

On the Complexity of Process Pipeline Scheduling

Melissa Lemos, Marco Antonio Casanova

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}@inf.puc-rio.br

***Abstract.** This paper explores how process pipeline scheduling may become a viable strategy for executing workflows. It first details a workflow optimization and execution algorithm that reduces runtime space. The optimization strategy pipelines the communication between as many processes as possible, within the bounds of the storage space available, and depends on generic properties of datasets and processes. Then, the paper proves that the process pipeline scheduling problem is NP-Complete. Finally, it presents a greedy process pipeline scheduling algorithm which has a viable performance.*

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. 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.

This paper addresses *process pipeline scheduling*, a strategy to execute workflows that optimizes runtime storage requirements and improves parallelism. 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 that p writes, 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 pipelining strategy can be applied to more than two processes, but it is limited by the amount of resources available, such as disk space.

Pipelining is extensively used in relational query optimization to avoid the materialization of intermediate results [Garcia-Molina 1999]. In this context, process 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. This paper extends this idea to workflows, exploring when application processes 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 execution algorithm that uses pipelining. The algorithm is not specific to a particular

application domain, but it rather depends on generic properties of datasets and processes, which we capture in an appropriate workflow model. Then, we prove that the process pipeline scheduling problem is NP-Complete. Finally, we present a greedy process pipeline scheduling algorithm that has acceptable performance.

The results described in this paper contribute to the construction of workflow management systems or Grid infrastructure middleware, which allow users to design, execute, re-run and monitor workflows. Examples of such systems from e-Science are METEOR [Hall 2003], Kepler [Altintas 2004], myGrid [Wroe 2004], Proteus [Cannataro 2004] and the Discovery Net [Rowe 2003], to name a few. In spite of their usefulness, these systems do not automatically optimize workflow execution. For example, the Proteus system only helps users manually parallelize a BLAST process 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.

The paper is organized as follows. Section 2 provides motivations to investigate process pipelining. Section 3 describes the workflow model adopted in the paper. Section 4 discusses the process pipeline scheduling algorithm. Section 5 contains the conclusions. The appendix contains a 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 pipelining 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 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 10km×10km). 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_k that maps sets of objects into sets of objects and is defined with the help of a second function g_k such that $f_k(S) = \{ g_k(s) / s \in S \}$. Then, given two such operations f_p and f_q , we can trivially show that:

- (1) $f_p(f_q(S)) = \{ g_p(g_q(s)) / s \in S \}$
- (2) $f_p(f_q(U \cup V)) = f_p(f_q(U)) \cup f_p(f_q(V)) = f_p(f_q(U)) \cup f_p(f_q(V))$

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 the strategy of computing $f_p(f_q(S))$ by partitioning S into smaller sets and then using (2). Of course, the two strategies can be combined into a single one, 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 basic workflow model combines two sub-models. The *application model* captures characteristics of the application domain programs that will be invoked during workflow execution. The *flow model* includes a data flow control abstraction and captures the workflow patterns: *sequence*, for a sequence of processes; *parallel split*, for a single thread of control that splits into multiple threads; and *synchronization*, for a multiple parallel threads that converge into a single thread [van der Aalst et al, 2000].

The application model is formalized as a set of classes with two top classes, *Process* and *Container*, whose instances are called, respectively, *processes* and *containers*.

Let p be an analysis program of the application domain. We assume that p may have named parameters, input datasets and output datasets. The application model must define a subclass $Pr[p]$ of *Process* whose instances model calls to p .

Class $Pr[\mathbf{p}]$ has the following properties:

- for each parameter A of \mathbf{p} , a *parameter* property $Pa[\mathbf{p},A]$ such that, for an instance p in $Pr[\mathbf{p}]$ (a call to \mathbf{p}), $Pa[\mathbf{p},A](p)$ is the value of A in p (the value of A in the call to \mathbf{p} that p represents).

