Tim H. Wong
June 2006
Computer Science

Scheduling Data-Intensive Workflows

Abstract

Scientific computations are often modeled as *dataflow process networks* of tasks operating on a set of remote data. Scientists refer to this type of computation as a *scientific workflow*. One main challenge of executing a scientific workflow in a distributed environment (the *Grid*) is the scheduling of task executions and data transfers. In this paper, we formalize this scheduling problem as the *Data Shipping Problem (DSP)*. We consider two variations of DSP, the *Task Handling Problem (THP)* and the *Shipping and Handling Problem (SHP)*. We give show that both THP and SHP are NP-complete, and we introduce heuristic algorithms for the DSP problem. These algorithms have been implemented in an abstract-to-concrete workflow mapping tool. This tool takes in an *abstract workflow* designed in the *Kepler Scientific Workflow System* and *Graphviz*, and transforms it into an executable *concrete workflow* using one of our scheduling algorithms.

Professor Bertram Ludäscher
Dissertation Committee Chair

Scheduling Data-Intensive Workflows

By

TIM H. WONG
B.S. (University of California at Davis) 2004

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____

____

____

Committee in charge

2006

Scheduling Data-Intensive Workflows

Copyright 2006
by
Tim H. Wong

To my parents,

for all of their support and love.

Acknowledgments

First, I want to thank Professor Bertram Ludaescher and Professor Michael Gertz for their excellent guidances in my academic life. Second, I want to thank Professor Charles Martel for his valuable inputs on my research work. At last, I want to thank all my lab members for their support.

Contents

List of Figures vii

List of Tables ix

Abstract 1

1 Introduction 2

2 Background 6
2.1 Modeling Scientific Workflow . . . 8
2.2 Middleware for the Grid . . . 10
2.2.1 Schedulers . . . 10
2.2.2 Resource Brokers . . . 13
2.3 Pegasus Workflow Planner . . . 15

3 Problem Formalization 17
3.1 Definition of a Workflow . . . 18
3.2 Graph Structures . . . 20
3.2.1 Intrees . . . 20
3.2.2 Minimal Series Parallel Graph . . . 21
3.3 Data Shipping Problem (DSP) . . . 22
3.3.1 Cost Model . . . 22
3.3.2 Problem Statement . . . 22
3.3.3 Search Space . . . 23
3.3.4 Graph Coloring Problem . . . 24
3.4 Algorithms . . . 26
3.4.1 Brute Force Algorithm . . . 26
3.4.2 Greedy Algorithm . . . 27
3.4.3 Dynamic Programming Algorithm . . . 30
3.5 Task Handling Problem (THP) . . . 35
3.5.1 Cost Model . . . 35
3.5.2 Problem Statement . . . 35
3.5.3 NP-completeness . . . 36
3.6 Shipping and Handling Problem (SHP) . . . 36
3.6.1 Cost Model . . . 36
3.6.2 Problem Statement . . . 36

3.6.3 NP-completeness 37

4 Related Work **38**
4.1 Distributed Query Optimization 38
4.2 Task Scheduling Problem 40
4.2.1 Heuristics 42
4.3 Multiprocessor Scheduling Problem 46
4.3.1 Mapping between MP and THP 46
4.3.2 Heuristics 48
4.4 Pegasus: Task Scheduling Strategies 50

5 Implementation **52**
5.1 Distributed Computing Models 52
5.2 Objectives 53
5.3 Program Infrastructure 55
5.4 Building Kepler Workflows 57
5.5 Examples 58

6 Future Research Directions **64**
6.1 Theoretical Research 64
6.2 Extensions to our Implementation 65

7 Conclusions **67**
7.1 Summary of Findings 67
7.2 Summary of Implementations 69

Bibliography **71**

List of Figures

2.1 Different layers of a Scientific Workflow System (SWS) 7
2.2 A simple Kepler workflow 9
2.3 Condor Kernel according to [TTL05] 11
2.4 Program Flow of Stork and Condor [KL05] 12
2.5 Sample Usage of SRM [SSG02] 13
2.6 A Pegasus workflow [DBG+04] 15
2.7 Workflow Scheduling in Pegasus [DBG+04] 16

3.1 Taxonomy of Shipping and Handling Problems 17
3.2 DAG Representation of a Workflow 19
3.3 An Intree Graph 20
3.4 An Minimal Series Parallel Graph 21
3.5 Vertex Coloring of DSP 24
3.6 A workflow with multiple function outputs 32
3.7 A workflow with implicit relationships between functions 34

4.1 Two join trees with the same structure 40
4.2 Dynamic communication cost between tasks 46
4.3 Example of the MP problem 47
4.4 Workflow Graph corresponded to Figure 4.3 48
4.5 Pegasus Workflow Mapping [BJD+05] 50
4.6 Adding artificial node to handle data retrieval 51

5.1 Program Flow of the Abstract-to-Concrete Workflow Scheduler 54
5.2 Example for generating a concrete workflow from an abstract workflow specified in Graphviz 54
5.3 Example for generating a concrete workflow from an abstract workflow specified in Kepler 55
5.4 Class Hierarchy 56
5.5 Data Actor 57
5.6 Function Actor 58
5.7 Simple Intree Abstract Workflow 59
5.8 Simple Intree Concrete Workflow Scheduled by the Greedy Algorithm . . . 60
5.9 Complex Intree Abstract Workflow 61
5.10 Complex Intree Concrete Workflow Scheduled by the Dynamic Programming Algorithm 62

5.11 MSP Abstract Workflow 63
5.12 MSP Concrete Workflow 63

List of Tables

3.1 Data Labeling Information for Figure 3.3 29
3.2 Data Labeling Information for Figure 3.4 30
3.3 Dynamic Programming Cost and Labels for Figure 3.4 33
3.4 Dynamic Programming Cost and Labels for Figure 3.6 34

4.1 Component Mapping between MP and THP 47

Abstract

Scientific computations are often modeled as *dataflow process networks* of tasks operating on a set of remote data. Scientists refer to this type of computation as a *scientific workflow*. One main challenge of executing a scientific workflow in a distributed environment (the *Grid*) is the scheduling of task executions and data transfers. In this paper, we formalize this scheduling problem as the *Data Shipping Problem (DSP)*. We consider two variations of DSP, the *Task Handling Problem (THP)* and the *Shipping and Handling Problem (SHP)*. We give show that both THP and SHP are NP-complete, and we introduce heuristic algorithms for the DSP problem. These algorithms have been implemented in an abstract-to-concrete workflow mapping tool. This tool takes in an *abstract workflow* designed in the *Kepler Scientific Workflow System* and *Graphviz*, and transforms it into an executable *concrete workflow* using one of our scheduling algorithms.

Professor Bertram Ludäscher
Dissertation Committee Chair

Chapter 1

Introduction

Scientific computations are often modeled as *dataflow process networks* of tasks operating on a set of remote data. Scientists refer to this type of computation as a *scientific workflow*. Consider, for example, the UC Davis Geostreams project [HG05]. This project requires continuous data retrieval from the Geostationary Operational Environmental Satellite (GOES) satellite as well as continuous processing and storing of raster images on different disk drives. The Geostreams data management and analysis pipelines can be modeled as scientific workflows. Another example is the Lawrence Livermore National Laboratory (LLNL) Terascale Supernova Initiative project [LMM+02] [Xin04]. This project first requires shipping a large amount of data from a remote site to a supercomputer at National Energy Research Scientific Computing Center (NERSC) / Lawrence Berkeley National Laboratory (LBNL). After the data arrive at the destination, a local simulation begins, submitting jobs to a batch queue at LBNL. At the end of the simulation, the output data have to been shipped to another remote High Performance Storage System (HPSS) mass-storage server for permanent storage. Other examples include the terabyte-sized Fusion Plasma Simulation [BKA+04], the Montage astronomy application [BDG+03], the Grid Physics Network [Gri02], and the Particle Physics Data Grid (PPDG) [PPD02]. We call these types of scientific workflow, which require a significant data handling and movement overhead, *data-intensive* scientific workflows. The computations involved in these projects are too large for any single computer to handle. Therefore, scientists must break down

the computation into sub-tasks, and distribute them to a set of remote machines and storage servers for processing, and then coordinate the data transfers and function executions to obtain the final result. Scientists refer to this type of distributed and heterogeneous computing environment as the *Grid* [RMdL+03].

A major challenge of running scientific workflows on the Grid is the scheduling of tasks on available servers. Put simply, scientists want to finish the whole computation in the shortest time possible. Even though this sounds like a simple problem, the scheduling algorithm might have to take care of different network topologies, server speeds, available transfer protocols, data replicas, storage capacities, various task lengths, etc. For example, given an overly simple scientific workflow $y^c = f^b(x^a)$, where the input data x is located at server a, the function f is located at server b, and the output y has to been stored at server c, there are already three different possible execution plans:

$$x^{a \rightsquigarrow b};\ y^b{:=}f^b(x^b);\ y^{b \rightsquigarrow c} \tag{1.1}$$

$$f^{b \rightsquigarrow a};\ y^a{:=}f^a(x^a);\ y^{a \rightsquigarrow c} \tag{1.2}$$

$$x^{a \rightsquigarrow c};\ f^{b \rightsquigarrow c};\ y^c{:=}f^c(x^c) \tag{1.3}$$

Plan 1. In 1.1, the input data x is first shipped to server b. Then the function f processes x locally at server b, and then ships the output y to server c.

Plan 2. In 1.2, the function f is first transferred to server a. Then f processes x locally at server a, and then ships the output y to server c.

Plan 3. In 1.3, both the input data x and the function f are shipped to server c. Then f processes x locally at server c. The output file y is available at server c once the execution of f is completed.

The total time or the *makespan* [Pin95] required to execute each of these plans depends on the speed of host (processor) a, b, and c, the network bandwidth between them, and the size of x, y and f. The number of plans grows exponentially with the number of hosts and functions that are added to the computation. In this paper, we formalize this scheduling problem as the data *Shipping and Handling Problem (SHP)*. The *shipping* portion of SHP refers to the optimization of data transfers [1], and the *handling* portion of

[1]We do not consider the problem of function shipping here because (a) in practice, dynamically shipping

SHP refers to the scheduling of tasks. It follows that there are special cases of SHP:

Data-Intensive Case. In this case, the cost of data transfers dominates the cost of function executions. A typical cost model for this case would assume non-uniform transfer cost between servers, and zero function execution cost. We call this problem the *Data Shipping Problem (DSP).*

Computation-Intensive Case. In this case, the cost of function executions dominates the cost of data transfers. A typical cost model for this case would assume non-uniform function execution cost, and zero communication cost between servers. We call this problem the *Task Handling Problem (THP).*

Beside scheduling, there are also other challenging problems when executing scientific workflows on the Grid. One of them is workflow modeling and design. Scientists need a tool to dynamically connect well-defined functional components (sometimes called *actors*) to form a complete workflow. Moreover, such a tool must allow scientists to specify all required parameters for each function such that once the workflow execution begins, it can run to completion without further intervention. We refer to these completely specified and executable workflows as *concrete workflows.* On the other hand, not all scientists want to specify the low-level host settings for executing functions at remote servers. Therefore, a workflow modeling tool must be capable to transform incomplete workflow specifications, *abstract workflows* (i.e., leaving open execution hosts and data shipping instructions) into executable ones.

Another challenge of running scientific workflows on the Grid is the coordination of resources. Grid computing environments involve distributed and heterogeneous settings. Many servers have different processing power, security policies, storage spaces, and transfer protocols. From a usability point of view, scientists should not have to worry about all these different settings when they are modeling a workflow. Instead, a workflow scheduler and a workflow engine should be able to dynamically coordinate the interaction between these different servers.

The rest of this paper is organized as follows. Chapter 2 gives some background

---

and installing functions (application programs/codes) is complex, and thus, rarely done, and (b) it simplifies on already challenging SHP.

information on scientific workflow and introduces some popular software tools that scientists use to tackle the aforementioned problems. Chapter 3 first gives the formal definition of a scientific workflow and discusses the workflow representation model that we use. Then, it introduces example graph structures that lead to simplifications of scheduling problems. Next it gives the formalization of the *Shipping and Handling Problem (SHP)*, the *Task Handling Problem (THP)*, and the *Data Shipping Problem (DSP)*. We analyze the complexity of each problem and introduce optimal and heuristic algorithms. Chapter 4 discusses some related work. Chapter 5 discusses an abstract-to-concrete workflow mapping tool that has been implemented in Java. Chapter 6 discusses future work. Chapter 7 gives the conclusion of our findings.

Chapter 2

Background

This chapter introduces some popular software tools for designing, scheduling, and executing scientific workflows. A combination of these tools can be viewed as a *scientific workflow system (SWS)*, and often consists of four layers:

Workflow Modeling Layer. This layer contains software tools for scientists to design and model scientific workflows. These workflows can be simple flow-diagrams or dataflow process networks [PPL95], which only show the general flow of the computation and the general components that are needed. Flow-diagrams can then be modeled as abstract workflows, which show the exact requirements for each function component and the choice of the model of computation. Abstract workflows can be implemented as concrete workflows, which have all the parameters of each function component (actor) set, and are ready to be executed. A *high-level planner* is a software tool, which helps scientists to transform an abstract workflow into a concrete workflow. Once a concrete workflow is constructed, it can be submitted to a workflow manager or a workflow engine to execute.

Job Management Layer. This layer contains software tools for scientists to manage the execution of scientific workflows. These tools are often referred to as *workflow managers* or *workflow engines*. They are used to maintain the execution progress of the workflow, and coordinate the interaction between function executions and data transfers. To provide these functionalities, workflow managers must be able to support different transfer protocols, provide failure recovery, and communicate with different job schedulers and

resource brokers.

Grid Execution Layer. This layer contains software tools to schedule tasks and data transfers. These tools belong to two main categories: (a) *task schedulers*, and (b) *resource brokers*. A task scheduler allocates sub-tasks of a workflow to available resources. To determine which resource is appropriate or optimal for a task, the scheduler must consult various metadata catalogs to gather information about the task and the resource. A resource broker manages a set of data storage sources. These data sources can be disk drives, database systems, tape drives, or a combination of them. To provide seamless accesses to all these resources, the resource broker must support different transfer protocols and provide a logical namespace to access all files in a transparent manner.

Physical Server Layer. This layer contains software tools to manage the actual data and computation servers. These software tools provide supports to task schedulers and resource brokers. On computation servers, the software typically provides a job queue for task schedulers to submit their jobs and a messaging service to notify the task scheduler of any system failure. On data servers, the software provides a *disk cache* or a *sandbox* area for resource brokers to retrieve or store intermediate data. These disk caches and sandboxes must provide sufficient space for the incoming data.

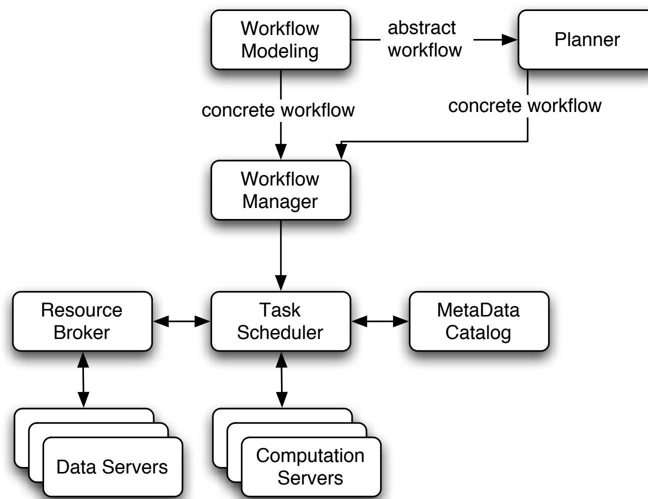

Figure 2.1: Different layers of a Scientific Workflow System (SWS)

The infrastructure of a Scientific Workflow System (SWS) is shown in Figure 2.1. In following sections, we are going to introduce some concepts and popular software tools used in each of these four layers.

2.1 Modeling Scientific Workflow

Flow-Based Programming. For scientists, a major advantage of modeling computation as a scientific workflow is the ability to automate complex tasks, to easily reconfigure workflows, and to reuse the same well-defined components to construct other workflows. This maximizes code-reuse, and improves efficiency in the design process. This idea of component-based programming is what Morrison [Mor96] refers to as *Flow-Based Programming (FBP)*. He described that a FBP system has the following major components: (a) a set of precoded and pretested functions with predefined plugs and sockets (we call these *actors*), (b) a central driver, which coordinates the interaction between different modules (we call this *director*), and (c) a notation for specifying how components are linked together. Using FBP, programmers can easily extend the system by adding/removing modules from the library, and redefine the semantics of the component interaction by just modifying the central driver.

Kepler. The KEPLER project [LAB+05] is a cross-institution collaboration to build a system for scientists to design and execute scientific workflows. The Kepler system is built on top of the Ptolemy II system [BL04]. The Ptolemy II project hosted at UC Berkeley focuses on the assembly and simulation of concurrent components. Its framework is a Java-based software program that includes a graphical user interface called Vergil. Concurrent components are implemented as *actors*, and the model of computation (MoC) is controlled by a central component called the *director*. A MoC defines the communication and execution semantics of a workflow. Using the Vergil interface, scientists can link various components together simply by dragging and dropping the corresponding icons on the canvas, and "wiring them up" to form the desired scientific workflow.

