

Optimizing Bioinformatics Workflow Execution Through Pipelining Techniques

Melissa Lemos^{1*}, Luiz Fernando B.Seibel¹, Antonio Basílio de Miranda², Marco Antonio Casanova¹

¹Department of Informatics, PUC-Rio, Rio de Janeiro, Brazil

²Laboratory for Functional Genomics and Bioinformatics, Department of Biochemistry and Molecular Biology, Oswaldo Cruz Institute, Fiocruz, Rio de Janeiro, Brazil

*Corresponding author

Email addresses:

ML: melissa@inf.puc-rio.br

LFBS: seibel@inf.puc-rio.br

ABM: antonio@fiocruz.br

MAC: casanova@inf.puc-rio.br

Abstract

Background

Genome projects usually start with a sequencing phase, where experimental data, usually DNA sequences, is generated, without any biological interpretation. The fundamental challenge researchers face lies exactly in analyzing these sequences to derive information that is biologically relevant. During the analysis phase, researchers use a variety of analysis programs typically combined, in the sense that the data collection one program produces is used as input to another program. This is best modelled as a workflow whose steps are the analysis programs.

Results

This paper addresses the use of workflows to compose Bioinformatics analysis programs that access data sources, thereby facilitating the analysis phase.

An ontology modelling the analysis program and data sources commonly used in Bioinformatics is first described. This ontology is derived from a careful study of the computational resources that researchers in Bioinformatics presently use.

Then, a software architecture for biosequence analysis management systems is described. The architecture has two major components. The first component is a Bioinformatics workflow management system that helps researchers define, validate, optimize and run workflows combining Bioinformatics analysis programs. The second component is a Bioinformatics data management system that helps researchers manage large volumes of Bioinformatics data. The architecture includes an ontology manager that stores Bioinformatics ontologies, such as that previously described.

The major contribution of the paper is the specification of a workflow optimization and execution component 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 the process ontology previously defined. The strategy has variations that cover both centralized and distributed environments.

Conclusions

The workflow optimization strategy reduces runtime storage requirements, improves parallelism in a distributed environment or in a centralized system with multiple processors, and may make partial results available to the users sooner than otherwise, thereby helping users monitor workflow execution.

Background

Computer-intensive scientific investigation, such as Genome projects, often involves applying off-the-shelf analysis processes to pre-existing datasets.

Genome projects usually start with a sequencing phase, where experimental data, usually DNA sequences, is generated, without any biological interpretation. DNA sequences have codes which are responsible for the production of protein and RNA sequences, while protein sequences participate in all biological phenomena, such as cell replication, energy production, immunological defence, muscular contraction, neurological activity and reproduction. DNA, RNA and protein sequences will be called biosequences in this paper.

The fundamental challenge researchers face lies exactly in analyzing these sequences to derive information that is biologically relevant.

During the analysis phase, researchers use a variety of analysis programs and access large data sources holding Molecular Biology data. The growing number of data sources and analysis programs indeed enormously facilitated the analysis phase. However, it creates a demand for systems that facilitate using such computational resources.

The analysis programs are typically combined, in the sense that the data collection one program produces is used as input to another program. This is best modelled as a workflow, or script, whose steps are the analysis programs.

Indeed, several packages already offer predefined workflows, whose steps are parameterized calls to analysis programs. For example, the Phred/Phrap [1] package has a predefined workflow that, given chromatograms (DNA sequencing trace files, obtained in laboratory), calls all programs that generate reads and contigs. For example, this workflow calls a program, called Crossmatch, which compares each read with a vector library and, if a vector is found, removes the vector (or part of it) from the read.

In addition to predefined workflows, packages also offer generic workflow (or script) languages. However, workflow definition involves, among other non-trivial tasks, the configuration of various parameters each analysis program depends on. Workflow maintenance and reuse is also often difficult.

Therefore, rather than packages of analysis programs, researchers would be better served by integrated systems that fully support the biosequences analysis phase. We will call such systems Bioinformatics data analysis and management systems (BioDAMS).

Given this scenario, we first introduce in this paper a reference architecture for BioDAMS. The architecture includes a knowledge base that stores process ontologies, capturing the generic properties of the analysis programs and datasets of Bioinformatics domain.

The core of this paper describes a workflow monitor, operating in the context of a BioDAMS. The monitor includes a pipelining strategy to optimize runtime storage requirements and to improve parallelism. The strategy explores generic properties of datasets and analysis programs, as specified in a process ontology.

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 workflow monitor 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 monitor will only start q after p finishes. This pipelining strategy can be applied to more than two processes, but it is limited to the available disk space. Indeed, the workflow monitor we propose tries to pipeline the communication between as many processes as possible, within the bounds of the disk space available.

The work described in this paper is best compared to scientific workflow management systems or Grid infrastructure middleware, such as METEOR [2], Kepler [3], myGrid [4], Proteus [5], Discovery Net [6], and WASA [7] to name a few. All these projects allow a researcher to design, execute, re-run and monitor scientific workflows. Also,

most of these systems feature an architecture based on services and support the management of scientific experiments in domain of Bioinformatics.

The optimization capability of these systems is somewhat limited. For example, the Proteus system helps users manually parallelize a BLAST [8] 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. Our strategy differs from previous work in that it automates the optimization phase and adopts a pipelining strategy to reduce runtime storage requirements and to improve parallelism.

Our workflow language is very simple and described as part of the process ontology, much in the style of OWL-S [9].

The paper is organized as follows. The first section (*Results*) is divided in 3 sub-sections. The first (*An Architecture for BioDAMS*) describes the reference architecture for BioDAMS. The second sub-section (*Bioinformatics Process Ontology*) describes the process ontology. The third sub-section (*Optimized Workflow Execution in a Centralized Environment*) outlines the optimization heuristics for centralized environments. The section *Discussion* presents extensions to distributed environments. Finally, the last section contains the conclusions.

Results

An Architecture for BioDAMS

This section briefly introduces a reference architecture for a Bioinformatics data analysis and management systems (BioDAMS).

