Implement the Grid Workflow Scheduling for Data Intensive Applications with CSF4

Zhaohui DING, Xiaohui WEI, Yifan ZHU, Yaoguang YUAN
College of Computer Science and Technology
Jilin University
ChangChun, P.R.China
zhaohui.ding@email.jlu.edu.cn, weixh@jlu.edu.cn

Wilfred W. LI
National Biomedical Computation Resource
University of California, San Diego
San Diego, California, U.S.
wilfred@sdsc.edu

Osamu TATEBE
Center for Computational Sciences
University of Tsukuba
Tennodai, Tsukuba, Japan
tatebe@cs.tsukuba.ac.jp

Abstract—Grid computing technology is able to integrate and share large-scale distributed computation and data resource to facilitate the scientific researches. Recently, the grid workflow support and large-scale distributed data management are becoming two main requirements of scientists and researchers in many fields, such as bioinformatics, high-energy physics etc. In this paper, we proposed to support grid workflow for data intensive applications using CSF4 scheduling plug-ins. The grid workflow scheduling and data aware scheduling policies are implemented in two scheduling plug-ins, grid workflow plug-in and grid data aware plug-in, respectively. The two scheduling plug-ins can work together smoothly. The data aware plug-in will automatically dispatch the workflow tasks to the grid sites which are close to data replicas. At last, the experiment results are given to show the improvement of system performance and optimization of scheduling.

Keywords-grid workflow; data intensive; scheduling; plugin; CSF4

I. INTRODUCTION

A. Background

With the rapid development of the grid computing technology, the focus of research efforts has begun to transform from the implementation of infrastructure to the resolution of real scientific problems. The workflow technologies, there have been many customized solutions targeting specific users by business vendors. While as distinguished from the common business workflow, the grid workflow engine should be geared to the distributed, heterogeneous and autonomic grid environment, and also, concentrate on the requirements of grid aware applications.

The grid workflow definition was given in [1]: the automation of the processes, which involves the orchestration of a set of grid services, agents and actors that must be combined together to solve a problem or to define a new service. Since a great deal of distributed, heterogeneous and autonomic grid resource was involved in, the scheduling issue is becoming the key point of the efficient use of system. Emerging classes of data-intensive applications that both access and generate large data sets are becoming another important issue, especially appear in complicated workflows. High-performance data-intensive computing and networking technology has become a vital part of large-scale scientific research projects in areas such as high energy physics, astronomy, space exploration, human genome projects, and computational simulations in biomedical research. For example, users often need to connect the input and output of several applications into a workflow, which required that all the applications are able to access their data on grids. The so-called Data Grid techniques were introduced to provide such kind of essential infrastructure, such as Grid data farm (Gfarm [2] for short).

Gfarm architecture is designed for global peta-scale data-intensive computing. It provides a global parallel file system with online peta-scale storage, scalable I/O bandwidth, and scalable parallel processing. However, when a huge amount of data I/O is involved, a network system’s performance will be degraded by network congestions without proper data management and job scheduling. Although user can allocate the file-affinity hosts for optimum execution of applications based on the available metadata by the Gfarm command line tools, the manual method is not scalable in a production environment with a large number of users running jobs concurrently. It is imperative to have an automated job scheduling and data management mechanisms.

Both grid workflow and grid data intensive applications will have their special scheduling demands, which pose particular challenges to the meta-scheduling. In this paper, we designed the scheduling policies for grid workflow and grid data intensive applications, and implemented them as two scheduling plug-ins, called grid workflow plug-in and grid data aware plug-in, in CSF4 meta-scheduler. CSF4 [3] is
a WSRF compliant meta-scheduler, developed by our team and released as an execution component of Globus Toolkit 4 [4]. In the previous work [5], we introduced and implemented a scheduling plug-in framework in CSF4. In this work, we redesigned and improved the framework to enable multiple scheduling plug-ins to work together smoothly, such as grid workflow and grid data aware. With the plug-in mechanism, the system has good extensibility and is fully configurable for new scheduling demands. Moreover, we have already integrated CSF4 with Opal [6], a web service wrapper for legacy applications, and developed CSF4 portal [7], a web-browser based interface, which provide more convenient use.

