Deriving and Managing Data Products in an Environmental Observation and Forecasting System

Laura Bright David Maier

Computer Science Department
Portland State University
Portland, Oregon
{bright,maier}@cs.pdx.edu

Abstract

Large-scale scientific workflows can perform many computationally intensive tasks and generate large volumes of derived data products. These systems pose many challenges to both creating and managing data products, including efficiently executing tasks and tracking data product lineage and metadata. In this paper we describe our experiences implementing an experimental data-product management system to address these challenges for the CORIE Environmental Observation and Forecasting System. We present a novel architecture to store both data products and the tasks that create them. Our system in addition supports tasks to automatically perform system maintenance, and enables data-intensive tasks to execute on multiple nodes of a Grid. We present several challenges to executing existing scientific workflows on a Grid, and propose several techniques to improve task scheduling in this environment. Preliminary performance results show the potential benefits of these techniques.

1 Introduction

Large-scale scientific workflows are common in many domains, including experimental physics, earth sciences, life sciences, and environmental forecasting. Such systems are characterized by data- intensive tasks that generate a large number of derived *data products.* These data products include datasets and images, and may serve as inputs to subsequent tasks.

There are many challenges to successfully implementing and executing large-scale scientific workflows. One set of challenges relates to managing the *creation* of data products. Simulation and analysis tasks that generate these products are both time-intensive and data-intensive, and many different tasks may compete for limited resources. There may be opportunities to speed up execution by running independent tasks on different nodes of a Grid, but determining when and where to execute each task is non-trivial. A second set of challenges relates to managing the data products themselves. A single data product may contain several hundred megabytes, and a single run can generate thousands of individual products. Efficient storage and management of these products is needed to reduce both search and data transfer overhead. In addition, tracking metadata and lineage information for each data product is crucial for scientists who reference these products after they are created.

In this paper we describe our experiences implementing an experimental data-product management system for an environmental observation and forecasting system. Our motivation is the CORIE System [4], which integrates real-time sensor data with numerical simulations. CORIE includes multiple data-intensive scientific workflows that generate large volumes of data. Daily forecasts alone generate up to 5GB of data and over 30,000 derived data products, including images, animations, aggregated results, and datasets.

The existing CORIE workflow implementations consist of a collection of programs in C, Fortran, and Perl. Adding or modifying tasks and data products requires manually editing the code, and there is no easy way to track which programs and inputs generated which data products. Further, this collection of programs does not allow us to execute independent tasks on different nodes of a Grid, which could significantly speed up execution. Thus, our goal was to

---

Proceedings of the 2005 CIDR Conference

build a data-product management system to improve the flexibility and scalability of the existing workflows.

We have implemented our system using Thetus^{TM}[13], a commercial product designed for non-text data management common in many scientific domains. Thetus provides facilities for tracking metadata and lineage information about data products. It also provides facilities to automatically launch workflow tasks when appropriate conditions are met, and enables executing workflows on one or more nodes of a Grid.

When we initially started this project we had several goals. Our first goal was to improve the scalability of workflow execution, in terms of both ease of adding and modifying tasks and data products, and speed of task execution. A second goal was to improve our ability to track lineage information about data products, and easily identify which executables and inputs generated each data product. Such information is useful for regenerating data products from new inputs, and for identifying which products are affected by changes in inputs or executables. Finally, we wanted to reuse existing workflow code as much as possible. Reusing code is important for two reasons. First, the existing workflows include a significant amount of code (over 10,000 lines for the forecasts alone). Implementing this existing functionality in a new language would be extremely time consuming and could introduce new bugs. Second, it allows programmers to continue programming in their language of choice, which facilitates adding new tasks to the current workflows.

We describe the steps we have taken towards meeting the goals above for the CORIE workflow using Thetus. In particular, we present specific features we have implemented in our system to address the challenges posed by the CORIE workflows. We have implemented a novel architecture for managing both data products and tasks. Our architecture includes two types of tasks: *generation* tasks that perform simulations or analysis to generate data products, and *management* tasks that automatically perform system maintenance to improve data product management and task execution. We discuss our experiences incorporating the existing CORIE workflows into our experimental architecture, and show how we successfully implemented a workflow with minimal modifications to existing code. We also consider the challenges in effectively scheduling the execution of CORIE workflow tasks on a Grid, and present preliminary results showing the performance of the existing workflow on our implementation. While we use CORIE as a motivation and testbed for our work, we believe that our solutions are applicable to other scientific domains. For example, the problems of lineage tracking, metadata management, and Grid execution apply to many scientific workflows.

We note that there is a considerable amount of research in large-scale Grid computing systems, e.g., Globus [6]. While our research shares many goals with such systems, including speeding up task execution and exploiting available resources, there are several key differences. Large-scale Grid-computing systems typically provide computing resources to a large set of geographically distributed users, who submit jobs to the system and share resources with other users. In contrast, the CORIE workflow is executed by a small group of scientists using a set of dedicated machines connected by a local area network. These machines are unlikely to be committed to a larger Grid, as their resources are generally completely consumed by regularly scheduled production runs. We therefore focus on speeding up individual workflow executions using multiple local computing nodes, rather than sharing globally distributed computing resources and data among a large number of users. Thus, our task scheduling and replica management solutions differ from those in large scale Grid systems.

Our main contributions are as follows:

- We present a case study of our experiences implementing a data product management system for an existing scientific workflow.
- We present an architecture for managing both data products and the tasks that generate them, including facilities for tracking lineage and metadata and maintaining multiple versions of files.
- We identify key challenges to executing scientific workflows on a Grid, and propose scheduling techniques to address these challenges.
- We evaluate our scheduling techniques using our prototype implementation.

This paper is organized as follows: Section 2 presents an overview of the CORIE system, describes some challenges posed by the existing CORIE workflows, and outlines the goals of our experimental implementation. Section 3 describes the details of our implementation, including special features we implemented and our experiences with the CORIE workflows. We discuss task scheduling challenges and present preliminary results in Section 4. We survey related work in Section 5, and conclude in Section 6.

