An Efficient and Reliable Scientific Workflow System[*]

Tulio Tavares[*], George Teodoro[*], Tahsin Kurc[†], Renato Ferreira[*],
Dorgival Guedes[*], Wagner Meira Jr[*], Umit Catalyurek[†],
Shannon Hastings[†], Scott Oster[†], Steve Langella[†], and Joel Saltz[†]

[*]Department of Computer Science
Universidade Federal de Minas Gerais, Brazil

[†]Department of Biomedical Informatics
The Ohio State University, USA

Abstract

This paper presents a fault tolerance framework for applications that process data using a distributed network of user-defined operations in a pipelined fashion. The framework saves intermediate results and messages exchanged among application components in a distributed data management system to facilitate quick recovery from failures. The experimental results show that the framework scales well and our approach introduces very little overhead to application execution.

1 Introduction

Scientific workflow systems [6, 5, 9] provide scientists with a suite of tools and infrastructure to build data analysis applications from reusable components and execute them. The challenges to implementing workflow middleware support for scientific applications are many. Analysis oftentimes requires processing of large volumes of data through a series of simple and complex operations. In biomedical domain, with advanced imaging sensors and scanners, it is possible to acquire very high resolution microscopic images of tissues and organs quickly. Each image can be up to Gigabytes in size and hundreds (even thousands) of images can be obtained from a single sample. These images are analyzed through a series of data processing operations, including signal extraction, registration, segmentation, and feature classification. To support such types of data processing efficiently, a workflow middleware system should leverage distributed computing power and storage space (both disk and memory space) and implement optimizations for large data retrieval and scheduling of I/O and computation components. Another challenging issue is to enable fault tolerance in the middleware fabric. An analysis workflow with complex operations on large data can take long time to execute. The probability of a failure during execution should be considered [8]. Efficient mechanisms are needed to support recovery from a failure without having to redo much of the computation already done.

In this paper, we propose and evaluate a fault tolerance framework for applications that process data using a pipelined network of user-defined operations in a distributed environment. In an earlier work [11], we described a middleware system to create workflows from compiled programs and shared libraries and manage their execution. We extend this framework to provide support for fault tolerance in scientific data analysis workflows. The extensions provide functionality and protocols to efficiently manage input, intermediate, and output data and associated metadata and to recover from certain types of faults that may occur in the system. In our approach, intermediate results and messages exchanged among application components are maintained in a distributed data management infrastructure along with additional metadata. The infrastructure consists of a persistent storage manager that stores check-pointed information in a distributed database and a distributed cache that reduces the overhead of check-pointing. We have developed a protocol among the various

[*]This research was supported in part by the National Science Foundation under Grants #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, CNS #0403342, #CNS-0426241, NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

components of the system to manage checkpoints and related information efficiently. The experimental results show that our approach provides an asynchronous data storage mechanism that minimizes overhead to the execution of the workflow.

2 Related Work

Bowers *et. al.* [2] describe a framework to support the structured embedding of generic control-flow components within dataflow process networks. The framework has some mechanisms that can be used to develop workflows via reusable control-intensive subtasks. The authors present a mechanism for data retransmission, but they do not examine the impact of this mechanism on fault tolerance. Kola [6] proposes a strategy of fault-isolation that decouples data placement and computation in order to reduce the need for re-computation. Kola's work also differentiates between persistent and transient failures and automatically recovers from transient faults. However, our work differs in that his approach requires input from the user and does not provide transparent mechanisms. The Phoenix [7] system provides some tools that allow fault tolerance for data intensive applications. These tools can detect failures and classify them into transient or persistent failures. The tools can also handle each transient failure appropriately. However, the recovery from such failures is carried out according the specifications provided by the user before execution. Tatebe presents the architecture of the Grid Datafarm [10]. This file system aims to manage Petabyte-scale distributed datasets. The Grid Datafarm system employs replication for fault tolerance. Our work, on the other hand, aims to support fault tolerance by maintaining the state of the system in a distributed environment.

3 Middleware Framework

In this section, we provide an overview of the core framework employed in our work. The framework is designed to support development and execution of data analysis as a network of interacting components in a distributed environment.