The rest of this paper is organized as follows: Section 1.2 gives a brief overview of the related works; section 2 introduces the working principles of CSF4 scheduling plug-in mechanism, as well as the design of grid workflow plug-in and grid data aware plug-in; section 3 describes the experiment results. At last, we conclude the paper and discuss the future work.

B. Related Works

In [19], a grid workflow taxonomy and a detailed investigation on existing workflow management systems are given. As the precursor, WebFlow [8] introduces a visual programming paradigm for the development of high performance distributed computing applications. Gridflow [9] provides a two-tier service framework with both global grid workflow management and local grid sub workflow scheduling and provides a flexible agent-based mechanism for dynamic scheduling of grid jobs within a global grid workflow service. GridAnt [10] uses the ant workflow processing engine, focuses on client side to provide a client-controllable workflow mechanism. DAGMan scheduler [11] supports the expression of dependencies of jobs, however, it can only work in the Condor system.

The work in this paper focuses on the grid workflow for data intensive application. In our design, the workflow management system and scheduling system are complete separate, so the workflow engine can be easily used in other scheduling system. The workflow scheduling system is implemented as a plug-in so that it is able to cooperate with other scheduling plug-in, like grid data aware plug-in.

Pegasus [12] (in GriPhyN project [13]) is also a workflow management system focuses on the optimization of data intensive application. It is able to generate and reduce the workflow based on data products which have already been computed earlier. It prunes the workflow based on the assumption that it is always more costly to compute the data product than to fetch it from an existing location. In our system, Gfarm provides a global file system and the grid data aware plug-in implements the automatic data replica location selection in multiple clusters environment.

Most of the above works, including CSF4, are using the protocols of Globus, such as GRAM for job submission and GSI for grid security.

II. DESIGN AND IMPLEMENTATION

In this section, we will describe the design and the implementation of workflow plug-in and data aware plug-in first, and how they work together to perform the scheduling for data intensive workflow jobs via CSF4 plug-in mechanism.

A. The CSF4 Plug-in Architecture

The CSF4 plug-in mechanism consists of the scheduler framework and a number of scheduler plug-in modules. In figure 1, the scheduler framework works as a motherboard with slots to hold scheduler plug-in modules. The framework maintains the elementary information for a scheduler, like pending jobs, available resources etc. Plug-in modules are able to access those data maintained by scheduler framework via CSF plug-in API. The particular scheduling policies are implemented inside plug-in modules.

![CSF4 Plug-in Architecture](image)

The plug-in modules are loaded dynamically by the framework at run time. The user can indicate which modules to be loaded via configuration. Using this mechanism, the users do not need to write a scheduler from scratch, but to provide just a plug-in module with the desired policy. Moreover, CSF4 plug-in mechanism supports the cooperation of multiple scheduling policies. Therefore, it enables the workflow scheduling to co-work with data aware scheduling policy, which is presented in this paper.

B. Workflow Plug-in

There are two main parts in the grid workflow plug-in, the workflow engine and the workflow job scheduling algorithm.

1) Workflow Engine

We adopt XPDL (XML Process Definition Language) [14] that is widely used in the distributed workflow modeling to describe grid workflow tasks. XPDL uses XML schema to describe workflows, and is supported by various workflow engines. For instance, XPDL can be mapped to an execution language like BPEL (business process execution language), and executed in an execution engine such as ActiveBPEL[15].

The following tags are used to describe a workflow task by our workflow engine. There is one <WorkflowProcesses> tag for each workflow task. <WorkflowProcesses> is the root tag for the task. There could be one or more <WorkflowProcess> tags inside <WorkflowProcesses> tag to represent the sub workflows of the task. <Activities> tag
will indicate all the sub jobs that will be really executed in a grid for a sub-workflow. \textit{<Transitions>} presents the dependencies among the sub jobs.