Kepler includes a set of actors for scientists to design and execute scientific work-

flows in a Grid environment. These actors are designed to run in dataflow-based computation models such as *Process Network (PN)* and *Synchronous Data Flow (SDF)*. The PN director is based on the Kahn Process Network model [Kah74], in which each actor is an individual process that can execute as long as enough input data are available. Actors communicate with each other through a set of one-way FIFO channels, which allow non-blocking write operations. In contrast, the SDF director [LM87] is a dataflow-based execution model, which schedules the execution of all actors in a topological order, taking into account token (data) consumption and production rates. The advantage of this approach is that the execution order is deterministic and can be statically determined prior to any function execution.

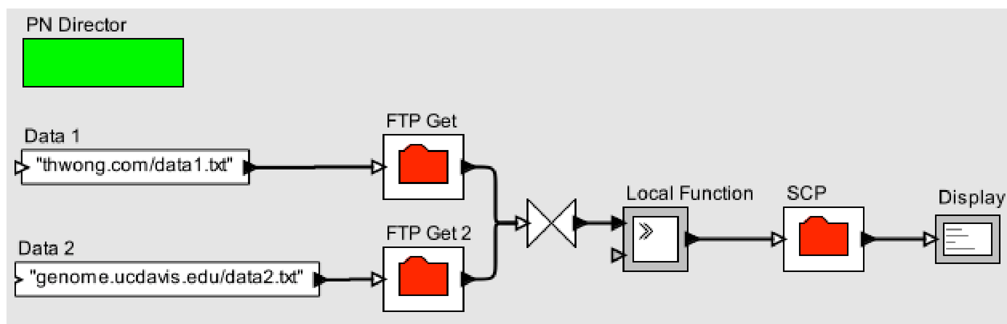

Figure 2.2: A simple Kepler workflow

Figure 2.2 shows a simple scientific workflow in Kepler. This workflow has three main steps, and its coordination is controlled by a PN director. First, two input data sets are concurrently fetched from remote sites using FTP actors. Second, these two input data are processed by a local function. Third, the output data are shipped to a remote host using a secure copy (SCP) actor. This simple example demonstrates a couple of important features of the Kepler system:

- Parameters can either be explicitly set using *String Constant* actors, or implicitly set within an actor's configuration (not shown in the figure). For example, the URLs of the FTP actors are explicitly given using two String Constant actors (Data 1 and Data 2 in Figure 2.2). In contrast, the URL of the SCP actor is implicitly set in its configuration, which can by accessed by double-clicking on the actor.

- Actors can easily be linked together by connecting their ports. In Ptolemy, connections are referred to as *relations*, which symbolize dataflows between actors.

- The MoC is solely controlled by the director, and is decoupled from the design of the workflow. This means that scientists can easily change the MoC by replacing the director. In our example, one can replace the PN director with a SDF director. In this case, the FTP transfers would be executed in a sequential order [1].

The concept of FBP and the Kepler system fit in workflow modeling layer of a SWS. Other popular workflow modeling tools include Chimera [FVWZ02], the Taverna workbench [Tav06], and the Triana workflow system [Tri06].

2.2 Middleware for the Grid

Grids are composed of a large number of distributed and heterogeneous servers. It would be a heavy burden for scientists to manage all these server settings when they are designing a scientific workflow. To cope with this problem, some Grid middleware tools are used to coordinate the interaction between servers and to provide seamless access to all resources.

This section presents some commonly used middleware for the Grid. These middleware are categorized into two groups: (a) schedulers, and (b) resource brokers. Schedulers such as Condor-G and Stork schedule task executions and data transfer of a concrete workflow. Resource brokers such as the Storage Resource Manager (SRM), Globus, and the Storage Resource Broker (SRB) provide seamless access to data sources. These software tools are associated with the job management, grid execution, and physical server layer of a SWS.

2.2.1 Schedulers

Condor. The Condor High-Throughput Computing system [TTL05] developed at the University of Wisconsin-Madison is a job and resource management system for compute-

[1] For a workflow to be executable in SDF, actors must declare a fixed token consumption and production rate. In the simplest (a common) case, an actor consumes and produces one token per port on each actor "firing".

intensive jobs. Its architecture is shown in Figure 2.3. Condor uses a workflow manager called *DAGMan* to manage concrete workflows submitted by users. DAGMan submits sub-tasks of a concrete workflow to different Condor *agents*, which interact with a centralized Condor *matchmaker* to find an appropriate *resource* for each task. The Condor matchmaker uses the "classified advertisement" information posted by Condor agents and Condor resources to find the appropriate matching. These classified advertisements are specified using a semi-structured language called *ClassAds*. Once a match is found, the resource will provide a *sandbox* area for the job, and a *shadow process* representing the user will start to execute. The combination of a matchmaker, a set of agents, and a set of resources forms a *Condor pool*. Using a Condor pool and its matchmaking technique, Condor provides functionalities such as job failure recovery, high resource utilization, and preemptive job scheduling.

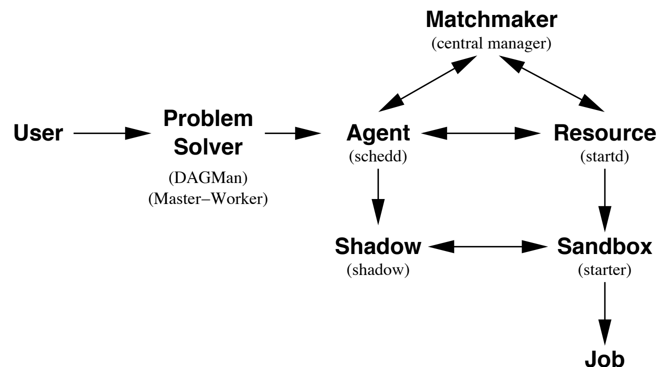

Figure 2.3: Condor Kernel according to [TTL05]

Condor-G. Condor-G is an extension of Condor, which allows users to access remote Condor pools. The interaction between these Condor pools is powered by the *Globus Grid Resource Access and Management (GRAM)* protocol. The main technology used in Condor-G is called *gliding in*, which dynamically provides an ad-hoc matchmaker for each Condor-G agent to access multiple Condor pools on-the-fly.

Stork. The Stork system [KL05] developed by the Condor team at the University of Wisconsin-Madison is a data placement system. It schedules all data transfer tasks of a concrete workflow such that data will be near or at the appropriate resources before the computation tasks begin to execute. Since Stork is intended to be used with the Condor system, researchers modify the internal control of the DAGMan workflow manager such that all computation tasks are forwarded to the Condor system, and all data placement tasks are forwarded to the Stork system. This flow is shown in Figure 2.4 [2]. Stork is able to perform third-party transfers, that is, users could transfer data from one remote site to another without first storing the data in a local cache. Moreover, Stork supports a large variety of transfer protocols including FTP, GridFTP, SRB, SRM, UniTree, etc. If a direct connection between any two hosts is not supported, an intermediate disk cache will be used to accomplish the transfer. Stork also provides other essential functionalities such as failure recovery and dynamic protocol selection.

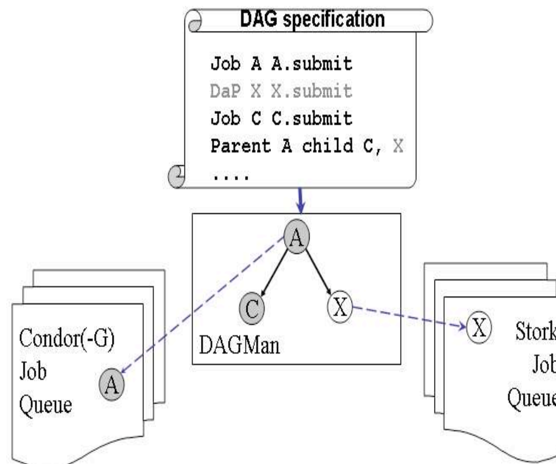

Figure 2.4: Program Flow of Stork and Condor [KL05]

[2]The DAGMan specification is only a *job dependency graph*. It does not show data shipment instructions between functions.

2.2.2 Resource Brokers

Storage Resource Manager. The Storage Resource Manager (SRM) [SSG02] developed at the Lawrence Berkeley National Laboratory (LBNL) is a data management system for large distributed datasets. SRM provides a set of resource managers to link heterogeneous datasets together. For example, a *Disk Resource Manager (DRM)* is used to manage data accesses to a disk drive or a RAID, and a *Tape Resource Manager (TRM)* or a *Hierarchical Storage Manager (HRM)* is used to manage data accesses to a tape system. When users want to transfer files between two different servers, the corresponding managers will choose the appropriate transfer protocol and handle failure recovery, deadlock, replica utilization, etc. Moreover, SRM also provides a metadata catalog to support mapping between physical and logical file names. When a logical file is requested, the Metadata Catalog translates the logical file name and retrieves the corresponding physical file from the server. This feature gives users a uniform logical repository to manage all physical files. Other important features of SRM include request queueing, disk caching, and data streaming.

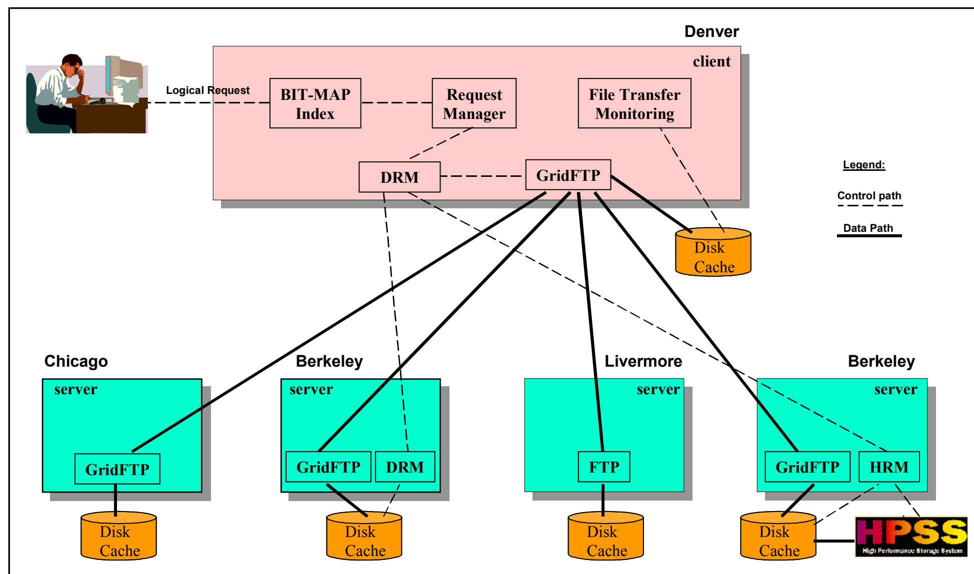

Figure 2.5: Sample Usage of SRM [SSG02]

Storage Resource Broker. The Storage Resource Broker (SRB) [RWM+03] developed at the San Diego Supercomputer Center (SDSC) is a data management system with a rich set of user interfaces and its own transfer protocol. Once SRB is installed on a server, it provides a logical namespace for any SRB client to access all registered physical files on that server. SRB differs from SRM in the way that it has its own transfer protocol to handle the interaction between SRB servers and clients. SRB also has a *Metadata Catalog (MCAT)* that gives users a single logical space to manage physical files on all SRB servers. A major feature of SRB is its rich set of user interfaces. Users or shell programs could use the *SCommand* interface to transfer, retrieve, replicate or delete files via unix-like commands such as *SPut*, *SGet*, etc. Java programs can make use of the *Jargon* API to access SRB functionalities. In terms of graphical user interfaces (GUI), SRB provides a web-portal called *MySRB* and a Windows program called *InQ* for clients to connect to SRB servers.

Styx. The Styx Grid Services System [BHH06] developed at the University of Reading, UK is a framework for wrapping command-line programs as services running on the Grid. Styx is not designed to be as powerful as other Grid middleware such as the Globus Toolkit. It is simply light-weight middleware that is easy to install and use. The key component of Styx is its protocol, which allows users to execute remote command-line programs just as they execute local programs. The same technique is also used to support third-party-transfer. For example, given two remote programs F1@ServerA and F2@ServerB, the intermediate file generated by $F1$ can be directly passed to $F2$ using the following commands:

1. SGSRun ServerA.com 9092 F1 input.dat -o intermediate.dat.sgsref

2. SGSRun ServerB.com 9092 F2 -i intermediate.data.sgsred -o output.dat

The SGSRun program is a general purpose command-line program for running Styx services, and the sgsref flag specifies a reference to the intermediate data generated by $F1$. Styx wraps command-line programs as services by specifying the program's input files, command-line parameters, and output files in a small XML file. Another advantage of Styx is its compatibility with firewalls. Styx does not require any port to be opened for its client service, and only one port to be opened for its server service. In the example above, port

9092 is opened.

2.3 Pegasus Workflow Planner

So far, we have introduced software tools for managing and executing concrete workflows on the Grid. However, if we have an abstract workflow, we need a workflow planner that can transform it into a concrete workflow by using metadata available in the workflow or in other metadata catalogs. These workflow planners are parts of the workflow modeling layer of a SWS.

Pegasus. The Pegasus system [DBG+04] developed at the University of Southern California Information Sciences Institute is a workflow planner for transforming abstract workflows into concrete workflows. Pegasus models abstract workflows as directed acyclic graphs (DAG) as shown in Figure 2.6. Each node represents a distinct task (i.e., T_1, T_2, ...), and each arc represents a dataflow and control-flow between two tasks. Input file names (i.e., F_1, F_2, ...) are shown as labels on arcs. When an abstract workflow is first submitted, Pegasus

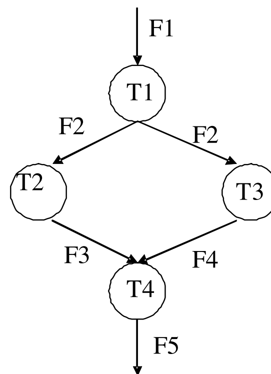

Figure 2.6: A Pegasus workflow [DBG+04]

consults the *Monitoring and Discovery Service (MDS)* and the *Pool Configuration File* to determine what resources are available. Then it consults the *Replica Location Service (RLS)* to determine if the workflow can be reduced by using existing (i.e., previously computed) intermediate data. Next, Pegasus assigns tasks to available resources by incorporating a selection algorithm. At the end, Pegasus determines if there are small tasks that can be

clustered and run on the same resource. Once the clustering is done, Pegasus adds a set of explicit transfer nodes to handle the data transfers, and then updates the RLS. The whole flow of the Pegasus system is shown in Figure 2.7.

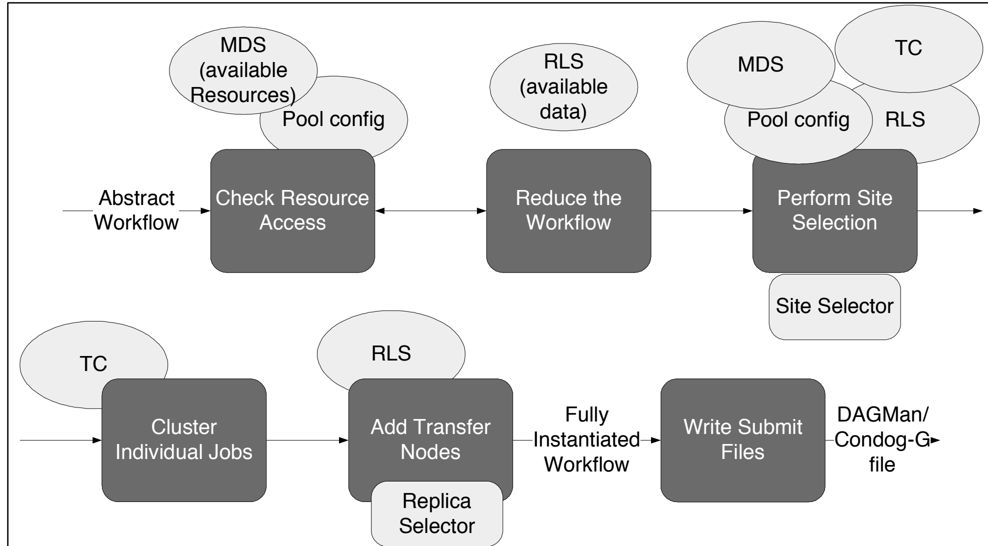

Figure 2.7: Workflow Scheduling in Pegasus [DBG+04]

From the set of possible execution plans, Pegasus picks an execution plan that has the shortest makespan using one of its heuristic scheduling algorithms: random, round-robin and min-min. In later chapters, we give a formalization of this workflow mapping problem, and introduce three heuristic algorithms that can produce better solutions using shorter time.