3.1 Anthill and Mobius

The framework and its components are implemented using the Anthill [3] and Mobius [4] middleware systems. Anthill supports execution of component-based applications in distributed heterogeneous environments. Its programming model is based on the filter-stream model implemented in a system called Datacutter [1].

In the filter-stream paradigm, an application is implemented as a network of data processing components, referred to as filters. A filter exchange data in application-specific chunks through streams, and has a main processing function that carries out application-specific computation on received data chunks. In many applications, processing of data can be done in a pipelined fashion, in which filters at different stages of the workflow execute concurrently and process. A filter in the system can receive data from multiple upstream filters and send data to multiple downstream filters. Each chunk can be associated with application-defined metadata. The filter-stream paradigm also allows for combined use of task-parallelism (via multiple filters executing concurrently on different hosts) and data-parallelism (through multiple copies of any of the application filters).

Anthill extends the original filter-stream model implemented in Datacutter [1] in several aspects. In particular, it allows applications to be modeled by general digraphs. Another specific feature it incorporates is the mechanism called *labeled stream* which allows the selection, based upon the data being transfered, of one specific filter copy when several are instantiated by the application. This feature is very important when the filter it refers to has a state that is partitioned across several hosts and the data is associated with specific portions of that state.

Mobius [4] is a distributed metadata and data management framework. It provides a virtualization of backend data sources as XML databases and enables on-demand creation of XML databases in a distributed environment such as the Grid. The Mobius Global Model Exchange (GME) service provides support for creating, managing, referencing, and versioning XML schemas under name spaces. Each document stored, indexed, and managed in the Mobius environment is required to conform to a schema registered in GME. The Mobius Mako service enables on-demand creation and management of databases across multiple host machines. Documents conforming to schemas can be stored, queried, and remotely retrieved from Mako instances. The data stored in a Mako is indexed so it can be queried efficiently using

XPath.

In our implementation, Anthill provides the runtime support for distributed execution. The components to support fault tolerance are implemented as Anthill filters. We use the Mobius middleware as our persistent data storage manager.

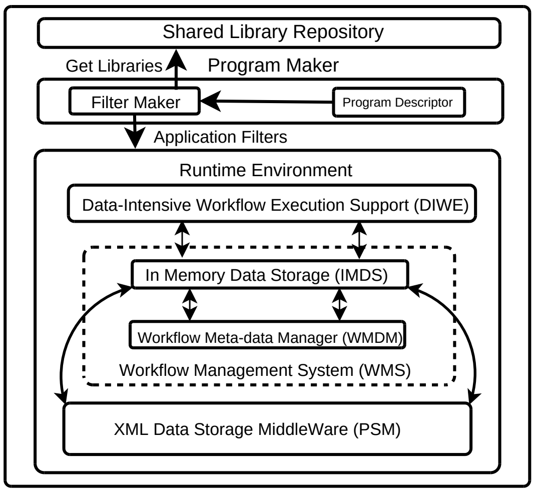
3.2 Architecture of Framework

The framework is composed of three main components (see Figure 1): the shared library repository, the program maker, and the run-time environment. The first two parts allows users to store and share components and provides a toolkit for generating workflows based on shared components from the repository. The run-time environment is divided into a data-intensive workflow execution support system, a distributed workflow management subsystem, and an XML data storage middleware (also referred to here as the *persistent storage manager*). The *data-intensive workflow execution* (DIWE) component is implemented on top of Anthill. The shared library repository, the program maker and the DIWE components are described in more detail in another paper [11]. In this work we introduce the workflow management system and the persistent storage system to support fault tolerance.

3.2.1 Workflow Management System (WMS)

This component is divided into the Workflow Meta-Data Manager (WMDM) subsystem and the In-Memory Data Storage (IMDS) subsystem. The WMDM is responsible for managing the entire workflow execution. The IMDS works as a intermediary between application filters and the persistent storage manager (PSM). We have extended the DIWE component to provide transparent communication with the Workflow Management System (WMS). These extensions facilitate exchange of information between application filters and the WMS. This information includes which filters are available for data processing, which chunks have been processed, and so on.

