Multisite Management of Scientific Workflows in the Cloud

Ji Liu

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**[*]
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Ji Liu**
ji.liu@inria.fr

Gestion multisite de workflows scientifiques dans le cloud

Soutenue le 03/11/2016 devant le jury composé de :

M. Pascal MOLLI	Professeur	Université de Nantes	Rapporteur
M. Sébastien MONNET	Professeur	Université de Savoie	Rapporteur
Mme. Christine MORIN	Directrice de recherche	INRIA	Examinatrice
Mme. Marta MATTOSO	Professeur	COPPE / Federal University of Rio de Janeiro	Co-encadrante
Mme. Esther PACITTI	Professeur	Université de Montpellier	Co-directrice
M. Patrick VALDURIEZ	Directeur de recherche	INRIA	Co-directeur

[*] **I2S:** ÉCOLE DOCTORALE INFORMATION STRUCTURES SYSTÈMES

Don’t lose faith,
as long as the unremittingly,
you will get some fruits.

—Hsue-Chu Tsien

Dedication

To my family.

Acknowledgments

First, I would like to express my sincere appreciation to my advisors: Esther Pacitti, Patrick Valduriez and Marta Mattoso for their patient, helpful, effective guidance. They gave me excellent advice to improve my writing skills, helped me to develop my own ideas and corrected my papers, posters, articles. In addition, they offered me many opportunities to connect with reputable experts in different disciplines.

I would like to also thank my family: Jirong Liu and Xiuqin Hao, for giving birth to me and providing me with continuous encouragement and support. You make me strong in heart so that I could pursue my dreams without hesitation.

I am also deeply grateful to my committee members, Pascal Molli and Sébastien Monnet and my evaluator Christine Morin for spending their time to evaluate my thesis.

I would like to express my gratitude towards my internship supervisors and collaborators at Microsoft, Seattle, USA: Vinod Nair, Markus Weimer, Sriram Rao and Raghu Ramakrishnan. I am thankful to the collaborators and contributors during my thesis. To Gabriel Antoniu, Alexandru Costan, Luis Pineda-Morales, Radu Tudoran, Raghu Ramakrishnan, Olivier Nano, Laurent Massoulie and Pierre-Louis Xech for the great collaboration that we had in the Z-CloudFlow project. To Vítor Silva, Daniel de Oliveira and Kary Ocaña for our productive collaboration in the MUSIC project. I would also like to thank Weiwei Chen, who helped me with the experiments and gave me valuable advice on my ideas. Special thanks go to Idy Camara, Leye Wang, Qiu Han, Yue Li, Haoyi Xiong and Xin Jin, who offered me their advice on my research and my career planning. Thanks also go to Amelie Chi Zhou, who shared her valuable research experience with me.

I would also like to record my sincere thanks to all the current and former members of the Zenith team. Special thanks to Reza Akbarinia, Maximilien Servajean, Miguel Liroz, Mohamed Reda Bouadjenek, Saber Salah, Oleksandra Levchenko and Rim Moussa, who gave me helpful advice on my research work. Thanks also go to Tristan Allard, Florent Masseglia, Daniel Gaspar, Djamel-Edine Yagoubi, Sen Wang and Carlyna Bondiombouy, with whom, I often recall the happy time we spent together.

I am grateful to my friends who are particular helpful along this path. Thanks go to Zhan Li, Wenlong Liu and Liuyang Zhang for all the fun we had and their help on my thesis.

Finally, I would like to thank everyone at UM, LIRMM, MSR-INRIA, INRIA and all the other people who had a direct or indirect contribution to this work and were not mentioned above. I appreciate their help and support.

Résumé

Les *workflows scientifiques* (*SWfs*) permettent d'exprimer facilement des activités de calcul sur des données, comme charger des fichiers d'entrée, exécuter des analyses, et agréger les résultats. Un SWf décrit les dépendances entre les activités, généralement comme un graphe où les nœuds sont des activités et les arêtes représentent les dépendances entre les activités. Les SWfs sont souvent orientés-données, manipulant de grandes quantités de données. Afin d'éxecuter des SWfs orientés-données dans un temps raisonnable, les *systèmes de gestion de workflows scientifiques* (*SWfMSs*) peuvent être utilisés et déployés dans un environnement de calcul à haute performance (HPC).

Parce qu'il offre des services stables et des ressources de calcul et de stockage quasiment infinies à un coût raisonnable, le cloud devient attractif pour l'exécution de SWfs. Un cloud est généralement constitué de plusieurs sites (ou *data centers*), chacun avec ses propres ressources et données. L'exécution de SWf doit alors être adaptée à un cloud multisite tout en exploitant les ressources de calcul ou de stockage distribuées.

Dans cette thèse, nous étudions le problème d'exécution efficace des SWfs orientés-données dans un cloud multisite. La plupart des SWfMSs ont été conçus pour des clusters ou grilles, et quelques uns ont été étendus pour le cloud, en les déployant simplement dans des machines virtuelles (VMs), mais seulement pour un seul site. Pour résoudre le problème dans le cas multisite, nous proposons une approche distribuée et parallèle qui exploite les ressources disponibles de chaque site. Pour exploiter le parallélisme, nous utilisons une approche algébrique, qui permet d'exprimer les activités en utilisant des opérateurs et les transformer automatiquement en de multiples tâches.

La principale contribution de la thèse est une architecture multisite et des techniques distribuées pour exécuter les SWfs. Les principales techniques utilisent des algorithmes de partitionnement de SWf, un algorithme dynamique pour le provisionnement de VMs, un algorithme d'ordonnancement des activités et un algorithme d'ordonnancement de tâches. Les algorithmes de partitionnement de SWfs décomposent un SWf en plusieurs fragments, chacun pour un site différent. L'algorithme dynamique pour le provisionnement de VMs est utilisé pour créer une combinaison optimale de VMs pour exécuter des fragments à chaque site. L'algorithme d'ordonnancement des activités distribue les fragments vers les sites, selon un modèle de coût multi-objectif, qui combine à la fois temps d'exécution et coût monétaire. L'algorithme d'ordonnancement de tâches distribue directement des tâches sur les différents sites en réalisant l'équilibrage de charge au niveau de chaque site. Nos expérimentations montrent que notre approche peut réduire considérablement le coût global de l'exécution de SWfs dans un cloud multisite.

Titre en français

Gestion multisite de workflows scientifiques dans le cloud

Mots-clés

- Workflow scientifique
- Système de gestion de workflows scientifiques
- Multisite cloud
- Ordonnancement

Abstract

Scientific Workflows (*SWfs*) allow scientists to easily express multi-step computational activities, such as load input data files, process the data, run analyses, and aggregate the results. A SWf describes the dependencies between activities, typically as a graph where the nodes are activities and the edges express the activity dependencies.

SWfs are often data-intensive, *i.e.* process, manage or produce huge amounts of data. In order to execute data-intensive SWfs within a reasonable time, *Scientific Workflow Management Systems* (*SWfMSs*) can be used and deployed in High Performance Computing (HPC) environments (cluster, grid or cloud). By offering stable services and virtually infinite computing, and storage resources at a reasonable cost, the cloud becomes appealing for SWf execution. SWfMSs can be easily deployed in the cloud using Virtual Machines (VMs). A cloud is typically made of several sites (or data centers), each with its own resources and data. Since a SWf may process data located at different sites, SWf execution should be adapted to a multisite cloud while exploiting distributed computing or storage resources.

In this thesis, we study the problem of efficiently executing data-intensive SWfs in a multisite cloud, where each site has its own cluster, data and programs. Most SWfMSs have been designed for computer clusters or grids, and some have been extended to operate in the cloud, but only for single site. To address the problem in the multisite case, we propose a distributed and parallel approach that leverages the resources available at different cloud sites. To exploit parallelism, we use an algebraic approach, which allows expressing SWf activities using operators and automatically transforming them into multiple tasks.

The main contribution is a multisite architecture for SWfMSs and distributed techniques to execute SWfs. The main techniques consist of SWf partitioning algorithms, a dynamic VM provisioning algorithm, an activity scheduling algorithm and a task scheduling algorithm. SWf partitioning algorithms partition a SWf to several fragments, each to be executed at a different cloud site. The VM provisioning algorithm is used to dynamically create an optimal combination of VMs for executing workflow fragments at each cloud site. The activity scheduling algorithm distributes the SWf fragments to the cloud sites based on a multi-objective cost model, which combines both execution time and monetary cost. The task scheduling algorithm directly distributes tasks among different cloud sites while achieving load balancing at each site. Our experiments show that our approach can reduce considerably the overall cost of SWf execution in a multisite cloud.

Title in English

Multisite Management of Scientific Workflows in the Cloud

Keywords

- Scientific workflow
- Scientific workflow management system
- Multisite cloud
- Scheduling

Equipe de Recherche

Zenith Team, INRIA & LIRMM

Laboratoire

LIRMM - Laboratoire d’Informatique, Robotique et Micro-électronique de Montpellier

Adresse

Université Montpellier
Bâtiment 5
CC 05 018
Campus St Priest - 860 rue St Priest
34095 Montpellier cedex 5

Résumé Étendu

Introduction

Les *workflows scientifiques* (*SWfs*) permettent d'exprimer des activités de calcul à étapes multiples, *p. ex.* charger les fichiers d'entrée, traiter les données, exécuter les analyses, et agréger les résultats. Les activités de calcul sont liées par des dépendances. Un SWf décrit les activités et les dépendances généralement sous forme de graphe, où les nœuds représentent les activités de calcul et les arêtes représentent les dépendances entre elles. Les SWfs sont largement utilisés dans plusieurs domaines, tels que l'astronomie [59], la biologie [137], la physique [138], la sismologie [56], la météorologie [190], et cetera.

