

Process Pipeline Scheduling

Melissa Lemos, Marco A. Casanova, Antonio L. Furtado

Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de S. Vicente, 225 – Rio de Janeiro, RJ – Brazil – CEP 22453-900

{melissa, casanova, furtado}@inf.puc-rio.br

***Abstract.** Two processes, p and q , may be scheduled in pipeline when q may start when p starts, and q may process data items from p , one-by-one, without waiting for p to write the complete set of data items. This paper explores how process pipeline scheduling may become a viable strategy for executing workflows. The paper first details a workflow model that captures the characteristics of the application programs that pipeline scheduling requires. It proceeds by showing that the process pipeline scheduling problem is NP-Complete. Then, it describes a specific algorithm that pipelines as many processes as possible, within the bounds of the storage space available, based on a greedy process scheduling heuristics that has acceptable performance. Finally, the paper presents a detailed example that illustrates how the algorithm schedules processes.*

1. Introduction

Computer-intensive scientific investigation often involves applying off-the-shelf analysis processes to existing datasets. Furthermore, researchers typically combine analysis processes, in the sense that the dataset one process produces is used as input to another process. This form of process composition is best modeled as a workflow [Yu and Buyya 2005]. Digital document publishing and other more familiar application areas offer similar examples of process compositions. Turning to traditional database technology, query plans can also be interpreted as workflows, which the query optimizer generates without user intervention [Garcia-Molina 1999].

We address in this paper *process pipeline scheduling*, a strategy to execute workflows that optimizes runtime storage requirements and may reduce overall execution time. Very briefly, suppose that a workflow contains two processes, p and q , such that p writes a set of data items that q will read. Then, depending on the semantics of the processes, q may read a data item from p as soon as p outputs it. Hence, if this condition is true, the process scheduling strategy may start q when it starts p , since q does not have to wait for p to write the complete set of data items to start processing them. If this condition is false, then the strategy will only start q after p finishes. This process pipeline scheduling strategy can be applied to more than two processes, but it is limited by the amount of resources available, such as disk space.

Pipeline was voted one of the ten most influential parallel and distributed processing concepts of the last millennium [Theysa et al. 2001]. It is extensively used in database query optimization to avoid the materialization of intermediate results [Garcia-Molina 1999]. In this context, pipelining is possible for two very simple reasons: (1) relations are sets of tuples; (2) the projection, selection and join operations can be applied one

tuple (or pair of tuples) at a time. In this paper, we extend this idea to workflows, exploring when application programs exhibit a behavior similar to the above-mentioned relational algebra operations.

The contributions of this paper lie in exploring how process pipeline scheduling may become a viable strategy for executing workflows. We start by describing a workflow model that captures the characteristics of the application programs that pipeline scheduling requires. We proceed by showing that the process pipeline scheduling problem is NP-Complete. Then, we describe a specific algorithm that pipelines as many processes as possible, within the bounds of the storage space available, based on a greedy process scheduling heuristics that has acceptable performance. Finally, we present a detailed example that illustrates how the algorithm schedules processes.

A detailed discussion about related work is postponed to the end of the paper.

Two implementation efforts were carried out to test the ideas described in this paper: a workflow controller, based on the process pipelining scheduling algorithm described in this paper [Silva 2006], and a workflow execution engine, based on a service domain architecture [Rosa 2006]. To help the reader assess the contributions of the paper, a brief summary of these implementation efforts is also included in Section 5.

The paper is organized as follows. Section 2 provides motivations for investigating process pipeline scheduling. Section 3 describes the workflow model adopted in the paper. Section 4 discusses how to implement the data structures that process pipeline scheduling uses. Section 5 describes in detail the process pipeline scheduling algorithm proposed in the paper, and discusses the complexity of the problem. Section 5 also summarizes the current implementation efforts. Section 6 illustrates how the algorithm works with an example. Section 7 discusses related work. Section 8 contains the conclusions. Appendix 1 presents a complete proof that the process pipeline scheduling problem is NP-Complete.

2. Motivation

The process pipeline scheduling strategy proposed in this paper is not specific to a particular application area, but it rather depends on generic properties of datasets and processes, which we capture in an appropriate workflow model, described in Section 3. We motivate the interest in process pipeline scheduling with examples from three application areas: Bioinformatics, Geographic Information Systems and Digital Document Publishing. The examples have common properties that we list at the end of this section.

Genome projects usually start with a sequencing phase, where experimental data, called *chromatograms*, are generated in laboratory, without any biological interpretation. The fundamental challenge researchers face lies in analyzing these sequences to derive information that is biologically relevant. There are several Bioinformatics programs which help researchers analyze their experimental data, such as *Phred*, *Phrap*, *Glimmer*, *Transeq* and *BLASTP*, used in the example that follows, whose exact description need not concern us here.

Consider the following simple workflow, typical of Bioinformatics:

1. For each chromatogram produced in laboratory, execute *Phred* to generate a sequence, called a *read*. This step creates a set *R* of *reads*.
2. Execute *Phrap* over all *reads* in *R* to produce a set *C* of sequences, called *contigs*.
3. For each *contig* in *C*, execute *Glimmer* to identify putative genes (a nucleotide sequence also called an *ORF - Open Read Frame*). This step creates a set *O* of ORFs.
4. For each ORF in *O*, execute *Transeq* to convert the ORF into an amino acid sequence. This step creates a set *A* of amino acid sequences.
5. For each sequence *s* in *A*, execute *BLASTP* to compare *s* with the public data source NR, which is a protein sequence database available at the NCBI site [NCBI 2006]. *BLASTP* returns sequences that are similar to *s*, together with their annotations stored at NCBI. These annotations will then help the researcher interpret *s*.

Since Steps 3, 4 and 5 operate on one object at a time, they can be pipelined, making partial results available to the user sooner than otherwise. Therefore, pipelining also helps users to better monitor workflow execution by observing partial results. This is useful in Bioinformatics because there are cases where it becomes important to prematurely stop execution and re-define the parameters of the programs.

As an example from the area of Geographic Information Systems, consider an environmental protection agency that decides to closely monitor an area for potentially illegal activities, such as forest burning, illegal mining, etc... The agency might set the following (highly simplified) workflow:

1. Search a remote sensing image repository for a set *R* of images such that: *R* covers the area of interest; each image in *R* has been acquired within the last few days; and each image in *R* has small cloud coverage.
2. For each image in *R*, pre-process the image (i.e., register and segment the image), creating a new set *S*.
3. For each pre-processed image in *S*, analyze the image for potential illegal activities, creating a list of (geo-referenced) areas and the potential threats to the environment.
4. Send a report to the law enforcement teams indicating the areas and the threats.

Note that, in the above workflow, the images one step creates can be pipelined to the next, and partial reports can be sent to the law enforcement teams. Indeed, if timely information is crucial, as in this application, pipelining is useful for a second reason, since it permits publishing partial results to the user sooner than otherwise, similarly to the Bioinformatics example. Furthermore, depending on the resources available, including human analysts, the processing of images in steps 2 and 3 can be parallelized.

However, pipelining is not always readily applicable. Consider the following alternative workflow:

1. Search a remote sensing image repository for a set R of images such that: R covers the area; each image in R has been acquired within the last few days; and each image in R has small cloud coverage.
2. Create a single image m by mosaicking the images in R .
3. Pre-process m , creating a new image p .
4. Analyze p for potential illegal activities, creating a list of (geo-referenced) areas and the potential threats to the environment.
5. Send a report to the law enforcement teams indicating the areas and the threats.

This new workflow does not offer opportunities for pipelining and parallelization since Steps 2, 3 and 4 operate on a single image, which potentially delays sending information to the law enforcement teams. To partly regain the benefits of the previous workflow, it suffices to partition the mosaic m , or the pre-processed image p , into tiles (say $10\text{km}\times 10\text{km}$). This means introducing a new step after Steps 2 or 3 to create the tiles and then pipelining the tiles to Step 4.

Finally, consider a third example from the area of Digital Document Publishing. The scenario is that of a company that has a collection of documents in XML to be published on a periodic basis, according to the following workflow:

1. For each document d in the input collection, expand d with any included document, creating a new document set E .
2. For each expanded document e in E , validate e using a local XML schema. If e is invalid, drop e from E and issue an error message.
3. For each remaining document f in E , transform and publish f using a specific set of stylesheets.

This example requires no additional comments.

These examples share in common the fact that each step corresponds to an operation f that maps sets of objects into sets of objects and is defined with the help of a second function g such that $f(S) = \{ g(s) / s \in S \}$. Then, given two such operations f_p and f_q , we trivially have that:

$$(1) \quad f_p(f_q(S)) = \{ g_p(g_q(s)) / s \in S \}$$

$$(2) \quad f_p(f_q(U \cup V)) = f_p(f_q(U)) \cup f_p(f_q(V)) = f_p(f_q(U)) \cup f_p(f_q(V))$$

The term *pipelining* is usually applied to name the strategy of computing $f_p(f_q(S))$ by using (1). That is, instead of computing $T=f_q(S)$ and then computing $U=f_p(T)$, U is incrementally created by computing $t=g_q(s)$ and then $g_p(t)$, for each $s \in S$. The term *data parallelization* is applied to name the strategy of computing $f_p(f_q(S))$ by partitioning S into smaller sets and then using (2). Of course, the two can be combined into a single strategy, where the level of parallelization depends on the amount of resources available to compute f_p and f_q . The rest of this paper generalizes these observations by carefully identifying when pipelining applies.

3. Workflow Model

The workflow model combines two sub-models. The *application model* captures the characteristics of the application programs that process pipelining scheduling 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], and assumes a given application model.

3.1 Application Model

An *application model* is defined as a set of application programs. The domain expert specifies an *application program* \mathbf{p} by defining its *parameters*, *input ports*, and *output ports*, as follows:

- *parameters*: a list of pairs $\mathbf{p}_k=(pn_k,pt_k)$, for $k \in [1,r]$, that defines the name pn_k and the type pt_k of each parameter \mathbf{p}_k of \mathbf{p}
- *input ports*: a list of pairs $\mathbf{i}_k=(in_k,it_k)$, for $k \in [1,m]$, that defines the name in_k and the type it_k of each *input port* \mathbf{i}_k from which \mathbf{p} reads data, where
 - o $it_k = \text{'gradual'}$ iff \mathbf{p} starts to read through \mathbf{i}_k as soon as data items become available, and \mathbf{p} reads each data item only once
 - o $it_k = \text{'non-gradual'}$ iff \mathbf{p} starts to read through \mathbf{i}_k only after all data items are available, and \mathbf{p} may reread a data item
- *output ports*: a list of pairs $\mathbf{o}_k=(on_k,ot_k)$, for $k \in [1,n]$, that defines the name on_k and the type ot_k of each *output port* \mathbf{o}_k to which \mathbf{p} writes data, where
 - o $ot_k = \text{'gradual'}$ iff \mathbf{p} makes a data item available for reading through \mathbf{o}_k immediately after writing it, and \mathbf{p} writes a data item only once
 - o $ot_k = \text{'non-gradual'}$ iff \mathbf{p} makes data items available for reading through \mathbf{o}_k only after writing all data items, and \mathbf{p} may rewrite data items

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

A *process* p models a call to \mathbf{p} and has a list pv_1, \dots, pv_r of *parameter values* such that pv_k matches the parameter type pt_k of \mathbf{p} , for $k \in [1,r]$. We also say that p is an *instance* of \mathbf{p} , and that the parameters, input ports, and output ports of \mathbf{p} are the parameters, input ports, and output ports of p .

A *container* models a data structure that holds the data that a process reads or writes through the input or output ports. Section 4 discusses containers in detail.

The domain expert must continue the specification of \mathbf{p} by defining two lists of *cost functions* for \mathbf{p} that estimate the data volumes that instances of \mathbf{p} write through the output ports, as follows:

- a list $data-volume=(v_1, \dots, v_n)$ of functions such that, for each output port \mathbf{o}_k , function v_k returns an estimation for the maximum data volume that an instance p of \mathbf{p} produces through \mathbf{o}_k , given the parameter values passed to p and estimations for the maximum data volumes that p reads through the input ports

- a list $item-size=(s_1, \dots, s_n)$ of functions such that, for each output port \mathbf{o}_k , function s_k returns an estimation for the maximum size of an item that an instance p of \mathbf{p} writes through \mathbf{o}_k , given the parameter values passed to p and estimations for the maximum sizes of the items that p reads through the input ports

We use $data-volume[\mathbf{o}_k]$ and $item-size[\mathbf{o}_k]$ to indicate the cost functions v_k and s_k defined for output port \mathbf{o}_k of \mathbf{p} . The role of these functions is entirely similar to cost functions defined for the traditional relational operations, in the context of relational query optimization [Garcia-Molina 1999].

