Proc. Int. Workshop on Database Programming Languages (DBPL), 1999

A Framework for Optimizing Distributed Workflow Executions

Guozhu Dong[1], Richard Hull[2], Bharat Kumar[2], Jianwen Su[3], and Gang Zhou[2]

[1] Department of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435. gdong@cs.wright.edu
[2] Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. {hull,bharat,gzhou}@research.bell-labs.com
[3] Department of Computer Science, University of California, Santa Barbara, CA 93106. su@cs.ucsb.edu

Abstract. A central problem in workflow concerns optimizing the distribution of work in a workflow: how should the execution of tasks and the management of tasks be distributed across multiple processing nodes (i.e., computers). In some cases task management or execution may be at a processing node with limited functionality, and so it is useful to optimize translations of (sub-)workflow schemas into flowcharts, that can be executed in a restricted environment, e.g., in a scripting language or using a flowchart-based workflow engine.

This paper presents a framework for optimizing the physical distribution of workflow schemas, and the mapping of sub-workflow schemas into flowcharts. We provide a general model for representing essentially any distribution of a workflow schema, and for representing a broad variety of execution strategies. The model is based on families of "communicating flowcharts" (CFs). In the framework, a workflow schema is first rewritten as a family of CFs that are essentially atomic and execute in parallel. The CFs can be grouped into "clusters". Several CFs can be combined to form a single CF, which is useful when executing a sub-schema on a limited processor. Local rewriting rules are used to specify equivalence-preserving transformations. We developed a set of formulas to quantify the metrics used for choosing a near optimal set of CF clusters for executing a workflow. The current paper focuses primarily on ECA-based workflow models, such as Flowmark, Meteor and Mentor, and condition-action based workflow models, such as ThinkSheet and Vortex.

1 Introduction

A workflow management system provides a framework and mechanism for organizing the execution of multiple tasks, typically in support of a business or scientific process. A variety of workflow models and implementation strategies have been proposed [GHS95,WfM99]. Several recent projects have focused on developing architectures and systems that support distributed execution of workflows [AMG+95,DKM+96,WW97,BMR96]. A central problem concerns how to optimally distribute a workflow, i.e., how should the management and execution of tasks be distributed across multiple processing nodes? For example, while communication costs may increase with distribution, execution of task management on the same node that that executes the tasks may reduce communication costs and overall execution time. In some cases, the processing node for executing a sub-workflow may have limited functionality. For this reason, it is useful to optimize translations of some sub-schemas into flowcharts, that can be executed in a restricted environment, e.g., in a scripting language or using a flowchart-based workflow engine. This paper

presents a framework and techniques for optimizing distributed execution of workflows that includes the physical distribution of workflow schemas, and the mapping of sub-workflow schemas into flowcharts.

The framework developed is similar to the framework used in relational database query optimization. In particular, it provides an abstract model for representing "logical execution plans" for workflow schemas. These plans give a partitioning of a workflow schema, indicate data and control flow dependencies between the partitions, and also indicate how the workflow sub-schemas should be implemented. Analogous to the relational algebra, this model is not intended for end-users or workflow designers. Rather it captures relevant features concerning distribution and execution of many workflow models, including those based on flowcharts, on petrinets (e.g., ICN [Ell79]), on an event-condition-action (ECA) paradigm (e.g., Flowmark [LR94], Meteor [KS95], Mentor [WWWD96]), and on a condition-action (CA) paradigm (e.g., ThinkSheet [PYLS96], Vortex [HLS+99a]). Furthermore, the model is closed under a variety of equivalence-preserving transformations and rewriting rules. This permits the exploration of a broad space of possible implementation strategies for a given workflow schema.

The abstract model is based on families of "communicating flowcharts" (CFs); these are flowcharts with specialized mechanisms to support inbound and outbound data flow, and to track control flow information. In the framework, a workflow schema is first rewritten as a family of CFs which are essentially atomic. These can be viewed as executing in parallel, while satisfying the synchronization constraints implied by the original workflow schema. The CFs can be grouped into "clusters"; in one approach to implementation each cluster is executed on a separate processing node, and data can be shared with minimal cost between the CFs in a cluster. These parts of the framework are useful in studying the costs and benefits of different distributions of the execution of a workflow schema, assuming that each of the processing nodes provides a workflow engine that supports parallel execution of workflow tasks within a single workflow instance.

In some applications, it may be appropriate to execute a sub-workflow on a processing node that does not support a general-purpose workflow engine. One motivation for this is to put task management on the same platform as task executions, e.g., if several related tasks are executed on the same platform. In such cases, the processing node might be a legacy system that does not support full-fledged parallel workflow execution. Indeed, the processing node might be part of a very limited legacy system, such as a component of a data or telecommunications network. A sub-workflow might be executed on such systems by translating it into a script, which is structured as a flowchart and invokes tasks in a synchronous fashion.

In other applications, it may be desirable to execute a sub-workflow on a restricted, flowchart-based workflow engine, in order to take advantage of existing interfaces to back-end components, rather than installing a general-purpose workflow engine and building new interfaces. (The customer-care focused workflow system Mosaix [Mos] is one such example.) In the abstract model presented in the current paper, several communicating flowcharts can be composed to form a single communicating flowchart; these are suitable for execution on limited processing nodes.

To summarize, there are three main components in the abstract model:

(a) The use of one or more flowcharts, executing in parallel, to specify the internal operation of a workflow schema;
(b) A coherent mechanism for describing how these flowcharts communicate with each other, including both synchronization and data flow; and
(c) An approach that permits data flow and control flow dependencies to be treated in a uniform manner.

In addition, the paper presents a first family of rewriting rules that can express a broad family of transformations on logical execution plans.

So far we have discussed logical execution plans, which are high-level representations of how a workflow can be distributed. Our framework also includes the notion of "physical execution plan", which incorporates information about what processing nodes different clusters will be executed on, what networks will be connecting them, and how synchronization and data flow will be handled. We have also developed formulas for computing the costs of different physical execution plans, based on the key factors of response time and throughput.

