An Environment for Evaluation and Testing of Service Workflow Schedulers in Clouds

Carlos R. Senna, Luiz F. Bittencourt, and Edmundo R. M. Madeira
Institute of Computing, University of Campinas - UNICAMP
Av. Albert Einstein, 1251, 13083-852, Campinas, São Paulo, Brazil
{crsenna, bit, edmundo}@ic.unicamp.br

ABSTRACT

Workflows built through service composition bring new challenges, making the scheduling task even more complex. Besides that, scheduling researchers need a wide range of workflows and their respective services to validate new achievements. In this paper we present a service oriented testbed where experiments can be conducted to develop and evaluate algorithms, heuristics, and scheduling policies for workflows. Our testbed offers an emulator service which allows the workflow characterization through the description of its services. With this, researchers can build workflows which have similar behavior to the real workflow applications, emulating them without the need of implementing all applications and services involved in a real application execution. To demonstrate the utilization of both the testbed and emulator, we conducted workflow executions emulating service workflow applications, such as Montage and LIGO.

KEYWORDS: Cloud computing, workflow, service, scheduling, performance evaluation.

1. INTRODUCTION

In the cloud computing paradigm, details about the infrastructure are abstracted from the users. Therefore, a cloud user does not need knowledge, experience, or control over the infrastructure. The cloud provides resources which scale up dynamically, which are frequently virtualized as services over the Internet. The models used by cloud computing are based on Service Oriented Computing (SOC), and allow users to establish connections among services, organizing them as workflows. However, such characteristics turn the resources management and scheduling even more complex. Scheduling researchers commonly need different sets of workflows with varying topologies, representing different application sizes and structures, to evaluate allocation policies. This evaluation is often done through simulations, since building and deploying an entire working set of applications is hard in two aspects: (i) the bigger the set of applications, the more personnel is demanded to build them within a reasonable time; (ii) resources configuration requirements (software and hardware) may be conflicting for two or more different applications, which makes it not possible to deploy all of them simultaneously. Thus, new tools to ease these evaluation processes are of interest.

In this paper we present a service oriented testbed, where experiments can be conducted to evaluate scheduling algorithms, policies, and strategies, mainly focusing on service workflows. Our testbed has an emulation service that allows the characterization of workflows through a description of their read and write requirements, and execution time of each service that is part of the workflow. To illustrate the testbed utilization, we present results from image processing workflows used in e-Science applications, such as Montage [1] and LIGO [2]. Through these emulations, we demonstrate how our testbed could be used to experiment different scheduling algorithms and to evaluate the resources performance, contributing to the improvement of workflow execution in clouds without deploying real services and its requirements.

This paper is organized as follows. In Section 2 we introduce our infrastructure. How workflows are characterized and how we built the emulator service are presented in Section3, while Section 4 presents the experimental results. Related works are discussed in Section 5, and Section 6 concludes the paper.

2. THE TESTBED INFRASTRUCTURE

Our infrastructure for experimental scheduling evaluation is based on a computational grid, a workflow management system, a scheduling service, and a resources monitor. The testbed receives as input a set of workflows and a set of scheduling algorithms, and it produces the schedules and execution times given for the workflows for each algorithm. The grid testbed is a Globus Toolkit (GT) [3] deployment, an OGSA (Open Grid Service Architecture) implementation. In the OGSA, all resources (physical or virtual), are modeled as services, bringing to the grid the concepts offered by SOC. Our base system (Figure 1) is a GT version 4 on 4 resources: Apolo (Intel Pentium IV HT, 3.0Ghz, 2GB RAM), Cronos and Dionisio (Intel Core 2 Quad, 2.4 Ghz, 4GB RAM), and Zeus (Intel Xeon, 2.66Ghz, 12 GB RAM), all with Debian Linux connected by gigabit Ethernet.

The management of the service compositions in our infrastructure is made by the GPO (Grid Process Orchestration) [4], a middleware for service workflows execution in the grid. The GPO allows the creation and management of application flows, tasks, and services. The GPO uses workflows built with the GPOL (GPO Language) [4]. The GPOL is based on concepts of service orchestration from WS-BPEL, with added specific directives for grids state maintenance, potentially transient services, notification, data-oriented services, and groups. The language includes variables, lifecycle, fabric/instance control, flow control, and fault handling. Additionally, it allows the user to start task executions, service executions, and workflow executions in sequence or in parallel. The scheduling service is responsible for distributing the workflow services to be executed in the available resources. To accomplish this, the scheduling service may implement different algorithms with different optimization objectives, and decide which one to use depending on the application or current environment characteristics. The scheduling algorithm can make its decisions based on information available from the resource monitoring service. Our testbed along with its emulation service can support the decision on which scheduler to use and, in consequence, which information to gather from the resource monitor. Information about the available resources in the cloud can be obtained through the resource monitor (RM). The RM operates in a distributed manner, maintaining one instance on each computing resource and providing on demand information for other services. Based on RM information, the scheduler indicates where to execute a service and/or how to reorganize the services distribution in the cloud to improve the global performance. New scheduling algorithms can be easily deployed, because the scheduler can be implemented as a cloud service. Our middleware provides an execution option to indicate which scheduler service, among the available ones, should be used. To support the implementation of new services, our testbed has the scheduler service DummySS, which is an interface model for a generic service. Simply fill this service model with the desired algorithm, publish it, and try it.

