Run-time Support for Efficient Execution of Scientific Workflows on Distributed Environments

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

*Department of Computer Science
Universidade Federal de Minas Gerais, Brazil
{george,ttavares,renato,meira,dorgival}@dcc.ufmg.br

†Department of Biomedical Informatics
The Ohio State University, USA
{kurc,tpan,jsaltz}@bmi.osu.edu

Abstract

Scientific Workflow Systems have been introduced in response to the demand of researchers from several domains of science who need to process and analyse increasingly larger experimental datasets. The idea is based on the observation that these operations can be composed as long pipelines of fairly standard computations that need to be executed on very large data collection. Researchers should then be able to share pieces of code among peers, and these several components can be integrated on different workflows for data processing. In this work we present a run-time support system that was customized for facilitating this type of computation on distributed computing environments. Our system is optimized for data-intensive workflows, meaning that we are very concerned with data management issues. Experiments with our system have shown that we can achieve linear speedups for fairly sophisticated applications, created from multiple components.

1 Introduction

Since the computer science started progressing and reaching other fields, the process of data analysis has become an activity extremely significant in many scientific researches. The challenges to design and implement support for efficient data analysis are many, mainly due to characteristics of scientific applications that generate and reference datasets. Datasets in many applications are very large in size, as they are generated by large scale experiments or long running simulations. One example is the Large Hadron Collider (LCD) project at CERN. Starting this year of 2006, this project will generate raw data on petabyte scale from four large underground particle detector each year (5). Projects like the Grid Datafarm (17) are being implemented to be able to process these datasets.

To help the researchers in their experiments and analysis, the Scientific Workflow Systems (12; 11; 2; 14) have been introduced. In most scientific applications, scientific workflows are data-centric and can be modeled as *dataflow process networks* (13). 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.

Some of the challenges to design workflow systems that support processing of large datasets are: store, query and manage large distributed databases, manage the input and output data and the scheduling and monitoring of these workflows execution in the distributed environment, and optimize the reuse of components in different workflows. Distributed environments, like a cluster of PCs, provide viable platforms to efficiently execute workflows on large datasets.

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, run the network of components on very large data collections on clusters of storage and computation nodes, and monitor the execution by examining, for example, intermediate data sets created during execution. Scientific workflow systems should also support component reuse. In other words, a components may be part of a specific workflow, but also can be reused in another application workflow. If we take a look at a specific example of a biomedical image analysis application, we can see its workflow as the one 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.

We proposed and developed the Anthill system (8), a system based on the filter-stream programming model that was originally proposed for Active Disks (1). In Anthill, filters are each stage of the computation and streams are an abstraction for communication between filters. Using this framework, applications are a set of filters over the network connected using streams, creating task parallelism as in a pipeline. During the execution, multiple copies of each filter are instantiated, allowing every stage of the pipeline to be replicated, creating data parallelism.

In an earlier paper (8), we demonstrated the efficacy and efficiency of Anthill for data mining tasks. In this paper, we present the results of a joint effort to extend the functionality of Anthill. These extensions include 1) a *program maker* component, which builds workflow executables from 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

Chimera (9) project is a virtual data system, which represents data derivation procedures and derived data for explicit data provenance. This information is used for re-executing the application and regenerating the derived data. Our approach focuses on storing the partial data results, we do not store a large amount information about data derivation but we are able to efficiently store the output data between each pipeline stage. The Pegasus (7) can create a virtual data system that saves the information about data derivation procedures and derived data using Chimera, but it also maps Chimera's abstract workflow into a concrete workflow DAG that the DAGMan (6) meta-scheduler executes using grids.

Kepler (14) addressees some issues about scientific workflow management, it provides support for web service based workflows. The authors show the composition of workflows based on the of notion actor oriented modeling, first presented in PTOLEMY II (16). Pegasus and Kepler systems have interesting solutions to managing workflows (e.g. schedules, web based solutions, etc), but they do not have good solutions to integrating the workflow execution with the data to be processed, our system have been constructed using the idea that we should provide a efficiently way to access data stored in distributed databases, providing data integration and scalable execution of workflows. NetSolve (4) provides access to computational resources, software and hardware, distributed across the network. To share the the software resource available in the network NetSolve create a infra-structure to call shared libraries that implement the available functionalities, other features like fault-tolerance and load balance are achieved by NetSolve.

