Published in final edited form as:
Int J Parallel Program. 2008 April ; 36(2): 250–266. doi:10.1007/s10766-007-0068-8.

A Run-time System for Efficient Execution of Scientific Workflows on Distributed Environments[*]

George Teodoro[*], **Tulio Tavares**[*], **Renato Ferreira**[*], **Tahsin Kurc**[†], **Wagner Meira Jr.**[*], **Dorgival Guedes**[*], **Tony Pan**[†], and **Joel Saltz**[†]

George Teodoro: george@dcc.ufmg.br; Tulio Tavares: ttavares@dcc.ufmg.br; Renato Ferreira: renato@dcc.ufmg.br; Tahsin Kurc: kurc@bmi.osu.edu; Wagner Meira: meira@dcc.ufmg.br; Dorgival Guedes: dorgival@dcc.ufmg.br; Tony Pan: tpan@bmi.osu.edu; Joel Saltz: jsaltz@bmi.osu.edu

[*]Department of Computer Science, Universidade Federal de Minas Gerais, 31270-010 Belo Horizonte, MG - Brazil, tel +55(31)3499-5860 - fax +55(31)3499-5858

[†]Department of Biomedical Informatics, The Ohio State University, Columbus, OH, 43210 - USA, tel +1(614)292-4778 - fax +1(614)688-6600

Abstract

Scientific workflow systems have been introduced in response to the demand of researchers from several domains of science who need to process and analyze increasingly larger datasets. The design of these systems is largely based on the observation that data analysis applications can be composed as pipelines or networks of computations on data. In this work, we present a runtime support system that is designed to facilitate this type of computation in distributed computing environments. Our system is optimized for data-intensive workflows, in which efficient management and retrieval of data, coordination of data processing and data movement, and check-pointing of intermediate results are critical and challenging issues. Experimental evaluation of our system shows that linear speedups can be achieved for sophisticated applications, which are implemented as a network of multiple data processing components.

Keywords

Scientific Workflows; Parallel Computing; Data-analysis

1 Introduction

Data analysis is a significant activity in almost every scientific research project. Challenges in designing and implementing support for efficient data analysis are many, mainly due to characteristics of scientific applications that generate and reference very large datasets. Large datasets are often generated by large scale experiments or long running simulations. One example is the Large Hadron Collider project at CERN. Starting this year, this project is expected to generate raw data on a petabyte scale from four large underground particle detectors every year [1]. Projects like the Grid Datafarm [2] are being implemented to be able to process these datasets.

To help the researchers in their experiments and analysis, scientific workflow systems [3, 4, 5, 6] have been introduced. In most scientific applications, analysis workflows are data-

[*]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.

centric and can be modeled as *dataflow process networks* [7]. That is, a data analysis workflow can be described as a directed graph, in which the nodes represent application processing components and the directed edges represent the flow of data exchanged between these components.

Distributed environments, like a PC cluster or collection of PC clusters, provide viable platforms to efficiently store large datasets and execute data processing operations. In a scientific workflow system, the user should be able to describe and create components based on the tasks they want to execute, arrange these components into a network of operations on data based on the application data processing semantics, and run the network of components on very large data collections on clusters of storage and computation nodes. Scientific workflow systems should also support component reuse. In other words, a component may be part of a specific workflow, but also can be reused in another application workflow. An example data analysis workflow in an image analysis application is shown in Figure 1. This example involves analysis of digital microscopy slides to study the phenotype changes induced by some genetic manipulations. In the figure, we can see four different tasks (image analysis operations) that should be applied in sequence to the slides. In summary, some of the challenges in designing workflow systems that support processing of large datasets are 1) to store, query and manage large distributed databases, 2) to manage the input and output data and the scheduling and monitoring of these workflows execution in the distributed environment, and 3) to optimize the reuse of components in different workflows.

We proposed and developed the Anthill system [8], a system based on the filter-stream programming model that was originally proposed for Active Disks [9], to address some of the issues in execution of scientific data-intensive workflows. In Anthill, filters represent different data processing components of the data analysis structure and streams are an abstraction for communication between filters. Using this framework, applications are implemented as a set of filters over the network connected using streams, creating task parallelism as in a pipeline. During execution, multiple copies of each filter can be instantiated, allowing every stage of the pipeline to be replicated, resulting in data parallelism. In an earlier work [8], we demonstrated the efficacy and efficiency of Anthill for data mining tasks.

In this paper, we report on the results of an effort to extend the functionality of Anthill. These extensions include 1) a *program maker* component, which builds workflow executables from dynamically loadable shared libraries and workflow description files, 2) a persistent storage layer, which provides support for management of meta-data associated with workflow components, storage and querying of input, intermediate, and output datasets in workflows, and 3) in-memory storage (cache) layer, which is designed to improve performance when data is check-pointed or stored in and retrieved from the persistent storage layer. The persistent storage layer builds on Mobius [10], which is a framework for distributed and coordinated storage, management, and querying of data element definitions, meta-data, and data instances. Mobius is designed as a set of loosely coupled services with well-defined protocols. Data elements/objects are modeled as XML schemas and data instances as XML documents, enabling use of well-defined protocols for storing and querying data in heterogeneous systems.

The extensions presented in this paper are generic in the sense that they can be applied in a range of situations, and our experiments have shown that we incur low overhead during execution.

2 Related Work

The Chimera [11] project has developed a virtual data system, which represents data derivation procedures and derived data for explicit data provenance. This information can be used for reexecuting an application and regenerating the derived data. Our approach focuses on storing the partial data results; we do not store a large amount of information about data derivation, but we are able to efficiently store datasets generated between each pipeline stage. The Pegasus [12] can create a virtual data system that saves the information about data derivation procedures and derived data using Chimera. It also maps Chimera’s abstract workflow into a concrete workflow DAG that the DAGMan [13] meta-scheduler executes. The Kepler [5, 6] system provides support for Web Service based workflows. The authors show the composition of workflows based on the notion of actor oriented modeling, first presented in PTOLEMY II [14]. Pegasus and Kepler systems have interesting solutions to the workflow management problem. However, they do not directly address the problem of integrating workflow execution with data management and retrieval. Our system is constructed to support efficient access to data stored in distributed databases and scalable execution of workflows in an integrated manner. NetSolve [15] provides access to computational software and hardware resources, distributed across a wide-area network. To support sharing of software resources available in the network, NetSolve creates an infrastructure to call shared libraries that implement the available functionalities. The other features of NetSolve include support for fault-tolerance and load balancing across computational resources.