2 CORIE Environmental Observation and Forecasting System

CORIE [4] is an environmental observation and forecasting system for the Columbia River estuary. The CORIE system both measures and simulates the physical properties of the estuary. Applications have addressed issues spanning salmon habitat and passage, hydropower management, navigation improvements and habitat restoration. The system includes both *forecast* and *hindcast* simulations. *Forecasts* are used to predict near-term conditions in the river, while *hindcasts* are run retrospectively for specific time periods using observed values. Hindcasts typically consist of an extended set of simulations covering a year. They may also include calibration runs to test the sensitivity of the model to empirical parameters.

2.1 CORIE Workflows

The existing CORIE workflows consist of a collection of executable programs written in Fortran, C, and Perl. In some cases one program directly calls one or more other programs. In other cases two programs execute concurrently, one of which appends data to files while the other periodically reads the files and processes the newly appended data.

We first present a sample of the CORIE forecast workflow and discuss some challenges for it, then present the goals that we address in our implementation. In the remainder of this paper we focus on the CORIE forecast workflow, but we note that the hindcasts pose similar challenges.

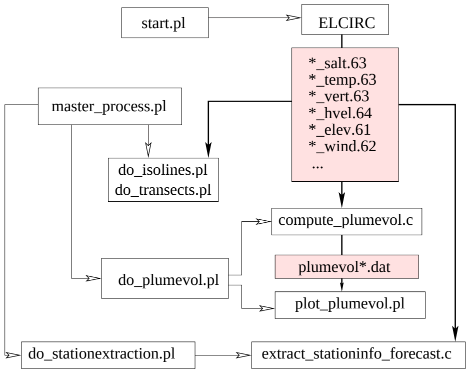

Figure 1: A segment of the CORIE forecast workflow. White boxes are tasks, shaded boxes are data files.

Figure 1 shows a segment of the forecast workflow. We note that this is only a subset of the tasks, inputs, and outputs in the CORIE workflow; however, this subset illustrates the challenges posed by both the forecast and hindcast workflows. For simplicity we omit the final derived data products (e.g., datasets, images, animations).

In this figure, white boxes represent executable tasks, which may be written in Fortran (e.g., ELCIRC), C (e.g., `compute_plumevol`), or Perl (e.g., `plot_plumevol`). Shaded boxes represent data files. Bold arrows show which data files are inputs to each executable, while light arrows indicate that one executable calls another. Files with names ending in `.61, .62, .63,` and `.64` are generated by the ELCIRC simulation [17]. These files contain values of different variables, such as salinity and velocity, computed over a 3D finite-element grid at multiple time steps. These files are used to generate many images such as transects (vertical slices of the river) and isoline plots (horizontal slices). A single forecast run generates several files of each type, e.g., `1_salt.63, 2_salt.63`, etc. The program `compute_plumevol.c` computes the volume of the *plume*, a region of water beyond the mouth of the river with salinity below a given threshold. It executes with several different threshold values and generates a `plumevol*.dat` file for each. These intermediate products are inputs to `plot_plumevol.pl`, which generates a plot of the derived data values.

Several Perl scripts start the simulation run (we show two of these, `start.pl` and `master_process.pl`, in Figure 1). The script `start.pl` launches the ELCIRC simulation, while the script `master_process.pl` executes several Perl scripts including `do_isolines.pl`, `do_transects.pl`, `do_plumevol.pl` and `do_stationextraction.pl`, each of which reads one or more of the files generated by ELCIRC as input. These scripts execute concurrently, as each Perl script is run periodically to process any data appended to the ELCIRC outputs since the script's last execution.

The forecast workflow is executed daily. A single forecast run generates about 5GB of data and over 30,000 individual files. Hindcasts are run as needed, and a single run generates about 20GB of data and over 10,000 individual files.

2.2 Challenges

We now consider specific challenges posed by this workflow. The first challenge is *executing the workflow on a Grid*. The CORIE workflow is characterized by large files and data- and memory-intensive tasks. The current workflow implementation runs on a shared filesystem, which provides convenient access to files across multiple nodes but poses several difficulties. First, all tasks in a single run execute on a single machine, which misses many opportunities for concurrent execution. There is no easy way to coordinate workflow execution among multiple machines. Second, there are no facilities for concurrency control or tracking changes to a file. Important files could potentially be corrupted or overwritten if multiple users execute tasks on them simultaneously. Third, storing and locating relevant inputs and data products is nontrivial. Earlier forecast runs may need to be archived onto tertiary storage, which involves transferring large volumes of data. The current workflow implementation relies on a hard-coded directory structure, so existing code may break if files are moved.

In contrast, executing these tasks on a Grid can overcome some limitations of filesystems and potentially reduce the overall execution time. However, many tasks require one or more large files as input. Transferring these files to a remote machine (hereafter referred to as a *node*) may take over 30 seconds or even several minutes, which may significantly increase the task's execution time. Thus, determining an optimal task scheduling strategy is non-trivial. Further, the current workflow code does not easily lend itself to Grid execution. In some cases a single executable performs the same computation on multiple files. Different files could be processed concurrently at different nodes to improve performance. However, in some cases it is more efficient to execute a task on a single node. Therefore, we must determine when splitting tasks into smaller tasks improves performance. A related challenge is adding and removing nodes with minimal overhead. Adding or removing nodes should not affect other nodes in the system, and should not require transferring large amounts of data.

Another challenge to executing this workflow is *adding and modifying tasks*. Adding tasks to the CORIE workflow requires manually editing several executables. Programmers must ensure the new task does not interfere with other tasks. We would like the addition of new tasks to be transparent to existing tasks. In addition, adding tasks should not require changes to nodes, and modifications to a task (e.g., fixing a bug, using a new simulation model) should be propagated to nodes in a scalable manner.

A third challenge is *tracking lineage and metadata*. *Lineage* refers to information on other files used in the generation of a data product, including both inputs and executables. *Metadata* refers to other parameters to describe the data including execution time and input parameters, as well as information that can be extracted from file names or headers and may be known only to task programmers. In earlier research [9] we described a system to automatically extract and organize metadata from files. We use similar techniques in our implementation as described in Section 3.3.3.

