this experiment, we vary the number of processors from 1 to 70. Fig. 3(a) shows the number of successes of the different heuristics using strategy S3. Between 1 and 20 processors, the number of successes 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 begins to find solutions when more than 24 processors are available. H2 already finds solutions when only 14 processors are available, but for the runs with more than 30 processors, it finds fewer solutions than H6. H1 finally finds solutions when at least 40 processors are available. In summary, H3 finds the most solutions, closely followed by H4 and H5. Results for other



(a) Successful runs, strategy S3



(b) Successful runs, strategy S4

Fig. 3. Experiment 1: increasing number of processors, successful runs.

processor selection strategies, not included here, show that the success rate of H3, H4 and H5 is best for strategy S2, followed by S3, S1, and S4. For H1, H2 and H6, strategy S3 achieves the best success rate. S1, S2, and S3 behave similarly in this experiment, whereas strategy S4 has a completely different behavior as can be seen in Fig. 3(b). H4 obtains remarkably fewer solutions than H3 and H5, while it still outperforms the random heuristics and H6. This notable success rate of S4 comes from the uncertainty during the mapping process. In the tree traversals (top down or bottom up) either parent node or child nodes are already mapped and we have to estimate the bandwidth amount needed for the communication with unmapped family members. As S4 takes network cards as reference for mapping choices in non-blocking mode, it accumulates estimation errors which finally may lead to infeasible solutions. S2, which also makes its choices according to network card capacities, is not affected by this problem because it uses the blocking mode.

All our heuristics except H1 attempt to reuse results from common subtrees. To measure the impact of this reuse, we also ran experiments in which this reuse mechanism is disabled. We run our heuristics on the same applications but without trying to reuse intermediate results. Fig. 4 shows the results in terms of the number of successful runs of our heuristics. Comparing with Fig. 3(a), we see that the results are very poor. Both TopDown heuristics (H3 and H4) outperform their competitors, but fail to find results when fewer than 35 processors are available. This observation holds regardless of the processor selection strategy in use.

While the choice of the processor selection strategy does not have a significant impact on success rate (apart from S4, as stated earlier), it does impact performance. Fig. 5(a) shows the relative performance of all heuristics using strategy S3, and Fig. 5(b) shows the same results but using strategy S1. We see that the performances of the heuristics differ significantly depending on the processor selection strategy. Using strategy S3 (and, as it turns out, also strategy S2), H4 performs better than H3, which performs better than H5. However, H5 outperforms both TopDown heuristics when strategy S1 is used. Using strategy S4, the ranking differs again: H3 is better than H5 which is better than H4. The results of S2 and S4 are not included here, but they can be consulted on the web [1]. The performance of H6 and of the random heuristics are similar, and poor,

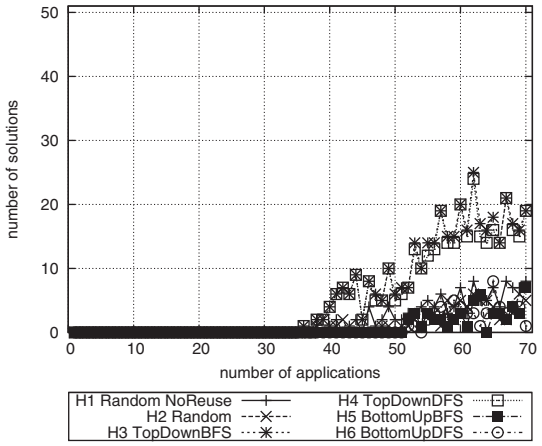
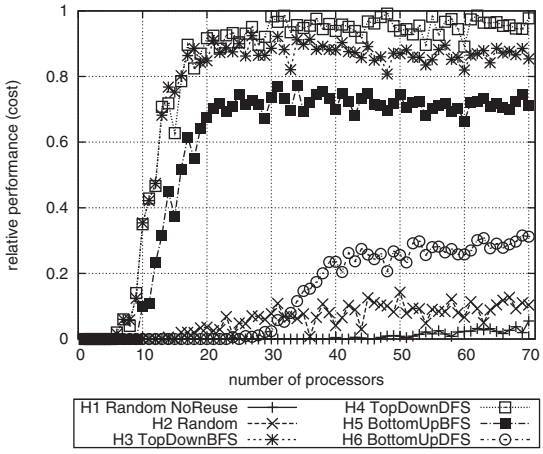
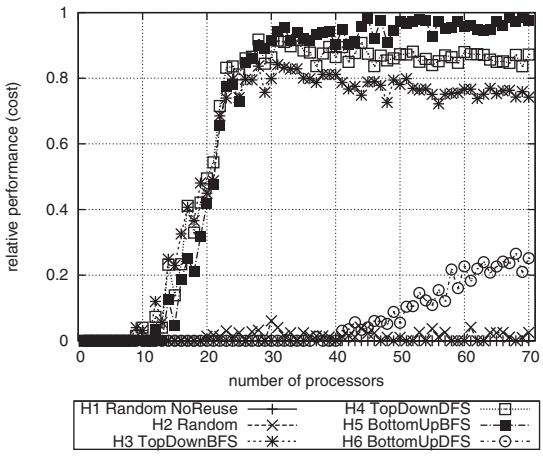