Chapter 3

Problem Formalization

This chapter first gives the formalization of the *Data Shipping Problem (DSP)*, and introduce two of its variations: the *Shipping and Handling Problem (SHP)*, and the *Task Handling Problem (THP)*. A taxonomy[1] of these problems is shown in Figure 3.1. We begin by giving the formal definition of a scientific workflow, and introduce some special graph structures that can simplify our scheduling problems. Next, we describe each of the following problems in order:

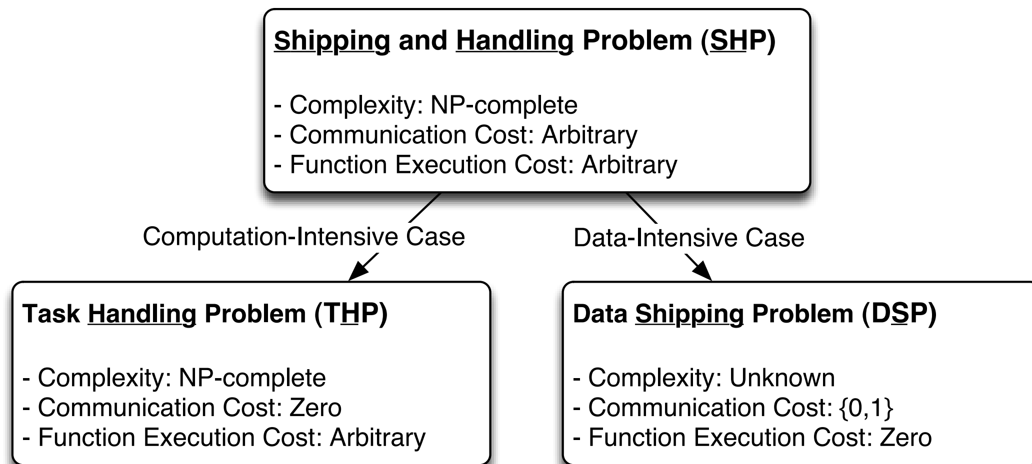

Figure 3.1: Taxonomy of Shipping and Handling Problems

[1]Both THP and DSP are special cases of SHP. While DSP focuses on the *data shipping* portion of SHP, THP focuses on the *task handling* portion of SHP

1. **Data Shipping Problem (DSP)**. The target machine of DSP assumes zero function execution cost and a unit communication cost for each data shipping between two different hosts. The objective of DSP is to find an execution plan P_W for the input abstract workflow W_A such that the total data shipping cost is minimal. The time complexity of finding an optimal solution for DAG input graph is still an open issue.

2. **Task Handling Problem (THP)**. The target machine of THP assumes zero communication cost, and arbitrary function execution cost. The objective of THP is to find an execution plan P_W for the input abstract workflow W_A such that the total function execution cost is minimal.

3. **Shipping and Handling Problem (SHP)**. The target machine of SHP assumes arbitrary function execution cost, and arbitrary communication cost based on the size of the transferred data. The objective of SHP is to find an execution plan P_W for the input abstract workflow W_A such that the sum of the total function execution and the total communication cost is minimal.

3.1 Definition of a Workflow

A workflow W is represented by a directed acyclic graph $G = (V, E)$, where $V = (\mathbf{D} \cup \mathbf{F})$, and E represents the precedence constraint. Having a directed edge from f_x to f_y means that f_y cannot start to execute until f_x is completed. Having a directed edge from d_x to f_y means that function f_y cannot start to execute until d_x is shipped to the appropriate execution host. The components are described as following:

1. A set of data $\mathbf{D} = \{d_1, \ldots, d_m\}$.

2. A set of functions $\mathbf{F} = \{f_1, \ldots, f_o\}$

3. A set of hosts $\mathbf{H} = \{h_1, \ldots, h_p\}$.

Workflow Labeling A *workflow labeling* $\mathbf{L}_W : \mathbf{V} \rightarrow \mathbf{H}$ is a mapping from the set of vertices to the set of hosts. It specifies the location of each data and function node. For

each workflow labeling, there is an associated *execution plan* P_W, which specifies all data shipping and function execution instructions.

Abstract Workflow. An *abstract workflow* W_A is a workflow that has a *data labeling* $\mathbf{L}_W^D : \mathbf{D} \rightarrow \mathbf{H}$, which specifies the location of all data nodes. However, the location of functions nodes is not specified, and has to be determined before execution.

Concrete Workflow. A *concrete workflow* W_C is a workflow that has both a *data labeling* $\mathbf{L}_W^D : \mathbf{D} \rightarrow \mathbf{H}$ and a *function labeling* $\mathbf{L}_W^F : \mathbf{D} \rightarrow \mathbf{H}$, which specifies the location of all data and function nodes. Note that $\mathbf{L}_W = \mathbf{L}_W^D \cup \mathbf{L}_W^F$; thus, each concrete workflow can be executed using its associated execution plan P_W.

Scheduling Problem. The scheduling problem that we consider is to transform an abstract workflow W_A into a concrete worfklow W_C by finding an *optimal execution plan* P'_W, which can complete the execution of the whole worklfow in the shortest time possible.

Workflow Example. Figure 3.2 shows a sample mapping between an abstract workflow and a concrete workflow. Data nodes are drawn as circles and function nodes are drawn as rectangles.

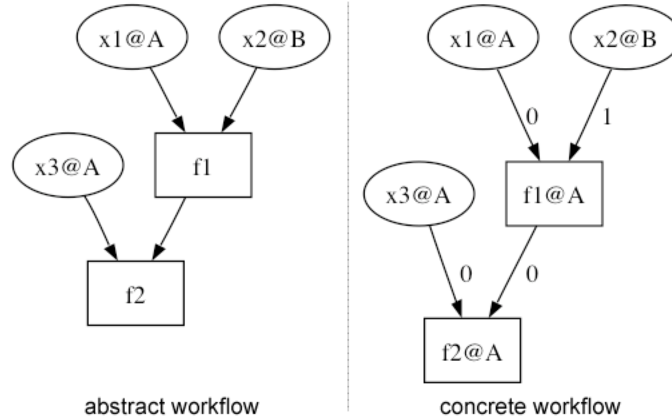

Figure 3.2: DAG Representation of a Workflow

3.2 Graph Structures

Many scheduling problems including ours are intractable in terms of time complexity if the input graph is a general DAG. This complexity is due to the constraint-free structure of DAG. In this section, we introduce two restricted graph structures: (a) *Intrees,* and (b) *Minimal Series Parallel (MSP)* graphs. One can show that the DSP can be solved in polynomial time on intrees and MSP (Section 3.4.2 - 3.4.3).

3.2.1 Intrees

Given a directed graph $G = (V, E)$, the in-degree D_v^- is the number of the incoming edges of a vertex v, and the out-degree D_v^+ is the number of outgoing edges of a vertex. An intree is a directed tree in which only one vertex has $D_v^+ = 0$. Figure 3.3 shows a workflow that has an intree structure. Intuitively, an intree corresponds to a conventional tree (an outree) by reversing the direction of each edge.

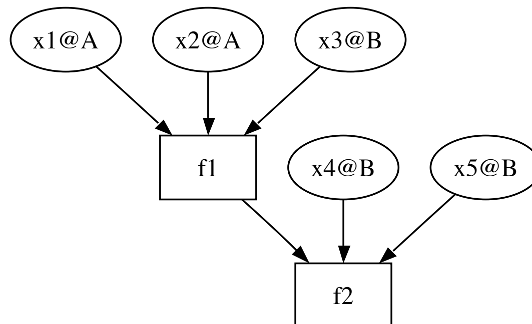

Figure 3.3: An Intree Graph

Solving DSP is easy on the intree structure because a local optimal host for a function can be determined in a greedy manner. Solving DSP is trivial on the outtree structure because an optimal solution is to assign each function to the same host that is assigned to the root function.

In practice, Intree is a common graph structure for scientific workflows that perform extensive data integration. These workflows usually start by querying a large number of data sources. Then the number of intermediate data storages reduces as the filtering

functions produce finer and smaller data. At the end, the final data is shipped to a single machine (i.e., the client machine) for storage and display.

3.2.2 Minimal Series Parallel Graph

A Minimal Series Parallel (MSP) graph [VTL79] is a directed graph that is composed by a set of recursively rules. In other words, every MSP graph can be decomposed into a set of MSP sub-graphs. These recursively rules are (a) *Parallel Composition Rule*, and (b) *Serial Composition Rule*.

1. **Parallel Composition Rule**. Given two MSP graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, a new MSP G_p can be formed by a parallel composition, where $G_p = (V_1 \cup V_2, E_1 \cup E_2)$.

2. **Serial Composition Rule**. Given two MSP graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, a new MSP G_s can be formed by a serial composition, where $G_s = (V_1 \cup V_2, E_1 \cup E_2 \cup (N_1 \times R_2))$, N_1 is the set of sinks of G_1, and R_2 is the set of sources of G_2.

Note that a directed graph with only one vertex and no edge is also a MSP graph. Figure 3.4 shows a workflow that has a MSP graph structure.

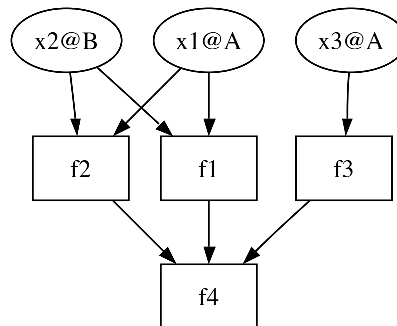

Figure 3.4: An Minimal Series Parallel Graph

Using our dynamic programming algorithm (Section 3.4.3), solving DSP is easy on the MSP structure because we can easily break down the original scheduling problem into sub-scheduling problems for MSP sub-graphs and then combine these local solutions to form the final optimal solution.

In practice, MSP graph is also an interesting graph structure for modeling scientific workflows. Many functions in a workflow are likely to share the same set of input data (represented by the parallel composition), and these data are ultimately merged and finally shipped to the final destination for display (represented by the serial composition rule).

3.3 Data Shipping Problem (DSP)

This section gives the formalization of the *Data Shipping Problem (DSP)*, which is a special case of SHP. The objective of this problem is to find an execution plan P_W for the input abstract workflow W_A such that the total data shipping cost is minimal. We assume that a data transfer between any two different hosts has a unit cost of one, and each function execution has a zero cost.

3.3.1 Cost Model

Given an input workflow graph $G = (V, E)$, which represents the input workflow W_A, the cost model of DSP has the following components:

1. $B_{x,y} = 1$ is a uniform bandwidth[2] between any two hosts h_x and h_y.
2. $C_{i,j} = 1$ is a unit transfer cost between vertex v_i and v_j no matter how large the data are.

T_C is the total communication time:

$$T_C = \sum_{i,j \in \mathbf{V}} C_{i,j} \mid (v_i, v_j) \in E \tag{3.1}$$

3.3.2 Problem Statement

Given an abstract workflow W_A and an initial labeling $\mathbf{L}_W^V$ as input, the *Data Shipping Problem* is to transform W_A into a concrete workflow W_C by finding an execution plan P_W defined by a function labeling $\mathbf{L}_W^F : \mathbf{F} \to \mathbf{H}$ such that the total data shipping cost [3] of P_W is minimal.

[2]This constraint states that only one data transfer can take place at any specific point in time. No parallel transfers can happen between the two same hosts.

[3]Since we do not consider parallel data transfers, T_C represents the *total shipping time*, which is differed from the *total response time* considered in other scheduling problems.

Decision Problem. Given an abstract workflow W_A, an initial labeling $\mathbf{L}_W^V$, and a time limit k, can one transform W_A into W_C by finding an execution plan P_W such that the total cost (the total time) of P_W is less than or equal to k?

DSP Graph Coloring Problem. One can view the scheduling problem DSP as an abstract graph coloring problem by considering:

1. The input workflow W_A as a DAG [4] graph $G = (V, E)$ [5].
2. The set of hosts $\mathbf{H}$ as a set of colors $C = \{c_0, \dots, c_n\}$.
3. The set of initial labeling $\mathbf{L}_W^V$ represents the set of initially colored vertices.
4. Each edge $e \in E$ has weight $w(e) = 1$ if its adjacent vertices have different colors (assigned to different hosts); otherwise, the edge has weight $w(e) = 0$.

Using this formulation, the objective is to find a vertex coloring L^C that maps each vertex $v \in V$ that is not initially colored to a color $c \in C$ such that the total edge weight $\sum_{e \in E} w(e)$ is minimal.

Given an optimal vertex coloring L^C, we can easily convert it into an equivalent function labeling $\mathbf{L}_W^F$ by mapping each color to its corresponding host. This function labeling then implies an optimal execution plan P for DSP.

3.3.3 Search Space

The *search space* for a given abstract workflow W_A is the set of all possible function labeling $\{\mathbf{L}_W^F\}$, and each function labeling represents a distinct execution plan P_W. Therefore, the following formula represents the total number of execution plans for a workflow W:

$$|Plans(W_A)| = |\mathbf{H}|^{|\mathbf{F}|} \tag{3.2}$$

[4] We consider directions on edges because our algorithms make the distinctions between incoming edges and outgoing edges.

[5] The graph G is essentially the same the graph G described in Section 3.1 except that this model does not make the distinction between function and data nodes. The set of E represents the precedence constraint between vertices

A typical distributed scientific workflow consists of a large number of function nodes (*actors*) and execution hosts. Therefore, finding an optimal solution by enumerating all plans is not feasible.

3.3.4 Graph Coloring Problem

As mentioned in Section 3.3.2, DSP can be represented as a graph coloring problem. However, its formulation is different from that of the traditional proper vertex coloring [GJ90], in which the objective is to use the minimum number of colors to color the whole graph such that no two adjacent vertices have the same color.

The initial labeling of the workflow graph could be used to determine the initial color assignment cost for each node. Consider Figure 3.5 as an example. The initial cost of assigning *Red* to f_1 would be one, because there is one blue node incident to f_1. Similarly, the initial cost of assigning *Blue* to f_1 would be three as there are three red nodes incident to f_1. The initial cost of assigning any color to node f_2 and f_3 would be zero, since the color of f_1 is not yet determined. These assignment costs will change accordingly as more and more nodes are colored.

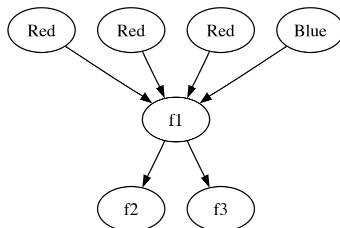

Figure 3.5: Vertex Coloring of DSP

There are two approaches to find an optimal color assignment for each node. The first approach is to assign the most common color to all nodes that do not have an initial color. Although this approach is very simple, it yields a near-optimal solution if the initial coloring of the graph is uniform, that is, many nodes initially have the same color.

The second approach is to determine an optimal color assignment for each node independently. The decision is based on the relationship between the current node and its

incident nodes, as well as its out degree. Let Δ_i be the difference between the lowest and second lowest assignment cost of node i. If Δ_i is bigger than or equal to its out degree D_i^+, then the color that has the minimal assignment cost c_i^{min} will be an optimal color for node i. If, on the other hand, Δ_i is less than D_i^+, then the color that has c_i^{min} may be a sub-optimal color for node i.

The intuition behind this is that Δ_i denotes the *saving* that we get by assigning c_i^{min} to i, that is, the number of incoming data transfers that we save. D_i^+ denotes the minimum*penalty* that we get if c_i^{min} is not an optimal color for i. Therefore, if $\Delta_i \geq D_i^+$, there is no *penalty* for assigning c_i^{min} to i.

Consider Figure 3.5 as an example, we have $Cost_{Red}(f_1) = 1$, $Cost_{Blue}(f_1) = 3$, $\Delta_{f_i} = 2$, and $D_{f_i}^+ = 2$. Therefore, *Red* is guaranteed to be an optimal color for f_1. This basic concept is very similar to that of the greedy algorithm. We are looking for the host that contains the most input data initially.

Both approaches suggest that assigning the same color to as many nodes as possible would always yield a better solution. That is, we should allocate as many functions as possible to a single host. This tendency is due to the zero function execution cost in our model. If we add back the function cost factor or add a limit on how many functions can be executed at a host simultaneously, then this monochromatic tendency will no longer exist. However, this will turn DSP into SHP, which is an NP-complete problem.

The cost difference Δ_i can also be used to describe the worst case *penalty* (the avoidable data transfers) of the greedy algorithm. A color assignment for a node i may yield a penalty when $\Delta_i < D_i^+$. Therefore, the maximum penalty of an algorithm is bounded by the following formula:

$$\sum_{i \in \mathbf{F}} p_i, \text{with } p_i = \begin{cases} D_i^+ - \Delta_i & \text{if } D_i^+ > \Delta_i; \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

This formula shows that the worst case *penalty* is basically the sum of the potential penalty yield by each color assignment.

3.4 Algorithms

In this section, we introduce three algorithms that we have developed for the DSP graph coloring problem. These algorithms take in a DAG graph $G = (V, E)$ representing the workflow and produce a vertex coloring L_C representing a plan for DSP.

1. **Brute Force Algorithm.** This algorithm finds an optimal solution. However, this algorithm has an exponential runtime, and thus, is not practical when the workflow consists of a large number of functions and hosts.

2. **Greedy Algorithm.** This algorithm finds a sub-optimal solution in a greedy manner, and has a polynomial runtime. This can also find an optimal solution if the input graph is an intree with special constraints.

3. **Dynamic Programming Algorithm.** This algorithm finds a sub-optimal solution in a breadth first search manner, and has a polynomial runtime. This can find an optimal solution if the input graph is an intree or a MSP.