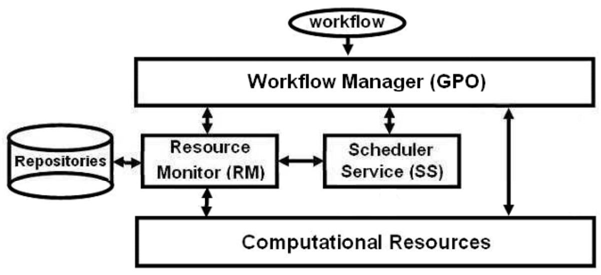


Figure 1. The Testbed Infrastructure.

Our middleware provides the monitoring of workflow executions. The GPO monitors the execution times of each section specified in the GPOL workflow, including the time spent on each operation invoked in the process, registering them in a log file exclusive for each workflow. To illustrate this procedure, we show below some fragments of a GPOL workflow which applies a median filter in an image file [4]:

```
<gpo:job name="ip-A3500x3500">
  <gpo:definitions name="job_Definitions">
(1) <gpo:variables>
      <gpo:variable name="retCode"  type="int" value="0"/>

    </gpo:variables>
(2)<gpo:gservices>
       <gpo:gsh name="Inst_Zeus-mf1"
         uri="http://10.3.77.5:8080/wsrf/services/FactIPService"  type="Factory"/>
       <gpo:gsh name="Inst_Zeus-mf2"
         uri="http://10.3.77.5:8080/wsrf/services/FactIPService" type="Factory"/>
       <gpo:gsh name="Inst_Cronos-mf2"
         uri="http://10.3.77.16:8080/wsrf/services/FactIPService"  type="Factory"/>
       <gpo:gsh name="Inst_Dionisio-mf3"
         uri="http://10.3.77.15:8080/wsrf/services/FactIPService"
         type="Factory"/>
    </gpo:gservices>
  </gpo:definitions>

<gpo:process name="p-ip-A3500x3500">
(3) <!--Splits file in 2 parts-->
<gpo:invoke name="Inst_Zeus-mf1" method="sliceMatrixToFiles">
   <gpo:argument variable="sliceIn" type="string"/>
<gpo:return variable="sliceOut" type="string"/> </gpo:invoke>
(4)
<gpo:flow name="F-SCP-send"> <!-- Send slices to resources -->
  <gpo:invoke name="Inst_Zeus-mf1" method="runCommand" tagname="sendS1ToCronos">
     <gpo:argument variable=S1ToCronos type="string"/>
     <gpo:return variable="retCode" type="int"/> </gpo:invoke>
  <gpo:invoke name="Inst_Zeus-mf2" method="runCommand" tagname="sendS1ToDionisio">
     <gpo:argument variable=S2ToDionisio type="string"/>
     <gpo:return variable="retCode" type="int"/> </gpo:invoke>
</gpo:flow>
(5)
<gpo:flow name="F_medianFilterMatrix"> <!Apply median filter in parallel -->
  <gpo:invoke name="Inst_Cronos-mf2" method="medianFilterMatrix" tagname="cronosMF">
     <gpo:argument variable="mFIn-1" type="string"/>
     <gpo:return variable="retCode" type="int"/> </gpo:invoke>
  <gpo:invoke name="Inst_Dionisio-mf3" method="medianFilterMatrix" tagname="dioMF">
     <gpo:argument variable="mFIn-2" type="string"/>
     <gpo:return variable="retCode" type="int"/> </gpo:invoke>
</gpo:flow>
(6)
<gpo:flow name="F-SCP-receive"> <!-- Receive slices from resources -->
   <gpo:invoke name="Inst_Zeus-mf1" method="runCommand" tagname="rsf_cronos">
      <gpo:argument variable=R1FromCronos type="string"/>
      <gpo:return variable="retCode" type="int"/> </gpo:invoke>
   <gpo:invoke name="Inst_Zeus-mf2" method="runCommand" tagname="rsf_dionisio">
      <gpo:argument variable=R1FromDionisio type="string"/>
      <gpo:return variable="retCode" type="int"/> </gpo:invoke>
 </gpo:flow>
```

```
(7)<!-- Merge slices files into a new filter file -->
<gpo:invoke name="Inst_Zeus-mf1" method="mergeSliceFiles">
   <gpo:argument variable="mergeIn" type="string"/>
    <gpo:return variable="mergeOut" type="int"/> </gpo:invoke>

   <!-- Return value to GPO Client -->
   <gpo:return variable="retCode" type="int"/>
 </gpo:process> </gpo:job>
```