Workflow Meta-data Manager (WMDM) works as a data manager for the entire workflow execution. This component was developed as an Anthill filter. The WMDM has information about each dataset read or written by the application on the fly. It is also responsible for deciding on demand which portions of the input data is processed by which filter. It monitors and manages the data chunks (messages) received and created on the fly by the workflow. It stores all the meta-data related to these messages. For example, it creates a unique identifier for each message and also knows where it is stored, which filter instance has processed or created the message, etc. It provides a protocol for accessing, creating and updating the necessary metadata during the execution. When the workflow starts running, the WMDM decides which data chunks will be sent to which filter instances. This data partition is done on demand as each time a filter reads input, a request is received by the WMDM. The goal is to always assign a local data chunk to the filter.



Shared Library Repository
Get Libraries
Program Maker
Filter Maker
Program Descriptor
Application Filters
Runtime Environment
Data-Intensive Workflow Execution Support (DIWE)
In Memory Data Storage (IMDS)
Workflow Meta-data Manager (WMDM)
Workflow Management System (WMS)
XML Data Storage MiddleWare (PSM)

Figure 1. Framework Components

In-Memory Data Storage (IMDS) works as a cache between the user application and the PSM. This component was also developed as an Anthill filter. It implements a distributed cache and mechanisms for reading and writing information (e.g., data chunks) from/to the PSM. Based on the meta-data provided by the WMDM, it reads the necessary data from the PSM and stores the outputs of each component. Multiple instances of the IMDS can be instantiated in the environment. Each of these instances has its own memory space to cache data chunks during workflow execution. When the user application is instantiated, the instances of the application filters are assigned to the IMDS instances. This assignment is done such that one IMDS instance is responsible for a subset of the application filter instances. For example, an IDMS instance can be responsible for application filter instances that are running on the same node as the IDMS instance.

During workflow execution, when a filter requests for a data chunk to process, the request is sent to the IMDS instance to which the filter is assigned. To decide which data chunk should be retrieved, the WMDM is consulted – there may be multiple WMDM instances; a WMDM instance is responsible for managing a subset of IMDS instances. Based on the meta-data provided by the WMDM, the IDMS instance reads the necessary data chunk from the PSM, stores it in memory, and sends it to the application filter. This component also stores the intermediate data sent through the streams between application filters, and the output data. When there is no more memory space to store data chunks in an IMDS instance, the IMDS instance sends a subset of data chunks in its cache to the PSM.

3.2.2 Persistent Storage Manager (PSM)

The persistent storage manager is built on top of Mobius [4] which is used to instantiate data stores in the distributed environment. The PSM is used to store the input and output datasets of the application workflows. These datasets are defined by XML schemas and are stored as XML databases. When the application outputs a document, this output is first sent to the IMDS that stores this data in the PSM. To facilitate fault tolerance in the system, the PSM is also used to store data chunks exchanged through application filter streams. Again such data chunks are first sent to the IMDS that interacts with the PSM to efficiently store them.

4 Support for Fault Tolerance

In this section, we present the approach employed in our framework for fault tolerance and describe the protocol used by the different components of the system to enable fault detection and fault recovery.

4.1 Message Logging

A number of mechanisms and protocols have been developed to add fault tolerance and achieve high availability in centralized and distributed systems [12]. The most common mechanisms are based on *checkpointing*, *message logging*, and *rollback recovery*. We employ the message logging approach. In message logging, all messages received by a process are saved in a *message log*, so the application can be restored from its most recent checkpoint by replaying the messages using the same order they were received. In our system, data chunks that are consumed and produced by application filters are check-pointed in the system as message logs.

In order for the system to correctly create message logs and recover an application from faults, the system needs to keep track of dependencies among all data chunks, which are being produced and consumed, and the processing status of each chunk. This information is utilized by the recovery protocol described in Section 4.4. In our framework, we identify four types of dependencies between output data chunks and input data chunks: (1) ***1 to 1*:** this is the simplest type. When a filter receives one message (i.e., one data chunk), it can process the data in this message and send the result to the next filter in the pipeline. (2) ***n to 1*:** a filter has to receive n messages, from one or more streams, before generating one output. The n may vary according to the application and also may assume different values during the application execution. (3) ***n to m*:** this is similar to the *n to 1*, but in this type, m output messages may be generated by one filter, in one or more streams. (4) ***1 to n*:** this is characterized by one filter receiving one message, and generating many outputs using one or more output stream.