3 Extended Anthill Framework

The architecture of the system, as shown in the 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 components and provides a toolkit for generating workflows based on shared components from the repository. The run-time environment is designed to support scientific workflows of 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* works as a data manager for the entire workflow execution. It has information for every 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 each filter. The *in-memory data storage* works as an intermediary between the user application and the Persistent Storage System. Based on the meta-data provided by the Workflow Meta-data Manager, it basically read the necessary data from the Persistent Storage System and store the outputs of each component. The *persistent storage system* uses the Mobius framework to expose and abstract data resources as XML and allows for the ad hoc instantiation of data stores and the federated management of existing, distributed databases.

We now proceed to detailing the implementation of each of these components. They have been designed to achieve scalable and efficient execution on distributed heterogeneous environments. The system provides mechanisms for efficiently saving partial results without introducing synchronization between the application implemented on our system and the run-time environment.

3.1 Program Maker

This component is a tool for allowing users to incorporate existing program components and libraries in the

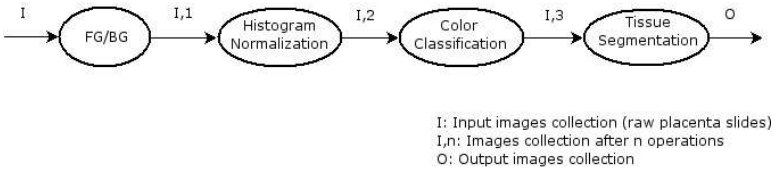


Figure 1. Example application workflow.

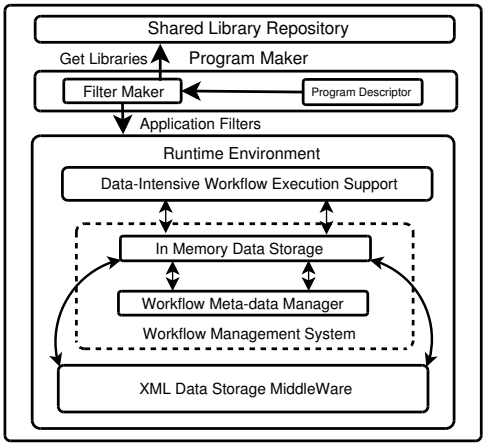


Figure 2. Framework Components

workflow system. To accomplish this task it creates additional code to each stage of the workflow pipeline to support execution of both: executables and 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

The easy incorporation of existing programs and the sharing of functionalities among users are two basic objectives of workflow management frameworks. To achieve these goals we developed a repository of shared libraries and executables (compiled code) that are used to share the available user programs. To ease the interaction with the repository, we implemented it using Mobius (10). The user can interact with the repository via three basic operations: upload, search and, download of programs and libraries.

The first operation, upload, requires the creation of a meta-data that describes the compiled code being uploaded. This meta-data, which is implemented as an XML file, contains all the information necessary to identify the type of data the program is able to work with, the structures used by each of its arguments, and the serialization/deserialization functions that need to be applied to the output and the input datasets as they flows out of and into the program. It also include some additional information about the system requirements for the particular program or library.

The second operation, search, is performed using Virtual Mako (Vmako), one of the tools included in Mobius, which creates a single virtual interface to visualize and search, the distributed databases. Using this interface the user can access the meta-data related to each stored element, perform XML queries for libraries, and so on. The third and last operation, download, is performed using a tool include in the framework, it receives a reference to the compiled code, download it from the repository and store it in the current directory.

3.1.2 Program Descriptor

This is the configuration file (in XML) of the entire pipeline, it is divided in four sections: *hostDec*, *placement*, *layout* and *compiledFilters*.

- hostDec: is used to describe all machines available in the environment. It is used to determine the available machines for each of the application components.
- placement: 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.
- layout: defines the connections between the components, their associated policies and direction of communication.
- compiledFilters: is used to provide information that the framework needs to acually execute the component. Information here is used to find out which library the code is coming 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/deserialize the data.