2.3 Goals

Based on the above challenges, we had several goals for our experimental system. The first three relate to the creation of data products, while the last two relate to the management of the data products themselves. We address each of these goals in Section 3.

1. Speeding up execution
2. Easily adding and modifying data products and tasks
3. Seamlessly adding nodes
4. Storing metadata
5. Tracking lineage

3 Implementation

We built our data product management system using the Thetus Publisher [13]. Thetus provides data management facilities for non-text data common in many scientific domains. We first present an overview of Thetus, including a description of its capabilities and architecture, along with terminology used in this paper. We then present several unique data management features we implemented using Thetus components, and discuss our experiences implementing the existing CORIE forecast workflow using Thetus.

3.1 Thetus Overview

Thetus consists of two main components, a *publisher* and one or more *task servers*. The publisher provides storage and query facilities for data products and their associated metadata. It also allows the definition of multiple namespaces for different domains. Finally, it provides storage for executable tasks and facilities to launch each task on one of the available task server nodes when appropriate launch conditions are met. Executing tasks on multiple nodes can potentially speed up workflow execution by allowing independent tasks to run concurrently. (We discuss performance and scheduling issues in detail in Section 4.)

Thetus includes the following entities:

- **Data Files:** These are files that are uploaded to Thetus and may be associated with a set of *properties* (described below). The Thetus Publisher was designed to store data products and their associated metadata properties. However, in our implementation we used the publisher to store other types of files as well (described in Section 3.3.1).

- **Properties:** These are metadata attributes associated with data files or descriptions (defined below). Properties can be of any basic type such as integer, float, or string. Properties can also be defined "on the fly" when a file is uploaded, and will be automatically created by the publisher with a type of "undefined". This capability means that users need not define a schema in advance, and can extend existing schemas to meet their needs.

- **Descriptions:** A *Description* is a set of property-value pairs that describe an object or concept and can be shared among multiple entities. Descriptions can be used for many different purposes in Thetus; we describe some uses later in this section.

- **Dictionaries:** A *Dictionary* is a set of properties for a domain (akin to a *Namespace* in XML).

- **Profiles:** Every data file is uploaded using one or more *data profiles*. Profiles contain metadata that identifies all data files associated with the profile. A profile may also include one or more *tasks* (described below) that are launched when certain conditions are met, e.g., when one or more files are uploaded, or the presence or absence of certain properties.

- **Tasks:** Thetus tasks are Java programs that execute on one or more data files as input and produce one or more data products as output. They may execute on any available *task server*. Thetus tasks may call programs written in any language. Calling existing programs from Thetus tasks enables programmers to reuse existing code as much as possible, as described in Section 3.4.

Every object in Thetus (e.g., profiles, properties, tasks, data files) is identified by a unique ID.

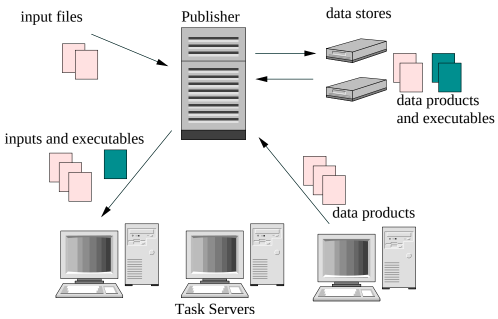

Figure 2: System Architecture

3.2 Architecture

We present our architecture in Figure 2. Data files stored at the publisher hold all relevant entities: inputs, executables, and data products. The publisher may be connected to one or more data stores, which allows new storage to be added to the system as the volume of data grows.

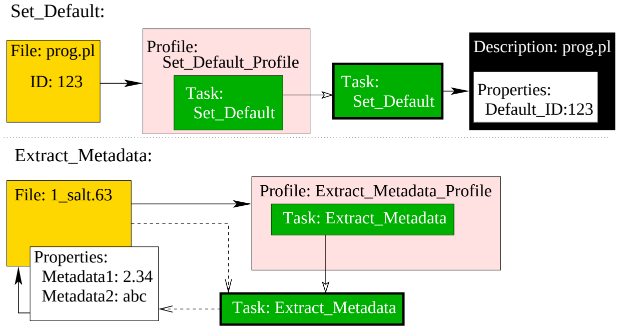

Figure 3: Execution process of two tasks

We illustrate the launching and execution of two tasks in our system in Figure 3. We defer the details of these tasks to Section 3.3. When a data file is uploaded (either by a user or by a task that generated it), it is associated with one or more *profiles* that may trigger associated *tasks*. For example, in Figure 3, when the file `prog.pl` is uploaded to the profile `Set_Default_Profile`, it launches the `Set_Default` task. A task may run on any of the available *Task Servers*, and the selected task server downloads all required inputs and executables for the task. After the task completes, the task server uploads the resulting data products back to the publisher.

We note that a shared filesystem could also provide transparent file access on a set of local nodes. However, we believe our architecture provides several advantages over this approach. In a shared filesystem, input files still need to be transferred to the machine where a task is executing. Since the CORIE workflow tasks scan their entire input files, these files need to be transferred in their entirety to nodes. Thus, a shared filesystem would incur a comparable data transfer overhead to our architecture, without any of the other benefits (described below) that our architecture provides. For example, our architecture allows storage of data files on local nodes (described in Section 3.3.4) to eliminate redundant downloads, while a shared filesystem may transfer a large file to a node multiple times if several tasks operate on the file.

A novel feature our implementation is providing storage and maintenance facilities for both data products and executables, which enables tracking their lineage and update histories and provides access to the actual executables that generated each product. Storing entire executable files is preferable to storing a path name, since moving or renaming files could cause a loss of lineage information.

Our implementation supports two types of tasks: *generation* tasks to generate derived data products, and *management* tasks to automatically maintain executables and data product metadata. Management tasks perform needed data and task maintenance to improve data-product management and task execution. Our implementation of management tasks exploits the workflow execution capabilities of Thetus; these tasks are automatically launched by Thetus when the needed conditions are met. Example management tasks include updating executable versions and extracting relevant metadata from data products when they are uploaded. We present detailed examples of management tasks in Section 3.3.