The architecture has four major components:

- the Bioinformatics workflow management subsystem (BioWfMS) helps researchers define, schedule, monitor, validate, optimize and execute workflows;
- the Bioinformatics scientific data management subsystem (BioDBMS) stores and manages scientific data;
- the Bioinformatics ontology management subsystem (BioOnMS) stores and manages all ontologies the BioDAMS uses, including a process ontology (see next section);
- the Bioinformatics administration subsystem (BioAdmS) offers a user interface to access system resources and is responsible for the communication between the various components.

The BioWfMS is further decomposed into four modules:

- the *workflow assistant* helps users define workflows;
- the *workflow compiler* is responsible for validating workflows, for translating them into an internal format, and for applying compile-time optimization heuristics;
- the *workflow monitor* is responsible for running workflows and for applying runtime optimization heuristics;
- the *workflow scheduler* is responsible for managing scheduled workflows.

The BioWfMS uses the process ontology for different tasks. First, the workflow assistant uses the ontology to help researchers define workflows that meet their goals, using data and process quality properties.

The workflow compiler uses the ontology to verify workflow consistency, such as input/output data coherency, and to perform compile-time optimizations. Finally, and the focus of this paper, the workflow monitor explores data and process properties to perform run-time optimizations.

Bioinformatics Process Ontology

This section defines a *Bioinformatics process ontology* that captures properties of the Bioinformatics data and analysis programs, and includes facilities to define process composition. The ontology is used in the next sub-section (*Optimized Workflow Execution in a Centralized Environment*) to design workflow optimization heuristics, which explore properties that capture process behaviour related to data consumption and production. It is also used in [10] to design a module that helps researchers define workflows, which takes advantage of a unified vocabulary and quality properties.

We assume that a process ontology combines a core process ontology and an application process ontology. The *core process ontology* contains the classes and properties of datasets and analysis programs that we consider as part of our reference architecture for ScDAMS. The *application process ontology* must model the analysis processes and datasets of the application domain in question as subclasses of these core classes and properties.

The core classes are *Process*, *Container*, *Connection* and *Workflow*, whose instances are called, respectively, *processes*, *containers*, *connections* and *workflows*. A process models a call to a program; a container models a data structure that stores a dataset shared between processes; a connection models how a process reads or writes a dataset; and a workflow models a process composition.

Processes

The class *Process* has instances, called *processes*, which model calls to Bioinformatics programs.

This class has the following subclasses: *Constructive*, *Filter*, *Internal Control* and *External Control*. Instances of the *Constructive* class are called *constructive processes*, and similarly for the other subclasses.

Intuitively, a constructive process creates new datasets pertaining to the Bioinformatics domain. The ontology contains various subclasses of the *Constructive* class, that resulted from a careful analysis of the literature (see, e.g. [11]) and several interviews with researchers from local groups.

A filter process analyses a dataset and extract some of the data for future processing.

An internal control process is a system's process included for various reasons, such as data conversion, flow control, etc.

The *Internal Control* class has several subclasses, two of which are:

- *Data Format Transformation*: an instance of this class applies a transformation to a dataset. For example, there is a process to transform a nucleotide sequence into an aminoacid sequence.
- *Inspection*: an instance of this class verifies, for example, if the data output from a process can be used as input data for the next process, and raised an error exception if not.

An external control process helps the user manage workflow execution. The *External Control* class has several subclasses, three of which are:

- *Stop*: an instance of this class indicates a temporary stop point in the workflow, possibly because the user wants to inspect some partial result;
- *Exit*: an instance of this class indicates a point where workflow execution must stop;
- *If*: an instance of this class is always used in conjunction with another process instance *p*, and indicates a condition, evaluated against the input containers of *p*, that must be satisfied for *p* to execute.

A process that models a call to a program *P* has *parameter* properties, or simply *parameters*, that correspond to the parameters *P* accepts.

The subclasses of the *Process* class have the following meta-properties, called *quality properties*:

- *performance*: measures the amount of computational resources, such as CPU time, I/O activity, consumed by the processes of class *C*;
- *popularity*: measures the percentage of researchers that use the processes of class *C*;
- *cost*: measures the (dollar) cost of executing a process of class *C*;
- *standard*: measures how much processes of class *C* are indicated as the standard option for the task they are designed for;
- *fidelity*: measures how much the results generated by processes of class *C* are close to the optimum. Indeed, since most of the analysis the Bioinformatics are based on heuristics, the results may contain errors (false positives and false negatives). Thus, this meta-property indicates which process class should be preferred.
- *adequacy*: indicates whether processes of class *C* are adequate or not to a given type of analysis project. Indeed, since most of the Bioinformatics analysis programs are based on heuristics, the results may contain errors (false positives and false negatives). Thus, this meta-property indicates which process class should be preferred.

Quality properties differentiate process classes from one another. Based on quality properties that the researcher prioritizes, the workflow management system may help him decide which is the most adequate process to adopt for the analysis in question. For example, suppose the researcher wants to compute the local alignment between a new sequence and a given sequence database. This task may be carried out by processes from the classes BLAST or FAST, depending on the quality properties the researcher prioritizes. If the researcher prioritizes performance, the system may indicate to use a BLAST process. However, if he prioritizes fidelity, the system may select a FAST process.

The system may change the values of the quality properties of a class over time. For example, the system may increase the popularity property of a class *C* if researchers consistently prefer processes from *C*.

Connections

The class *Connection* has instances, called *connections*, which model how processes read or write data items into containers. This class has the following properties:

- *type*, that assumes the values ‘*gradual*’ or ‘*non-gradual*’;
- *origin*, that assumes values from the class *Container* or from the class *Process*;

- *destiny*, that assumes values from the class *Process*, if the value of the *origin* property of the connection is an instance of the class *Container*, or from the class *Container*, if the value of the *Origin* property of the connection is an instance of the class *Process*.

Let *N* be a connection. *N* is *gradual* (or *non-gradual*) iff the value of the *type* property of *N* is ‘*gradual*’ (or ‘*non-gradual*’). *N* is a *read connection* (or a *write connection*) iff the value of the *usage* property of *N* is ‘*read*’ (or ‘*write*’).