3 Extended Anthill Framework

The architecture of the extended Anthill system, as shown in Figure 2, is composed of two main parts: the program maker and the run-time environment. The first part allows users to store and share data processing components in a repository and provides a toolkit for generating workflows based on shared components from the repository. The run-time environment is designed to support analysis workflows in data intensive applications. The run-time environment is further divided into a distributed workflow meta-data manager, a distributed in-memory data storage, and a persistent storage system. The *workflow meta-data manager* (WFMDM) works as a data manager for the workflow execution. It stores information for datasets read or written by the application on the fly. It is also responsible for deciding on demand which portions of the input data are processed by each filter. Note that the WFMDM can be executed in distributed fashion across multiple machines or as a centralized entity. The *in-memory data storage* (IMDS) subsystem works as an intermediary between the application and the Persistent Storage Management System (PSM). Based on the meta-data provided by the WFMDM, the IDMS basically reads the necessary data from the PSM and stores the outputs of each component in the PSM. The PSM uses the Mobius framework [10] to expose and virtualize data resources as XML databases and to allow for ad hoc instantiation of data stores and federated management of existing, distributed databases. The system also provides mechanisms for efficiently saving partial results without introducing synchronization between the application and the run-time environment.

We now proceed to detailing the implementation of each of these components. They are designed to achieve scalable and efficient execution in distributed and heterogeneous environments.

3.1 Program Maker

This component is a tool for allowing users to incorporate existing program components and libraries in the workflow system. To accomplish this task, it creates additional code in each stage of the workflow pipeline to support execution of program executables and dynamically loadable shared libraries. The Program Maker is divided into three parts: Shared Libraries and Executables Repository, Program Descriptor and Filter Maker.

3.1.1 Shared Libraries and Executables Repository—In our framework data analysis workflows can be created from dynamically loadable libraries and program executables. We have developed a repository component to enable management of function libraries and executables so that users can store and search for application processing components and use them in workflows. To support efficient management and querying of the repository, we implemented it using Mobius [10]. The user can interact with the repository via three basic operations: upload, search for, and download programs and libraries.

The first operation, *upload*, requires the creation of meta-data that describes the compiled code being uploaded. This meta-data, which is implemented as an XML document, contains all the information necessary to identify the type of data the program is able to work with, the data structures used by each of its arguments, and the de-serialization and serialization functions that need to be applied to the input and output datasets of the program. It also includes additional information about the system requirements of the particular program or library (e.g., hardware platform requirements, dependencies on other libraries). The second operation, *search*, is used to perform queries to search for stored libraries and program executables and to access the meta-data related to each stored element. The last operation, *download*, receives a reference to a compiled code or library, downloads it from the repository, and stores it in a local directory.

3.1.2 Program Descriptor—This is the configuration file (represented as an XML document) of the entire data processing pipeline of an application. It is divided into four sections: *hostDec*, *placement*, *layout*, and *compiledFilters*.

- *hostDec* is used to describe all machines available in the environment. It is used to determine the resources for each of the application components.
- *placement* is used to declare the components comprising a particular workflow application, the library in which they are located, and the number of instances that should be created for each component.
- *layout* defines the connections between the components, the policies associated with each connection (e.g., each data buffer exchanged between two components over the connection can be check-pointed), and the direction of communication.
- *compiledFilters* is used to provide information that the framework needs to be able to execute a given component. Information here is used to find out which library the component code comes from, the number and types of parameters that should be passed or returned to/from the component, and the data transformation functions that need to be called to serialize/de-serialize the input and outpur data of the component.

3.1.3 Filter Maker—This component receives a Program Descriptor configuration file and executes the workflow described in that file. It generates the source code of the *connection filter* for each application component declared in the configuration file, as well as the *Makefile* required for compiling and linking the entire application workflow. The user should define an environment variable pointing to the directory where it is stored so that the filter maker subsystem can determine the location of the application-specific libraries to be linked to the workflow. The *connection filter* wraps the application specific data processing component so that it can be executed properly. A high level definition of the connection filter is given in Algorithm 1. It executes a loop that reads data from the input stream, deserializes and passes it to the application component code, which is invoked in the *process* method. It then proceeds to serializing any output that is generated by the application component and sending it out the next filter in the workflow.

3.2 Run-Time Environment

The run-time environment, shown in Figure 2, is divided into three main components: the Data-Intensive Workflow Execution Support System, which is responsible for instantiating the workflow program, the Workflow Management System, which is responsible for managing the entire work-flow execution, and the Persistent Storage System.

3.2.1 Data-Intensive Workflow Execution Support System—This component is implemented on top of Anthill [8], which is responsible for instantiating the components on distributed platforms and managing the communication between them. Anthill is based on the filter-stream programming model, which means that in this environment applications are decomposed into a set of filters that communicate through streams. At execution time, multiple instances of each filter can be spawned on different nodes on a distributed environment, achieving data parallelism as well as pipelined task parallelism.

We have extended the Anthill run-time to provide transparent communication between the application and the Workflow Management System (WMS). These modifications provide support for exchanging information across application components and the WMS. This information includes, for instance, which filters are available for data processing, which documents have been processed, and so on.

3.2.2 Workflow Management System—This component is divided into two subcomponents: the Workflow Meta-Data Manager (WFMDM) and the In-Memory Data Storage (IMDS). The WFDMD works as the data manager of the entire workflow execution. It maintains information about all the data involved in the application execution, either read or written. When the workflow execution is initiated, the WFMDM receives a XPath query [16] that specifies the input dataset. It then relays the query to all instances of the Persistent Storage Manager (PSM) and builds a list of all matching documents with the associated meta-data. Each document of the list goes through three different states as the execution progresses:

Not processed: This state applies to all documents that compose the input dataset at the beginning of the execution. It means that they are available to be processed.

Being processed: input documents sent to filters are in this status as well as documents sent across filters, because they have been created and are being processed by one or more filters.

Processed: a documents is marked processed when it has been processed by a filter and the result has been stored in the IMDS.