Data chunks can assume one of the following three different processing states: (1) **Not Processed**: Every data chunk in the initial dataset is considered *"not processed"* in the beginning of the workflow execution. This means they are available to be processed. (2) **Being Processed**: The data chunk is in the *"being processed"* status when a filter in the workflow gets it to process. All the data chunks created, on the fly, by the workflow filters are considered they are *being processed*, once they were created and sent to other filter to process. (3) **Processed**: One data chunk is considered *"processed"* only when the data chunks that are dependent to it are created and stored in the PSM.

4.2 Creating Message Logs

The creation and management of message logs are coordinated by the WMDM. In our current implementation, the fault tolerance support can recover applications with filters that present data dependency types of *1 to 1* or *n to 1*. Log creation is done as follows. The IMDS is responsible for storing the data chunks exchanged through streams and for keeping the state of each data chunk. For improved fault tolerance, the state of a data chunk

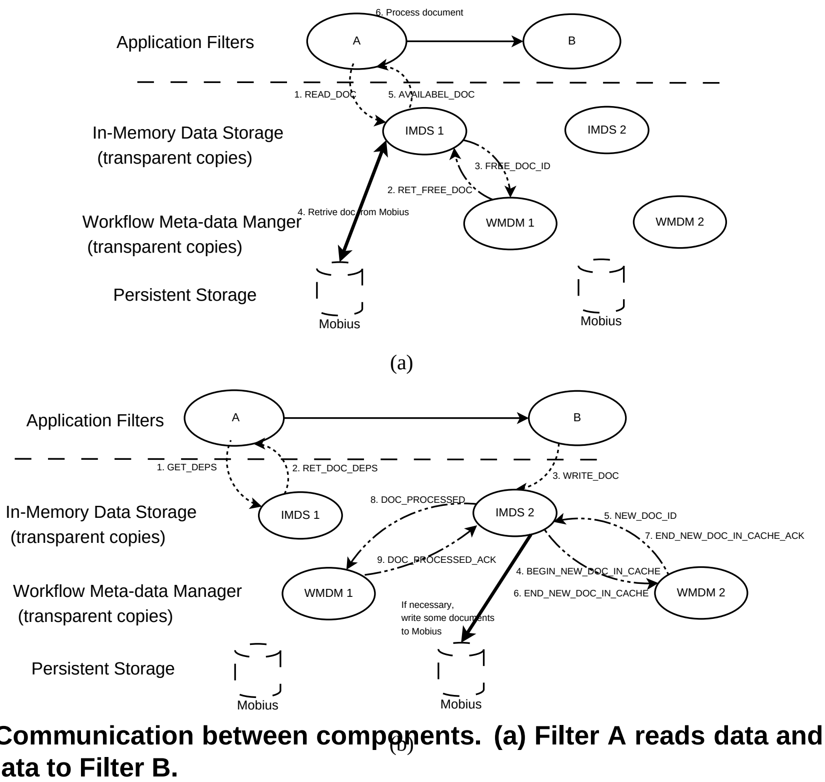


Figure 2. Communication between components. (a) Filter A reads data and (b) Filter A writes data to Filter B.

is also replicated in the WMDM. If either the IMDS or the WMDM fails, the other can be used to reconstruct the global state. When the IMDS receives a data chunk, it stores it in its memory cache. When the available local memory space is exceeded, a subset of data chunks are staged to the PSM. The set and number of data chunks to be staged is determined based on the amount of memory needed to store the new data chunk in memory.

When an application filter instance wants to read a data chunk from storage, a READ_DOC message (shown in Figure 2(a)) is generated and sent to the IMDS. The IMDS asks (via message RET_FREE_DOC) the WMDM for one data chunk, related to the filter A input stream. The WMDM searches its database for a data chunk that is *not processed*. It then returns the meta-data of this data chunk to the IMDS via message FREE_DOC_ID. Using the chunk metadata, the IMDS is able to retrieve the data chunk either from its local memory or from the PSM (by sending the AVAILABLE_DOC message).