3.1.3 Filter Maker

This component receives the configuration we just described and basically executes it. It generates the additional Anthill code, a filter skeleton, that need to be running so that the component can execute properly, connected to the application workflow. A high level overview of what such a skeleton filter look like is shown in Algorithm 1. As we can see, it is basically a loop that reads data from the input

stream, de-serializes and pass it to the component code. It then proceeds to serializing again any output that might have been created and sending it out the next application on the workflow.

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
```

It generates the source code of the connection filter for each compileFilter declared in the workflow as well as the *Makefile* required for compiling and linking the entire application workflow. To know the location of the libraries to be linked the user should also define an environment variable pointing to the directory where it is stored.

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, that is responsible for instantiating the workflow program, the Workflow Management System that is responsible for managing the entire workflow 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 thus data parallelism as well as the pipelined task parallelism.

We have extended the Anthill run-time to provide transparent communication between the application and the Workflow Management System (WMS), described in Section 3.2.2. 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 (WMS)

This component is divided into two subcomponents: the Workflow Meta-Data Manager (WMDM) and the In-Memory Data Storage (IMDS). The Workflow Meta-Data Manager works as data manager of the entire workflow execution. It maintains information about all the data involved in the application execution, either read of written. When the workflow execution is initiated, the WMDM receives a Xpath query (3) that defines the input dataset. It then proceeds to relay the query to all instances of the Persistent Storage Manager (PSM) and builds a list of all matching documents, with the associated meta-deta. Each document of the list goes through three different status as the execution progresses:

Not processed: all documents that compose the input dataset in the beginning of the execution, what means that they are available to be processed.

Being processed: documents sent to filters are in this status as well as the the documents sent across filters because they have been created and sent to other filter process.

Processed: the documents is processed when it is processed by one filter and its result have been stored in the IMDS, describe bellow.

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

The In-Memory Data Storage works as an intermediary between the application filters and the persistent storage manager. It is also 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 local IMDS. When there is no local IMDS for a given filter, than another one is assigned to it.

At run-time, as the filters request data, these requests are passed down to the local IMDS (or to the assigned one). It acts pretty much as a disk caching system, relaying requests to unavailable data to de WMDM. The several instances of the IMDS are distributed across the available machines and work independently meaning that multiple instances can be reading different portions of the data simultaneously. It is like 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 communication as documents in Mobius. The messages exchanged by the filters are sent to IMDS, which will behave again like a write-back caching scheme, releasing the application code from having to wait for the I/O operation to complete. Like in the case of reads, multiple write operations can be executed in parallel.