For example, "\textit{<Transition Id="16" Name="" From="a" To="b"/>}" represents that sub task "a" is the \textbf{predecessor} of sub task "b", or sub jobs "b" is the \textbf{successor} of sub job "a". Then, any grid workflow can be described as a directed acyclic graph (DAG). A typical workflow is shown in Figure 2, where the nodes of the graph represent a sub job to be performed and the directed edges represent dependencies between tasks. Due to the \textit{<WorkflowProcess>} tag, a node could also be a sub workflow, like the rectangle labeled with "Sub Workflow" in Figure 2. The corresponding workflow descriptions in XPDL are partially given out in Figure 3.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{An Example of Workflows}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{The XPDL description of the workflow}
\end{figure}

In addition, we extended XPDL to support RSL\cite{16}, which is the standard language to specify a grid job’s resource requirements, to describe the sub tasks. Then, these sub jobs’ resource requirements, in particular for their data requirements, will be recognized by data aware plug-in. In fact, the workflow plug-in does not care about such RSL descriptions.

2) \textbf{Scheduling Algorithm}

As described earlier, a workflow’s XPDL description will indicate how many sub jobs are there in the workflow, and what kind of dependencies they have. The sub jobs are the entities that are really executed in the grid resources to perform the data processing. The workflow scheduling algorithm is to decide the sub job dispatch order according to the job dependencies. We call the jobs whose dependencies are satisfied as ready jobs. And the data aware scheduling is to map the ready jobs to data affinity hosts according to its resource requirements. Just like workflow plug-in does not care a sub job’s RSL description, the data aware plug-in does not recognize the workflow task’s XPDL description either. Hence, the workflow plug-in need to generate real jobs for each ready job and insert them to the meta-scheduler’s waiting job queue. Then, the scheduling policies provided by other plug-in modules can be applied.

In this paper, we focus on data intensive workflow tasks, whose sub jobs normally take the output files from its predecessors as input and generate new data files for its successors. The goal of the job scheduling is to minimize the space cost and the whole workflow’s execution time.

\textbf{Space Cost}

See figure 1, we assume that each job in the workflow will generate an output file, for example, file Fn is the output of Jobn. File Fn will be removed once all Jobn’s successors have completed. The space cost of the workflow is the total disk space occupied by above data files. Depth-first and breadth-first are two popular graph traveling algorithms. If there are enough available hosts in the grid, then all the ready jobs will be executed in parallel. There is no difference between depth-first and breadth-first algorithm. However, if the available hosts are limited due to resource competing, the two algorithms will have different impacts on the workflow execution.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{Workflow Examples}
\end{figure}

To make it simple, we consider the special case that there is only one available host. For a simple workflow as Fig 4-A, the space costs of depth-first and breadth-first algorithms are exactly same. However, they are different for the relatively complex workflows showed by Fig 4-B and Fig 4-C. The Table I shows the space costs of Fig4-B.

At time T4, depth-first algorithm just need keep two files instead of breadth-first algorithm’s there in disk. Although the sizes of data files are not equal, in average the less files will take less disk space. We can also show that depth-first algorithm will have lower space cost than breadth-first algorithm for Figure4-C. The reason is that the data files will be consumed quicker by the successor jobs with depth-first algorithm. Hence, depth-first algorithm is better than breadth-first algorithm in terms of space cost.
TABLE 1. SPACE COST OF FIGURE 4-B WITH DEPTH-FIRST AND BREADTH-FIRST