The key difference between our framework and that for query optimization is that Our framework is based on a language for coordinating tasks, rather than a language for manipulating data. As a result, the major cost factors and optimization techniques are different, and related to those found in distributed computing and code re-writing in compilers.

The focus of this paper is on the presentation of a novel framework for optimizing the distribution and execution of workflow schemas. Due to space limitations, the paper does not explore in depth the important issue of mechanisms to restrict or otherwise help in exploring the space of possible implementations. The techniques used in query optimization may be an appropriate starting point on this problem.

In this paper we focus on compile-time analysis of workflow schemas, and mapping of parallel workflows into flowcharts. However, we expect that the framework and principles explored here are also relevant to other situations. For example, decisions concerning the distribution of work for an individual instance of a workflow schema could be performed during runtime. Specifically, at different stages of the execution of the instance, the remaining portions of a workflow schema could be partitioned and distributed to different processors. Further, if required by some of the processors, selected sub-workflows could be translated into flowcharts. The specific partitioning, and the optimal flowcharts, will typically be dependent on the data obtained from the processing of the workflow instance that has already occurred.

Related Work: The area of optimization of workflow executions is a relatively new field, and few papers have been written on the topic. Distribution is clearly advantageous to support scalability, which is becoming a serious issue as more and more users access popular web interfaces for digital libraries and online shopping etc. As noted above, several projects have developed distributed workflow systems [AMG+95,DKM+96,WW97,BMR96], but to our knowledge the literature has not addressed the issue of optimal distributions of workflows.

A family of run-time optimizations on centralized workflows is introduced in [HLS+99b,HKL+99], and studied in connection with the Vortex workflow model.

One kind of optimization is to determine that certain tasks are *unneeded* for successful execution of a workflow instance. This is especially useful in contexts where the workflow is focused on accomplishing specified "target" activities, and where intermediate activities can be omitted if not needed. Another kind of optimization, useful when parallel task execution is supported, is to support eager parallel execution of some tasks in order to reduce overall response time. References [HLS+99b,HKL+99] explore how these and other optimizations can be supported at *runtime*. The current paper provides a complementary approach, focusing on the use of these and related ideas for optimizations used primarily at *compile-time*.

Our model of CFs is closely related to that of communicating sequential processes (CSP) with two main differences: (1) in our model, a set of flowcharts (processes) starts at the beginning and they will not spawn new processes, and (2) we carefully distinguish between communications of getting an attribute value instantaneously and that of getting the value only after it has been defined.

Organization: §2 presents some motivating examples. §3 introduces a formal model of CFs, and describes approaches for implementing the model. §4 presents a physical model for distributed workflows and analyzes key cost factors for them. §5 presents a representative family of rules and transformations on clusters of CFs. §6 offers brief conclusions.

2 Motivating Examples

We provide a brief introduction to the framework for workflow distribution, including: (a) the abstract model used to represent distributed workflow schemas; (b) the basic approach to partitioning workflow schemas across different nodes; and (c) techniques for creating and manipulating flowcharts which can execute workflow subschemas on limited processing nodes. §3 gives a formal presentation of the abstract model, and §3 and §4 discuss different options for implementing the model.

Fig. 1 shows a workflow schema with 5 tasks (shown as rectangles). We assume that each task has the form $A = f(A_{i_1}, ..., A_{i_n})$, where f is a function call that returns a value for attribute A, and may have side-effects. As typical with many workflow models, there are two activities with each task: task *management* and task *execution*. In a centralized WFMS all tasks (and associated data management) are managed by one node, and the task executions are carried out by one or more other nodes. In our framework, the task management and execution may be performed by the same or different nodes. We generally assume in this paper that the data management associated with a task is performed by the node that is performing the task management, but this assumption can be relaxed.

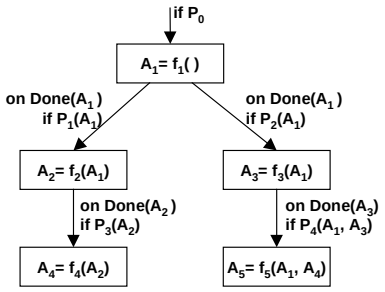


Fig. 1. Example ECA workflow schema

In workflow specifications, the attribute names have global scope, and so data flow is expressed implicitly, rather than using explicit data channels. This follows

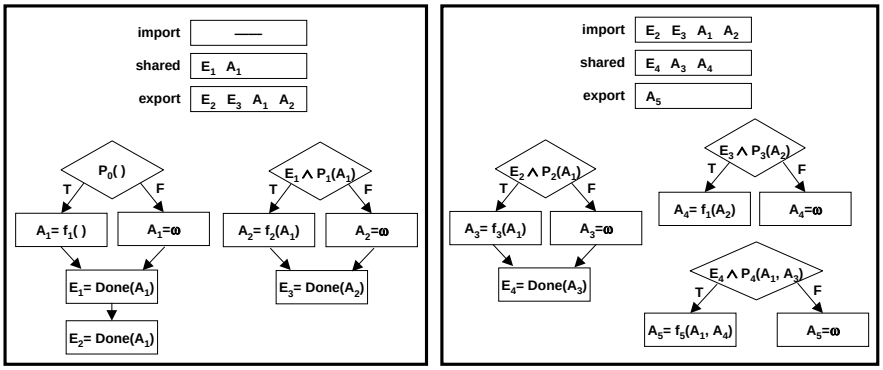


Fig. 2. Decomposition of schema into flowcharts, and partition onto two processing nodes

