Resource Allocation for Multiple Concurrent In-Network Stream-Processing Applications

Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, Yves Robert

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

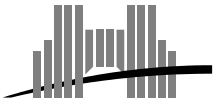
Resource Allocation for Multiple Concurrent In-Network Stream-Processing Applications

Anne Benoit ,
Henri Casanova ,
Veronika Rehn-Sonigo ,
Yves Robert

February 2009

Research Report N° 2009-07

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Resource Allocation for Multiple Concurrent In-Network Stream-Processing Applications

Anne Benoit , Henri Casanova , Veronika Rehn-Sonigo , Yves Robert

February 2009

Abstract

This paper investigates the operator mapping problem for in-network stream-processing applications. In-network stream-processing amounts to applying one or more trees of operators in steady-state, to multiple data objects that are continuously updated at different locations in the network. The goal is to compute some final data at some desired rate. Different operator trees may share common subtrees. Therefore, it may be possible to reuse some intermediate results in different application trees.
The first contribution of this work is to provide complexity results for different instances of the basic problem, as well as integer linear program formulations of various problem instances. The second second contribution is the design of several polynomial-time heuristics. One of the primary objectives of these heuristics is to reuse intermediate results shared by multiple applications. Our quantitative comparisons of these heuristics in simulation demonstrates the importance of choosing appropriate processors for operator mapping. It also allow us to identify a heuristic that achieves good results in practice.

Keywords: in-network stream-processing, trees of operators, multiple concurrent applications, operator mapping, polynomial heuristics.

Résumé

Dans ce rapport, on s'intéresse à des applications qui traitent des flux de données de manière pipelinée. Chaque application consiste en un arbre d'opérateurs qu'on applique sur les données successives. En régime permanent, les opérateurs interrogent des bases de données qui sont mises à jour périodiquement. L'objectif est de calculer le résultat final à un débit fixé. Des arbres d'opérateurs distincts peuvent partager des résultats. De ce fait, il peut être possible de réutiliser quelques résultats intermédiaires pour traiter différentes applications.
La première contribution de ce travail est l'obtention de résultats de complexité pour les différentes instances du problème, ainsi que la formulation de ces instances en terme de programme linéaire. La deuxième contribution est le développement de plusieurs heuristiques polynomiales. La réutilisation des résultats intermédiaires partagés par plusieurs applications est un objectif premier de ces heuristiques. Nos comparaisons par simulation des heuristiques démontrent toute l'importance du choix des processeurs pour le placement des opérateurs. De même elles nous permettent d'identifier une heuristique performante, qui obtient de bons résultats dans la pratique.

Mots-clés: traitement de flux en réseau, arbres d'opérateurs, multiples applications concurrentes, placement d'opérateurs, heuristiques polynomiales.

1 Introduction

We consider the execution of applications structured as trees of operators, where the leaves of the tree correspond to basic data objects that are distributed over servers in a distributed 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. This problem is called *stream processing* [5] and arises in several domains.

An important domain of application is the acquisition and refinement of data from a set of sensors [27, 22, 8]. For instance, [27] outlines a video surveillance application in which the sensors are cameras located at different locations over a geographical area. The goal of the application could be to identify monitored areas in which there is significant motion between frames, particular lighting conditions, and correlations between the monitored areas. This can be achieved by applying several operators (e.g., filters, pattern recognition) to the raw images, which are produced/updated periodically. Another example arises in the area of network monitoring [14, 28, 13]. 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). A continuous query is applied continuously, i.e., at a reasonably fast rate, and returns results based on recent data generated by the data streams. Many authors have studied the execution of continuous queries on data streams [4, 20, 10, 26, 19].

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. Although a simple solution is to send all basic objects to a central compute server, it often proves unscalable due to network bottlenecks. Also, this central server may not be able to meet the desired target rate for producing results due to the sheer amount of computation involved. The alternative is then 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, 12, 17, 11, 23, 28, 9, 21]. These systems all face the same question: where should operators be mapped in the network?

In this paper we address the operator-mapping problem for *multiple concurrent in-network stream-processing applications*. The 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 [7] 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 and the objective is to meet this requirement while using as few resources as possible. In this paper we extend the work in [7] 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 while meeting QoS requirements. Instead, in [7], we studied a "constructive" scenario in which resources could be purchased and the objective was to spend

as little money as possible. Second, and more importantly, while in [7] we studied the case of a single application, in this paper we focus on multiple concurrent applications that contend for the servers. Each application has its own QoS requirement and the goal is to meet them all. In this case, a clear opportunity for higher performance with a reduced resource consumption is to reuse common sub-expression between operator trees when applications share basic objects [24]. We restrict our study to trees of operators that are general binary trees and discuss relevant special cases (e.g., left-deep trees [18]). We consider target platforms that are either fully homogeneous, or with a homogeneous network but heterogeneous servers, or fully heterogeneous. Our specific contributions are twofold: (i) we formalize operator mapping problems for multiple in-network stream-processing applications and give their complexity; (ii) we propose a number of algorithms to solve the problems and evaluate them via extensive simulation experiments.

The rest of this paper is organized as follows. In Section 2 we define our application and platform models, and we formalize a number of operator mapping problems. In Section 3 we discuss the computational complexity of our mapping problems and in Section **??** we give integer linear programming formulations. In Section 5 we propose several heuristics, which we evaluate in Section 6. Finally we conclude in Section 7 with a summary of our results and future directions.

2 Framework

2.1 Application Model

We consider $\mathcal{K}$ applications, each needing to perform several operations organized as a binary tree (see Figure 1). Operators are taken from the set $\mathcal{OP} = \{op_1, op_2, \ldots\}$, and operations are initially performed on basic objects from 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 require to download basic objects periodically.