Finally, the domain expert concludes the specification of \mathbf{p} by defining two lists of *data rate functions* for \mathbf{p} that estimate the average rates that instances of \mathbf{p} read or write through the input and output ports, as follows:

- a list $read-rate=(r_1, \dots, r_m)$ of functions such that, for each input port \mathbf{i}_k , function r_k returns an estimation for the average rate at which an instance p of \mathbf{p} reads data items through \mathbf{i}_k , given the parameter values passed to p
- a list $write-rate=(w_1, \dots, w_n)$ of functions such that, for each output port \mathbf{o}_l , the function w_l returns an estimation for the average rate at which an instance p of \mathbf{p} writes data items through \mathbf{o}_l , given the parameter values passed to p

We use $read-rate[\mathbf{i}_k]$ and $write-rate[\mathbf{o}_l]$ to indicate the data rate functions r_k and w_l defined for the input port \mathbf{i}_k and the output port \mathbf{o}_l of \mathbf{p} . The role of these functions will become clear in Section 4.3.

3.2 Flow Model

The *flow model* introduces a simple data flow abstraction, that we call workflow graphs. Briefly, a workflow graph w indicates when a process p reads data produced by another process q . The flow of data determines the flow of control in the sense that a process may be executed as soon as the data it requires is available, which in turn depends on the types of input and output ports. Section 5 discusses this point in detail since it depends on a careful analysis of process pipeline scheduling.

A *workflow graph*, or simply a *workflow*, is a labeled bipartite graph such that (see Figure 1):

- 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 \mathbf{p} :
 - o for each input port \mathbf{i} of \mathbf{p} , there must be an arc (c,p) from a container node c to p such that $name((c,p)) = name[\mathbf{i}]$ and $type((c,p)) = type[\mathbf{i}]$
 - o for each output port \mathbf{o} of \mathbf{p} , there must be an arc (p,c) from p to a container node c such that $name((p,c)) = name[\mathbf{o}]$ and $type((p,c)) = type[\mathbf{o}]$
 - o these are the only arcs of the workflow graph.

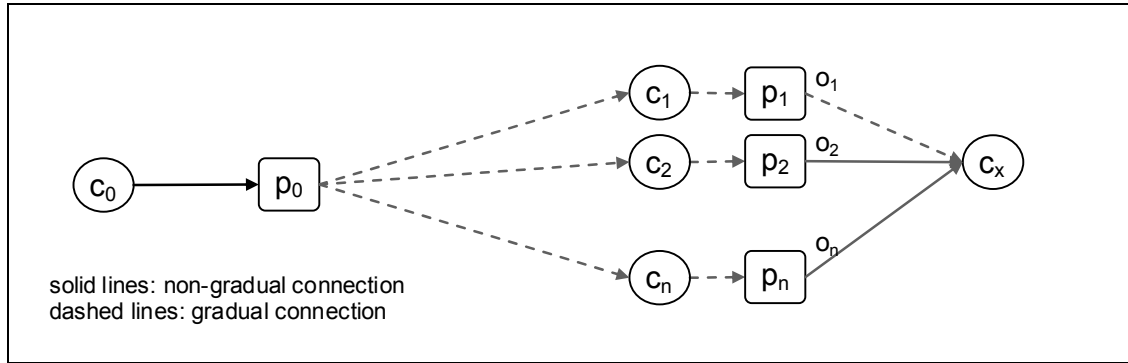


Fig. 1 A schematic example of a workflow graph.

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) is just a convenience to avoid 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 algorithms described in Section 5, as well as the estimation of container sizes, discussed towards the end of this section.

Let (c,p) be an arc from a container node c to a process node p . Assume that $name((c,p))=name[i]$ and $type((c,p))=type[i]$. Then, we say that: c is the *origin* and p is the *destination* of the arc; the arc is a *read connection*; process p is a *consumer* of c ; process p reads from c through i ; container c is *connected to* input port i of p ; the arc is a *gradual read connection* or a *gradual arc*, and p is a *gradual consumer* of c iff $type[i]='gradual'$; the arc is a *non-gradual read connection* or a *non-gradual arc*, and p is a *non-gradual consumer* of c iff $type[i]='non-gradual'$.

Likewise, let (p,c) be an arc from a process node p and a container node c . Assume that $name((p,c))=name[o]$ and $type((p,c))=type[o]$. Then, we say that: p is the *origin* and c is the *destination* of the arc; the arc is a *write connection*; process p is a *producer* of c ; process p writes into c through o ; container c is *connected to* output port o of p ; the arc is a *gradual write connection* and p is a *gradual producer* of c iff $type[o]='gradual'$; the arc is a *non-gradual write connection* and p is a *non-gradual producer* of c iff $type[o]='non-gradual'$.

Let w be a workflow graph and c be a container of w . We say that c is an *input container node* of the workflow graph iff c is not the destination of any arc (that is, c is a *source node* of w), and that c is an *output container node* of the workflow graph iff c is not the origin of any arc (that is, c is a *sink node* of w). We also say that: c is *gradual* iff all arcs starting or terminating in c are gradual; c is *non-gradual* iff all arcs starting or terminating in c are non-gradual; c is *mixed* otherwise.

We inductively extend the functions *data-volume* and *item-size* to the container nodes in w , and define a new cost function, *non-gradual-data-volume*, by traversing w from the input container nodes towards the output container nodes, which is always possible since w is acyclic by assumption.

Let c be an input container node of w . Then, c contains input data to some processes in the workflow graph. In this case, $data-volume(c)$ and $item-size(c)$ are two constants that give estimations for the input data volume and for the maximum size of any input data

item (which we assume to be available a priori since c contains input data). We leave *non-gradual-data-volume*(c) undefined since it will not be used.

Let c be a container and $q_1, \dots, q_r, q_{r+1}, \dots, q_n$ be the processes that write into c . Assume that, for each $k \in [1, n]$, for each container d that q_k reads from, the values of *data-volume*(d) and *item-size*(d) are already defined. Assume also that q_1, \dots, q_r write through gradual connections into c , and q_{r+1}, \dots, q_n write through non-gradual connections into c . Then, we define *data-volume*(c) and *item-size*(c) as follows. For each $k \in [1, n]$, let q_k be the application program of which q_k is an instance, and let \mathbf{o}_{km_k} be the output port of q_k such that $\text{name}((q_k, c)) = \text{name}[\mathbf{o}_{km_k}]$. Recall that q_k has cost functions *data-volume*[\mathbf{o}_{km_k}] and *item-size*[\mathbf{o}_{km_k}]. Then, we define the two cost functions for c as follows:

$$\text{data-volume}(c) = \sum_{k=1, \dots, n} \text{data-volume}[\mathbf{o}_{km_k}](\vec{t}_k)$$

$$\text{non-gradual-data-volume}(c) = \sum_{k=r+1, \dots, n} \text{data-volume}[\mathbf{o}_{km_k}](\vec{t}_k)$$

$$\text{item-size}(c) = \max_{k=1, \dots, n} \text{item-size}[\mathbf{o}_{km_k}](\vec{u}_k)$$

where \vec{t}_k is the vector of parameter values of q_k and of estimations for the maximum data volumes that p_k reads through the input ports, and \vec{u}_k is the vector of parameter values of q_k and of estimations for the maximum sizes of the items that q_k reads through the input ports.

Finally, we also extend the functions *read-rate* and *write-rate* to the container nodes in w , and define a new data rate function, *gradual-write-rate*, as follows. Let c be a container, p_1, \dots, p_m be the processes that read from c , and $q_1, \dots, q_r, q_{r+1}, \dots, q_n$ be the processes that write into c . Assume that q_1, \dots, q_r write through gradual connections into c , and q_{r+1}, \dots, q_n write through non-gradual connections into c . For each $l \in [1, m]$, let p_l be the application program of which p_l is an instance, and let \mathbf{i}_{lm_l} be the input port of p_l such that $\text{name}((c, p_l)) = \text{name}[\mathbf{i}_{lm_l}]$. For each $k \in [1, n]$, let q_k be the application program of which q_k is an instance, and let \mathbf{o}_{km_k} be the output port of q_k such that $\text{name}((q_k, c)) = \text{name}[\mathbf{o}_{km_k}]$. Recall that p_l has a data rate function *read-rate*[\mathbf{i}_{lm_l}] and q_k has data rate function *write-rate*[\mathbf{o}_{km_k}]. Then, we define the data rate functions for c as follows:

$$\text{read-rate}(c) = \min_{l=1, \dots, m} \text{read-rate}[\mathbf{i}_{lm_l}](\vec{t}_l)$$

$$\text{write-rate}(c) = \sum_{k=1, \dots, n} \text{write-rate}[\mathbf{o}_{km_k}](\vec{u}_k)$$

$$\text{gradual-write-rate}(c) = \sum_{k=1, \dots, r} \text{write-rate}[\mathbf{o}_{km_k}](\vec{u}_k)$$

where \vec{t}_l is the vector of parameter values of p_l , and \vec{u}_k is the vector of parameter values of q_k . Note that *read-rate*(c) is defined as the minimum of the read rates of the processes that read from c , since each data item in c has to be read by all such processes before it can be freed. Therefore, the slowest read process determines the read rate of c . However, note that *write-rate*(c) is defined as the aggregated write rates of the processes that write into c , and similarly for *gradual-write-rate*(c). This point is retaken in Section 4.3.

4 Container Implementation

If a container has to store all data items until all processes that read from it terminate, then the container may potentially occupy considerable space. However, one of the reasons for adopting pipelining is exactly to pass each data item d that a process writes directly to the processes that consume d , thereby drastically reducing the space c requires. We argue in this section that, if c is a gradual container, that is, when all arcs starting or terminating in c are gradual, then c may be implemented using data structures similar to buffers, which may save considerable space; in all other cases, c has to be implemented using data structures similar to files, or a combination of both.

Section 4.1 details the characteristics of the data structures we propose to implement containers, Section 4.2 discusses criteria for selecting each of the data structures, and Section 4.3 provides estimations for their size.

4.1 Data Structures

A *LimitedBuffer* is a data structure, similar to a buffer, that stores a fixed, finite number of data items (see Table 1 for the details). It offers *Get* and *Put* methods which are similar to those of a buffer. A data item is removed from a *LimitedBuffer* as soon as it is read by all consumers, and a data item is available for reading immediately after a producer writes it into the *LimitedBuffer*.

An *UnlimitedBuffer* is similar to a *LimitedBuffer*, except that the number of data items it may hold is not limited. It offers methods similar to those of *LimitedBuffer*, except that *Put* never blocks a producer.

A *File* is a data structure that stores an unbounded number of data items (see Table 2 for the details). It offers *Read*, *Write*, *Reread* and *Rewrite* methods as for a conventional file, in addition to the *Get* and *Put* methods of *LimitedBuffers*.

A *FileLimitedBuffer* combines a *File* with a *LimitedBuffer*, respectively called the *file component* and the *buffer component* of the data structure. The methods *Open*, *Close*, *Write*, *Rewrite* and *Read* are similar to the *File* methods, and the methods *Put* and *Get* are similar to the *LimitedBuffer* methods.

A *FileUnlimitedBuffer* is similar to a *FileLimitedBuffer*, except that the buffer component is implemented as an *UnlimitedBuffer*.

In what follows, we use *LimitedBuffer* to refer to a data structure of the class *LimitedBuffer*, and likewise for the other types of data structures.

4.2 Choice of Container Implementation

Let c be a container. There are two scenarios to consider.

Scenario 1: the data rates for c are not available.

Suppose that all connections of c are gradual. Then, c may be implemented using a *LimitedBuffer* whose size is equal to $item-size(c)$. Indeed, since all connections of c are gradual, the data items that producers write into c may be passed, one by one, using the *LimitedBuffer*, to the consumers that read from c .

Table 1 Characteristics of the LimitedBuffer Class.