Fig. 4. Experiment 1: increasing number of processors, successful runs without reuse, strategy S3.



(a) Relative performance, strategy S3



(b) Relative performance, strategy S1

Fig. 5. Experiment 1: increasing number of processors, relative performance.

regardless of the processor selection strategy in use. The superiority of the TopDown heuristics in combination with S2 and S3 can be explained due to the neighborhood behavior of TopDown versus BottomUp. TopDown favors a better grouping of parent and child nodes which in the end leads to less expensive solutions since processor capacities are better exploited.

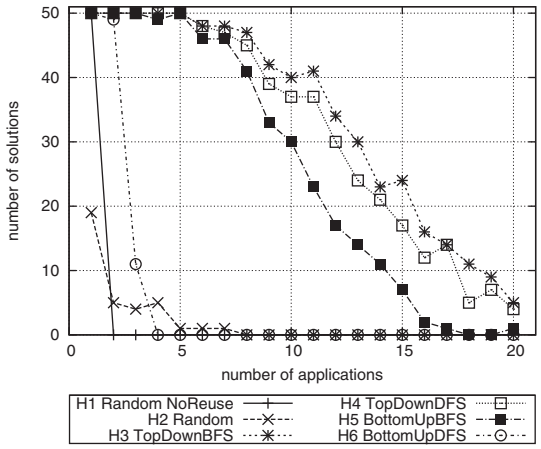
5.3. Experiment 2: number of applications

In this second set of experiments, we vary the number of applications, $\mathcal{K}$. The number of successful runs for strategies S2 and S3 and for all our heuristics are shown in Fig. 6(a) and (b). The main observation is that strategy S3 outperforms strategy S2 in almost all cases. Other results, not included here, show that strategy S1 (resp. S4) leads to similar performance results as strategy S2 (resp. S3), but with lower success rate. Regardless of the processor strategy used among S2 to S3, H3 always finds the most solutions, followed by H4 and by H5. Strategy S4 is an exception: H4 is not as effective as H3 and H5. The bottom up approach better tolerates the variations of the bandwidth estimations used in the heuristics than the top down traversals.

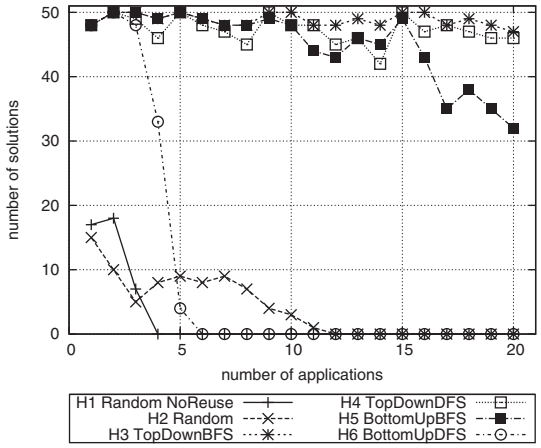
Fig. 7(a) and (b) shows the relative performance of the heuristics for processor selection strategies S1 and S3. Here again, regardless of the processor selection strategy used, both TopDown heuristics show a better relative performance than H5, with the only exception being the case in which strategy S1 is used with fewer than 6 applications (left hand-side of Fig. 7(a)). H6 only finds solutions for up to 6 applications. It is even outperformed by H2, as H2 finds solutions for up to 11 applications (see Fig. 6(b)). Using strategy S4 (see Fig. 7(c)), H5 has a better relative performance than H4, which reflects the success rate of S4. H6 and both random heuristics perform poorly.