3.2.3 Persistent Storage Manager(PSM)

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 (using XPath) efficiently. 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.

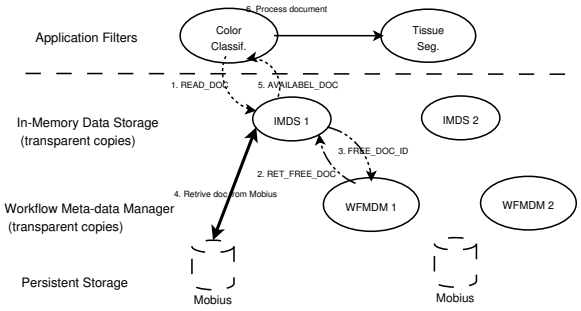
3.2.4 Communication Protocol

The workflow applications created using our system are composed by a set of filters like to the one shown in algorithm 1. These filters receive data on their input streams, apply some kind of processing to the data and produce an output to the next filters down the pipeline.

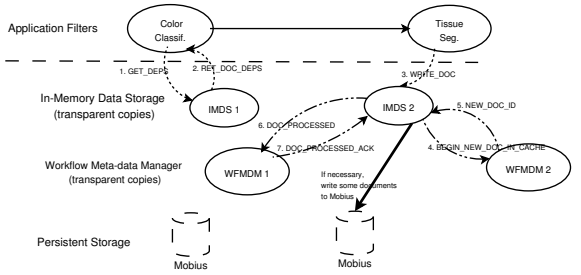
The applications created using our Program Maker interface comunicates with the rest of the run-time support transparently. Each application component is just concerned with receiving its own input data, processing it, and generating its output. In Figure 3 we illustrate the internal communication across the several components of the run-time infrastructure, we use the fifth stage in the pipeline of an image processing application (described later, see Figure 5). We see in the figure that there are two filters involved on that stage: color classification and tissue segmentation, both being fairly standard image analysis algorithms.

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

Figure 3(b) illustrates the write operations. It is only slightly more complicated operation: as color classification code outputs it date, the stub has to first create the dependencies (documents used to create othr documents) on the local IMDS. After that, the data is sent from one stub to the next, using the streams infrastructure within Anthill. Once the stub on the receiving end gets the data, it creates a local copy of it before sending it up to the application code. As this copy of the data is stored locally, the IMDS will notify the local WFMDM about the local copy available locally and it also notifies the sender's WFMDM that the data was successfully received, which will cause the change in the state (to processed) on the input document that generated that particular output document. The IMDS will eventually move the data from its memory to some disk using Mobius. It happens in the background so that the application is not penalized.



(a)



(b)

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

4 Application Example

In this section we 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 microscopic imaging to study the phenotype change on mouse placenta induced by some genetic manipulations. It handles the segmentation of the images that compose the 3D mouse placenta into regions corresponding to the three tissue layers: the labyrinth, spongiotrophoblast, and glycogen, as described in (15).

We have divided this application into in six stages, as seen in figure 4, and mapped four of the most expensive stages as components of our workflow system, as discussed bellow. The basic description of each of the four stages are:

Foreground/Background Separation (FG/BG): The images are converted from RGB to CMYK and a combi-

nation of the color channels are thresholded to get the foreground tissue.

Histogram Normalization: The images need to be 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: The pixels in an image are classified using a Bayesian classifier. The classification 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.

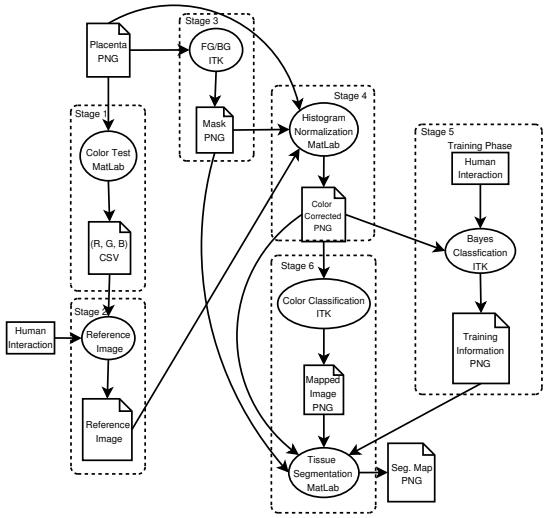


Figure 4. Mouse Placenta Application

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

4.2 Application filters

The main work to integrate an application into our system consists in constructing the section *compiledFilter* of our "Program Descriptor", that is a configuration file describing the entire pipeline (see Section 3.1.2). In the "compiledFilter" section, the user describes details about each filter that should perform user's compiled functions.