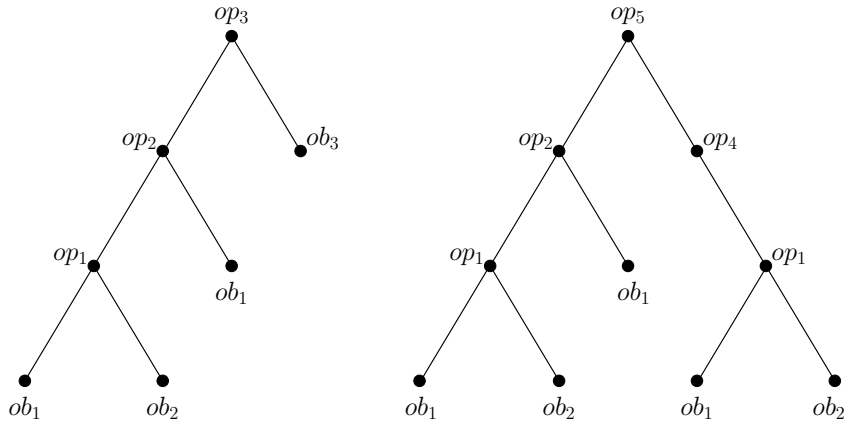


Figure 1: Sample applications structured as binary trees of operators.

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. We have the constraint that $|objects(p)| + |operators(p)| \leq 2$ since application trees are binary. An application is fully defined by the operator at the root of its tree. For instance, if we consider Fig. 1, we have one application rooted on op_3, and another application rooted on op_5. Operator op_1 needs to download objects ob_1 and ob_2, while operator op_2 downloads only object ob_1 but also 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*, $\rho^{(k)}$, and the specification of the target throughput is 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 the target application throughput $\rho^{(k)}$. Conceptually, operator op_p executes two concurrent threads in steady-state:

• It periodically downloads the most recent copies of the basic objects in $objects(p)$, if any. Note that these downloads may simply amount to constant streaming of data from sources that generate data streams. Each download has a prescribed cost in terms of bandwidth based on application QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). A basic object ob_j has a size d_j (in bytes) and needs to be downloaded by the processors that use it for application k with frequency $f_j^{(k)}$. Therefore, these basic object downloads consume an amount of bandwidth equal to $rate_j^{(k)} = d_j \times f_j^{(k)}$ on each network link and network card through which this object is communicated for application k. Note that 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 $rate_j = \max_k\{rate_j^{(k)}\}$.

• It receives intermediate results computed by $operators(p)$, if any, and it performs some 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 which will be sent to another operator, or the final result of the application (root operator). 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 target distributed network is a fully connected graph (i.e., a clique) interconnecting a set of processors $\mathcal{P}$. These processors can be assigned operators of the application tree and perform some computation. Some processors also hold and update basic objects. Each 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. In addition, each processor $P_u \in \mathcal{P}$ is characterized by a compute speed s_u. Resources operate under the full-overlap, bounded multi-port model [16]. In this model, a processor P_u can be involved in comput-

ing, sending data, and receiving data simultaneously. The "multi-port" assumption states that each processor can send/receive data simultaneously on multiple network links. The "bounded" assumption states that the total transfer rate of data sent/received by processor P_u is bounded by its network card bandwidth B_u. The case in which some dedicated processors are only providing basic objects but cannot be used for computations is obtained simply by setting their compute speed to 0.

2.3 Mapping Model and Constraints

Our 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 processor's network card. Each used processor is in charge of one or several nodes. If there is only one node on processor P_u, while the processor computes for the t-th final result it sends to its parent (if any) the data corresponding to intermediate results for the $(t-1)$-th final result. It also receives data from its children (if any) for computing the $(t+1)$-th final result. All three activities are concurrent (see Section 2.2). Note however that different nodes can be assigned to the same processor. In this case, the same overlap happens, but possibly on different result instances (an operator may be applied for computing the t_1-th result while another is being applied for computing the t_2-th). A particular case is when several nodes with the same operator are assigned to the same processor. In this case, computation is done only once for this operator, but it should occur at the highest required rate among those of the corresponding applications.

A basic object can be duplicated, and thus available and updated at multiple processors. We assume that duplication of basic objects is achieved in some out-of-band manner specific to the target application (e.g., due to the use of a distributed database infrastructure that allows consistent data replication). In this case, a processor can choose among multiple data sources when downloading 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 continuously download that 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 $\rho^{(k)}$.

The formal definition of $Ch(u)$ and $Par(u)$ is as follows. We first define two sets of tuples, $ACh(u)$ and $APar(u)$, used to account for communications for the same data but for different

applications:

$$ACh(u) = \left\{(p,v,k) \mid \exists i, p' \quad p \in a_{op}(v); p' \in a_{op}(u); p \in operators(p'); op_p = op(n_i^{(k)}); op_{p'} = op(n_{\lfloor i/2 \rfloor}^{(k)}\right\}$$

$$APar(u) = \left\{(p,v,k) \mid \exists i, p' \quad p \in a_{op}(v); p' \in a_{op}(u); p \in operators(p'); op_p = op(n_i^{(k)}); op_{p'} = op(n_{\lfloor i/2 \rfloor}^{(k)}\right\}$$

Then we determine which application has the higher throughput for redundant entries, where arg max randomly chooses one application if there are equalities:

$$kchosen(p,v,X) = \arg\max_{k \in \mathcal{K}} \left\{ \rho^{(k)} \mid \exists (p,v,k) \in X \right\}$$

Finally, $X(u) = \{(p,v,kchosen(p,v,AX)) \mid op_p \in \mathcal{OP}, P_v \in \mathcal{P}\}$. X stands for Ch or Par, and we have thus thus fully defined $Ch(u)$ and $Par(u)$.

Given these notations, we can now express constraints for the application throughput: each processor must compute and communicate fast enough to respect the prescribed throughput of each application which is being processed by it. The computation constraint is expressed below. 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 a child or the parent of a given node and this node are mapped on different processors. In other terms, we neglect intra-processor communications. 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. This throughput, as well as the processors with which P_u needs to communicate, are obtained via $Ch(u)$ and $Par(u)$. In these expressions $v \neq u$ since we neglect intra-processor-communications.

The first part of Eq. 2 expresses constraints on receiving, while the second part refers to 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.

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

Finally, we need to express the fact that 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:

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

2.4 Optimization Problems

The overall objective of the operator-mapping problem is to ensure that a prescribed throughput per application is achieved while minimizing a cost function. Several relevant problems can be envisioned. Proc-Nb minimizes the number of processors enrolled for computations (processors that are allocated at least one node); Proc-Power minimizes the compute capacity and/or the network card capacity of processors enrolled for computations (e.g., a linear function of both criteria); BW-Sum minimizes the sum of the bandwidth capacities used by the application; and finally BW-Max minimizes the maximum percentage of bandwidth used on all links (minimizing the impact of the applications on the network for other users).