Structure	<ul style="list-style-type: none"> – limited size queue, with multiple read and write connections, all gradual – a data item is removed when read by all consumers – data items are not ordered (unlike conventional queues) – there is no priority among the connections, whether they are blocked or not 	
Methods	<i>Open(c)</i>	– process opens the connection c with the container
	<i>Close(c)</i>	– process closes the connection c with the container
	<i>Put(c,d)</i>	<ul style="list-style-type: none"> – connection c must be a write connection – if the buffer has space enough available to store d, then d is added to the container; otherwise c is blocked until the buffer has enough space to store d – d is available for reading by consumers
	<i>Get(c)</i>	<ul style="list-style-type: none"> – connection c must be a read connection – if the container has a data item d not yet read through c, then d is returned; otherwise, c is blocked until the container has a data item not yet read through c

Table 2 Characteristics of the File Class.

Structure	<ul style="list-style-type: none"> – unlimited size file, with multiple read and write connections, gradual or non-gradual – data items generated by each non-gradual write connections are ordered – there is no priority among the connections, whether blocked or not 	
Methods	<i>Open(c)</i>	– process opens a connection c with the container
	<i>Close(c)</i>	– process closes the connection c with the container
	<i>Put(c,d)</i>	<ul style="list-style-type: none"> – connection c must be a gradual write connection – adds d to the container – d is available for reading by consumers
	<i>Write(c,d)</i>	<ul style="list-style-type: none"> – connection c must be a non-gradual write connection – adds d to the container – the method returns the position of d in the file – d is available for reading by a consumer only after c is closed
	<i>Rewrite(c,d,i)</i>	<ul style="list-style-type: none"> – connection c must be a non-gradual write connection – rewrites d in position i of the file
	<i>Get(c)</i>	<ul style="list-style-type: none"> – connection c must be a gradual read connection – if the container has a data item d not yet read through c, then d is returned; otherwise, c is blocked until the container has a data item not yet read through c and which is available for reading
	<i>Read(c,i)</i>	<ul style="list-style-type: none"> – connection c must be a non-gradual read connection – returns the data item in position i

Suppose that at least one read connection of c is non-gradual. Then, c must be implemented using a *File*, since all data items that producers write into c have to be stored in the *File* before any non-gradual consumer starts reading from c .

Suppose that all read connections of c are gradual and at least one write connection of c is non-gradual. Then, c may be implemented using a *FileLimitedBuffer*. Indeed, data items that non-gradual producers write into c have to be stored in the file component, to be subsequently read by the gradual consumers, whereas data items that gradual producers write into c will be passed, one by one, using the buffer component, to the gradual consumers that read from c .

Scenario 2: the data rates for c are available.

Let $R = \text{read-rate}(c)$ be the minimum read rate of processes that read from c , $W = \text{write-rate}(c)$ be the aggregated write rate of processes that write into c , and $G = \text{gradual-write-rate}(c)$ be the aggregated write rate of processes that write into c through gradual write connections.

Suppose that all connections of c are gradual. If $R \geq W$, then c may be implemented using a *LimitedBuffer*. Indeed, since $R \geq W$, then it is expected that the *LimitedBuffer* will not block producers too long.

If $R < W$, then c may be implemented using an *UnlimitedBuffer*. Indeed, if the slowest read rate is less than the aggregated write rate, it is expected that a *LimitedBuffer* will block producers too often. Therefore, an *UnlimitedBuffer* is preferred. A *File* might also be an option. However, an *UnlimitedBuffer* allows freeing a data item as soon as it is read by all consumers, since by assumption all connections are gradual, differently from a *File*, which retains all data items written until the container is discarded.

Suppose that at least one read connection of c is non-gradual. The only possible choice is again to implement c using a *File*, for the reasons already explained in the previous scenario.

Suppose that all read connections of c are gradual and at least one write connection of c is non-gradual. If $R \geq G$, then c may be implemented using a *FileLimitedBuffer*, since again it is expected that the buffer component will not block producers too long. If $R < G$, then c may be implemented using a *FileUnlimitedBuffer*, thereby allowing faster gradual consumers to keep reading newer data items, through the (unlimited) buffer component as soon as gradual producers write them, while slower consumers will still read older data items. The file component handles data items written by the non-gradual producers.

4.3 Estimation of Container Sizes

Let w be a workflow graph and c be a container of w . Recall that $\text{data-volume}(c)$ is an estimation for the aggregated data volume that producers write on c , $\text{non-gradual-data-volume}(c)$ is an estimation for the aggregated data volume that non-gradual producers write into c , and $\text{item-size}(c)$ is an estimation for the maximum size of an item that processes write on c .

The container node has a label *size*, initialized with the maximum amount of disk space reserved for the data structure chosen to implement c , as follows:

- $\text{size}(c) = \text{item-size}(c)$, if c is implemented as a *LimitedBuffer*
- $\text{size}(c) = \text{data-volume}(c)$, if c is implemented as an *UnlimitedBuffer*, a *File* or a *FileUnlimitedBuffer*
- $\text{size}(c) = \text{non-gradual-data-volume}(c) + \text{item-size}(c)$, if c is implemented as a *FileLimitedBuffer*

The maximum amount of disk space reserved for an *UnlimitedBuffer* or a *FileUnlimitedBuffer* is equivalent to that of a *File*, since they do not block producers. However, unlike a *File*, they allow a data item to be freed as soon as all consumers read

it. Therefore, they are not equivalent to a *File* as far as the average amount of disk space is concerned.

Finally, the maximum amount of disk space reserved for a *FileLimitedBuffer* is less than that for an *UnlimitedBuffer*, a *File* or a *FileUnlimitedBuffer*, since its buffer component, which occupies $item-size(c)$ units of disk space, handles gradual write connections to the container.

5 A Process Pipeline Scheduling Algorithm

In this section, we describe a process pipeline scheduling algorithm that takes into account the amount of disk space available. We also show that the process pipeline scheduling problem is NP-Complete.

The algorithm is based on the workflow model defined in Section 3, and on the container implementations discussed in Section 4. In particular, it uses the container size estimations to compute the amount of disk space that must be pre-allocated to each container node.

The reader may cover Section 6 first, which contains an example of how the process pipeline scheduling algorithm works, before going through the details of this section.

5.1 Augmented Workflow Graph

The algorithm depends on a variation of the workflow graph introduced in Section 3, and modified in Section 4.3 to include the size of container implementations. Since the changes are minor, we continue to refer to this graph as the workflow graph.

The variation again includes new node and arc labels. A process node has a new label, the *state*, with values '*initial*', '*pending*', '*executing*', and '*finished*'. Likewise, an arc has a new label, also called *state*, with values '*idle*', '*open*' and '*closed*'.

A process node p is *ready* (for execution) when its state is '*initial*' or '*pending*' and the state of each arc from a container node to p is '*open*', i.e., all input connections that p needs are already open.

Let w be a workflow graph, with the process and container state labels.

The *finished subgraph* of w , denoted $Finished[w]$, is the subgraph of w containing all process nodes of w labeled with '*finished*', all container nodes of w that such processes read from or write into, and all arcs in w that connect these nodes. We likewise define the *execution subgraph*, denoted $Exec[w]$, and the *ready subgraph* of w , denoted $Ready[w]$, by considering process nodes of w labeled with '*executing*', and with '*initial*' or '*pending*', respectively.

Let g be one of the above subgraphs. We say that c is an *input container node* of g iff c is a source node of g ; node c is an *output container node* of g iff c is a sink node of g ; and node c is a *frontier container node* of g iff there is an arc in g which is incident to c and there is an arc in w , but not g , which is incident to c . Note that, an input container node of $Exec[w]$ may be an input container node of w , or an output container of $Finished[w]$. Furthermore, note that an input container node of $Ready[w]$ may be an input container node of w , or an output container of $Finished[w]$ or $Exec[w]$.

The *size* of $Exec[w]$ is the sum of the sizes of (the implementations of) the containers in $Exec[w]$. The *size* of $Ready[w]$ is the sum of the sizes of the containers in $Ready[w]$ that are not in $Exec[w]$.

Finally, we observe that the algorithm may not consider for execution, at the same time, all processes that write into or read from a container. This implies that the algorithm may have to allocate a container holding data items for processes that are now executing, as well as for processes whose execution is postponed. However, the algorithm always allocates the container with the maximum size required for all processes that write into it, including those processes that are postponed.

5.2 Controlling Process Node States

Let w be a workflow graph. When the algorithm starts interpreting w , it sets the state of each process node in w to *'initial'*.

Suppose now that the process and container state labels of w reflect a certain stage of the execution. The algorithm selects the processes to be executed next in two phases. In the first phase, the algorithm does not take into account disk space and only creates the ready subgraph of w .

In the second phase, the algorithm modifies the ready subgraph by pruning nodes until the subgraph contains only process nodes that can be executed in parallel, together with those processes already in execution, within the limits of disk space available. Note that the total amount of disk space consumed is the sum of the sizes of the containers already allocated plus the sum of the sizes of the new containers required to run the processes selected to be executed. The pruning step is the heart of the algorithm and is discussed in detail in Sections 5.4 through 5.7.

The algorithm signals that a process node p is pruned and, hence, has to be postponed, by changing its state to *'pending'*, and it signals to start a process node p in the ready graph (after the pruning step) by setting its state to *'executing'*.

If the state of a process p is changed to *'finished'* then, for each container c that p reads from, all write connections to c are in state *'closed'*. In other words, a process can only be considered finished when all data items it must read have already been written.

5.3 Controlling Arc States

Let w be a workflow graph. When the algorithm starts interpreting w , it initializes the state of each arc as follows. It first sets the state of each arc to *'idle'*, except those arcs that connect input container nodes to processes, which it sets to *'open'*. We call this rule the *arc initialization rule*. Then, it recursively applies the pipelining rules described below, perhaps changing the state of certain arcs to *'open'*. As a consequence, several processes immediately become ready for execution, by definition of ready process. Intuitively, the data items they require are either stored in the input containers, or they may be obtained by pipelining from other processes.

When the algorithm changes the state of a process p to *'executing'*, it changes the state of each arc that connects p to a container node to *'open'* (if not already in this state). Note that the state of each arc from a container node to p must already be *'open'* since

the algorithm changes to the *'executing'* state only processes in the ready graph (and, by definition, the state of each arc from a container node to a ready process is *'open'*).

A process p that is executing may change the state of an arc to or from p at any time during execution.

When the algorithm changes the state of a process p to *'finished'*, it changes the state of each arc that connects a container node to p or that connects p to a container node to *'closed'*, as p does not need to produce or consume more data.

Changing the state of a process p to *'pending'* does not affect the state of any arcs.

The algorithm also changes the state of other arcs by recursively applying the following *pipelining rules*:

- S1. When all arcs from process nodes to a container node c are in state *'closed'*, then the state of each non-gradual arc from c to a process node in state *'initial'* must be set to *'open'*.
- S2. When there is at least one arc from a process node to a container node c such that the arc is in state *'closed'* and is non-gradual, or the arc is in state *'open'* and is gradual, then the state of each gradual arc from c to a process node in state *'initial'* must be set to *'open'*.
- S3. When all arcs from container nodes to a process node p are in state *'open'* and p is in state *'initial'*, then the state of each gradual arc from p to a container node must be set to *'open'*.

Let p be a process node. First observe that we need not consider p , if p is in states *'executing'* or *'finished'*, since all arcs to or from p are already in states *'closed'* or *'open'*, respectively. Also, we need not consider p , if p is in state *'pending'*, since p must have already passed through a stage where p was in state *'initial'* and, therefore, these rules have already been applied.

Rule S1 is fairly simple and reflects the fact that, if all processes cease to write into a container c , all processes that read from c through a non-gradual connection may do so.

Rules S2 and S3 are mutually recursive and capture the core of pipeline scheduling. From the perspective of a container c , Rule S2 says that a process p may start reading data items from a container c through a gradual connection as soon as: (1) a process that writes into c through a non-gradual connection has already closed the connection; or (2) a process starts writing into c through a gradual connection. This is a direct consequence of the definition of gradual and non-gradual connections.

From the perspective of a process p , Rule S3 says that as soon as p starts processing, or is ready to be processed, then p may pipeline data items to other processes through gradual connections.

The pipelining rules indeed summarize the central strategy of process pipelining scheduling embedded in the algorithm. However, the scheduling of new processes is conditional to the available disk space, as discussed in Sections 5.4 through 5.7.

Finally, the algorithm identifies each container node c whose incident arcs are all in state *'closed'*. This situation reveals that processes ceased to read from or write into c

and, hence, c can be discarded, and the disk space that c occupies can be reclaimed. The algorithm then loops back and selects the next processes to be executed.