Due to space limitations, we do not elaborate any further on the format of the configuration file other than to say that it is XML document containing a detailed description of each of the application components, with all the information required for automatically generating the stub filter.

4.3 Application Workflow

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

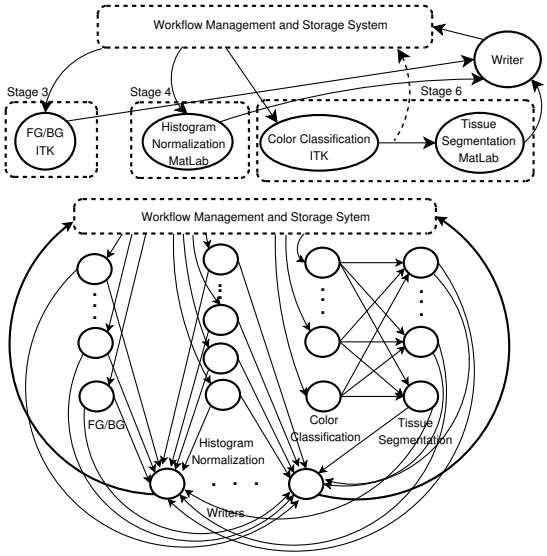


Figure 5. Mouse Placenta Application Work-Flow

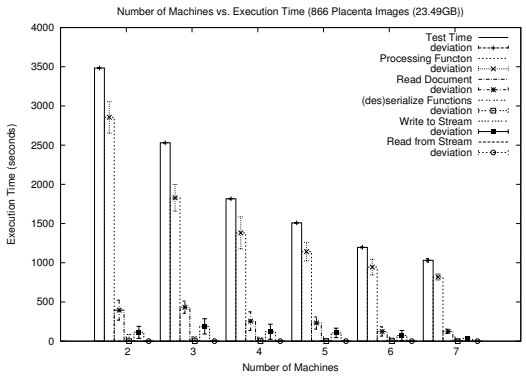
In the example application, inputs needed by stage 3 are the outputs of stage 2, so we have a clear data dependency between them. During stage 2 execution, our framework takes care of creating new data collections on the fly across all available machines and store the outputs and the related meta-data to be used as stage 3 input. Once stages 2 and 3 have completed execution, stage 4 can be executed. Again the the framework take care of storing the outputs from this stage. Stage 6 has a stream between two application filters. The figure 5 shows this stream and a dotted arrow from it to the Workflow Management System. This arrow represents an optional efficient stream storage mechanism. This feature is important for many application because it allows storage of partial results, which can be used to re-start the execution from the stored stream reducing the execution time.

5 Experimental Results

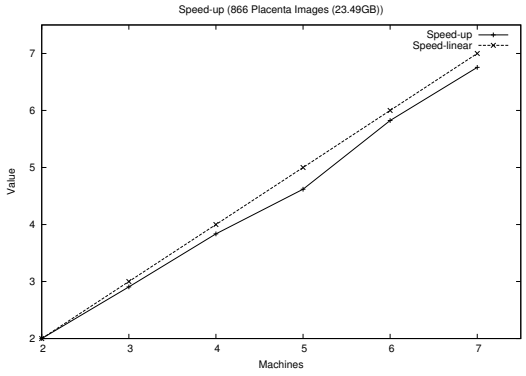
In this section we evaluate the implementation of the example image analysis application developed using the framework described in this paper. The experiments were run on a cluster with 7 PCs connected using switch Fast Ethernet. Each node has two 1.4GHz AMD Opteron(tm) Processor 240 with 7 GB main memory, running Linux 2.4.

To evaluate our system, we have used a dataset of 866 images, that have been created by dividing a mouse pla-

centa in sections and scanning it, as described in (15). The size of the whole dataset is 23.49 GB. The dataset has been stored in the PSM component; we ran one 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 have instantiated one IMD on each machine and one WMDM on one of the machines. To better use multiple processors on each machine, we have also created two transparent copies of the pipeline filters per machine.



(a)



(b)

Figure 6. (a) The dissection of the execution time of the FB/BG stage. (b) the speed-up values.

Figure 6 shows some experimental evaluations of the Foreground/Background Separation (FG/BG) stage. These numbers show the good scalability of our system, which achieves almost linear speed-up, as seen in figure 6(b). In figure 6(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. The "Read Document" time represents the time spent by our system to retrieve the data required by the application. The time spent using two machines is smaller that that spent using three machines. To explain this, we have measured the average deviation of the data size stored in each machine, see table 1. This measures load imbalance between the machines. There is more load imbalance on three nodes than on two nodes, resulting in the increase in data retrieval time.

Table 1. Avg. Input Dataset Deviation

Number of Machines	Value
2	8.75749
3	9.35287
4	7.44299
5	4.26831
6	3.97418
7	3.56119

Figure 7(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 6000 seconds and the speed-up almost linear. Figure 7(b) shows the speed-up of the last and most expensive stage, the "Color Classification" and "Tissue Segmentation". For this stage, the execution time using 2 machines is about 30000 seconds and the speed-up is almost linear.