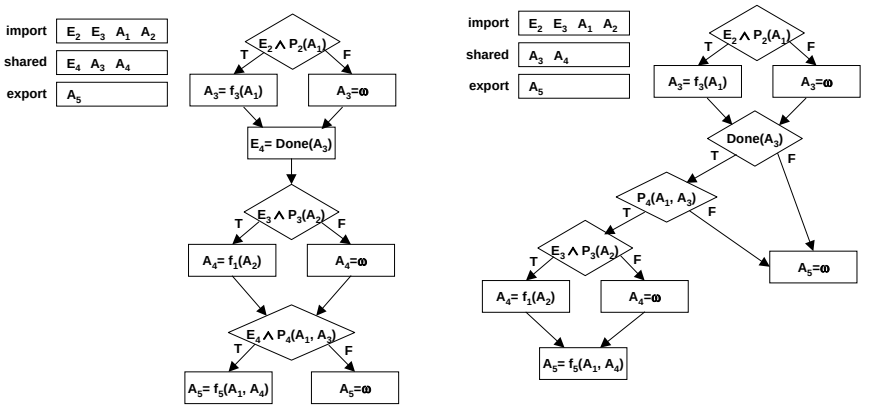
ThinkSheet and Vortex, and contrasts with the syntax of models such as FlowMark or Meteor. The use of global attributes or explicit data channels does not affect the expressive power of the models.

In Fig. 1 the edges indicate control flow. Except for A_1, the conditions include both events and predicates. In this example the events have the simple form "Done(A)", but other kinds of events can be incorporated. Data flow in this schema is indicated by how attributes are used in tasks; e.g. attribute A_4 is used for the task of computing A_5. The semantics of this workflow schema follows the spirit of ECA-style workflow models, such as Flowmark, Meteor, Mentor. A task (rectangle node) should be executed if its enabling condition is true, i.e., if the event in that condition is raised, and if subsequently the propositional part of the condition is satisfied.

We now use Fig. 2 to introduce key elements of the abstract model and our framework. First, focus on the 5 "atomic" flowcharts (ignore the two large boxes). Each corresponds to a single task in the schema of Fig. 1. These are *communicating flowcharts* (CFs). In principle, a parallel execution of these flowcharts is equivalent to an execution of the original workflow schema using a generic, parallel workflow engine. When these flowcharts are executed, attributes will be assigned an actual value if the corresponding enabling condition is true, and will be assigned the null value ω (for "disabled") if the condition is evaluated and found to be false.

The abstract model supports two different perspectives on attributes, that correspond to how attributes are evaluated in ECA models (e.g., FlowMark, Meteor, Mentor) vs. in condition-action models (e.g., ThinkSheet, Vortex). In ECA models, an attribute is read "immediately" when its value is needed, for this we use $read(A)$ operations. In CA models an attribute is read only after the attribute has been initialized, either by executing a task and receiving an actual value, or by determining that the attribute is disabled and receiving value ω. We use the operation $get(A)$ to indicate that the value of A is to be read only after it is initialized.

In addition to permitting support for different kinds of workflow models, the use of two perspectives on attribute reads permits us to treat events as a special kind of attribute. In Fig. 2 the events are represented as attributes E_i, using the get semantics; an attribute E is considered to remain uninitialized until some flowchart gives



(a) (b)

Fig. 3. Two flowcharts, each equivalent to the three flowcharts of Fig. 2(b)

it a value. (The launching of a flowchart for a given workflow is not represented explicitly in the abstract model, and different implementations of this are possible.)

In Fig. 2, each large square denotes a cluster of atomic flowcharts. Each cluster could be executed on a different node. The cluster includes a listing of attributes that are imported by the cluster, exported by it, or used internally (i.e., shared by the CFs within the cluster). We typically assume that each node has a *data repository* that maintains this data, which includes asynchronously receiving import data as it becomes available, and transmitting export data to the appropriate places.

Fig. 2 shows one execution plan for the initial workflow schema of Fig. 1, where tasks A_1 and A_2 are executing on one node, and tasks A_3, A_4, and A_5 are executing on another node. Another execution plan is formed by moving the flowchart for task A_3 from the second node to the first (and adjusting the schemas of the data repositories accordingly). What is the difference in the costs of these two execution plans (call them P_1 and P_2)? There are various factors, including communication costs between the two clusters (an important issue here is the relative sizes of expected values for A_2 and A_4, and between the clusters and the location of task executions). Another factor, that may arise if one of the clusters is on a limited node, concerns the overall CPU and data management load that the cluster imposes on that node.

Fig. 3 shows two ways that the three atomic CFs of Fig. 2(b) can be combined to form larger CFs. Fig. 3(a), shows one way to combine the three CFs into a single flowchart. This corresponds to a topological sort of the sub-schema of Fig. 1 involving A_3,A_4,A_5. An analogous flowchart could be constructed using the order A_4, A_3, A_5. If tasks A_3 and/or A_4 has a side-effect, then executing in one or the other order may be preferable, e.g., to have the side-effect occur as quickly as possible.

Assume now that in the initial workflow tasks A_1, A_3, and A_5 are "target" tasks, but that the other two are for internal purposes only. In this case it would be desirable, for a given workflow instance, to omit execution of A_4 if it is not needed for successful completion of the target tasks for that instance. Fig. 3(b) shows CF that is equivalent to that of Fig. 3(a). (§5 describes how the first CF can be transformed into the second one using a sequence of equivalence preserving rewrite rules.) In

particular, in Fig. 3(b) task A_4 is not executed if task A_5 will not be executed, i.e., if $E_4 \wedge P_4(A_1, A_3)$ turns out to be false.

Our abstract framework permits a more flexible semantics of workflow execution than found in some ECA models, such as FlowMark or Meteor. In those models, when an event fires the associated conditions should be queued immediately and tested soon thereafter, and likewise if the condition is true then the associated task should be queued immediately and launched soon thereafter. In our model, we generally require only that conditions are tested sometime after the associated event, and tasks launched sometime after the associated condition comes true. More stringent timing requirements can be incorporated, if appropriate for some application.

3 A Model of Communicating Flowchart Clusters (CFC)

This section describes a formal model of communicating flowchart clusters for representing logical execution plans of tasks in workflow schemas, and discusses some implementation issues.