<table>
<thead>
<tr>
<th>Time</th>
<th>Data Files in disk</th>
<th>Time</th>
<th>Data Files in disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:Job1 finish</td>
<td>F1</td>
<td>T1:Job1 finish</td>
<td>F1</td>
</tr>
<tr>
<td>T2:Job2 finish</td>
<td>F1, F2</td>
<td>T2:Job2 finish</td>
<td>F1, F2</td>
</tr>
<tr>
<td>T3:Job3 finish</td>
<td>F1, F2, F3</td>
<td>T3:Job3 finish</td>
<td>F1, F2, F3</td>
</tr>
<tr>
<td>T4:Job5 finish</td>
<td>F1, F3</td>
<td>T4:Job4 finish</td>
<td>F2, F3, F4</td>
</tr>
<tr>
<td>T5:Job4 finish</td>
<td>F4, F5</td>
<td>T5:Job5 finish</td>
<td>F4, F5</td>
</tr>
<tr>
<td>T6:Job6 finish</td>
<td>F6</td>
<td>T6:Job6 finish</td>
<td>F6</td>
</tr>
</tbody>
</table>

Makespan Time

Now, we consider the workflow’s execution time. We use Figure 4-B as example. Job2, Job3 and Job4 are all Job1’s successors. If there are two available hosts, which two jobs should go first? From the below chart, the whole workflow will have a shorter makespan time if Job 2 and Job3 go first.

![Figure 5. Makespan time](Image 5)

In solution two, the host2 is wasted as either Job5 or Job6 is not ready to go at time t3. In solution one, Job2 is executed at time T2 instead of Job4 so that at time T3 Job4 and Job5 can start in concurrent. And same reason for Figure 4-C, Job 2 and Job3 should go first instead of Job 3 and Job 4. Compared with Job4, there are more jobs are dependent on Job2’s execution. So starting Job2 earlier means more jobs can be executed concurrently later on. In another word, the scheduling algorithm should maximize the parallel processing of the workflow when the grid can provide more available resources.

Algorithm

Before presenting the algorithm, we give out the following definitions.

**Definition 1:** A job’s dependent job set consists of all the sub jobs in the workflow that cannot start unless this job has finished. For Figure 1-B as instance, Job2’s dependent job set is \{Job5, Job6\}. According to the job successor/predecessor definitions in section 2.2.1, Job2’s successor job set is \{Job5\} and Job5’s predecessor job set is \{Job2, Job3\}.

**Definition 2:** Ready job means a sub job in a workflow that all its dependencies have been satisfied. The job is ready to be executed at any time from now. In another words, all the jobs in a ready job’s predecessor job set should be finished.

**Definition 3:** Function $N_{DJS}(jobp)$ is to calculate the number of jobs in jobp’s dependent job set.

Once a workflow is submitted, the workflow plug-in will interpret the workflow’s XPDL description and calculate out the predecessor job set, successor job set and dependent job set for each sub job in the workflow. After that, the following algorithm will be performed to decide the sub jobs’ dispatch order.

At the very beginning, the algorithm will do the below operations:

1. Insert all the jobs whose predecessor job sets are NULL to New Ready Job List;
2. Sort New Ready Job List in descending order based on $N_{DJS}()$ value;
3. Copy New Ready Job List to Ready Job List;
4. Generate real jobs with RSL description for New Ready Job List;
5. Insert new generated real jobs to the meta-scheduler’s waiting job queue;
6. Sort all the real jobs in the same order as Ready Job List;

During the workflow running, the algorithm will do as below,

```java
While (not done) {
    Once a sub job finish {
        Set New Ready Job List to this job’s successor job set;
        Remove non-ready jobs from New Ready Job List;
        If (New Ready Job List) {
            Sort New Ready Job List in descending order based on $N_{DJS}()$ value;
            Insert New Ready Job List to the head of Ready Job List;
            Generate real jobs for New Ready Job List;
            Insert new generated real jobs to the meta-scheduler’s waiting job queue;
            Sort all the real jobs in the same order as Ready Job List;
        }
    }
    Once a sub job start to run {
        Remove the sub job from Ready Job List;
    }
}
```