The *origin* of *N* is the process or container that is the value of the *origin* property of *N*. The *destination* of *N* is similarly defined.

N is a *read connection* iff *N* has a container as its origin and a process as its destination. Intuitively, a read connection indicates that the process reads data from the container using the connection; in this case the process is called a *consumer of the container*. Symmetrically, *N* is a *write connection* iff it has a process as its origin and a container as its destination. It indicates that the process writes data into the container using the connection; in this case the process is called a *producer for the container*.

As an example, consider the Phrap process, which assembles fragments. Phrap accepts a set of *reads* (which are sequences of nucleotides) as input and generates a set of *contigs* (which are also sequences of nucleotides) as output. To perform fragment assembly, Phrap must analyze all *reads*, which implies that Phrap accesses *reads* in a non-gradual way. During the analysis, Phrap generates *contigs* one-by-one, which means that Phrap generates *contigs* in a gradual way.

The *connection-type* property of a connection specifies how the process reads data from the container, or writes data into the container, as follows:

- in a gradual read connection, the consumer starts to read as soon as data items become available, and it reads each data item only once;
- in a non-gradual read connection, the consumer starts to read only after all data items are available, and it may re-read a data item;
- in a gradual write connection, the producer makes a data item available for reading immediately after writing it, and it may write a data item only once;
- in a non-gradual write connection, the producer makes data items available for reading only after writing all data items, and it may rewrite data items.

Containers

The class *Container* has instances, called *containers*, which model the data structures that store and manage the datasets the workflow processes share.

Containers can be the origin or destination of connections. Containers that are not the destiny of any connection are called *resources*. They typically model biosequence databases or other data sources that the processes use.

A container has a property, *data-format*, which indicates the format of the data the container holds. For example, nucleotide sequences from the NR database may be stored in the original format of a Genbank [12] record, in the FASTA format, or in XML. Usually, analysis programs require that all data items stored in a container have the same format. For example, BLASTP accepts as input only databases in the FASTA format, and generates results in several formats, such as HTML and XML.

Containers have quality properties. For example, ‘*curated*’ or ‘*redundant*’ are qualities of a resource in the Bioinformatics domain. The Swiss-Prot [13] is a curated database, which means that its data were evaluated by a researcher. By contrast, the

TrEMBL database [13] holds sequences that are automatically generated by translating the nucleotide sequences stored in the EMBL database and which are not in the Swiss-Prot database.

A container is *gradual*, *non-gradual* or *mixed* depending on the value of the property *type*. The value of this property is derived from the connection properties, as defined in Table 1.

Data Volume

Each subclass C of the *Process* class has two meta-properties, *minimum-data-volume* and *maximum-data-volume*, that associate each process p in C with estimations for the minimum (total) data volume and the maximum (total) data volume that p will output through w , for each write connection w originating from p . These estimations are based on the sizes of the containers connected to p by a read connection and on the parameter values of p . Note that we freely treat these estimations as meta-properties of C because they will most likely be specified as functions, which is problematic to express in ordinary ontology languages.

Likewise, the meta-property *maximum-item-size* associates each process p in C with an estimation for the maximum size of the data items that p will output through w , for each write connection w originating from p .

Let c be a container connected to processes p_1, \dots, p_n by write connections. Then, c also has three properties, *minimum-container-size*, *maximum-container-size* and *maximum-container-item-size*, derived from the similar meta-properties of the classes that p_1, \dots, p_n belong to.

For example, for processes of the class BLAST, these meta-properties are defined based on the sizes of the new biosequence and the biosequence database given as input and on an estimation of their similarity [8].

Naturally, these meta-properties are important for the optimization heuristics. Next section will discuss implementations for container and estimations for the minimum and maximum container sizes.

Projects

The class *Project* has instances, called *projects*. This class has two subclasses, *EST* and *Complete*, whose instances are called *EST genome projects* and *complete genome projects*.

One may define other subclasses if it becomes necessary to differentiate projects, for example, according to the organism under investigation.

Determining the class of a project is important because it influences the choice of the appropriate processes. For example, Phrap and CAP3 are both processes of the fragment assembly class, but Phrap is a better choice for complete genome projects, whereas CAP3 is more adequate for EST genome projects.

Workflow

The class *Workflow* has instances, called *Bioinformatics workflows* or simply *workflows*, that model process composition.

A workflow has three properties that model its structure (property names were chosen to avoid conflicts with class names):

- *process-used*, that relates a workflow to a process;
- *container-used*, that relates a workflow to a container;

– *connection-used*, that relates a workflow to a connection.

All these properties are multi-valued since a workflow may naturally have more than one process, container or connection.

A workflow also has a fourth property, called *associated-project*, that relates a workflow to a project.

By the very definition of connection, a workflow induces a bipartite graph, with nodes representing processes, called *process-nodes*, nodes representing containers, called *container-nodes*, and arcs that connect process-nodes to container-nodes, or container-nodes to process-nodes, called *connection arcs*.

Note that the graph is indeed bipartite since a connection arc never connects a process-node to a process-node, or a container-node to a container-node.

Optimized Workflow Execution in a Centralized Environment

This section describes a workflow monitor implementation that includes optimization heuristics to reduce runtime storage requirements. The heuristics are based on the Bioinformatics process ontology and explore properties that capture process behaviour related to data consumption and production.

The environment is characterized by a centralized system, which runs all BioAMS components and all user programs, and which stores all the required data.

Container Implementation

Given the nature of analysis processes and datasets, a container may potentially occupy considerable space. However, depending on the container type, it is possible to reduce the required space by adopting one of the special data structures: *LimitedBuffer*, *UnlimitedBuffer*, *File*, *FileLimitedBuffer* and *FileUnlimitedBuffer*.

Data Structures for Container Implementation

LimitedBuffer

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 behaviour 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*. Table 2 shows the characteristics of the *LimitedBuffer*.

UnlimitedBuffer

An *UnlimitedBuffer* is similar to a *LimitedBuffer*, except that its space is not limited. In particular, a data item is removed when read by all consumers, and a data item is also available for reading immediately after a producer writes it into the *UnlimitedBuffer*. The *UnlimitedBuffer* methods are similar to those of *LimitedBuffer*, except that method *Put* never blocks a connection.