We assume the existence of attributes (and domains). A *task* is an expression $A = f(A_1, ..., A_n)$ where f is an (uninterpreted) function (procedure) name, A_i's are input attributes and A is an output attribute (*defined* by the task). In general we treat tasks as black boxes, i.e., the functions are uninterpreted. However, we note that a task may have "side-effects".

A *condition* over attributes $A_1, ..., A_k$ is a Boolean combination of predicates involving $A_1, ..., A_k$ and constants in their respective domains; the attributes used in the condition are also called the input attributes to the condition.

Definition. A *flowchart* is a tuple $\mathsf{f} = (T, C, E, s, L, I, O)$ where (1) T is a set of (attribute or task) nodes, (2) C is a set of (condition) nodes disjoint from T; (3) $s \in T \cup C$ is the *entry* node; (4) $E \subseteq (T \cup C) \times (T \cup C)$ such that (a) each $t \in T$ has at most one outgoing edge, (b) each $c \in C$ has at most two outgoing edges, and (c) every $x \in T \cup C - \{s\}$ is reachable from s; (5) L maps each task node to a task and each condition node to a condition; (6) I is a set of import attributes used by some tasks in f; and (7) O is a set of export attributes defined in f. Moreover, the flowchart f is *trivial* if $T \cup C = \{s\}$ and E is empty.

Let $\mathsf{f} = (T, C, E, s, L, I, O)$ be a flowchart. We denote import and export attributes of f as $in(\mathsf{f}) = I$ and $out(\mathsf{f}) = O$. In this paper we focus on "acyclic" flowcharts: A flowchart (T, C, E, s, L, I, O) is *acyclic* if the graph $(T \cup C, E \cup D)$ has no cycles, where $D = \{(u, v) \mid u \text{ defines an attribute used by } v\}$.

Example 1. *Fig. 2(a) shows two flowcharts. The one in the left-hand side can be described as* $\mathsf{f}_1 = (T, C, E, c, L, \emptyset, \{A_1, E_1, E_2\})$*, where* $T = \{a_1, ..., a_4\}$*,* $C = \{c_2\}$*,* a_1, a_2 *(resp.* a_3, a_4*) are two nodes defining attribute* A_2 *(resp.* E_1, E_2*),* c_2 *the entry condition node and* $E = \{(c_2, a_1), (c_2, a_2), (a_1, a_3), (a_2, a_3), (a_3, a_4)\}$*. In fact,* f_1 *is acyclic.*

Semantics (or *execution*) of a flowchart is defined in the straightforward way with the exception of acquiring values for import attributes. Initially all attributes have the null value $\perp$ (stands for "uninitialized"). One method of acquiring a value

for an import attribute A, called *immediate read* and denoted as *read*(A), is to retrieve the current value of the attribute, regardless of whether A has a proper value or the null value. This method is used in many workflow systems such as FlowMark and Meteor. Immediate read is however undesirable sometimes because the timing of tasks and external events may cause delays in computing attribute values which may cause nondeterministic behaviors of the workflow system.

In acyclic flowcharts, a task may be executed at most once. This allows an alternative method, called *proper read*, which does the following. When a value for an import attribute A is requested, if A has a proper value (non-$\perp$), the value is fetched; if A currently has the value $\perp$, the task requesting A waits until A is assigned a proper value, and then the new value is fetched. We denote this operation as *get*(A). This operation is used in the Vortex paradigm which provides a declarative workflow specification language.

From now on, we assume that each input attribute of every task must be specified with either a *read* or *get* operation. Although *get* operations provide a clean interaction between tasks, they may cause the execution of CFCs to "stall". For instance, if there are no tasks defining A prior to a *get*(A) operation in a single flowchart, the *get* operation will be blocked forever. We assume that the flowcharts will never stall.

Proper read operations alone do not guarantee determinism. A flowchart $\mathsf{f} = (T, C, E, s, L, I, O)$ is said to be *write-once* if in each execution of f, each attribute will be assigned a non-$\perp$ value at most once. It can be verified that every write-once flowchart with only proper read operations for all import attributes has a deterministic behavior. The complexity of checking the write-once property of flowcharts depends on the condition language.

Definition. A *communicating flowchart cluster* (*CFC*) is a quadruple (F, I, S, O) where F is a set of flowcharts with pairwise disjoint sets of nodes, I (resp. O) a set of import (resp. export) attributes for flowcharts in F such that $I \subseteq \cup_{\mathsf{f} \in F} in(\mathsf{f})$ ($O \subseteq \cup_{\mathsf{f} \in F} out(\mathsf{f})$), and $S = (\cup_{\mathsf{f} \in F} in(\mathsf{f})) \cap (\cup_{\mathsf{f} \in F} out(\mathsf{f}))$ a set of attributes defined and used by different flowcharts in F.

Example 2. *Fig. 2(b) shows a single CFC that includes three flowcharts, where the import/export/shared attributes are listed in the top.*

Definition. Let $\mathbf{F}$ be a set of CFCs and Tg a set of attributes (called the target attributes of the CFCs). $\mathbf{F}$ is said to be *well-formed* w.r.t. Tg if (a) for each CFC $F \in \mathbf{F}$ and each import attribute A in F, there is a CFC $F' \in \mathbf{F}$ which exports A, and (b) each attribute of Tg is exported by some CFC in $\mathbf{F}$.

For each well-formed set $\mathbf{F}$ of CFCs, we define a *dependency graph* $G_{\mathbf{F}}$ which characterizes the dependencies of control and data flows within flowcharts within $\mathbf{F}$. Specifically, $G_{\mathbf{F}} = (V, E)$ where V is the union of all flowchart nodes in CFCs of $\mathbf{F}$, and E contains all flowchart edges in CFCs of $\mathbf{F}$ and all edges (u, v) such that u defines an attribute that is used in v. A well-formed set $\mathbf{F}$ of CFCs is said to be *acyclic* if $G_{\mathbf{F}}$ has no cycles. The set of two CFCs in Fig. 2 is acyclic.