Although the results in Fig. 6(b) show that, using the best processor selection strategy S3, H5 has a success rate close to that of the TopDown heuristics, it turns out in Fig. 7(b) that it leads to solutions of lower quality.

We conclude that the best approach is to use strategy S3 in combination with one of the TopDown heuristics. S3 is a non-blocking strategy which tries to fully exploit fast processors. TopDown performs the best local grouping of nodes, which reduces communication costs. The combination of the two leads to the best results in our experiments. Furthermore, our results suggest that H3 should be used for more than 10 applications and H4 for fewer than 10 applications.

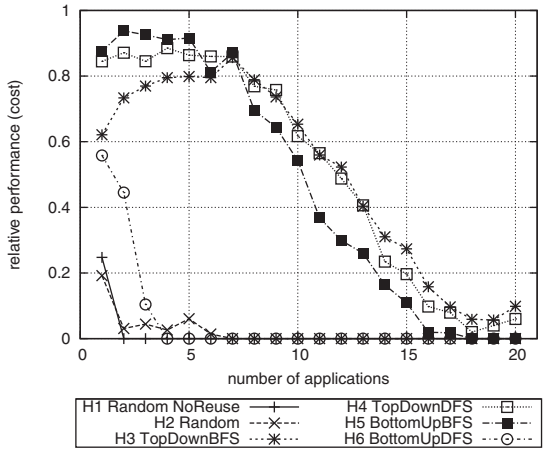


(a) Successful runs, strategy S1

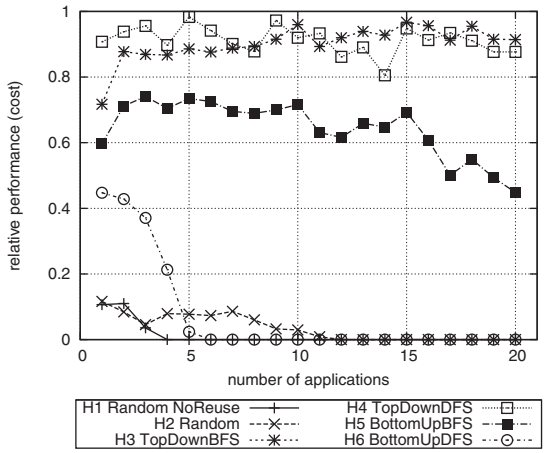


(b) Successful runs, strategy S3

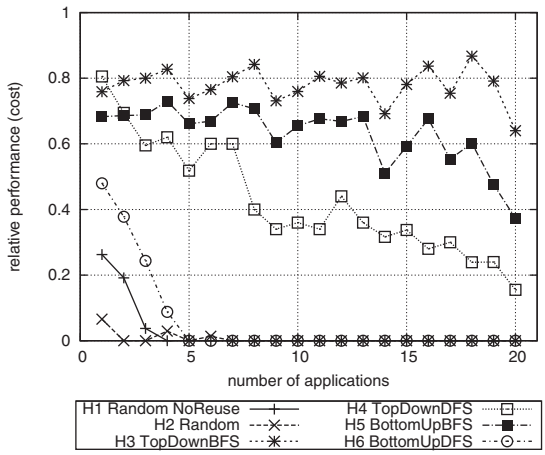
Fig. 6. Experiment 2: increasing number of applications, successful runs.



(a) Relative performance, strategy S1



(b) Relative performance, strategy S3



(c) Relative performance, strategy S4

Fig. 7. Experiment 2: increasing number of applications, relative performance.

5.4. Experiment 3: application size

Increasing application size has roughly the same effect as increasing the number of applications, and results are therefore similar to those discussed in the previous section. Results for the number of successful runs, not included here, show that