Different platform types may be considered depending on the heterogeneity of the resources. We consider the case in which the platform is fully homogeneous ($s_u = s$, $B_u = B$ and $b_{u,v} = b$), which we term Hom. The heterogeneous case in which network links can have various bandwidths is termed Het.

Each combination of problems and platforms could be envisioned, but we will see that Proc-Power on a Hom platform is actually equivalent to Proc-Nb. Proc-Nb makes more sense in this setting, while Proc-Power is used for Het platforms only. 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. This is true even for a simple case: a Hom platform and a single application ($|\mathcal{K}| = 1$), that is structured as a left-deep tree, 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 a technical report for the proof [6], which relies on a straightforward reduction to 3-Partition, which is known to be NP-hard in the strong sense [15]. 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 problem as 2-Partition, which is known to be NP-hard [15]. Here is a sketch of the straightforward proof, which holds even in the case of 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 problem is then to partition the set of operators in two subsets so that the bandwidth consumption on the two network links in use is as equal as possible. This is exactly the 2-Partition problem.

The BW-Sum problem is NP-hard because it can be reduced to the Knapsack problem, which is NP-hard [15]. 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 these problems can be solved thanks to an integer linear program (see Section 4). However, they cannot be solved in polynomial time (unless P=NP). Therefore, in the Section 5 we describe polynomial-time heuristics for one of these problems.

4 Linear Programming Formulation

In this section, we give an integer linear program (ILP) formulation of the Proc-Power-Het, BW-Sum-Het and BW-Max-Het problems, in terms of an integer linear program (ILP). These are the most general versions of our operator-mapping problems. More restricted versions, e.g., with Hom platforms, can be solved using the same ILPs. We describe the input data to the ILP, its variables, its constraints, and finally its objective functions.

In all that follows, i and i' are indices spanning nodes in set of nodes of an application tree; p and p' are indices spanning operators in $\mathcal{OP}$; j is an index spanning objects in $\mathcal{OB}$; u, u', and v are indices spanning processors in $\mathcal{P}$; k is an application index spanning $\mathcal{K}$.

4.1 Input Data

Parameters $_i, w_i$ for operators, $rate_j^{(k)}$ for object download rates, and $s_u, B_u, b_{u,v}$ for processors and network elements, are rational numbers and defined in Section 2. $\rho^{(k)}$ is a rational number that represents the throughput QoS requirement for application k. For convenience, we also introduce families of boolean parameters: *par*, *oper*, and *object*, that pertain to application trees; and *obj*, that pertain to location of objects on processors. We define these parameters hereafter:

- $par(k, i, i')$ is equal to 1 if internal node $n_i^{(k)}$ is the parent of $n_{i'}^{(k)}$ in the tree of application k, and 0 otherwise.
- $oper(k, i, p)$ is equal to 1 if $op(n_i^{(k)}) = p$, and 0 otherwise.
- $object(k, i, j)$ is equal to 1 if node $n_i^{(k)}$ needs object ob_j (i.e., $p \in objects(op(n_i^{(k)}))$), and 0 otherwise.
- $obj(u, j)$ is equal to 1 if processor P_u owns a copy of object ob_j, and 0 otherwise.

4.2 Variables

- $x_{k,i,u}$ is a variable equal to 1 if node $n_i^{(k)}$ is mapped on P_u, and 0 otherwise.
- $d_{j,u,v,k}$ is a variable equal to 1 if processor P_u downloads object ob_j for application k from processor P_v, and 0 otherwise.
- $y_{k,i,u,i',u'}$ is a variable equal to 1 if $n_i^{(k)}$ is mapped on P_u, $n_{i'}^{(k)}$ is mapped on $P_{u'}$, and $n_i^{(k)}$ is the parent of $n_{i'}^{(k)}$ in the application tree.
- $used_u$ is a variable equal to 1 if there is at least one node mapped to processor P_u, and 0 otherwise.
- $xop_{k,p,u}$ is a variable equal to 1 if op_p of application k is mapped to processor P_u, and 0 otherwise.

- $yop_{k,p,u,p',u'}$ is a variable equal to 1 if op_p of application k is mapped on processor P_u, $op_{p'}$ of application k is mapped on processor $P_{u'}$, and op_p is a parent of $op_{p'}$ in application k, and 0 otherwise.
- $Ch_{u,p,v,k}$ is a variable equal to 1 if $(p, v, k) \in Ch(u)$, and 0 otherwise.
- $Par_{u,p,v,k}$ is a variable equal to 1 if $(p, v, k) \in Par(u)$, and 0 otherwise.
- $rho_{u,p}$ is a rational variable equal to the throughput of op_p if it is mapped on processor P_u, and 0 otherwise.
- $ratemax_{j,u,v}$ is a rational variable equal to the download rate of object ob_j by processor P_u from processor P_v, and 0 otherwise.

4.3 Constraints

We first give constraints to guarantee that the allocation of nodes to processors is valid, and that each required download is done from a server that holds the relevant object.

- $\forall k, i \quad \sum_u x_{k,i,u} = 1$: each node is placed on exactly one processor;
- $\forall j, u, v, k \quad d_{j,u,v,k} \leq obj(v, j)$: object ob_j can be downloaded from processor P_v only if P_v holds it;
- $\forall i, j, u, k \quad 1 \geq \sum_v d_{j,u,v,k} \geq x_{k,i,u}.object(k, i, j)$: processor P_u must download object ob_j from exactly one processor P_v if there is a node $n_i^{(k)}$ mapped on processor P_u that requires ob_j.

The next two constraints aim at properly defining variables y. Note that a straightforward definition would be $y_{k,i,u,i',u'} = par(k, i, i').x_{k,i,u}.x_{k,i',u'}$, but this leads to a non-linear program. Instead we write, for all k, i, u, i', u':

- $y_{k,i,u,i',u'} \leq par(k, i, i')$; $y_{k,i,u,i',u'} \leq x_{k,i,u}$; $y_{k,i,u,i',u'} \leq x_{k,i',u'}$: $y_{k,i,u,i',u'}$ is forced to be 0 if one of these three conditions does not hold.
- $y_{k,i,u,i',u'} \geq par(k, i, j).\left(x_{k,i,u} + x_{k,i',u'} - 1\right)$: $y_{k,i,u,i',u'}$ is forced to be 1 only if the three conditions are true (otherwise the right term is lower than or equal to 0).