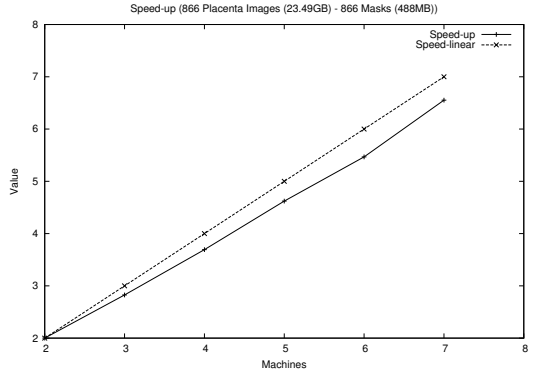
Figure 8 shows the performance of the system when partial results from a stage is saved in the system. In the figure, the execution time of the last stage of the application is shown, when the partial results from the Color Classification filter are saved or not saved in the system. As is seen from the figure, the overhead is very small and less than 5% on average.

6 Conclusion and Future Work

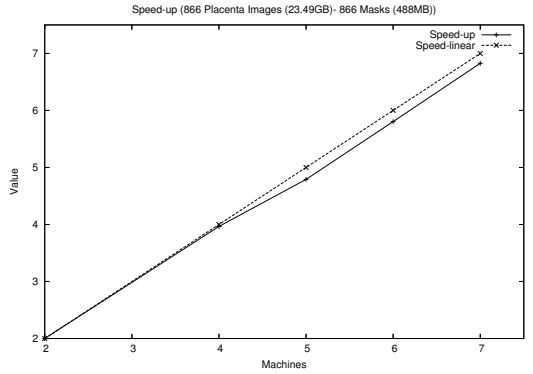
In this paper we have presented our run-time support system for efficient execution of scientific workflows on distributed environments. Our system was built on top of Anthill, and consists of a stub Anthill filter that can be automatically generated from a high level description of the application components. In order to provide data management at low overhead, we use Mobius. The stub filters can actually run arbitrary user code with a simple interface. 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 fairly sophisticated applications, with multiple components, with linear speedups. This means that our system imposes very little overhead on the applications.

Our next step in future work is towards building a robust, dependable workflow system. Fault tolerance is important in any environment with a large number of CPUs running for non-trivial periods of time. Our goal is to use our efficient data management scheme to store checkpoints



(a)



(b)

Figure 7. Speed-up: Histogram and Color Classification stages

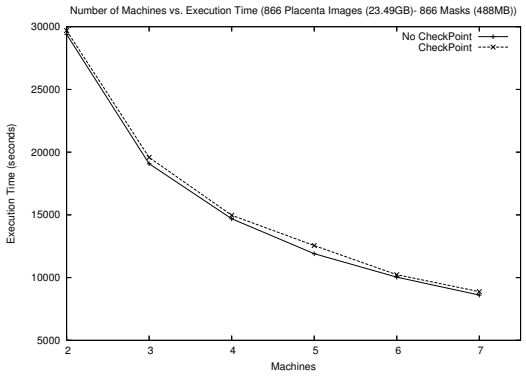


Figure 8. Color Classification and Tissue Segmentation Test: doing and not doing checkpoint

and then allow the applications to resume execution from arbitrary configurations.

References

[1] A. Acharya, M. Usysal, and J. Saltz. Active discks: Programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operations Systems (ASPLOS VIII)*, pages 81–91, Oct 1998.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows, 2004.

[3] A. Berglund, S. Boag, D. Chamberlim, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath). In *World Wide Web Consortium (W3C)*, August 2003.

[4] D. J. Casanova H. Netsolve: A network enabled server for solving computational science problems. In *International Journal of Supercomputer*, pages 212–223, 1997.

[5] CERN. Large hadron collider (lhc) - http://www.interactions.org/lhc/.

[6] Condor. The directed acyclic graph manager, 2003.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *In Journal of Grid Computing*, pages 25–39, 2003.

[8] R. Ferreira, W. M. Jr., D. Guedes, L. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. Ferreira. Anthill:a scalable run-time environment for data mining applications. In *Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2005.

[9] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: Avirtual data system for representing, querying, and automating data derivation. In *The 14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, 2002.

[10] S. Hastings, S. Langella, S. Oster, and J. Saltz. Distributed data management and integration framework: The mobius project. In *Global Grid Forum 11 (GGF11) Semantic Grid Applications Workshop*, pages 20 – 38. IEEE Computer Society, 2004.

[11] 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.

[12] 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.

[13] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, may 1995.

[14] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, and Y. Z. J. Tao. Scientific workflow management and the kepler system, to appear, 2005.

[15] T. C. Pan and K. Huang. 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.

[16] U. B. PTOLEMYII project, Deparment of EECS. http://ptolemy.eecs.berkeley.edu/ptolemyII/, 2004.

[17] 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.