During the workflow execution, the WFMDM is responsible for assigning documents to filters. This data partitioning is done on demand as each time a filter reads input data, a request is received by the WFMDM. The goal is to always assign a local document to the filter.

The IMDS works as an intermediary between application filters and the persistent storage manager (PSM). It is implemented as a filter, which is instantiated on multiple machines based on user configuration. The system always tries to have filter requests for data answered by a local IMDS. When there is no local IMDS for a given filter, another one is assigned to the filter by the run-time system.

As filters request data during execution, these requests are passed down to the local IMDS (or to the assigned one). The IMDS acts pretty much as a caching system, only relaying requests for unavailable data to the WFMDM. Several instances of the IMDS can be distributed across available machines and work independently, meaning that multiple instances can be reading different portions of the data simultaneously. This is similar to a classic parallel I/O approach, except that it is on top of a distributed XML database.

The task of saving intermediate results is also executed by the IMDS. It can save all data sent through the stream. During execution, the IMDS creates, on the fly, distributed databases for each stream and stores all the data exchanged over a given stream as documents in Mobius. It behaves like a write-back caching mechanism, releasing the application code from having to wait for the I/O operation to complete. As in the case of reads, multiple write operations can be executed in parallel.

3.2.3 Persistent Storage Manager—We use Mobius [10] as our persistent data storage manager. We employ the Mobius Mako services to store all data used in workflows. The Mobius Mako provides a platform for distributed storage of data as XML documents. Databases of data elements can be created on-demand. The data is stored and indexed so that it can be queried efficiently using XPath. Data resources are exposed to the environment as XML data services with well-defined interfaces. Using these interfaces, clients can access a Mako instance over the network and carry out data storage, query, and retrieval operations.

3.2.4 Communication Protocol—Application components (implemented as filters) communicate with the rest of the run-time support transparently. Each application component is just concerned with receiving its own input data, processing it, and generating its output. In Figure 3 we illustrate the internal communication structure across the several components of the run-time infrastructure. We use a stage in the pipeline of an image processing application (described later, see Figure 7) as an example. As seen in the figure, there are two filters involved in that stage: color classification and tissue segmentation, both being fairly standard image analysis algorithms.

In Figure 3(a), we detail the communication within the run-time components for the case of a read operation (assume the first filter, color classification, is reading its next image). The process starts with a message from the application's filter requesting the next document from the IMDS. This operation will then invoke a request to the local WFMDM instance for an available document for processing, and will receive an ID of some document, potentially available locally. With that information, the IMDS can serve the original requester with the data. It may need to query the PSM, if the data is not available in the IMDS already.

Figure 3(b) illustrates the communication protocol for write operations. It is a slightly more complicated protocol. As the color classification code outputs its data, the filter has to first create the dependencies (i.e., the documents used to create other documents) on the local IMDS instance. After that, the data is sent from one filter to the next, using the streams infrastructure within Anthill. Once the filter on the receiving end gets the data, it creates a local copy of the data before passing it to the application code. As this copy of the data is stored locally, the IMDS notifies the local WFMDM instance about the local data copy and the sender's WFMDM instance that the data was successfully received. This will prompt a change the change in the state (to *processed*) of the input document that generated that particular output document. The IMDS will eventually move the data from its memory to the PSM. This happens in background so that the application is not penalized.

3.3 Support for Matlab Filters

In biomedical image analysis studies Matlab is a commonly used system. Through its scripting capabilities and built-in libraries and functionality, it provides an environment for researchers to quickly prototype their algorithms and evaluate them. It also provides compilation functionality by which a Matlab program can be compiled into a shared library or an executable. We developed support in the framework described in this paper to facilitate composition and execution of work-flows consisting of Matlab programs compiled as shared libraries. In this section, we describe how to create an XML configuration file for applications whose filters are generated using a compiled Matlab code. The first three sections of the configuration file (hostdec, placement and layout), are consecutively used to describe the available machines, the filters in the pipeline and how to connect these filters. For filters, that are automatically generated from compiled Matlab executables or shared libraries, we add an extra section in the configuration file; this section is marked as (`matLab`). In the `matLab` section, for each filter metadata such as filtername and libraryname are specified. After analyzing the `matLab` section of the configuration file, our system determines which filters correspond to Matlab shared libaries and the inputs and outputs of these filters. At this point, we can generate the filter's code to call these functions. Figure 4 shows how the conf file is translated in filters.

In the configuration file, for each Matlab filter, a <matLabFilter> tag is created. This tag contains the following two attributes:

- name: Name of the filter that use compiled code
- matlablibname: Name of the shared library where the function that implements this filter is located.

The function that will be used by the filter should also be declared. This can be done using the <function> tag. The attributes of this tag are:

- headername: The header name of the function
- numoutputs: The number of arguments used as outPut (reference)
- numinputoutputs: The number of arguments used as input and output
- numinputs: Number of arguments that are used just as a function input

Finally, each argument of the function is specified using the <argument> tag. This tag has the following attributes:

- argType: The type of this argument.
- inputType: is used to know if this variable is initialized from a user line comand of from message.
  - userargindex: Its similar to the index of argv variable in a C program call
  - msgindexin: Identifies the location of input in the received msg
- order: The order of the parameters in the function call
- serializefunction: The output (inputoutput) argument identifies the serialize function that should be used to pack the data in the output msg
- serializelibname: the name of the library that contains the serialize function.
- deserializefunction: The input (inputoutput) argument most inform what are the deserialize function that should be used to unpack the data in the in msg before call the function that implements the filter

- deserializelibname: the name of tha lib that contains the deserialize function.
- msgIndexOut: identifies the order that the serialize functions must be called

The user can provide serialization and deserialization functions. We also developed a suite of serialization/deserialization functions for common Matlab data types.

An example configuration file for a workflow composed of Matlab filters is given in Figure 5.

4 Application Example

In this section we briefly describe an example application and how it is mapped into a workflow using the tools available in our framework.

4.1 Application Overview

The example application uses high-resolution digitized microscopic imaging to study phenotype changes in mouse placenta induced by genetic manipulations. It handles the segmentation of images that compose the 3D mouse placenta into regions corresponding to the three tissue layers: the labyrinth, spongiotrophoblast, and glycogen, as described in [17].

