Workflow Parallelization by Data Partition and Pipelining

Marco Antonio Casanova, Melissa Lemos

Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de S. Vicente, 225 – Rio de Janeiro, Brazil CEP 22453-900
{melissa, casanova}@inf.puc-rio.br

Abstract. This paper first introduces a workflow parallelization strategy, based on a set of clearly defined heuristics, including data partition and a variation of pipelining. Then, the paper outlines an architecture for a workflow manager, based on Web services, that incorporates the heuristics. Finally, the paper reports on the current status of the implementation of the workflow manager.

Keywords: Workflow optimization, data partition, pipelining, Web services

1. Introduction

Computer-intensive scientific investigation often involves a variety of analysis processes and large datasets. The analysis processes are typically combined, in the sense that the datasets a process produces are used as input to other processes. This form of process composition is best modeled as a workflow (Yu and Buyya 2005).

Given the sheer size of the datasets and the complexity of the analysis processes, workflow execution often requires careful optimization, especially in a distributed environment. However, it is unrealistic to assume that the domain experts, who specify the workflows, have sufficient expertise about the computing platform to carry on this optimization step. With this motivating background, we introduce a workflow parallelization strategy, based on data partition and pipelining, which depend only on indicating how application programs read their input datasets. We then outline an architecture for a workflow manager, based on Web services, that incorporates the heuristics. Finally, we remark on the current status of the implementation of the workflow manager.

The major contributions of the paper lie in the precise definition of data partition and of pipelining, and on the clear formulation of heuristics that incorporate such transformations and, yet, lead to the implementation of a viable workflow manager. The workflow parallelization strategy contributes to the construction of workflow management systems, or Grid infrastructure middleware. Examples of such systems from e-Science are Kepler (Altintas et al. 2004) and myGrid (Wroe et al. 2004), to name just two. Bent (2005) proposes a data-driven batch scheduling system that allows a careful coordination of both data and CPU allocation. In spite of their usefulness, these systems do not automatically optimize workflow execution. The

architecture we propose leverages on the ability of service domains (Tan et al. 2006) to select and monitor the appropriate Web service instances. Examples of strategies for dynamic Web service selection are reported, for example, in (Hu et al. 2006; Huang et al. 2005; Maximilien et al. 2004; Zhang 2005). The proposed architecture is consistent with the reference model proposed by the WfMC (Hollingsworth 1995).

This paper is organized as follows. Section 2 describes the workflow model. Section 3 introduces the workflow parallelization strategy. Section 4 discusses an architecture based on Web services. Section 5 addresses an implementation following the architecture. Finally, Section 6 contains the conclusions.

2. Workflow Model

The workflow model combines two sub-models. The *application model* captures the characteristics of the application programs that our workflow parallelization strategy requires, and that must thereby be specified by the domain expert. The *flow model* introduces a simple data flow abstraction [van der Aalst et al. 2000].

An *application model* is defined as a set of application programs. An *application program* $\boldsymbol{p}$ is defined by a list of *parameters*, *input ports*, and *output ports*, as follows:

- *parameters*: a list of pairs $\boldsymbol{p_k} = (pn_k, pt_k)$, for $k \in [1,r]$, that define the name pn_k and the type pt_k of each parameter $\boldsymbol{p_k}$ of $\boldsymbol{p}$
- *input ports*: a list of pairs $\boldsymbol{i_k} = (in_k, it_k)$, for $k \in [1,m]$, that define the name in_k of each *input port* $\boldsymbol{i_k}$ from which $\boldsymbol{p}$ reads a set of data items of type it_k
- *output ports*: a list of pairs $\boldsymbol{o_k} = (on_k, ot_k)$, for $k \in [1,n]$, that define the name on_k of each *output port* $\boldsymbol{o_k}$ to which $\boldsymbol{p}$ writes a set of data items of type ot_k

We use *name[f]* and *type[f]* to indicate the name and the type of a parameter, an input port, or an output port $\boldsymbol{f}$ of $\boldsymbol{p}$.

An instance of $\boldsymbol{p}$ models a call to $\boldsymbol{p}$, and is denoted by an expression of the form $\boldsymbol{p}(v_1,...,v_r,F_1,...,F_m) = (G_1,...,G_n)$, where $v_1,...,v_r$ is a list of *parameter values* such that v_k is of type pt_k, for $k \in [1,r]$, $F_1,...,F_m$ is a list of *input datasets* such that F_k contains data items of type it_k, for $k \in [1,m]$, and $G_1,...,G_n$ is the list of *output datasets* that the instance creates (G_k therefore contains data items of type ot_k, for $k \in [1,n]$).