3.3 Features

We now discuss the details of several unique features of our implementation that enable us to meet the goals outlined in Section 2.3.

3.3.1 Storing Executables

The Thetus publisher provides useful facilities for storing and querying data products. We found that it was also useful for storing executables. We use the term *executable* to refer to any existing program in the workflow that takes one or more files as input and generates one or more data products. These include Perl scripts and compiled C programs[1]. We also used the Thetus publisher to store templates and header files that serve as inputs to the executables. A key advantage to storing executables as Thetus objects is that they can have associated metadata, e.g., platform or version information.

Storing executables in Thetus helps us meet several of our goals described in Section 2.3. First, it helps us meet Goal 2 of being able to easily add and modify tasks. When an executable is modified (e.g., a bug fix or a new simulation model), the new version is uploaded to Thetus and assigned a unique Thetus ID. Older versions of the same executable remain stored in Thetus for historical purposes, i.e., to track the lineage of older data products and regenerate them as needed. (We discuss identifying the current or default version of an executable in Section 3.3.2.)

Second, storing executables in Thetus helps us meet Goal 3 of easily adding task server nodes. There is no need to copy executables to new nodes (nodes need only utilities such as Perl and JRE to ensure they can run executables). Executables are downloaded on-demand by nodes when tasks are executed.

Third, storing executables helps us meet Goal 5 of tracking lineage information on data products. When a newly generated data product is uploaded to the task server, we set one of its properties to identify the Thetus ID of the executable that generated it. We also store the IDs of each input file as discussed in Section 3.4. Storing these values allows us to easily identify which data products were generated by each version of the executable, and associates each data product with the actual file that generated it rather than a file name or path. We note that the executable files are small relative to the size of the derived data products, so the storage overhead is negligible.

3.3.2 Storing Current Versions

Programmers may modify executables to fix bugs, try new models, or define new data products. We store all versions of executables in Thetus to track lineage of older data products and to enable regenerating products or providing similar runs. However, a challenge is identifying which version to download for a particular execution. Identifying an executable explicitly by its Thetus ID is not a good option, as it would require modifying all tasks that use this executable every time a new version is uploaded to the publisher.

Our solution to this problem is to implement a management task to track the "current" version of a file. Tracking the current version helps us meet Goal 2 of easily adding and modifying tasks. Our implementation enables any task to reference an executable by name without any knowledge of which version is currently in use. Thus, new versions of executables can be seamlessly incorporated into existing workflows.

In our implementation (task `Set_Default`, shown in Figure 3), we create a description associated with each executable file name (e.g., `prog.pl`), and assign a property to this description with the Thetus ID of the current version of the file. When a new `prog.pl` is uploaded to `Set_Default_Profile`, Thetus launches the `Set_Default` task, which assigns the ID of the new file to the `DefaultID` property of description `prog.pl`. Storing the ID of the current version provides two key benefits: First, it allows Thetus tasks to refer to executables by name without knowing a Thetus ID. Second, it makes changes in executables transparent to any Thetus task that uses them, so existing Thetus tasks do not need to be modified when an executable changes.

This feature also allows us to revert to a previous version of an executable. For example, if a new simulation model turns out to be less accurate than an earlier model, we can change the default version of the executable back to the earlier model. We can also store multiple implementations of executables (e.g., serial and parallel, versions for different platforms) and we plan to use management tasks to support this functionality.

3.3.3 Metadata Tracking and Extraction

A key benefit to using the Thetus publisher is that metadata can be associated with a data product *when it is uploaded*, which facilitates tracking information about the file such as lineage, input parameters, and time and date of execution. Our implementation builds on this capability by automatically executing management tasks to extract metadata from files when they are uploaded, using scripts written by task programmers as described in previous work [9]. We show an example in Figure 3. When a file with a name ending in `.63` (e.g., `1_salt.63`) is uploaded to the `Extract_Metadata` profile, Thetus launches the `Extract_Metadata` task and assigns the metadata as properties of the data file. Automatically extracting metadata enables users to immediately query the file's metadata properties, and helps us meet Goal 4.

3.3.4 Storing data at nodes

Many input files, data products, and executables are used multiple times in the same workflow. Recall from Section 2 that input files in the CORIE workflow may be 300MB or larger. Transferring these files takes 30 seconds or longer on our local area network, and data products generated at a given node may serve as input to subsequent tasks. In these cases it is wasteful to repeatedly download a file from the publisher, as this consumes large amounts of bandwidth and could add

---

[1] We note that in our current implementation all nodes have the same platform. However, our implementation can easily be extended to store different compiled programs for different platforms, or to compile the source code locally at the node.

several minutes to a task's execution time. However, we want to ensure that the local copy of a data product or executable is the correct version for a given task.

Our solution is to implement a local data store at each node that stores all files downloaded from the publisher along with their Thetus IDs. Local data stores reduce the number of files downloaded from the publisher and can significantly reduce the running time of a task, thus helping achieve Goal 1 of speeding up task execution. When a task needs an input file or executable from the Thetus publisher, it first contacts the publisher to get the ID of the current version. Since contacting the publisher requires a small data transfer over a local network, it has negligible overhead compared to the cost of downloading large files. If the ID matches the ID of the local version, the task does not need to download the file. If the IDs do not match, the new version of the file is downloaded from the publisher. Checking file IDs ensures that nodes always use the current version of a file with minimal overhead at the nodes and the publisher.

We are currently investigating replacement policies when local data stores are full; however, we believe that Least Recently Used (LRU) or existing web caching policies may work well in many cases.

3.3.5 Scheduling at Multiple Nodes

Scheduling independent tasks at multiple nodes is another way to meet Goal 1 of speeding up workflow execution. However, simply adding nodes to the system without intelligent task scheduling does not guarantee significant performance improvements. Tasks vary in both their running time and the size of their inputs and outputs, and data transfer overhead may be high. Task execution times vary from under 30 seconds to several hours. As we will show in Section 4, it is important to consider both the size of the input and the expected running time of a task when scheduling task execution on the available nodes. Poor scheduling decisions may result in performance *worse* than executing the entire workflow on a single machine.