We have divided this application into six stages, as seen in Figure 6, and mapped four of the most expensive stages as the components of the workflow. The basic description of each of the four stages are:

Foreground/Background Separation (FG/BG): Images are converted from the RGB color space to the CMYK color space and a combination of the color channels are thresholded to get the foreground tissue.

Histogram Normalization: Images are corrected for color variations. This process consists of three sub-operations: computing the average colors for the images; selecting one image as the color normalization target; and generating a histogram for each of the red, blue and green channels.

Color Classification: Pixels in an image are classified using a Bayesian classifier. The classifi-cation of a pixel puts it in one of 8 different categories: dark nuclei, medium intensity nuclei, light nuclei, extra light nuclei, red blood cell, light cytoplasm, dark cytoplasm, and background.

Tissue Segmentation: In this step, using a Bayesian classifier, each tissue is classified into one of the three tissue types: Labyrinth, Spongiotrophoblast, and Glycogen.

In the rest of this section we describe how a developer can implement this application using our system.

4.2 Application filters

The main work to integrate an application into our system consists in constructing the *com-piledFilter* section of the Program Descriptor configuration file, describing the entire data analysis pipeline of the application (see Section 3.1.2). In the *compiledFilter* section, the user describes details about each filter that perform application specific data processing functions. Due to space limitations, we do not elaborate on the format of the configuration file other than to say that it is a XML document containing a detailed description of each of the application components (filters), with all the information required for automatically generating filters.

4.3 Application Workflow

In the workflow composition phase, the user needs to specify what filters are in the workflow and the connection between them. This information is part of the *placement* and *layout* sections of the Program Descriptor file. After this information is specified, the user can call a script with the program parameters and a XML query that identifies the data elements, which are stored in and managed by the PSM, that should be processed.

In the example application, inputs needed by stage 3 are the outputs of stage 2, so we have a clear data dependency between them. During the execution of stage 2, our framework creates new data collections on the fly across available machines and stores the output data elements and the related meta-data to be used as stage 3 input. Once stages 2 and 3 have completed execution, stage 4 can be executed. Again the the framework takes care of storing the output from this stage. Stage 6 has a data stream between two application filters. Figure 7 shows this stream and a dotted arrow from it to the WMS. This arrow represents an optional efficient stream storage mechanism. This feature allows storage of partial results during execution. This check-pointing facility can be used to re-start the execution from the last set of stored partial results.

5 Experimental Results

In this section we evaluate the implementation of the example image analysis application developed using the framework described in this paper. 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

To evaluate our implementation, we used a dataset of 866 images that have been created by digitizing sections from a mouse placenta, as described in [17]. The size of the whole dataset is 23.49 GB. The dataset has been stored in the PSM; we ran one Mobius Mako service instance on each storage node and distributed the images in the dataset across multiple Mako nodes in round-robin fashion. During the experiments, we instantiated one IMDS on each machine and one WFMDM instance on one of the machines.

Figure 8 shows an experimental evaluation of the Foreground/Background Separation (FG/BG) stage. The numbers illustrate the good scalability of our system, which achieves almost linear speed-up as seen in Figure 8(b). In Figure 8(a), the details of the execution time are shown. The results show that the execution time is dominated by the time spent in the process function.

Figure 9(a) shows the speed-up results of the Histogram Normalization stage. This stage uses images and associated image masks as input. The execution time using 2 machines is about 7000 seconds and the speed-up is almost linear. Figure 9(b) shows the speed-up of the last and most ex- pensive stage, the “Color Classification” and “Tissue Segmentation”. For this stage, the execution time using 2 machines is about 60000 seconds and the speed-up is almost linear.

Figure 10 shows the performance of the system when partial results from a stage is saved in the system. In the figure, the execution time of the last stage of the application is shown, when the partial results from the Color Classification filter are saved or not saved in the system. As is seen from the figure, the overhead is very small and less than 5% on average.

6 Conclusion and Future Work

In this paper we have presented extensions to a run-time system, Anthill, for efficient execution of scientific workflows on distributed environments. The new components can create a stub Anthill filter (also called the connection filter) automatically from a high level description of a given application component. These filters can run user code with a simple interface.

In order to provide data management with low overhead, we use the Mobius infrastructure. The modules of the run-time support are also built as a set of Anthill filters which communicate among themselves and with Mobius transparently to the user code. Our experiments have shown that our implementation can be used to execute sophisticated applications, with multiple components, with almost linear speedups. This means that our system imposes very little overhead.

Our next step is to work toward building a robust, dependable workflow system. Fault tolerance is important in any environment with a large number of machines and processes running for nontrivial periods of time. We plan to use the efficient data management mechanisms presented in this paper to store data checkpoints and allow applications to resume execution from check-pointed data.

References

1. CERN. Large hadron collider. http://www.interactions.org/lhc/
2. Tatebe, O.; Morita, Y.; Matsuoka, S.; Soda, N.; Sekiguchi, S. Grid datafarm architecture for petascale data intensive computing. 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid); 2002.
3. Kola, George; Kosar, Tevfik; Frey, Jaime; Livny, Miron; Brunner, Robert J.; Remijan, Michael. Disc: A system for distributed data intensive scientific computing. Proceeding of the First Workshop on Real, Large Distributed Systems (WORLDS'04); San Francisco, CA. December 2004;
4. Hastings S, Ribeiro M, Langella S, Oster S, Catalyurek U, Pan T, Huang K, Ferreira R, Saltz J, Kurc T. Xml database support for distributed execution of data-intensive scientific workflows. SIGMOD Record. 2005; 34
5. Altintas, I.; Berkley, C.; Jaeger, E.; Jones, M.; Ludscher, B.; Mock, S. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In the 16th Intl. Conference on Scientific and Statistical Database Management(SSDBM); Santorini Island, Greece. June 2004;
6. Ludascher, B.; Altintas, I.; Berkley, C.; Higgins, D.; Jaeger-Frank, E.; Jones, M.; Lee, E.; Tao, J.; Zhao, Y. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows. 2005. Scientific workflow management and the kepler system.
7. Lee, Edward A.; Parks, Thomas M. Dataflow process networks. Proceedings of the IEEE; may 1995; p. 773-799.
8. Ferreira, R.; Meira, W., Jr; Guedes, D.; Drummond, L.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R.; Ferreira, G. Anthill: A scalable run-time environment for data mining applications. Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD); 2005.
9. Acharya, A.; Uysal, M.; Saltz, J. Active disks: Programming model, algorithms and evaluation. Eighth International Conference on Architectural Support for Programming Languages and Operations Systems (ASPLOS VIII); Oct 1998; p. 81-91.
10. Hastings, Shannon; Langella, Stephen; Oster, Scott; Saltz, Joel. Distributed data management and integration framework: The mobius project. Global Grid Forum 11 (GGF11) Semantic Grid Applications Workshop; IEEE Computer Society. 2004. p. 20-38.
11. Foster, Ian; Voeckler, Jens; Wilde, Michael; Zhao, Yong. Chimera: A virtual data system for representing, querying, and automating data derivation. The 14th International Conference on Scientific and Statistical Database Management (SSDBM'02); 2002.