We can now prove a property of ready subgraphs that reflect the pipelining rules.

Proposition 1: Let w be a workflow graph, with the process and container state labels.

- a) For each arc (c,q) in $Ready[w]$, one of the following conditions must be met:
- i) c is an input container node of $Ready[w]$
 - ii) (c,q) is gradual, and c is a frontier container node of $Ready[w]$
 - iii) (c,q) is gradual, and there is an arc (p,c) in $Exec[w]$ or $Ready[w]$ such that (p,c) is gradual
- b) For each arc (p,c) in $Ready[w]$, then
- i) every arc (d,p) in w is in state 'open'
 - ii) if (p,c) is gradual, then (p,c) is in state 'open'

Proof. Let w be a workflow graph.

(a) Let (c,q) be an arc in $Ready[w]$. Since q is ready, (c,q) must in state 'open', by definition of ready processes. But arc (c,q) could have reached state 'open' only in three situations (note that Rule S3 does not apply in this case):

- (1) By applying the arc initialization rule, which implies that c is an input container node of w , and hence of $Ready[w]$. Therefore, condition (i) holds.
- (2) By applying Rule S1, which implies that every arc (p,c) in w is in state 'closed', and hence p is 'finished'. Hence, c is also an input container node of $Ready[w]$. Therefore, condition (i) holds.
- (3) By applying Rule S2, which implies that there is at least one arc (p,c) such that (p,c) is in state 'closed' and is non-gradual, or (p,c) is in state 'open' and is gradual, and (c,q) is gradual. If (p,c) is in state 'closed', p must be 'finished', which implies that (p,c) is not in $Ready[w]$. Therefore, c is a frontier container node of $Ready[w]$. Therefore, condition (ii) holds. If (p,c) is in state 'open' and is gradual, process p cannot be in state 'finished'. Hence, p is in $Exec[w]$ or $Ready[w]$. Therefore, condition (iii) holds.

(b) Let (p,c) be an arc in $Ready[w]$. Then, p must be a ready process, which implies that every arc (d,p) is in state 'open', and p is in states 'initial' or 'pending', by definition of ready process. If p is in state 'initial', by Rule S3, if (p,c) is gradual, then (p,c) is set to state 'open'. If p is in state 'pending', in an earlier stage, p was in state 'initial'. Hence, again by Rule S3, if (p,c) is gradual, then (p,c) is set to state 'open'. \square

5.4 Naïve Process Scheduling

Given a workflow graph w , in view of Proposition 1, pruning nodes from $Ready[w]$ may not result in a correct process scheduling. We therefore define that a subgraph H of $Ready[w]$ is *consistent* iff the following conditions hold:

- 1) the sources of H are input container nodes of $Ready[w]$
- 2) for each container node c in H , there is a process node p in H that reads from or writes into c
- 3) for each process node p in H , for each container node c in w that p reads from or writes into, c is in H and the arc connecting c and p is in H
- 4) for each arc (c,q) in H , one of the following conditions must be met:
 - a) c is an input container node of $Ready[w]$
 - b) (c,q) is gradual, and c is a frontier container node of $Ready[w]$
 - c) (c,q) is a gradual, and there is an arc (p,c) in $Exec[w]$ or H such that (p,c) is gradual

Let B be the total amount of disk space available to execute new processes. That is, B is the total amount of disk space reserved to run workflows, less the sum of the sizes of the containers already allocated. We say that H is *consistent with B* iff H is consistent and the following additional condition holds:

- 5) the container nodes in H consume less than B units of disk space

Moreover, we say that a subgraph H of $Ready[w]$ is *optimal for B* iff H is consistent with B and H has the largest number of process nodes among all consistent subgraphs of $Ready[w]$. Finding the optimal subgraph therefore amounts to pipelining as many processes as possible, respecting the amount of disk space available.

A naïve algorithm for finding the optimal subgraph of $Ready[w]$ is shown in Figure 2. However, we can prove that the algorithm is exponential.

Proposition 2: The worst-case running time for the Naïve Process Scheduling is $O(2^n)$.

Proof. Let G be a ready subgraph, with n nodes, of a workflow graph. The worst-case running time for each step is as follows. Step 1 is $O(2^n)$ since it requires constructing all consistent subgraphs of G , and there are at most 2^n such subgraphs. Step 2 is also $O(2^n)$ since it requires visiting each subgraph produced by Step 1 and selecting that with the largest number of process nodes. \square

Naïve Process Scheduling

Input:

G - a ready subgraph, with n nodes, of a workflow graph
 B - the total amount of disk space available

Output:

H - an optimal subgraph of G

1. Construct all consistent subgraphs of G with k nodes, where k ranges from 1 to n .
2. Choose an optimal consistent subgraph H of G .

Fig. 2 Naïve Process Scheduling.

5.5 Complexity of the Process Pipeline Scheduling Problem

We may in fact prove that the problem we are trying to solve is NP-Complete. Let $|S|$ denote the cardinality of set S , and Z^+ denote the set of all positive integers. Recall the *Partially Ordered Knapsack* problem is defined as follows [Garey and Johnson 1979]:

Instance: A finite set U , a partial order $<$ over U , functions $s:U \rightarrow Z^+$ and $v:U \rightarrow Z^+$ that assign to each $u \in U$ a size $s(u)$ and a value $v(u)$ (both of which are positive integers), and positive integers B and K .

Question: Is there a subset $U' \subseteq U$ such that, for any $u, u' \in U$, if $u \in U'$ and $u' < u$ then $u' \in U'$, and $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

A set U' that satisfies the above conditions is called a solution to the Partially Ordered Knapsack problem instance.

We define the *Process Pipeline Scheduling* problem as follows:

Instance: A finite set P of processes, a partial order \ll over P , where $p \ll p'$ indicates that process p can pipeline data items to process p' , a function $r:P \rightarrow Z^+$ that assigns, to each $p \in P$, the number $r(p)$ of resource units that p requires, and positive integers B and K .

Question: Is there a process schedule $P' \subseteq P$ such that, for any $p, p' \in P$, if $p \in P'$ and $p' \ll p$, then $p' \in P'$, and $\sum_{p \in P'} r(p) \leq B$ and $|P'| \geq K$?

Likewise, a set P' that satisfies the above conditions is called a solution to the Process Pipeline Scheduling problem instance.

The Process Pipeline Scheduling problem is defined to capture just the basic question of how many processes the process pipeline scheduling algorithm can schedule in each step by taking advantage of pipelining. P represents the set of all processes that are ready in a given step, together with all processes that can be pipelined with them, and P' indicates the set of all processes that are selected to run in that step. The condition $\sum_{p \in P'} r(p) \leq B$ captures that resource consumption is limited to B units, and the condition $|P'| \geq K$ indicates that at least K processes are scheduled.

We can then prove that:

Theorem 1: The Process Pipeline Scheduling problem is NP-Complete.

(see Appendix 1 for the proof).

5.6 Greedy Process Scheduling

In view of Theorem 1, a greedy heuristics [Rajasekaran 1996, chapter 4] for finding a consistent subgraph of $Ready[w]$, whose worst-case running time is polynomial, is shown in Figure 3.

Greedy Process Scheduling

Input:

W - a workflow graph (with state labels)
G - a ready subgraph, with n nodes, of W
B - the total amount of disk space available

Output:

H - a consistent subgraph of G, or
the empty graph, if the scheduling is infeasible for B

```
1.  H = G
2.  L =  $\emptyset$  /* L is an auxiliary list */
3.  while H is not consistent with B and
      H has a container node
      which is not an input container node of G
4.  do
5.    for each sink container node s of H
6.      do
7.        (g,P) = Gain(s,H,W) /*see Figure 4 */
8.        add (g,P) to L
9.      end
10.   let (g,P) be the pair in L with the highest gain g
11.   Prune all nodes in P from H
12. end
13. if H contains only input container nodes of G
14.   then return  $\emptyset$  /* the scheduling is infeasible for B */
15.   else return H
```

Fig. 3 Greedy Process Scheduling.

Let H be the result of pruning the ready subgraph. To compute the gain (in disk space) when a sink container node s of H is pruned, we proceed as follows (see Figure 4). When s is pruned, each process node p that produces data in s also has to be pruned (lines 8 to 12 in Figure 4), as well as each container node c such that p is the last process to write into c (lines 14 to 18 in Figure 4), as so recursively (implemented with the help of an auxiliary queue L in Figure 4). Hence, when a container node s is pruned, a set of process and container nodes must also be pruned.

Gain

Input: W - a workflow graph (with state labels)
H - a ready subgraph of W
s - a sink container of H that has to be pruned

Output: P - a set of nodes, that includes s, such that
if all nodes in P are pruned from H,
the resulting graph is consistent
g - the total gain obtained by pruning s

```
1.  mark s as visited
2.  initialize L with s /* L is an auxiliary queue */
3.  while L is not empty
4.  do
5.      n = first(L) /* remove the first element from L */
6.      if n is a container node
7.          then
8.              for each unvisited process node p in H
                  such that p reads from or writes into n
9.                  do
10.                     add p to L
11.                     mark p as visited
12.                 end
13.             else /* n is a process node */
14.                 for each unvisited container node c in H such that
                        n writes into c and
                        c is not a frontier container node of H and
                        all process nodes in H that write into c
                        are marked as visited
15.                     do
16.                         add c to L
17.                         mark c as visited
18.                     end
19.             end
20.             g = 0
21.             for each container node c in H marked as visited
22.                 do
23.                     g = g + size(c)
24.                 end
25.                 for each unvisited container node c in H such that
                        c is implemented as a LimitedBuffer and
                        there is a process, marked as visited, that reads from c
26.                 do
                /* the size of c increases from that of a LimitedBuffer
                   to that of a File.
                   Therefore, the gain decreases by the difference of
                   the sizes of both data structures
                */
27.                     g = g - (size of c when implemented as a File)
                           + (size of c when implemented as a LimitedBuffer)
28.                 end
29.             Construct P as the set of all nodes in H marked as visited.
30.             return P and g
```

Fig. 4 Computing the gain obtained from the potential pruning of a container.

Moreover, the pruning process has to re-consider the implementations adopted, and thereby the disk space required, for some containers in H . A container node c falls into this situation when c satisfies two conditions: (1) c was previously implemented by a *LimitedBuffer* and used to pipeline data items to a process q ; and (2) process q was previously ready for execution but, as a result of the pruning process, it is now pending. In this situation, c will have to store the complete set of data items that q will read when scheduled for execution, that is, c now has to be implemented by a *File*. Thus, the size of c actually increases by postponing q . Therefore, pruning some container nodes, and the process nodes that require them, will not always reduce the required space. In general, the disk space gained from pruning a set of container and process nodes is the difference between the sum of the sizes of the container nodes that are pruned and the sum of the extra space required by the data structures that will now implement the container nodes that had their implementation strategy changed. We can prove that Greedy Process Scheduling has acceptable performance:

Proposition 3: The worst-case running time for the Greedy Process Scheduling algorithm is $O(n^2(n+e))$.

Proof. Let G be an execution graph with n nodes and e arcs. The worst-case running time for each step of the algorithm in Figure 3 is as follows. By Step 1, in the first stage and in all subsequent stages, H also has at most n nodes and e arcs. Observe that $Gain(s,H)$ (in Figure 4) is $O(n+e)$ since it is similar to a breath-first search of H , starting with the sink node s , except that only certain adjacent nodes are visited (Steps 8 and 14 in Figure 4). Since there are at most n sink container nodes in H , function $Gain$ is called at most n times. Therefore, Step 5 in Figure 3 is $O(n(n+e))$. Step 10 is $O(n)$ since the length of L is at most n . Step 11 is $O(n+e)$ since P has at most n elements and all arcs adjacent to nodes in P have to be deleted from H . Thus, the cost of Steps 5 to 10 is $O(n(n+e))$. Since in each loop of Step 3, the graph is at least one node smaller, Steps 5 to 10 are repeated at most n times. Hence, the total cost is $O(n^2(n+e))$. \square

Finally, we can combine the discussion in this and the previous subsections of Section 5 in a single result that states that the process pipeline scheduling algorithm (with either the naïve or the greedy strategies) always creates a correct process scheduling, in the following sense:

Theorem 2: Let B be a positive integer representing the total amount of disk space available. Let w be a workflow graph with n process nodes. Assume that w_k is w with the process and container state labels created by the process pipeline scheduling algorithm after changing k process nodes to state ‘*executing*’, for any $k \leq n$. Then, $Exec[w_k]$ defines a *correct process scheduling* in the sense that:

- i) For each process node p in $Exec[w]$, for each arc (c,p) in w , (c,p) is in state ‘*open*’.
- ii) The size of $Exec[w]$ is less than B .