Class $Pr[\mathbf{p}]$ has the following meta-properties (that is, properties of the class itself) that capture how \mathbf{p} reads and writes data:

- *input-ports*, whose value is a list (i_1, \dots, i_m) of names that identify the input datasets that \mathbf{p} consumes. The elements in *input-ports* are called *input ports*;
- *output-ports*, whose value is a list (o_1, \dots, o_n) of names that identify the output datasets that \mathbf{p} produces. The elements in *output-ports* are called *output ports*;
- *input-ports-type*, whose value is a list (t_1, \dots, t_m) such that, for each input port i_k :
 - $t_k = \text{'gradual'}$ iff \mathbf{p} starts to read through input port i_k as soon as data items become available, and that \mathbf{p} reads each data item only once;
 - $t_k = \text{'non-gradual'}$ iff \mathbf{p} starts to read through input port i_k only after all data items are available, and that it may re-read a data item;
- *output-ports-type*, whose value is a list (v_1, \dots, v_n) such that, for each output port o_k :
 - $v_k = \text{'gradual'}$ iff \mathbf{p} makes a data item available for reading through port o_k immediately after writing it, and that it writes a data item only once;
 - $v_k = \text{'non-gradual'}$ iff \mathbf{p} makes data items available for reading through port o_k only after writing all data items, and that it rewrites data items.

We use the notation $type[i_k]$ to indicate the *type* t_k of an input port i_k , and likewise for an output port.

Finally, class $Pr[\mathbf{p}]$ has the additional meta-properties that estimate the volume of data that \mathbf{p} will produce through each output port, given values for the parameter of \mathbf{p} and estimations for the data volumes of the input ports:

- *maximum-data-size*, whose value is a list (s_1, \dots, s_n) of functions such that, for each output port o_k , the function s_k returns an estimation of the maximum size of the output dataset corresponding o_k , given values for the parameters of \mathbf{p} and estimations for the sizes of the input datasets;
- *maximum-item-size*, whose value is a list (r_1, \dots, r_n) of functions such that, for each output port o_k , the function r_k returns an estimation of the maximum size of an item in the output dataset corresponding to o_k , given values for the parameters of \mathbf{p} and estimations for the maximum sizes of an item in each input dataset.

We use the notation $maximum-data-size[o_k]$ and $maximum-item-size[o_k]$ to indicate the *cost function* s_k and r_k of an output port o_k . The role of these functions is entirely similar to cost functions defined for the traditional relational operations, in the context of relational query optimization.

Given an application model, the *flow model* is formalized as a single class *Workflow*. An instance of the class *Workflow* is a labeled bipartite graph, called a *workflow graph* or, simply, a *workflow*, such that (see Figure 1):

- the set of nodes contains processes, called *process nodes*, or containers, called *container nodes*, which are instances of the classes of the application model;
- 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, with two labels, *port* and *connection-type*. The set of arcs satisfy the following property. Let p be a process node and assume that p is an instance of class $Pr[p]$. Then, we have:
 - for each input port \mathbf{i} of p , there must be an arc (c,p) from a container node c to p such that $port((c,p)) = \mathbf{i}$ and $connection-type((c,p)) = type[\mathbf{i}]$;
 - for each output port \mathbf{o} of p , there must be an arc (p,c) from p to a container node c such that $port((p,c)) = \mathbf{o}$ and $connection-type((p,c)) = type[\mathbf{o}]$;
 - these are the only arcs of the workflow definition 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 since arcs are labeled with the names of input or output ports. Assumption (2) simplifies the algorithm described in Section 4, which can however be modified to cover cycles.

Let (c,p) be an arc from a container node c to a process node p . Assume that $port((c,p)) = \mathbf{i}$ and $connection-type((c,p)) = type[\mathbf{i}]$. 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 port \mathbf{i} ; container c is *connected to* input port \mathbf{i} of p ; the arc is a *gradual read connection* iff $type[\mathbf{i}] = \text{'gradual'}$ and a *non-gradual read connection* iff $type[\mathbf{i}] = \text{'non-gradual'}$.