We are currently developing an intelligent scheduler that considers the size and current locations of input files in addition to the capacity of each node and task execution times. We discuss scheduling challenges and present preliminary performance results in Section 4.

3.4 Implementing CORIE Workflows

We now describe our experiences implementing the existing CORIE workflow using Thetus. We first present our implementation and schema. We then discuss how we ran the existing executables as Thetus tasks.

3.4.1 Workflow Implementation

We have implemented a large portion of the CORIE forecast workflow using Thetus, including the tasks `do_plumevol`, `do_intrusionlength`, `do_isolines`, and `do_transects`. Recall from Figure 1 that all of these files take a subset of the `.61,.62,.63,.64` ELCIRC files as inputs. We describe how the details of each of these tasks and how we implemented them in Thetus below.

The plume volume and intrusion length tasks each take the files `1_salt.63` and `2_salt.63` as inputs, and generate a set of intermediate data files, which are then used to generate images. To implement the plume volume and intrusion length tasks in Thetus, we created two profiles. We describe the implementation of the plume volume tasks, the implementation of intrusion length is comparable. The first profile, *do_plumevol*, launches the `do_plumevol.pl` task when the needed ELCIRC files are present. This task generates the intermediate data files and uploads them to the publisher. The second profile, *plot_plumevol*, launches a task after all of the intermediate data files are uploaded to the publisher. This task generates the final images and uploads them to the publisher.

The isolines and transects tasks generate images for multiple ELCIRC input files, each of which is independent of the other input files and takes unique input parameters. In our implementation we created a separate profile for each of the ELCIRC input files, each of which launches the isolines (resp. transects) task with the appropriate input parameters for the file. Note that uploading a single file, e.g., `1_salt.63`, may launch tasks from multiple profiles.

The CORIE forecast workflow depends on the arrival of several inputs, including observations and data from other models. In our current implementation we initiate tasks manually by uploading all needed files. Initiation of workflows is an area of ongoing work.

3.4.2 Schema

We created two dictionaries, one for CORIE-specific metadata properties such as temperature, salinity, and depth, and one for workflow-specific properties. Workflow properties of particular interest are *GeneratedFromExecutable*, which contains the Thetus ID of the stored executable file that generated the data product, and *GeneratedFromInput*, which contains the Thetus ID of an input file to the executable that generated the data product. A single data item may contain multiple occurrences of the *GeneratedFromInput* property, one for each input file. Another workflow property is *dateString*, a unique string identifying the forecast run for a particular day that labels a set of data products generated from the same run. Thetus includes additional built-in task metadata properties, such as the time executed, user, and the node where the task executed, that are recorded automatically whenever a task executes.

3.4.3 Incorporating Existing Code

One of our goals was to design a system that could efficiently execute existing workflows with minimal modifications to the existing code. Clearly, rewriting existing scripts would require excessive overhead and would be difficult to convince programmers to adopt. Instead, we implemented Thetus tasks in Java to call the existing executables.

We faced several practical challenges to running existing CORIE executables as Thetus tasks. First, many tasks require a variety of inputs including template and header files in addition to the uploaded files that trigger the task execution. Thus, Thetus tasks that run executables must also download all additional required inputs. Second, existing C programs and Perl scripts in the CORIE forecast workflow rely on a predefined directory hierarchy. These tasks expect input files located at hard-coded paths and generate data products in other hard-coded directories. A challenge to implementing the CORIE workflow was enabling such tasks to run on any node without requiring modifications to the node's local directory structure, and with minimal modifications to the existing code.

Our solution to this problem is to have the Thetus tasks create local directories in `/tmp` as needed, corresponding to the directories specified in the executable code, before running the executable. The Thetus task deletes the directories when the executable completes. This solution makes no assumptions about the directory structure on the nodes, which allows new nodes to be added seamlessly and does not interfere with a node's local filesystem.

We also modified several tasks, e.g., `do_transects.pl` and `do_isolines.pl`, to enable task splitting (i.e., breaking a larger task into smaller tasks, described in Section 4). In the existing CORIE workflow code both of these tasks iterate over multiple files, generating transect or isoline images and animations for each. In our implementation, we modified this code to generate a single transect or isoline image, given the appropriate file name and parameters as input. This change required modifying only a few lines of the existing Perl code, so the programming overhead was minimal.

For some workflow tasks, the execution time is small and there may be little or no benefit to task splitting. We discuss this point further in Section 4. The CORIE workflow also includes several incremental tasks, where one tasks periodically appends data to a file that another task reads. We are developing techniques to incorporate such tasks into our implementation.

4 Task Scheduling

In this section we describe the current status of our task-server nodes and scheduler. We then discuss the several issues to be considered by a scheduler, and present preliminary performance results showing the potential benefits of using an intelligent scheduler.

Node	OS	Memory	Speed
1	Fedora Core 1	1GB	2.80GHz
2	Redhat Linux 9	1GB	2.60GHz
3	Redhat Linux 8	3GB	2.40GHz

Table 1: Node Properties

4.1 Current Implementation

We currently have three task server nodes, labelled 1, 2, and 3. We summarize information on each of these machines in Table 1. Node 1 runs on the same machine as the publisher, although the publisher treats the node as a remote machine in our current implementation. Nodes 2 and 3 run on two remote machines connected to the publisher by a local area network.

A *task scheduler* runs on the Publisher at Node 1. When a task is ready to execute, the scheduler assigns it to one of the available nodes. By default, the scheduler chooses among nodes 1, 2, and 3 in a round-robin manner, without considering current load on the node or the location of large files on any of the three nodes. Clearly this naive scheduling could potentially perform poorly. In the next section, we discuss issues to be considered by the scheduler that we are currently incorporating into our implementation. We present preliminary performance results in Section 4.3.