We say that $\boldsymbol{p}$ reads from an input port $\boldsymbol{i_h}$ *locally* iff the dataset that any instance of $\boldsymbol{p}$ reads through $\boldsymbol{i_h}$ must be stored in the same processor as the instance executes; otherwise, we say that $\boldsymbol{p}$ reads from the input port $\boldsymbol{i_h}$ *globally*.

We say that $\boldsymbol{p}$ reads from an input port $\boldsymbol{i_h}$ *gradually* iff, for any instance $\boldsymbol{p}(v_1,...,v_r,F_1,...,F_h,...,F_m)=(G_1,...,G_n)$ of $\boldsymbol{p}$, for any partition of F_h into d disjoint subsets $F_{h1},...,F_{hd}$, we have that, if $\boldsymbol{p}(v_1,...,v_r,F_1,...,F_{hj},...,F_m)=(G_{j1},...,G_{jn})$ is the instance of $\boldsymbol{p}$ obtained by replacing F_h by F_{hj}, for each $j \in [1,d]$, then $G_j = G_{j1} \cup ... \cup G_{jd}$, for each $j \in [1,d]$. Intuitively, we can compute $\boldsymbol{p}(v_1,...,v_r,F_1,...,F_h,...,F_m)=(G_1,...,G_n)$ by partitioning F_h, applying $\boldsymbol{p}$ to each fragment of F_h in parallel, maintaining the parameter values and the rest of the input datasets unchanged, and then combining the partial output datasets obtained.

A *process* is an instance of an application program. When the parameter values, the input and the output datasets do not matter, we denote a process in lower case italics.

A *container* is a dataset that one or more processes read or write through input or output ports. The *flow model* introduces a simple data flow abstraction, called *workflow graphs*, or simply *workflows*, defined as labeled bipartite graphs such that (see Figures 1 and 2 for examples):

- the set of nodes consists of processes or containers of the application model, called *process nodes* and *container nodes*, respectively
- the set of arcs contains pairs of the form *(c,p)* or *(p,c)*, where *p* is a process node and *c* is a container node. Each arc has two labels, *name* and *type*. The set of arcs and their labels is such that, for each process node *p* which is an instance of an application program ***p***:
  - for each input port ***i*** of ***p***, there must be an arc *(c,p)* from a container node *c* to *p* such that *name((c,p)) = name[**i**]* and *type((c,p))= 'gradual'* iff ***p*** reads from the ***i*** gradually, and *type((c,p))= 'non-gradual'*, otherwise
  - for each output port ***o*** of ***p***, there must be an arc *(p,c)* from *p* to a container node *c* such that *name((p,c)) = name[**o**]* and *type((p,c))* is undefined
  - these are the only arcs of the workflow graph.

We assume that the workflow graph has two properties: (1) the graph is not a multi-graph, that is, all arcs are distinct; and (2) the graph is acyclic. Assumption (1) avoids notational complexities and can be easily relaxed. It implies that a process cannot read (or write) from a container through more than one connection. Assumption (2) simplifies the discussion in Sections 3 and 4. Note that a process, with its input and output datasets, naturally is a special case of a workflow.

Let *(c,p)* be an arc from a container node *c* to a process node *p*. Assume that *name((c,p))=name[**i**]* and that *p* is an *instance* of ***p***. Then, we say that the arc is a *gradual read connection* and that *p gradually reads from c* iff ***p*** reads from the input port ***i*** gradually.

We say that *c* is an *input container* of the workflow graph iff *c* is not the destination of any arc, and that *c* is an *output container* of the workflow graph iff *c* is not the origin of any arc.

Finally, we define that two workflows, *w* and *v*, are *equivalent* iff there is an one-to-one mapping μ_I from the input containers of *w* into those of *v*, and an one-to-one mapping μ_O from the output containers of *w* into those of *v*, such that: (1) for each input container *c* of *w*, if $\mu_I(c)=d$ then $c=d$; and (2) for each output container node *e* of *w*, if $\mu_O(e)=f$ then $e=f$.