3.4.1 Brute Force Algorithm

This brute force algorithm[6] performs a complete enumeration on all possible execution plans of a given workflow, and then calculate the total communication cost T_C of each plan. An optimal execution plan is simply a plan that has the lowest cost.

Brute Force Algorithm (BF-DSP)

Input: $G = (V, E)$

/* $\mathbf{G_C}$ contains a set of fully colored graphs */

$\mathbf{G_C} \leftarrow$ The set of all possible combinations of vertex coloring of G

min_cost $\leftarrow |E|$

$L^* \leftarrow \emptyset$ /* L^* is an optimal vertex coloring */

/* Going through each graph in $\mathbf{G_C}$ to find an optimal vertex coloring */

[6] In the abstract-to-concrete workflow transformation tool that we describe in Chapter 5, this algorithm is implemented as BruteForce.class.

for each $G' = (V', E') \in \mathbf{G_C}$

cost $\leftarrow 0$

for each vertex $v \in V$

/* For each vertex, checks each of its parent vertices */

for each $v' \in V \mid (v, v') \in D_v^-$ /* D_v^- is the set of incoming edges of v*/

if color(v) $\neq$ color(v') **then**

cost $\leftarrow$ cost + 1 /* since these two hosts differ, a data transfer is required */

end if

end for

end for

if cost $<$ min_cost **then**

min_cost $\leftarrow$ cost

$L^* \leftarrow L^C_{G'}$ /* $L^C_{G'}$ = vertex coloring of G' */

end if

end for

Given a workflow, there is a total of $|\mathbf{H}|^{|\mathbf{F}|}$ plans ($|\mathbf{F}|$ is the number of nodes that are initially uncolored). For each plan, the brute force algorithm needs to perform $|\mathbf{F}|$ computations. Therefore, the time complexity of the brute force algorithm is $O(|\mathbf{F}| * (|\mathbf{H}|^{|\mathbf{F}|}))$. This brute force algorithm finds an optimal solution for any input graph structure. However, its exponential runtime makes it impractical when the workflow consists of a large number of functions and hosts.

3.4.2 Greedy Algorithm

This greedy algorithm[7] finds a sub-optimal solution for workflow that has the general DAG structure. It finds a local optimal host for each function in a greedy manner.

[7] In the abstract-to-concrete workflow transformation tool that we describe in Chapter 5, this algorithm is implemented as GreedyAlg.class.

Greedy Algorithm (GA-DSP)

Input: $G = (V, E)$

/* assume that $\{v_0, \ldots, v_n\}$ is a topological sort of V based on its precedence constraint*/

for each vertex $v \in \{v_0, \ldots, v_n\}$

$max \leftarrow 0$

for each pair of vertex and color $(v,\ c)$

/* $count_{v,c}$ is the number of parent vertices of v that have the same color as v */

/* D_v^- is the set of incoming edges of v */

$count_{v,c} \leftarrow |V'|$ with $v' \in V' \mid (v', v) \in D_v^-$ and $color(v') = color(v)$

if $count_{v,c} >$ max **then**

$color(v) \leftarrow c$

$max \leftarrow count_{v,c}$

end if

end for

end for

The time complexity of the greedy algorithm is $O(|E|)$. This greedy algorithm finds a sub-optimal plan for general DAG or MSP graph. However, it can find an optimal solution if the input graph is an Intree, and there exists exactly one local optimal host for each function, that is only one host has the *max* value for each function.

Proposition 3.4.1 *A local optimal host for a function node f is a host that initially contains most of its input data.*

Proof To locally minimize the number of data transfers required for each function f, one needs to find a "maximal input host" $MIH(f)$. Let $CNT(f, h)$ be the number (the count) of input nodes of f that are assigned to h, then the $MIH(f)$ can be described as:

$$MIH(f) = \{h \mid CNT(f, h) = max(CNT(f, h') \mid h' \in \mathbf{H}\} \tag{3.4}$$

Let $ID(f)$ be the in-degree of f,the local minimum cost for function f can be described as:

$$MIN_COST(f) = ID(f) - MIH(f) \tag{3.5}$$

It is clear that all the input data of f must be first transfered to a single host before running f. Therefore, the number of data transfers is minimal if f is assigned to a host h which initially contains the most input data. ∎

There are cases in which there is more than one local optimal host for a function f, i.e, a workflow with MSP graph structure. In this case, one of the optimal hosts may yield a lower cost for the workflow based on the local optimal hosts of other functions. The greedy algorithm would fail to take into account this type of dependence, as it only focuses on the input data of a single function.

Greedy Algorithm Examples. Consider Figure 3.3 (page 20) as an example. An optimal function labeling would be $\mathbf{L}_W^F = \{(f_1, A), (f_2, B)\}$ in which the total communication cost T_C of the associated execution plan P_W is two. Table 3.1 shows the data labeling information. The data labeling of f_2 is a range value [min..max] because the location of the input data depends on the label of f_1.

Function	A	B
f_1	2	1
f_2	0..1	2..3

Table 3.1: Data Labeling Information for Figure 3.3

The greedy algorithm produces an optimal function labeling for this workflow, since there is only one local optimal host for each function.

Now consider Figure 3.4 (page 21) as an example. This input graph has a MSP structure. An optimal plan would be $\mathbf{L}_W^F = \{(f_1, A), (f_2, A), (f_3, A), (f_4, A)\}$, in which the total communication cost T_C of the associated execution plan P_W is two. Table 3.2 shows the data labeling information. Similar to the previous example, the data labeling of f_4 is a range value [min..max] because the location of the input data location depends on the function labeling of f_1, f_2, and f_3.

For function f_1 and f_2, the greedy algorithm would choose either host A or host B as the optimal host since they initially contain the same number of input data. However, it is clear that assigning both f_1 and f_2 to host A would yield a lower communication cost

Function	A	B
f_1	1	1
f_2	1	1
f_3	1	0
f_4	0..3	0..3

Table 3.2: Data Labeling Information for Figure 3.4

since host A is the only optimal host for function f_3. This example shows how the greedy algorithm fails to produce an optimal function labeling for a workflow that contains more than one optimal host for a function.

3.4.3 Dynamic Programming Algorithm

This dynamic programming algorithm[8] finds a local optimal host for each function by computing the label cost of each function at each host. The label cost $label_cost(f, h)$ is computed as follows: (a) the algorithm iterates through the set of input data of f, (b) if the input data is a data node and is located at a host different from h, the corresponding communication cost is added to the label cost, (c) if the input data is a function node, the algorithm checks if it is cheaper to execute both function f and its parent function at h, or run this parent function at its optimal host and then transfer the output data to h.

Dynamic Programming Algorithm (DPA-DSP)

Input: $G = (V, E)$

for each vertex v in V

/* L_v^* is an optimal vertex coloring for vertex v and all of its precedent vertices */

/* M_v is the total edge weight of L_v^* */

$L_v^* \leftarrow \emptyset$

$M_v^* \leftarrow |D_v|$

/* for each vertex v, we calculate the cost of mapping v to c */

for each color $c \in C$

[8] In the abstract-to-concrete workflow transformation tool that we describe in Chapter 5, this algorithm is implemented as DynProgAlg.class.

```
/* L_v^c is a vertex coloring for using color c with vertex v and its precedent vertices*/
/* M_v^c is the total edge weight of L_v^C */
L_v^c ← ∅
M_v^c ← 0
/* checking each parent vertex v' of v */
for each v' | (v', v) ∈ D_v^-
    /* if v' is an initially colored node (a data node in DSP)
     and is different from the current color c, then increment M_v^C */
    if v' is initally colored and color(v') ≠ c then
        M_v^c ← M_v^c + 1
    /* if v' is not initially colored, (a function node in DSP)
     then check which color to use for both v' and v */
    else if v' is not initially colored then
        /* if this is cheaper to use the color c for both v' and v */
        if M_{v'}^c < M_{v'}^* then
            M_v^c ← M_v^c + M_{v'}^c
            L_v^c ← L_v^c ∪ L_{v'}^c ∪ {(v,c)}
        /* if this is cheaper to keep the color for v' and just use color c for v */
        else
            M_v^c ← M_v^c + M_{v'}^* + 1
            L_v^c ← L_v^c ∪ L_{v'}^* ∪ {(v,c)}
        end if
    end if
end for
if M_v^c < M_v^* then
    M_v^* = M_v^c
    L_v^* = L_v^c
end if
end for
```

end for

The time complexity of the dynamic programming algorithm is $O(|\mathbf{F}| * |\mathbf{H}|)$. A local optimal label for each function is determined by computing the label cost of each function at each host. Let $\mathbf{F}_f^{in}$ be the set of parent functions for f and s be the number of data that are not located at h, the cost formula of assigning f to h is described as follows:

$$cost(f,h) = s + \sum_{i \in \mathbf{F}_f^{in}} min(cost(i,h), cost(i,*) + 1) \tag{3.6}$$

and the minimal label cost of f is descried as:

$$cost(f,*) = \{min_cost(f,h) \mid h \in \mathbf{H}\} \tag{3.7}$$

This dynamic programming algorithm finds an optimal solution for input graphs that have Intree or MSP structure. However, it only finds a sub-optimal solution for DAG input graphs, since it may fail to recognize the implicit dependency between *neighbor functions*, which are functions that have the same number of precedent functions. An example is illustrated in Figure 3.6

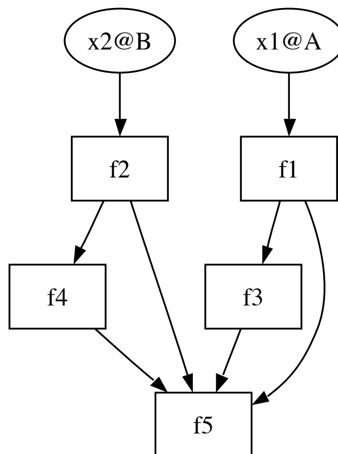

Figure 3.6: A workflow with multiple function outputs

Dynamic Programming Algorithm Example. Consider Figure 3.4 (page 21) as an example. An optimal function labeling is $\mathbf{L}_W^F = \{(f_1, A), (f_2, A), (f_3, A), (f_4, A)\}$, in which

the total communication cost T_C of its associated execution plan P_W is two. Table 3.3 shows the label cost and the associated function labeling for each function at each host.

Cost	Labels
$cost(f_1, A) = 1$	$\{(f_1, A)\}$
$cost(f_1, B) = 1$	$\{(f_1, B)\}$
$cost(f_1, *) = 1$	$\{(f_1, A)\}$
$cost(f_2, A) = 1$	$\{(f_2, A)\}$
$cost(f_2, B) = 1$	$\{(f_2, B)\}$
$cost(f_2, *) = 1$	$\{(f_2, A)\}$
$cost(f_3, A) = 0$	$\{(f_3, A)\}$
$cost(f_3, B) = 1$	$\{(f_3, B)\}$
$cost(f_3, *) = 1$	$\{(f_3, A)\}$
$cost(f_4, A) = 2$	$\{(f_1, A), (f_2, A), (f_3, A), (f_4, A)\}$
$cost(f_4, B) = 3$	$\{(f_1, B), (f_2, B), (f_3, B), (f_4, B)\}$
$cost(f_4, *) = 2$	$\{(f_1, A), (f_2, A), (f_3, A), (f_4, A)\}$

Table 3.3: Dynamic Programming Cost and Labels for Figure 3.4

It is clear that the dynamic programming algorithm produces an optimal function labeling (represented by $cost(f_4, *)$). Now, consider Figure 3.6 (page 29) as an example. An optimal function labeling would be $\mathbf{L}_W^F = \{(f_1, A), (f_2, A), (f_3, A), (f_4, A)\}$, which has total communication cost $T_C = 1$.

The dynamic programming algorithm fails to find an optimal function labeling in this case. By assigning f_1 and f_2 to the same host, we could eliminate the data transfers from f_3 and f_4. However, since the dynamic programming algorithm only considers the label cost of the current function and its precedent functions, it does not recognize the implicit dependency between function f_1 and f_2, and thus, it fails to find an optimal function labeling.

One may apply a tweak to the dynamic programming algorithm by forcing the algorithm to use the same label for both the current function and its parent function if $cost(f, L) = cost(f, *) + 1$. This tweak would work for the example in Figure 3.6 just merely because the communication cost in our model happens to be one. It will not work for a more complicated case like the one in Figure 3.7, which represents a typical parallel data processing procedure in scientific workflows.

Cost	Labels
$cost(f_1, A) = 0$	$\{(f_1, A)\}$
$cost(f_1, B) = 1$	$\{(f_1, B)\}$
$cost(f_1, *) = 0$	$\{(f_1, A)\}$
$cost(f_2, A) = 1$	$\{(f_2, A)\}$
$cost(f_2, B) = 0$	$\{(f_2, B)\}$
$cost(f_2, *) = 0$	$\{(f_2, A)\}$
$cost(f_3, A) = 0$	$\{(f_1, A), (f_3, A)\}$
$cost(f_3, B) = 1$	$\{(f_1, A), (f_3, B)\}$
$cost(f_3, *) = 0$	$\{(f_1, A), (f_3, A)\}$
$cost(f_4, A) = 1$	$\{(f_2, B), (f_4, A)\}$
$cost(f_4, B) = 0$	$\{(f_2, B), (f_4, B)\}$
$cost(f_4, *) = 0$	$\{(f_2, B), (f_4, B)\}$
$cost(f_5, A) = 2$	$\{(f_1, A), (f_2, A), (f_3, B), (f_4, B), (f_5, A)\}$
$cost(f_5, B) = 2$	$\{(f_1, A), (f_2, A), (f_3, B), (f_4, B), (f_5, B)\}$
$cost(f_5, *) = 2$	$\{(f_1, A), (f_2, A), (f_3, B), (f_4, B), (f_5, A)\}$

Table 3.4: Dynamic Programming Cost and Labels for Figure 3.6

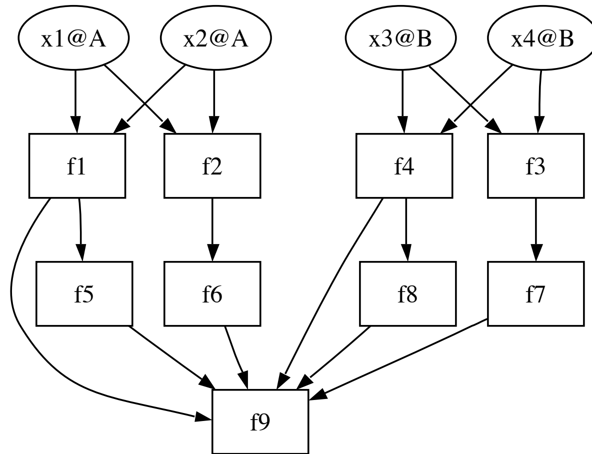

Figure 3.7: A workflow with implicit relationships between functions

3.5 Task Handling Problem (THP)

This section introduces the *Task Handling Problem (THP)*, which is a variation of DSP. Instead of minimizing the total number of data transfers, the objective of THP is to minimize the total execution cost of all functions assuming that the communication cost is zero. It is clear that the cost model of THP is differed from DSP in the sense that it must consider the processor speed of each host, the computing requirement of each function, and the dependency between function nodes, etc.

3.5.1 Cost Model

Given an input workflow graph $G = (V, E)$ with $V = (D \cup F)$ and E represents the precedence constraint, the cost model of THP has the following components:

1. D_f is the task length of function node f.
2. S_h is the processor speed of host h.
3. SC_f is the setup cost of function node h.
4. $B_{i,j}$ is the bandwidth between host h_i and h_j.

3.5.2 Problem Statement

Given an abstract workflow W_A, and an initial labeling $\mathbf{L}_W^V$ as input, the *Task Handling Problem* is to find an execution plan P_W defined by a function labeling $\mathbf{L}_W^F : \mathbf{F} \rightarrow (\mathbf{H}, t)$, which maps each function f to a host h at a specific time $t \in \{0, \ldots, k-1\}$ such that the total execution cost of P_W is minimal, and the execution order of all functions obeys the precedence constraint specified in W_A.

Decision Problem. Given an abstract workflow W_A, an initial labeling $\mathbf{L}_W^V$, and a time limit k, can one transform W_A into W_C by finding an execution plan P_W such that the total execution time of all functions is less than or equal to k, and the execution order of all functions obeys the precedence constraint specified in W_A?

3.5.3 NP-completeness

By creating a mapping between THP and the multiprocessor scheduling problem (MP) as shown in Chapter 4.3.1, one can show that MP is a subproblem of THP, and one can verify the correctness of an optimal solution for THP simply by checking if the completion time of each function is less or equal to k. It follows that THP is a NP-complete problem.

3.6 Shipping and Handling Problem (SHP)

This section introduces the *Shipping and Handling Problem (SHP)*, which is a variation of DSP. In SHP, the objective is to find an execution plan that minimizes both the total number of data transfers, and the total execution time of all functions. Therefore, an input workflow of SHP is considered to be both Data-Intensive and Computation-Intensive.