4.2 Issues

4.2.1 Task Splitting

The first issue relates to determining when to split an existing task into smaller tasks. As mentioned in Section 3.4, some existing workflow tasks perform the same computation sequentially on multiple files, and can easily be executed concurrently. For example, the tasks `do_isolines.pl` and `do_transects.pl` (hereafter referred to as isolines and transects) generate images and animations of horizontal and vertical cross-sections of different conditions of the river for variables such as salinity (`salt`), temperature (`temp`), and horizontal and vertical velocity (`hvel` and `vert`). Executing isolines and transects on some of these files can take several minutes (see Figure 4 below). The task execution time is significantly longer than the time needed to transfer files to the execution nodes. Thus, splitting isolines and transects into smaller tasks, one for each file, may be beneficial. Splitting the tasks allows them to execute simultaneously at separate nodes. We use this example in the results presented in Section 4.3.

However, some tasks do not significantly benefit from this increased concurrency and should be left intact. For example, the task `do_intrusionlength.pl` operates on the files `1_salt.63` and `2_salt.63` (334 MB each). For a single `salt.63` file, the task generates data products for 5 different input parameters, each of which takes about 10 seconds. Thus, for a single `salt.63` file the running time is about 50 seconds, and the total running time for both files is about 100 seconds. Recall that downloading a `salt.63` file to a node takes up to 40 seconds in our implementation. Thus, running the entire task on a single node takes about 40+40+100 = 180 seconds, while executing the `1_salt.63` and `2_salt.63` tasks on separate nodes takes about 90 seconds at each node. In the best case, executing on separate nodes would speed up execution by 90 seconds, but in practice the speedup is likely to be less due to both file transfer delays and waiting for nodes to become available (i.e., waiting for currently executing tasks to complete). Further, if the `salt.63` files are already stored locally at one or more nodes, splitting would speed up execution by only 50 seconds in the best case.

Splitting small tasks may also delay the execution of other ready tasks that could use some of the available nodes, and in the worst case could increase total running time. We are currently investigating heuristics to determine when task splitting is beneficial.

4.2.2 Data Location

The second issue relates to choosing nodes based on the location of input files. We refer to this as *data-aware* scheduling. Data-aware scheduling aims to assign tasks to nodes to minimize the volume of data downloaded to each node. For example, consider a task requiring one or more large inputs. If two nodes are available, the node that has more of the required input files stored locally may be the best choice because it can significantly reduce the data transfer time. We note that the optimal choice depends on other factors as well including the load and available bandwidth at each node. Another approach we plan to investigate is proactively downloading files to nodes to reduce data transfer times.

4.2.3 Global Workflow Information

A third issue relates to incorporating global workflow information into scheduling decisions. We refer to incorporating global information as *workflow-aware* scheduling. Workflow-aware scheduling considers both currently ready tasks and future tasks when assigning tasks to nodes and aims to find the optimal grouping of tasks among available nodes. For example, consider a set of several short tasks and one long task executing on two nodes. If the short tasks become ready before the long task, a naive scheduler may divide short tasks evenly among both available nodes. However, if the long task is expected to be ready soon, it may be more efficient to assign all the small tasks to the same node, leaving the other node free to execute the long task when it becomes ready. We note that Thetus metadata tracking facilities enable us to store task statistics such as running times and nodes where tasks executed, which facilitates workflow-aware scheduling.

4.3 Results

We now present preliminary performance results on our implementation that show the potential benefits of task splitting, and data and workflow-aware scheduling. We are developing a scheduler that will incorporate this information into scheduling decisions. As our example we use `do_isolines` and `do_transects` tasks described in Section 4.2. Running the existing scripts on a single machine with 1GB of memory and a speed of 2.60GHz (Node 2 from Table 1) together takes 19-20 minutes on average, assuming all input files and executables reside on the machine. In a single forecast run, these tasks are run on four different sets of inputs for a total running time of up to 80 minutes.

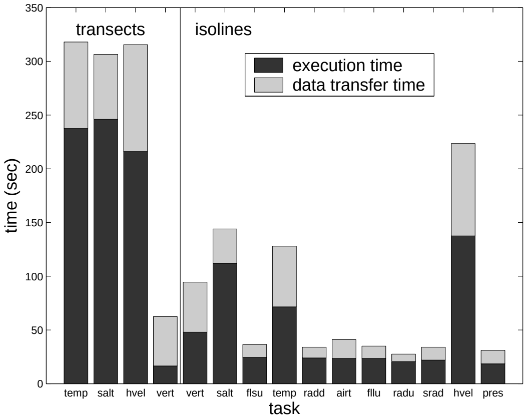

Figure 4: Data Transfer and Execution Times of `do_isolines` and `do_transects` on task server nodes for each input file

Figure 4 shows the average execution times and data transfer times for the transects and isolines tasks on each file when we ran them in our implementation. These values are the averages over several runs on all three nodes. We note that the data transfer overhead was slightly lower on Node 1 since it runs on the same machine as the publisher, but the overall performance of the three nodes was comparable. For these experiments we did not use any local file storage, so all needed inputs and executables were downloaded once per task. The upper portion of each bar shows the amount of time to download all needed files to a node, including both inputs and executables. The lower portion of each bar shows the total time to execute the task at the node, including the time to upload any data products generated by the task.

The file transfer overhead is largest for tasks that run on `hvel.64` files (about 655MB each), and is also significant on the `temp.63, salt.63, and vert.63` files (about 334MB each). The remaining files that the isolines task runs on (`flsu.61, radd.61, airt.61, fllu.61, radu.61, srad.61, pres.61`) are about 23MB each. While these smaller files have a much lower data transfer overhead, it is significant compared to the total task execution time. Transects tasks have slightly higher data transfer overhead than the isolines tasks because they take several header and template files as inputs in addition to the larger files. The execution time is largest for the transects task on the `temp, salt` and `hvel` files. The execution time is also significant for the isolines task on these files.

4.4 Assigning tasks to nodes

We now consider two possible assignments of the tasks above to the three available nodes, to illustrate both potential pitfalls and potential benefits of executing the existing workflow on a Grid.

In the first assignment, we used the scheduler's default settings and allowed it to arbitrarily assign tasks to nodes in a round-robin manner. Table 2 shows which files and tasks the scheduler assigned to each node. The first letter (i or t) indicates the task (isolines or transects), followed by the name of the file processed by the task. We note that in this experiment we did not store data files locally at any nodes. However, the scheduler assigned the isolines and transects tasks on each of the large files (`salt, temp, vert, hvel`) to two different nodes, so there was no benefit to storing these files locally at the nodes.