File

A *File* offers a data structure, with unlimited space, 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. Table 3 shows the characteristics of the File.

FileLimitedBuffer

A *FileLimitedBuffer* combines a *File*, with unlimited space, with a *LimitedBuffer*. The *File* component will store the data items received from the non-gradual write connections of the container, and the *LimitedBuffer* component will store the data items received from the gradual write connections. Both components will provide data items to the consumers. The methods *Open*, *Close*, *Write*, *Rewrite* and *Read* are similar to *File* methods and the methods *Put* and *Get* are similar to the *LimitedBuffer* methods.

FileUnlimitedBuffer

Likewise, a *FileUnlimitedBuffer* combines a *File* with an *UnlimitedBuffer*. The file will store data items received from the non-gradual write connections, the limited buffer will store data items received from the gradual write connections and both types of write connections will provide data items for the read connections. The *FileUnlimitedBuffer* has methods similar to those of *FileLimitedBuffer*, without space limitations.

Estimation of Container Sizes

The minimum and maximum sizes are important parameters for the optimizations heuristics. Note, from Table 4, that a limited buffer always occupies the smallest space.

An *UnlimitedBuffer* requires space which is equivalent to that of a *File*, since it does not block producers. Hence, apparently, it is equivalent to a *File* as far as maximum space consumption is concerned. However, an *UnlimitedBuffer*, unlike a *File*, allows a data item to be freed as soon as all consumers read it. Therefore, an *UnlimitedBuffer* is not equivalent to a *File* as far as average space consumption is concerned.

A *FileLimitedBuffer* occupies less space than a *File* since the *LimitedBuffer* (holding at least one data item) is sufficient to handle gradual write connections to the container.

The comparison between a *FileUnlimitedBuffer* and a *File* parallels that between an *UnlimitedBuffer* and a *File*.

In summary, as far as maximum space consumption is concerned, one may reduce container implementation to three options: *LimitedBuffer*, *File* and *FileLimitedBuffer*. However, as far as average space consumption is concerned, *UnlimitedBuffer*'s and *FileUnlimitedBuffer*'s are also useful options.

Recommended Container Implementations

The rest of this section discusses the recommended container implementations. Let c be a container, R be the aggregated rate of the read connections for c , W be the aggregated rate of the write connections for c , and G be the aggregated rate of the gradual write connections for c .

Table 5 summarizes the recommended container implementations.

Suppose that c is a gradual container. Then, three implementations are recommended. If $R \geq W$ and the consumption rates are approximately equal, it is recommended to implement c using a *LimitedBuffer*. Indeed, any consumer may start reading immediately after the buffer contains a data item, while producers continue to write new data items as long as space is available. Since $R \geq W$, a *LimitedBuffer* suffices to serve the producers. Buffer size will depend on the available system space and on

estimates for R and W . If it is not possible to, *a priori*, estimate buffer size, the workflow execution engine will choose buffer size adaptatively.

However, if $R \geq W$ but the consumption rates are not approximately equal, then consumers may synchronize. Indeed, each data item stored in c may only be discarded when all consumers read it (and the buffer space is limited). Hence, slower consumers will slow down faster consumers. In this case, it is recommended to implement c using an *UnlimitedBuffer*. Indeed, faster consumers will keep reading newer data items as soon as producers write them, while slower consumers will still read older data items.

If $R < W$, it is recommended to implement c using an *UnlimitedBuffer*. Indeed, if the aggregated consumption rate is less than the aggregated production rate, a *LimitedBuffer* would eventually become full. 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 c is a non-gradual container. Then, c must be implemented using a *File* since processes with non-gradual connections may use *Read*, *Write*, *Reread* and *Rewrite* methods.

Suppose that c is a mixed container. Then, three implementations are recommended, depending on somewhat more complex factors. If $R \geq G$ and all read connections are gradual, then it is recommended to implement the mixed container using a *FileLimitedBuffer*, thereby reducing space requirements. Indeed, since all read connections are gradual, data items sent through the write gradual connections may be relayed, through the *LimitedBuffer*, to the gradual read connections.

If $R < G$ and all read connections are gradual, then it is recommended to implement the mixed container using a *FileUnlimitedBuffer*, thereby also reducing space requirements.

If some read connection is non-gradual, then one must use a *File*, since processes with non-gradual connections may use *Read*, *Write*, *Reread* and *Rewrite* methods.

An Example of Container Size Estimation

This section exemplifies how to estimate the container size and the execution time for processes of the BLAST family, chosen for their importance.

The BLAST process family includes, among others, the following: BLASTP, BLASTN, BLASTX, TBLASTN and TBLASTX. We use the term “BLAST” when the discussion is not specific to any particular process of the family.

Basic BLAST algorithm

BLAST receives as input a DNA or protein sequence s and a DNA or protein database D (for simplicity, we assume just one input sequence) and computes *local* sequence alignments between segments (or subsequences) of s and segments of the sequences stored in D . It proceeds as follows:

- Step 1. For each segment s_i of size w of the input sequence s , determine a list L with all segments that align with s_i and that have a score of at least T . Let s' and s'' be two aligned segments of the same size. The score of the alignment is the sum of the similarity values for each pair of aligned characters (bases for nucleotide sequences, and residues, for amino acid sequences). A score matrix, which is a parameter of the algorithm, defines the similarity values.

- Step 2. Find alignments between the segments in the list L , created in Step 1, and segments of the database whose score is greater than or equal to T . Such alignments are called *hits*.
- Step 3. For each hit found in Step 2, determine if it occurs within an alignment whose score is greater than or equal to a parameter S . This is executed by extending the hit in both directions until the alignment score reaches a value controlled by parameter X .

X is an integer that captures the maximum allowed reduction in alignment score for the segments being extended. A segment pair thus obtained is called a *high score segment pair* (HSP).

The result of the algorithm is the list of HSPs that Step 3 generates.

The user may configure T , w , X and the score matrix, or accept their default values [14].