Les SWfs sont souvent *orientés-données*, *c.-à-d.* traitent, gèrent ou produisent d'énormes quantités de données. La gestion et la manipulation des SWfs orientés-données avec des outils traditionnels de programmation (*p. ex.* des bibliothèques de code, des langages de script) devient très difficile à mesure que la complexité augmente. Par conséquent, les systèmes de gestion de workflows scientifiques (*SWfMSs*) ont été spécialement mis au point afin de faciliter le traitement de SWfs, qui incluent de nombreux aspects tels que la modélisation, la programmation, le débogage, et l'exécution de SWfs. Les SWfMSs peuvent générer des données de provenance en cours d'exécution des SWfs. Les données de provenance, qui retracent l'exécution de SWfs et la relation entre les données d'entrée et les données de sortie, sont parfois plus importantes que l'exécution elle-même. Pour exécuter les SWfs orientés-données dans un délai raisonnable, les SWfMSs exploitent les techniques de parallélisme avec des ressources de calcul à haute performance (HPC) dans un environnement de cluster, grille ou cloud. Quelques SWfMSs existants, *p. ex.*, Pegasus [60, 61], Swift [201], et Chiron [139], sont accessibles au public pour l'exécution et la gestion de SWfs. Cependant, la plupart d'entre eux sont conçus pour les environnements de cluster ou grille. Dans les environnements de cloud, les SWfMSs utilisent généralement les mêmes approches conçues pour le calcul de clusters ou de grilles, qui ne sont pas optimisées pour les environnements de cloud.

En offrant des ressources quasi infinies, des services évolutifs et divers, la qualité de service stable et des politiques de paiement flexibles, le cloud devient une solution attractive pour l'exécution de SWfs. Les SWfMSs peuvent être facilement déployés dans le cloud en exploitant des Machines Virtuelles (VMs). Avec une méthode de pay-as-you-go, les utilisateurs de cloud n'ont pas besoin d'acheter des machines physiques et la maintenance des machines est assurée par les fournisseurs de cloud. Ainsi, les environnements de cloud deviennent les infrastructures intéressantes pour l'exécution de SWfs.

Un cloud est typiquement *multisite* (composé de plusieurs sites ou centres de données), avec chacun ses propres ressources et données et est explicitement accessible aux utilisateurs du cloud. En raison d'une faible latence et des problèmes de propriété, les données sont généralement stockées dans le site de cloud où sont localisées les sources de données. En conséquence, les données d'entrée d'un SWf peuvent être distribuées géographiquement. Par exemple, les données climatiques dans le système terrestre de grille [189], les grandes quantités de données brutes de la chromodynamique quantique (QCD) [149] et les données du projet ALICE [1] sont distribuées géographiquement. Puisqu'un SWf peut traiter des données distribuées géographiquement, l'exécution de SWf doit être adaptée à un cloud multisite en exploitant les ressources de calcul ou de stockage distribuées au-delà d'un site de cloud. Les approches existantes restent limitées à des environments avec un seul cluster, dans une grille ou un cloud, et ne sont pas adaptées à un environnement multisite.

Cette thèse a été préparée dans le cadre de deux projets scientifiques : Z-CloudFlow (projet du centre MSR-Inria avec l'équipe Inria Kerdata) et MUSIC (projet FAPERJ-Inria avec des équipes de Rio de Janeiro) avec l'objectif principal d'exécuter efficacement les SWfs orientés-données dans un cloud multisite, où chaque site a son propre cluster, ses données et ses programmes. Cette thèse contient 5 chapitres principaux : état de l'art, partitionnement de SWfs, provisionnement de VMs dans un seul site, ordonnancement multi-objectif de SWfs dans un cloud multisite et ordonnancement de tâches avec les données de provenance. Elle commence par un chapitre d'introduction et se termine par un chapitre de conclusion qui résume les contributions et propose des directions de recherche futures.

État de l'art

Un SWf est l'assemblage d'activités scientifiques de traitement de données avec des dépendances de données entre elles [57]. Un SWfMS est un outil efficace pour exécuter les SWfs et gérer des ensembles de données dans différents environnements informatiques. Afin d'exécuter un SWf dans un environnement donné, un SWfMS génère un plan d'exécution de workflow (WEP), qui est un programme qui saisit les décisions d'optimisation et les directives d'exécution, typiquement le résultat de la compilation et l'optimisation d'un workflow, avant l'exécution. Cette section présente les techniques existantes de SWfs et SWfMSs, y compris l'architecture fonctionnelle, les techniques de parallélisation, l'analyse de SWfMSs différents et l'environnement de cloud multisite.

L'architecture fonctionnelle d'un SWfMS peut être décrite en couches comme suit [60, 201, 21, 139] : présentation, services aux utilisateurs, génération de WEP, exécution de WEP et infrastructures. Un utilisateur interagit avec un SWfMS à travers la couche de présentation et réalise les fonctions souhaitées dans la couche de services aux utilisateurs. La couche de services d'utilisateur prend généralement en compte les données de provenance, qui sont les métadonnées qui capturent l'histoire de dérivation d'un ensemble de données. Un SWf est traité dans la couche de génération de WEP pour produire un WEP, qui est exécuté dans la couche d'exécution de WEP. Afin de réduire le temps d'exécution, les SWfs sont généralement exécutés en parallèle. Le SWfMS accède aux ressources physiques à travers la couche d'infrastructure pour l'exécution de SWfs.

L'exécution en parallèle de SWfs comprend le parallélisme et l'ordonnancement. Le parallélisme de SWfs identifie les tâches qui peuvent être exécutées en parallèle. Il y a deux niveaux de parallélisme : le parallélisme à gros grain et le parallélisme à grain fin. Le parallélisme à gros grain, qui est effectué au niveau de SWf, est obtenu en exécutant des fragments de SWfs en parallèle. Un fragment de SWf (ou fragment pour faire court) peut être défini comme un sous-ensemble des activités et des dépendances de données d'un SWf original, qui est généré par le partitionnement de SWf. Le parallélisme à grain fin réalise le parallélisme en exécutant différentes activités en parallèle dans un SWf ou un fragment du SWf. L'ordonnancement de SWfs est un processus d'attribution de tâches aux ressources informatiques (*c.-à-d.* nœuds de calcul) à exécuter [33]. Les méthodes d'ordonnancement peuvent être statiques, dynamiques ou hybrides. L'ordonnancement statique génère un plan d'ordonnacement (SP) qui attribue toutes les tâches exécutables aux nœuds de calcul avant l'exécution et le SWfMS respecte strictement le SP pendant toute l'exécution de SWf [33]. Il est efficace lorsque l'environnement d'exécution varie peu au cours de l'exécution de SWfs, et quand le SWfMS a suffisamment d'informations sur les capacités informatiques et de stockage des nœuds de calcul correspondants. L'ordonnancement dynamique produit des SPs qui distribuent les tâches exécutables aux nœuds de calcul lors de l'exécution de SWfs [33]. Ce type d'ordonnacement est approprié pour les SWfs dont la charge de travail des tâches est difficile à estimer, ou pour les environnements où les capacités des nœuds de calcul varient beaucoup pendant l'exécution. Les méthodes d'ordonnancement statiques et dynamiques ont leurs propres avantages. Elles peuvent être combinées en méthode d'ordonnacement hybride pour obtenir de meilleures performances.

Nous avons étudié huit SWfMSs typiques : Pegasus, Swift, Kepler, Taverna, Chiron, Galaxy, Triana [173], Ascalon [68] ; ainsi que le portail de SWfMSs WS-PGRADE/gUSE [105]. Pegasus et Swift ont un excellent soutien sur l'évolutivité et la haute performance de SWfs orientés-données. Pegasus, Swift, Kepler, Taverna et WS-PGRADE/gUSE sont largement utilisés dans l'astronomie, la biologie, et cetera. Par contre, Galaxy ne peut exécuter que les SWfs bioinformatiques. Tous les frameworks supportent le parallélisme à grain fin, l'ordonnancement dynamique et trois d'entre eux (Pegasus, Kepler et WS-PGRADE/gUSE) supportent l'ordonnancement statique. Tous ces systèmes supportent l'exécution de SWfs dans l'environnement de grille et de cloud. Chiron exploite une approche algébrique [146] pour gérer l'exécution en parallèle de SWfs orientés-données. Il utilise un modèle de données algébriques pour exprimer toutes les données comme les relations et représentent les activités de SWfs comme des expressions algébriques dans la couche de présentation. Une relation contient des ensembles de tuples composés d'attributs de base. Une expression algébrique consiste en activités algébriques, opérandes supplémentaires, opérateurs et relations d'entrée et de sortie. Une activité algébrique contient un programme ou une expression SQL, et les schémas de relations d'entrée et de sortie.

Un opérande supplémentaire est l'information latérale pour l'expression algébrique, qui peut être une relation ou un ensemble d'attributs de regroupement. Il y a six opérateurs qui peuvent automatiquement transformer une activité en de multiples tâches à exécuter.

Il y a des cas importants où les SWfs devront être exécutés sur plusieurs sites de cloud, *p. ex.* parce que les données accessibles par le SWf sont dans les bases de données de différents groupes de recherche dans les différents sites ou parce que l'exécution d'un SWf a besoin de plus de ressources que celles d'un seul site. Les grands fournisseurs de cloud tels que Microsoft et Amazon ont généralement plusieurs centres de données distribués géographiquement dans les différents sites. Dans un cloud multisite, nous pouvons exécuter un SWf avec le parallélisme à grain fin ou le parallélisme à gros grain au niveau multisite. L'exécution de SWfs avec le parallélisme à grain fin est d'attribuer toutes les tâches dans chaque site de cloud. Bien qu'il existe des méthodes d'ordonnancement [65, 145], elles n'ont pas de support à la gestion des données de provenance, celles qui sont importantes pour l'exécution de SWfs. Avec le parallélisme à gros grain, un SWf est partitionné en fragments. Chaque fragment est affecté à un site spécifique et ses tâches sont allouées dans les VMs de ce site. Certaines méthodes [38, 39] sont proposées pour permettre d'exécuter les SWfs dans un cloud multisite par le partitionnement de SWfs, ils se concentrent généralement sur un seul objectif, *c.-à-d.* la réduction du temps d'exécution, avec une contrainte de stockage mais le cas multi-objectif reste un problème, *p. ex.* réduire à la fois le temps d'exécution et le coût monétaire. En outre, ils n'ont généralement pas de support pour le provisionnement dynamique de VMs dans le cloud, ce qui est essentiel pour l'exécution dans un environnement de cloud. Chiron est adapté au cloud grâce à son extension, Scicumulus [51, 52], qui supporte le provisionnement de calcul dynamique [50]. Cependant, cette approche se concentre sur un environnement de cloud mono site.