Likewise, let (p,c) be an arc from a process node p and a container node c . Assume that $port((p,c)) = \mathbf{o}$ and $connection-type((p,c)) = type[\mathbf{o}]$. 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* in c through port \mathbf{o} ; container c is *connected to* output port \mathbf{o} of p ; the arc is a *gradual write connection* iff $type[\mathbf{o}] = \text{'gradual'}$ and a *non-gradual write connection* iff $type[\mathbf{o}] = \text{'non-gradual'}$.

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

Finally, a workflow graph is associated with cost estimation functions, as follows. Let c be an input container node of the workflow graph. Then, c corresponds to an input dataset of the workflow graph as a whole. In this case, we define two constants, *maximum-container-data-size(c)* and *maximum-container-item-size(c)*, that give estimations for the size of the input dataset that c represents and for the maximum size of a data item in c .

Let p be a process node and assume that p is an instance of class $Pr[p]$. Then, for each output port \mathbf{o} of p , we already have *maximum-data-size[o]* and *maximum-item-size[o]*. Let c be a container connected to output ports $\mathbf{o}_1, \dots, \mathbf{o}_n$ of processes p_1, \dots, p_n . Then, we define two cost functions for c as follows:

$$maximum-container-data-size(c) = \sum_{i=1, \dots, n} (maximum-data-size[\mathbf{o}_i](\vec{t}))$$

$$\text{maximum-container-item-size}(c) = \max_{i=1, \dots, n}(\text{maximum-item-size}[o_i](\vec{t}))$$

where \vec{t} is the vector of parameter values of $\text{maximum-data-size}[o_i]$, which are the values for the parameters of p in p (recall that p represents a call to p) and estimations for the sizes of the input datasets of p , which are given by the estimations of the containers connected to input ports of p .

Therefore, $\text{maximum-container-data-size}$ and $\text{maximum-container-item-size}$ are iteratively computed starting from the input containers of the workflow graph towards the output container nodes.

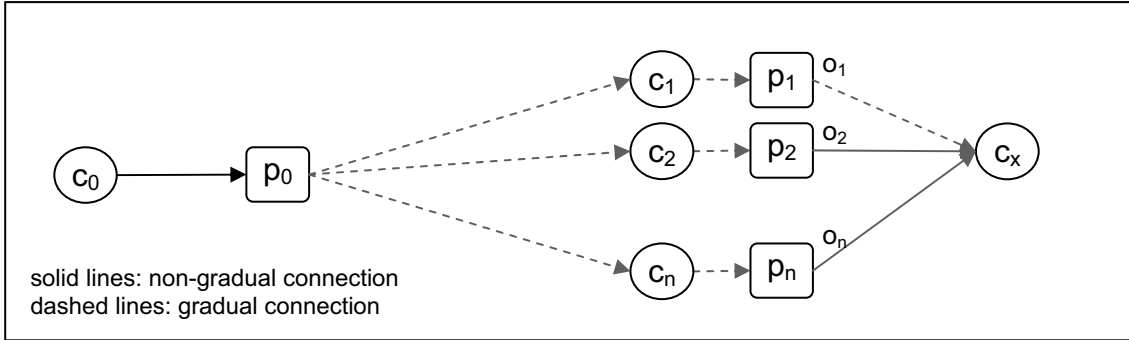


Figure 1: A schematic example of a workflow definition graph.

4 Workflow Execution Algorithm

In this section, we briefly describe a workflow execution algorithm that pipelines process nodes, within the bounds of the 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. In particular, it uses the container size estimations to compute the amount of disk space that must be pre-allocated to each container node.

4.1 Container Implementation

Given the nature of analysis processes and datasets, a container node may potentially occupy considerable space. However, it is possible to reduce the required space by adopting special data structures.

For example, the process scheduling algorithm can choose to implement a container node c using a *LimitedBuffer* when all connection arcs entering and leaving c are gradual. Otherwise, it can decide to use a *File*.

A *LimitedBuffer* is a data structure, with limited space, that can be used to transfer data items from multiple producers to multiple consumers. It offers *Get* and *Put* methods which are similar to those of a queue, except that there is no order among the data items, since this is not essential to capture the data transfer behavior of the processes we intend to model. 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*.