The following two constraints ensure that $used_u$ is properly defined:

- $\forall u \; used_u \leq \sum_{k,i} x_{k,i,u}$: processor P_u is not used if no node is mapped to it;
- $\forall k, i, u \; used_u \geq x_{k,i,u}$: processor P_u is used if at least one node n_i is mapped to it.

The following four constraints ensure that $xop_{k,p,u}$ and $yop_{k,p,u}$ are properly defined:

- $\forall i, k, p, u \; xop_{k,p,u} \geq x_{k,i,u}.oper(k, i, p)$: xop is forced to be 1 if operator op_p of application k is mapped on processor P_u;
- $\forall k, p, u \quad xop_{k,p,u} \leq \sum_i x_{k,i,u}.oper(k, i, p)$: xop is forced to be 0 if operator op_p of application k is not mapped on processor P_u;

- $\forall k, p, p', u, u', i, i'$ $yop_{k,p,u,p',u'} \leq xop_{k,p,u}$; $yop_{k,p,u,p',u'} \leq xop_{k,p',u'}$; $yop_{k,p,u,p',u'} \leq par(k,i,i')$; $yop_{k,p,u,p',u'} \leq oper(k,i,p)$; $yop_{k,p,u,p',u'} \leq oper(k,i',p')$: $yop_{k,p,u,p',u'}$ is forced to be 0 if one of these conditions does not hold;
- $\forall k, p, p', u, u', i, i'$ $yop_{k,p,u,p',u'} \geq par(k,i,i').oper(k,i,p).oper(k.i',p').(xop_{k,p,u}+xop_{k,p',u'}-1)$: $yop_{k,p,u,p',u'}$ is forced to be 1 only if all five conditions are true.

The next four constraints ensure that $Ch_{u,p,v,k}$ and $Par_{u,p,v,k}$ are defined properly:

- $\forall u, p, v, k$ $Ch_{u,p,v,k} \leq \sum_{p'} yop_{k,p',u,p,v}$: in application k, if the parent operator of operator op_p, which is mapped on P_v, is not mapped on processor P_u, Ch is forced to be 0;
- $\forall p', u, p, v, k$ $Ch_{u,p,v,k} \geq yop_{k,p',u,p,v}$: in application k, if operator op_p of application k is mapped to P_v and its parent operator in the application tree is mapped to P_u, Ch is forced to be 1.
- $\forall u, p, v, k$ $Par_{u,p,v,k} \leq \sum_{p'} yop_{k,p',v,p,u}$: in application k, if the parent operator of operator op_p, which is mapped to P_u, is not mapped to processor P_v, Par is forced to be 0;
- $\forall p', u, p, v, k$ $Par_{u,p,v,k} \geq yop_{k,p',v,p,u}$: in application k, if operator op_p is mapped to P_u and its parent operator in the application tree is mapped to P_v, Par is forced to be 1.

The following two constraints ensure that the throughput QoS requirement of each application, $\rho^{(k)}$, is met:

- $\forall k, u, p$ $rho_{u,p} \geq xop_{k,p,u}.\rho^{(k)}$: the throughput of processor P_u, to which operator op_p of application k is mapped, has to satisfy the throughput QoS requirement of application k;
- $\forall k, p, u, v$ $ratemax_{p,u,v} \geq d_{p,u,v,k}.rate_p^{(k)}$: the update rate of operator op_p on processor P_u has to satisfy the throughput QoS requirement of application k;

The following constraint ensures that the compute capacity of each processor is not exceeded while meeting QoS throughput requirements:

- $\forall u$ $\sum_p rho_{u,p} \frac{w_p}{s_u} \leq 1$.

The following two constraints ensure that the bandwidth capacity of network elements are not exceeded:

- Bandwidth constraint for the processor network cards:

$$\forall u \quad \sum_{p,v,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,v,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{j,v,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,v,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq B_u \tag{5}$$

- Bandwidth constraints for links between processors:

$$\forall u, v \quad \sum_{p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq b_{u,v} \tag{6}$$

4.4 Objective Function

We have to define the objective function to optimize. We have a different definition for each problem:

PROC-POWER-HET:

$$\min \left(\sum_{u,p} rho_{u,p} \frac{w_p}{s_u} \right) . \tag{7}$$

BW-SUM-HET:

$$\begin{aligned} \min \sum_{u,v,p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{u,v,p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ \sum_{u,v,j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{u,v,j,k} d_{j,v,u,k}.ratemax_{j,v,u} . \end{aligned} \tag{8}$$

BW-MAX-HET: For this problem we need to add one variable, $bwmax$, and $|\mathcal{P}|^2$ constraints:

$$\begin{aligned} \forall u, v \quad \sum_{p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ \sum_{j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq bwmax \end{aligned} \tag{9}$$

and the objective becomes: $\min(bwmax)$.

5 Heuristics

In this section we propose several polynomial heuristics[1] for the PROC-POWER problem, in which we consider only the compute capacity of processors enrolled for computation. Two heuristics use a random approach to process application nodes, while the others are based on tree traversals. As for the choice of an appropriate resource for the current node, four different processor selection strategies are implemented (and shared by all heuristics). Two selection strategies are *blocking* and two are *non-blocking*. *Blocking* means that once chosen for a given operator op_1, a processor cannot be reused later for another operator op_2, and it is only possible to add relatives (i.e., father or children) of op_1 to this processor. On the contrary, *non-blocking* strategies impose no such restrictions. We start with a description of the four processor selection strategies, and then we move to a brief overview of each heuristic.

Processor Allocation Strategies

(1) Fastest processor first (blocking) – Every time we have to chose a processor, the fastest remaining (not already chosen) processor is chosen.

(2) Biggest network card first (blocking) – Every time we have to chose a processor, the remaining processor with the biggest network card is chosen.

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

(3) Fastest remaining processor (non-blocking) – The actual amount of computation is subtracted from the computation capability, and the processor with the most remaining computation power is chosen.

(4) Biggest remaining network card (non-blocking) – In this strategy the current (already assigned) communication volume is subtracted from the network card capacity to evaluate the processor whose remaining communication capacity is the biggest. This processor is chosen.