Some processes of the BLAST family, such as BLASTX, TBLASTN and TBLASTX, translate nucleotide input sequences to amino acid sequences. BLASTX, for example, receives as input a nucleotide sequence and a database holding sequences of amino acids. It then translates the input nucleotide sequence to 6 amino acid sequences. Finally, it compares these 6 translated sequences with the amino acid database. This means that the three steps of above are executed 6 times.

Container Size

BLAST outputs all local alignments found between the input sequence s and the input database D . The output size can be estimated as:

$$(1) \text{ output size} = B * \text{hsp_max} * \text{max_size_alin}$$

where (see [14]):

- B is the maximum number of sequences from DB allowed to appear in the output (with default value equal to 250)
- hsp_max is the maximum number of alignments from each sequence in the DB allowed to appear in the output (with default value equal to 1000)
- max_size_alin is the maximum estimated size of an alignment, estimated as $\log(KMN)/H$ [15], where:
 - M is the size of the input sequence (number of bases or residues)
 - N is the size of the database (number of bases or residues of all sequences in the database)
 - K and H are defined by formulas [16] that estimate when an alignment is true or is a false positive. These parameters are computed by BLAST [14].

Execution Time

The execution time of BLAST can be estimated as [8]

$$(2) \text{ execution time} = aW + bN + cNW/mw$$

where

- a , b and c are constants
- W is the number of segments Step 1 generates

- N is the number of bases or residues that Step 2 touches when scanning the database
- NW estimates the computational effort in Step 3
- m is equal to 4 (number of nucleotides) or 20 (number of amino acids), depending on the specific BLAST process

W increases exponentially when T decreases, increases linearly with the size of the input sequence.

The specific aspect of the input sequence influences BLAST's execution time [15]. Sequences with highly repetitive regions frequently do not have phylogenetic relevance, but they are fairly common in various organisms. This causes BLAST to find many hits, when in fact they are false positives. Examples of repetitive regions are the homopolimeric (such as AAAA...AAAAA), or regions with repetitive dinucleotides (such as AGAGAGAG...AGAGAG that repeats the dinucleotide AG) or trinucleotides (such as AGGAGGAGGAGG...AGGAGGAGG that repeats the trinucleotide AGG), and so on.

As mentioned before, some processes of the BLAST family, such as BLASTX, TBLASTN and TBLASTX, translate nucleotide input sequences to amino acid sequences. When this translation occurs, W must take into account the translated input sequence, and N must be computed according to the translated database.

Container Sharing

Equation (2), which estimates BLAST's execution time, also helps choosing the container type as follows.

Consider two BLAST processes, b_1 and b_2 , that compare the same set of input sequences against two different databases, bd_1 and bd_2 , respectively. Assume that b_1 and b_2 read the set of input sequences from the same container c_1 , and that c_1 is generated by a process p_{p1} . In this case, the process, b_1 or b_2 , which has the smaller consumption rate may delay the other process.

Figures 1a and 1b illustrate two possible configurations that differ on how p_{p1} generates the input sequences in c_1 .

If p_{p1} generates the input sequences in a non-gradative way, represented by a continuous line connecting p_{p1} and c_1 in Figure 1a, processes b_1 and b_2 do not interfere with each other.

However, if p_{p1} generates the input sequences in a gradative way, represented by a dashed line connecting p_{p1} and c_1 in Figure 1b, and if b_1 and b_2 have different consumption rates, then b_1 may delay the execution of b_2 , or vice-versa. Thus, if we can compute their consumption rate using Equation (2), we may decide the best container implementation for c_1 : a limited buffer, if b_1 and b_2 have approximately the same consumption rate; or an unlimited buffer, otherwise.

An Optimized Workflow Monitor for Centralized Environments

In this section, we describe an optimized workflow monitor that reduces runtime storage requirements and improves parallelism, in the context of a centralized environment. The heuristics explores properties, defined in the process ontology, that capture process behaviour related to data consumption and production.

Data Structures

The workflow monitor receives a workflow specification and constructs the workflow bipartite graph, together with labelling functions that reflect the properties defined in the process ontology, as well as the following new labelling functions:

- a process node has a label, *process-state*, with values ‘initial’, ‘pending’, ‘executing’, and ‘finished’;
- a container node has labels *temporary*, which is a Boolean, and *size*, which is an integer;
- a connection arc has a label, *connection-state*, with values ‘idle’, ‘open’ and ‘closed’.

The monitor initializes these new labels as follows:

- for each process node *p*, *state* is set to ‘initial’ to indicate that *p* has not yet started executing;
- for each container node *c* that is a resource, *temporary* is set to ‘false’, otherwise it is set to ‘true’;
- for each connection arc, if it starts on a container node that models a resource, *state* is set to ‘open’, otherwise it is set to ‘idle’;
- for each container node *c* that does not model a resource, *size* is initialized with the amount of disk space reserved for the data structure chosen to implement *c* (see the next subsection). If *c* models a resource, *size* is set to 0, since the data is already stored on disk.

Container Implementation

Given the nature of analysis processes and datasets, a container may potentially occupy considerable space. However, it is possible to reduce the required space by adopting one of the data structures: *LimitedBuffer*, *UnlimitedBuffer*, *File*, *FileLimitedBuffer* and *FileUnlimitedBuffer*. We briefly comment here on the first three, since the last two are combinations of the others.

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 behaviour 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*.

An *UnlimitedBuffer* is similar to a *LimitedBuffer*, except that its space is not limited.

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.

Very briefly, the monitor chooses to implement a container *c* using a *LimitedBuffer* when all connection arcs entering and leaving *c* are gradual. Otherwise, it decides between the other data structures, again based on the types of the connections reaching *c* (see [10] for the details).

Finally, the minimum size of a *LimitedBuffer* is the size of the largest data item that the container must hold, and the maximum size can be arbitrarily set to a multiple of the minimum size. The minimum and maximum sizes of the other data structures can be set to the minimum and maximum sizes of the container, computed from the process ontology.

Process Node State Change

The workflow monitor may run in three different modes, which we call *aggressive*, *conservative* and *optimistic*. In the aggressive mode, the monitor tries to run in parallel as many process nodes as possible, regardless of the disk space available. In the conservative mode, the monitor takes disk space into account and uses the maximum size estimations to compute the amount of disk space that must be pre-allocated to each container implementation (i.e., it sets *size* to maximum size estimation). In the optimistic mode, the monitor also takes disk space into account, but uses the minimum size estimations.