3. Workflow Parallelization

In this section, we first define two transformations that map workflows into equivalent workflows, that we call data partition and pipelining. Then, we introduce a workflow parallelization strategy that includes such transformations.

Data partition, denoted by δ, transforms workflows by replacing a process by a parallel composition of replicas of the process. More precisely, let *w* be a workflow and *p* be a process in *w*. Assume that at least one read connection of *p* is gradual. Let *c* be a container that *p* gradually reads from, and let *C* be the set of the other input

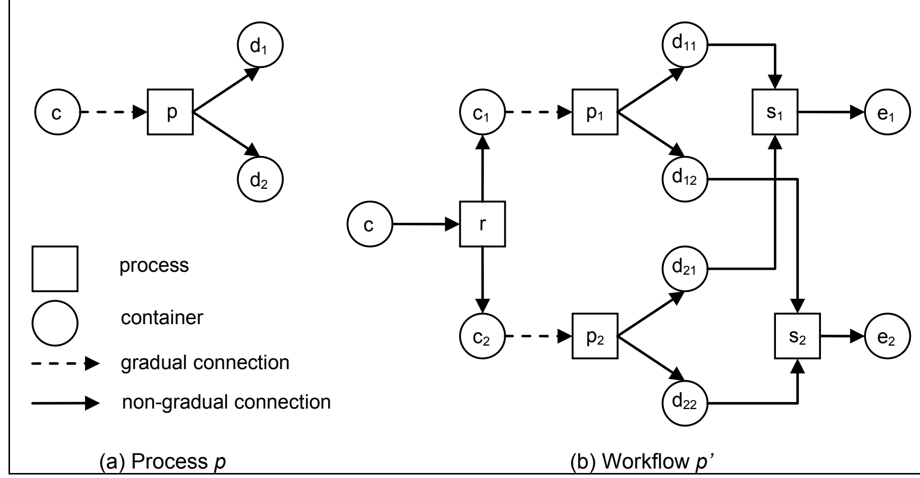

Fig. 1 Example of process parallelization by data partition.

containers of p. Let $D=\{d_1,...,d_m\}$ be the set of output containers of p. Then, data partition transforms w into a new workflow $\delta(w)$ by replacing p by the workflow p' (shown in Figure 1(b), where $m=n=2$ and C is the empty set, for simplicity), where

- process r partitions container c into containers $c_1,...,c_n$
- for each $i\in[1,n]$, process p_i is a replica of p that reads container c_i instead of c, creating output containers $d_{i1},...,d_{im}$
- for each $j\in[1,m]$, process s_j combines $d_{1j},...,d_{nj}$ into an output container e_j

Since p gradually reads from c, the execution of p and p' are equivalent, in the sense that containers d_i and e_i will have the same set of data items, for each $i\in[1,m]$. Indeed, this is a direct consequence of the definition of gradual read connection. Hence, w and $\delta(w)$ are also equivalent.

Define a cost function σ as the overall execution time of a workflow. Then, data partition is advantageous if the gain obtained by parallelizing p is not offset by the overhead of partitioning c and combining the partial output containers.

We define three variations of pipelining. *Pairwise pipelining*, denoted by π, is a transformation that parallelizes a pair of processes, working in combination with data partition. More precisely, let w be a workflow, and p and q be two processes in a w. Assume that: (1) data partition applies to p; (2) q gradually reads from an output container d_h of p; and (3) q does not read from any other output container of p.

Suppose that data partition is applied to w with respect to p. Let $\delta(w)$ be the new workflow obtained. As in the definition of data partition, let p' be the workflow that replaces p, for each $i\in[1,n]$, let p_i be the replica of p that creates output containers $d_{i1},...,d_{im}$ and, for each $j\in[1,m]$, let s_j be the process that combines $d_{1j},...,d_{nj}$ into an output container e_j.

Pairwise pipelining transforms $\delta(w)$ into a new workflow $\pi(\delta(w))$ as follows (shown in Figure 2(b), where $m=n=2$, for simplicity):

- drop container e_h and process s_h, if q is the only process that reads from d_h in w and no other process in w writes into d_h
- replace q by replicas $q_1,...,q_n$ of q such that q_k reads from d_{kh}, for each $k \in [1,n]$
- introduce processes that combine the corresponding output containers of $q_1,...,q_n$ into single output containers

As for data partition, since q gradually reads from d_h, workflows w and $\pi(\delta(w))$ are equivalent. The second assumption for applying pairwise pipelining is justified as follows. Suppose that it does not hold. Then, q reads from another output container e_g that p writes. Therefore, $q_1,...,q_n$ would only start after all processes $p_1,...,p_n$ finish and process s_g combines $d_{1g},...,d_{ng}$ into e_g. This is not consistent with the intention of pipelining, which is to execute each pair of processes p_k and q_k independently of the other pairs, for $k \in [1,n]$. Hence, the second assumption is justified.