3.6.1 Cost Model

Given an input workflow graph $G = (V, E)$ with $V = (D \cup F)$ and E represents the precedence constraint, the cost model of THP has the following components:

1. D_f is the task length of function node f.

2. S_h is the processor speed of host h.

3. SC_f is the setup cost of function node h.

4. $B_{i,j}$ is the bandwidth between host h_i and h_j.

5. $C_{i,j}$is the communication cost between host i and host j.

3.6.2 Problem Statement

Given an abstract workflow W_A and an initial labeling $\mathbf{L}_W^V$ as input, the *Shipping and Handling Problem* is to find an execution plan P_W defined by a function labeling $\mathbf{L}_W^F : \mathbf{F} \rightarrow (\mathbf{H}, t)$ such that the total cost (the sum of the total function execution cost and

the total communication cost) of P_W is minimal, and the execution order of all functions obeys the precedence constraint specified in W_A.

Decision Problem. Given an abstract workflow W_A, an initial labeling $\mathbf{L}_W^V$, and a time limit k, can one transform W_A into W_C by finding an execution plan P_W such that the sum of total execution cost of all functions and the total communication cost is less than or equal to k, and the execution order of all functions obeys the precedence constraint specified in W_A?

3.6.3 NP-completeness

Since THP is a subproblem of SHP, and the correctness of an optimal execution plan for SHP can be verified in polynomial time, one can easily show that SHP is also a NP-complete problem.

Chapter 4

Related Work

In this chapter, we describe some well known problems that are related to the scheduling problems that are aforementioned. They are (a) distributed query optimization, (b) task scheduling, (c) multiprocessor scheduling, and (d) workflow mapping in Pegasus. The objective of distributed query optimization is to minimize the total data transfer time, which is similar to the objective of the DSP. The objective of task scheduling is to minimize both the total function execution cost and the total communication cost, which is similar to the objective of SHP. The objective of multiprocessor scheduling is to minimize the total execution time of all functions, which is similar to the objective of THP. The objective of the Pegasus system is to transform abstract workflows into concrete workflows, which is similar to the objective of SHP.

4.1 Distributed Query Optimization

Distributed query optimization refers to the process of finding a distributed query execution plan, which gives the shortest response time. The total time consists of communication time, I/O time, and CPU time. However, since the communication time is usually the dominant factor, many cost models totally ignore the I/O time and the CPU time. Moreover, many models only focus on join operations since they have the most important effect on query execution performance.

Search Space. The search space for a given distributed query is the set of all equivalent query execution plans produced by applying the commutativity and associativity rules of the join operations. Each execution plan is defined by a unique join tree. Given N relations, the number of query execution plans is $O(N!)$. Some models, however, consider only linear ordered join trees, which reduce the number of plans to $O(2^n)$. Kiyoshi Ono et al. [OL90] proved that finding an optimal execution plan for a distributed query is NP-complete.

Heuristic Algorithms. Some popular heuristic algorithms for distributed query optimization include distributed INGRES [WV90], R^* [SAC$^+$79], and SDD-1 [BGW$^+$81]. The distributed INGRES algorithm is implemented in the INGRES database management system. It uses a dynamic programming approach to find an optimal execution plan in a breadth first search manner. The R^* algorithm is implemented in the System R database management. It uses a greedy algorithm approach to find an optimal execution plan in a depth first search manner. Both Distributed INGRES and R^* make use of full join in every optimization step. The SDD-1 algorithm, in contrast, uses semijoins to reduce the size of intermediate query result and finds an optimal plan via a sequence of recursive steps based on the hill-climbing algorithm [Won77]. Studies [ES80] showed that the Distributed INGRES algorithm produces the most accurate result, but it also has the longest runtime. The semijoin strategy employed by SDD-1 requires additional selection operations, and thus, increase the workload of the processors. This means that SDD-1 would yield better performance than other algorithms only if the communication cost of the model dominates the execution cost of the join operations.

Similarities. Distributed query optimization is similar to DSP in the sense that its objective is to minimize the total communication time T_C. In fact, the structure of a join tree can be viewed as a workflow consisting of only join operations. For example, the two join trees shown in Figure 4.1 have the same structure, but the joins are executed at different locations. Thus, by considering the join tree structure as a workflow and the databases as input data $\mathbf{X}$, the objective of the distributed query optimization can be viewed as finding a function labeling $\mathbf{L}_j^f$ for each join tree structure j. Moreover, the goal of distributed

query optimization is often choosing operations, which reduce data size. Thus, its heuristic algorithms are suitable for some data-intensive scientific workflows.

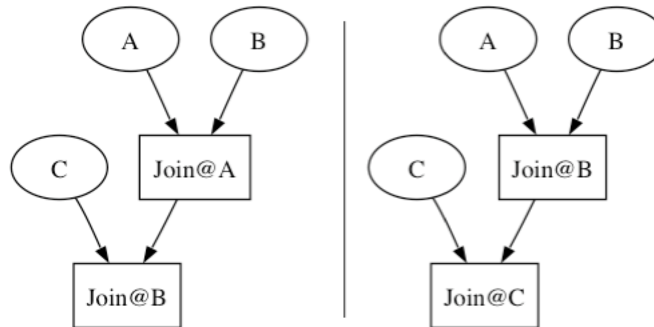

Figure 4.1: Two join trees with the same structure

Differences. Distributed query optimization, on the other hand, differs from DSP in the sense that it mainly focuses on join operations. In DSP, all functions may act differently on the input data, and may have different computing constraints. Beside, all heuristic algorithms for distributed query optimization assume that there are statistics available for the data or (at least for the relations, as in distributed INGRES). In DSP, we could not assume that we have statistics available for all input data. Even if we have these data statistics, they may not help our functions at all.

4.2 Task Scheduling Problem

Task Scheduling refers to the process of finding an assignment of tasks to a set of processors such that the total function execution time is minimal. Hesham El-Rewini et al. [ERLA94] describe the general form of the task scheduling problem as follows. Given a set of n tasks T, a partial order $<$ on T, a weight A_i for each task, a set of m processors, and a time limit k, is there a total function h that maps T to $\{0, 1, \ldots, k-1\}$ such that:

1. if $i < j$, then $h(i) + A_i \leq h(j)$, (the start time of $h(j)$ must be bigger than or equal to the finish time of $h(i)$)

2. for each i in T, $h(i) + A_i \leq k$, (the finish time of each task must be less than or equal to the time limit k)

3. for each t, $0 \leq t \leq k$, there are at most m values of i for which $h(i) \leq t \leq h(i) + A(i)$, (there are at most m tasks running at anytime)

The target machine for this task scheduling problem consists of the following components:

1. P is the set of m processors that are fully connected

2. S_i specifies the speed of the processor p_i

3. I_i specifies the startup cost of initiating a message on processor p_i

4. B_i specifies the startup cost of initiating a process on processor p_i

5. $R_{i,j}$ is the transmission rate between processors p_i and p_j

Search space. The search space of the task scheduling problem is the set of all equivalent function assignments that have a total execution time less than or equal to k. Richard M. Karp [Kar72] proved that this general form of task scheduling problem is NP-complete. Later on, researchers also proved that many restricted models of the task scheduling problem are also NP-complete. These include the *Two Processors, one or two time-units scheduling* model proved by Jeffrey D. Ullman [Ull75], the *Two Processor, interval-order scheduling* model proved by Christos H. Papadimitriou et al. [PY79], and the *Single Execution Time, Opposing Forests* model proved by Michael R. Garey et al. [GJTY83].

Special cases. Although the task scheduling problem is NP-complete in general, Richard M. Karp [ERLA94] described three special cases in which an optimal solution can be found in polynomial time. These cases are: (a) the task graph is a tree where Hu [Hu61] found a linear line algorithm, (b) the task graph is interval-ordered where Papadimitriou et al. [PY79] found a linear time algorithm, and (c) the target machine only contains two processors where Fuji et al. [FKN69] found a polynomial time algorithm. Kwong Kwok et al. [KA99] developed a taxonomy for all these cases and described a large number of heuristic algorithms.

4.2.1 Heuristics

Blythe [BJD+05] introduced two different heuristics to tackle the task scheduling problem: (a) task-based approach (TBA) and (b) workflow-based approach (WBA). TBA iteratively allocates all available jobs in a greedy fashion until all jobs are scheduled. WBA, in contrast, maps a priority to each task, and finds an overall optimal schedule via a pre-defined number of iterations. The cost model used for these heuristics contains the following components:

1. $EET(j, r)$ is the estimated execution time of job j on resource (host, processor) r
2. $EAT(j, r)$ is the estimated time it takes for resource r to be available to execute job j
3. $FAT(j, r)$ is the estimated time it takes to transfer all input files of job j to resource r
4. $ECT(j, r)$ is the estimated time it takes to complete job j on resource r

The TBA approach is based on the min-min heuristic [BSB01]. This heuristics is first developed to handle task scheduling of independent tasks. Blythe applies this algorithm to workflow-based DAG by considering all available tasks independently. A task is available when all of its precedent functions are executed. For example, in Figure 4.5, $J1$ is the only available job at the beginning. Once $J1$ is executed, $J2$ and $J3$ become available. The algorithm of the min-min heuristic is described below:

Min-min heuristic:

    while not all jobs have been executed **do**
        availableJobs = $\{j \mid parents(j) \text{ are executed}\}$
        schedule(availableJobs)
    end while
procedure schedule(availableJobs)
    while not all availableJobs are scheduled **do**
        for each $j \in$ availableJobs **do**
            for each $r \in \mathbf{R}$ **do**

comptue $ECT(j, r)$

end for

find $min(ECT(j, r))$ for j and $r \in \mathbf{R}$

end for

find $min(min(ECT(j, r)))$ for $j \in$ availableJobs and $r \in \mathbf{R}$

schedule(j) | j has the min-min ECT value

update $EAT(r)$ for $r \in \mathbf{R}$

end while

The min-min heuristic algorithm has a $O(|\mathbf{R}| \times |\mathbf{J}|^2)$ runtime. Since the min-min heuristic proceeds in a level-by-level manner, it fails to detect the data dependency between parent and child functions, and thus, can only produce sub-optimal solutions.

The WBA approach is based on the Greedy Randomized Adaptive Search Procedures (GRASP) heuristic [PR01]. GRASP has been used as a heuristic for many combinatorial optimization problems such as the Job Shop Scheduling problem [GJ90]. The main idea of GRASP is to find an initial schedule using a greedy algorithm, and then iteratively search through the neighbor schedules in a random manner to find better solution. Therefore, the quality of the solution depends on both the number of iterations and the parameter used for the random search. Blythe applies GRASP to workflow-based DAG by first finding an initial solution using the min-min heuristic. Then, it iteratively checks if any neighbor solution can lead to a makespan increase I that is less than $[I_min + \alpha(I_max - I_min)]$. The parameter α is in the range $[0 \leq \alpha \leq 1]$ and is used for the random search. The variable I_min and I_max represent the minimum and the maximum makespan increase of neighbor solutions. The GRASP algorithm with an input workflow W and an iteration limit k is described below:

GRASP Algorithm:

bestSolution $\leftarrow$ min-min-algorithm(W)

while iteration $\leq k$ **do**

solution ← createMapping(W)

if makespan(solution) < makespan(bestSolution) **do**

bestSolution ← solution

end if

end while

procedure createMapping(W)

while not all jobs $\in W$ are mapped **do**

availJobs ← {j | j is unmapped and parents(j) are mapped }

map(availableJobs)

end while

procedure map(availableJobs)

while not all jobs are mapped **do**

for each job $j \in$ availableJobs **do**

for each resource $r \in \mathbf{R}$ **do**

compute $ECT(j, r)$

end for

end for

$I_min \leftarrow min(makespan_Increase(j, r))$ for $j \in$ availableJobs and $r \in \mathbf{R}$

$I_max \leftarrow max(makespan_Increase(j, r))$ for $j \in$ availableJobs and $r \in \mathbf{R}$

availabePairs ← {(j',r') | $makespan_Increase(j', r') \leq I_min +$

$\alpha \times (I_min - I_max)$}

(j'', r'') ← random choice from availPairs

map(j'', r'')

update $EAT(j'', r'')$

end while

Although the GRASP algorithm produces better solutions than the min-min algorithm, its runtime is relatively long given that k is large. Each iteration takes $O(|\mathbf{J}| \times |\mathbf{R}|)$ to execute. Thus, the runtime of GRASP is $O(k \times |\mathbf{J}| \times |\mathbf{R}|)$.

Blythe also proposed a new local selection heuristic called weighted min-min. The main idea is to minimize the idle time of the servers in the local selection process. The intuition is that if $FAT(j, r)$ dominates $EAT(j, r)$, then the server will be idle during the input file transfer. Therefore, the local selection algorithm should optimize both the $ECT(j, r)$ value and the server idle time. Blythe defined a weighted sum of a job j on a resource r as:

$$WT(j, r) = \gamma IT(j, r) + (1 - \gamma)ECT(j, r) \quad (4.1)$$

The weight factor γ is in the range $[0 \leq \gamma \leq 1]$. It is used to determine how much weight the selection algorithm puts on the server idle time $IT(j, r)$ and the $ECT(j, r)$ value. The idle time for job j on resource r is represented as:

$$IT(j, r) = IT(r) + max(0, (FAT(j, r) - EAT(j, r))) \quad (4.2)$$

The server idle time $IT(r)$ is the total idle time of all current jobs scheduled on resource r. Blythe found that the weighted min-min heuristic helps both the TBA and WBA algorithm to produce better solutions, and an optimal weight factor value is γ=0.5.

Similarities. The task scheduling problem is similar to SHP is the sense that its objective is to minimize both the total communication tome T_C and the total execution time T_E. Each instance of the task scheduling can be viewed as an instance of SHP by considering: (a) the task graph T as the input workflow W, (b) the set of initial labeling of W is empty, (c) there is no setup cost for each function.

Differences. The task scheduling problem, on the other hand, differs from SHP in the sense that its total communication time T_C only depends on the communication cost $\{C_{i,j} \mid i, j \in \mathbf{F}\}$. Therefore, if one takes away the function execution time and processors' capacity factor from the cost model, one would be able to minimize T_C by assigning as many functions as possible to a single host. In SHP, T_C depends on both the communication cost $\{C_{i,j} \mid i, j \in \mathbf{F}\}$ and the setup cost $\{U_{i,k} \mid i \in \mathbf{F}, k \in \mathbf{H}\}$. The value of U_i, k varies based on the initial data labeling $\mathbf{L}^x$. Consider Figure 4.2 as an example. Given an execution plan $P_1 = \{(f_1, A), (f_2, A)\}$, we have $C_{f_1,f_2} = 0$, $U_{f_1,A} = 0$, $U_{f_2,A} = 2$, and $T_C = 2$. Now consider

another execution plan $P_2 = \{(f_1, A), (f_2, B)\}$, we have $C_{f_1,f_2} = 1$, $U_{f_1,A} = 0$, $U_{f_2,B} = 0$, and $T_C = 1$. This simple example shows that minimizing C_i, j does not necessary minimize T_C in SHP.

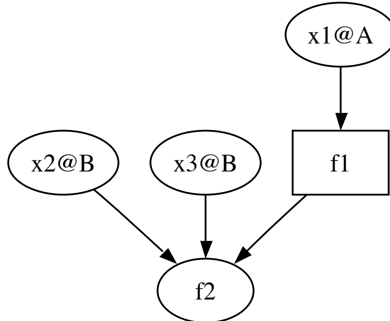

Figure 4.2: Dynamic communication cost between tasks

4.3 Multiprocessor Scheduling Problem

The multiprocessor scheduling problem (MP) [KA99] is analogous to the task scheduling problem (TSP) described in Section 4.2. Its goal is to assign a parallel application (a DAG) to a set of processors such that the completion time of the application is minimal. However, MP differs from TSP in the sense that MP assumes zero communication cost. That is, MP omits the message startup cost I_i in the TSP's cost model.

4.3.1 Mapping between MP and THP

As stated in Section 3.5.2, one can show that MP is a subproblem of THP by creating a mapping between MP and THP components. This mapping is shown in Table 4.1. By looking at this mapping, one can see that each instance I of MP can be viewed as an instance I' of THP. By finding an optimal execution plan P_W defined by $\mathbf{L}_W^F : \mathbf{F} \rightarrow (\mathbf{H}, t)$, one caan construct an optimal total function h for MP by mapping each task t (represented by f) to the corresponding processor p (represented by h) at time t. This shows that THP is as least as hard as MP. In fact, THP is more general than MP in the sense one can specify an initial labeling $\mathbf{L}_W^V$, which assigns a fixed host to a particular vertex before the

Component	MP	THP
Tasks/Functions	T	$\mathbf{F}$
Processors/Hosts	P	$\mathbf{H}$
Task Graph/Abstract Workflow	G	G (representing W_A)
Task Weight/Task Length	A_i	D_f
Processor Speed	S_i	S_h
Process Startup Cost	B_i	SC_f
Transmission Rate/Bandwidth	$R_{i,j}$	$B_{i,j}$
Total Function/Execution Plan	h	P_W

Table 4.1: Component Mapping between MP and THP

scheduling starts.