The processes that can be executed in parallel at any given time are organized 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*.

In the aggressive mode, the monitor stops at the first phase and uses the graph thus formed. In the other two modes, the monitor goes into a second phase, which takes the available disk space into account. At this point, recall that the optimistic and the conservative modes differ on the way they estimate the sizes of the container implementations.

The monitor modifies the execution graph by pruning container nodes and process nodes until the graph contains only process nodes that can be executed in parallel within the limits of disk space available. As a result of this phase, the execution of some process node p may have to be postponed, indicated by changing its state to *'pending'*.

The workflow execution monitor starts processing each process node p in the execution graph by setting its state to *'executing'*. Also, the state of the connection arcs that touch p may change, as described in the next subsection. When process node p finishes executing, the monitor changes its state to *'finished'*.

When a process node finishes executing, the monitor 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 monitor then loops back and re-computes the execution graph.

Arc State Change

During workflow execution, 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 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 monitor 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 the workflow monitor.

Pruning

A naïve algorithm to decide the optimum choice of container nodes that must be pruned to cut B bytes would be: (1) construct all sets of container nodes which are sinks of the execution graph, of cardinality 1 to n , where n is the total number of sink container nodes in the execution graph; (2) compute the gain, in disk space, when all containers in each of these sets is pruned; (3) choose the best solution, i.e, the set of container nodes that frees the least amount of space S such that $S \geq B$. This naïve algorithm has exponential complexity and is unacceptable. Indeed, the problem we face can be proved to be NP-Complete [17].

We therefore adopted a greedy pruning heuristics [18] that has acceptable performance. The heuristics proceeds from the sink nodes to the source nodes of the execution graph and has the following major steps: (1) compute the gain from pruning each sink container node; (2) order the nodes in decreasing gain (see the next subsection); (3) prune nodes in decreasing gain order until there is enough space on disk to run all ready process nodes.

Finally, recall that the monitor ignores disk space limitations, in the aggressive mode, and uses estimated container sizes to prune the execution graph before each execution cycle, in the other two modes. Therefore, it may run into disk space overflow, either because it ignored the problem or because it used just estimated sizes. When this occurs, the monitor may resort to the pruning procedure to select which process nodes to cancel and which container nodes to discard in order to free enough disk space.

Computing the Gain from Pruning

Let c be a sink container node c in the execution graph. To compute the gain in disk space when c is pruned, we proceed as follows.

When c is pruned, each process node p that consumes or 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 forces the monitor to re-consider the implementations adopted, and thereby the disk space required, for some containers that were maintained in 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 p . 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, which is a desirable feature.

Example of Pipeline Optimization

To illustrate the optimization method introduced before, consider the workflow bipartite graph W shown in Figure 2, where c_i denotes a container node, p_j denotes a process node and a_{ij} denotes an arc. Dashed lines indicate gradative arcs and solid lines, non-gradative arcs.

Figure 3a shows the workflow bipartite graph with labelling functions indicating arc and process states.

The initial state of an arc is *inactive*, represented by $e_a=i$, or *open*, represented by $e_a=a$; the initial state of a process is *initial*, represented by $e_p=i$.

Suppose that there is a fictitious initial process p_0 . When p_0 terminates, its state becomes *finished*, and the state of arc a_{00} that connects p_0 to c_0 becomes *open*, as well as the state of a_{01} . Observe that p_5 has a container c_8 that stores a database. The initial state of this container is *open* and has attribute *size* set to 0, and the attribute *temporary* set to *false*.

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

We first construct the execution graph, S_I , shown in Figure 3b. Note that the processes that are ready to fire are $\{p_1, p_2, p_3, p_4, p_6, p_7\}$. Only p_5 cannot be fired since it consumes, in a non-gradative way, the data that p_3 produces (arc a_{35} is of type *non-gradative* and has state equal to *inactive*).

In the aggressive mode, the monitor stops at this time and uses the graph thus formed. In the other two modes (optimistic and conservative), the monitor goes into a second phase, which takes the available disk space into account. Suppose the monitor works in the conservative mode. Therefore, we construct a new execution graph, R_I , as follows.

Suppose that the size of S_I is greater than k . Then, we must start the pruning process.

R_I is initially a copy of S_I . The container nodes in the frontier of the R_I are $\{c_3, c_6, c_7\}$ (see Figure 3c).

Suppose that the gain in space obtained by removing c_3, c_6 and c_7 is g_3, g_6 and g_7 , respectively.

To illustrate, let us compute g_7 . The set of processes that must be pruned starts with $\{p_7\}$, because p_7 is the only process that produces data in c_7 . Since p_7 does not produce data in any other container, no other process is added to set $\{p_7\}$. When we delete c_7 and p_7 , the container node c_4 , that was consumed by p_7 in a gradative way (arc a_{47} is

gradative), will have to store all data until p_7 is finally fired in a subsequent execution stage. Hence, the set of containers that change size is simply $\{c_4\}$.

The space gained G when c_7 is removed is the current size of c_7 , that is, the value of the label *size* of c_7 . The net gain, g_7 , is the difference between G and the extra space required by c_4 , which had its implementation strategy changed.

Suppose that c_7 has the best gain. We then create the execution graph R_1 by deleting c_7 , as shown in Figure 3c. Note that the frontier of R_1 contains the container nodes $\{c_3, c_6\}$.

Suppose that the space R_1 requires still exceeds the available space. We then repeat the above procedure.

Suppose that gain in space obtained by removing c_3 and c_6 is g_3 and g_6 , respectively, and that $g_6 > g_3$. Then, we prune c_6 and p_6 . We note 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 that p_4 produces. Hence, the gain when pruning c_6 is then equal to the size of c_6 .

Figure 3d shows the new execution graph, R_2 , after pruning c_6 and p_6 .

Suppose that the space R_2 requires still exceeds the available space. We then prune c_3 and p_3 , constructing R_3 , as shown in Figure 3e.