In (1) the workflow variables are created with information of the used services. In (2) the service instances of the workflow are created. In this example, we created two instances of the *FactIPService* in *Zeus*, one in *Cronos*, and one in Dionisio. In (3) we utilized one instance from *Zeus* to split the image file in two parts, which are sent to *Cronos* and *Dionisio* in parallel in (4). The step (5) is the median filter application, which is made simultaneously by *Cronos* and *Dionisio* instances. The resulting files are sent back to *Zeus* in (6), where they are used to generate the final image file (7). At each execution of this workflow, the GPO adds to the log file the following information:

```
... GPOMaestro Vr. GPO Maestro Service 1.4.5 Dec 06 2010
Processing ip-A3500x3500.gpol at Sun Dec 06 13:09:36 BRST 2010
. ExecProcess: gpo:invoke Inst_Zeus-mf1 sliceMatrixToFiles  end - Time: 244
. flow.ProcessInvoke: Inst_Zeus-mf1 runCommand sendS1ToCronos end - Time: 629
. flow.ProcessInvoke: Inst_Zeus-mf2 runCommand sendS1ToDionisio end - Time: 659
.ExecProcess: gpo:flow F-SCP-send end - Time: 659
. flow.ProcessInvoke: Inst_Cronos-mf2 medianFilterMatrix cronosMF end - Time: 5408
. flow.ProcessInvoke: Inst_Dionisio-mf3 medianFilterMatrix dioMF end - Time: 5914
.ExecProcess: gpo:flow F_medianFilterMatrix end - Time: 5915
. flow.ProcessInvoke: Inst_Zeus-mf2 runCommand rsf_dionisio end - Time: 550
. flow.ProcessInvoke: Inst_Zeus-mf1 runCommand rsf_cronos end - Time: 501
.ExecProcess: gpo:flow F-SCP-receive end - Time: 551
.ExecProcess: gpo:invoke Inst_Zeus-mf1 mergeSliceFiles  end - Time: 250
.Process Execution Time: 7621
--- GPO Processing ip-A3500x3500.gpol Time: 10877 Sun Dec 06 13:09:47 BRST 2011
```

We can see in the log that 244ms were taken to split the file in two parts (sliceMatrixToFiles), 629ms to send one file slice to *Cronos* (sendS1ToCronos), and 659ms to send another slice to *Dionisio* (sendS1ToDionisio). The filter application itself took 5408ms in *Cronos* (cronosMF), while *Dionisio* spent 5914ms (dioMF). To bring the resulting slice back to *Zeus* 501ms were taken from *Cronos* (rsf_cronos) and 550ms from *Dionisio* (rsf_dionisio). The file merge operation took 250ms in *Zeus* (mergeSliceFiles). Finally, the total processing time was 7621ms (Process Execution Time), which is not the sum of all times because many operations occurred in parallel. Such performance information is logged and also maintained in the repositories associated with the RM on each resource in the cloud. This way, the scheduler can use historical information to make new schedules of the same (or similar) services.

3. WORKFLOWS CHARACTERIZATION AND THE EMULATION SERVICE

The set of information provided by our middleware is fundamental to support the scheduler in its task of choosing the best resource for each service that composes the workflow. However, there exist limitations to implement all the necessary services for all workflows and deploy them on all available resources. Such limitations include

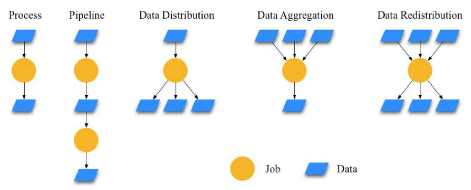


Figure 2. Basic Workflow Structures [5].

personnel and software requirements, which can be conflicting, making it not possible to experiment the necessary service-resource combinations to evaluate scheduling policies and resources performance.