Example of MP. Figure 4.3 shows an example of a MP task graph G and an optimal total function h. Graph G shows the precedent relationship of tasks. The set of processors (p_1 and p_2) are listed on the Y-axis. The elapsed time of tasks are shown on the X-Axis. In this example, we assume that the processing power of processor p_1 and p_2 both equal to one (that is $c_{p_1} = c_{p_2} = 1$), and tasks have the following duration: $A_1 = 1$, $A_2 = 2$, $A_3 = 2$, $A_4 = 1$, and $A_5 = 1$.

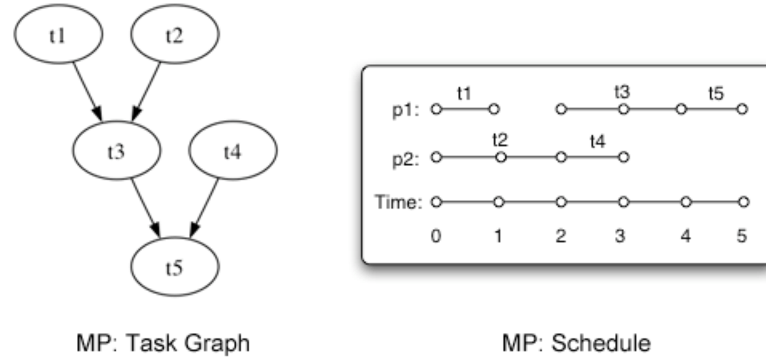

Figure 4.3: Example of the MP problem

Consider Figure 4.3 as an example. The corresponding workflow graph for THP is shown in Figure 4.4. One could construct the same optimal total function h for MP by finding an optimal execution plan P_W for Figure 4.4. This plan is given by $\mathbf{L}_W^F =$

$\{(f_1, h_1, 0), (f_2, h_2, 0), (f_3, h_1, 2), (f_4, h_2, 2), (f_5, h_1, 4)\}$.

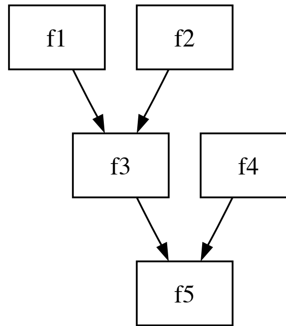

Figure 4.4: Workflow Graph corresponded to Figure 4.3

4.3.2 Heuristics

Kwok and Ahamd [KA99] published a very comprehensive survey on heuristics used to tackle different versions of the multiprocessor scheduling problem. Some of these heuristics assume (a) a DAG input graph structure, (b) zero communication cost, and (c) arbitrary function execution cost. These heuristics are especially relevant to THP. Kwok divides these heuristics into three main categories: (a) Level-based Heuristics, (b) Branch-and-Bound Heuristics, and (c) Analytical Performance Bound Heuristics.

Level-based Heuristics. The following heuristics are first discussed by Adam et al. [ACD74]. The main idea behind these heuristics is to schedule tasks in a level-by-level manner. All tasks in the same level are scheduled based on their priority. Many of these heuristics refer to the *critical path*, which is simply the longest path on the input graph.

1. Highest Level First with Estimated Times (HLFET): This algorithm computes a *level* value for each task by summing up its computation cost from the task node to the final sink. The path must follow along the *critical path*.

2. Highest Level First with No Estimated Times (HLFNET): This algorithm is essentially the same as HLFET except that each task is assumed to have a uniform execution cost.

3. Random: Each task is assigned a random priority.

4. Smallest Co-levels First with Estimated Times (SCFET): This algorithm computes a *co-level* value for each task by summing up the computation cost from the source nodes to the task node.

5. Smallest Co-levels First with No Estimated Times (SCFNET): This algorithm is essentially the same as SCFET except that each task is assumed to have a uniform execution cost.

In Adam's study, these algorithms are ranked by the quality of their solutions. The order sorted from best-to-worst is: HLFET, HLFNET, SCFNET, Random, and SCFET. For detail information of these algorithms, we refer the reader to Adam's original paper [ACD74].

Branch-and-Bound Heuristics. Kasahara [KN84] introduced a depth-first search algorithm that is based on a banch-and-bound approach described by [KS74]. This algorithm first assigns a priority to each task using the *critical path / most immediate successors first* (CP/MISF) technique, then finds a solution by running a depth-first search on the DAG. Kasahara shows that this algorithm is both time and memory efficient, and can produce near-optimal solution.

Analytical Performance Bound Heuristics. Researchers such as Graham [GLLK79] and Rammamoorthy [RCG72] have proposed algorithms to find a bound on the schedule length. Graham proposed an algorithm that is based on general list methods. Using this algorithm, the relationship between the schedule length SL and the optimal schedule length SL_{opt} is

$$SL \leq (2 - \frac{1}{p}) \times SL_{opt} \tag{4.3}$$

where p is the number of processors. In contrast, Rammamoorthy's algorithms focus more on the lower bound of the number of processors needed to finish a computation. His study mainly covers two algorithms: (a) an algorithm that gives the minimum number of processors require to process a DAG in the shortest time possible, and (b) an algorithm that gives the minimum time required to process a DAG given k processors.

4.4 Pegasus: Task Scheduling Strategies

Pegasus is a workflow planner that transforms abstract worklows into concrete workflows using some selection algorithms. Blythe [BJD+05] gave an overview of Pegasus' cost model and its standard selection algorithms.

In Pegasus, a workflow is modeled as a DAG of jobs $J = \{j_1, \ldots, j_m\}$, a set of resources $R = \{r_1, \ldots, r_n\}$, and a set of intermediate files $F = \{f_1, \ldots, f_j\}$. The target machine has the following components:

1. $t(j)$ is the estimated runtime of the job j
2. $s(r)$ is the intrinsic speed of the resource r
3. $\frac{t(i)}{s(r)}$ is the estimated runtime of job j on resource r
4. $b(i, j)$ is the bandwidth between resources i and j
5. $l(f)$ is the size of file f
6. $\frac{l(f)}{b(i,j)}$ is the time it takes to transfer f between resources i and j
7. $q(i)$ is a processing queue for resource i

A sample abstract-to-concrete workflow mapping is shown in Figure 4.5.

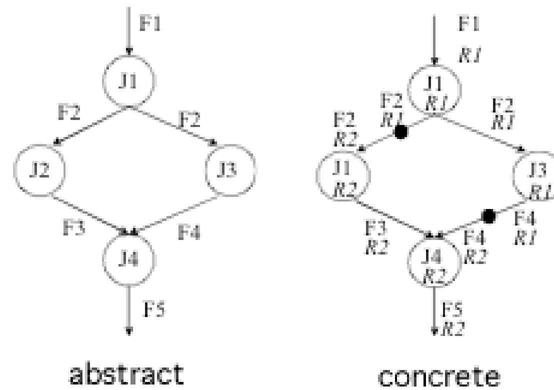

Figure 4.5: Pegasus Workflow Mapping [BJD+05]

One disadvantage of Pegasus' model is that data files are not modeled as nodes, but as labels on arcs. If a job requires a file that is not an output file from previous functions,

one would need to introduce an artificial job to simulate the file retrieval. For example, in Figure 4.5, if $J2$ needs to access a new file $F6$, one would need to create an artificial job $J5$ and link it to $J2$ as shown in Figure 4.6.

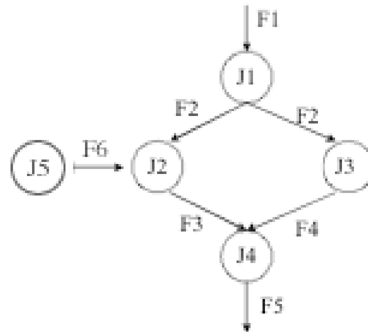

Figure 4.6: Adding artificial node to handle data retrieval

Another disadvantage of this model is that each job does not have a minimum computing requirement. That is each job can run on each resource. This is not likely to happen in a highly heterogeneous environment like the Grid.

Similarities. The workflow mapping problem of Pegasus is similar to SHP in the sense that its objective is to find an execution plan for an abstract workflow such that the total time is minimal. In fact, the workflow graph structure used in the Pegasus' model is equivalent to the workflow graph structure used in SHP. One could transform a Pegasus graph into a SHP graph by adding artificial jobs to handle data retrieval operations. An example is shown in Figure 4.6.

Differences. The workflow mapping problem of Pegasus, on the other hand, differs from SHP in the sense that its cost model does not contain setup time for each function. Many sub-tasks in a scientific workflow make use of both the output data of its parents and data from other sources. Thus, the total communication cost of SHP makes the distinction between function communication cost and function setup cost. Moreover, the Pegasus model assumes that every function can run at every host. In SHP, however, every function has a minimal processing power requirement.

Chapter 5

Implementation

This chapter describes an abstract-to-concrete workflow mapping tool implemented in Java. First, we describe our distributed computing model in Kepler, which is suitable for DSP. Then we explain the objective of our implementation and discuss its infrastructure. At the end, we show how this program works using a couple of examples.

5.1 Distributed Computing Models

A *distributed computing model* (DCM) defines how functions are being executed on distributed machines, and how inter-process communication is done. For example, a program running at host A communicates with another program running at host B by exchanging signals or messages directly with each other, or they may communicate with the help of a central coordinator such as the director in Kepler. The choice of DSM depends on the application domain, and the setting of the computing resources. For example, Cuadrado [Cua05] is studying the Distributed-SDF Domain, in which a centralized SDF director distributes tasks to multiple servers while keeping the program flow synchronized. This model is suitable for applications that have computation cost dominates data transfer cost. In terms of Grid computing, Altintas [Alt05] is studying another approach called the Kepler-Grid. The main idea is to use the JXTA technology to form a P2P network, in which a centralized P2P/JXTA director handles the coordination between peers and peer groups. This model is suitable for applications that require peer discovery, collaboration on-the-fly,

and where the data communication cost is significant.

Distributed computing model for DSP. The DCMs studied by Cuadrado and Altintas are not ideal for DSP because: (a) the DSP model does not have a P2P network infrastructure, and (b) both function and data communication cost are significant in our model. Therefore, we developed a simple and straightforward DCM for DSP. Our DCM has the following characteristics:

1. Remote functions are executed using SSH.
2. Files are transferred using SCP.
3. Files are accessed via local file name references.
4. Access permissions are stored in SSH identity files.
5. Control-flow is coordinated by a PN director.

Using this DCM, each function is represented by a SSH command (a SSH actor in Kepler). The input files of each function are specified by the corresponding local file names. To transfer files between servers, the source uses SCP to push the file to the destination. The PN director maximizes the parallelism of data transfers and function executions by allowing each actor to *fire* (execute) as soon as all input files are in place. This is different from having a sequential execution order enforced by a SDF director.

5.2 Objectives

The objective of our implementation is to test our algorithms for DSP and to prototype an easy-to-use tool for scientists to transform abstract workflows into executable workflows. Users can choose to use Kepler or Graphviz[1] to design their abstract workflows. Also, users can choose what scheduling algorithm to use. The main flow of the program is shown in Figure 5.1.

If the input file is in Graphviz format, our program will generate two complete Graphviz files. One showing the abstract workflow, the other one showing the concrete

[1] http://www.graphviz.org/

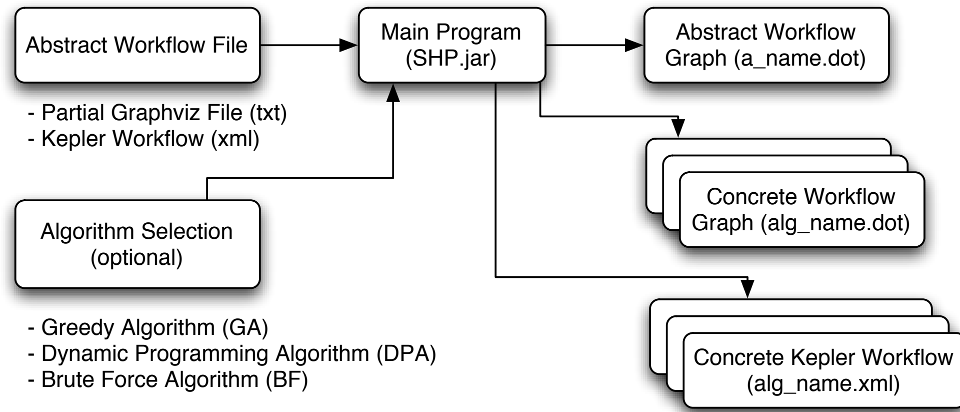

Figure 5.1: Program Flow of the Abstract-to-Concrete Workflow Scheduler

workflow. The cost of the schedule is displayed on the terminal console. An example of this program flow is shown in Figure 5.2.

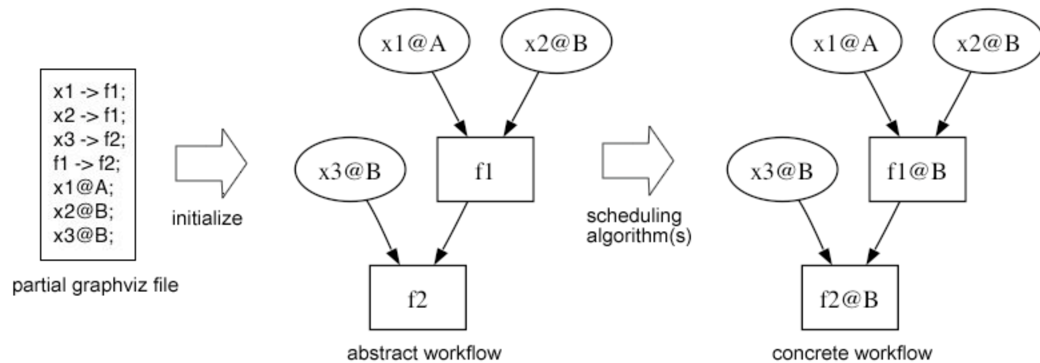

Figure 5.2: Example for generating a concrete workflow from an abstract workflow specified in Graphviz

If the input file is in Kepler MoML format (an XML format), our program will first convert the Kepler workflow into graphviz format, and then generate the corresponding abstract and concrete graphviz files. Next, it converts the concrete graphviz flie back to a concrete Kepler workflow that is executable. An example of this program flow is shown in Figure 5.3.

Recall that the objective of scheduling in DSP (Section 3.3) is to assign a host

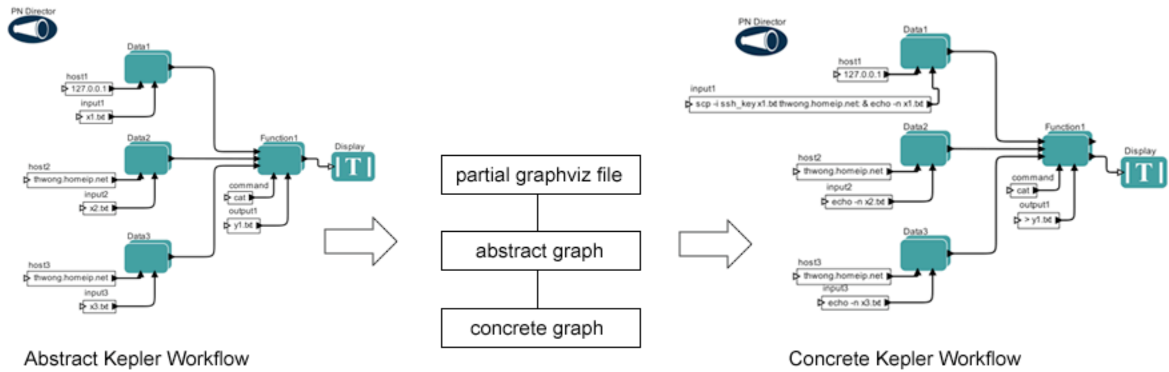

Figure 5.3: Example for generating a concrete workflow from an abstract workflow specified in Kepler

$(A, B, C \dots)$ to each function node $(f_1, f_2, \dots)$ such that the total data transfer time of the whole workflow is minimal assuming that each function execution has zero cost.

5.3 Program Infrastructure

Three DSP algorithms (Section 3.4) are implemented in our program. These algorithms all extend the base algorithm class (Algorithm.class), which contains the cost matrix and other main methods for scheduling the task graph.

- Dynamic Programming Algorithm (DynProgAlg.class)
- Greedy Algorithm (GreedyAlg.class)
- Brute Force Algorithm (BruteForce.class)

Each graph node (node.class) has a name, a set of parent nodes, and a host label. All input file names are assumed to start with the letter x. All output file names are assumed to start with the letter y. The graphviz input file is parsed by the Graph Reader (Reader.class), and the complete graphviz files are generated by the Graph Writer (GraphWriter.class). The parsing of the Kepler workflow is managed by the MoML Handler (MomlHandler.class). The class hierarchy is shown in Figure 5.4.