Significance of Node Reuse

Our heuristics, except RandomNoReuse (H1), are designed for node reuse. This means that we try to benefit from the fact that different applications may have common subtrees, i.e., subtrees composed of the same operators. Instead of recomputing the result for such a subtree, we aim at reusing the result. For this purpose we try to add additional communications as can be seen in Figure 2. The processor that computes the left 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 by op_2 in application 2 when we add the communication between op_2 in application 1 and op_3 in application 2.

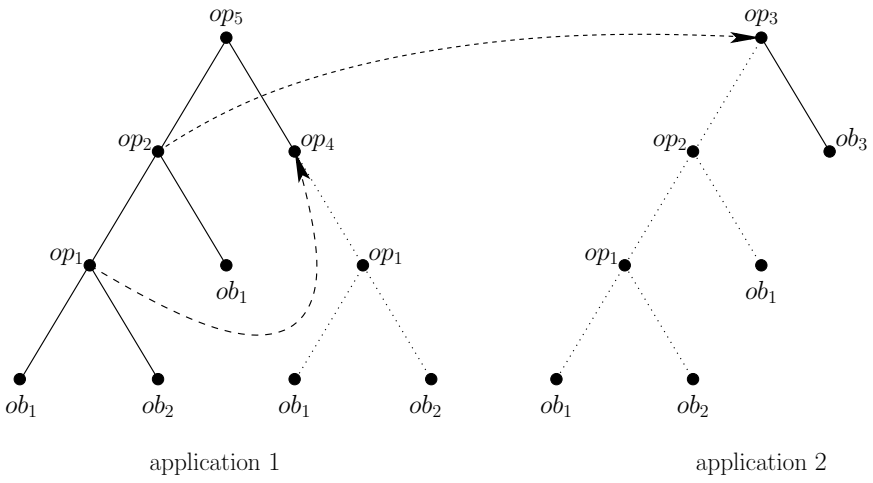


Figure 2: Example for the reuse of nodes. op_1 is only computed once and its result is reused for the computation of op_2 and op_4. op_3 uses the result of op_2 in application 1 for its computation.

We give hereafter a brief overview of each heuristic:

H1: RandomNoReuse

The H1 heuristic does not reuse any result. While there are unassigned operators, H1 randomly picks one of them. If the father is already mapped, it tries to map the operator on the father's processor, or it tries the children's processors, if those are already mapped. If none of these mappings is possible, H1 chooses a new processor according to the processor selection strategy, and maps the operator. If this is not possible, H1 fails.

H2: Random

The H2 random heuristic is more sophisticated as it tries to reuse common results. If the randomly chosen operator has not already been mapped, possibly for another application, we use the same mechanism as in H1: first try to map the operator on its father's processor or one of the children's, and in case of failure choose a new processor. But, if the operator has already been mapped somewhere else in the forest, we try to add a link from the already mapped operator to the father of the actual operator to reuse the common result. When this is possible, we mark the whole subtree (rooted at the operator) as mapped. Otherwise, we choose a new processor.

H3: TopDownBFS

The H3 heuristic performs a breadth-first-search (BFS) traversal of all applications. We use an artificial root node to link all applications, i.e., all application roots become children of the artificial root. For each operator, we check whether the operator has not been mapped yet and whether its father 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. In the case where the actual operator has already been mapped onto one or more processors, H3 tries to add a communication link between the mapped operator and the father of the actual operator: the mapped operator sends its result not only to its father but also to the father of the actual operator. If none of these two conditions holds, or if the mapping was not possible, H3 tries to map the operator onto a new processor. The processor is chosen according to the processor selection strategy. When the mapping is successful, the BFS traversal is continued, otherwise H3 fails.

H4: TopDownDFS

The H4 heuristic uses the same mechanism as H3, but operators are treated in depth-first-search (DFS) manner. Thus, each time a mapping of a node is successful, the heuristic continues the DFS traversal of the current application tree.

H5: BottomUpBFS

As the H4 heuristic, the H5 heuristic makes a BFS traversal of the application forest. For this purpose we use the same mechanism of a new artificial root that links all applications. For each operator, H5 verifies whether it has already been mapped on a processor. In this case a communication link is added (if possible), connecting the mapped operator and the father of the unmapped operator. If the operator is not yet mapped and if it has some children, we try to map the operator to one of its children's processors. If no such possibility is successful, or if the operator is at the bottom of a tree, H5 tries to map the operator onto a new processor (where the processor is chosen according to the processor selection strategy). When the mapping is successful, the BFS traversal continues, otherwise H5 fails.

H6: BottomUpDFS

The H6 heuristic is similar to H5, but instead of a BFS traversal, it performs a DFS traversal of the application forest. This makes the heuristic a little bit more complicated, as there are

more cases to be considered. For each node we check if its operator has already been mapped on a processor, and none of its children are. In this case we go up in the tree until we reach the last node n_1 such that there exists a node n_2 somewhere else in the forest which is already mapped, and such that $op(n_1) = op(n_2)$. In this case we try to add a communication between n_2 and the father of n_1 to benefit from the calculated result. If the children have already been mapped we simply try 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, according to the processor selection strategy. Otherwise H6 fails.

6 Experimental Results

We have conducted several experiments to assess the performance of the different heuristics described in Section 5. In particular, we are interested in the impact of node reuse on the number of solutions found by the heuristics.

6.1 Experimental Plan

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 parameters for 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.

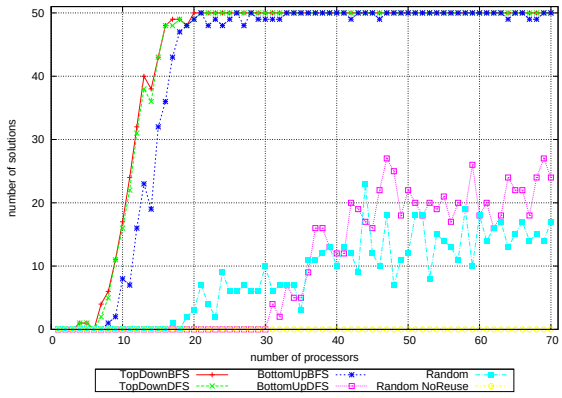
Throughout most of our experiences we use the following platform configuration (variants will be mentioned explicitly when needed.) We dispose of 30 processors. Each processor is equipped with a network card, whose bandwidth limitation varies between 50MB and 180MB. We use the same range for computation power, i.e., CPU speeds of 50MIPS to 180MIPS. The different processors are interconnected via heterogeneous communication links, whose bandwidth 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.