After filter A instance has generated an output data chunk, it can send the output data chunk to upstream filter B. With the output data chunk, its dependencies are also transferred. The dependency information will be needed later to update the status of the data chunks to *processed*. Thus, messages (GET_DEPS) and (RET_DOC_DEPS) in Figure 2(b) are used to retrieve data chunk dependency information from the IMDS.

When filter B in Figure 2(b) receives a data chunk through its input stream, the data chunk is also stored in the IMDS and the dependencies of this chunk are updated to the *processed* status. The chunk and its dependency information are transferred to the IMDS with the WRITE_DOC message. The IMDS checks the available memory to cache this chunk. If no space is available, some of the cached data chunks are staged to the PSM. The IMDS then notifies the WMDM that one data chunk was created by one of the application filters and that the chunk needs an identifier to be stored. For identifier creation,a *Two Phase Commit Protocol* was used. This protocol is carried out as follows. With message BEGIN_NEW_DOC_IN_CACHE, the IMDS notifies the WMDM of the new data chunk. An identifier is created for the chunk and is returned to the IMDS in message NEW_DOC_ID. To complete the protocol, messages 6 and 7 are also transmitted. At the end of message exchanges, both the IMDS and the WMDM know that the data chunk has been stored in the IMDS with the correct identifier. The next step is to update the dependencies of the data chunk to *processed* in the WMDM. This is done via message DOC_PROCESSED. Message DOC_PROCESSED_ACK is sent back to the IMDS to make sure the WMDM has received and updated the status.

4.3 Detecting Faults

Our fault detection mechanism is built on the mechanisms included in the Anthill infrastructure, which are based on the Parallel Virtual Machine (PVM) fault detection package. In our fault model, we define two types of failures: process faults and node faults. All other faults are mapped to one of these two types. A process fault occurs, when a process exits unexpectedly or is killed. When a machine becomes unreachable (due to network failure or machine crash), it is treated as a node fault. Node faults are detected through timeouts. Thus, unlike process faults, a node fault may not be detected immediately. In our framework, we assume any of the following components can fail: any application filter instance, the IMDS, the WMDM, and the PSM. The failure of an application filter instance, or the IMDS, or the WMDM is detected via the Anthill fault detection mechanism, since these components are implemented as Anthill filters. The PSM is considered in failed state, when the IMDS or the WMDM tries to communicate with it N times without success[1].

4.4 Recovering from Faults

In this section we describe how the system recovers from the three different points of failure: user application filter, the IMDS and the WMDM. We assume that these components can fail at any time of the execution. Once one fault has occurred, another fault cannot happen until the the system is recovered. In our current implementation, the system cannot recover using a saved state, if the PSM fails. That is, the execution is stopped and initialized from the beginning.

4.4.1 Application Filter Faults

When a fault occurs in an application filter, all the application filter instances are killed by the system and the workflow is restarted. This approach is chosen because the state of the workflow can be reconstructed from the dependency information and the data chunk state information saved in the IMDS and/or the PSM. In the recovery phase, the system has to determine which chunks will have to be re-processed and re-generated and when they will be re-processed. Re-processing of data chunks, which were generated by a filter but not yet processed by the next filter, can be carried out using one of three different approaches. The first is to re-process the data chunks at the end of execution of the workflow. Another approach is to re-process these chunks first, and then start processing the remaining data chunks. The last approach is to start processing new chunks and re-processing data chunks that have been saved at the same time. Our current implementation employs the last strategy.

[1]In our implementation ,the default value of N is 10.

4.4.2 In-Memory Data Storage Faults

When an instance of the IMDS fails, the Anthill fault mechanism restarts the failed IMDS instance. It then sends a notification message to all other instances of the IMDS. The restarted IMDS instance and the other IMDS instances all execute the recovery step. This is because, during the execution of the workflow, it is not guaranteed that the metadata associated with data chunks, i.e., the processing status of a data chunk (*not processed*, *being processed*, *processed*), is stored in the same IMDS instance, where the data chunks themselves are stored. The IMDS instances ask the WMDM (i.e., all of its instances) for the data chunk states stored in the WMDM. This is done to guarantee that in the beginning of an execution, after a failure, all data chunks stored in the IMDS instances will have the correct status. The failed IMDS filter also needs to notify the WMDM that the data chunks that it locally maintained were lost and that it will have to do a rollback based on data chunk dependencies. For a data chunks that is lost, the status of its dependencies must be updated to *not processed*, so those data chunks are retrieved and re-processed in the workflow.