Finally we discuss some key issues in developing implementation models for CFCs. Clearly, the control structures of flowcharts are very simple and generally

available in a variety of script languages; even a direct implementation (of the control structures) is also straightforward. There are, however, three decisions that need further discussion. The first issue is the mapping from flowcharts to processors. The conceptual model of CFCs assumes that each processor may be capable of executing multiple flowcharts concurrently. Thus the mapping is simply to assign each CFC to a processor. The second issue is about how the flowcharts in a set of CFCs are invoked. This can be done by having a single entry to the CFC, that will spawn all flowcharts in the CFC (e.g., through remote procedure calls).

The third issue is related to communication, i.e., passing attribute values between tasks. There are three different ways an attribute value can be sent around. (1) The attribute is defined and used in the same flowchart. If the attribute is accessed through a *read* operation, we just need to provide some storage space local to a flowchart for storing the attribute value. For acyclic flowcharts, since each *get* operation of an attribute is required to follow the task defining the attribute, *get* and *read* operations are effectively the same. (2) The attribute is defined and used in different flowcharts in the same CFC. In this case, we need to provide a common buffer (write exclusive) for all flowcharts (processes) involved. While *read* operations do not require extra support, *get* operations would require a list to be maintained for each attribute that have not received a proper value. (3) In the most general case, the attribute is defined in one CFC and used in another. There are many ways to implement *read* and *get*. For example, both "producer" and "consumer" processors maintain a buffer for holding the attribute value. Passing the value from the producer buffer to the consumer buffer can be done by push or pull. Similar to case (2), *get* operations still rely on maintaining a list of requests. One possible architecture for handling communications is through a data repository.

4 Optimization Issues in Choosing Execution Plans

In this section we present a physical model for workflow distribution that extends the logical model of the previous section. We identify key cost factors in the physical model, and illustrate how they affect the cost of different physical execution plans. We then develop a very preliminary cost model, by presenting formulas that can be used to give bounds on the costs of physical execution plans. The two bounds we consider are: total running time for processing a workflow instance, maximum throughput, i.e., maximum number of instances processed in a time unit, and total network load resulted from transferring data between CFCs.

4.1 A physical model for executing communicating flowcharts

We assume that each CFC is executed on a separate processing node. The exchange of attribute values between the CFCs can use push or pull approaches. The execution of tasks in a CF may involve accessing data at a data repository on either the local or a remote processing node. To capture the costs for transferring attribute values between a task manager and the task execution in a uniform way, we conceptually replace the task node in a CFC by three nodes: an entry node, a node for data retrieval at the task execution machine, and an exit node. The in-edges of the original node are now in-edges of the entry node, while the out-edges of the original node

are now out-edges of the exit node. There is an edge from the entry node to the data retrieval node and one from there to the exit node. If the data repository is local, the data retrieval node remains in the same CFC. Otherwise, this node will reside in a separate CFC.

When a new workflow instance is created, we assume for simplicity all the CFs are invoked for this instance at the same time. If an execution step in a CF depends on attribute(s) which have not been defined, the CF waits for the definition of the attribute(s), i.e., until a true value or null value ω is present.

Processing in a flowchart for a given workflow instance terminates when a leaf node of the flowchart completes.

4.2 Comparing alternative CFCs

We give two examples here to illustrate the differences in performance and cost that alternative execution plans for the same workflow schema may have. We first show the difference in terms of response time and network load. Recall the example in Fig. 2 (a) and (b). Suppose that computing attribute A_3 (in CFC(b)) requires retrieving data from a different processing node where CFC(a) resides. A data retrieval node for A_3 is then created in CFC(a) with two edges crossing the CFC boundary leading to the CFC(b) cluster. If we move the CF for computing A_3 to CFC(a), these two edges will be eliminated, which most likely reduces network load and response time. This effect will be shown in our formulas presented in the next section.

The second example concerns Fig. 3. We show that we can transform the CFCs to eliminate unneeded evaluation of attributes during the execution of certain workflow instances. In Fig. 3 (a), the node $A_5 = \omega$ is moved much "closer" to the root node of the CF. In the cases when A_5 is evaluated to ω, we can avoid an unneeded execution of task A_4.

4.3 Cost factors

The logical model of CFCs permits the management of tasks and actual execution of tasks to be on two distinct clusters. For ease of reasoning, we assume that task management nodes and task execution nodes are explicitly specified in a flowchart plan f. The notion of clusters is thus generalized, since all nodes (management or execution) need to be on some processing node. In the rest of this section, we do not distinguish between the two types of management and execution nodes, and refer to them simply as flowchart nodes.

We first define some notation. Let F_A represent the CFC in which an attribute A is computed. Let $|A|$ denote the expected size of A in number of bytes (this includes cases where the transmitted value of A is ω or $\perp$, which is viewed as having a size of one). Let $P_u(F, A)$ denote the probability of attribute A being used in the CFC F. $P_d(A)$ denotes the probability of attribute A being computed. We also use $P_d(X)$ to refer to the probability of any flowchart node X being executed, rather than simply the probability of an attribute being computed. For example, even condition nodes have probabilities associated with them, and reflect the probability of the condition node being evaluated (not the value of the condition being true or false). The actual meaning of X in $P_d(X)$ should be evident from the context.

We now derive bounds on response time and achievable throughput. These bounds depend on the cost associated with executing a workflow instance, which can primarily be broken down into two parts: 1) processing cost of executing flowchart nodes within clusters, and 2) communication cost of passing data between clusters.

Let $Work(A)$ denote the expected number of units of work required to compute node A for one workflow instance. Let $Nodes(F)$ denote the set of nodes in cluster F. $Work(F)$ denotes the total processing cost of cluster F for one workflow instance, and is given by: $Work(F) = \sum_{A \in Nodes(F)} (P_d(A) \cdot Work(A))$.

Communication cost depends on the amount of data transferred between clusters, the effective network bandwidth on the communication link between the clusters, and the method of communication between the clusters. We consider two possible methods of communication, namely, *push* and *pull*.