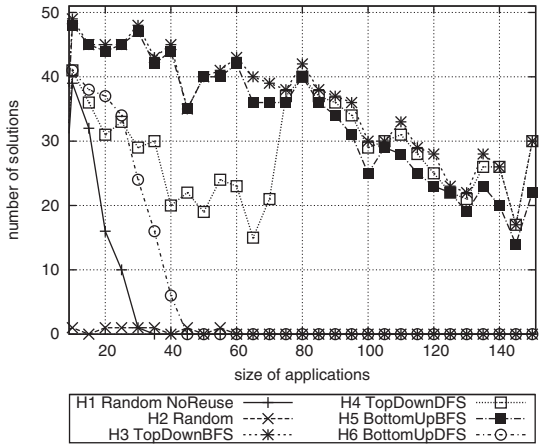
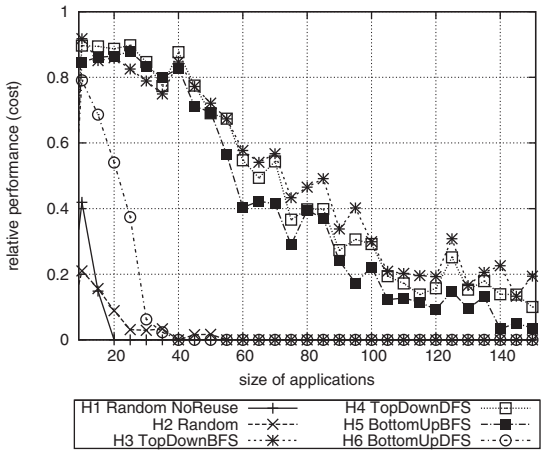
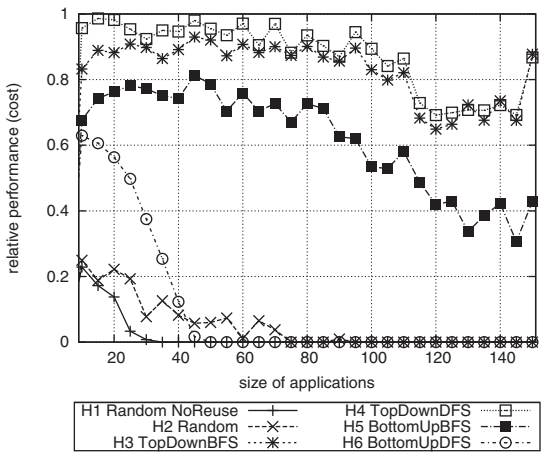


Fig. 8. Experiment 3: increasing application sizes, successful runs, strategy S4.



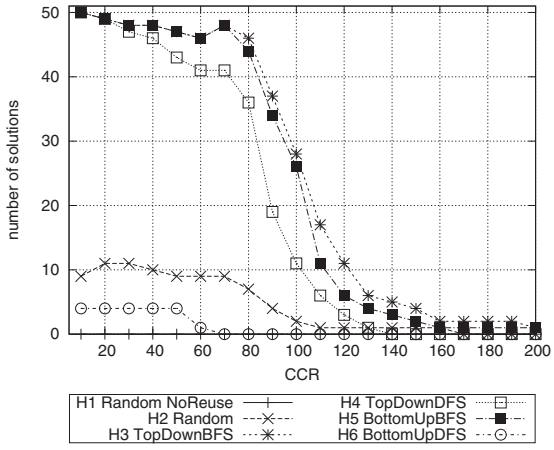
(a) Relative performance, strategy S1



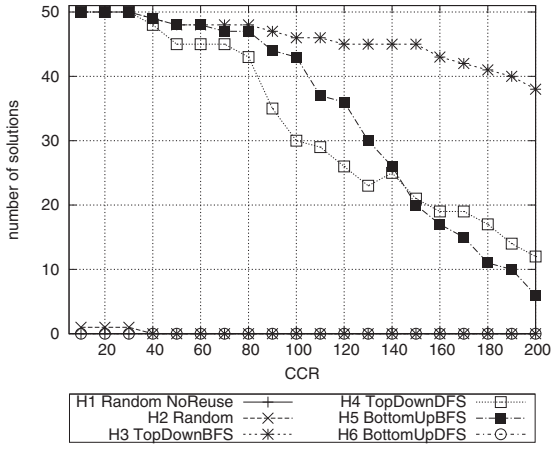
(b) Relative performance, strategy S3

Fig. 9. Experiment 3: increasing application sizes, relative performance.