To assess performances, 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 the 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 by one of the heuristics for run r, and $cost_h(r)$ is the cost involved by the solution proposed by heuristic h. Note that in the definition of the relative performance we do account for the case when a heuristic fails on a given instance. The number of runs is fixed to 50 in all experiments. The complete set of figures summarizing all experimental results is available on the web [1].

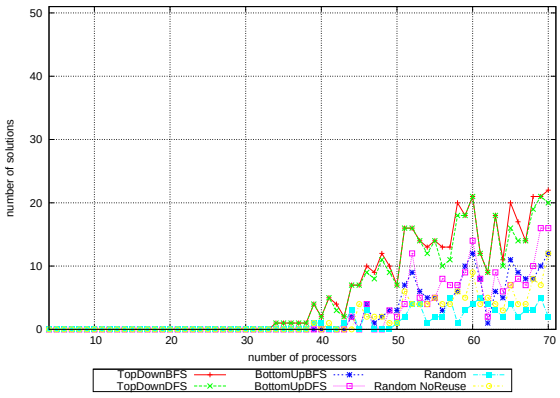
6.2 Results

6.2.1 Experiment 1: Number of Processors

In a first set of experiments, we test the influence of the number of available processors, varying it from 1 to 70. Figure 3(a) shows the number of successes of the different heuristics using selection strategy 3 (biggest remaining network card). Between 1 and 20 processors, the number of solutions steeply increases for TopDownDFS, TopDownBFS and BottomUpBFS and for higher numbers of processors all three heuristics find solutions for most of the 50 runs. BottomUpDFS finds solutions when more than 30 processors are available. Random already finds solutions when only 20 processors are available, but for the runs with more than 30 processors, it finds fewer solutions than BottomUpDFS. RandomNoReuse is not successful at all, it does not find any solution. To summarize, TopDownBFS finds the most solutions, shortly followed by TopDownDFS and BottomUpBFS. Comparing the success rates of the different selection strategies, all heuristics find the most solutions using strategy 3, followed by strategy 4, strategy 2, and finally strategy 1. 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 4(a) shows the relative performance using strategy 3. Comparing with Figure 4(b), we can conclude that for the same number of successful runs, the performances of the heuristics significantly differ according to the selected processor selection strategy. Using strategy 3 (and also strategies 2 and 4), TopDownDFS performs better than TopDownBFS, which performs better than BottomUpBFS. However, BottomUpBFS outperforms both TopDown heuristics when strategy 1 is used. The performance of BottomUpDFS 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 (strategy 3) or even 60 (strategy 2). Independently of the processor selection strategy, both TopDown heuristics outperform all other heuristics in success and performance, but the results are poor (see Figure 3(b)).

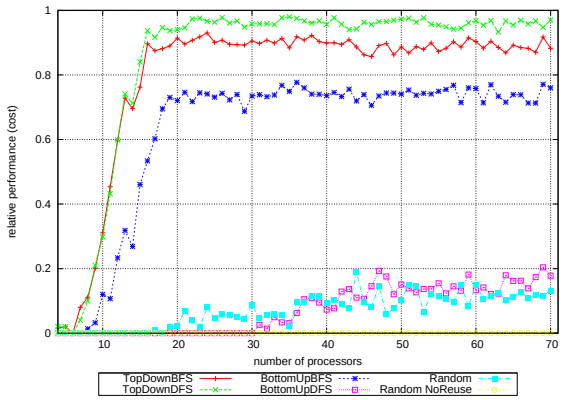


(a) Successful runs, strategy 3

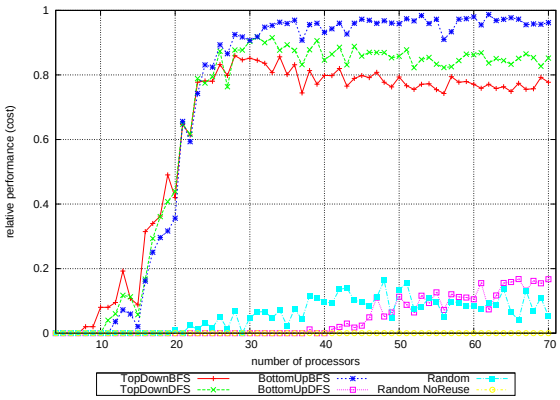


(b) Successful runs without reuse, strategy 3

Figure 3: Experiment 1: Increasing number of processors. Number of successful runs.



(a) Relative performance, strategy 3

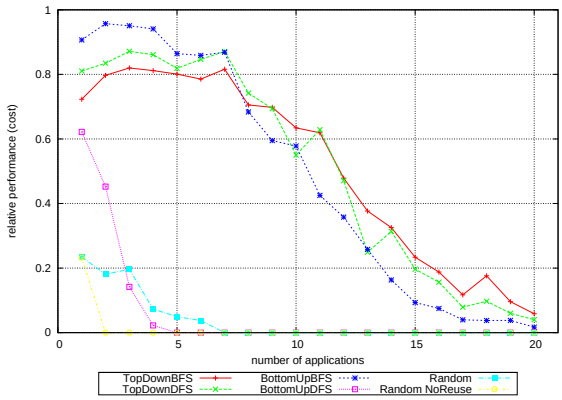


(b) Relative performance, strategy 1

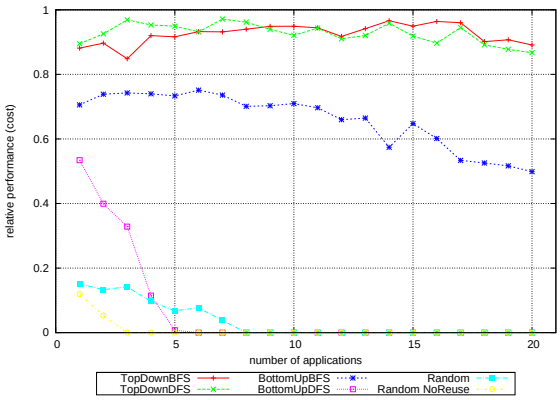
Figure 4: Experiment 1: Increasing number of processors. Relative performance.