12. Deelman E, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Lazzarini A, Arbree A, Cavanaugh R, Koranda S. Mapping abstract complex workflows onto grid environments. In Journal of Grid Computing. 2003:25–39.
13. Frey, James; Tannenbaum, Todd; Foster, Ian; Livny, Miron; Tuecke, Steven. Condor-G: A computation management agent for multi-institutional grids. Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10); IEEE Press. Aug 2001;
14. PTOLEMYII project. Department of EECS; US Berkeley: 2004. http://ptolemy.eecs.berkeley.edu/ptolemyII/
15. Casanova H, Dongarra J. Netsolve: A network enabled server for solving computational science problems. International Journal of Supercomputer. 1997:212–223.
16. Berglund, Anders; Boag, Scott; Chamberlim, Don; Fernández, Mary F.; Kay, Michael; Robie, Jonathan; Siméon, Jérôme. Xml path language (xpath). World Wide Web Consortium (W3C); August 2003;
17. Pan, Tony C.; Huang, Kun. Virtual mouse placenta: Tissue layer segmentation. Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC2005); Sep 2005;

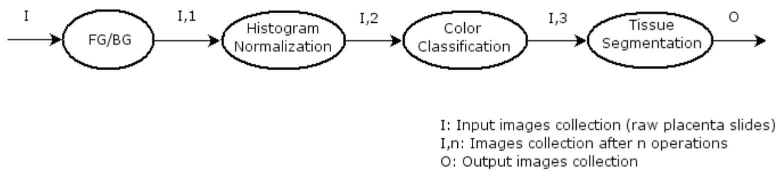

I
FG/BG
I,1
Histogram Normalization
I,2
Color Classification
I,3
Tissue Segmentation
O
I: Input images collection (raw placenta slides)
I,n: Images collection after n operations
O: Output images collection

Figure 1.
Example application workflow.

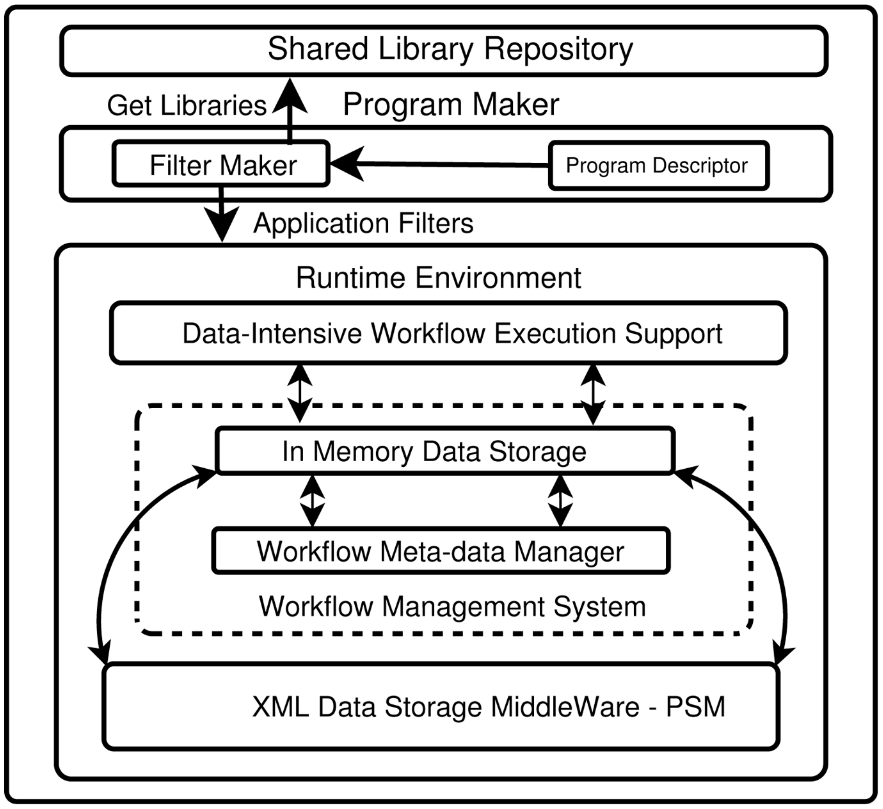
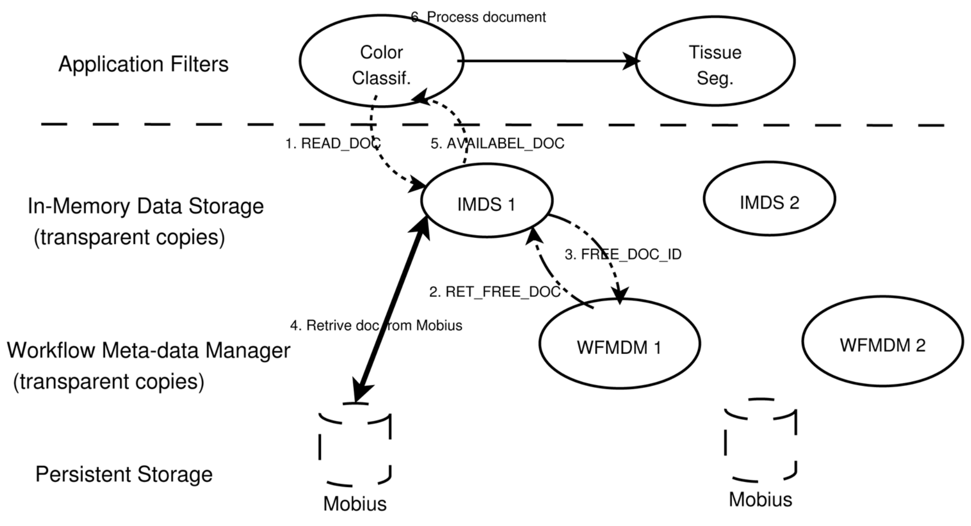
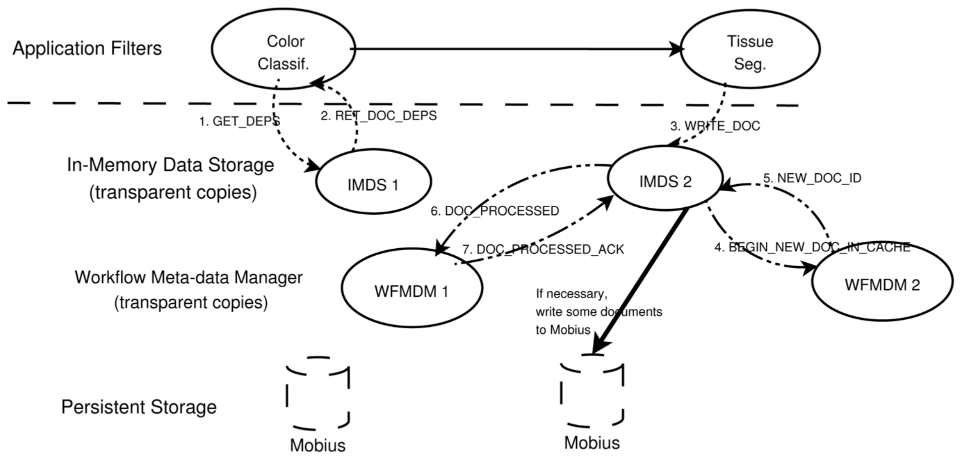