To contour this situation, we created an emulation service which mimics many aspects of the workflows execution. To do that, it was necessary to characterize the basic blocks of service operations and data transfers. To make this characterization, we considered the work in [5] as a base. In that work, the authors show how to characterize the scientific workflow activities through basic structures that combine tasks and data. They define 5 structures types: Process, Pipeline, Data Distribution, Data Aggregation, and Data Redistribution, as shown in Figure 2. The *Process* structure receives a data set and produces an output. *Pipeline* is composed of a sequence of *Processes*. *Data Distribution* receives data from a single source and produces data which is consumed by multiple parts. *Data Aggregation* receives multiple inputs and produces a data set which will be consumed by a single part, while *Data Redistribution* receives multiple data sets from different sources and produces data for multiple parts.

In our work we consider the *Process*, *Data Distribution*, and *Data Aggregation* structures. We constructed our service emulator implementing operations for each one of these three structures, and we added one operation that allows file transferring among resources. The operations which implement the structures basically read all input files and generate all output files with random content. To emulate the data processing, while the stipulated execution time for a service emulation is not reached, a small benchmark code is executed. This small code portion, besides consuming CPU, can be used as a performance metric for each resource by counting how many times this code is executed within a given time space. Simple modifications in the median filter workflow presented in Section 2.2 are necessary for the emulation process:

```
  <gpo:definitions name="job_Definitions">
    <gpo:variables>
      <gpo:variable name="retCode"  type="int" value="0"/>
(1) <gpo:variable name="mProc-1"  type="string"
        value="A-3500x3500_s001.txt A-3500x3500_smf001.txt 5408"/>
     <gpo:variable name="mProc-2"  type="string"
        value="A-3500x3500_s002.txt A-3500x3500_smf002.txt 5914"/>
     </gpo:variables>
(2)<gpo:gservices>
        <gpo:gsh name="Inst_Zeus-mf1"
          uri="http://10.3.77.5:8080/wsrf/services/FactRCService" type="Factory"/>
          ...
    </gpo:gservices></gpo:definitions>

<gpo:process name="p-ip-A3500x3500">
(3) <gpo:invoke name="Inst_Zeus-mf1" method="dataDistribution"
         tagname="sliceMatrixToFiles">  <!--Splits file in 2 parts-->
     <gpo:argument variable="sliceIn" type="string"/>
     <gpo:return variable="sliceOut" type="string"/> </gpo:invoke>
(4)  ...
(5)<gpo:flow name="F_medianFilterMatrix"> <!Apply median filter in parallel -->
   <gpo:invoke name="Inst_Cronos-mf2" method="mProcess" tagname="cronosMF">
      <gpo:argument variable=" mProc-1" type="string"/>
      <gpo:return variable="retCode" type="int"/> </gpo:invoke>
   <gpo:invoke name="Inst_Dionisio-mf3" method="mProcess" tagname="dioMF">
      <gpo:argument variable=" mProc-2" type="string"/>
      <gpo:return variable="retCode" type="int"/> </gpo:invoke>
   </gpo:flow>
 (6)  ...
 (7)<gpo:invoke name="Inst_Zeus-mf1" method="dataAggregation"
      tagname="mergeSliceFiles">
     <gpo:argument variable="mergeIn" type="string"/>
     <gpo:return variable="mergeOut" type="int"/> </gpo:invoke>

     <gpo:return variable="retCode" type="int"/>
</gpo:process>
```

In (1) we included the *mProc-1* and *mProc-2* variables, which are arguments for the *mProcess* operations that emulate the median filter in the created slices. Both contain the input and output file names and the minimum running time for the emulation of *mProcess*. In (2), 4 instances of the emulation service *FactRCService* are created. In (3) we used the *DataDistribution* operation to emulate the original *sliceMatrixFiles*. In (5) we emulate the median filter through the *mProcess* service from the emulator. It reads the input file, generates a new output file with the same size, and, while the minimum execution time stipulated is not reached, it executes the benchmark code. Finally, in (7) we emulate the *mergeSliceFiles* using the *dataAggregation* operation from the emulator. This operation reads both input files and generates a new output file with the same size of the original image. Steps (4) and (6) did not need any change.

3.1. Emulated Workflows

Using our emulator service we built emulation workflows which present a quite similar behavior to the real application workflows, namely the median filter, Montage, and LIGO. The **median filter** is an image processing application that can be executed in parallel by splitting the image into pieces and merging the results back into one single image. The median filter substitutes the value of a pixel by the surrounding values on its neighborhood. Figure 3 shows the median filter workflow structure. **Montage** is an application that makes mosaics from the sky for astronomy research. Its workflow size depends on the square degree size of the sky to be generated. For example, for an 1 square degree of the sky, a workflow with 232 jobs is executed. For a 10 square degrees of the sky a $20,652$ jobs workflow is executed, dealing with an amount of data near to 100 GB. The full sky is around $400,000$ square degrees [6]. In our experiments we used Montage workflows with 25 and 50 nodes. **LIGO** (Laser Interferometer Gravitational-Wave Observatory Inspiral Analysis) is used in gravitational wave analysis to detect distortion in the space-time.