4.4.3 Workflow Meta-data Manager Faults

When an instance of the WMDM fails, the Anthill fault mechanism restarts the failed WMDM instance and notifies the other instances. In a fault, all of the state of the WMDM instance is lost. The restarted instance initializes the data chunk state information based on the initial client query, which defined the input set of data chunks. Then, it asks all the IMDS instances, which are assigned to this WMDM instance, for their data chunk states. Afterward, it asks all the IMDS instances for the data chunks that are *processed*. After this last step, the internal state of the WMDM instance has been reconstructed.

The next steps are executed by all the WMDM instances. The first step is to verify if there is any pending transactions that need to be executed. The WMDM waits for the IMDS notification saying, if there are any transactions to be executed. The second step is to update all the data chunks with state "being processed" to "not processed". These data chunks are the data chunks that were being processed just before the fault occurred, or that had just been read by the IMDS, or that have been processed but have not had their state updated to "processed".

5 Experimental Results

We have evaluated the performance of our framework and the cost of recovering from faults using a biomedical image analysis application. The workflow of the application is given in Figure 3. We used a dataset consisting of 866 images, acquired from tissue specimens from a mouse placenta using a high-resolution digital microscopy scanner. The size of the dataset is 23.94 Gigabytes. In our experiments, the dataset was stored in the Persistent Storage Manager across multiple nodes in the system. The experiments were run on a cluster of 20 PCs, which are connected using a Fast Ethernet Switch. Each node has a AMD Athlon(tm) Processor 3200+ and 2 GB main memory and runs Linux 2.6 as the operating system. In the experiments one IMDS instance was instantiated on each cluster node and one WMDM on one of the nodes.

The first set of experiments evaluated the system's performance, when partial results from a stage are saved in the system, as the number of nodes is scaled. In Figure 4, the execution time of the last stage of the workflow is shown, when the partial results form the "Color Classification" filter are saved or not saved in the system. As is seen form the figure, the execution time decreases as more nodes are added for application execution. In addition, the overhead is very small and about 6.7% of the total execution time on average.

To evaluate the cost of fault detection and recovery, the following types of faults were introduced during execution: (1) one instance of the application's filters is killed around halfway through the experiment. (2) one filter instance is killed in the first part of the experiment and other in the second part of the experiment. (3)

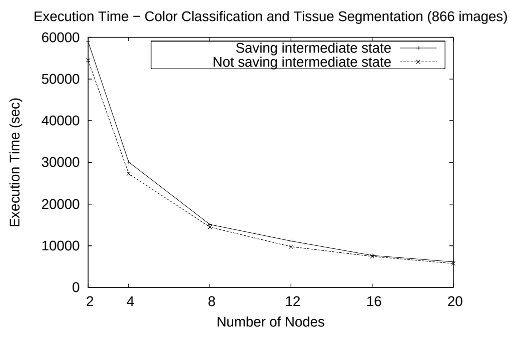


Figure 4. Performance results for the color classification and tissue segmentation stages.

The experiment is divided in three parts and one filter instance is killed in each of them. (4) The IMDS or the WMDM is killed around the halfway through the experiment. Figure 5 shows the increase in the execution time when each of the fault types are introduced in the application. In the charts, "Test Time" denotes the total execution time and "Processing Function" is the time spent in processing data in the process function of the filter. The processing function time increases when the system recovers from faults, as some of the data has to be reprocessed. Another factor contributing to the increase in the total execution time is the fact that all the application filters have to be restarted and the data lost during fault should be read again. However, as the charts show, the cost of recovery is low when compared to the overall execution time.