Partitionnement de SWfs

Nous attaquons le problème de partitionnement de SWfs afin d'exécuter les SWfs dans un cloud multisite. Notre objectif principal est de permettre l'exécution de SWfs dans un cloud multisite pour partitionner les SWfs en fragments afin de réduire le temps d'exécution, tout en assurant que certaines activités soient exécutées sur les sites de cloud spécifiques.

Il y a essentiellement deux techniques de partitionnement de SWfs, *c.-à-d.* le partitionnement de DAG et le partitionnement de données. Le partitionnement de DAG transforme un DAG composé des activités en un DAG composé des fragments tandis que chaque fragment est un DAG composé des activités et des dépendances. Le partitionnement de données divise les données d'entrée d'un fragment généré par le partitionnement de DAG en plusieurs ensembles de données, dont chacun est encapsulé dans un fragment nouvellement généré. Nous nous concentrons sur le partitionnement de DAG dans ce travail. Il y a une technique de partitionnement général, *c.-à-d.* l'encapsulation des activités, qui encapsule chaque activité dans un fragment de SWf.

Nous proposons trois méthodes de partitionnement de SWfs, *c.-à-d.* la confidentialité scientifique (SPr), la minimisation de la transmission de données (DTM) et l'adaptation de la capacité informatique (CCA). SPr partitionne les SWfs en mettant les activités de blocage et ses activités suivantes disponibles à un fragment, pour mieux supporter la surveillance de l'exécution sous la contrainte de sécurité. DTM partitionne les SWfs avec la prise en compte des activités de blocage, tout en minimisant le volume de données à transférer entre les fragments de SWfs différents. CCA partitionne les SWfs selon la capacité de calcul de différents sites de cloud. Cette technique tente de mettre plus d'activités au fragment à exécuter dans un site de cloud avec une plus grande capacité de calcul. Nos techniques de partitionnement sont adaptées aux différentes configurations de cloud afin de réduire le temps d'exécution des SWfs. De plus, nous proposons également d'utiliser des techniques de raffinage de données, *c.-à-d.* la combinaison de fichiers et la compression de données, afin de réduire le temps de la transmission de données entre les différents sites.

Nous prenons Buzz SWf [64] comme un cas d'utilisation et adaptons Chiron pour l'exécution de SWfs dans le cloud multisite. Nous évaluons largement nos techniques de partitionnement proposées en exécutant Buzz avec Chiron déployé dans deux sites, *c.-à-d.* Europe occidentale et l'Est des États-Unis, du cloud Azure. Nous considérons deux configurations de cloud : homogène et hétérogène. Le cas où tous les sites ont les mêmes nombres et types de VMs correspond à la configuration homogène tandis que dans une configuration hétérogène les sites ont des nombres ou types de VMs différents. Les résultats de nos expérimentations montrent que DTM avec des techniques de raffinage de données est adapté ($24,1\%$ du temps épargné par rapport à ACC sans raffinage de données) à exécuter les SWfs dans un cloud multisite avec une configuration homogène, et qu'ACC fonctionne mieux ($28,4\%$ du temps épargné par rapport à la technique SPr sans raffinage de données) avec une configuration hétérogène. En outre, les résultats montrent que les techniques de raffinage de données peuvent réduire considérablement le temps de la transmission de données entre deux sites différents.

Provisionnement de VMs dans un site

Nous traitons le problème de la génération de plans de provisionnement de VMs pour l'exécution de SWfs dans un seul site de cloud pour plusieurs objectifs. Notre principale contribution est de proposer un modèle de coût et un algorithme afin de générer des plans de provisionnement de VMs pour réduire à la fois le temps d'exécution et le coût monétaire pour l'exécution de SWfs dans un seul site de cloud.

Pour résoudre le problème, nous concevons un modèle de coût multi-objectif pour l'exécution de SWfs dans un seul site de cloud. Le modèle de coût est une fonction pondérée avec les objectifs de réduction du temps d'exécution et le coût monétaire basé sur le temps d'exécution et le coût monétaire souhaité par les utilisateurs. L'importance de l'objectif du temps d'exécution doit être supérieure à zéro et inférieure à 1, *p. ex.* 0.1 ou 0.9. Notre modèle de coût considère la charge de travail séquentiel et le coût pour démarrer les VMs, qui est plus précis par rapport aux modèles de coûts existants, *p. ex.* GraspCC [47]. Le temps d'exécution estimé est basé sur la loi d'Amdahl [170]. Le coût monétaire estimé est basé sur le temps d'exécution estimé et le coût monétaire pour utiliser des VMs dans une unité de temps.

En nous appuyant sur le modèle de coût, nous proposons un algorithme de provisionnement dans un seul site (SSVP) pour générer des plans de provisionnement à exécuter les SWfs dans un seul site de cloud. SSVP calcule d'abord un nombre optimal de cœurs de processeurs pour l'exécution de SWfs basé sur le modèle de coût multi-objectif. Ensuite, il génère un plan de provisionnement et améliore itérativement le plan de provisionnement afin de réduire le coût basé sur le modèle de coût et le nombre optimal de cœurs de processeurs. Enfin, SSVP génère un plan de provisionnement de VMs correspondant au coût minimum pour exécuter les SWfs avec l'importance spécifique de chaque objectif.

Nous avons réalisé des évaluations approfondies à comparer notre modèle de coût et celui de GraspCC. Nous avons exécuté le workflow SciEvol [137] avec différentes quantités de données d'entrée et de différentes importance du temps d'exécution, en déployant Chiron dans le site de l'Est du Japon du cloud d'Azure. Les résultats des expérimentations montrent que notre algorithme peut adapter les plans de provisionnement de VMs aux différentes configurations, *c.-à-d.* générer différents plans de provisionnement de VMs pour les différentes importances du temps d'exécution. SSVP génère de meilleurs plans de provisionnement $(53,6\%)$ par rapport à GraspCC. Les résultats révèlent également que notre modèle de coût est plus $(76,7\%)$ précis pour estimer le temps d'exécution et le coût monétaire par rapport à GraspCC, en raison de la prise en compte de la charge de travail séquentiel et le coût pour démarrer les VMs.

Ordonnancement multi-objectif de SWfs dans un cloud multisite

Nous résolvons le problème de l'ordonnancement des fragments de SWfs avec plusieurs objectifs, afin de permettre l'exécution de SWfs dans un cloud multisite avec une contrainte de données stockées. Nous imaginons que les données stockées dans un site spécifique ne peuvent pas être autorisées à être transférées vers d'autres sites en raison de la propriété ou grandes quantités de données, qui s'appelle la contrainte de données stockées. Dans ce travail, nous avons pris en compte les différents prix des VMs et les données stockées dans des sites différents.

Le modèle de coût multisite est une fonction pondérée composée du temps d'exécution et du coût monétaire basé sur le temps d'exécution et le coût monétaire souhaité par l'utilisateur. Toutefois, puisqu'il est difficile d'estimer le temps d'exécution et le coût monétaire globale pour exécuter un SWf dans un cloud multisite, nous proposons un modèle de coût multisite comme une combinaison du coût pour exécuter chaque fragment du SWf. Nous décomposons également le temps d'exécution et le coût monétaire souhaité d'un SWf en une combinasion de ceux de chaque fragment du SWf. Basé sur le temps d'exécution et le coût monétaire souhaité de chaque fragment du SWf, on peut estimer le coût pour exécuter chaque fragment du SWf sur un site prévu basé sur le modèle de coût correspondant à SSVP. Enfin, notre modèle de coût multisite peut estimer le coût global en considérant le coût à transférer des données à travers différents sites avec un plan d'ordonnancement.

Nous présentons deux algorithmes d'ordonnancement que nous avons adaptés, l'ordonnancement basé sur les localisations de données (LocBased) et l'ordonnancement gourmand de sites (SGreedy), et l'algorithme que nous proposons : l'ordonnancement gourmand des activités (ActGreedy). LocBased exploite DTM à partitionner les SWfs et attribue les fragments aux sites où les données d'entrée sont stockées. Cet algorithme ignore le coût monétaire et peut encourir un coût énorme à exécuter les SWfs. SGreedy prend l'avantage de la technique d'encapsulation des activités pour partitionner les SWfs et attribue le meilleur fragment à chaque site. Il attribue les activités d'un pipeline des activités aux sites différents, qui conduit à une grande transmission de données intersite et un temps d'exécution plus long. ActGreedy partitionne les SWfs avec la technique d'encapsulation des activités et regroupe de petits fragments en plus gros fragments pour réduire la transmission de données entre les différents sites et attribue chaque fragment au meilleur site. Cet algorithme permet de réduire le temps d'exécution global en comparant le coût pour exécuter des fragments dans chaque site, qui est généré basé sur SSVP.

Nous avons évalué notre algorithme d'ordonnancement en exécutant SciEvol avec différentes quantités de données d'entrée et les différentes importances des objectifs dans trois sites du cloud d'Azure. Les trois sites de cloud sont l'Ouest de l'Europe, l'Ouest du Japon, et l'Orient du Japon et les coûts d'utilisation de VMs sur chaque site sont différents. Nous avons utilisé SSVP pour générer des plans de provisionnement de VMs et Chiron pour exécuter des fragments du SWf dans chaque site. Les résultats des expérimentations montrent qu'ActGreedy fonctionne mieux en termes du coût pondéré pour exécuter les SWfs dans un cloud multisite par rapport à LocBased (jusqu'à 10.7%) et SGreedy (jusqu'à $17,2\%$). En outre, les résultats révèlent également que le temps d'ordonnancement d'ActGreedy est raisonnable par rapport aux deux approches générales, *c.-à-d.* Brut Force et Genetic.