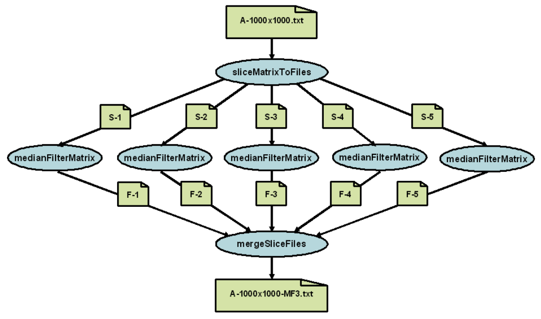


Figure 3. Median Filter Workflow Example.

4. EXPERIMENTAL RESULTS

In order to demonstrate the usefulness of our testbed along with the workflow emulator, we present experimental results in three steps: (i) the emulation precision results set aims at analyzing the precision of the emulator, comparing the desired execution times with the emulator results; (ii) the workflow emulation results show the emulation of entire workflows with different resource allocation strategies; and (iii) the real median filter application execution times for different resource allocation strategies.

4.1. Emulation precision

An important factor in a service emulator is to precisely mimic the execution duration of a given service. In this section we provide an overview of execution times given by the emulator in comparison with the execution times of single real application services. We present such numbers for the median filter, Montage, and LIGO services. The first step was to determine the real execution time of each service to be emulated. For the median filter we obtained these execution times from GPO runs of the filter, using the monitors and logs provided by the infrastructure. For Montage and LIGO services we obtained the execution times from the DAX files, described in [5].

Table 1 shows real execution times for services that compose the median filter workflow (Figure 3) for different matrix sizes: $3,500 \times 3,500$; $5,000 \times 5,000$; and $10,000 \times 10,000$. Columns labeled with R show the real execution times, while columns labeled with E show the emulated times. The **Median Filter P.E.** row shows the time spent to execute all 5 median filter services in parallel perceived by the client, which includes the remote calls/returns to the services. These results are for the execution of 5 parallel median filter jobs on the same resource.

Table 1. Real Execution Times and Emulation Times for the Median Filter Services.

Median Filter					
	3,500			5,000	10,000
Zeus (Z)	R	E	$\frac{E}{R}$	$\frac{E}{R}$	$\frac{E}{R}$
medianFilterMatrix-1	3,715	3,757	1.011	1.006	1.001
medianFilterMatrix-2	3,846	3,936	1.023	1.014	1.003
medianFilterMatrix-3	3,779	3,881	1.027	1.013	1.003
medianFilterMatrix-4	3,819	3,912	1.024	1.014	1.003
medianFilterMatrix-5	3,784	3,888	1.027	1.015	1.004
Median Filter P.E.	4,195	3,940	0.939	0.920	0.908
Cronos (C)	**3,500**			**5,000**	**10,000**
medianFilterMatrix-1	3,500	3,536	1.010	1.005	1.001
medianFilterMatrix-2	3,545	3,662	1.033	1.016	1.003
medianFilterMatrix-3	3,572	3,677	1.029	1.016	1.003
medianFilterMatrix-4	3,471	3,581	1.032	1.016	1.003
medianFilterMatrix-5	3,566	3,674	1.030	1.023	1.003
Median Filter P.E.	3,827	3,693	0.965	0.966	0.937
Dionisio (D)	**3,500**			**5,000**	**10,000**
medianFilterMatrix-1	3,843	3,878	1.009	1.005	1.001
medianFilterMatrix-2	3,929	4,063	1.034	1.017	1.004
medianFilterMatrix-3	3,924	4,052	1.033	1.017	1.004
medianFilterMatrix-4	3,908	4,048	1.036	1.015	1.004
medianFilterMatrix-5	3,891	4,025	1.034	1.016	1.004
Median Filter P.E.	4,156	4,082	0.982	0.963	0.951

The execution times for the median filter services show that the emulator can effectively mimic the execution times of the real workflow application jobs on heterogeneous resources. Execution times of the median filter services took equivalent time to the real execution, with difference of at most 3.6%. During the emulation, the emulator takes into account the creation, write, read, and transfer of files needed by each service. With this, if the execution times are much higher than the expected, the user can investigate the occurrence of potential disk I/O concurrency and/or memory requirements of the application.