Shared Library Repository
Get Libraries
Program Maker
Filter Maker
Program Descriptor
Application Filters
Runtime Environment
Data-Intensive Workflow Execution Support
In Memory Data Storage
Workflow Meta-data Manager
Workflow Management System
XML Data Storage MiddleWare - PSM

Figure 2.
Framework Components



(a)



(b)

Figure 3.
Communication protocol inside the run-time support components.

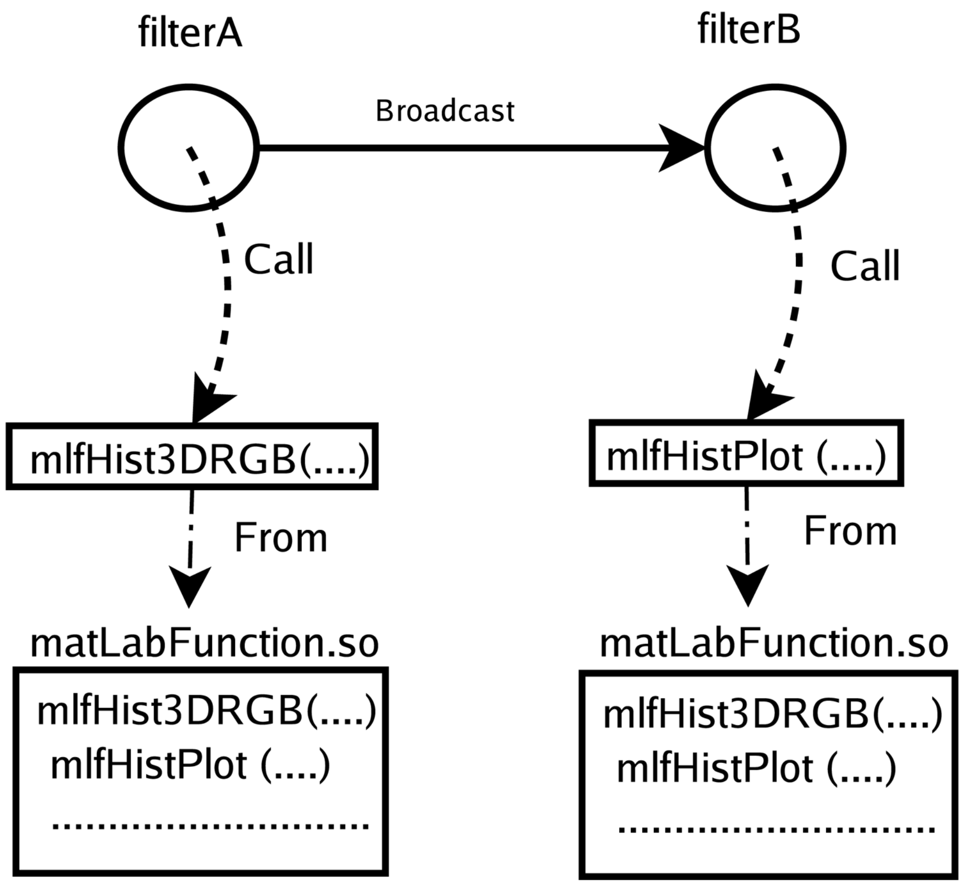
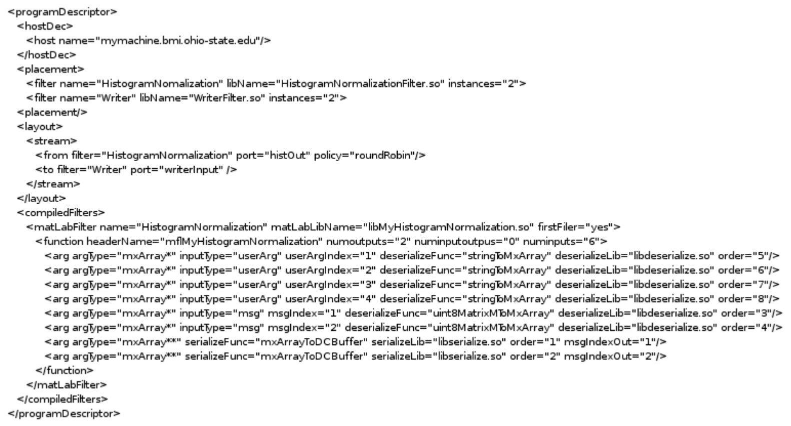