We define *flexible pairwise pipelining* exactly as pairwise pipelining, except that the number of replicas of q need not be the same as that of p. This means that some fragments of the output container d_h may be combined or further partitioned before being passed to replicas of q.

Lastly, *full pipelining*, or simply *pipelining*, is defined as the repeated application of flexible pairwise pipelining to a sequence of processes such that one process gradually reads from a container that the previous one writes. We call such sequence a *process pipe*. Note that, to define full pipelining, we have to consider the case where p is parallelized as a result of data partition or flexible pairwise pipelining.

Figure 3a shows a workflow w, and Figure 3b exhibits workflow v, a possible result of parallelizing w, where the boxes labeled with X and + denote container partition and container combination, respectively. The workflow in Figure 3b is obtained by: (1) applying data partition to parallelize p_1 into two replicas; (2) applying pipelining to p_2 and p_3, recreating two process pipes; (3) applying data partition to the replica of p_2 in one process pipe, and to the replica of p_3 in the other process pipe.

Assume that the cost function is again the overall execution time. The goal of a workflow optimizer based on these transformations would therefore be: Given any

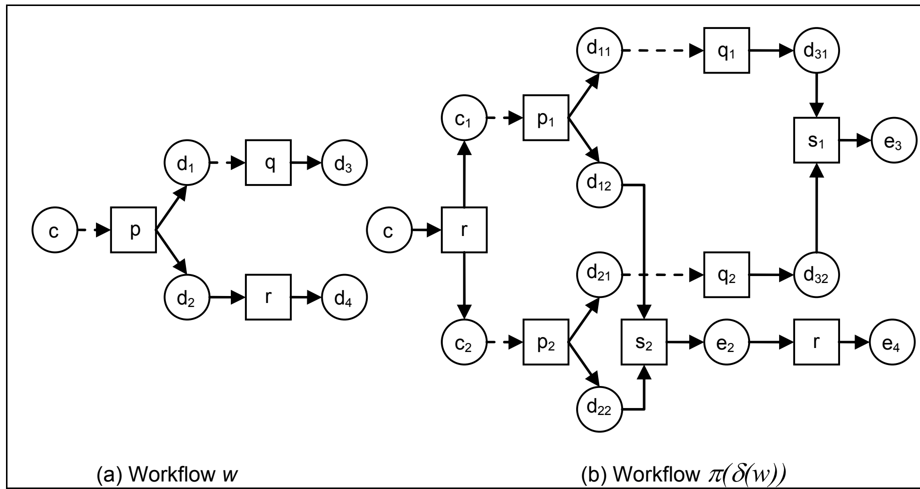

Fig. 2 Example of process parallelization by pipelining.

workflow *w*, transform *w* into a new workflow *v* by repeatedly applying data partition and pipelining such that *v* has the least cost among the transformations of *w*.

We first observe that finding the parallelized workflow that has the least cost is impractical. The complexity steams from the fact that data partition and pipelining interact, since pipelining avoids combining partial output containers, reducing the cost of data partition, while data partition generates process pipes. In fact, in (Lemos and Casanova 2006), we showed that a simpler problem is already NP-Complete.

Therefore, we proceed to describe heuristics that lead to a practical workflow optimizer. The heuristics fall into two categories. In this section, we discuss only those that depend on the characteristics of the original workflow and of the parallelized workflow obtained thus far. In Section 4, we refine the heuristics to take into account characteristics of the target computational environment.

Let *w* be the workflow currently being optimized. The first heuristics restricts the optimizer to make local decisions only, in the following sense:

Heuristics 1: When applying data partition, select the number of fragments to partition a container based only on the characteristics of the container and of the process that (gradually) reads it.