The total data transfer times and execution times of the tasks listed in Table 2 are shown in Figure 5 for each of the three nodes. Node 1 has the smallest total data transfer and execution overhead (about 400 seconds, just under 7 minutes). It runs tasks on three of the largest input files (`temp, salt, vert`) each about 334 MB for a total of about 1GB. It runs the isolines task on `temp` and `salt` (i-temp and i-salt), which together take under 200 seconds, as well as three tasks with execution times around 20 seconds, transects on `vert` (t-vert), i-flsu, and i-srad.

In contrast, Node 2 has significantly higher data transfer times and execution times. It executes tasks on `temp, hvel` and `vert`, which have a combined size of over 1.3GB, and runs two tasks with high execution times, (t-temp and i-hvel, together just under 400 seconds) as well as i-vert (about 50 seconds) and two smaller isolines tasks.

Finally, two of the three tasks with the longest execution times (t-hvel and t-salt, over 450 seconds together) both run on Node 3, which explains its large execution time. The two largest input files, `hvel` and `salt` together are about 1GB. We note that the data transfer time at Node 3 is higher than Node 1 partly because the transects task contains larger inputs and executables than isolines, and partly because Node 1 runs on the same machine as the Publisher.

Node 3 has a total data transfer and task execution time of over 800 seconds (over 13 minutes), which is a the maximum execution time for the entire workflow with this grouping. While this grouping improves on the original 20 minute execution time, it does not fully exploit the potential benefits of our system.

1	2	3
i-temp	t-temp	t-salt
i-salt	i-hvel	t-hvel
t-vert	i-vert	i-fllu
i-flsu	i-radd	i-radu
i-srad	i-airt	i-pres

Table 2: A (suboptimal) grouping of tasks among the three available nodes

1			2	3
t-vert	t-temp	i-radd	t-salt	t-hvel
i-vert	i-temp	i-radu	i-salt	i-hvel
i-airt	i-pres	i-srad	i-fllu	i-flsu

Table 3: An improved grouping of tasks among the three available nodes

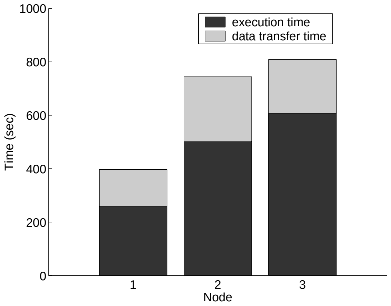

Figure 5: Data transfer times and execution times for the task grouping in Table 2

Next, we consider an improved grouping of the tasks that uses both data and workflow-aware scheduling. We manually assigned tasks to each node as shown in Table 3. We describe our optimizations below.

We first performed data-aware scheduling by assigning isolines and transects tasks that operate on the same inputs (`temp, salt, vert, hvel`) to the same node. We next performed workflow-aware scheduling. Observe from Figure 4 that the combined execution times of isolines and transects on `hvel` and `salt` are comparable, and the combined execution time of isolines and transects on `temp` and `vert` together is comparable to both of these (in all cases, 370-400 seconds). Thus, we assigned both `temp` and `vert` tasks to Node 1, both `salt` tasks to Node 2, and both `hvel` tasks to Node 3. Since Node 1 has a slightly lower file transfer overhead than Nodes 2 and 3, we assigned five of the smaller isolines tasks to this node as well, and assigned the remaining two tasks to Nodes 2 and 3.

The data transfer times and execution times are

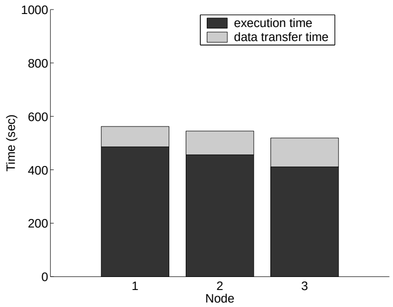

Figure 6: Data transfer times and execution times for the task grouping in Table 3

shown in Figure 6. In this case, all three nodes have comparable combined data transfer and execution times. Using this schedule, the total data transfer and execution time at each node is under 10 minutes, so we have reduced the running time of isolines and transects by half. Since these two tasks are run 4 times in each forecast, we can reduce the total running time from 80 minutes to under 40 minutes. Thus, intelligent scheduling can significantly speed up existing workflows. Naive scheduling algorithms do not guarantee significant benefits, and in pathological cases may perform worse than running the tasks on a single node. We are extending our implementation to attain similar benefits for other workflow tasks.

5 Related Work

5.1 Grid Computing

There is a significant amount of interest in using Grid computing resources to execute large scale scientific workflows. Systems such as Globus [6] and Condor [10] provide facilities to submit jobs for execution on a set of distributed nodes. However, these systems do not consider workflow management. Further, they are designed for a large number of globally distributed users and nodes, and are not suited to efficiently executing a single workflow on a set of dedicated nodes.

JOSH [15] provides load-aware and data-aware multi-site scheduling, and shares our goal of efficiently executing a set of tasks on a Grid. In this system, clients can submit tasks for execution on one or more nodes. The system chooses a node to execute the task based on both the cost of transferring the required files to the node and the current load at the node. It does not consider storing files locally or global workflow information, so it may make poor scheduling decisions.

The problem of replica management in Grid environments has also received a considerable amount of attention [2, 5, 16]. However, the emphasis of this research is on improving global data availability, and does not consider storing data products at nodes or data-aware scheduling. Research on replica selection for Grid applications [16] does not consider storing local data copies to reduce workflow execution time or considering replica location when scheduling tasks as we do. It also does not track which replica is the "original" copy and does not provide strict consistency guarantees among replicas. Weak replica consistency is well-suited to the large scale distributed environment considered in Grid replica management services [2], but is inappropriate for our applications.

5.2 Scientific workflows

Several tools [1, 3, 7, 11] provide facilities for the specification and execution of scientific workflows.