(The proof is by induction on the number of stages the algorithm has already executed).

5.7 Extending the Scheduling Strategies

In this section, we very briefly indicate how we can extend the scheduling strategy to more complex environments. A complete discussion can be found in [Lemos 2004a].

Both the Naïve Process Scheduling and the Greedy Process Scheduling algorithms consider a single parameter, B , which expresses the amount of disk space available. However, if secondary storage is partitioned into multiple volumes, we have to consider a list of space parameters, B_1, \dots, B_n , and extend the algorithms to decide in which volume each container should be allocated. The allocation of containers to volumes cannot be decided by traversing the pruned ready subgraph from sink nodes to source nodes, since we run the risk of finding no volume with enough disk space to allocate the source container nodes, which should have been allocated first. In this case, we have to be more conservative and analyze the subgraph from sources to sinks instead.

In another direction, consider a distributed environment with multiple processors accessing multiple disk volumes. Suppose first that the cost of accessing remote data is of the same order of magnitude as the cost of accessing local data (as in a Gigabit local area network). Very briefly, in this case, we may consider pipelining processes running in different processors, with the proviso that transferring partial results from one processor to another does not override the parallelization gains.

Suppose now that the cost of accessing remote data is much higher than the cost of accessing local data. In this case, a process is best allocated to a processor that requires the least amount of data transfer to run the process. This implies that pipelining from one process to another is best used when both processes run on the same processor.

However, it might be the case that, to schedule a set of processes, one cannot avoid moving a container from one processor to another. In this case, we have to verify if c is a resource that can be moved to other processors, or accessed by processes running on processors different from that where c is stored. If this is not the case, we call c an *anchor*, and we have to run the processes that read from or write into c in the processor P where c resides, irrespectively of the cost of moving to P the other container nodes that such processes require.

In general, in any distributed environment, if we may run a process in more than one processor, we have to devise a policy to choose the best alternative. The cost function may combine processor capacity, occupancy, space availability, as well as data transfer costs.

5.8 An Architecture for a Workflow Manager based on Web Services

In this section, we briefly describe an architecture for a workflow manager, based on Web services, that incorporates process pipeline scheduling. This architecture leverages on the ability of service domains [Tan et al. 2006] to select and monitor the appropriate Web service instances.

Figure 5 shows the top level components, grouped into the *Workflow Domain* and one or more *Service Domains*.

The *Workflow Domain* is composed of two modules: *Workflow Assistant* and *Workflow Controller*.

The *Workflow Assistant* helps the user create a workflow, re-define a workflow (when intermediate or final results are judged not be useful or interesting), validate, optimize and schedule workflow execution, and analyze workflow results.

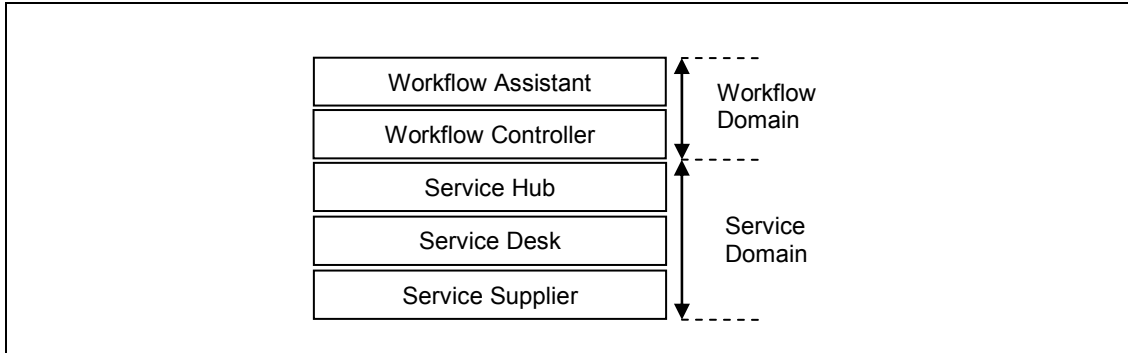


Fig. 5 An architecture for a workflow manager, based on Web services.

The *Workflow Controller* is responsible, among other tasks, for implementing the process pipeline scheduling algorithm. It features a registration service that keeps a description of the services each service domains controls. The description of a process p should indicate how p reads and writes from containers, as discussed in Section 3. If such information is unavailable, *non-gradual* is assumed by default. 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 , and sends the request to SD . It signals to the service hub that no pipelining will apply to p by sending a *complete* process execution request, that is, a request indicating that all input containers are completely available; it signals that pipelining may apply to p by sending a *partial* request, that is, a request indicating that only fragments of an input container are available.

The *Service Domain* is composed of three modules: *Service Hub*, *Service Desk* and *Service Supplier*.

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*.

A *Service Desk* typically selects the appropriate Web service instance to process a Web service request. In addition to pipelining, *Service Desks* implement data partition as a parallelization strategy. The dynamic selection of a Web service instance is not just based on availability, but it also takes into account quality of service characteristics, real-time data, data recorded during prior executions of a service and users' preference. 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].

A *Service Supplier* simply implements one or more Web services.

Implementations of the *Workflow Assistant* and of the *Workflow Controller* are discussed in [Lemos et al. 2004b] and in [Silva 2006], respectively. The implementation of the *Workflow Controller* was tested by simulating processes and containers, rather than using actual application programs and data.

An implementation of the *Service Domain* component is described in [Rosa 2006; Lemos et al. 2007]. Internally, it incorporates Web service selection strategies, such as randomization or round-robin, and is prepared to incorporate new strategies.

The *Service Domain* 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]. The instantiation is called *BioService domain*. 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 developed. However, there is a single *BioServiceHub* class, which must be adapted only when new programs are incorporated.

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.

6. Example

To illustrate the process pipeline scheduling algorithm described in Section 5, using the greedy heuristics and assuming a single volume secondary storage, consider the workflow graph w shown in Figure 5, where c_i denotes a container node, p_j denotes a process node and a_{ij} denotes an arc. Dashed lines indicate gradual arcs and solid lines, non-gradual arcs.

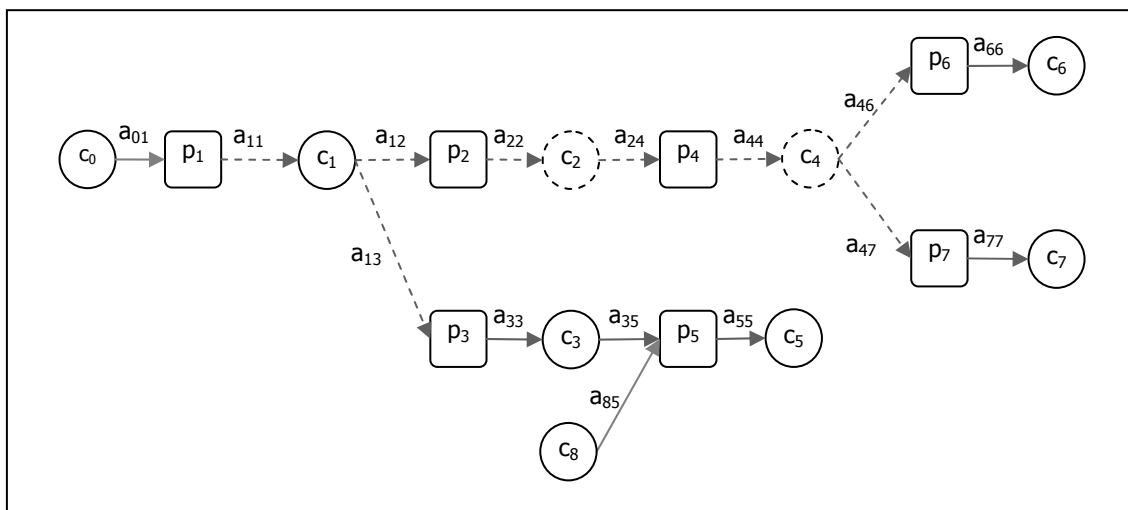


Fig. 5 Workflow graph w .

Tables 3 and 4 show the process and the arc states after initialization. Each state of a process or an arc is indicated by its initial letter ('i' for 'initial', etc...). Observe that, since c_0 and c_8 are input container nodes, the states of a_{01} and a_{85} are set to 'open', by the arc initialization rule. Since a_{01} is 'open', process p_1 becomes ready. By applying the pipelining rule S3, the state of a_{11} is then set to 'open'. Now, by applying the pipelining rule S2, the states of a_{12} and a_{13} are set to 'open', and likewise for the rest of the arcs in Table 4.

Table 3. Process States.

Process	State
p_1	i
p_2	i
p_3	i
p_4	i
p_5	i
p_6	i
p_7	i

Table 4. Arc States.

Arc	State	Arc	State	Arc	State
a_{01}	o	a_{24}	o	a_{77}	i
a_{11}	o	a_{44}	o	a_{33}	i
a_{12}	o	a_{46}	o	a_{35}	i
a_{13}	o	a_{47}	o	a_{55}	i
a_{22}	o	a_{66}	i	a_{85}	o

Suppose that B is the disk space available for workflow execution. We compute the next processes to execute as follows.

We first construct the ready subgraph, H_1 , shown in Figure 6. Note that only p_5 is not ready since it reads, through a non-gradual connection, the data items that p_3 writes (arc a_{35} is non-gradual and has state equal to ‘inactive’).

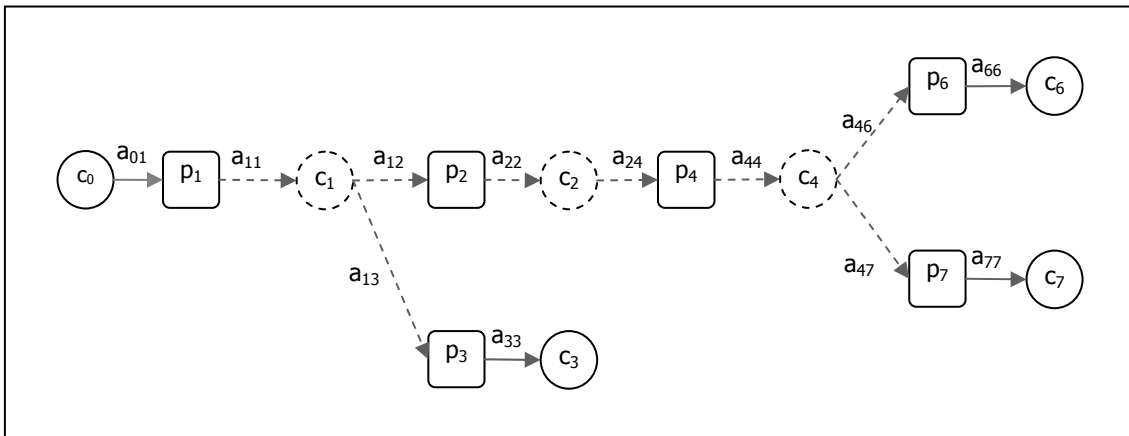


Fig. 6 Ready subgraph H_1 .

Suppose that the size of H_1 is greater than B . Then, we must start the pruning process. The output container nodes (the sinks) of H_1 are c_3, c_6, c_7 . Suppose that g_3, g_6 and g_7 are the gains in space obtained by removing c_3, c_6 and c_7 , respectively.

As an example, we discuss how to compute g_7 . The list of nodes that must be pruned starts with (c_7) . Because p_7 is the only process that writes into c_7 , it is the only node added to the list, which now becomes (c_7, p_7) . Since p_7 does not write into any other container, no other node is added to the list. When we delete c_7 and p_7 , the container node c_4 , that p_7 reads from through a gradual connection (arc a_{47} is gradual), will have to store all data items, that is, implemented as a file, until p_7 is finally executed in a subsequent stage. Hence, the set of containers that change size when c_7 is pruned is simply $\{c_4\}$.

When c_7 is removed, the space gained, g , is the current size of c_7 , that is, the value of the label *size* of c_7 . But the net gain, g_7 , is the difference between g and the extra space required by c_4 , which had its implementation strategy changed to a file.