Ordonnancement de tâches avec les données de provenance

Nous traitons le problème d'ordonnancement de tâches pour l'exécution de SWfs en multisite avec le soutien sur les données de provenance. L'objectif principal est de permettre l'exécution de SWfs avec les données d'entrée distribuées dans les différents sites dans un délai minimum avec le soutien sur les données de provenance, tandis que les bandes passantes entre les différents sites sont différentes. Dans ce travail, nous proposons Chiron Multisite et un algorithme d'ordonnancement de tâches.

Chiron Multisite est une extension de Chiron pour les environnements de cloud multisite. Chiron met en œuvre une approche algébrique pour exprimer les SWfs et optimiser l'exécution de SWfs dans un seul site. Chiron Multisite permet d'exécuter simultanément des tâches d'une activité sur des sites différents pour traiter les données distribuées.

Nous proposons aussi le modèle de provenance multisite pour Chiron Multisite. Dans un cloud multisite, nous proposons différentes méthodes pour le transfert de données intersite. Nous utilisons notre méthode d'ordonnancement à deux niveaux, *c.-à-d.* l'ordonnancement multisite et l'ordonnancement d'un seul site, pour l'ordonnancement de tâches dans un cloud multisite. Multisite Chiron correspond au parallélisme à grain fin au niveau multisite, qui permet aux différentes tâches d'un fragment d'être exécutées dans les différents sites de cloud.

Nous proposons un algorithme de l'ordonnancement multisite de tâches orientés-données (DIM) pour attribuer des tâches au niveau multisite en considérant le support des données de provenance, différentes bandes passantes entre les différents sites et la distribution de données d'entrée. D'abord, DIM attribue les tâches en fonction de localisations de données sur différents sites. Puis, il redistribue les tâches afin d'atteindre l'équilibre de charge entre les différents sites de cloud basés sur un modèle de coût pour estimer la charge d'exécution de chaque site. L'équilibre de charge représente qu'il faut exécuter les tâches dans le même temps dans chaque site. Le modèle de coût prend en compte le temps de transférer les données d'entrée des tâches entre les différents sites et le temps de transférer les données de provenance à une base de données centralisée.

Nous avons évalué nos algorithmes avec Chiron Multisite en exécutant Buzz et Montage [6] dans trois sites de cloud Azure, *c.-à-d.* US centrale, l'Ouest de l'Europe et le Nord de l'Europe. Nous avons exécuté buzz avec différentes quantités de données d'entrée et Montage avec différents degrés en utilisant le Chiron Multisite. Les résultats expérimentaux montrent que DIM est beaucoup mieux que deux algorithmes d'ordonnancement existants, *c.-à-d.* MCT (jusqu'à $24,3\%$) et OLB (jusqu'à $49,6\%$), en termes du temps d'exécution. DIM peut également réduire de manière significative (jusqu'à plus de 7 fois) les données transférées entre les sites, comparé avec MCT et OLB. En outre, les résultats révèlent également que le temps d'ordonnancement de DIM est raisonnable par rapport à la durée d'exécution globale de SWfs (moins de 3%). En particulier, les expérimentations montrent que la distribution de tâches est adaptée en fonction de différentes bandes passantes entre les différents sites pour la génération de données de provenance.

Conclusion

Dans cette thèse, nous avons traité le problème de l'exécution de SWfs orientés-données dans un cloud multisite, où les données et les ressources informatiques peuvent être distribués aux différents sites de cloud. Pour cette raison, nous avons proposé une approche distribuée et parallèle qui exploite les ressources disponibles dans les différents sites de cloud. Dans notre étude de l'état de l'art, nous avons proposé une architecture fonctionnelle de SWfMS en analysant et en catégorisant les techniques existantes. Pour exploiter le parallélisme, nous avons utilisé une approche algébrique, ce qui permet d'exprimer les activités de SWfs en utilisant des opérateurs à automatiquement les transformer en de multiples tâches. Nous avons proposé l'algorithmes de partitionnement de SWfs, un algorithme dynamique de provisionnement de SWfs dans un seul site, un algorithme d'ordonnancement multi-objectif dans un cloud multisite, un algorithme d'ordonnancement de tâches avec les données de provenance et Chiron Multisite. Différents algorithmes de partitionnement de SWfs partitionnent un SWf à plusieurs fragments. L'algorithme de provisionnement de VMs est utilisé pour créer dynamiquement une combinaison optimale de VMs pour exécuter des fragments d'un SWf dans chaque site de cloud. L'algorithme d'ordonnancement multi-objectif distribue les fragments d'un SWf aux sites de cloud avec le coût minimum basé sur un modèle de coût multi-objectif. L'algorithme d'ordonnancement de tâches distribue directement des tâches entre les différents sites de cloud tout en réalisant l'équilibrage de charge au niveau de chaque site basé sur un SWfMS multisite, Chiron Multisite. Chiron Multisite est une extension de Chiron pour exécuter les SWfs dans les environnements de cloud multisite. Nous avons évalué nos solutions proposées en exécutant des SWfs réels dans le cloud de Microsoft Azure. Nos résultats expérimentaux montrent les avantages de nos solutions par rapport aux techniques existantes.

Nos contributions peuvent être utilisées comme le point de départ pour la recherche future. Nous proposons les futures directions de recherche suivantes :

- **Distribution de la provenance**. La gestion de données de provenance distribuées peut réduire le temps de générer ou de récupérer des données de provenance dans chaque site afin de réduire le temps d'exécution global de SWfs dans un cloud multisite.

- **Transfert de données**. Une solution possible de transférer efficacement des données entre deux sites de cloud est de sélectionner plusieurs nœuds sur chaque site pour envoyer ou recevoir des données, en exploitant le transfert de données en parallèle et en faisant le transfert de données plus efficace.

- **Spark multisite**. Nos algorithmes et optimisations d'ordonnancement multisite pourraient être adaptées pour le framework Spark pour l'exécution de SWfs dans un cloud multisite.

- **Architecture** : Une architecture peer-to-peer peut être utilisée pour atteindre la tolérance aux pannes lors de l'exécution de SWfs dans un seul site de cloud ou un cloud multisite.

- **Ordonnancement dynamique**. L'ordonnancement dynamique des activités ou des tâches peut être mieux adapté à l'environnement d'exécution en considérant les paramètres mesurés lors de l'exécution de SWfs, *p. ex.* le coût monétaire des VMs, la bande passante pour transferer ou recevoir des données dans les VMs, etc.

Publications

Les contributions ont conduit aux publications suivantes :

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Daniel de Oliveira and Marta Mattoso. Multi-Objective Scheduling of Scientific Workflows in Multisite Clouds. Dans BDA'

2016 : Gestion de données - principles, technologies et applications, 2016. À paraître.

- Luis Pineda-Morales, *Ji Liu*, Alexandru Costan, Esther Pacitti, Gabriel Antoniu, Patrick Valduriez, and Marta Mattoso. Managing Hot Metadata for Scientific Workflows on Multisite Clouds. Dans IEEE International Conference on Big Data, 2016. À paraître.

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso. Scientific Workflow Scheduling with Provenance Support in Multisite Cloud. Dans 12th International Meeting on High Performance Computing for Computational Science (VECPAR), 2016, $1-8$.

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Daniel Oliveira, Marta Mattoso. Multi-objective scheduling of Scientific Workflows in multisite clouds. Dans Future Generation Computer Systems, 2016, volume 63, $76-95$.

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso. A Survey of Data-Intensive Scientific Workflow Management. Dans Journal of Grid Computing, 2015, volume 13, numéro 4, $457-493$.

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso, Parallelization of Scientific Workflows in the Cloud, Rapport de recherche RR-8565, 2014.

- *Ji Liu*, Vítor Silva, Esther Pacitti, Patrick Valduriez, Marta Mattoso. Scientific Workflow Partitioning in Multi-site Clouds. Dans BigDataCloud'2014 : 3rd Workshop on Big Data Management in Clouds in conjunction with Euro-Par, 2014. Springer, Lecture Notes in Computer Science, 8805, $105-116$.

- *Ji Liu*. Multisite Management of Data-intensive Scientific Workflows in the Cloud. Dans BDA'2014 : Gestion de données - principles, technologies et applications, 2014, $28-30$.

Contents

Chapter 1

Introduction

Scientific Workflows (*SWfs*) enable scientists to easily express multi-step computational activities, such as load input data files, process the data, run analyses, and aggregate the results. The computational activities are related by dependencies. A SWf describes activities and the dependencies typically as a graph, where vertexes represent data processing activities and edges represent dependencies between them. As the computation in scientific experiments becomes complex and analyzes big amounts of data, SWfs are widely used in multiple domains, such as astronomy [59], biology [137], physics [138], seismology [56], meteorology [190] and so on.

SWfs are often *data-intensive*, *i.e.* process, manage or produce huge amounts of data. Managing and manipulating data-intensive SWfs with traditional programming tools (*e.g.* code libraries, scripting languages) becomes very hard and impossible as complexity increases. Therefore, *SWf Management Systems* (*SWfMSs*) have been specifically developed to ease dealing with SWfs, which includes many aspects such as modeling, programming, debugging, and executing SWfs. SWfMSs generally generate provenance data during SWf execution. Provenance data, which traces the execution of SWfs and the relationship between input data and output data, is sometimes more important than SWf execution itself. In order to execute data-intensive SWfs within a reasonable time, SWfMSs exploit parallelism techniques with High Performance Computing (HPC) resources in a cluster, grid or cloud environment. Some existing SWfMSs, *e.g.* Pegasus [60, 61], Swift [201], and Chiron [139], are publicly available for SWf execution and management. However, most of them are designed for computing cluster or grid environments. In cloud environments, SWfMSs generally use the same approaches designed for computing clusters or grids, which are not optimized for cloud environments.

By offering virtually infinite resources, diverse scalable services, stable service quality and flexible payment policies, the *cloud* becomes an appealing solution for SWf execution. SWfMSs can be easily deployed in the cloud exploiting *Virtual Machines* (*VMs*). With a pay-as-you-go method, the users of clouds do not need to buy physical machines and the maintenance of the machines is ensured by cloud providers. Thus, cloud environments become interesting infrastructures for SWf execution.