Chimera [7] provides a virtual data system that stores procedures to derive data products, enabling users to track lineage or regenerate data products. It also considers replica selection and task scheduling, but does not address task splitting or data and workflow-aware scheduling. Zoo [1] facilitates scientific workflow execution by defining workflows as object-oriented database schemas. The emphasis is on easily specifying workflows by exploiting DBMS functionality, and this work does not consider the challenges of executing tasks on a Grid or managing data products discussed in this paper.

More recently, GridDB [11] provides a data-centric overlay for Grid data analysis. As in Zoo, this work exploits DBMS functionality for specifying and running scientific workflows. In addition, it supports task execution on a Grid and presents several services to improve performance. Specifically, it supports interactive query processing, allowing users to prioritize task execution on a data subset of interest; and memoization, allowing reuse of previously generated data products. While these features are useful for many scientific workflows, GridDB does not address many of the challenges we face. For example, the authors do not consider the problems of determinining where to execute tasks, instead relying on existing middleware [6, 10]. They also do not explicitly address the challenges of task splitting and managing large files and data products.

Kepler [3] provides interfaces and tools to specify and execute scientific workflows. The emphasis is on formalizing workflows and providing access to heterogeneous data. Kepler provides a set of predefined Java *actors* that can perform computations on data, and provides facilities to implement new actors. However, support for calling external programs in other languages is limited, and the system does not provide storage or lineage tracking facilities for data products.

5.3 Lineage Tracking

Recently there has been considerable interest in tracking the lineage or provenance of data products [12]. PASOA [14] provides a system for automatically tracking data lineage during workflow execution. This work addresses many of the same challenges we do including identifying which tasks generated a data product and reproducing results. However, this system requires modifications to the workflow execution engine, and recording lineage may require excessive communications between clients and servers. The Earth System Science Workbench (ESSW) [8] provides a non-intrusive data management infrastructure for tracking data product lineage. ESSW uses wrappers around existing scripts to log lineage information and stores lineage in XML files. As in our work, their approach requires minimal modifications to existing code. However, this work does not track versions of files and does not provide workflow management facilities.

6 Conclusions and Future Work

There has recently been considerable interest in improving the management and execution of large-scale scientific workflows. However, little attention has been given to managing workflow execution locally on a set of dedicated nodes, where multiple workflow tasks execute on the same files and data transfer costs may consume a large percentage of total execution time.

In this paper we have described our experience implementing a data product management system for an existing scientific workflow, and presented several data management challenges and solutions in this environment. We have presented a novel architecture for storing both data products and executables. Our system supports *generation tasks* to derive data products and *management tasks* that create and maintain executables and metadata. We have shown how this system can execute tasks on multiple nodes of a Grid, and illustrated the benefits of incorporating data location and global workflow information into task scheduling.

We are considering several areas of future work:

Data-Aware and Workflow-Aware Scheduling: We are developing heuristics to implement data and workflow aware scheduling as described in Section 4. We plan to store statistics on execution times and data transfer times, and use heuristics based on global workflow information to assign tasks to nodes.

Task Sets: We plan to support *task sets*, grouping a set of related tasks into a workflow. Identifying related tasks is useful for workflow-aware scheduling, and can identify failures if some tasks in a set fail to execute.

System Monitoring: We are developing an interface for clients to monitor task execution, including what tasks are currently running, what has completed, and status of tasks, e.g., error messages, if any. An interface can also highlight tasks that have failed to launch, and identify problems with workflow execution.

Production Planning: We plan to provide facilities for *production planning*. Production planning allows users to predefine a workflow or set of workflows for future execution. For example, a user may specify a year's worth of hindcast runs and schedule them to begin at pre-specified times.

7 Acknowledgments

We thank everyone at Thetus, especially Danielle Forsyth, Roy Hall, Tyler Nugent, Philip Pridmore-Brown, Cory Riddell, and Dan Weston, for providing software and technical support. We also thank Bill Howe and David Hansen for helpful discussions, Philip Little for programming support, and the members of the CORIE group for their cooperation. This research is supported by NSF grant CCF-0121475.

References

[1] A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific workflow management by database management. *Proc. SSDBM*, 1998.

[2] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, et al. Data management and transfer in high-performance computational grid environments. *Parallel Computing Journal*, 28(5), 2002.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. Mock. Kepler: Towards a grid-enabled system for scientific workflows. *Proc. GGF10 Workshop on Workflow in Grid Systems*, 2004.

[4] A. Baptista, M. Wilkin, P. Pearson, P. Turner, and P. Barrett. Coastal and estuarine forecast systems: A multi-purpose infrastructure for the columbia river. *Earth System Monitor, NOAA*, 9(3), 1999.

[5] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. *Proc. IEEE Symposium on High Performance Distributed Computing (HPDC-13)*, 2004.

[6] I. Foster et al. Globus: A metacomputing infrastructure toolkit. *Int'l Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[7] I. Foster, J. Vockler, Michael Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *Proc. SSDBM*, 2002.

[8] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. *Proc. SSDBM*, 2001.

[9] B. Howe, K. Tanna, P. Turner, and D. Maier. Emergent semantics: Towards self-organizing scientific metadata. *Proc. Int'l Conf. on Semantics for a Networked World*, 2004.

[10] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. *Proc. Int'l Conf. on Distributed Computing Systems*, 1988.

[11] D. Liu and M. Franklin. Griddb: A data-centric overlay for scientific grids. *Proc. VLDB*, 2004.

[12] Data provenance/derivation workshop. *http://people.cs.uchicago.edu/ỹongzh/position_papers.html.*

[13] Thetus Publisher. *http://www.thetuscorp.com.*

[14] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. *Proc. ODBASE*, 2003.

[15] JOSH V1.1. *http://gridengine.sunsource.net/project/ gridengine/josh.html.*

[16] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. *Proc IEEE/ACM Int'l Conference on Cluster Computing and the Grid (CCGRID)*, 2001.

[17] Y. Zhang, A. Baptista, and E. Myers. A cross-scale model for 3d baroclinic circulation in estuary-plume-shelf systems: I. formulation and skill assessment. *Cont. Shelf Res.*, (accepted for publication).