Figure 4a shows the arc and process states after firing p_1 , p_2 and p_4 . The state of some process nodes changed from *initial* to *executing*, represented by $e_p=e$, or to *pending*, represented by $e_p=p$, the state of some arcs changed from *inactive* to *open*, represented by $e_a=a$.

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 4b shows the execution graph at this phase, S_2 , after p_1 terminates. Note that S_2 has two connected components and that the set of processes ready to be fired is $\{p_3, p_6, p_7\}$. The processing of S_2 follows similarly.

Discussion

In this section, we very briefly indicate how we can extend the strategy to distributed environments.

Recall that, in a centralized environment, the monitor prunes process nodes that are sinks of the execution graph until the remaining process nodes can be run in parallel. In a distributed environment, the pruning process is somewhat more complex since we now have several processors, each with its own disk space limitations. The monitor must then choose a processor to allocate each process node in the graph. If this step fails, the monitor prunes process and container nodes until the remaining containers can all be allocated (perhaps in distinct processor). The allocation of process nodes to processors cannot be executed by traversing the graph from sink nodes to source nodes, since we run the risk of finding no processor with enough disk space to run the source process nodes, which must be executed first. In this case, we have to be more conservative and analyze the graph from sources to sinks instead.

We may adopt a pessimistic or an optimistic strategy to estimate container sizes.

In the pessimistic strategy, we assign a process node p to a processor R only if R has enough space to hold, with their maximum size, all container nodes that p generates. This strategy is pessimistic in the sense that, even if c is a gradual container node, it will allocate c with its maximum size.

In the optimistic strategy, we allocate gradual container nodes with its minimum size. In this case, since we analyze a process node p that consumes c after we allocate c , we may be left with no processor to assign p to. Consequently, we will have to prune p . In this case, we will have to re-compute the allocations already done, since c will have to hold more data than its minimum size. Indeed, process nodes that have been pruned will consume c in another execution stage.

Suppose that the cost of accessing remote data is of the same order of magnitude as 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 accessing local data. In this case, a process node is best allocated to a processor node that requires the least amount of data transferral to run the process. This implies that pipelining is best used when both processes will run on the same processor, obviously.

However, before considering the cost of moving a container c , we have to verify if c is a resource that can indeed be moved. If c cannot be moved, we call it an *anchor* and we will have to run the process in the same node N as c , irrespectively of the cost of moving the other container nodes to N .

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.

Conclusions

This paper described a strategy for workflow execution that uses pipelining to achieve three goals. First, it optimizes runtime storage requirements, which is important in a centralized environment with limited storage resources (compared to the size of the datasets involved). Second, it improves parallelism in a distributed environment or in a centralized system with multiple processors. Finally, through pipelining, the workflow monitor may make partial results available to the users sooner than otherwise, thereby helping users monitor workflow execution.

We presented a detailed description of the workflow monitor for a centralized environment. However, the monitor can be extended to distributed environments, as briefly discussed in the previous section. We refer the reader to [10] for a detailed discussion about implementation and performance issues in both centralized and distributed environments, in the context of the BioNotes system [19, 20].

Finally, we stress that the strategy can be extended to other application domains by changing the process ontology, which captures the properties that are relevant to deciding when to apply pipelining.

Authors' contributions

This work is a summary of the Ph.D. Thesis of MLC, under the orientation of MAC, co-orientation of ABM and with the collaboration of LFBS. All those involved have read and approved the final manuscript.

Acknowledgements

We would like to thank Paulo Ferreira and Orlando Martins for several interesting discussions. This work was partially supported by CNPq under grants no. 141938/2000-5 for Melissa Lemos and no. 552040/2002-9 for Marco A. Casanova.

References

1. Ewing B et al: **Base-Calling of Automated Sequencer Traces using Phred. I. Accuracy Assessment.** *Genome Research* 1998, **8**: 175-185.
2. Hall RD, Miller JA, Arnold J, Kochut KJ, Sheth AP, Weise MJ: **Using Workflow to Build an Information Management System for a Geographically Distributed Genome Sequence Initiative.** In *Genomics of Plants and Fungi*. Edited by Rolf A. Prade and Hans J. Bohnert. Marcel Dekker Inc. New York NY; 2003: 359-371.
3. Altintas I, Berkley C, Jaeger E, Jones M, Ludäscher B, Mock S: **Kepler: An Extensible System for Design and Execution of Scientific Workflows.** In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management: 21-23 June 2004; Santorini Island, Greece*. IEEE Computer Society, 2004: 423-424.
4. Wroe C, Goble C, Greenwood M, Lord P, Miles S, Papay J, Payne T, Moreau L: **Automating Experiments Using Semantic Data on a Bioinformatics Grid.** *IEEE Intelligent Systems* 2004, **19**(1): 48-55.
5. Cannataro M, Comito C, Lo Schiavo, Veltri P: **Proteus, a Grid based Problem Solving Environment for Bioinformatics: Architecture and Experiments,** *IEEE Computational Intelligence Bulletin* 2004, **3**(1): 7-18.
6. Rowe A, Kalaitzopoulos D, Osmond M, Ghanem M, Guo Y: **The discovery net system for high throughput Bioinformatics,** *Bioinformatics* 2003, **19** Suppl. 1: i225-i231.
7. Medeiros C, Vossen G, Weske M: **WASA: A Workflow-Based Architecture to Support Scientific Database Applications.** In *Proceedings of the 6th International Conference on Database and Expert Systems Applications: 4-8 September 1995; London, United Kingdom*. Edited by Norman Revell, A. Min Tjoa, 1995: 574-583.
8. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: **A basic local alignment search tool,** *J. of Molecular Biology*, 1990, **215** (3): 403-410.
9. **OWL-S: Semantic Markup for Web Services**
[<http://www.w3.org/Submission/2004/SUBM-OWL-S20041122/Overview.html>]
10. Lemos M: **Workflow para Bioinformática.** *PhD thesis*. Pontifícia Universidade Católica do Rio de Janeiro, Computer Science Department; 2004.