A cloud is typically *multisite*, *i.e.* made of several sites (or data centers), each with its own resources and data and is explicitly accessible to cloud users. Because of low latency and proprietary issues, the data are generally stored at the cloud site where the data sources are located. For instance, the climate data in the Earth System Grid [189], large amounts of raw data from Quantum Chromodynamics (QCD) [149] and the data of the ALICE project [1] are geographically distributed. As a consequence, the input data of a SWf can be geographically distributed and SWf execution should be adapted to a multisite cloud while exploiting distributed computing or storage resources beyond one cloud site. The existing approaches focus on the computing cluster, grid or a single site cloud environment, which leave space for executing SWfs in multisite cloud environments.

1.1 Thesis Context

This thesis has been prepared in the context of two collaborative research projects: Z-CloudFlow and MUSIC (MUltiSite Cloud data management). The Z-CloudFlow project is supported by the Microsoft Research-INRIA joint center (France). It focuses on the data management of SWfs in the cloud. The goal of this project is to propose a framework to efficiently execute SWfs with large data volumes while leveraging the cloud infrastructure capabilities. MUSIC is a joint project between LNCC, COPPE/UFRJ and UFF (Brazil) and INRIA, focusing on a multisite cloud model where each site is visible from outside. The main objective of this project is to develop a multisite cloud architecture for processing, managing and analyzing scientific data, which can be heterogeneous data or complex big data, possibly using SWfs and SWfMSs. In this thesis, we use an algebraic SWfMS (Chiron) developed at COPPE/UFRJ.

We consider the problem of efficiently executing data-intensive SWfs in a multisite cloud, where the data and computing resources are distributed in different cloud sites. There are basically three challenges:

- How to execute SWfs with distributed data in the multisite cloud? The data can be distributed at different sites but may not be allowed to be moved to other sites because of large size or proprietary reasons. We call this the data location constraint. This data, which cannot be moved, can be input data or configuration data of a SWf. The input data is the data to be processed by SWfs. During SWf execution, intermediate data can be generated by processing the input data by one or several activities. The intermediate data, which is the input data of following activities, can be of large size and moved across multiple sites. Some configuration data located at specific sites are used for SWf execution. Thus, during SWf execution, the data location constraint should be considered for the scheduling of activities or tasks at multiple sites.

- How to deal with heterogeneous features of each cloud sites for SWf execution? Within one cloud site, the bandwidth between any two computing nodes may be similar while the bandwidth between two computing nodes located at different cloud sites may vary significantly. In addition, the cost to use VMs at different cloud sites can be very different. Thus, the challenge is how to schedule the execution of SWfs in order to reduce execution time and monetary cost with the consideration of these heterogeneous features in a multisite cloud.

- How to manage the VM provisioning in the cloud for SWf execution? A major difference between cloud and grid or cluster is that we can dynamically provision VMs before or during SWf execution in the cloud. However, the challenge of VM provisioning, *i.e.* how to decide the number and types of VMs for SWf execution in order to reduce both execution time and monetary cost, remains critical for SWf execution in the cloud.

In order to address these challenges, we deal with the following aspects:

- Partitioning of SWfs for multisite execution considering the data stored at each site while reducing execution time.
- Provisioning of VMs for SWf execution in the clouds in order to reduce both execution time and monetary cost.
- Scheduling of activities in a multisite cloud considering the distributed data and different costs of using VMs at different cloud sites while reducing execution time and monetary cost.
- Adapting a single site SWfMS to multisite, which can execute the tasks at different sites to process the distributed data.
- Scheduling tasks with provenance support and distributed data for a single activity while considering different bandwidths among different sites in order to reduce execution time.

1.2 Contributions

The main objective of this thesis is to efficiently execute data-intensive SWfs in a multisite cloud, where each site has its own cluster, data and programs. Our survey (see Chapter 2 on State of the Art) shows that most SWfMSs have been designed for computer clusters or grids, and some have been extended to operate in the cloud, but only for a single site. In order to achieve the objective, we propose a distributed and parallel approach that leverages the resources available at different cloud sites. To exploit parallelism, we use an algebraic approach, which allows expressing SWf activities using operators and automatically transforming them into multiple tasks.

The main techniques consist of SWf partitioning algorithms, a dynamic VM provisioning algorithm, an activity scheduling algorithm and a task scheduling algorithm. Different SWf partitioning algorithms partition a SWf to several fragments according to different cloud configurations. The VM provisioning algorithm is used to dynamically create an optimal combination of VMs for executing SWf fragments at each cloud site, based on a multi-objective cost model composed of execution time and monetary cost. The activity scheduling algorithm distributes the SWf fragments to the cloud sites with the minimum cost based on a multi-objective cost model, which combines both execution time and monetary cost. The task scheduling algorithm directly distributes tasks among different cloud sites while achieving load balancing at each site. This scheduling algorithm is based on a multisite SWfMS, which generates provenance data for multisite SWf execution using a centralized method. Our experiments show that our approach can reduce considerably the overall cost of SWf execution in a multisite cloud.

The contributions of thesis are:

- **A survey of techniques to execute data-intensive SWfs in a multisite cloud.** First, we define the important concepts, *e.g.* SWfs, SWfMSs. We propose a general functional architecture of SWfMSs and identify different parallelism techniques and scheduling approaches for SWf execution. We also present the parallelization techniques to execute SWfs in clouds. Furthermore, we analyze the features of different systems including frameworks and eight widely used SWfMSs. Finally, we propose some research issues for SWf execution in a multisite cloud.

- **A non-intrusive method to execute SWfs in a multisite cloud.** Most SWfMSs can be used in a single site cloud. However, some activities of a SWf may need to be executed at different specific cloud sites. To this end, we propose a non-intrusive method with three SWf partitioning techniques for SWf execution in a multisite cloud in order to reduce execution time. We consider using the existing VMs at each cloud site and do not change the VMs before or during SWf execution. The three partitioning techniques are based on scientific privacy, computing capacity and data transfer minimization respectively. With each partitioning technique, a SWf can be partitioned to several SWf fragments. Each fragment can be executed at a cloud site with a single site SWfMS. In addition, SWf fragments are scheduled by respecting all the data dependencies in the original SWf. The partitioning techniques are validated by executing the Buzz SWf in Microsoft Azure multisite cloud with a variation of the Chiron SWfMS. Our experiment results reveal that different partitioning techniques can reduce execution time for different cloud configurations.

- **A VM provisioning algorithm for SWf execution in a single site cloud.** The users of SWfMSs generally have multiple objectives to execute SWfs in a cloud, *e.g.* reducing execution time and monetary cost. In order to achieve multiple objectives without modifying SWfMSs and scheduling approaches, we propose a VM provisioning algorithm for single site SWf execution with a proposed cost model. This is a base for the SWf execution in a multisite cloud. The cost model is composed of monetary cost and execution time, with the consideration of sequential workload of SWf execution and the cost to initialize VMs in the cloud. Based on the cost model, we propose a VM provisioning algorithm (SSVP) in order to generate VM provisioning plans for SWf execution with the minimum cost. SSVP calculates an optimal number of virtual CPU cores for SWf execution and then generates a VM provisioning plan corresponding to the minimum cost to execute the SWf. SSVP is compared with an existing algorithm, *i.e.* GraspCC, by executing SciEvol using Chiron in the Azure cloud. The experimental results show that our proposed algorithm (SSVP) generates better (smaller cost) VM provisioning plans for different configurations of SWf execution compared with GraspCC.

- **A multi-objective general approach to executing SWfs in a multisite cloud.** In a multisite cloud, the configuration data of some activities may be stored at specific cloud sites. Because of the stored data, some activities can be just executed at the site where the configuration data is stored. In addition, the cost of using VMs at different cloud sites are different. In this situation, we propose a general multi-objective approach to executing SWfs in a multisite cloud with the stored data constraint. First, we propose a system model for multisite SWf execution with coarse-grained parallelism at the multisite level, *i.e.* one SWf fragment can only be executed at one cloud site. An activity can only be executed at a single cloud site with the coarse-grained parallelism. Then, we propose a multi-objective cost model for multisite SWf execution in the cloud. The cost model is also composed of monetary cost and execution time. Based on the multisite multi-objective cost model, SWf partitioning methods and the SSVP algorithm, we propose a multisite fragment scheduling algorithm (ActGreedy) and adapt two existing scheduling algorithms (LocBased and SGreedy) to multisite cloud environments. We validate our proposed scheduling algorithm by executing the SciEvol SWf with Chiron at three sites of the Azure cloud. The experimental results show that ActGreedy performs much better than LocBased and SGreedy in terms of the cost to execute SWfs in the multisite cloud.

- **Multisite Chiron.** Multisite Chiron is an extension of Chiron for multisite cloud environments. Chiron implements an algebraic approach to express SWfs, optimize SWf execution in a single cluster. Multisite Chiron enables task execution of an activity at different sites to process the distributed data simultaneously. We also propose the multisite provenance model for multisite Chiron. In a multisite cloud, we propose different data communication methods for multisite Chiron. We use our two level scheduling method, *i.e.* multisite scheduling and single site scheduling, for task scheduling in a multisite cloud. Multisite Chiron corresponds to fine-grained parallelism at the multisite level, which is different from the coarse-grained parallelism. The fine-grained parallelism enables different tasks of one activity to be executed at different cloud sites.

- **A multisite task scheduling (DIM) algorithm.** DIM is a multisite scheduling algorithm with the assumption that the provenance data is stored at a centralized site. DIM schedules tasks to different sites with the consideration of data location and different bandwidths among different sites for provenance data generation. In this work, we also propose a model to estimate the time to execute bags of tasks at a site. We use Buzz and Montage SWfs to validate our proposed algorithm using the multisite Chiron. The experimental results reveal that DIM is much better than two baseline algorithms in terms of execution time and intersite data transfer.