Table 2 shows emulation results for the Montage workflow with 25 nodes in a single resource, as well as an excerpt from the emulation of Montage with 50 nodes, showing services *mDiffFit-01* to *mDiffFit-28*. For the Montage 25, the real execution times in the table were taken from the DAX file, as described in [5]. We can observe that the services emulation behavior follows the real execution times. However, note that for *mDiffFit-6* to *mDiffFit-9* the emulator took more time to run than the real execution time stipulated. This is a consequence of disk I/O concurrency introduced by 9 tasks running in parallel, as well as the limited number of cores on each resource. Additionally, for *Cronos* and *Dionisio*, which have 4 cores each, emulations of *mDiffFit-6* to *mDiffFit-9* took more time than the emulation in *Zeus*, which has 8 cores. Thus, besides I/O, the CPU concurrency also has a role in delaying the execution when all services are in parallel on the same resource. In the Montage 50 case, the emulation shows an even higher execution time for these jobs, since the I/O concurrency and extrapolation of number of cores is higher than in Montage with 25 nodes. With the emulation, the user can test different execution configurations in a wide range of resources without the need of deploying all the necessary services and files to run the real application on them. After the emulation, the user can configure only the necessary resources to be able to execute the application.

Table 2. Execution Times for Emulated Services from Montage 25 and Montage 50.

Montage 25				Montage 50			
	(Z)	(C)	(D)		(Z)	(C)	(D)
	$\frac{E}{R}$	$\frac{E}{R}$	$\frac{E}{R}$		$\frac{E}{R}$	$\frac{E}{R}$	$\frac{E}{R}$
mProjectPP-1	1.01	1.01	1.01	mDiffFit-01	1.03	1.03	1.02
mProjectPP-2	1.03	1.01	1.01	mDiffFit-02	1.04	1.04	1.03
mProjectPP-3	1.02	1.01	1.01	mDiffFit-03	1.03	1.03	1
mProjectPP-4	1.01	1.01	1.01	mDiffFit-04	1.07	1.07	1.02
mProjectPP-5	1.01	1.01	1.01	mDiffFit-05	1.06	1.06	1.03
mProjectPP P.E.	1.03	1.01	1.01	mDiffFit-06	1.1	1.1	1.04
mDiffFit-1	1.01	1.03	1	mDiffFit-07	1.12	1.12	1.04
mDiffFit-2	1.01	1.01	1.02	mDiffFit-08	1.1	1.1	1.01
mDiffFit-3	1.01	1.02	1	mDiffFit-09	1.43	1.43	1.72
mDiffFit-4	1.01	1.03	1.02	mDiffFit-10	1.51	1.51	1.63
mDiffFit-5	1.01	1.03	1.01	mDiffFit-11	1.47	1.47	1.72
mDiffFit-6	1.07	1.24	1.17	mDiffFit-12	1.49	1.49	1.66
mDiffFit-7	1.06	1.24	1.17	mDiffFit-13	1.59	1.59	1.82
mDiffFit-8	1.08	1.24	1.17	mDiffFit-14	1.68	1.68	1.6
mDiffFit-9	1.06	1.25	1.18	mDiffFit-15	1.72	1.72	1.73
mDiffFit P.E.	1.06	1.23	1.15	mDiffFit-16	1.75	1.75	1.76
mConcatFit	1.03	1	1.02	mDiffFit-17	1.72	1.72	1.7
mBgModel	1.01	0.99	1.01	mDiffFit-18	1.79	1.79	1.72
mBackground-1	1.06	1.05	1.02	mDiffFit-19	1.84	1.84	1.69
mBackground-2	1.04	1	1.02	mDiffFit-20	1.61	1.61	1.67
mBackground-3	1.09	1.07	1.07	mDiffFit-21	1.9	1.9	1.67
mBackground-4	1	1.05	1.02	mDiffFit-22	1.93	1.93	1.71
mBackground-5	1	1.06	1.04	mDiffFit-23	1.84	1.84	1.65
mBackground P.E.	1.16	1.11	1.14	mDiffFit-24	1.87	1.87	1.66
mImgTbl	1.01	0.97	1.01	mDiffFit-25	1.86	1.86	1.74
mAdd	0.68	0.96	1	mDiffFit-26	1.97	1.97	1.82
mShrink	1	0.96	1	mDiffFit-27	1.96	1.96	1.78
mJPEG	1.04	0.99	1.02	mDiffFit-28	1.93	1.93	1.74

Execution times for LIGO services emulation are shown in Table 3, where *TmpltBlank, Inspiral, Thinca*, and *TrigBank* are services used by LIGO workflow. We can note a different behavior when concurrency appears. Note, for instance, that *Inspirals* and *TrigBanks* show different emulation time on different machines. This is a result of the different emulation times of such tasks. While in Montage the whole set of *mDiffFit* tasks has similar execution times, in LIGO this does not happen. With this, tasks make I/O at different times, and each task can interfere on each other on unpredictable ways, introducing some stochastic behavior in the execution times. In any case, this situation could be foreseen by the user during the emulation, and avoided at production time.