The algorithm has two main features. First, the sub jobs are dispatched in a combination of least makespan time and depth-first order. The sub jobs with higher $N_{DJS}()$ value will have more chance to run earlier. And the new generated real jobs are inserted to head of the Ready Job List. Second, the real jobs are generated for the ready jobs only. Hence, the real jobs are inserted to the scheduler job queue gradually instead of all at once. It will reduce the scheduler’s memory cost and improve its scheduling performance at same time.
C. Grid Data Aware Plug-in Design

The data aware plug-in is to find the data file locations via Gfarm APIs, and map the jobs to the hosts close to the data. Using Gfarm API, data aware plug-in can retrieve the IP address of the hosts that have data replica from Gfarm Meta Server. Then the meta-scheduler will forward the grid jobs to the clusters that close to the data.

Hence, we need to determine which cluster is closest to the data. The simplest way is to maintain a static data-cluster mapping list, but it is low efficient and inflexible. We adopt a dynamic way. We compute the end-to-end delays between the host with replica and all clusters’ master hosts by using ping tool[20]. Generally, the network delays inside a cluster are much lower than that between two clusters. So we think the above estimation is reasonable. Then we choose the cluster with the lowest delay as the job’s destination.

The CSF4 framework will merge the scheduling decisions of all the plug-in modules and dispatch the jobs to grid resources. Besides that, the CSF4 framework will do all the dirty works that a job scheduler has to do, such as collecting resource availability, monitoring job status change etc. Once a real job starts to run or finish, a notification will be sent to workflow plug-in and other plug-ins who are interested in such event.

III. Experiments

Our experiment environment was made up of three clusters. Each cluster has four machines and managed by LSF scheduler. All the machines installed Red Hat 9 Linux and Gfarm. We use scp to test the average network transfer speed. The speed between the all the machines is 10.4MB/s.

The tested workflow was described in Figure 2. We added 3 data dependencies between the sub jobs into the workflow (MWF0-MWF1, MWF3-MWF5 and SWF0-SWF1), and we used three sizes of data for the testing. For example, the output data of sub job MWF0 will be 60MB/500MB/2.5GB respectively in the test, which is the input file of sub job MWF1.

We executed the workflow in two different ways. In the first way, we only use grid workflow plug-in and distribute the files on each cluster randomly. In the second way, we use both grid workflow plug-in and grid data aware plug-in.

Figure 7 compares the average execution time of the whole workflow in the two ways with different data size. Without data aware plug-in, the workflow sub jobs have to involve heavy network I/Os to process the input/output data by itself. In the contrast, the grid data-aware plug-in is able to dispatch sub jobs close to data files so that a considerable amount of network I/Os are saved. It will reduce the network congestion and speed up the job execution significantly as well.

We also recorded the average time cost of all the stage of meta-scheduling, which include the workflow XPDl parsing, sub job processing and real job generation, data aware scheduling, job/resource matching and job dispatch. See figure8, the workflow XPDl parse will do only once for each workflow. The scheduling time for each sub job consists of sub job generation and data aware scheduling, which is about 0.24s. The job dispatch is mainly caused by CSF job/resource mapping via GRAM protocol.
exceeds the computing capability of one single cluster, how to distribute the relevant data files among multiple clusters is a challenge as well. We are planning to touch the above issues in the future work, and test our system with real bioinformatics applications on PRAGMA grid testbed [17], such as NAMD [18], which required both workflow support and the management of large mount of data.

ACKNOWLEDGMENT

The authors would like to acknowledge support from China NSF with Grant No.60703024, and Jilin Department of Science and Technology under Grant No.20070122 and 20060532. Wilfred W. Li would like to acknowledge support from the NIH P41 RR 08605 to NBCR; TATRC W81XWH-07-2-0014 to PWA and WWL; the NSF Grant No.INT-0314015 and OCI-0627026 for the PRAGMA project.

REFERENCES