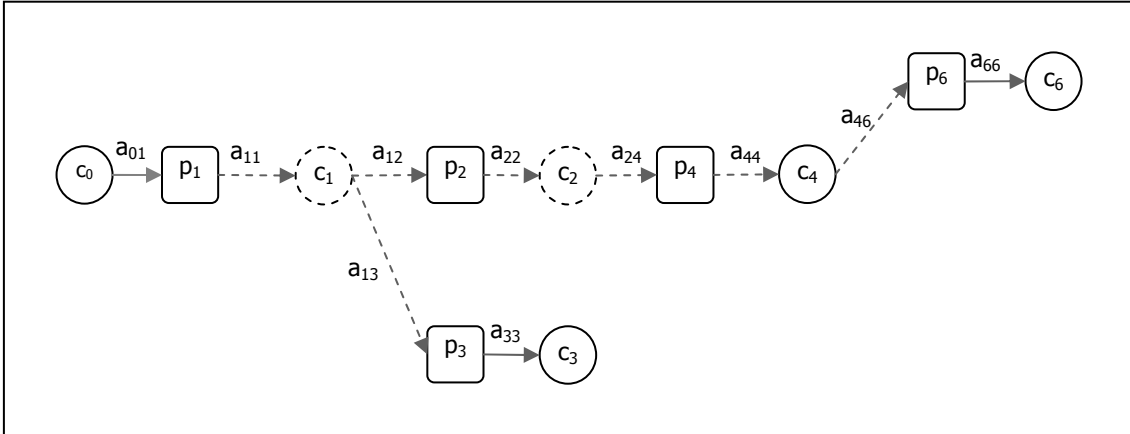


Fig. 7 Ready subgraph H_2 .

Suppose that c_7 has the best gain. We then create the pruned ready subgraph H_2 by deleting c_7 and p_7 , as shown in Figure 7.

Suppose that the space H_2 requires still exceeds the available space. We then repeat the above procedure.

The output container nodes of H_2 are c_3 and c_6 . Suppose that g_3 and g_6 are the gains in space obtained by removing c_3 and c_6 , respectively, and that $g_6 > g_3$. Then, we prune c_6 and p_6 . We note that this pruning does not require pruning any other container or process node. Furthermore, it does not affect the size of c_4 , which already stores all data items that p_4 writes. Hence, the gain obtained by pruning c_6 is equal to the size of c_6 .

Figure 8 shows the new pruned ready subgraph, H_3 , after deleting c_6 and p_6 .

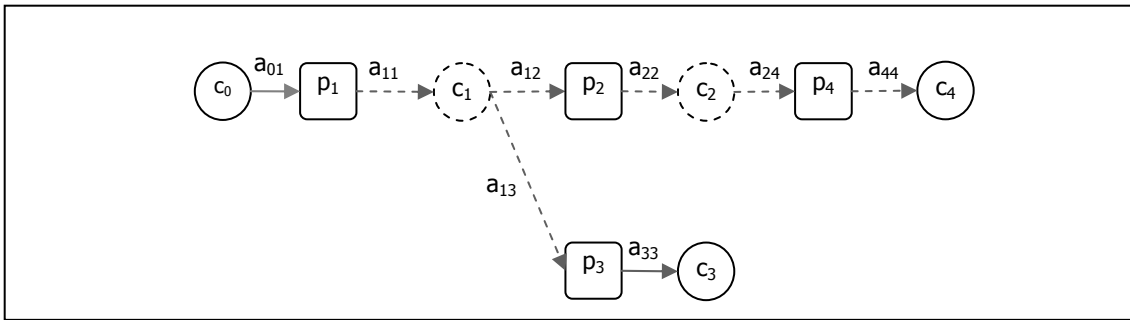


Fig. 8 Ready subgraph H_3 .

Suppose that the space H_3 requires still exceeds the available space. Suppose that the best gain is obtained by pruning c_3 and p_3 . We then construct the new pruned ready subgraph, H_4 , as shown in Figure 9. Note that c_1 now has to be implemented as a file to store the data items that p_3 will read.

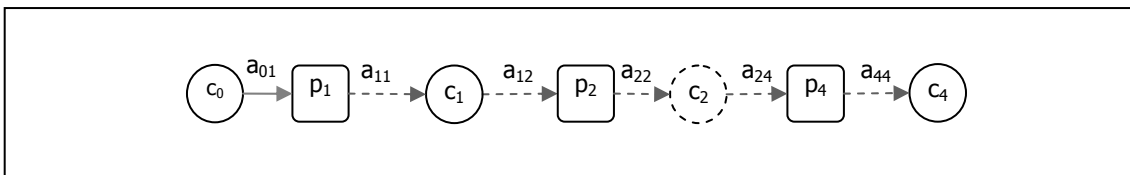


Fig. 9 Ready subgraph H_4 .

Table 5. Process States.

Process	State
p_1	e
p_2	e
p_3	p
p_4	e
p_5	i
p_6	p
p_7	p

Table 6. Arc States.

Arc	State	Arc	State	Arc	State
a_{01}	o	a_{24}	o	a_{77}	i
a_{11}	o	a_{44}	o	a_{33}	i
a_{12}	o	a_{46}	o	a_{35}	i
a_{13}	o	a_{47}	o	a_{55}	i
a_{22}	o	a_{66}	i	a_{85}	o

Tables 5 and 6 show the process and arcs states after firing p_1 , p_2 and p_4 . Note that the state of some process nodes changed from ‘initial’ to ‘executing’.

Suppose that p_1 terminates. Then, the state of p_1 changes to ‘finished’ and the state of arcs a_{01} and a_{11} change to ‘closed’.

Figure 10 shows the ready subgraph, H_5 , after p_1 finishes. Note that H_5 has two connected components and that the ready processes are p_3 , p_6 , p_7 . The processing of H_5 follows similarly.

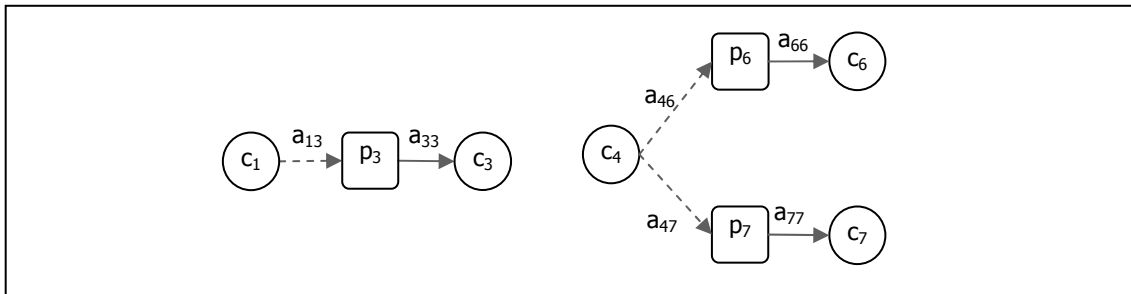


Fig. 10 Ready subgraph H_5 .

7. Related Work

In this section, we explore work related to the results reported in this paper in five areas: workflow specification languages, e-Science middleware, data warehouse systems, data stream management systems, and parallel and multi-query optimization.

The workflow model described in Section 3 bears some similarity with the Pipeline workflow language [Walsh and Maler 2002] in that both define the execution control flow by the data flow. However, the Pipeline language allows cyclic data flows, which our model forbids, since we are interested in estimating, a priori, the data volumes that processes generate. Our model is also simpler than other workflow languages, such as BPEL4WS [Andrews et al. 2003], since it does not provide sophisticated process composition constructs [van der Aalst et al. 2000], but it rather concentrates on the essential characteristics of processes that pipelining scheduling may explore. When compared with OWL-S [Martin et al. 2004], our model can also be viewed as defining a process ontology, but it introduces just the process properties that pipelining can take advantage of, which in fact OWL-S lacks.

Examples of e-Science [De Roure and Hendler 2004] middleware that incorporates some form of workflow support are METEOR [Hall 2003], Kepler [Altintas 2004], myGrid [Wroe 2004], Proteus [Cannataro 2004] and the Discovery Net [Rowe 2003]. In spite of their usefulness, these systems do not automatically optimize workflow execution. For example, Proteus is an environment that facilitates creating and executing Bioinformatics applications [Kanehisa 2003]. However, it only helps users to manually parallelize BLAST processes [Altschul 1990], one of the most popular applications in Bioinformatics, by: (1) splitting the input dataset S into subsets S_1, \dots, S_n ; (2) running BLAST against each S_i in parallel; and (3) combining the partial results into a single output.

Such systems do not attempt to optimize workflow execution mostly because the semantics of application programs is unknown, or difficult to take into account. By contrast, the model introduced in Section 3 only requires an indication of how application programs read and write data. Therefore, it imposes minimal requirements on existing application or middleware software to benefit from the strategies described in Section 5.

Turning to data warehouse systems, pipelining has been extensively explored as a strategy to optimize Extraction-Transformation-Loading (ETL) processes, modeled as workflows.

Vassiliadis et al. (2005) use data-centric workflows, where the output of a certain activity can either be stored persistently or passed to a subsequent activity, and employ a declarative database programming language, LDL, to define the semantics of each activity. Although fairly sophisticated, their model does not cover process characteristics that lead to automated pipelining definition, as we explore in this paper.

Simitsis et al. (2005) also model ETL processes as workflows, quite similarly to our approach. They address the problem of workflow optimization by generalizing the approach taken for relational query optimization. Among the transformations, they consider pipelining two activities. However, they do not consider space limitations, as we do.

Karakasidis et al. (2005) propose a framework for active data warehousing, and implement ETL activities over queue networks. They also adopt a model similar to ours, which pipelines data between activities, but they focus on performance and tuning issues, whereas we address space limitations.

Adzic and Fiore (2003) describe a data warehouse population platform that features an in-memory database, used by data transformation processes. The loading module pipelines read, transform and write activities through buffer pools to improve performance.

In another direction, data stream management systems also make extensive use of pipelining [Babcock et al. 2002; Abadi et al. 2003; Chandrasekaran et al. 2003]. Such systems support the continuous queries that are typical of data stream applications and adopt a workflow model similar to ours.

Arasu et al. (2006) describe CQL, a continuous query language for the STREAM prototype data stream management system [Arasu et al. 2003], that features time-varying relations, which are akin to the data structures we introduce to model the

passing of data items from one process to the next. Each query plan operator reads from one or more input queues, processes the input based on its semantics, and writes its output to an output queue [Babcock et al. 2002; Arasu et al. 2006]. Currently queues are stored entirely in a common pool of main memory.

Babcock et al. (2003) studied the problem of operator scheduling in data stream systems, with the goal of minimizing memory requirements for buffering tuples. Query execution is modeled as a data flow diagram, as in this paper, where nodes are pipelined operators that process tuples, and edges represent composition of operators. They assume that all operators execute in a streaming or pipelined manner. They introduce the Chain scheduling strategy and investigate in detail memory consumption for the case of single-stream queries with selections, projections, and foreign key joins with static stored relations. They also showed that Chain scheduling performs well for other types of queries, including multiple-stream queries, but do not formally investigate memory consumption.

Babu et al. (2004) consider the problem of pipelined filters, where a continuous stream of tuples is processed by a set of commutative filters. Pipelined filters are common in stream applications and capture a large class of multiway stream joins. The authors focus on the problem of ordering the filters adaptively to minimize processing cost in an environment where stream and filter characteristics vary unpredictably over time.

Babu et al. (2005) expand the discussion in Babu et al. (2004) and address the problem of executing continuous multiway join queries in unpredictable and volatile environments. The authors focus on the problem of adaptive placement and removal of caches to optimize join performance, and provide algorithms that estimate cache benefit and cost online. A cache is understood as an associative store that is used to process a contiguous segment of join operators in a pipeline.

Munagala et al. (2004) consider the problem of determining the optimal order in which to evaluate a set of selections, where data is pipelined from one selection to the next. The authors present efficient approximation algorithms for this NP-Hard problem.

Jeffery et al. (2006) describe the Extensible Sensor stream Processing (ESP), a framework for building sensor data cleaning infrastructures. In their approach, stream processing consists of a programmable pipeline of cleaning stages intended to operate on-the-fly as sensor data are streamed through the system. The pipeline in ESP is just a straightforward linear composition of cleaning stages.