All these contributions have been published in the following publications:

- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Daniel de Oliveira and Marta Mattoso. Multi-Objective Scheduling of Scientific Workflows in Multisite Clouds. In BDA'2016: Gestion de données - principles, technologies et applications, 2016. To appear.
- Luis Pineda-Morales, *Ji Liu*, Alexandru Costan, Esther Pacitti, Gabriel Antoniu, Patrick Valduriez, and Marta Mattoso. Managing Hot Metadata for Scientific Workflows on Multisite Clouds. In IEEE International Conference on Big Data, 2016. To appear.
- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso. Scientific Workflow Scheduling with Provenance Support in Multisite Cloud. In 12th International Meeting on High Performance Computing for Computational Science (VECPAR), 2016, $1-8$.
- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Daniel Oliveira, Marta Mattoso. Multi-objective scheduling of Scientific Workflows in multisite clouds. In Future Generation Computer Systems, 2016, volume 63, $76-95$.
- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso. A Survey of Data-Intensive Scientific Workflow Management. In Journal of Grid Computing, 2015, volume 13, number 4, $457-493$.
- *Ji Liu*, Esther Pacitti, Patrick Valduriez, Marta Mattoso, Parallelization of Scientific Workflows in the Cloud, Research Report RR-8565, 2014.
- *Ji Liu*, Vítor Silva, Esther Pacitti, Patrick Valduriez, Marta Mattoso. Scientific Workflow Partitioning in Multi-site Clouds. In BigDataCloud'2014: 3rd Workshop on Big Data Management in Clouds in conjunction with Euro-Par, Aug 2014. Springer, Lecture Notes in Computer Science, 8805, $105-116$.
- *Ji Liu*. Multisite Management of Data-intensive Scientific Workflows in the Cloud. In BDA'2014: Gestion de données - principles, technologies et applications, 2014, $28-30$.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2: State Of The Art. This chapter is a survey of the existing techniques for SWf execution. First, it introduces a general definition of SWfs and SWfMSs, and presents the functional architecture of SWfMSs, the features, and techniques for data-intensive SWfs. Then, it presents parallelism techniques, including coarse-grained parallelism and fine-grained parallelism (data parallelism, independent parallelism, pipeline parallelism, and hybrid parallelism), and scheduling techniques, *i.e.* static scheduling, dynamic scheduling, and hybrid scheduling. Afterward, it focuses on the cloud environment for SWf execution including multisite management, data storage, and the techniques to execute SWfs in the cloud. Furthermore, it analyzes the features of different systems including frameworks and eight widely used SWfMSs. Finally, it analyzes the limitations of the existing approaches and proposes research directions for SWf execution in a multisite cloud.

Chapter 3: SWf Partitioning. In this chapter, we propose an approach to executing SWfs with SWf partitioning techniques in a multisite cloud. First, we propose a preliminary system model. Then, we present DAG partitioning, data partitioning, and a general DAG partitioning techniques. Afterward, we propose three DAG partitioning techniques, *i.e.* Scientist Privacy (SPr), Data Transfer Minimization (DTM) and Computing Capacity Adaptation (CCA), and a data refining technique composed of data combining and compression. We validate the techniques by executing a Buzz SWf with the Chiron SWfMS in the Azure multisite cloud. The results show that DTM performs better when all the cloud sites have the same amounts and types of VMs and that CCA is suitable for the environment where not all the cloud sites have the same amounts or types of VMs. The results also show that data refining technique can significantly reduce the data transfer time between two cloud sites.

Chapter 4: VM Provisioning for a single site cloud. In this chapter, we propose a VM provisioning approach for SWf execution in a single site cloud with multiple objectives, *i.e.* reducing execution time and monetary cost. We present our cost model and detail our proposed Single Site VM provisioning (SSVP) algorithm. SSVP considers the time to initialize VMs and the sequential part of the workload in SWf execution. Then, we validate the cost model and algorithm by executing SciEvol in Azure and compare SSVP with an existing approach. The results show that SSVP can generate better VM provisioning plans compared with the existing approach, *i.e.* GraspCC, with the different importance of objectives. In addition, the results show that our cost model is precise. Furthermore, the results reveal that because of our cost model, SSVP is sensitive to the different importance of objectives, which can generate better provisioning plans for different configurations.

Chapter 5: Multi-objective Fragment scheduling. In this chapter, we propose a multi-objective fragment scheduling algorithm for multisite SWf execution in a multisite cloud. First, we define the fragment scheduling problem with a stored data constraint and present the system architecture. Then, we show our multi-objective cost model for multisite SWf execution in the cloud. Afterward, we propose the fragment scheduling algorithms including two adapted scheduling algorithms, *i.e.* Data Location Based Scheduling (LocBased) and Site Greedy Scheduling (SGreedy), and our proposed scheduling algorithm, namely Activity Greedy Scheduling (ActGreedy). Finally, we validate our proposed scheduling algorithm by executing the SciEvol SWf at three sites of the Azure cloud. The results show that ActGreedy corresponds to less cost compared with LocBased and SGreedy and that the scheduling time of our proposed algorithm is reasonable.

Chapter 6: Task Scheduling with Provenance Support. In this chapter, we propose a task scheduling approach and the Multisite Chiron. First, we define the task scheduling problem and present multisite Chiron, including the architecture and the provenance model for multisite SWf execution with a centralized provenance database. Then, we propose our task scheduling algorithm, *i.e.* Data-Intensive Multisite task scheduling (DIM), which considers the time to transfer intersite data, including input data of activities and provenance data. In addition, DIM can achieve load balance of each site in order to reduce overall execution. We validate DIM based on multisite Chiron by executing Buzz and Montage in the Azure cloud with three sites. The experimental results reveal that our scheduling algorithm performs much better in terms of both execution time and the amounts of intersite data transfer compared with two existing algorithms.

Chapter 7: Conclusion. In this last chapter, we summarize our contributions, discuss the limitations, and point out the future research directions.

Chapter 2

State of the Art

Nowadays, more and more computer-based scientific experiments need to handle massive amounts of data. Their data processing consists of multiple computational steps and dependencies within them. A data-intensive scientific workflow (SWf) is useful for modeling such process. Since the sequential execution of data-intensive SWfs may take much time, Scientific Workflow Management Systems (SWfMSs) should enable the parallel execution of data-intensive SWfs and exploit the resources distributed in different infrastructures such as grid and cloud. This chapter provides a survey of data-intensive SWf management in SWfMSs and their parallelization techniques. This chapter is based on [120][119].

Section 2.2 gives an overview of SWf management, including system architectures and basic functionality. Section 2.3 focuses on the techniques used for parallel execution of SWfs. Then, Section 2.4 details the cloud computing including file system, multisite management in the cloud and the adaptation of SWfMSs to a multisite cloud environment. Afterwards, Section 2.5 presents the recent frameworks for parallelization, eight SWfMSs and a science gateway to execute SWfs. Finally, Section 2.6 summarizes the main findings of this study and discusses the open issues raised for executing data-intensive SWfs in a multisite cloud.

2.1 Overview and Motivations

Many large-scale scientific experiments take advantage of SWfs to model data operations such as loading input data, data processing, data analysis, and aggregating output data. SWfs allow scientists to easily model and express the entire data processing steps and their dependencies, typically as a directed graph or Directed Acyclic Graph (DAG). As more and more data is consumed and produced in modern scientific experiments, SWfs become data-intensive. In order to process large-scale data within a reasonable time, they need to be executed with parallel processing techniques in the grid or the cloud.

A SWf Management System (SWfMS) is an efficient tool to execute workflows and manage data sets in various computing environments. A SWfMS gateway framework is a system for SWfMS users to execute SWfs with different SWfMSs. Several SWfMSs, *e.g.* Pegasus [60, 61], Swift [201], Kepler [21], Taverna [141], Galaxy [82], Chiron [139] and SWfMS gateway frameworks such as WS-PGRADE/gUSE [105] are now used intensively by various research communities, *e.g.* astronomy, biology, computational engineering. Although many SWfMSs exist, the architecture of SWfMSs have common features, in particular, the capability to produce a Workflow Execution Plan (WEP) from a high-level workflow specification. Most SWfMSs are composed of five layers, *e.g.* presentation layer, user services layer, WEP generation layer, WEP execution layer and infrastructure layer. These five layers enable SWfMSs users to design, execute and analyze data-intensive SWfs throughout the workflow lifecyle.

Since the sequential execution of data-intensive SWfs may take much time, SWfMSs should enable the parallel execution of data-intensive SWfs and exploit large amounts of distributed resources. Executable tasks can be generated based on diverse types of parallelism and submitted to the execution environment according to different scheduling approaches.

The ability to exploit large amounts of computing and storage resources for SWf execution is provided by cluster, grid and cloud computing. Grid computing enables access to distributed, heterogeneous resources using web services. These resources can be data sources (files, databases, web sites, etc.), computing resources (multiprocessors, supercomputers, clusters) and application resources (scientific applications, information management services, etc.). These resources are owned and managed by the institutions involved in a virtual organization.

Cloud computing is the latest trend in distributed computing and has been the subject of much hype. The vision encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage and networking resources. Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, system administration, or application deployment, to very large data centers operated by cloud providers. Since the resources are accessed through services, everything gets delivered as a service. Thus, as in the services industry, this enables cloud providers to propose a pay-as-you-go pricing model, whereby users only pay for the resources they consume. A cloud is typically made of several sites (or data centers), each with its own resources and data. Thus, in order to use more resources than available at a single site or to access data at different sites, SWfs could also be executed in a distributed manner at different sites.

There have been a few surveys of techniques for SWfMSs. Some [33] provide an overview of parallelization techniques for SWfMSs, including their implementation in real systems, and discuss major improvements to the landscape of SWfMS. Some other work [195] examines the existing SWfMSs designed for grid computing, and proposes taxonomies for different aspects of SWfMSs, including workflow design, information retrieval, workflow scheduling, fault tolerance and data movement. In this chapter, we provide a survey of data-intensive SWf management in SWfMSs and their parallelization techniques and we focus on the following points:

1. A SWfMS functional architecture, which is useful to discuss the techniques for data-intensive SWfs. This architecture can also be a baseline for other work and

help with the assessment and comparison of SWfMSs.