A *File* is a data structure, with unlimited space, that can be used to store data items. It offers *Read*, *Write*, *Reread* and *Rewrite* methods as for a conventional file, in addition

to those of *LimitedBuffers*. Note that a *File* may store data items generated by multiple producers, offering these data items to multiple consumers. Furthermore, note that a consumer may only start reading the file after all producers stop writing data items, since a producer may rewrite a data item.

The container node has a label *size*, initialized with the amount of disk space reserved for the data structure chosen to implement *c* and according to the functions *maximum-container-data-size* and *maximum-container-item-size*. If *c* is a *LimitedBuffer*, *size* is set to *maximum-container-item-size*, which is the size of the largest data item that the container must hold. If *c* is a *File*, *size* is set to *maximum-container-data-size*, which has value equals the total space required for storing all data items that the container must hold (written by all processes).

4.2 Augmented Workflow Graph

The workflow execution algorithm depends on a small variation of the workflow graph introduced in Section 3, where process nodes and arcs have new labels. Since the changes are minor, we will continue to refer to this graph as the workflow graph.

A process node has a label, the *process state* (referred to as a *state*, when there is no ambiguity), with values ‘*initial*’, ‘*pending*’, ‘*executing*’, and ‘*finished*’. When the workflow execution algorithm starts interpreting a workflow graph *w*, each process node *p* in *w* has its state is set to ‘*initial*’ to indicate that *p* has not yet started executing. During execution, the state value changes, as discussed in Section 4.3.

Likewise, a connection arc has a new label, *connection state* (again referred to as a *state*, when there is no ambiguity), with values ‘*idle*’, ‘*open*’ and ‘*closed*’. When the workflow execution algorithm starts interpreting a workflow graph *w*, each connection arc has its state set to ‘*idle*’. During execution, the state value changes, as discussed in Section 4.4.

4.3 Controlling Process Node States

The workflow execution algorithm organizes the processes that can be executed in parallel at any given time also as a graph, created in two phases. The first phase does not take into account disk space and considers only those processes which are ready for execution. We say that a process node *p* is *ready* (for execution) when its state is ‘*initial*’ or ‘*pending*’ and all read arcs that end in *p* are *open*, i.e., all input data that *p* needs are already available. The collection of these process nodes and container nodes, together with the arcs that connected them in the original workflow graph, is called the *execution graph*.

The algorithm modifies the execution graph by pruning nodes until the graph contains only process nodes that can be executed in parallel within the limits of disk space available. A process node *p* that was pruned has to be postponed, which is indicated by changing its state to ‘*pending*’.

The workflow execution algorithm starts each process node *p* in the execution graph (after the pruning step) by setting its state to ‘*executing*’. Also, the state of the connection arcs that touch *p* may change, as described in Section 4.4.

When a process node p finishes executing, the workflow execution algorithm changes its state to *finished*. Then, it identifies container nodes that store temporary data and whose read and write arcs are all in state *closed*. This situation reveals that the disk space allocated to the container can be freed and, hence, the available disk space recomputed. The algorithm then loops back and re-computes the execution graph.

4.4 Controlling Container Node States and Arc States

During process scheduling, the state of an arc changes to *open* to indicate that the arc is *open* for production and consumption, and to *closed*, to indicate the opposite.

When a process node p finishes executing, the state of each arc that connects a container node to p or that connects p to a container node is set to *closed*, as p does not need to produce or consume more data. Moreover, if p is the last process node writing data into a container node c , the state of each non-gradual arc that connects c to a process node is set to *open*.

Suppose that p is a process node and that p starts executing. The state of each arc that connects a container node to p does not change because the arc is already *open*. The state of each arc that connects p to a container node is changed to *open*.

Also, if p is the first process to write data into a container node c , the state of each gradual arc that connects c to a process node q is set to *open*. The fact that the arc is gradual exactly indicates that data may be pipelined from p to q through c , which may reduce storage requirements, depending on the implementation chosen for c . Furthermore, if all other read connections that q uses are also *open*, the program may start executing q as soon as it starts executing p , which improves parallelism. However, this is conditional to the available disk space, as discussed in the previous subsection.