Chandrasekaran et al. (2003) describe how query processing is performed in TelegraphCQ by routing tuples through query modules. These modules are pipelined, non-blocking versions of standard relational operators. They present the system architecture, but do not address storage limitations issues.

Viglas and Naughton (2002) propose a rate-based optimization strategy as a way to optimize select-project-join queries. Their query execution model assumes that data streams flow between the relational operators through pipelines.

Finally, pipelining has always been at the core of parallel and multi-query optimization algorithms. We selected just a limited number of references to illustrate how pipelining is used.

Liu and Rundensteiner (2005) investigate how best to combine pipelined parallelism with alternate forms of parallelism to achieve an overall effective processing strategy for multi-join queries. The authors assume that the aggregated memory of all available machines is sufficient to hold the intermediate results of the join relations. In situations when main memory is not enough to hold all join states at the same time, they divide the query into several pieces with each piece being processed sequentially. They adopt two cost functions: total work and total processing time. They describe a segmented parallel processing strategy for complex multi-join queries that balances independent parallelism, pipelined parallelism and partitioned parallelism.

Cruanes, Dageville, and Ghosh (2004) summarize the optimization of parallel SQL queries in the Oracle 10g database. They elaborate on the use of pipelining and data partition and illustrate how the Oracle parallel execution engine is used to transparently parallelize ETL processes.

Dalvi et al. (2001) incorporate pipelining in multi-query optimization. They first define a model for describing a valid pipeline schedule, i.e., a schedule that can be executed without materializing any edge marked as pipelined and using limited buffer space. Next, they prove that the validity of a pipeline schedule can be checked in polynomial time, and that the problem of finding least cost schedules is NP-hard, adopting a cost function that counts the total number of read and write operations. Finally, they present an exhaustive search algorithm and a polynomial time greedy algorithm for finding the least cost pipeline schedule.

In general, research in the last three areas heavily depends on the semantics of the query operators. This sharply contrasts with the model introduced in section 3 of this paper, which requires no knowledge of the application programs, except how they read and write data. Furthermore, no reference in the last three areas addresses space limitations with the level of detail described in Section 5, which is essential to deciding when to pipeline or to materialize intermediate results. Furthermore, the adoption of a workflow model based on a bipartite graph, with process nodes and container nodes, was a simple decision that enhanced the expressiveness of the model and facilitated the development of the results in Section 5.

8. Conclusions

We explored how process pipeline scheduling may become a viable strategy for executing workflows [Lemos 2004a, 2004b]. In special, we proved that the Process Pipeline Scheduling problem is NP-Complete and outlined a greedy process scheduling algorithm that has acceptable performance.

Process pipeline scheduling has three interesting characteristics. First, it optimizes runtime space, which is important when the size of intermediate datasets is large, as in e-Science applications. Second, it improves parallelism in distributed systems or in centralized systems with multi-threading. Finally, through pipelining, the application can make partial results available to the users faster than otherwise, thereby helping users monitor workflow execution.

Our choice of runtime space as the cost function just reflects the idea that pipelining is essentially a strategy to avoid the materialization of intermediate results. However, by

changing the cost estimation functions of the workflow model, the results in this paper can be adapted to cost functions that capture other forms of resource consumption.

Finally, the results are not specific to any application domain, and only depend on an application model that captures properties relevant to deciding when to apply pipelining, i.e., how application programs read and write data. Therefore, the strategies described in Section 5 impose minimal requirements to be incorporated into existing middleware software. The implementation sketched in Section 5.8 is a step in this direction.

Acknowledgements

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 algorithms described in the paper.

References

- Abadi, D.; Carney, D.; Cetintemel, U.; Cherniack, M.; Convey, C.; Erwin, C.; Galvez, E.; Hatoun, M.; Maskey, A.; Rasin, A.; Singer, A.; Stonebraker, M.; Tatbul, N.; Xing, Y.; Yan, R.; Zdonik, S. (2003) "Aurora: a data stream management system". In: Proc. of the 2003 ACM SIGMOD International Conference on Management of Data, June 09-12, 2003, San Diego, California.
- Adzic, J.; Fiore, V. (2003) "Data Warehouse Population Platform". In: Proc. 5th Intl. Workshop on the Design and Management of Data Warehouses (DMDW'03), Berlin, Germany, September 2003.
- Altintas, I.; et. al. (2004). "Kepler: An Extensible System for Design and Execution of Scientific Workflows", Proc. 16th Int'l Conf. on Scientific and Statistical Database Management.
- 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.
- Andrews, T. et al. Business Process Execution Language for Web Services Version 1.1, 5 May 2003. Available at: <ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf>
- Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: (2003) STREAM: The Stanford Stream Data Manager. In: Proceedings of the 2003 ACM SIGMOD Int'l Conf. on Management of Data, p. 665 (2003)
- Arasu, A.; Babu, S.; Widom, J. (2006) "The CQL continuous query language: semantic foundations and query execution". The VLDB Journal — The International Journal on Very Large Data Bases, v.15 n.2, p.121-142, June 2006
- Babcock, B.; Babu, S.; Datar, M.; Motwani, R.; Widom, J. (2002) "Models and issues in data stream systems". In: Proc. 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 03-05, 2002, Madison, Wisconsin.
- Babcock, B.; Babu, S.; Datar, M.; Motwani, R. (2003) "Chain: Operator Scheduling for Memory Minimization in Data Stream Systems". In: Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of data, San Diego, California, pp 253 - 264

- Babu, S., Motwani, R., Munagalat, K., Nishizawa, I., Widom, J. (2004) "Adaptive ordering of pipelined stream filters". In: Proceedings of the 2004 ACM SIGMOD Int'l Conf. on Management of Data, pp. 407–418.
- Babu, S.; Munagalat, K.; Widom, J.; Motwani, R. (2005) "Adaptive caching for continuous queries". In: Proc. 21st International Conference on Data Engineering – ICDE 2005. 5-8 April 2005, pp. 18 – 129.
- Cannataro, M.; et al. (2004). "Proteus, a Grid based Problem Solving Environment for Bioinformatics: Architecture and Experiments", IEEE Computational Intelligence Bulletin, vol. 3, no. 1, pp. 7–18.
- Chandrasekaran, S. et al.. (2003) "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World". In: CIDR. 2003.
- Cruanes, T.; Dageville, B.; Ghosh, B. (2004) "Parallel SQL execution in Oracle 10g". In: Proc. 2004 ACM SIGMOD Int'l Conf. on Management of Data. Paris, France, pp. 850 – 854.
- Dalvi, N.N.; Sanghai, S.K.; Roy, P.; Sudarshan, S. (2001) "Pipelining in multi-query optimization". In: Proc. 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. Santa Barbara, California, United States. pp. 59–70.
- De Roure, D.; Hendler, J.A. (2004) "E-Science: The Grid and the Semantic Web," *IEEE Intelligent Systems*, vol. 19, no. 1, pp. 65-71, Jan/Feb, 2004.
- Garcia-Molina, H.; Ullman, J.D.; Widom, J. (1999). *Database system implementation*. Prentice Hall, New York.
- Garey, M. and Johnson, D. (1979). *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.
- Hall, D.; et. al. (2003) "Using Workflow to Build an Information Management System for a Geographically Distributed Genome Sequence Initiative", *Genomics of Plants and Fungi*. In Prade, R.A. and Bohnert, H.J. (eds): Marcel Dekker, Inc., New York, NY, pp. 359–371.
- 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.
- Jeffery, S.R.; Alonso, G.; Franklin, M.J.; Hong, W.; Widom, J. (2006) "Declarative Support for Sensor Data Cleaning". In: Proc. of the 4th Pervasive Conf, 2006.
- Kanehisa, M., Bork, P. (2003) "Bioinformatics in the post-sequence era". *Nature Genetics* 33, pp. 305–310. Available at: <http://www.nature.com/cgi-taf/DynaPage.taf?file=/ng/journal/v33/n3s/full/ng1109.html>.
- Lemos, M. (2004a). "Workflow for Bioinformatics", Doctoral Dissertation, Dept. Informatics, PUC-Rio (in Portuguese).
- Lemos, M.; Casanova, M.A.; Seibel, L.F.B.; Macedo, J.A.; Basílio, A. (2004b) Ontology-Driven Workflow Management for Biosequence Processing Systems. In: Proc. 15th Int'l Conf. on Database and Expert Systems Applications - DEXA '04, Zaragoza, Spain, pp. 781–790.
- 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. (2007) Bioinformatics Service Domain. Technical Report MCC 07-3. Dept. of Informatics, PUC-Rio.

- Liu, B.; Rundensteiner, E.A. (2005) "Revisiting pipelined parallelism in multi-join query processing". In: Proc. 31st international conference on Very large data bases, Trondheim, Norway, pp. 829–840.
- Martin, D. (Ed.) (2004) "OWL-S: Semantic Markup for Web Services", W3C Member Submission, 22 Nov. 2004. Available at: <http://www.w3.org/Submission/OWL-S>
- Maximillien, E.M.; Singh, M.P. (2004) A Framework and Ontology for Dynamic Web Services Selection, IEEE Internet Computing.
- Munagala, K.; Babu, S.; Motwani, R.; Widom, J. (2004) "The Pipelined Set Cover Problem". In: Proc. 10th Int' Conf. Database Theory - ICDT 2005, Edinburgh, UK, January 5-7, 2005. LNCS, Vol. 3363/2004. Springer Berlin / Heidelberg.
- NCBI (2006). "NCBI FTP – NR Database", <ftp://ftp.ncbi.nlm.nih.gov/blast/db/>.
- Rajasekaran, S., Horowitz, E., Sahni, S. (1996). *Computer Algorithms C+: C++ and Pseudocode Versions*. W. H. Freeman.
- Rosa, R.A.P. (2006) "Rhodnius Prolixus Workflow Parallelization through a Service Domain Architecture". Final Graduation Project, Dept. Informatics, PUC-Rio (in Portuguese).
- Rowe, A.; et. al. (2003). "The Discovery Net System for High Throughput Bioinformatics", *Bioinformatics*, vol. 19, suppl. 1, pp. i225–i231.
- Silva, R.C. (2006) "Bioinformatics Workflow Otimization using Pipelining". Final Graduation Project, Dept. Informatics, PUC-Rio (in Portuguese).
- Simitsis, A.; Vassiliadis, P.; Sellis, T. (2005) "Optimizing ETL processes in data warehouses". Proc. 21th Int'l Conf. on Data Engineering (ICDE 2005). 5-8 April 2005, pp. 564- 575.
- 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>
- Theysa, M.D.; Ali, S.; Siegel, H.J.; Chandy, M.; Hwang, K.; Kennedy, K.; Sha, L.; Shin, K.G.; Snir, M.; Snyder, L.; Sterling, T. (2001) "What Are the Top Ten Most Influential Parallel and Distributed Processing Concepts of the Past Millenium?". Panel Report held at the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000). *Journal of Parallel and Distributed Computing* Volume 61, Issue 12, December 2001, pp. 1827-1841.
- van der Aalst, W. M. P.; Barros, A. P.; Hofstede, A. H. M.; Kiepuszewski, B. (2000) "Advanced workflow patterns." In: Conf. on Coop. Information Systems, pp. 18–29.
- Vassiliadis, P.; Simitsis, A.; Georgantas, P.; Terrovitis, M.; Skiadopoulos, S. (2005) "A generic and customizable framework for the design of ETL scenarios", *Information Systems*, v.30 n.7, pp.492-525, November 2005.
- Viglas, S.; Naughton, J. (2002) "Rate-based query optimization for streaming information sources". In: Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data, June 2002.
- Walsh, N.; Maler, E. (Ed.) (2002) "XML Pipeline Definition Language Version 1.0", W3C Note 28 February 2002. Available at: <http://www.w3.org/TR/xml-pipeline/>.
- Wroe, C.; et. al (2004). "Automating Experiments Using Semantic Data on a Bioinformatics Grid", IEEE Intelligent Systems Special Issue on e-Science.

Yu, J. Buyya, R. (2005) “A taxonomy of scientific workflow systems for grid computing”. ACM SIGMOD Vol. 34, Issue 3, pp. 44–49 (Sept. 2005).

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

Appendix 1 - NP-Complete of the Process Pipeline Scheduling Problem

Theorem 1: The Process Pipeline Scheduling problem is NP-Complete.