2. A taxonomy of SWf parallelization techniques and SWf scheduling algorithms, and a comparative analysis of the existing solutions.

3. A discussion of research issues for improving the execution of data-intensive SWfs in a multisite cloud.

2.2 Scientific Workflow Management

This section introduces SWf management, including SWfs and systems. First, we define SWfs and SWfMSs. Then, we detail the functional architecture and the corresponding functionality of SWfMSs. Finally, we discuss the features and techniques for data-intensive SWfs used in SWfMSs.

2.2.1 Basic Concepts

A SWfMS manages a SWf all along its life cycle. This section introduces the concepts of SWfs, SWf life cycle, SWfMS and illustrates with SWf examples.

2.2.1.1 Scientific Workflows

A workflow is the automation of a process, during which data is processed by different logical data processing activities according to a set of rules. Workflows can be divided into business workflows and SWfs. Business workflows are widely used for business data processing. According to the workflow management coalition, a business workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [43]. A business process is a set of one or more linked procedures or activities that collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships [43]. Business workflows make business processes more efficient and more reliable.

Different from business workflows, SWfs are typically used for modeling and running scientific experiments. SWfs can assemble scientific data processing activities and automate the execution of these activities to reduce the makespan, which represents the entire SWf execution time. A SWf is the assembly of complex sets of scientific data processing activities with data dependencies between them [57]. A SWf may contain one or several sub-workflows. A sub-workflow is composed of a subset of activities and data dependencies in the SWf while representing a step to process data. SWfs can be represented in different ways. The most general representation is a directed graph, in which nodes correspond to data processing activities and edges represent the data dependencies. But most often, a SWf is represented as a DAG or even as a sequence (pipeline) of activities which is sufficient for many applications. Directed Cyclic Graphs (DCG) are harder to support since iteration is needed to represent repeated activities, *e.g.* with a whiledo construct [195].

Although business workflows and SWfs have some similarities, they are quite different. The first difference is the abstraction level. Business workflows take advantage of traditional programming languages while SWfs exploit higher abstraction level tools to prove a scientific hypothesis [25]. The second difference is the interaction with participants. In business workflows, data can be processed by different participants, which can be data processing machines or humans. In SWfs, data is processed only by machines while the scientists just need to monitor the workflow execution or control execution when necessary. The interaction of humans during the execution of SWfs is much less than that of business workflows. The third difference lies in the data flows and control flows [194]. Business workflows focus on procedural rules that generally represent the control flows while SWfs highlight data flows that are depicted by data dependencies [25]. This is reasonable since scientific experiments may need to deal with big experimental data. A data-intensive SWf processes, manages or produces huge amounts of data during execution. In addition, SWfs must be fully reproducible [25], which is not necessary for business workflows.

An activity is a description of a piece of work that forms a logical step within a SWf representation. In a SWf, an activity defines the associated data formats and data processing methods but requires associated data and computing resources to carry out execution. The associated data in an activity consists of input data and configurable parameters. When the configurable parameters are fixed and the input data is provided, the execution of a workflow activity is represented by several tasks. A task is the representation of an activity within a one-time execution of this activity, which processes a data chunk. An activity can correspond to a set of tasks for different parts of input data. Sometimes, "jobs" are used to represent the meaning of tasks [33] or activities [38, 61].

2.2.1.2 Scientific Workflow Life Cycle

The life cycle of a SWf is a description of the state transitions of a SWf from creation to completion [57, 86]. A SWf life cycle generally contains four phases. Görlach *et al.* [86] propose that a SWf life cycle contains modeling phase, deployment phase, execution and monitoring phase, and analysis phase. Deelman *et al.* [57] argue that a SWf life cycle consists of composition phase, mapping phase, execution phase and provenance phase. Provenance data represents information regarding workflow execution [74]. We present provenance in more details in the next section. In [130], a provenance database is proposed to represent and relate data from several phases of the workflow life cycle. In this chapter, we adopt a combination of workflow life cycle views [57, 86, 130] with a few variations, condensed in four phases:

1. The composition phase [57, 130] is for the creation of an abstract SWf. An abstract SWf is defined by the functionality of each activity (or sub-workflow) and data dependencies between activities (or sub-workflows) [87, 174]. SWf composition

can be done through a textual or Graphical User Interface (GUI). SWfMS users can reuse the existing SWfs with or without modification [94].

2. The deployment phase [86] is for constructing a concrete SWf, which consists of concrete methods (and associated codes) for each functional activity (or sub-workflow) defined in the composition phase, so that the SWf can be executed.
3. The execution phase [57, 130] is for the execution of SWfs with associated data, during which input data is processed and output data is produced.
4. The analysis phase [86, 130] is to apply the output data to scientific experiments, to analyze SWf provenance data and to share the SWf information.

2.2.1.3 Scientific Workflow Management Systems

A Workflow Management System (WfMS) is a system that defines, creates, and manages the execution of workflows. A WfMS is able to interpret the workflow process definition typically in the context of business applications. A SWfMS is a WfMS that handles and manages SWf execution. It is powerful tool to execute SWfs in a SWf engine, which is a software service that provides the runtime environment for SWf execution [24]. In order to execute a SWf in a given environment, a SWfMS typically generates a Workflow Execution Plan (WEP), which is a program that captures optimization decisions and execution directives, typically the result of compiling and optimizing a workflow, before execution.

To support SWf analysis, SWfMS should support additional functionality such as provenance. SWf provenance may be as (or more) important as the scientific experiment itself [74]. Provenance is the metadata that captures the derivation history of a dataset, including the original data sources, intermediate datasets, and the SWf computational steps that were applied to produce this dataset [46, 50, 84, 95]. Provenance data is used for SWf analysis and SWf reproducibility.

2.2.1.4 Scientific Workflow Examples

SWfs have been used in various scientific domains. In the astronomy domain, Montage[1] is a computing and data-intensive application that can be modeled as a SWf initially defined for the Pegasus SWfMS. This application is the result of a national virtual observatory project that stitches tiles of images of the sky from diverse sky surveys into a photorealistic single image [59]. Montage is able to handle a wide range of astronomical image data including the Two Micron All Sky Survey, (2MASS[2]), the Digitized Palomar Observatory Sky Survey, (DPOSS[3]), and the Sloan Digital Sky Survey (SDSS[4]) [100]. Each

[1] Montage project: http://montage.ipac.caltech.edu/

[2] 2MASS: http://www.ipac.caltech.edu/2mass/

[3] DPOSS: http://www.astro.caltech.edu/~george/dposs/

[4] SDSS: http://www.sdss.org/

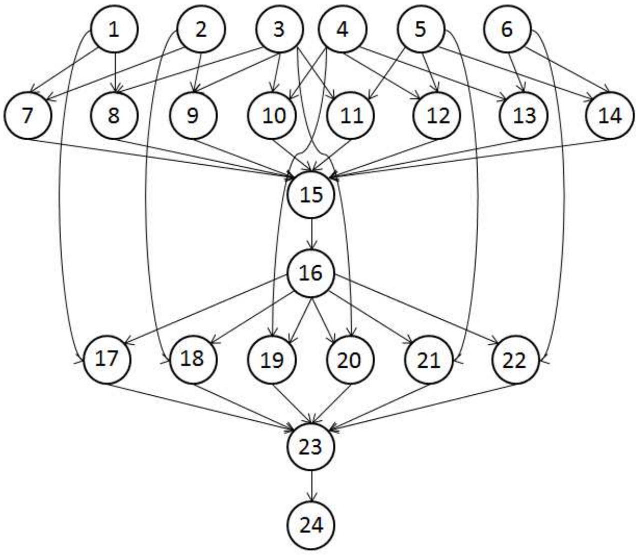


Figure 2.1: **The structure of a small Montage workflow [167].**

survey possesses huge amounts of data and covers a corresponding part of sky in visible wavelengths or near-infrared wavelengths. 2MASS has roughly 10 terabytes, DPOSS has roughly 3 terabytes and SDSS contains roughly 7.4 terabytes. All the data can be downloaded from a corresponding server at the aforementioned links and then staged into the execution environment, such as a shared-disk file system or a database, in order to be processed.

The structure of a small Montage workflow (at task level) is shown in Figure 2.1, where each node represents a task. The number within a node represents the name of a task in the SWf. The tasks at the same line represent one activity. The first activity (Tasks $1-6$) has no parent activities. Each of them exploits an mProject program to project a single image to the scale defined in a pseudo-FITS header template file. The second activity (Tasks $7-14$) utilizes an mDiffFit program to create a table of image-to-image difference parameters. The third activity (Task 15) takes advantage of an mFitplane program to fit the images generated by former activities ($7-14$) to an image. The fourth activity (Task 16) uses an mBgModel program to interactively determine a set of corrections to apply to each image to achieve a "best" global fit according to the image-to-image difference parameter table. The fifth activity (Tasks $17-22$) removes a background from a single image through an mBackground program. The sixth activity (Task 23) employs an mImgtbl program to extract the FITS header information (information about one or more scientific coordinate systems that are overlaid on the image itself) from a set of files and to create an ASCII image metadata table. Finally, the seventh activity (Task 24) pieces together the projected images using the uniform FITS header template and the information from the same metadata table generated by Task 23. This activity applies an mAdd program.