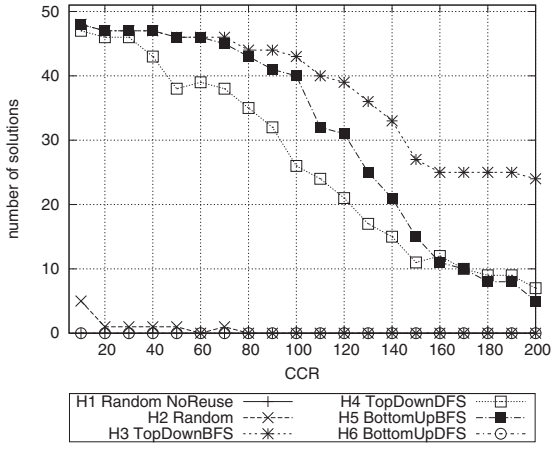
processor selection strategy S3 (closely followed by S4) is the most robust. Using S3, the three best heuristics, still in terms of success rate, are the two TopDown heuristics and H5, and their ranking does not depend on the processor strategy in use: H3 finds more solutions than H4, which, in turn, finds more solutions than H5. The other heuristics, like in Experiment 2, find



(a) Successful runs, strategy S3



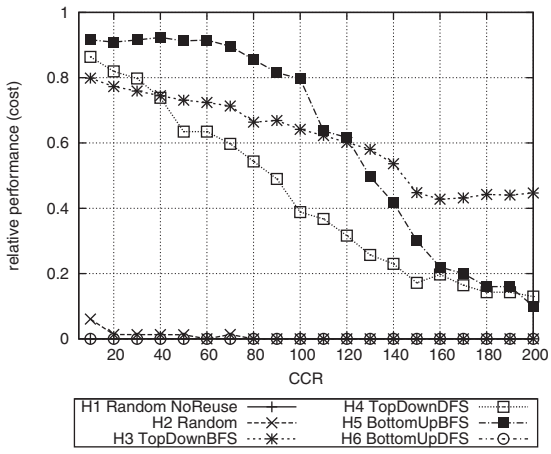
(b) Successful runs, strategy S2



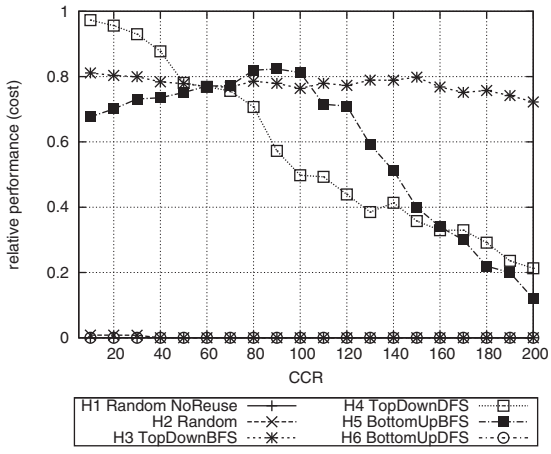
(c) Successful runs, strategy S1

Fig. 10. Experiment 4: increasing communication-to-computation ratio (CCR), successful runs.

significantly fewer solutions. Heuristics that do not reuse results from common sub-trees no longer find results when application sizes exceed 40 operators. Fig. 8 shows the success rates with strategy S4. Interestingly, H4 is ranked third after H3 and H5 for application sizes up to 75 nodes, and switches to the second place behind the other top down heuristic for larger applications. Big applications fare better with this processor selection strategy. This might also be the reason for the poorer



(a) Relative performance, strategy S1



(b) Relative performance, strategy S2

Fig. 11. Experiment 4: increasing communication-to-computation ratio (CCR), relative performance.

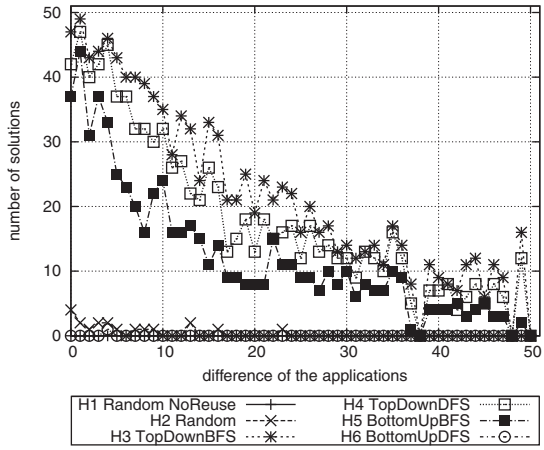
results of H4 in combination with S4 in the precedent experiments, as in those experiments the application sizes were fixed to 50 nodes.

Fig. 9(a) and (b) shows the relative performance of the heuristics for the S1 and S3 processor selection strategy. The results are similar to those shown in Fig. 7(a) and (b) with an increasing number of applications. Also, S2 (resp. S4) achieves similar but slightly poorer results than S1 (resp. S3). When using strategy S1, both TopDown heuristics and H5 achieve comparable results. But, when using the better S3 strategy, the two TopDown heuristics outperform H5, with a slight advantage for H3. We conclude that using H3 combined with the S3 processor selection strategy is the best approach.