Now observe that, if a process *p* gradually reads from more than one container, the optimizer may partition any such input containers. But this would create an exponential number of replicas of *p* to handle all possible combinations of the input fragments from distinct containers. Therefore, we introduce a second heuristics:

Heuristics 2: If a process gradually reads from more than one container, select only one such container to partition.

Furthermore, the choice of which container to partition is not straightforward. The third heuristics tries to increase the gains obtained by choosing the largest container.

Heuristics 3: Among the containers that a process gradually reads, give preference to partition the largest container.

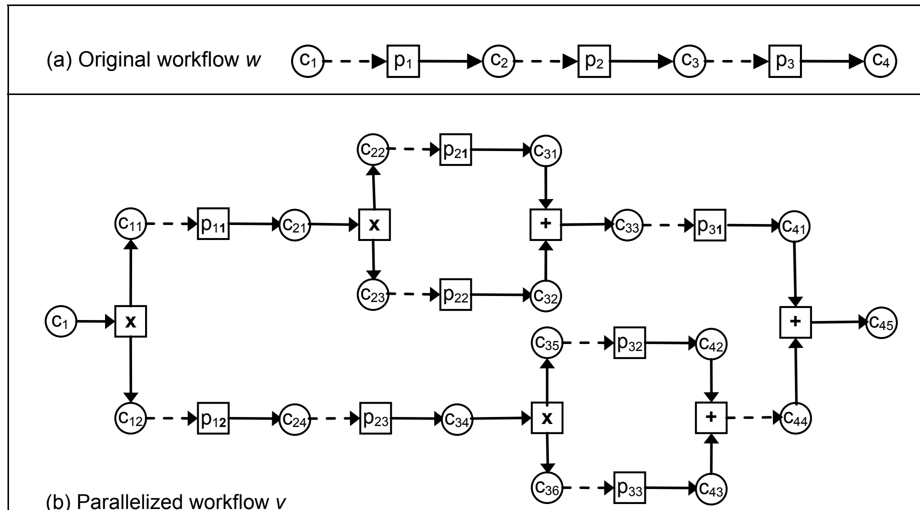

Fig. 3 Example of workflow parallelization.

The last heuristics reduces the search space by applying pipelining immediately after data partition or, recursively, pipelining.

Heuristics 4: Start the parallelization of the workflow by trying to apply data partition to the processes that read from the input containers of the workflow. At any stage of the parallelization of the workflow, if *p* is a process to which data partition or pipelining was applied in the previous stage, then try to apply pipelining to any process *q* that gradually reads from an output container that *p* writes.

4. An Architecture based on Web Services

In this section, we briefly introduce an architecture for a workflow manager, based on Web services, that reflect the heuristics outlined in Section 3. Figure 5 shows the top level components, grouped into the *workflow domain* and one or more *service domains*.

The *workflow assistant* helps the user create a workflow, as discussed in (Lemos et al. 2004).

The *workflow controller* governs workflow execution, maintaining a list of the processes that are ready to be executed, among other data, in a workflow execution *state*. It features a registration service that keeps a description of the services each service domains controls. For each process *p* in a workflow *w*, if *p* is ready to be executed, the controller selects a service domain *SD* to execute *p*, creates a process execution request for *p* (see below), and sends the request to *SD*.

The workflow controller is responsible for implementing pipelining (part of Heuristics 4). It signals to the service hub that no pipelining will apply to *p* by sending a *complete* process execution request, that is, a request such that all input containers are completely available; it signals that pipelining may apply to *p* by sending a *partial* request, that is, a request such that only fragments of an input container are available.

A *service hub* receives process execution requests from the workflow controller. It features a registration service that keeps a description of the service desks that pertain to the service domain, containing sufficient information to permit the hub to select the appropriate service desk to relay each process execution request. It therefore provides one level of indirection between the workflow controller and the service desks.

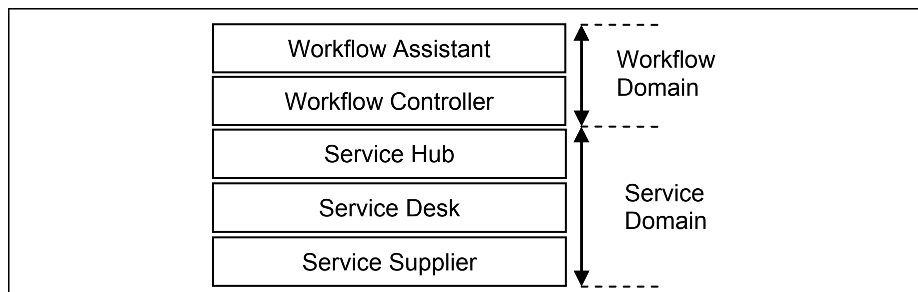

Fig. 5 Architecture.