6.2.2 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 1 and 2 than with strategies 3 and 4, and relative performance is poorer as well. Regardless of the strategy used, both TopDown heuristics show a better relative performance than BottomUpBFS, with the only exception using strategy 1 with a small number of applications (Figure 5(a)). BottomUpDFS and both random heuristics perform poorly. For instance, BottomUpDFS only finds solutions with up to 4 applications. The best strategy seems to be strategy 3 in combination with TopDownBFS for more than 10 applications and TopDownDFS for less than 10 applications (see Figure 5(b)).



(a) Relative performance, strategy 1



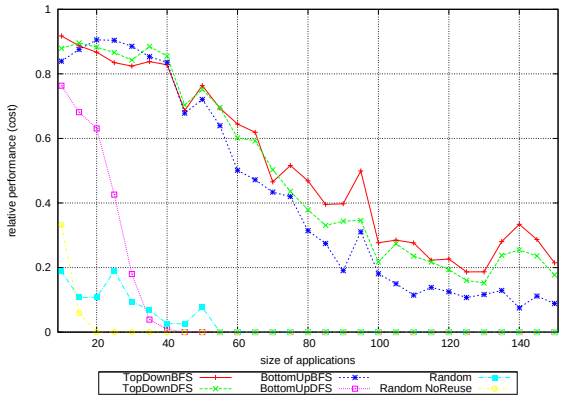
(b) Relative performance, strategy 3

Figure 5: Experiment 2: Increasing number of applications.

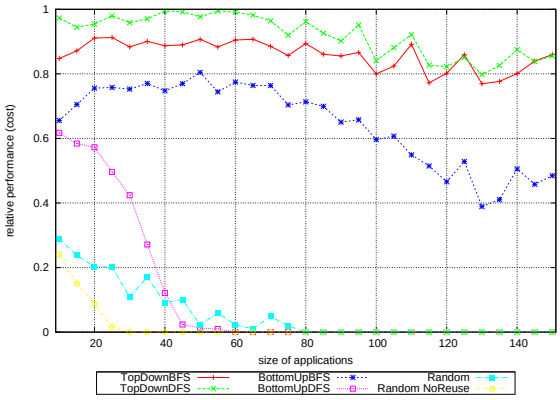
6.2.3 Experiment 3: Application Size

When increasing the application sizes, strategy 3 is the most robust. Up to application sizes of 40 operators, the other strategies are competitive, but for applications bigger than 40 oper-

ators both TopDown heuristics and BottomUpBFS achieve the best relative performance and find the most solutions. The success ranking of the three heuristics is the same, independently of the strategy: TopDownBFS finds more solutions than TopDownDFS, which, in turn, finds more solutions than BottomUpBFS. RandomNoReuse finds solutions for applications with fewer than 20 operators, BottomUpBFS up to 40 operators and Random 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 bigger than 20 using strategy 3. BottomUpDFS is competitive when using strategy 1 for applications smaller than 40 operators (compare Figures 6(a) and 6(b)). As for the heuristics without reuse of common subtrees, they no longer find results when application sizes exceed 40 operators. TopDown heuristics perform better, and the best strategy is one of the two non-blocking ones (3 or 4).



(a) Relative performance, strategy 1



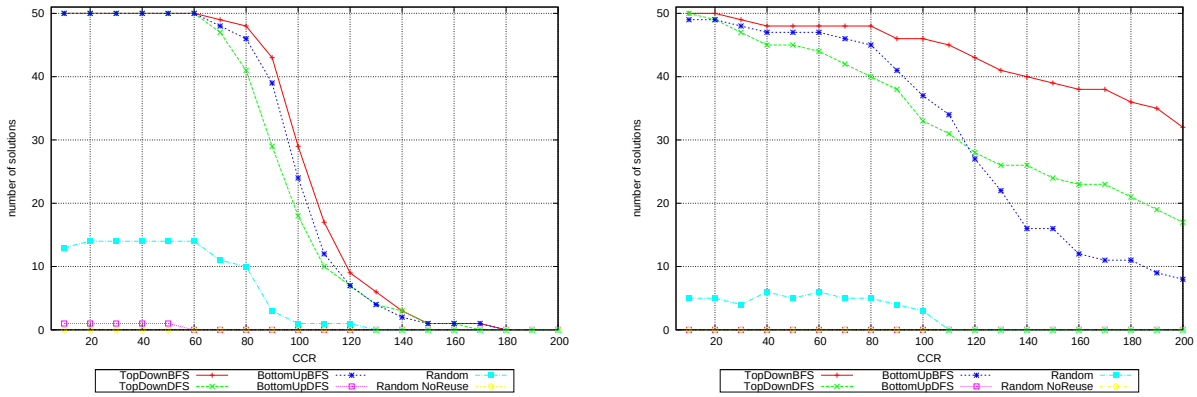
(b) Relative performance, strategy 2

Figure 6: Experiment 3: Relative performance for increasing application sizes.

6.2.4 Experiment 4: Communication-to-Computation Ratio (CCR)

For this experimental set we introduce a new parameter, the CCR, which is 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 3 and 4 react very sensitively. As can be seen in Figure 7(a), TopDownBFS, TopDownDFS and BottomUpBFS have a 100% success rate for CCR ≤ 60, but then the success decreases drastically until no solution is found at all for a CCR of 180 (using strategy 2, TopDownBFS still finds 32 solutions). BottomUpDFS is largely outperformed by Random, and RandomNoReuse fails completely. In this experiment, strategy 2 seems to be the most successful processor selection strategy (see Figure 7(b)). TopDownBFS achieves the best results, followed by BottomUpBFS for CCR < 120, and by TopDownDFS for CCR > 120. Interestingly, the relative performances of the heuristics using the different strategies do not directly mirror their success rates. Compare Figures 8(a) and 8(b): BottomUpBFS finds fewer solutions using strategy 1 than 2, but its relative performance using strategy 1 and CCR smaller than 80 is better than when using strategy 2. Furthermore, TopDownBFS using strategy 1 always finds the most solutions of all heuristics, but its relative performance is only the best when the

CCR becomes bigger than 120. Also, TopDownDFS finds fewer solutions than TopDownBFS and BottomUpBFS using strategy 2 and CCR= 30, but its relative performance is the best.