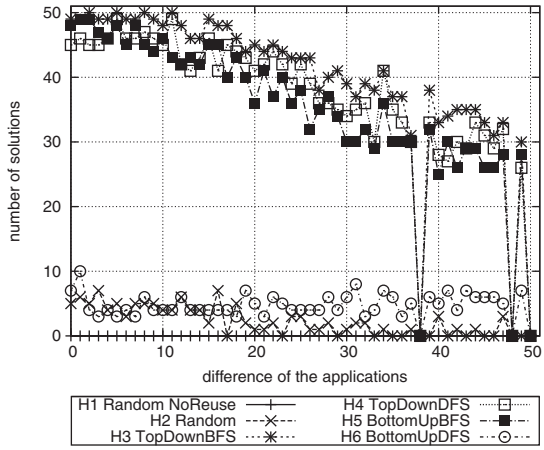
5.5. Experiment 4: communication-to-computation ratio (CCR)

We define the CCR as the ratio between the mean amount of communication and the mean amount of computation, where the communication corresponds to the sum of the output sizes of operators δ_i and the computation corresponds to the sum of the computational costs w_i of the operators. Increasing the CCR allows us to study the sensitivity of our heuristics to communication-intensive scenarios.

Fig. 10(a) shows success rate versus CCR for all our heuristics using processor selection strategy S3. Results are similar for the S4 strategy. These two non-blocking strategies are very sensitive to the CCR. As the CCR increases, heuristics H3, H4, and H5 have an excellent success rate for CCR $\leqslant 60$, but the success rate decreases drastically until no solution is found at all for a CCR of 180. Contrasting these results with those obtained for strategy S2, shown in Fig. 10(b), we see that H3 still finds 38 solutions out of 50 instances even for CCR = 200. These three heuristics have much higher success rates than H6, H2, and H1. Out of these, using strategy S2, only H2 finds some solutions. In this experiment we observe the importance of the blocking processor selection. The non-blocking strategies are prone to network card congestion as processors are assigned with operators that are not necessarily neighbors in the application trees. This can lead to a higher amount of communication, compared to the case in which only neighbor nodes are mapped onto the same processor. When communication is negligible



(a) Successful runs, strategy S2



(b) Successful runs, strategy S3

Fig. 12. Experiment 5: increasing dissimilarity of applications, successful runs.

compared to computation, this does not create any problems and therefore the processor selection strategies S3 and S4 behave better.

Fig. 11(a) and (b) shows relative performance results using the S1 and S2 processor selection strategy, which both achieve a reasonable success rate even for high CCR. We see that in both cases the best heuristics are the two TopDown heuristics and H5. However, the ranking of these three heuristics is different depending on the processor scheduling strategy in use.

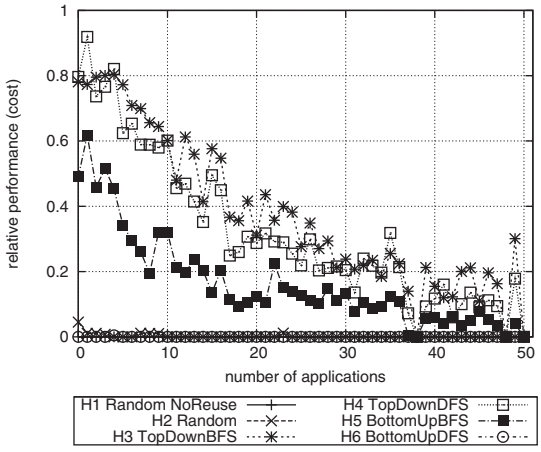
If we compare Fig. 10(b) and (c), we note that H5 finds fewer solutions using strategy S1 than S2, but its relative performance using strategy S1 and a CCR smaller than 60 is better than when using strategy S2, as seen in Fig. 11(a) and (b). Furthermore, H3 using strategy S1 always finds the most solutions 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.

Overall, regardless of which strategy is used, H3 always finds at least as many solutions as the other heuristics using the same strategy, but its relative performance outperforms the other only when the CCR is high. For small CCR values (up to 60), we recommend to use H4 with strategy S2 or S3 to achieve good relative performance.