In the push mode, the value of an output attribute A is pushed to all the clusters F which may use that value, i.e., where $A \in in(F)$. This can result in a reduction of response time by hiding communication latency, since the target attribute does not have to wait for a source attribute value to be delivered when required. Moreover, efficient multicast algorithms are available for distributing a value from a single source to multiple targets. However, one drawback of the push model is that it results in increased network load, since the source value is distributed to clusters which may not use the value in the actual execution.

The pull mode defers communication until it is necessary. When a flowchart node is being computed that requires attribute values from other clusters, communication is initiated to retrieve those values at that point. The advantage of this model is reduced network load, however, reduction in communication latency (as in the push mode) is not achieved. We only consider the pull mode in deriving expressions for communication costs. The costs for the push mode can be derived similarly.

We assume we are given a function $Comm_{F_i,F_j}(d)$ that denotes the communication cost of transferring d bytes of data from cluster F_i to F_j. An example of this function would be: $Comm_{F_i,F_j}(d) = ts_{F_i,F_j} + tb_{F_i,F_j} d.$ where ts_{F_i,F_j} (resp. tb_{F_i,F_j}) denotes the startup time (resp. byte transfer rate) for the communication between clusters F_i and F_j. However, any appropriate cost model can be chosen instead of the above. The total communication cost $Comm$ is then given by: $Comm = \sum_{F \in \mathsf{f}} \sum_{A \in in(F)} P_u(F, A) \cdot Comm_{F_A,F}(|A|)$

We use the term *network load* to refer to the total amount of data transferred between the clusters during the execution of a workflow instance. As we explain later, network load is an important concept used for determining the maximum throughput. The total network load NL is given by: $NL = \sum_{F \in \mathsf{f}} \sum_{A \in in(F)} P_u(F, A) \cdot |A|$

Response Time: We now give present some worst case bounds on average instance response time for a particular flowchart plan f. We assume that the time for computing attributes, and communicating attribute values between clusters, dominates the execution time. We assume that the source attributes for f are available at time $= 0$. We compute the expected response time of f by recursively computing the finish times of each flowchart node, by starting from the source nodes and proceeding in a topological sort order until the target nodes are reached.

Let T_A denote the finish time of the node A. Let $pred(A)$ denote the predecessor set of node A in F_A, and $input(A)$ denote the set of input attributes for node A. Let $Tenabled_A$ denote the expected time instant when it is the turn of A to execute (note that the inputs of A might not be ready at this time). Then, $Tenabled_A = \sum_{B \in pred(A)} (P_d(B) \cdot T_B)$ Now, let $Tready_A$ denote the time when the inputs to A are also ready. The set of inputs to A can be broken down into two categories, 1) those that are computed in the same CFC, and 2) those computed in other CFCs. Note that the acyclicity property of the CFCs implies that the first set of inputs will always be computed by time $Tenabled_A$. For the second set of inputs, assuming actual read, the execution of A is delayed until those inputs are pulled from the corresponding CFCs. Hence,
$Tready_A = \max[Tenabled_A, \max_{B \in (input(A) \cap in(F_A))} (T_B + Comm_{F_B, F_A}(|B|))]$.
Note: for immediate read, the term T_B in this expression would not be present. If $Time(A)$ represents the expected processing time of A, we get $T_A = Tready_A + Time(A)$. The expected response time for the entire workflow is then given by $\max_{A \in T_g}(T_A)$. The above analysis gives an approximation of the response time. To be more accurate, we should incorporate the value for the expected load on the processing nodes, since that impacts the processing time of each node.

Throughput: The maximum achievable throughput is bounded by two factors: 1) processing power of the processing nodes, and 2) network bandwidth between the processing nodes.

Let Cap_F denote the processing capacity (in terms of number of processing units per unit time) of the processing node that executes cluster F. Assuming infinite network bandwidth, the maximum throughput achievable for a given processing node is given by the processing capacity of that node divided by the amount of work to be performed by the corresponding cluster. We denote this by $Throughput_F$: $Throughput_F = \frac{Cap_F}{Work(F)}$. Similarly, we assume that the bandwidth of the link between two processing nodes (clusters) F_i and F_j is $Bandwidth_{F_i,F_j}$. The set of attributes whose value may be transferred from F_i and F_j is $(I_{F_i} \cap O_{F_j})$, and those from F_j to F_i is $(I_{F_j} \cap O_{F_i})$. The maximum throughput achievable due to this link (denoted by $Throughput_{F_i,F_j}$) is given by the link bandwidth divided by the network load generated due to traffic on this link. Assuming a pull mode, we get $Throughput_{F_i,F_j} = \frac{Bandwidth_{F_i,F_j}}{(\Sigma_{A \in (I_{F_i} \cap O_{F_j})}(P_u(F_i,A) \cdot |A|)) + (\Sigma_{A \in (I_{F_j} \cap O_{F_i})}(P_u(F_j,A) \cdot |A|))}$.
Hence, given a flowchart plan, the maximum achievable throughput is given by: $Throughput = \min(\min_{F \in \mathbf{f}}(Throughput_F), \min_{F_i,F_j \in \mathbf{f}}(Throughput_{F_i,F_j}))$

Note that the above analysis presents upper bounds on the throughput. Actual throughput would vary depending on the expected network load.

5 Representative Transformations and Rewriting Rules

We now present some representative transformations and rewriting rules on logical execution plans. The first rule covers moving flowcharts between clusters, and the others focus on manipulating one or two CFs within a cluster. The application of these transformations should be guided by optimization goals and heuristics. These transformations are intended to be illustrative, but not necessarily complete.

The transformations rules take as input a logical execution plan, i.e., a family of CFCs, and produce as output another logical execution plan. To get started, given a workflow schema, we assume that an atomic CF is created for each task in that schema, and that a separate cluster is created for each CF. (An alternative would be to put all of the atomic CFs into a single cluster.)