11. **A Classification of Tasks in Bioinformatics**
[<http://imgproj.cs.man.ac.uk/tambis/questionnaire/bio-queries.html>]
12. **National Center of Biotechnology Information**
[<http://www.ncbi.nlm.nih.gov/>]
13. Boeckmann B, Bairoch A, Apweiler R, Blatter MC, Estreicher A, Gasteiger E, Martin MJ, Michoud K, O'Donovan C, Phan I, Pilbout S, Schneider M: **The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003**, *Nucleic Acids Research* 2003, **31**(1): 365-370.
14. **Washington University BLAST Archives**
[<http://blast.wustl.edu/doc/blast1.pdf>]
15. Gish W (personal communication), 2004.
16. Karlin S, Altschul SF: **Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes**. In *Proceedings of the National Academy Sciences of the United States of America* 1990, **87**(6): 2264-2268.
17. Garey M, Johnson D: *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA; 1979.
18. Rajasekaran S, Horowitz E, Sahni S: *Computer Algorithms C+ : C++ and Pseudocode Versions*. W. H. Freeman; 1996.
19. Lemos M, Casanova MA, Seibel LFB, Macedo JAF, Miranda AB: **Ontology-Driven Workflow Management for Biosequence**. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications: August 30-September 3 2004; Zaragoza, Spain*. Edited by Fernando Galindo, Makoto Takizawa, Roland Traunmüller, 2004: 781-790.
20. Lemos M, Casanova MA, Seibel LFB: **BioNotes: A System for Biosequence Annotation**. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications: September 1-5 2003; Prague, Czech Republic*. IEEE Computer Society 2003: 16-20.

Figures

Fig. 1. BLAST's sharing containers.

Fig. 2. Workflow bipartite graph W .

Fig. 3. (a) Initial workflow bipartite graph with labelling functions (b) Ideal stage S_1 (c) Real stage R_1 (d) Real stage R_2 (e) Real stage R_3 .

Fig. 4. (a) Workflow bipartite graph after firing p_1 , p_2 and p_4 (b) Ideal stage S_2 .

Tables

Table 1 - Container types.

Type	Definition
<i>gradual</i>	– all input and output connections are gradual
<i>non-gradual</i>	– all input and output connections are non-gradual
<i>mixed</i>	– there is an input or an output connection that is gradual, and – there is an input or an output connection that is non-gradual

Table 2 - Characteristics of the LimitedBuffer Class.

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

Table 3 - Characteristics of the File Class.

Organization
<ul style="list-style-type: none">– data structure implemented as an unlimited size queue, 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 they are blocked or not
Methods
<p><i>Open(c):</i></p> <ul style="list-style-type: none">– process opens a connection <i>c</i> with the container– workflow topology determines if the connection is a read or write connection <p><i>Close(c):</i></p> <ul style="list-style-type: none">– process closes the connection <i>c</i> with the container <p><i>Put(c,d):</i></p> <ul style="list-style-type: none">– connection <i>c</i> must be a gradual write connection– adds <i>d</i> to the container– <i>d</i> is available for reading by consumers <p><i>Write(c,d):</i></p> <ul style="list-style-type: none">– connection <i>c</i> must be a non-gradual write connection– adds <i>d</i> to the container– <i>d</i> receives a number indicating its position in the file, which is returned by the method– <i>d</i> is available for reading by a consumer only after <i>c</i> is closed <p><i>Rewrite(c,d,i):</i></p> <ul style="list-style-type: none">– connection <i>c</i> must be a non-gradual write connection– rewrites <i>d</i> in position <i>i</i> of the file <p><i>Get(c):</i></p> <ul style="list-style-type: none">– connection <i>c</i> must be a gradual read connection– if the container has a data item <i>d</i> not yet read by <i>c</i>, then <i>d</i> is returned; otherwise, <i>c</i> is blocked until the container has a data item not yet read by <i>c</i> and which is available for reading <p><i>Read(c,i):</i></p> <ul style="list-style-type: none">– connection <i>c</i> must be a non-gradual read connection– returns the data item in position <i>i</i> of the file

Table 4 - Estimations for Container Sizes.

LimitedBuffer

Minimum – space required for storing the largest data item any process writes

Maximum – (set by the workflow monitor)

UnlimitedBuffer

Minimum – total space required for storing all data items
– all processes write

Maximum – (same as minimum)

File

Minimum – total space required for storing all data items
– all processes write

Maximum (same as minimum)

FileLimitedBuffer

Minimum – total space required for storing all data items
all processes write through a non-gradual connection,
added to space required for storing the largest data item any process
writes through
a gradual connection

Maximum – total space required for storing all data items
all processes write

FileUnlimitedBuffer

Minimum – total space required for storing all data items
– all processes write

Maximum – (same as minimum)

Table 5 - Recommended container implementations¹.

<i>LimitedBuffer</i>
<ul style="list-style-type: none"> - container is gradual - aggregated consumption rate is greater than or equal to the aggregated production rate - consumption rates are approximately equal
<i>UnlimitedBuffer</i>
<ul style="list-style-type: none"> - container is gradual - aggregated consumption rate is less than the aggregated production rate
<ul style="list-style-type: none"> - container is gradual - it is not possible to estimate the aggregated consumption rate or the aggregated production rate
<ul style="list-style-type: none"> - container is gradual - aggregated consumption rate is greater than or equal to the aggregated production rate - consumption rates are not approximately equal
<i>File</i>
<ul style="list-style-type: none"> - container is non-gradual
<ul style="list-style-type: none"> - container is mixed - some read connection is non-gradual
<i>FileLimitedBuffer</i>
<ul style="list-style-type: none"> - container is mixed - all read connections are gradual - aggregated consumption rate is greater than or equal to the aggregated production rate of the gradual connections
<i>Class FileUnlimitedBuffer</i>
<ul style="list-style-type: none"> - container is mixed - all read connections are gradual - aggregated consumption rate is greater than or equal to the aggregated production rate of the gradual connections
<ul style="list-style-type: none"> - container is mixed - all read connections are gradual it is not possible to estimate the aggregated consumption or production rates

(1) The condition for choosing a given implementation is the conjunction of the assertions in each table cell.