Hitherto we observed that the emulator can effectively emulate the execution of services, obeying the real execution times when possible while providing information to the user to foresee potential problems during parallelization in the workflows. Next we show how the emulator can be used to evaluate workflow scheduling policies.

Table 3. Results for LIGO Workflow.

LIGO 30					
	Zeus			Cronos	Dionisio
	R	E	E / R	E / R	E / R
TmpltBank (jobs 0-6)	5469	5832	1.07	1.04	1.12
Inspiral (jobs 7-13)	202422	202480	1	1.14	1.2
Thinca (job 14)	1683	1702	1.01	1.01	1.01
TrigBank (jobs 15-21)	1731	2237	1.29	1.55	1.47
Inspiral (jobs 22-28)	187914	187947	1	1.16	1.01
Thinca (job 29)	1455	1473	1.01	1.01	1.01

4.2. Workflow Emulation

We distributed services among the available resources in order to evaluate different allocation policies. In this section we assume that each resource is capable of executing one service at a time. Therefore, tasks are queued one after another in the resource for execution. This is a common assumption in the scheduling research field. When executing jobs in a heterogeneous system it is important to know the resources performance to decide where to execute each part of the workflow. In Table 4 we show the execution time of one single median filter job of different image sizes ($3,500 \times 3,500$; $5,000 \times 5,000$; $7,500 \times 7,500$; and $10,000 \times 10,000$) for each resource used in the testbed. We utilized the *Normalized Total* column as the relative processing capacity of each resource, which was used to stipulate the emulation times of the same job on different resources. With that, a job that would take, for instance, $100s$ in *Zeus*, should be emulated with an execution time of $217s$ in *Apolo*. This performance information was used by the scheduling algorithms and in the emulation duration specification to represent heterogeneous resources.

To illustrate how our testbed and emulator could be used to evaluate allocation policies, we scheduled the median filter, Montage, and LIGO workflows using three different scheduling algorithms, as follows.

Random: Randomly takes one service s_i from the workflow such that all s_i predecessors have already been scheduled or s_i has no predecessors. Randomly take a resource r_k from the repository and schedule s_i in r_k. Repeat until all services are scheduled.

Round Robin: Randomly takes one service s_i from the workflow such that all s_i predecessors have already been scheduled or s_i has no predecessors. Schedule s_i in the resource r_b such that r_b has the best processing capacity and r_b's queue has the minimum number of services among all resources.

PCH (Path Clustering Heuristic): Creates *clusters* of services that are in the same path in the workflow. Tasks in the same cluster are scheduled in the same resource considering a priority order and resources performance. For a detailed description of PCH, please refer to [7].

Table 4. Execution Time of a Single Median Filter Job.

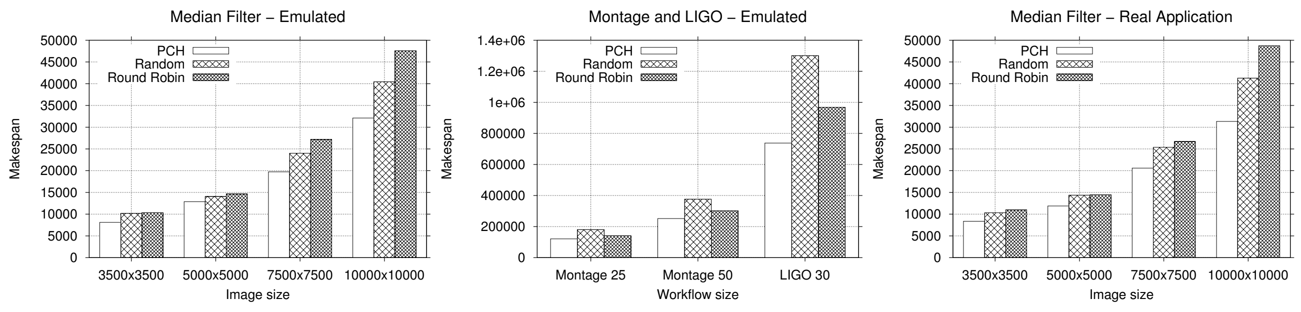
	Image size				
	$3,500$	$5,000$	$7,500$	$10,000$	Normalized Total
Apolo	4833.2	9343	20139.4	34400.8	**2.17**
Cronos	2201.6	4446.2	10133.6	17482.2	**1.08**
Dionisio	2291.8	4421.6	9750.6	17406.8	**1.07**
Zeus	2064.8	4113	9366.8	16187.8	**1**