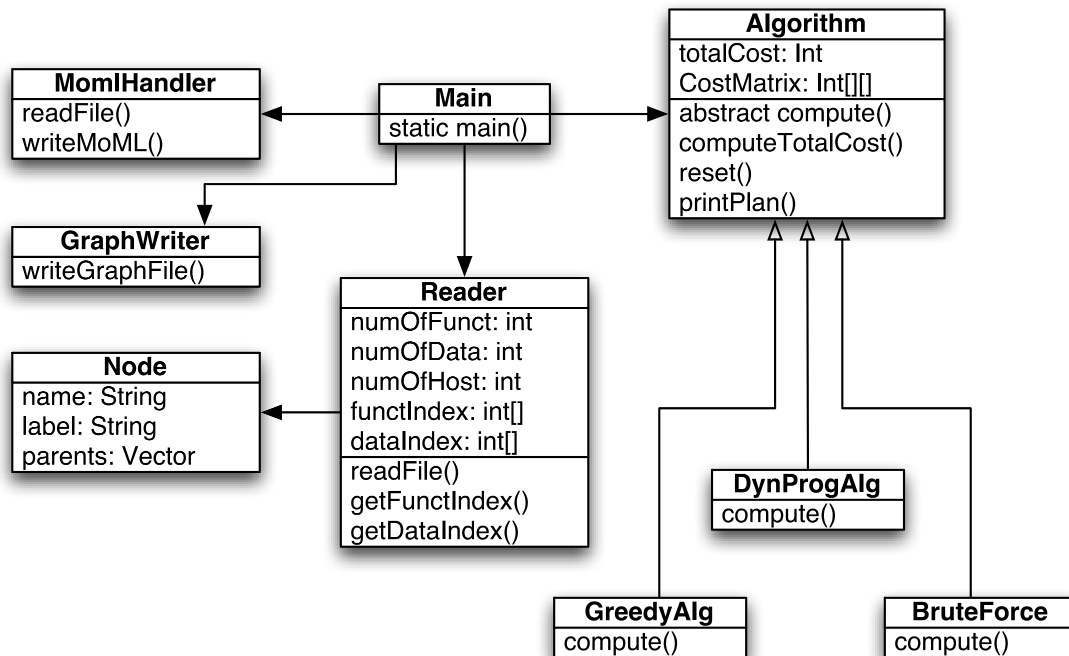

Figure 5.4: Class Hierarchy

5.4 Building Kepler Workflows

This section describes how one can create an abstract workflow, and use our program to transform it into a concrete workflow. An abstract workflow in Kepler is composed of a set of *data actors* and a set of *function actors*.

Data Actor. The structure of a data actor is shown in Figure 5.5. Each data actor contains the following inputs: (a) a *host string* parameter that specifies the location of the data, (b) a *command string* parameter that specifies the file name in an abstract workflow, and the actual SSH command in a concrete workflow, (c) a *user string* that specifies the user name used to access the specified host, and (d) a SSH command actor that executes the specified command in a concrete workflow. The output of a data actor is a file name, which is used as input file for its child actors. If the data actor and its child actors are assigned to the same host, then the SSH command will be an *echo command*, which passes the file name to its child actors. If the data actor and its child actors are assigned to different hosts, then the SSH command will be a sequence of *scp* commands that transfer the data to the appropriate destinations followed by an *echo command* that passes the file name to its child actors.

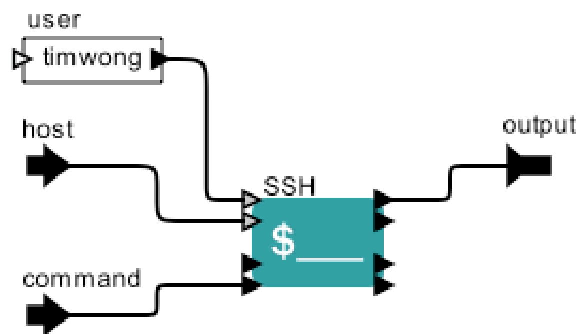

Figure 5.5: Data Actor

Function Actor. The structure of a function actor is shown in Figure 5.6. Each function actor contains the following inputs: (a) a *command string* that specifies the function to use,

(b) a set of *input file strings* that specific the input file names, (c) a *string accumulator* actor that concatenates the command with all the input file names and use the final string as the SSH command, (d) a *user string* that specifies the user name used to access the specified host, (e) a SSH command actor that executes the specified command, and (f) an *output file string* that specifies the output file name. The output of a function actor is a file name, which is used as input file for its child actors.

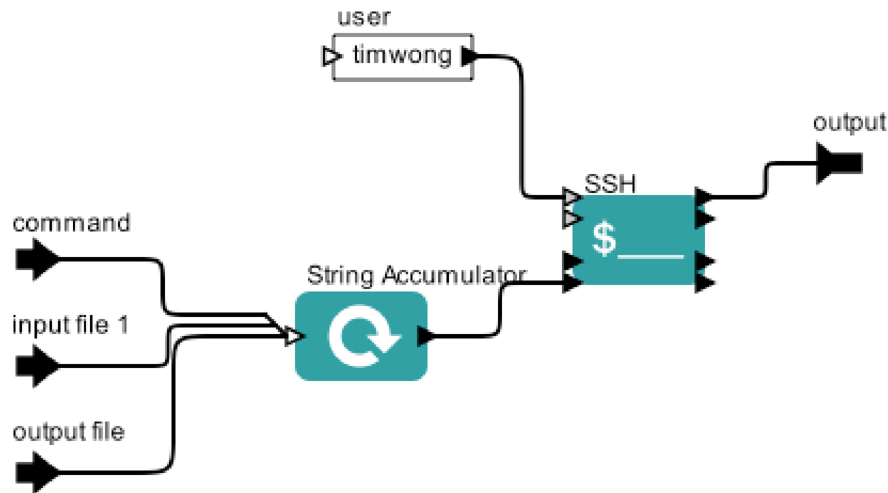

Figure 5.6: Function Actor

To transform an abstract workflow into a concrete workflow using a specific scheduling algorithm, one can run the following command:

java -jar SHP.jar ⟨input file name⟩ ⟨algorithm name⟩

To transform an abstract workflow into multiple concrete workflows using all scheduling algorithms, one can run the following command:

java -jar SHP.jar ⟨input file name⟩

5.5 Examples

This section shows several abstract and concrete Kepler workflows to give the user an idea of how these workflows are constructed. The structure of these workflows is based on the workflow examples shown in pervious chapters. These are special cases where certain

heuristic algorithms can not produce an optimal schedule.

Simple Intree. This simple Intree example is composed of three data actors and one function actor. An optimal plan is to execute the function on thwong.homeip.net since it initially contains the most input data. The greedy algorithm, the dynamic programming algorithm, and the brute force algorithm can all generate this optimal schedule. The abstract workflow is shown in Figure 5.7.

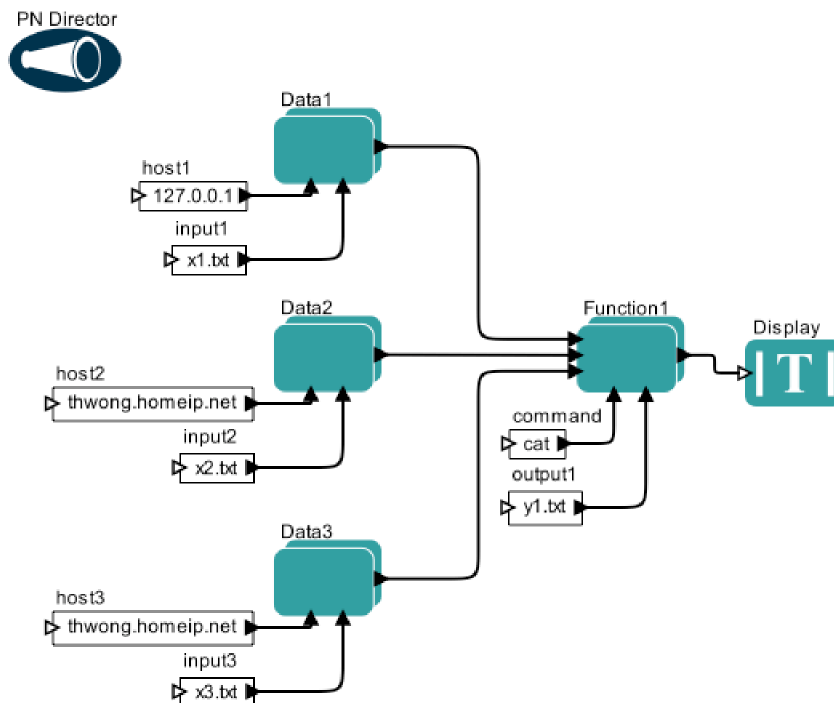

Figure 5.7: Simple Intree Abstract Workflow

The abstract workflow shows: (a) all data file names, (b) all machine to be used, (c) all functions to execute, (d) initial labeling of each data, and (e) the data-flow. However, this workflow cannot be executed, since the files are not yet transferred to the appropriate hosts. For example, before Function1 executes, the file x1.txt must first be transferred to thwong.homeip.net.

The concrete workflow generated by our program will schedule all tasks on the abstract workflow and specify all SSH commands. The concrete workflow is shown in

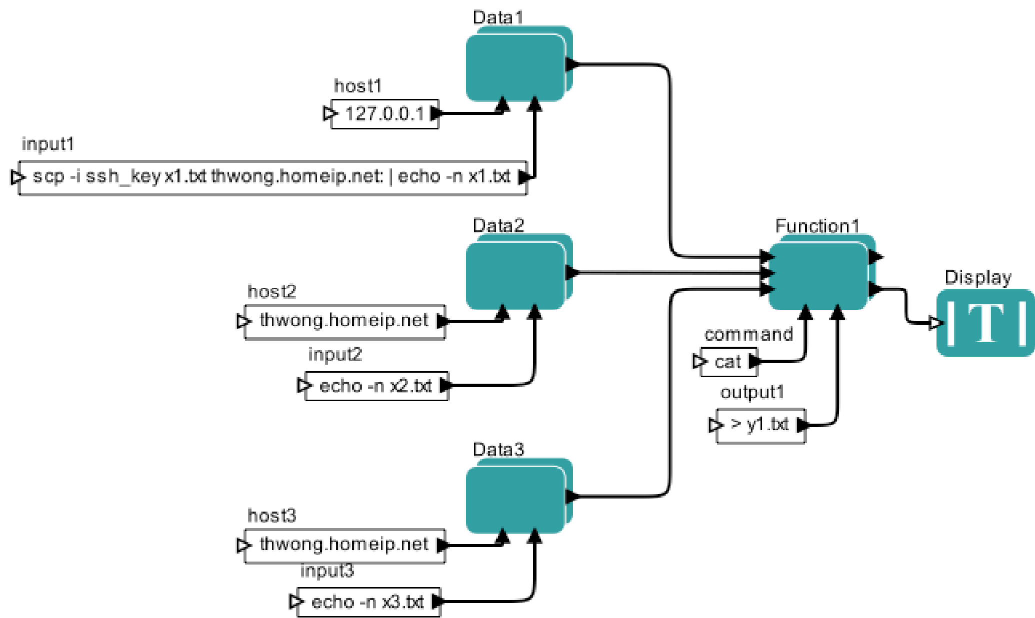

Figure 5.8: Simple Intree Concrete Workflow Scheduled by the Greedy Algorithm

Figure 5.8. Note the changes on *input1, input2, and input3.* If a data actor does not need to transfer any file, it will use the echo command to pass the file name to the function actor. If a data actor needs to transfer a file, it will first use the scp command to push the file to the destination, and then use the echo command to pass the file name to the function actor.

Complex Intree. This complex intree example is composed of three data actors and two function actors. An optimal solution is to execute both functions at thwong.homeip.net. Both the dynamic programming algorithm and the brute force algorithm can produce this optimal schedule. The greedy algorithm; however, would try to execute both functions at 127.0.0.1. The abstract and concrete workflow are shown in Figure 5.9 and Figure 5.10 respectively.

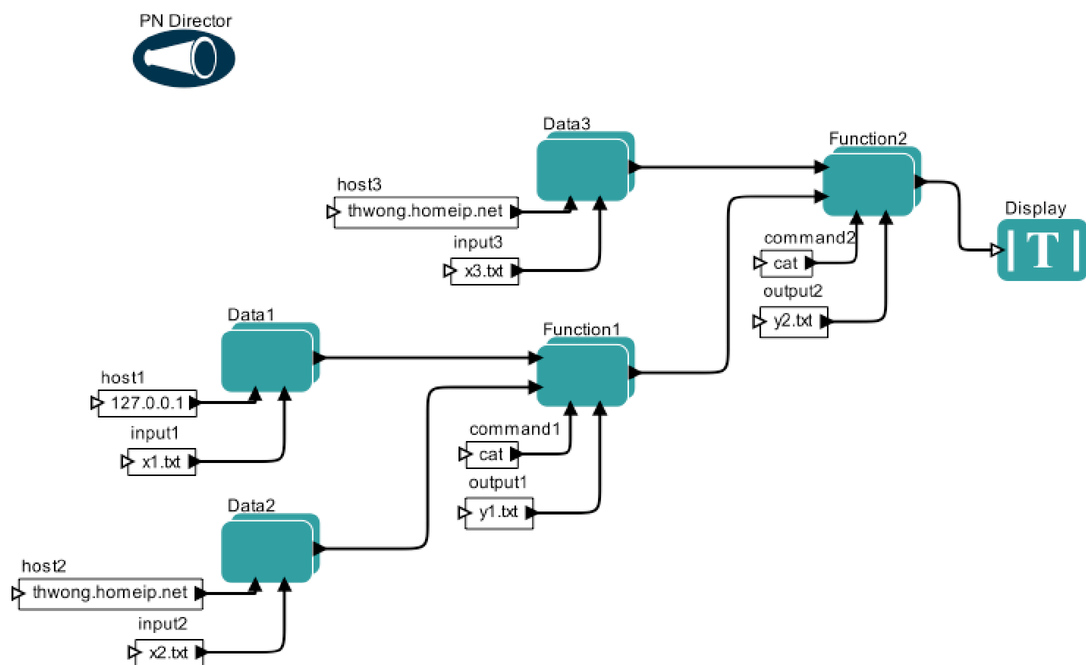

Figure 5.9: Complex Intree Abstract Workflow

Minimal Series Parallel Graph. This MSP graph example is composed of two data actors and five function actors. An optimal solution is to execute all functions at thwong.homeip.net.

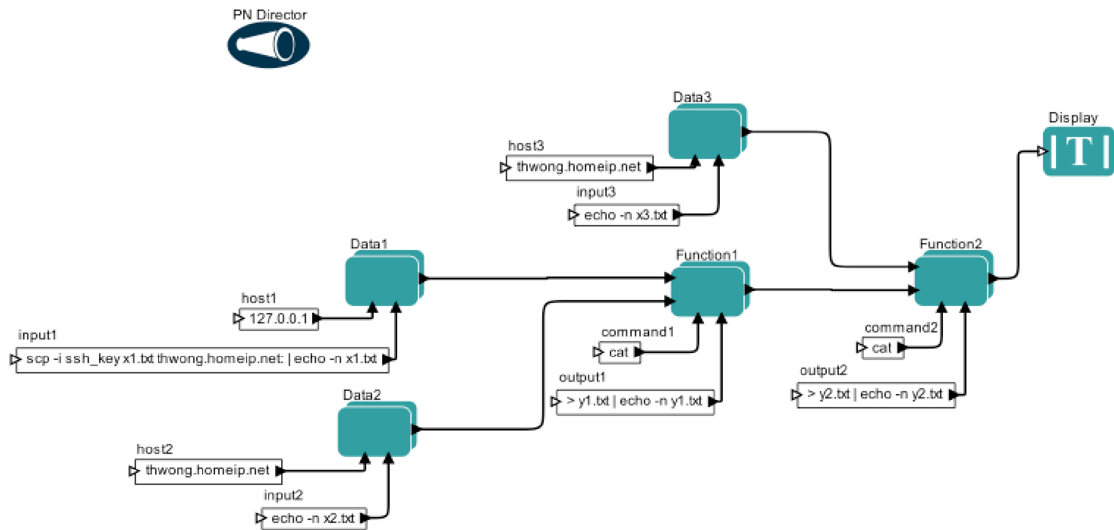

Figure 5.10: Complex Intree Concrete Workflow Scheduled by the Dynamic Programming Algorithm

Only the brute force algorithm can produce this optimal schedule. Both the greedy algorithm and the dynamic programming algorithm would attempt to execute Function2 and Function4 at 127.0.0.1. The abstract and concrete workflow are shown in Figure 5.11 and 5.12 respectively.