A *service desk* maps a process execution request into one or more Web service invocations, sent to the service suppliers. The service desk is at the heart of the optimization process. It uses information about the computational environment and information contained in the process execution request to decide whether the process execution will be parallelized or not, and which input containers will be partitioned. The service desk therefore implements Heuristics 1 since it makes entirely local decisions about how to parallelize a process execution request.

A *service supplier* simply implements one or more Web services.

The rest of this section outlines how the workflow controller and the service hubs interact and how the service desk decides whether a process execution request will be parallelized or not.

The workflow controller invokes a service hub by sending a *process execution request*, containing

- the description of the process: the name, parameters, input and output containers of the process;
- the characteristics of each input connection: whether it is gradual or not, and whether it is local or global;
- the characteristics of each output connection: whether the corresponding output container can be passed back to the workflow controller fragment by fragment, or not, and whether the connection is local or global;
- the characteristics of each input container: whether the request corresponds to the complete container, to a full partition of the complete container, or to just a fragment of the container; in the last case, whether the fragment is the first one, the last one, or just an intermediate fragment (required to implement pipelining).

A process execution request generates one or more replies. Each reply carries information about a complete output container, or about a fragment of an output container. A reply is also generated to inform the workflow controller whether the process will suffer data partition, or not. Upon receiving a reply, the workflow controller updates the ready process list. We note again that, to implement pipelining, the controller may pass fragments created by one process to the next, which affects how it updates the ready process list.

Service desks are responsible for implementing data partition. Upon receiving a process execution request, a service desk decides to apply data partition by considering, first of all, the characteristics of the input containers, as discussed in Section 3. If the request is the first one for the process, the service desk has to decide which input container, if any, will be partitioned (by Heuristics 2, only one container will be partitioned). It therefore tries to apply Heuristics 3 based on the information obtained thus far about container sizes.

A service desk handles (the crucial tasks of) container partition, container recombination, and local/global container access.

It may generate internal processes to partition an input container (or a fragment of an input container), combine fragments of an input container, or combine fragments of output containers into a complete container. Furthermore, the service desk must guarantee that globally accessed containers are already allocated to some processor,

and that locally accessed containers are allocated in the same processor as the Web services that will implement the process whose execution was requested.

A service desk may partly implement flexible pairwise pipelining by waiting until it receives several fragments of an input container, sent through various requests for the same process.

Finally, a service desk decides the number of Web services that will execute the process in parallel based on the (current) occupation of the service suppliers under its jurisdiction. This is in fact why most of the optimization decisions were delayed to runtime and delegated to the service desk.

5. Remarks on the Implementation

In this section, we report on the status of the implementation of a workflow manager that follows the architecture outlined in Section 4.

The service domain component is fully implemented (Rosa 2006; Lemos at al. 2007). Its interface supports process execution requests, as described in Section 4. Internally, it incorporates processor allocation strategies, such as randomization or round-robin, and is prepared to incorporate new strategies.

The component was implemented as a framework whose hotspots are the *ServiceHub, ServiceDesk* and *ServiceSupplier* abstract classes. As a proof of concept, the framework was instantiated for a suite of Bioinformatics applications (Kanehisa and Bork 2003). For each application supported, instantiations of *ServiceDesk* and *ServiceSupplier* were implemented. For example, to support the BLAST application (Altschul 1990), classes *BlastServiceDesk* and *BLASTServiceSupplier* were implemented. However, there is a single *BioServiceHub* class, which must be adapted only when new programs are incorporated. The instantiation is called *BioService domain*.

To independently test the BioService domain, the Taverna software tool (Taverna 2006), developed under the myGrid Project (Stevens et al. 2003), was used to create and execute workflows over the service domain.

In parallel, a workflow controller was implemented that incorporates most of the features described in Section 4. In particular, it supports a variation of pipelining that passes a container fragment from one process to the next (Silva 2006).