6 Conclusions

This paper proposed a framework to incorporate fault detection and recovery in workflows that process large volumes of data in a pipelined fashion. The framework employs a "message logging" approach and implements a distributed mechanism to maintain intermediate results and messages exchanged among application components. The experimental evaluation shows that it introduces little overhead, scales well, and can enable fast recovery from certain types of faults. This makes the framework a viable platform for long running, data-intensive workflows.

References

[1] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing

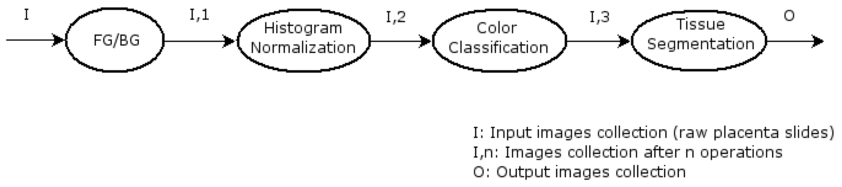
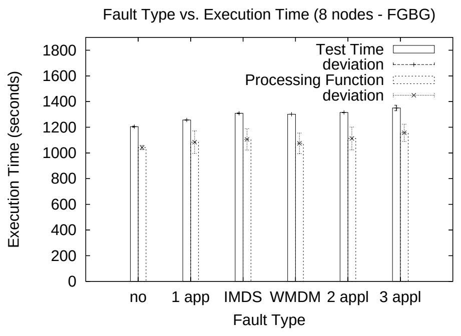
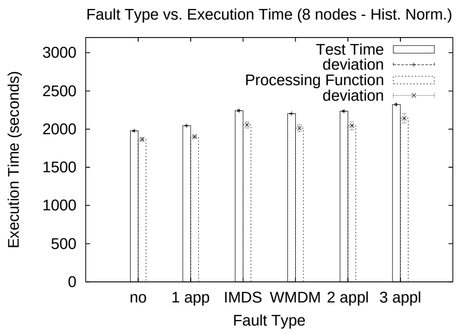

I
FG/BG
I,1
Histogram Normalization
I,2
Color Classification
I,3
Tissue Segmentation
O
I: Input images collection (raw placenta slides)
I,n: Images collection after n operations
O: Output images collection

Figure 3. Application workflow. Four different analysis operations are applied to digitized microscopy images.



(a)



(b)

Figure 5. Performance of the system when faults are introduced.

of very large datasets with datacutter. *Parallel Comput.*, 27(11):1457–1478, 2001.

[2] S. Bowers, B. Ludascher, A. H. Ngu, and T. Critchlow. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. In *Proceedings of IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow 2006)*, Atlanta, GA, April 2006.

[3] R. A. Ferreira, W. M. Jr., D. Guedes, L. M. A. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Arajo, and G. T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *17th International Symposium on Computer Architecture and High Peformance Computing*, Rio de Janeiro, RJ, 2005.

[4] S. Hastings, S. Langella, S. Oster, and J. Saltz. Distributed data management and integration: The mobius project. In *Global Grid Forum Semantic Grid Applications Wkshp*, Jun. 2004.

[5] S. Hastings, M. Ribeiro, S. Langella, S. Oster, U. Catalyurek, T. Pan, K. Huang, R. Ferreira, J. Saltz, and T. Kurc. Xml database support for distributed execution of data-intensive scientific workflows. *SIGMOD Record*, 34, 2005.

[6] G. Kola, T. Kosar, J. Frey, M. Livny, R. J. Brunner, and M. Remijan. Disc: A system for distributed data intensive scientific computing. In *Proceeding of the First Workshop on Real, Large Distributed Systems (WORLDS'04)*, San Francisco, CA, December 2004.

[7] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[8] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. In *Proceedings of 11th European Conference on Parallel Processing (Euro-Par 2005)*, Lisbon, Portugal, August 2005.

[9] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 18(10):1039–1065, 2005.

[10] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2002.

[11] G. Teodoro, T. Tavares, R. Ferreira, T. Kurc, W. Meira, D. Guedes, T. Pan, and J. Saltz. Run-time support for efficient execution of scientific workflows on distributed environmments. In *International Symposium on Computer Architecture and High Performance Computing*, Ouro Preto, Brazil, October 2006.

[12] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. Dec. 2005.