Figure 4.
A sample group of filters using Matlab libraries.



```
<programDescriptor>
  <hostDec>
    <host name="mymachine.bmi.ohio-state.edu"/>
  </hostDec>
  <placement>
    <filter name="HistogramNomalization" libName="HistogramNormalizationFilter.so" instances="2">
    <filter name="Writer" libName="WriterFilter.so" instances="2">
  <placement/>
  <layout>
    <stream>
      <from filter="HistogramNormalization" port="histOut" policy="roundRobin"/>
      <to filter="Writer" port="writerInput" />
    </stream>
  </layout>
  <compiledFilters>
    <matLabFilter name="HistogramNormalization" matLabLibName="libMyHistogramNormalization.so" firstFiler="yes">
      <function headerName="mflMyHistogramNormalization" numoutputs="2" numinputoutpus="0" numinputs="6">
        <arg argType="mxArray*" inputType="userArg" userArgIndex="1" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="5"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="2" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="6"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="3" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="7"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="4" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="8"/>
        <arg argType="mxArray*" inputType="msg" msgIndex="1" deserializeFunc="uint8MatrixMToMxArray" deserializeLib="libdeserialize.so" order="3"/>
        <arg argType="mxArray*" inputType="msg" msgIndex="2" deserializeFunc="uint8MatrixMToMxArray" deserializeLib="libdeserialize.so" order="4"/>
        <arg argType="mxArray**" serializeFunc="mxArrayToDCBuffer" serializeLib="libserialize.so" order="1" msgIndexOut="1"/>
        <arg argType="mxArray**" serializeFunc="mxArrayToDCBuffer" serializeLib="libserialize.so" order="2" msgIndexOut="2"/>
      </function>
    </matLabFilter>
  </compiledFilters>
</programDescriptor>
```

Figure 5.

A sample configuration file for a workflow consisting of Matlab filters.