For this section we assume that all condition nodes have two outgoing edges, one for true and one for false, and that all tasks eventually finish (or are determined to be "dead"). These assumptions ensure that all possible executions of a flowchart will eventually reach a terminal node.

5.1 Moving flowcharts between clusters

We can move flowcharts from one cluster to another, using the move transformation.

Move: This transformation moves a flowchart f from one CFC F_1 to another CFC F_2. In addition to removing f from F_1 and adding f to F_2, we also modify the sets of import, export and shared attributes of the two clusters to reflect the move: We remove those import attributes that are only used by f from the set of import attributes of F_1, and remove the attributes defined by f from the set of export attributes of F_1. Appropriate actions are also needed for F_2.

For example, consider Fig. 2. If we move the flowchart for task A_4 from the CFC of (b) to that of (a), we need to (i) remove E_3 and A_2 from the imports of (b), (ii) remove A_4 from the shared attributes of (b), (iii) add A_4 to the imports of (b), (iv) add A_4 to the exports of (a), (v) add E_3 and A_2 to the shared attributes of (a), and (vi) remove E_3 and A_2 from the exports of (a).

5.2 Combining and decomposing flowcharts

There is one rule for combining flowcharts, and one for decomposing them.

Append: This transformation appends a flowchart f_2 to another f_1 to produce a third f, provided that f_1 does not use any attribute defined in f_2: the node set and edge set of f are the union of the respective sets of f_1 and f_2, and moreover f has the edge (u, v), for each terminal node u of f_1 and the entry node v of f_2.

For example, consider Fig. 2. If we append the flowchart f_2 for task A_5 as entry node to the flowchart f_1 with for task A_3 then we add an edge from $E_4 = Done(A_3)$ to the root of f_2. Observe that we cannot append these two flowcharts in the other order because of the data flow dependency between them.

By combining the Append rule with the Reorder rules (given below) it is possible to achieve other ways of combining flowcharts. As a simple example, given two linear flowcharts with no data flow dependencies between them, one can form a flowchart interleaves the tasks of the two flowcharts in an arbitrary manner.

Split: This is the inverse of append; it splits one flowchart f into two flowcharts, f_1 and f_2. Splitting of f can be done at any node v which is a node cut of f (viewed as a graph): We invent a new (event) attribute E_{f_v}. f_1 consists of everything of f except the branch of f starting from v; furthermore, the edge leading to v now points to a new node with the assignment task $E_{\mathsf{f}_v} = True$. f_2 consists of the branch of f starting from v, plus a new condition node "$E_{\mathsf{f}_v} = True$" whose outgoing True edge goes to v, and outgoing False edge goes to a no-op terminal node. (We can avoid the use of E_{f_v}, if the branch from v does not depend on the rest of f.)

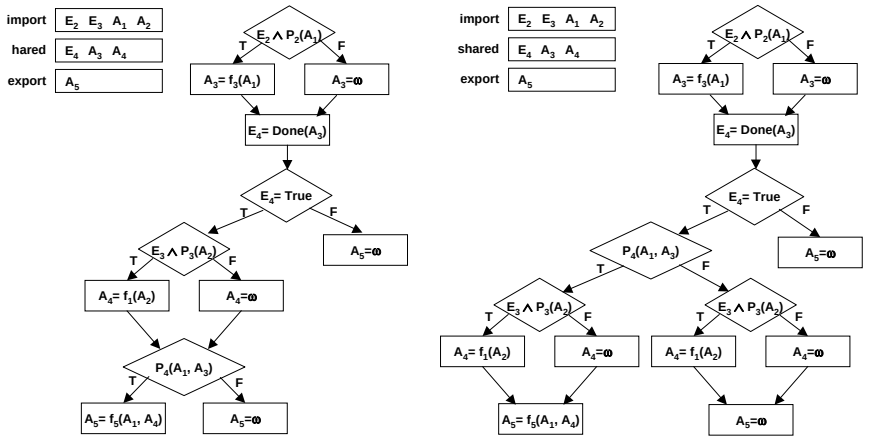


Fig. 4. Flowcharts arising at intermediate points in transformation from Fig. 3(a) to Fig. 3(b)

Split can also be performed using larger node cuts, although it is more complicated. First, we need to create several exit events, one for each element of the node cut. Second, we have to construct series of condition nodes, that essentially perform a case statement that checks which event is true in order to start processing at the correct part of the split-off flowchart.

5.3 Modifying a flowchart

We now present several representative rules for modifying individual flowcharts. These are essentially equivalent to transformations that are used for code rewriting in compiler optimization. We shall illustrate some of the transformations using Fig. 4, which shows two flowcharts that arise when transforming the flowchart of Fig. 3(a) into the flowchart of Fig. 3(b).

A flowchart is said to be *single-entry-single-exit* if it has exactly one entry and one exit node. Observe that a singleton node is a single-entry-single-exit sub-flowchart. For the sake of simplifying the discussion, we assume without loss of generality that the single exit node is a no-op terminal node, by appending one such node if necessary. Observe that this no-op terminal node is always absorbed by the subsequent entry node in the examples.

Reorder: This transformation changes the ordering of two single-entry-single-exit sub-flowcharts f_1 and f_2 when there is no data dependency between them. More specifically, let f_1 and f_2 be two single-entry-single-exit sub-flowcharts of a flowchart such that (i) the exit node of f_1 goes to the entry node of f_2, (ii) there are no other edges leaving f_1, and (iii) there are no other edges entering f_2. This transformation will then exchange the ordering of f_1 and f_2.

For example, consider Fig. 3a. Let the four nodes headed by $E_2 \wedge P_2(A_1)$ be f_1, and the three nodes headed by $E_3 \wedge P_3(A_2)$ plus a no-op terminal node be f_2. Then the ordering of f_1 followed by f_2 can be changed to f_2 followed by f_1.