For each scheduling algorithm we present the *makespan* of the workflows emulated, i.e., the time taken to emulate the whole workflow execution. Figure 4(a) shows the average *makespan* of the median filter with 5 services in parallel with four different image sizes. This emulation suggests that PCH can create schedules that will result in the smallest execution time among the three algorithms. On the other hand, the round robin approach does not seem suitable for such workflow, resulting in the worse average makespan in all cases. Emulations for Montage with 25 and 50 nodes, and for LIGO with 30 nodes are shown in Figure 4(b). A different pattern can be observed, with the round robin approach turning better than the random approach in all cases. This emulation suggests that the round robin algorithm is more suitable for Montage and LIGO workflows than the random approach, while PCH is still better than both. Finally, to effectively validate the emulation process, in Figure 4(c) we show the execution times of the real median filter workflow for the three algorithms evaluated by emulation in Section 4.2. We can observe the same behavior as in the emulated case (Figure 4(a)). Therefore, the emulator successfully accomplished its role by giving to the user results which are in consonance with the real executions.

The presented emulation results demonstrate that the emulation can help in the development of scheduling algorithms, as well as it can support the decision of which algorithm to choose to schedule different workflows.

5. RELATED WORKS

Testbeds are being explored in a variety of ways. In [5] the authors characterize workflows focusing on data movement, how it is shared and aggregated among tasks from complex structures in the workflows. In our work, we utilized three structures proposed in [5]. Our testbed is directed to the execution of service compositions where orchestration is very important. [8] defines a grid testbed requirements, with strategies for resource allocation, and makes an empirical analysis of their performance using a grid simulation environment, but there is no mention to services or workflows. In addition, [9] shows some simulations environments which complement the emulation approach proposed in this paper. The functionalities in our



(a) Median filter emulation (b) Montage and Ligo emulation (c) Median filter real application

Figure 4. Emulations and Real Application Execution Times.

emulator service are close to the main characteristics found in real applications (consumption of CPU, network, etc).

6. CONCLUSION

How to distribute services to execute in a cloud determines the execution time of the whole workflow. Therefore, the development of scheduling algorithms is important to improve the execution in such systems. To evaluate scheduling algorithms through experiments, it is necessary to test a wide range of workflow applications. A scheduler can potentially allocate a task on any resource, which implies that for evaluating the algorithm it is necessary to deploy all the necessary services from all workflows on all resources, which may be not viable due to software and hardware restrictions. In this paper we present a testbed and an emulation tool, which mimics the execution of a service workflow without the need of deploying the application in fact. This allows the evaluation of scheduling algorithms in a more real manner than through simulations, but without all the constrains of deploying dozens of services in hundreds of resources. The presented results show that the emulator is capable of mimicking the real application executions. Also, the results show that the emulated execution times for three different scheduling algorithms are equivalent to the real execution times given for the median filter application workflow.

As future works, the emulator can be improved with the introduction of other mechanisms to control the emulation. For example, new parameters that control the level of CPU consumption or disk I/O to be followed by an emulation.

ACKNOWLEDGEMENTS

We would like to thank FAPESP (09/15008-1), FINEP, and CNPq for the financial support.

REFERENCES

[1] "Montage: An astronomical image engine." 2011. [Online]. Available: http://montage.ipac.caltech.edu

[2] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensive workflows onto storage-constrained distributed resources," in *CCGRID '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 401–409.

[3] G. Alliance, "Globus toolkit," 2011. [Online]. Available: http://http://www.globus.org/toolkit/

[4] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira, "Execution of service workflows in grid environments," *International Journal of Communication Networks and Distributed Systems (IJCNDS)*, vol. 5, no. 1/2, pp. 88–108, 2010.

[5] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi., "Characterization of scientific workflows," in *WORKS 2008: 3rd Workshop on Workflows in Support of Large-Scale Science*, 2008.

[6] E. Deelman, "Clouds: An opportunity for scientific applications? (keynote in the 2008 Cracow Grid Workshops)," 2008.

[7] L. F. Bittencourt and E. R. M. Madeira, "A performance-oriented adaptive scheduler for dependent tasks on grids," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 9, pp. 1029–1049, 2008.

[8] J. Blythe, Y. Gil, and E. Deelman, "Coordinating workflows in shared grid environments," in *ICAPS '04*, Whistler, British Columbia, Canada, 2004.

[9] "Cloudsim," 2011. [Online]. Available: http://www.buyya.com/gridbus/cloudsim/