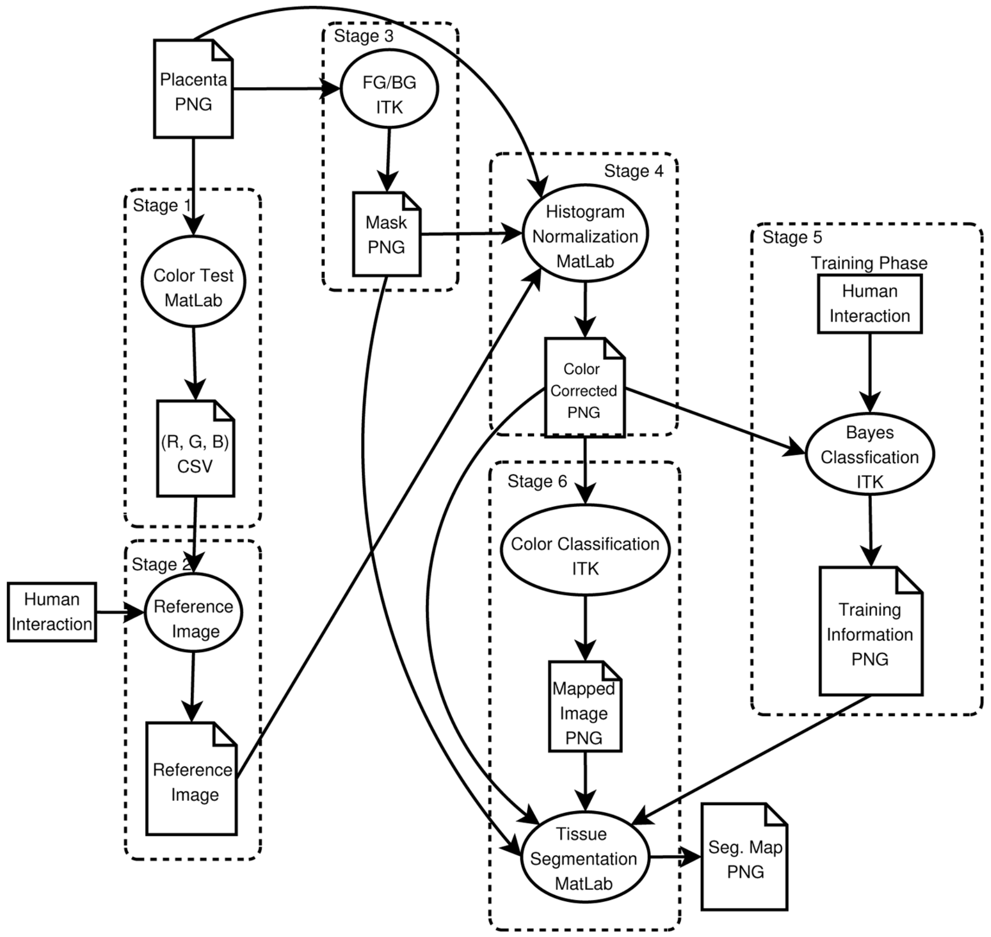


Figure 6.
Mouse Placenta Application

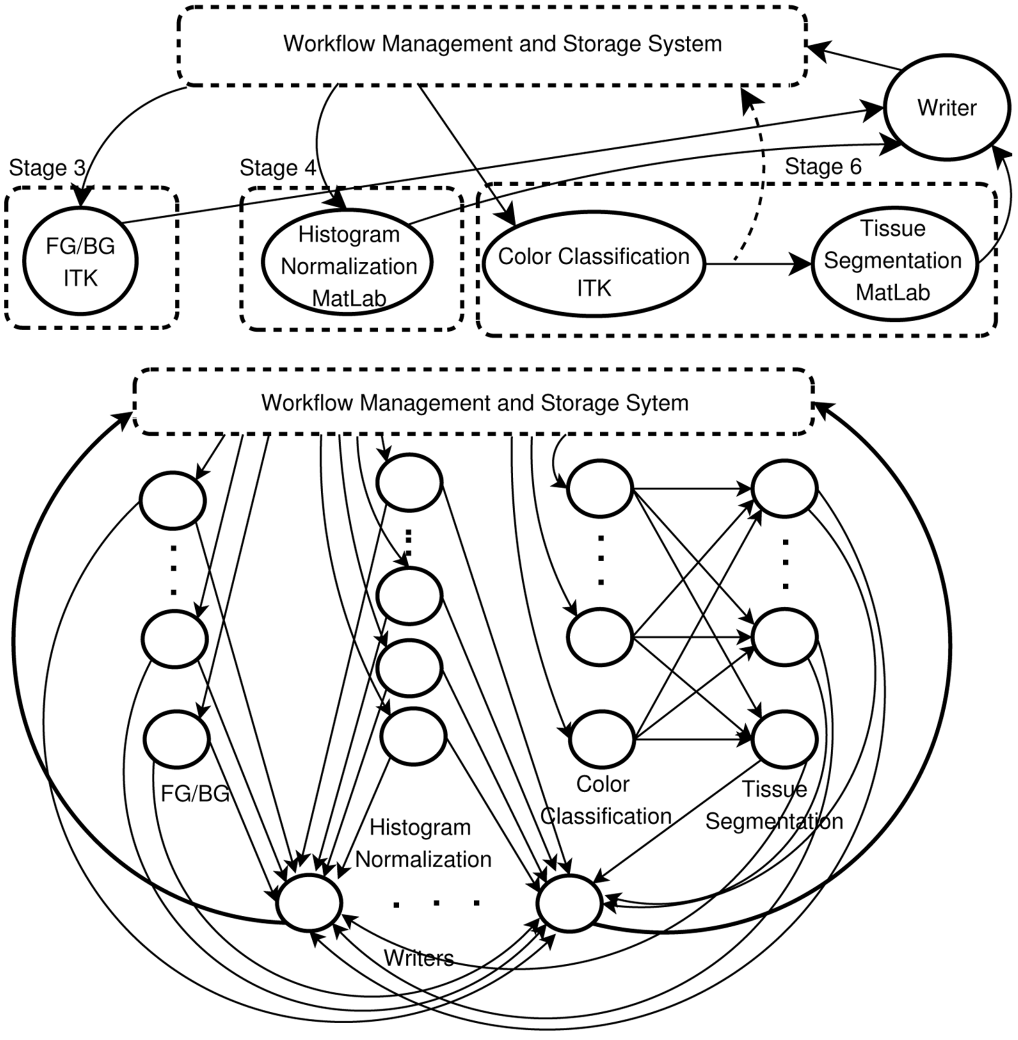
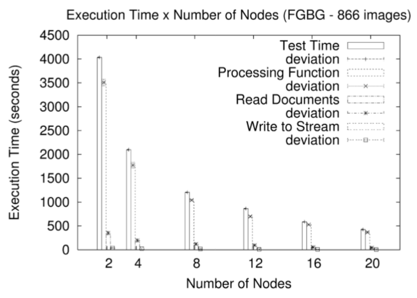
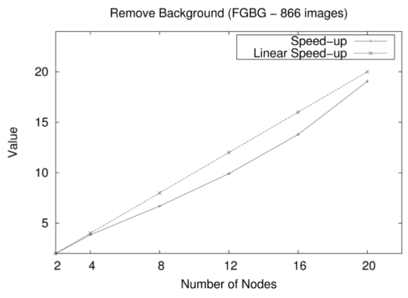


Figure 7.
Mouse Placenta Application WorkFlow



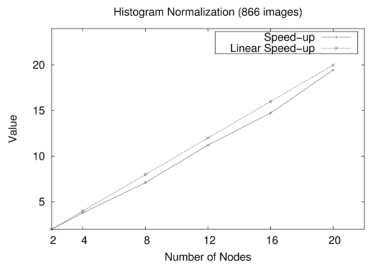
(a)



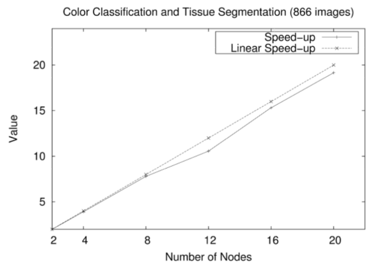
(b)

Figure 8.

(a) The dissection of the execution time of the FB/BG stage. (b) the speed-up values.



(a)



(b)

Figure 9.
Speed-up: Histogram and Color Classification stages

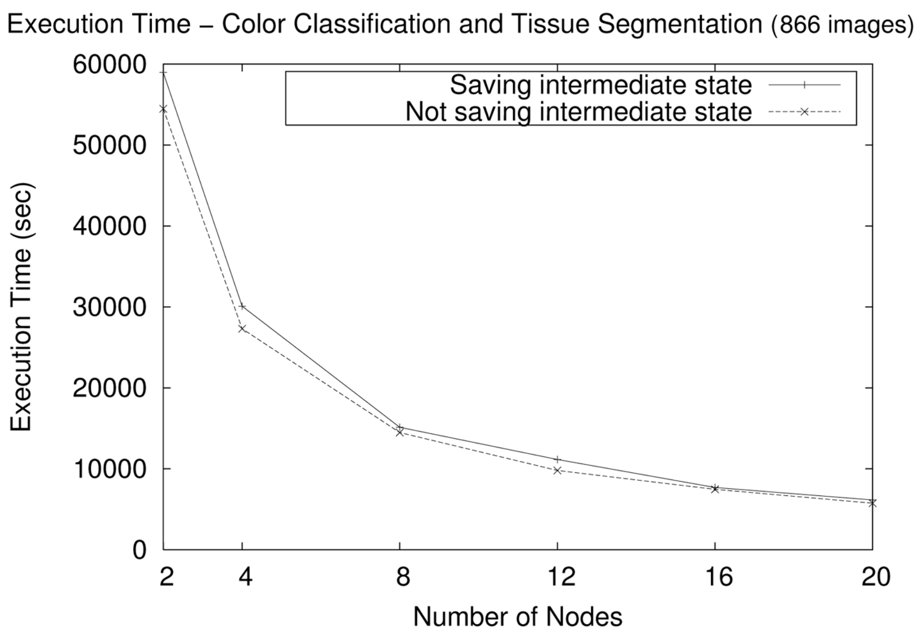


Figure 10.
Color Classification and Tissue Segmentation Test: doing and not doing checkpoint

Algorithm 1

Application Filter

```
while there is data to be processed do
    read(data)
    inputData = de-serialize(data)
    outPutData = process(inputData)
    if there is any outPutData to be written then
        outPut = serialize(outPutData)
        write(outPut)
    end if
end while
```