Proof. We first show that the Process Pipeline Scheduling problem is in NP. Let PPS be an instance of the problem, characterized by a finite set P , a partial order \ll over P , a function $r:P \rightarrow \mathbb{Z}^+$ and positive integers B and K . A non-deterministic algorithm to answer the question “Is there a process schedule $P' \subseteq P$ such that, for any $p, p' \in P$, if $p \in P'$ and $p' \ll p$, then $p' \in P'$, and $\sum_{p \in P'} r(p) \leq B$ and $|P'| \geq K$?” would simply guess a subset P' of P and test if P' is a solution to PPS, the problem instance, which can be done in polynomial time.

We now show that the Process Pipeline Scheduling problem is NP-Complete by transformation from the Partially Ordered Knapsack problem.

Let POK be an instance of the Partially Ordered Knapsack problem, characterized by a finite set $U = \{u_1, \dots, u_m\}$, a partial order $<$ over U , functions $s:U \rightarrow \mathbb{Z}^+$ and $v:U \rightarrow \mathbb{Z}^+$, and positive integers B and K .

Construct an instance PPS of the Process Pipeline Scheduling problem as follows. The set P of processes of PPS is such that:

(1) for each $u_i \in U$, P contains processes $\pi[i,1], \dots, \pi[i,n(i)]$, where $n(i) = v(u_i)$

We say that $P_i = \{\pi[i,1], \dots, \pi[i,n(i)]\}$ is the i^{th} process group, $\pi[i,1]$ is the master process of P_i and $\pi[i,2], \dots, \pi[i,n(i)]$ are the secondary processes of P_i . Note that the set of secondary processes of a group is empty, if $v(u_i) = 1$.

The partial order \ll of PPS is such that, for each $i, j \in [1, m]$, with $i \neq j$, we have:

(2a) $\pi[i,n(i)] \ll \pi[j,1]$ iff $u_i < u_j$

(2b) if $n(i) > 1$, for each $k \in [2, n(i)]$, we have $\pi[i,k-1] \ll \pi[i,k]$

Condition (2a) says that the output of the last process in the i^{th} group can be pipelined to the first process of the j^{th} group iff $u_i < u_j$. Condition (2b) says that the processes in the i^{th} group can be scheduled in pipeline.

Let N be a positive integer such that

(3a) $N > \sum_{u \in U} v(u)$

(3b) $N > m$

That is, N is greater than or equal to the total sum of the values and also greater than or equal to m , the number of elements in U .

Construct function $r:P \rightarrow Z^+$ of PPS as follows:

$$(4a) \quad r(\pi[i,1]) = N.s(u_i) - v(u_i) + 1 \quad \text{for each } i \in [1,m]$$

$$(4b) \quad r(\pi[i,k]) = 1 \quad \text{for each } i \in [1,m] \text{ and } k \in [2,n(i)], \text{ if } n(i) > 1$$

Note that, since $N > \sum_{u \in U} v(u)$ and $s(u_i) > 0$, we have that $r(\pi[i,1]) = N.s(u_i) - v(u_i) + 1 > N.s(u_i) - N + 1 = N.(s(u_i) - 1) + 1 > 0$. Hence, $r(\pi[i,1]) > 0$, which together with (4b) implies that r is well-defined.

Finally, define:

$$(5a) \quad R = N.B$$

$$(5b) \quad L = K$$

Before proceeding, we establish two properties of process groups:

$$(6a) \quad |P_i| = v(u_i)$$

$$(6b) \quad \sum_{p \in P_i} r(p) = N.s(u_i)$$

Equation (6a) says that the number of processes in P_i is $v(u_i)$. Equation (6b) says that the total number of resource units consumed by processes in P_i is $N.s(u_i)$. Equation (6a) follows from (1) and equation (6b) follows from (1) and (4a), (4b):

$$(6c) \quad \begin{aligned} \sum_{p \in P_i} r(p) &= r(\pi[i,1]) + r(\pi[i,2]) + \dots + r(\pi[i,n(i)]) \quad \text{by definition of } P_i \\ &= (N.s(u_i) - v(u_i) + 1) + (v(u_i) - 1) = N.s(u_i) \quad \text{by (4a), (4b) and (1)} \end{aligned}$$

We now show that POK has a solution iff PPS does. Suppose first that POK has a solution. Then, there is $U' \subseteq U$ such that:

$$(7a) \quad \text{for any } u, u' \in U, \text{ if } u \in U' \text{ and } u' < u \text{ then } u' \in U'$$

$$(7b) \quad \sum_{u \in U'} s(u) \leq B$$

$$(7c) \quad \sum_{u \in U'} v(u) \geq K$$

Construct P' such that P' contains all processes in the i^{th} process group iff $u_i \in U'$:

$$(8) \quad P' = \bigcup_{u_i \in U'} P_i$$

We have to prove that:

$$(9a) \quad \sum_{p \in P'} r(p) \leq R$$

$$(9b) \quad |P'| \geq L$$

$$(9c) \quad \text{for any } p, p' \in P, \text{ if } p \in P' \text{ and } p' < p, \text{ then } p' \in P'$$

We first prove (9a):

$$(10) \quad \begin{aligned} \sum_{p \in P'} r(p) &= \sum_{p \in P_i \wedge u_i \in U'} r(p) && \text{by (8)} \\ &= \sum_{u_i \in U'} \sum_{p \in P_i} r(p) && \text{by distributing the summation} \\ &= \sum_{u_i \in U'} N.s(u_i) && \text{by (6b)} \\ &= N \cdot \sum_{u_i \in U'} s(u_i) \end{aligned}$$

$$\leq N.B = R$$

by (7b) and (5a)

We now prove (8b):

$$(11) |P'| = \left| \bigcup_{u_i \in U'} P_i \right| = \sum_{u_i \in U'} v(u_i) \geq K = L \quad \text{by (8), (6a), (7c) and (5b)}$$

Finally, we prove (9c). Let $p \in P'$ and $p' \in P$. Assume that $p' \ll p$. We have to show that $p' \in P'$. Since $p' \ll p$, by (2a) and (2b), there are two cases to consider:

Case 1: There are $i, j \in [1, m]$, with $i \neq j$, such that $p' = \pi[i, n(i)]$ and $p = \pi[j, 1]$ and $u_i < u_j$.

Since, by assumption, $p = \pi[j, 1] \in P'$, we have that $u_j \in U'$, by construction of P' . Now, since $u_i < u_j$ and $u_j \in U'$, by (6a), $u_i \in U'$. Therefore, $p' = \pi[i, n(i)] \in P'$, again by construction of P' .

Case 2: There is $i \in [1, m]$ and $k \in [2, n(i)]$ such that $p' = \pi[i, k-1]$ and $p = \pi[i, k]$.

Since, by assumption, $p = \pi[i, k] \in P'$, we have that a process of the i^{th} group is in P' . So, by construction of P' , all processes of the i^{th} group are in P' . Thus, $p' = \pi[i, k-1] \in P'$.

We have established that (8a), (8b) and (8c) holds. Therefore, P' is a solution for PPS.

Suppose now that PPS has a solution. Then, there is $P' \subseteq P$ such that:

$$(12a) \quad \text{for any } p, p' \in P, \text{ if } p \in P' \text{ and } p' \ll p, \text{ then } p' \in P'$$

$$(12b) \quad \sum_{p \in P'} r(p) \leq R$$

$$(12c) \quad |P'| \geq L$$

We first prove that we may create a second solution P'' that contains all processes in a group iff P' contains the master process of the group. That is, P'' contains only complete groups. Indeed, define P'' as follows:

$$(13) \quad P'' = \bigcup_{\pi[i, 1] \in P'} P_i$$

We have to prove that P'' is a solution to PPS, that is, that P'' must satisfy:

$$(14a) \quad \text{for any } p, p' \in P, \text{ if } p \in P'' \text{ and } p' \ll p, \text{ then } p' \in P''$$

$$(14b) \quad \sum_{p \in P''} r(p) \leq R$$

$$(14c) \quad |P''| \geq L$$

Indeed, by construction of P'' , (2b) and (12a) imply (14a). Moreover, since P'' has more processes than P' , then (12c) implies (14c). It remains to prove (14b). First observe that

$$(15a) \quad P'' - P' = \{ p \in P / \pi[i, 1] \in P' \wedge p \in P_i \wedge p \notin P' \}$$

$$(15b) \quad P'' - P' \text{ contains only secondary processes}$$

Indeed, $P'' - P'$ is the set of secondary processes p in P_i such that the master process of P_i is in P' but p itself is not in P' .

First observe that we have:

$$(16) \quad |P'' - P'| \leq \sum_{\pi[i,1] \in P'} v(u_i) \quad \text{by (15) and (6a)}$$

By (4a), if p is a secondary process, $r(p)=1$. Moreover, by (15b), $P''-P'$ contains only secondary processes. Hence, we have:

$$(17) \quad \sum_{p \in P'' - P'} r(p) \leq \sum_{\pi[i,1] \in P'} |P_i| = \sum_{\pi[i,1] \in P'} v(u_i) \quad \text{by (6a)}$$

Therefore, we have:

$$(18) \quad \begin{aligned} \sum_{p \in P''} r(p) &= \sum_{p \in P'} r(p) + \sum_{p \in P'' - P'} r(p) \\ &\leq R + \sum_{\pi[i,1] \in P'} v(u_i) \quad \text{by (12b) and (17)} \\ &< N.B + N \quad \text{by (5a) and (3a)} \end{aligned}$$

Now, since P'' contains complete groups, we also have:

$$(19) \quad \sum_{p \in P''} r(p) = N \cdot \sum_{\pi[i,1] \in P''} s(u_i) \quad \text{by (6b)}$$

Therefore, we have:

$$(20) \quad N \cdot \sum_{\pi[i,1] \in P''} s(u_i) < N.B + N \quad \text{by (18) and (19)}$$

or

$$(21) \quad \sum_{\pi[i,1] \in P''} s(u_i) < B + 1$$

But, by definition of function $s:U \rightarrow \mathbb{Z}^+$, $\sum_{\pi[i,1] \in P''} s(u_i)$ is a positive integer.

Moreover, B is also a positive integer. Hence, from (21), we have:

$$(21) \quad \sum_{\pi[i,1] \in P''} s(u_i) \leq B$$

Therefore, from (18) and (20), we establish (16b):

$$(22) \quad \sum_{p \in P''} r(p) \leq N.B = R \quad \text{by (19), (21) and (5a)}$$

Therefore, we may assume that the solution P'' for PPS satisfies (14a), (14b) and (14c).

Returning to the main proof, using P'' , we have to prove that POK has a solution. Construct U'' as follows:

$$(23) \quad U'' = \{ u_i \in U / \pi[i,1] \in P'' \}$$

That is, U'' contains u_i iff P'' contains the master process of the i^{th} group (that is, the complete group P_i , by (13)). We prove that:

$$(24a) \quad \sum_{u \in U''} s(u) \leq B$$

$$(24b) \quad \sum_{u \in U''} v(u) \geq K$$

$$(24c) \quad \text{for any } i, j \in [1, m], \text{ with } i \neq j, \text{ if } u_j \in U'' \text{ and } u_i < u_j \text{ then } u_i \in U''$$

We immediately have (24a):

$$(24) \quad \sum_{u \in U''} s(u) = \sum_{\pi[i,1] \in P''} s(u_i) \leq B \quad \text{by (23) and (21)}$$

Since, from (13), P'' contains only complete process groups, by the construction of U'' , we have (24b):

$$(25) \quad \sum_{u_i \in U''} v(u_i) = \sum_{\pi[i,1] \in P''} |P_i| = |P''| \geq L = K \quad \text{by (23), (6a), (14c) and (5b)}$$

To prove (24c), let $i, j \in [1, m]$, with $i \neq j$, and assume that $u_j \in U''$ and $u_i < u_j$. Then, since $u_j \in U''$, by construction of U'' , we have $\pi[j, 1] \in P''$. Since $u_i < u_j$ and $\pi[j, 1] \in P''$, by (2a), $\pi[i, n(i)] \ll \pi[j, 1]$. Therefore, by (16a), $\pi[i, n(i)] \in P''$. But, by (13), P'' contains only complete groups. Therefore, we have $\pi[i, 1] \in P''$, which implies that $u_i \in U''$, by construction of U'' , which establishes (23c). Hence, U'' is a solution for POK.

We have therefore shown that POK has a solution iff PPS does, which concludes the proof that the Process Pipeline Scheduling problem is NP-Complete. \square