6. Conclusion

We first described a workflow parallelization strategy, based on data partition and pipelining, and introduced simple heuristics that lead to a viable implementation of a workflow manager. The strategy depends only on indicating which datasets an application program reads gradually, thereby avoiding a detailed analysis of the application semantics. We then outlined an architecture for a workflow manager, based on Web services, that incorporates the heuristics. Finally, we remarked on the current status of the implementation of the workflow manager. We refer the reader to

(Lemos et al. 2007) for performance data obtained by running sample Bioinformatics workflows, as well as implementation details, which we omit here for brevity.

Acknowledgments. This work was partially supported by FAPERJ for Melissa Lemos and by CNPq under grant 550250/2005-0 for Marco Antonio Casanova. The authors wish to thank Rodrigo Rosa and Rafael Silva for their help with the implementation of the service domain framework and the workflow controller.

References

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B., Mock S. (2004). “Kepler: An Extensible System for Design and Execution of Management. Scientific Workflows”, Proc. 16th Int’l Conf. on Scientific and Statistical Database.

Altschul S.F.; Gish W.; Miller W.; Myers E.W.; Lipman D.J. (1990). “A basic local alignment search tool”, J. of Molecular Biology, 215 (3), pp. 403–410.

Bent, J. (2005) *Data-Driven Batch Scheduling*. Ph.D. dissertation. Univ. Wisconsin-Madison.

Hollingsworth, D. (1995) *The Workflow Reference Model.* Workflow Management Coalition. Document Number TC00-1003, Issue 1.1, 19-Jan-95.

Hu, J.; Guo, C.; Wang, H.; Zou, P. (2005) Quality Driven Web Services Selection. In: Proc. IEEE International Conf. on e-Business Engineering (ICEBE'05), pp. 681–688.

Huang, L.; Walker, D.W.; Huang, Y.; Rana, O.F. (2005) Dynamic Web Service Selection for Workflow Optimisation. In: Proc. 4th UK e-Science Programme All Hands Meeting (AHM), Nottingham, UK.

Kanehisa, M.; Bork, P. (2003) “Bioinformatics in the post-sequence era”. Nature Genetics 33, pp. 305–310.

Lemos, M.; Casanova, M.A.; Seibel, L.F.B.; Macedo, J.A.; Basílio, A. (2004) Ontology-Driven Workflow Management for Biosequence Processing Systems. In: Proc. 15th Int. Conf. on Database and Expert Systems Applications - DEXA '04, Zaragosa, Spain.

Lemos, M.; Casanova, M.A. (2006) On the Complexity of Process Pipeline Scheduling In: Proc. XXI Brazilian Symposium on Data Bases, Florianópolis, Brazil.

Lemos, M.; Casanova, M.A.; Rosa, R.; Silva, R. (2007) *Workflow Parallelization by Data Partition and Pipelining.* Technical Report MCC 07-19. Dept. of Informatics, PUC-Rio.

Maximillien, E.M.; Singh, M.P. (2004) A Framework and Ontology for Dynamic Web Services Selection, IEEE Internet Computing.

Rosa, R.A.P. (2006) “Rhodnius Prolixus Workflow Parallelization through a Service Domain Architecture”. Final Graduation Project, Dept. Informatics, PUC-Rio (in Portuguese).

Stevens, R.D.; Robinson, A.J.; Goble, C.A. (2003) myGrid: personalised bioinformatics on the information grid, *Bioinformatics*, 19, pp. 302–304.

Taverna Project Website (2006) http://taverna.sourceforge.net/

Tan, Y.; Topol, B.; Vellanki, V.; Xing, J. (2004) Business service Grid, Part 1: Introduction. Manage Web services and Grid services with Service Domain technology. http://www-128.ibm.com/developerworks/library/gr-servicegrcol.html

Wroe, C., Goble, C., Greenwood, M., Lord, P., Miles, S., Papay, J., Payne, T., Moreau, L. (2004). “Automating Experiments Using Semantic Data on a Bioinformatics Grid”, IEEE Intelligent Systems Special Issue on e-Science, 19 (1), pp. 48-55.

Yu, J.; Buyya, R. (2005) A taxonomy of scientific workflow systems for grid computing. ACM SIGMOD Record, Vol. 34, Issue 3. Special section on scientific workflows, pp. 44–49.

Zhang, D. (2005) Implementation of a Resource Broker Prototype within a Service-Oriented Architecture. M.Sc. Dissertation, Dept. Computer Science, Univ. Adelaide.