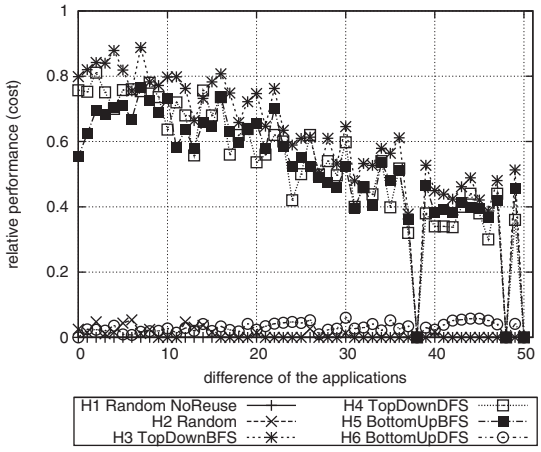
5.6. Experiment 5: similarity of applications

In this experiment we 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 similar or dissimilar. For this purpose we create applications with an increasing number of non-shared operators. Results are shown in Fig. 12(a) and (b), for processor selection strategies S2 and S3, with the *x*-axis showing the number of operators that differ between the two applications.

Taking into account the results of the other processor selection strategies, not included here, we observe the following ranking of the different strategies: strategy S3 leads to the best results, followed by strategy S4. Finally, strategies S1 and



(a) Relative performance, strategy S2



(b) Relative performance, strategy S3

Fig. 13. Experiment 5: increasing dissimilarity of applications, relative performance.

S2, which are more sensitive to application differences, lead to similar results, which are not as good as those obtained with strategy S3.

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. Finally, H1 always return poor results. H3 has the best relative performance using the blocking strategies (see Fig. 13(a)), whereas with non-blocking strategies H4 achieves the best results (see Fig. 13(b)), which is important as its success rate is slightly poorer. H5 always ranks third.

5.7. Summary of experiments

Our results show that a random node mapping approach for multiple applications is not feasible. More generally, failing to reuse results from common sub-trees dramatically limits the success rate and also the quality of the operator mapping. The TopDown approaches turn out to be the best, and in most cases BFS traversal achieves the best result. The BottomUp approaches are competitive only when using a BFS traversal. The DFS traversal seems unable to reuse results from common sub-trees efficiently (it often finds itself with no bandwidth left to perform the necessary communications). Furthermore, we see a strong impact of the processor selection strategy on the quality of the solution. The blocking strategies are generally not as effective as the non-blocking strategies, with the exception of scenarios when the CCR is large. Overall, H3 (TopDown, BFS traversal) and strategy S3 (select the fastest processor, non-blocking) proves to be a solid combination.

6. Conclusion

We have studied the problem of mapping multiple concurrent in-network stream-processing applications onto a collection of heterogeneous processors. These stream-processing applications are structured as trees of operators, in which some

operators have to download basic objects continuously at different sites of the network, and at the same time they have to process the corresponding data in order to produce results. We have considered the problem under a non-constructive scenario, in which a fixed set of computation and network resources is available and the goal is to use as few resources as possible. We have identified four relevant operator mapping problems, which are all 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 results show that judicious node reuse across applications is paramount: it increases the likelihood of successfully finding a valid mapping and leads to significantly better mappings. Our results also show 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) achieves good results across the board.

Due to the lack of accepted model for in-network stream-processing applications, in our experiments we have instantiated application and platform configurations using a broad range of parameters. Our results should thus hold in most practical settings for a range of applications and platform. Nevertheless, in future work it would be interesting to select one or more actual stream-processing applications, benchmark the applications to derive their structures and the parameters that define their operators, deploy them on a dedicated testbed, and evaluate and compare our proposed heuristics in this practical setting.

In the longer term, a clear extension of this work would be to develop heuristics for the other optimization problems defined in Section 2.4. We could also envision a more general computational function that depends both on the operator and the processor ($w_{i,u}$ would denote the time required to compute operator i on processor u). This would make it possible to express affinities between certain operators and certain processors, mandating the development of more complex heuristics. Also, we believe that it would be interesting to add a storage cost for objects downloaded onto processors, which could lead to new objective functions. Finally, one could address more complicated scenarios with many, likely conflicting, relevant criteria to consider simultaneously, some related to performance (throughput, response time), some related to fault-tolerance (replicating some computations to increase reliability), and some related to environmental costs (resource costs, energy consumption).