Reorder through condition: In some cases, it is useful to push a single-entry-single-exit sub-flowchart through a condition node. In the case of pushing a sub-

flowchart downwards through a condition node, the sub-flowchart will be duplicated. Consider the node labeled $P_4(A_1, A_3)$. Let f_1 be the sub-flowchart of three nodes above that condition, along with a no-op terminal node. Fig. 4(b) shows the result pushing f_1 downwards through the condition node. This transformation can also be applied in the opposite direction.

Condition splitting: Suppose there is a condition node with label $C_1 \wedge C_2$. This can be split into two condition nodes, one for C_1 and one for C_2 in the natural manner. A similar transformation can be applied to conditions of form $C_1 \vee C_2$. For example, Fig. 4(a) can be obtained from Fig. 3(a) by the following steps. First, split the condition $E_4 \wedge P_4(A_1, A_3)$, and then use reordering to push E_4 upwards.

Duplicate: Suppose there is a single-entry-single-exit sub-flowchart f_1 whose root has two (or more) in-edges. The duplicate rule permits creating a duplicate copy of f_1, putting one copy below the first in-edge and the other copy below the second in-edge. If the tail of the original f_1 was not a terminal node, then the tails of the copies can be brought together and connected where the original tail was.

Delegate: Intuitively, this transformation works as if it is delegating the task of a node to a different processor. This can be used to increase parallelism. For this to work correctly, we must ensure that the dependencies are taken care of properly. Formally, we replace the old task node with a chain of two nodes: the first one spawns the action of the old task node (e.g., with keyword $spawn(A)$), and the second one is the action of waiting for that task to finish (e.g., by $get(A)$). Reordering can then be used to push the $get(A)$ node downwards in the flowchart.

Remove Unneeded: If a workflow schema has specified target and non-target tasks, then in some cases unneeded tasks can be detected an eliminated from flowcharts. For example, in Fig. 4(b) the two right-most tasks assigning A_4 are not needed anywhere below. As a result, these tasks, and the condition node above, can be deleted. This shows how Fig. 4(b) can be transformed into Fig. 3(b).

Other transformations (e.g.the unduplicate transformation) are also possible.

6 Conclusions

Our paper has focused on distributing workflow tasks among a set of processing nodes and on optimizing execution of the tasks. We developed a general framework for representing logical execution plans using communicating flowchart clusters and an initial set of transformations on logical executions plans. With intuitive examples from a simple workflow schema that can be defined in many workflow systems, we illustrated some heuristics of applying the rewrite rules in optimizing execution plans. We also presented the main components of a model of physical execution plans, along a preliminary analysis of key cost factors.

The technical results reported in this paper are preliminary; there are many interesting questions that deserve further investigation, ranging from theoretical foundation to practical application. In one extreme, there are many decision problems arising from this, such as: Is there a (sound and) complete set of rewrite rules? What is the complexity of testing if a set of CFCs have deterministic behavior? In the other extreme, it is fundamental to investigate further the issue of cost models, and heuristics for identifying efficient physical execution plans. It is also unclear how

this kind of compile-time optimization techniques compares with the ones based on adaptive scheduling such as [HLS+99b,HKL+99].

Acknowledgment Part of work by Jianwen Su was supported by NSF grants IRI-9411330, IRI-9700370, and IIS-9817432.

References

[AMG+95] G. Alonso, C. Mohan, R. Gunther, D. Agrawal, A. El Abbadi, & M. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. IFIP WG8.1 Working Conf. on Information Systems for Decentralized Organizations*, 1995.

[BMR96] D. Barbara, S. Mehrotra, & M. Rusinkiewicz. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management, Special Issue on Multidatabases*, 7(1), 1996.

[DKM+96] S. Das, K. Kochut, J. Miller, A. Sheth, & D. Worah. Orbwork: A reliable distributed corba-based workflow enactment system for meteor$_2$. Technical Report UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, 1996.

[Ell79] C. A. Ellis. Information control nets: A mathematical model of office information flow. In *ACM Proc. Conf. Simulation, Modeling & Measurement of Computer Systems*, pp225–240, Aug 1979.

[GHS95] D. Georgakopoulos, M. Hornick, & A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed & Parallel Databases*, 3(22):119–154, April 1995.

[HKL+99] R. Hull, B. Kumar, F. Llirbat, G. Zhou, G. Dong, & J. Su. Optimization techniques for data-intensive decision flows. Technical report, Bell Laboratories, Lucent Technologies, 1999. see http://www-db.research.bell-labs.com/projects/vortex.

[HLS+99a] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, & G. Zhou. Declarative workflows that support easy modification & dynamic browsing. In *Proc. of Intl. Joint Conf. on Work Activities Coordination & Collaboration (WACC)*, pp69–78, Feb 1999.

[HLS+99b] R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, & G. Zhou. Adaptive execution of workflow: Analysis & optimization. Technical report, Bell Laboratories, Lucent Technologies, 1999. see http://www-db.research.bell-labs.com/projects/vortex.

[KS95] N. Krishnakumar & A. Sheth. Managing heterogeneous multi-systems tasks to support enterprise-wide operations. *Distributed & Parallel Databases*, 3(2), 1995.

[LR94] F. Leymann & D. Roller. Business process management with FlowMark. In *Proc. of IEEE Computer Conference*, pp230–234, 1994.

[Mos] Mosaix, Incorporated. 1999. http://www.aiim.org/wfmc.

[PYLS96] P. Piatko, R. Yangarber, D. Lin, & D. Shasha. Thinksheet: A tool for tailoring complex documents. In *Proc. ACM SIGMOD*, page 546, 1996.

[WfM99] Workflow management coalition, 1999. http://www.aiim.org/wfmc.

[WW97] D. Wodtke & G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proc. of Intl. Conf. on Database Theory*, pages 230–246, 1997.

[WWWD96] D. Wodtke, J. Weissenfels, G. Weikum, & A. K. Dittrich. The Mentor project: Steps towards enterprise-wide workflow management. In *Proc. of IEEE Intl. Conf. on Data Engineering*, New Orleans, 1996.