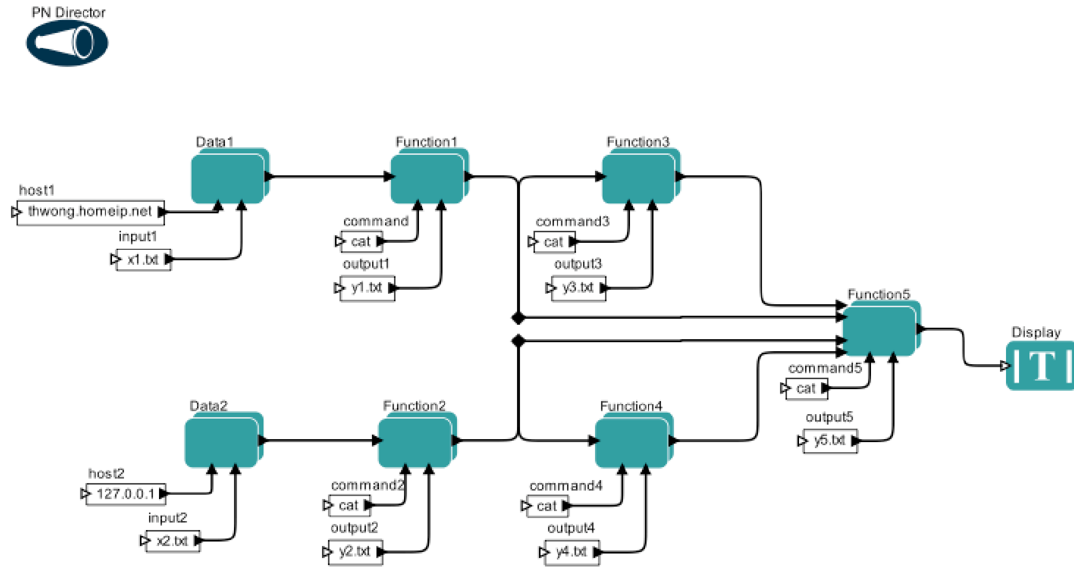

Figure 5.11: MSP Abstract Workflow

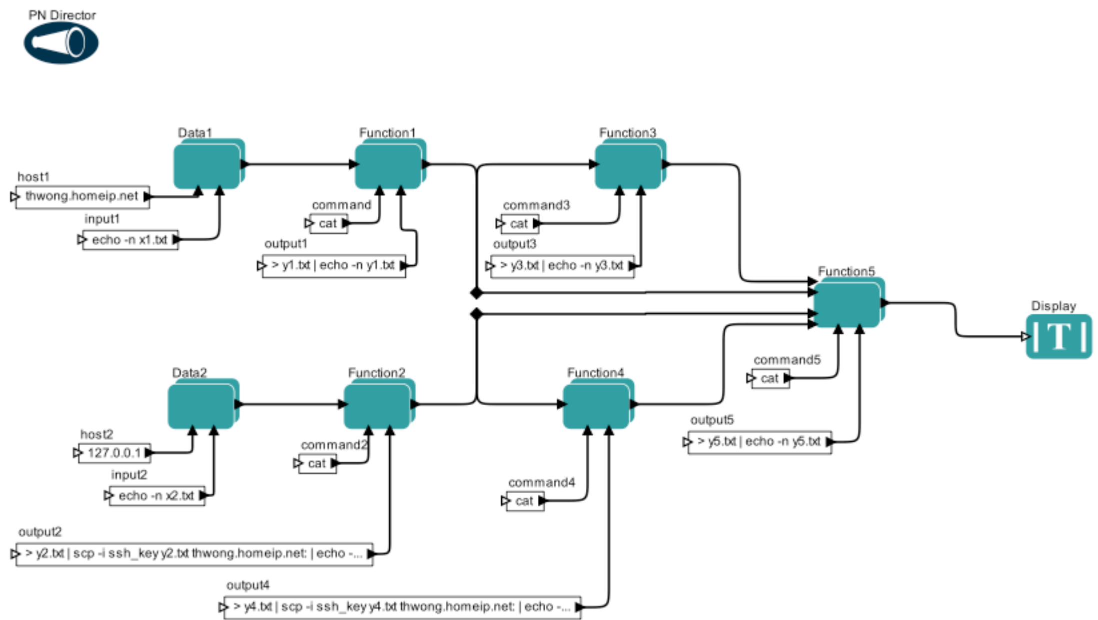

Figure 5.12: MSP Concrete Workflow

Chapter 6

Future Research Directions

This chapter discusses some possible future directions of our research. First, we discuss possible next steps of our research on the theoretical side. Then, we discuss how one can extend our implementation to make it more beneficial for the scientific user community.

6.1 Theoretical Research

This section describes the future work on the theoretical side of our research. There are three open issues that needed to be addressed: (a) the complexity of DSP, (b) the quality of our algorithms, and (c) the variation of SHP in other distributed computing models.

Complexity of DSP. DSP is currently an open problem. We have been trying to find a polynomial time algorithm that can find an optimal solution for DAG input graph. In previous chapters, we show special cases where our greedy algorithm and our dynamic programming algorithm fail to produce an optimal solution. Heuristics for the task scheduling problem and the multiprocessor scheduling problem do not fit because the function execution cost of DSP is assumed to be zero. Heuristics for the distributed query optimization problem do not fit as well because our set of functions contains more than just natural join operations.

We also tried to prove that DSP is NP-complete by finding a reduction from other

well-known NP-complete scheduling problems. A reduction from the task scheduling problem or the multiprocessor scheduling problem is not immediate since the function execution cost in our model is zero. A reduction from the k-coloring problem is not immediate since DSP is not a proper coloring problem. We have also looked at (a) different variations of the shop scheduling problem, (b) the clique problem, and (c) different versions of the shortest path problem, but none of these reductions are immediate.

Quality of our Heuristics. We have a formula (3.6) to calculate the maximum *penalty* (avoidable data transfers) for a schedule generated by our greedy algorithm. However, we do not have a formula that can show the quality of our algorithms. For example, having a formula that shows how far off the schedule is compared to the optimal schedule.

Distributed Computing Models. The formalization of our problems assumes that the input graph is a DAG, and all hosts are connected by a certain network topology. One can find how our problems change when we have a different distributed computing model. For example, if new hosts can be added on-the-fly as in the case of a P2P network, the cost model and the objective of our problems would vary.

6.2 Extensions to our Implementation

This section describes some future extensions one can add to the abstract-to-concrete workflow mapping tool. These extensions include: (a) the implementation of SHP and THP algorithms, (b) the integration with Kepler, and (c) the support for other distributed computing models.

Implementation of SHP and THP Algorithms. All the algorithms implemented in our tool are for DSP. One can implement algorithms used for SHP and THP, and allow users to choose which algorithm to use. These implementations would also allow us to compare our algorithms with heuristics used for the task scheduling problem and for the workflow mapping problem in Pegasus.

Kepler Integration. Currently, our tool is decoupled from Kepler. One can integrate our tool into Kepler either as an actor, which takes in an abstract workflow file and transforms it into a concrete workflow, or as a director, which schedules the actor executions and data transfers on-the-fly.

Support for Other Distributed Computing Models. Currently, our tool uses SSH to execute all functions and uses SCP to perform all data transfers. One can extend our tool such that users can execute functions using other actors, and perform data transfers using other transfer protocols such as GridFTP, FTP, SRB, etc. Moreover, one can extend our tool such that it supports the distributed SDF domain and the KeplerGrid in the future.

Chapter 7

Conclusions

This chapter gives the conclusions of this paper. We first give a summary of our theoretical research. Then, we give an overview of the features provided by our implementation.

7.1 Summary of Findings

This section gives the summary of our theoretical findings. The key contributions of our research include: (a) the formalization of DSP, and its coloring problem, (b) the introduction to its two variations THP and SHP, (c) the hint to the NP-completeness proof for THP and SHP, (d) the design of the brute force algorithm, the greedy algorithm, and the dynamic programming algorithm for DSP.

Formalizations. We give the formalization of DSP by showing its target machine and cost model. Then we give the introduction to its two variations: THP and SHP. The cost model of THP has arbitrary function execution cost and zero communication cost. The cost model of SHP has arbitrary communication cost and arbitrary function execution cost.

Coloring Problem for DSP. We give the formalization of a coloring problem for DSP. The goal of this problem is to color every vertex of a DAG such that the total edge weight is minimal. This objective is differed from traditional graph coloring problems, whose objective is to find a *proper vertex coloring*. A proper coloring is to use the minimum

number of colors to color every vertex of a graph such that no two adjacent vertices have the same color.

NP-completeness. We give hints to the NP-completeness proof of the THP problem. By creating a mapping between THP and the multiprocessor scheduling problem (MP), one can easily see that MP is a sub-problem of THP. In addition, since THP is a sub-problem of SHP, one can also show that SHP is NP-complete.

Heuristic Algorithms. We develop a dynamic programming algorithm for SHP, THP, and DSP. The main idea is to calculate the label cost of each function on each host, and finds the best combination of function labels. This dynamic programming algorithm has a $O(|\mathbf{F}| \times |\mathbf{H}|)$ runtime. In addition, we develop a greedy algorithm and a brute force algorithm for DSP. The main idea of the greedy algorithm is to look at each function individually, and assigns it to a host which contains most of its input data. This greedy algorithm has a $O(|\mathbf{F}|)$ runtime. The main idea of the brute force algorithm is to enumerate all possible execution plans for an abstract workflow, and finds an execution plan that has the minimal total time. This brute force algorithm has a $O(|\mathbf{F}| \times |\mathbf{H}|^{|\mathbf{F}|})$ runtime.

Relationship between SHP and other problems. We show that SHP is similar to the task scheduling problem and the multiprocessor scheduling problem in the sense that their goal is to find a schedule for the input task graph such that the total time is minimal. However, SHP has the notion of an initial labeling, and also has the distinction between function communication cost and function setup cost, which makes its cost model differed form other scheduling problems. We also show that DSP is similar to the distributed query optimization problem in the sense that their goal is to minimize the total number of data transfers. However, the set of functions in DSP contains more than just natural join operations, and statistics may not be available for all input data. This makes the heuristics for the distributed query optimization problem inappropriate for DSP.

7.2 Summary of Implementations

This section gives the summary of the features provided by our abstract-to-concrete workflow mapping tool. The key contributions of our tool include: (a) the design of a DCM that is suitable for SHP, (b) the design of the data and function composite actor for our DCM, (c) the implementation of our heuristic algorithms, (d) the kepler-to-graphviz transformation feature, and (e) the abstract-to-concrete Kepler workflow transformation feature.

Distributed Computing Model. We define a DCM that is suitable for SHP, THP and DSP. In this DCM, every function is executed using a SSH actor, and every data transfer is performed by a SCP command. When a data transfer takes place, the data source will use SCP to push the data to the appropriate destination. The MoC is controlled by a PN director. This maximizes the parallelism between data transfers and function executions.

Data and Function Actor. We develop the data and function composite actor for our DCM. A data actor is used to pass the reference of a file to the function actor, and is used to perform all required data transfers. Each data actor is composed of a SSH actor, and a set of String Constant actors to control the server setting. A function actor is used to execute a local or a remote function. Each function actor is composed of a SSH actor, a String Accumulator actor to concatenate the input function with the input file names, and a set of String Constant actors to control the server setting.

Implementation of Heuristic Algorithms. We have implemented the greedy algorithm, the dynamic programming algorithm, and the brute force algorithm for DSP. After execution, each algorithm will generate a graphviz file to show the abstract workflow, and a graphviz file to show the concrete workflow. Moreover, the algorithm will print out the total cost of the solution on the terminal console.

Transformations. If the input file is in Kepler MoML file format, our tool will convert it into a graphviz file before executing the scheduling algorithm. The graphviz file allows

scientists to easily see the structure of the workflow. Once the execution of the scheduling algorithm finishes, our program will convert the graphviz file back to an executable Kepler workflow.

Bibliography

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[Alt05] Ilkay Altintas. Distributed computing in kepler. *Sixth Biennial Ptolemy Miniconference Program*, 2005.

[BDG+03] G. B. Berriman, E. Deelman, J. Good, J. Jacob, D. S. Katz, C. Kesselman, A. Laity, T. A. Prince, G. Singh, and M. Su. Montage: a grid enabled engine for delivering custom science-grade image mosaics on demand. *Proceedings of the SPIE Conference on Astronomical Telescopes and Instrumentation*, 2003.

[Ber79] Philip A. Bernstein, editor. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.* ACM, 1979.

[BGW+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[BHH06] J.D. Blower, A.B. Harrison, and K. Haines. Styx grid services: Lightweight, easy-to-use, middleware for scientific workflows, 2006.

[BJD+05] Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. *CCGrid*, 2005.

[BKA+04] Viraj Bhat, Scott Klasky, Scott Atchley, Micah Beck, Doug McCune, and Manish Parashar. High performance threaded data streaming for large scale simulations. In Buyya [Buy04], pages 243–250.

[BL04] C. Brooks and Edward A. Lee. Ptolemy ii - heterogeneous concurrent modeling and design in java. Technical Report UCB/ERL M04/27, EECS Department, University of California, Berkeley, 2004.

[BSB01] Tracy D. Braun, Howard Jay Siegel, and Noah Beck. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel and Distributed Computing*, 61:810–837, 2001.

[Buy04] Rajkumar Buyya, editor. *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*. IEEE Computer Society, 2004.

[Cua05] Daniel Lazaro Cuadrado. The distributed-sdf domain. *Sixth Biennial Ptolemy Miniconference Program at UC Berkeley*, 2005.

[DBG+04] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus : Mapping scientific workflows onto the grid. *Across Grids Conference 2004, Nicosia, Cyprus*, 2004.

[DBL80] *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*. IEEE Computer Society, 1980.

[DJP+94] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

[ERLA94] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[ES80] Robert S. Epstein and Michael Stonebraker. Analysis of distributed data base processing strategies. In *VLDB* [DBL80], pages 92–101.

[FKN69] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal of Applied Mathematics*, 17(1), 1969.

[Fre05] James Frew, editor. *17th International Conference on Scientific and Statistical Database Management, SSDBM 2005, 27-29 June 2005, University of California, Santa Barbara, CA, USA, Proceedings*, 2005.

[FVWZ02] I. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation. In *14th International Conference on Scientific Database Management*, Edinburgh, 2002.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GJTY83] M. Garey, D. Johnson, R. Tarjan, and M. Yannakakis. Scheduling opposing forests. *J. Alg. Dis. Math.*, 4(1):72–92, 1983.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math.*, 5:287–326, 1979.

[Gri02] GriPhyN. The grid physics network. http://www.griphyn.org, 2002.

[HG05] Quinn Hart and Michael Gertz. Querying streaming geospatial image data: The geostreams project. In Frew [Fre05].

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 1974.

[Kar72] Richard M. Karp. *Reducibility among combinatorial problems.* Plenum Press, 1972.

[KL05] Tevfik Kosar and Miron Livny. A framework for reliable and efficient data placement in distributed computing systems. *Journal of Parallel and Distributed Computing*, 2005.

[KN84] Hironori Kasahara and Seinosuke Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Computers*, 33(11):1023–1029, 1984.

[KS74] Walter H. Kohler and Kenneth Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *J. ACM*, 21(1):140–156, 1974.

[LAB+05] B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.

[LM87] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[LMM+02] M. Liebendoerfer, O.E.B. Messer, A. Mezzacappa, W. R. Hix, F.-K. Thielemann, and K. Langanke. The importance of neutrino opacities for the accretion luminosity in spherically symmetric supernova models. *Proceedings of the 11th Workshop for Nuclear Astrophysics*, 2002.

[Mor96] J. Paul Morrison. Flow-based programming. *1st International Workshop on Software Engineering for Parallel and Distributed Systems, Berlin*, 1996.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the sixteenth international conference on Very large databases*, pages 314–325, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Pin95] M. Pinedo. Scheduling: Theory, algorithms, and systems. *Prentice-Hall, Englewood Cliffs, NJ*, 1995.

[PPD02] PPDG. Particle physics data grid. http://www.ppdg.net, 2002.

[PPL95] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cyclostatic dataflow. *Proc. IEEE Asilomar Conference on Signals, Systems and Computers Pacific Grove, CA*, 1995.

[PR01] L. Pitsoulis and M. Resende. Greedy randomized adaptive search procedures. *AT&T Labs Research Technical Report*, 2001.

[PY79] Christos H. Papadimitriou and Mihalis Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Comput.*, 8(3):405–409, 1979.

[RCG72] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez. Optimal scheduling stategies in a multiprocessor system. *IEEE Trans. on Computers*, C-21:137–146, 1972.

[RMdL+03] Daniel A Reed, Celso L Mendes, Chang da Lu, Ian Foster, and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*. Morgan Kaufman, San Francisco, CA, second edition, 2003. pp.513-532.

[RWM+03] Arcot Rajasekar, Michael Wan, Reagan Moore, Wayne Schroeder, George Kremenek, Arun Jagatheesan, Charles Cowart, Bing Zhu, Sheau-Yen Chen, and Roman Olschanowsky. Storage resource broker - managing distributed data in a grid. *Computer Society of India Journal, Special Issue on SAN, Vol. 33, No. 4, pp. 42-54*, 2003.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Bernstein [Ber79], pages 23–34.

[SSG02] Arie Shoshani, Alex Sim, and Junmin Gu. Storage resource managers: Middleware components for grid storage. *Nineteenth IEEE Symposium on Mass Storage Systems, 2002 (MSS '02)*, 2002.

[Tav06] Taverna. Taverna workbench. http://taverna.sf.net/, 2006.

[Tri06] Triana. Triana workflow engine. http://trianacode.org, 2006.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[Ull75] Jeffrey D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[VTL79] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12, New York, NY, USA, 1979. ACM Press.

[Won77] Eugene Wong. Retrieving dispersed data from sdd-1: A system for distributed databases. In *Berkeley Workshop*, pages 217–235, 1977.

[WV90] Antoni Wolski and Jari Veijalainen. 2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. In *PARBASE / Databases*, pages 268–287, 1990.

[Xin04] Xiaowen Xin. Case study: Terascale supernova initiative workflow. *LLNL Technical Note*, 2004.