Acknowledgments

The authors thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper. A. Benoit and Y. Robert are with the Institut Universitaire de France. This work was supported in part by the ANR *StochaGrid* project.

References

[1] Diagrams of all experiments. Available from: <http://graal.ens-lyon.fr/vsonigo/code/query-multiapp/diagrams/>.
[2] Source code for the heuristics. Available from: <http://graal.ens-lyon.fr/vsonigo/code/query-multiapp/>.
[3] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The Design of the borealis stream processing Engine, in: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA, January 2005.
[4] S. Babu, J. Widom, Continuous queries over data streams, SIGMOD Record 30 (3) (2001).
[5] B. Badcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the International Conference on Very Large Data Bases, pp. 456–467, 2004.
[6] A. Benoit, H. Casanova, V. Rehn-Sonigo, 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] P. Bonnet, J. Gehrke, P. Seshadri, Towards sensor database systems, in: Proceedings of the Conference on Mobile Data Management, 2001.
[8] J. Chen, D. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for internet databases, in: Proceedings of the SIGMOD International Conference on Management of Data, pp. 379–390, 2000.
[9] J. Chen, D.J. DeWitt, J.F. Naughton, Design and evaluation of alternative selection placement strategies in optimizing continuous queries, in: Proceedings of ICDE, 2002.
[10] L. Chen, K. Reddy, G. Agrawal, GATES: a grid-based middleware for processing distributed data streams, High performance distributed computing, in: Proceedings. 13th IEEE International Symposium on, pp. 192–201, 4–6 June, 2004.
[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, S. Zdonik, Scalable distributed stream processing, in: Proc. of the CIDR Conference, January 2003.
[12] E. Cooke, R. Mortier, A. Donnelly, P. Barham, R. Isaacs, Reclaiming network-wide visibility using ubiquitous end system monitors, in: Proceedings of the USENIX Annual Technical Conference, 2006.
[13] Ilog cplex: high-performance software for mathematical programming and optimization. Available from: <http://www.ilog.com/products/cplex/>.
[14] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, O. Spatscheck, Gigascope: high-performance network monitoring with an SQL interface, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 623–633, 2002.
[15] M.R. Garey, D.S. Johnson, Computers and Intractability, Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.
[16] B. Hong, 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.
[17] R. Huebsch, J.M. Hellerstein, N.L. Boon, T. Loo, S. Shenker, I. Stoica, Querying the Internet with PIER, September 2003.
[18] Y.E. Ioannidis, Query optimization, ACM Computing Surveys 28 (1) (1996) 121–123.
[19] J. Kräme, B. Seeger, A temporal foundation for continuous queries over data streams, in: Proceedings of the International Conference on Management of Data, pages 70–82, 2005.
[20] L. Liu, C. Pu, W. Tang, Continual queries for internet scale event-driven information delivery, IEEE Transactions on Knowledge and Data Engineering 11 (4) (1999) 610–628.
[21] D. Logothetis K. Yocum, Wide-scale data stream management, in: Proceedings of the USENIX Annual Technical Conference, 2008.

[22] S. Madden, M. Franklin, J. Hellerstein, W. Hong, The design of an acquisitional query processor for sensor networks, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 491–502, 2003.
[23] S. Nath, A. Deshpande, Y. Ke, P.B. Gibbons, B. Karp, S. Seshan, IrisNet: An Architecture for Internet-scale Sensing Services.
[24] V. Pandit, H. Ji, Efficient in-network evaluation of multiple queries, in: HiPC, pp. 205–216, 2006.
[25] P. Pietzuch, J. Leflie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), pp. 49–60, 2006.
[26] B. Plale, K. Schwan, Dynamic querying of streaming data with the dQUOB system, IEEE Transactions on Parallel and Distributed Systems 14 (4) (2003) 422–432.
[27] V. Rehn-Sonigo, Multi-criteria Mapping and Scheduling of Workflow Applications onto Heterogeneous Platforms, PhD thesis, École Normale Supérieure de Lyon and Universität Passau, 2009.
[28] U. Srivastava, K. Munagala, J. Widom, Operator placement for in-network stream query processing, in: Proceedings of the 24th ACM International Conference on Principles of Database Systems, pp. 250–258, 2005.
[29] R. van Rennesse, K. Birman, D. Dumitriu, W. Vogels, Scalable management and data mining using astrolabe, in: Proceedings from the First International Workshop on Peer-to-Peer Systems, pp. 280–294, 2002.