The arc state change policy just described indeed summarizes the central strategy of process pipelining scheduling embedded in the workflow execution algorithm.

4.5 Naïve Process Scheduling Algorithm

Let B be the total amount of disk space available. We say that a sub-graph H of an execution graph G is consistent iff the following conditions hold:

- For any node n of H , all ancestors of n in G are also in H .
- The sinks of H are container nodes.
- If two nodes n and m are in H and there is an arc e in G that connects n and m , then e is also in H .
- The container nodes of H consume less than B units of disk space.

Moreover, we say that a subgraph H of an execution graph G is *optimal* iff H is consistent and H has the largest number of process nodes among all consistent sub-graphs of G . Finding the optimal sub-graph therefore amounts to pipelining as many processes as possible, respecting the amount of disk space available.

A naïve algorithm to decide the optimum choice of processes to schedule would be:

Naïve Process Scheduling

Input:

- G - an execution graph with n nodes
- 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 .
-

We can prove that the Naïve Process Scheduling algorithm is exponential:

Proposition 1: The worst-case running time for the Naïve Process Scheduling algorithm is $O(2^n)$.

Proof. Let G be an execution graph with n nodes. 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 counting the number of process nodes in each subgraph produced by Step 1. \square

4.6 Complexity of the Process Pipeline Scheduling Problem

We can in fact prove that the problem we are trying to solve is NP-Complete. We use $|S|$ to denote the cardinality of set S and Z^+ to denote the set of all positive integers. Recall from [Garey and Johnson, 1978] that the *Partially Ordered Knapsack* problem is defined as follows:

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 $<<$ over P , where $p << p'$ indicates that process p can pipeline output records 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 $p' << p$, 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 scheduling algorithm can schedule in each step by taking advantage of pipelining. P represents the set of all processes that are ready to be fired 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 the Appendix for the proof).

4.7 A Greedy Process Scheduling Algorithm

In view of Theorem 1, we describe a greedy process scheduling heuristics [Rajasekaran 1996, chapter 4] whose worst-case running time is $O(n^2)$.

Greedy Process Scheduling

Input:

- G - an execution graph with n nodes
- B - the total amount of disk space available

Output:

- H - a consistent subgraph of G

1. Proceed from the sink nodes to the source nodes of G.
2. Compute the gain from pruning each sink container node.
3. Order the nodes in decreasing gain.
4. Prune nodes in decreasing gain order until there is enough space on disk to run all ready process nodes.
5. Output the resulting subgraph H, which is consistent by construction.

To compute the gain (in disk space) when a container node c is pruned, we proceed as follows. Let c be a sink container node in the execution graph. When c is pruned, each process node p that produces data in c also has to be pruned, as well as each container node c' such that p produces data into c' , as so, recursively. Hence, when a container node c is pruned, a set of process and container nodes must also be pruned.

Moreover, the pruning process has to re-consider the implementations adopted, and thereby the disk space required, for some containers that were maintained in the execution graph. A container node c falls into this situation when: (1) c was previously implemented by a *LimitedBuffer* and used to pipeline data items from a process p 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 dataset 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 will be the difference between the sum of the sizes of the container nodes that will be pruned and the sum of the extra space required by the data structures that will now implement the remaining container nodes that had their implementation strategy changed.

As a consequence of this discussion, container nodes implemented by a *LimitedBuffer* will tend to have the least gain when pruned. Therefore, the greedy pruning heuristics will preserve, as much as possible, container nodes that participate in a pipeline.

We can prove that the heuristics has acceptable performance:

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

Proof. Let G be an execution graph with n nodes and e arcs. The worst-case running time for each step is as follows. Step 1 is $O(n+e)$ since it requires a topological sort of G (using adjacency lists). Step 2 is $O(n^2)$ since, for each sink container node s , it requires visiting at most n other nodes walking backwards from s , and there are at most n sink container nodes. Step 3 is $O(n^2)$ and results in a list L with at most n nodes. Step 4 is $O(n^2)$ since, for each s in L , it again requires visiting at most n other nodes walking backwards from s , and there are at most n nodes in L . \square

5. 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 may 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.