(a) Successful runs, strategy 3

(b) Successful runs, strategy 2

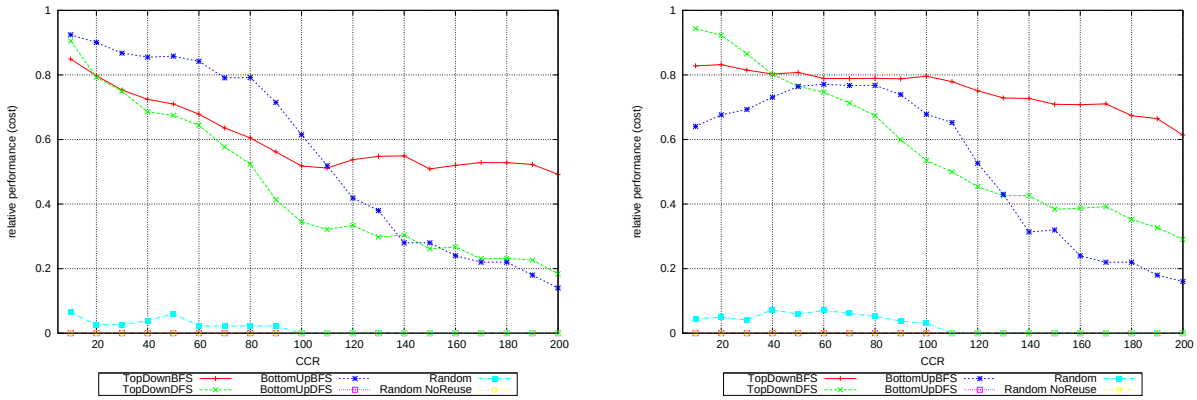
Figure 7: Experiment 4: Communication-Computation Ratio CCR. Number of successful runs.

6.2.5 Experiment 5: Similarity of Applications

In this last experiment, we use only two applications for each run and the processing platform is smaller, consisting of only 10 processors. We study the influence on our heuristics when applications are very similar or completely different. For this purpose we create applications that differ in more and more operators. Strategies 1 and 2 are more sensitive to application differences and we observe the following ranking for the success of the heuristics: strategy 3 > strategy 4 > strategies 1 and 2, which have similar success rates (compare Figures 9(a) and 9(b).) The ranking of the heuristics within the different strategies is the same: TopDownBFS is the most successful, followed by TopDownDFS and BottomUpBFS. BottomUpDFS and Random keep the fourth place, while RandomNoReuse fails. TopDownBFS has the best relative performance using the blocking strategies, whereas in the non-blocking cases TopDownDFS achieves the best results, which is important as its success rate is slightly poorer. BottomUpBFS always ranks at the third position.

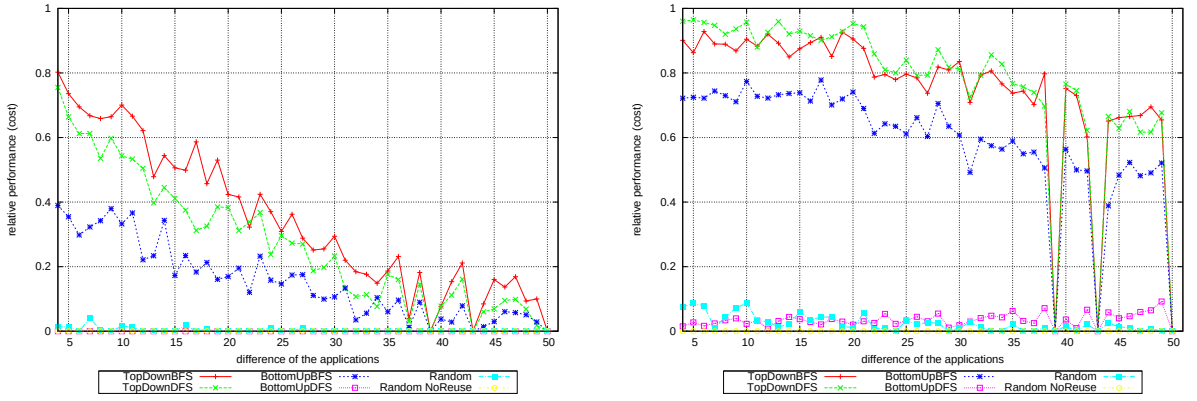
6.2.6 Summary of Experiments

Our results show that a random approach for multiple applications is not feasible. Neglecting the possibility to reuse 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, whereupon 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. In the other cases, TopDownBFS in combination with strategy 3 proves to be a solid combination.



(a) Relative performance, strategy 1 (b) Relative performance, strategy 2

Figure 8: Experiment 4: Communication-Computation Ratio CCR. Relative performance.



(a) Successful runs, strategy 2 (b) Successful runs, strategy 3

Figure 9: Experiment 5: Similarity of applications.

7 Conclusion

In this paper, 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 considered the problem under a non-constructive scenario, in which a fixed set of computation and communication resources is available and the goal is to minimize a cost function. Four different optimization problems were identified. All are NP-hard but can be formalized as integer linear programs. On the practical side we focused on one of the optimization problems, for which we designed a set of polynomial-time heuristics. We evaluated these heuristics via extensive simulations, and our experiments showed 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 concluded that top-down traversals of the application trees is more efficient than bottom-up approaches, and in particular the combination of a top-down traversal with a

breadth-first search (i.e., our heuristic TopDownBFS) 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 safety (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] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan 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 and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[5] 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.

[6] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves Robert. Resource Allocation Strategies for Constructive In-Network Stream Processing. Research Report 2008-20, LIP, ENS Lyon, France, June 2008.

[7] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves 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.

[8] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Conference on Mobile Data Management*, 2001.

[9] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the SIGMOD Intl. Conf. on Management of Data*, pages 379–390, 2000.

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

[11] Liang 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.

[12] 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.

[13] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[14] 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.

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

[16] B. Hong and V.K. 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] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER, September 2003.

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

[19] 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.

[20] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

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

[22] 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.

[23] Suman Nath, Amol Deshpande, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. IrisNet: An Architecture for Internet-scale Sensing Services.

[24] Vinayaka Pandit and Huibo Ji. Efficient in-network evaluation of multiple queries. In *HiPC*, pages 205–216, 2006.

[25] 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.

[26] 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.

[27] 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.

[28] 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.