Moreover, the results are not specific to any application domain, but they rather depend on an application model that captures properties that are relevant to deciding when to apply pipelining.

Finally, as a complement to the theoretical work reported in this paper, we observe that three implementation efforts are under way: (1) to obtain experimental performance data in specific Bioinformatics workflows executed with pipelining; (2) to obtain experimental performance data by simulating a workflow execution engine that incorporates pipelining; (3) to implement a workflow execution engine that incorporates pipelining. The last two efforts use the workflow language described in [Lemos 2004a].

Acknowledgements

This work was partially supported by FAPERJ for Melissa Lemos and by CNPq under grant 551241/05-5 for Marco Antonio Casanova.

References

- 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.
- Cannataro, M.; et al. (2004). “Proteus, a Grid based Problem Solving Environment for Bioinformatics: Architecture and Experiments”, IEEE Computational Intelligence Bulletin, v.3, n.1, 7-18.
- Garey, M. and Johnson, D. (1978). *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco.
- 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 359-371.
- Garcia-Molina, H.; Ullman, J.D.; Widow, J. (1999). *Database system implementation*. Prentice Hall, New York.
- Lemos, M. (2004a). “Workflow para Bioinformática”, Doctoral Dissertation, Dept. Informatics, PUC-Rio.
- Lemos, M.; et. al. (2004b). “Ontology-Driven Workflow Management for Biosequence”, Proc. 15th Int’l. Conf. on Database and Expert Systems Applications - DEXA '04, Zaragoza, Spain, 781-790.
- 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.
- Rowe, A.; et. al. (2003). “The Discovery Net System for High Throughput Bioinformatics”, Bioinformatics, Vol. 19, suppl. 1, i225–i231.
- 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, p. 18–29.
- Wroe, C.; et. al (2004). “Automating Experiments Using Semantic Data on a Bioinformatics Grid”, IEEE Intelligent Systems Special Issue on e-Science.

Appendix 1

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 $r:P \rightarrow \mathbb{Z}^+$ of PPS as follows:

- (4a) $r(\pi[i, 1]) = N.s(u_i) - v(u_i) + 1$ for each $i \in [1, m]$
(4b) $r(\pi[i, k]) = 1$ for each $i \in [1, m]$ and $k \in [2, n(i)]$, 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) $R = N.B$
(5b) $L = K$

Before proceeding, we establish two properties of process groups:

- (6a) $|P_i| = v(u_i)$
(6b) $\sum_{p \in P_i} r(p) = N.s(u_i)$

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) $\sum_{p \in P_i} r(p) = r(\pi[i, 1]) + r(\pi[i, 2]) + \dots + r(\pi[i, n(i)])$ by definition of P_i

$$= (N.s(u_i) - v(u_i) + 1) + (v(u_i) - 1) = N.s(u_i) \quad \text{by (4a), (4b) and (1)}$$

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):

$$\begin{aligned} (10) \quad \sum_{p \in P'} r(p) &= \sum_{p \in P_i \wedge u_i \in U'} r(p) && \text{by (8)} \\ &= \sum_{u_i \in U'} \sum_{p \in P_i} r(p) && \text{by distributing the summation} \\ &= \sum_{u_i \in U'} N \cdot s(u_i) && \text{by (6b)} \\ &= N \cdot \sum_{u_i \in U'} s(u_i) \\ &\leq N \cdot B = R && \text{by (7b) and (5a)} \end{aligned}$$

We now prove (8b):

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

Finally, we prove (9c). Let $p \in P'$ and $p' \in P$. Assume that $p' << p$. We have to show that $p' \in P'$. Since $p' << p$, 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.

The proof that, if PPS has a solution, then POK also has a solution follows likewise. Therefore, we have that POK has a solution iff PPS does, which establishes that the Process Pipeline Scheduling problem is NP-Complete. \square