In the bioinformatics domain, SciEvol [137] is a SWf for molecular evolution reconstruction that aims at inferring evolutionary relationships on genomic data. To be executed in the Chiron SWfMS, SciEvol consists of 12 activities as shown in Figure 2.2. The first

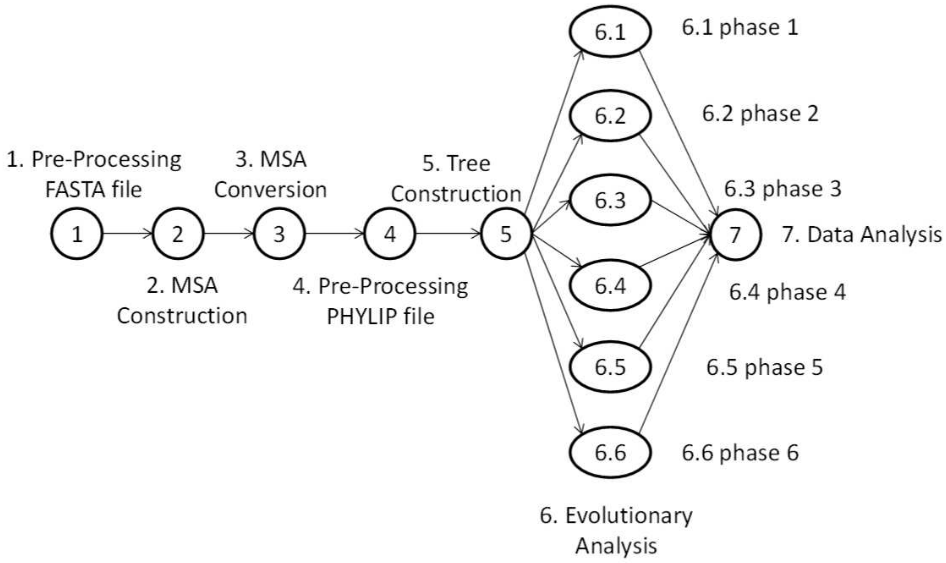


Figure 2.2: **SciEvol workflow [137].**

activity (pre-processing FASTA file) is a Python script to format the multi-fasta input file. FASTA file is a textual presenting format for nucleotide or peptide sequences. The second activity (MSA construction) constructs a Multiple Sequence Alignment (MSA) using a MAFFT program (or other MSA programs). A MAFFT program is generally for generating the alignment of three or more biological sequences (protein or nucleic acids) of similar length. The third activity (MSA conversion) executes ReadSeq to convert the MSA in FASTA format to that in PHYLIP format, which is used in the phylogenetic tree construction activity. The fourth activity (pre-processing PHYLIP file) formats the input file (referenced as “phylip-file-one”) according to the format definition and generates a second file (referenced as “phylip-file-two”). The fifth activity (tree construction) receives the “phylip-file-one” as input and produces a phylogenetic tree [72] as output. The sixth activities (evolutionary analysis from 6.1 to 6.6) analyze the phylogenetic tree with corresponding parameters and generate a set of files containing evolutionary information as output. Each of the activities (evolutionary phases) is related to one of six codon substitution models, which are used to verify if the groups of genes are under positive Darwinian selection. These activities exploit the same program using different parameters. The last activity (data analysis) automatically processes the output files obtained from the previous activities.

There are many other data-intensive SWfs in bioinformatics. For instance, SciPhylomics [53] is designed for producing phylogenomic trees based on an input set of protein sequences of genomes to infer evolutionary relationships among living organisms. SciPPGx [63] is a computing and data-intensive pharmacophylogenomic analysis SWf for providing thorough inferring support for pharmacophylogenomic hypotheses. SciPhy [136] is used to construct phylogenetic trees from a set of drug target enzymes found in protozoan genomes. All these bioinformatics SWfs have been executed using SciCumulus

SWfMS [51].

The components of SWfs can be classified by their functionality, motifs, or structure patterns. The functionality can be data processing, activity scheduling, activity execution, and resource management [21]. The motifs may be data-oriented and workflow-oriented. Data-oriented motifs consist of recurring activities such as data storage [24], data analysis, data cleaning, data moving [24] and data visualization. Workflow-oriented motifs may correspond to remote invocations, repetitive activities, parameter sweep workflows and meta-workflows [24, 78]. A parameter sweep workflow is a workflow with multiple input parameter sets, which needs to be executed for each input parameter set [41, 87]. A meta-workflow is a workflow composed of sub-workflows. Workflow structure patterns can be patterns for parallelization, *e.g.* representing SWfs as algebraic expressions [139], or component structure patterns, *e.g.* single activity with one or more input/output dependencies, sequential control and sequential/concurrent data, synchronization of sequential data, data duplication [194]. Moreover, similar structure patterns of SWfs can be found based on a similarity model of nodes and edges in the workflow DAG [27]. Identifying SWfs or workflow components of the same type enables workflow information sharing and reuse (see Section 2.2.2.2) among workflow designers [194].

2.2.2 Functional Architecture of SWfMSs

The functional architecture of a SWfMS can be layered as follows [60, 201, 21, 139]: presentation, user services, WEP generation, WEP execution and infrastructure. Figure 2.3 shows this architecture. The higher layers take advantage of the lower layers to realize more concrete functionality. A user interacts with a SWfMS through presentation and realizes the desired functions at user services layer. A SWf is processed at WEP generation layer to produce a WEP, which is executed at the WEP execution layer. The SWfMS accesses the physical resources through the infrastructure layer for SWf execution. The combination of WEP generation layer, WEP execution layer and infrastructure layer corresponds to a SWf execution engine.

2.2.2.1 Presentation Layer

The presentation layer serves as a User Interface (UI) for the interaction between users and SWfMSs at all stages of the SWf life cycle. The UI can be textual or graphical. This interface is responsible for designing a SWf by assembling data processing activities linked by dependencies. This layer also supports the functionality of showing execution status, expressing SWf steering and information sharing commands.

The language for the textual interface is largely used for designing SWfs in SWfMSs. Different from batch scripts, the textual language supports parallel computations on distributed computing and storage resources. The configuration or administration becomes complicated in this environment while the language defined by a SWfMS should be easy to use. Most SWfMS languages support the specification of a SWf in a DAG structure while some SWfMS languages also support iteration for DCG.

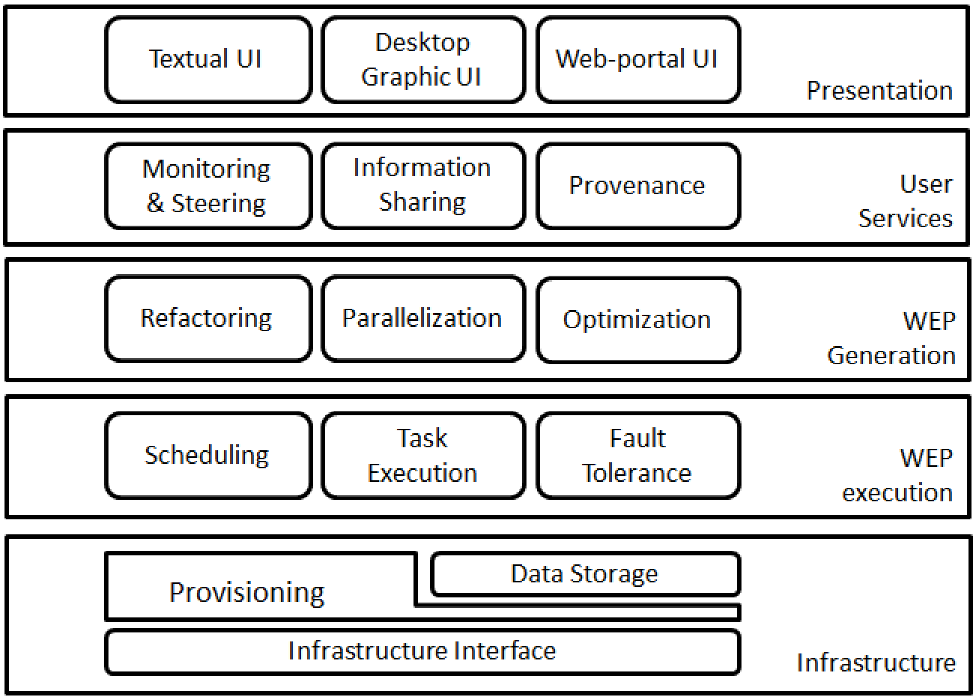


Figure 2.3: **Functional architecture of a SWfMS.**

Wilde *et al.* [188] propose a distributed parallel scripting language called Swift. Swift supports SWf specifications in both DAG and DCG. It is a C-like syntax that describes data, data flows and applications by focusing on concurrent execution, composition and coordination of independent computational activities. Variables are used in Swift to name the local variables, arguments, and returns of a function. The variables in Swift have three types: primitive, mapped, and collection. Primitive variables have the basic data structures such as integer, float, string, boolean and array. Mapped variables refer to files external to the Swift script. Collection variables are in the structures that contain a set of variables, such as arrays. Swift operations have three categories: built-in functions, application interface functions and compound functions. Built-in functions are implemented by the Swift runtime system to perform various utility functions such as numeric conversion, string manipulation, etc. An application interface function provides the information to the Swift runtime system to invoke a program. A compound function is a function that invokes other functions.

Pegasus uses Wings to create SWfs [81]. The SWfs are created through three stages in Pegasus/Wings: the first stage specifies the abstract structure of the SWf and creates a SWf template; the second stage specifies what data to be used in the SWf and creates a SWf instance; the third stage specifies the data replicas and their locations to form an executable SWf. The later stage is done by Pegasus while the first two are realized by Wings. In Wings, workflow and its activities are represented as semantic objects. The programs are represented as SWf components to process data, which is represented as individual files or file collections. An activity is represented as a node that contains a set of computations, which may contain one computation component or a collection of computations. The data dependencies are represented as links to carry data from or to a SWf node. After the presentation of programs, activities and data dependencies, a SWf template is created. With the binding of input data sets, a SWf instance is generated as a DAG in XML format. Then, Pegasus automatically maps the SWf instance to distributed computing nodes to form an executable workflow and manages SWf execution.

Chiron [139] also represents the SWfs activities and dependencies as a DAG in XML textual format. Ogasawara *et al.* [138] propose an algebraic language implemented in Chiron to encapsulate the SWf activities in six operators: Map, SplitMap, Reduce, Filter, SRQuery and JoinQuery. The Map operator consumes and produces a basic data chunk, which represents the data chunk that has a smallest amount of data while it contains all the necessary data to be processed in an activity. The SplitMap operator consumes a basic data chunk while it produces several basic data chunks. The Reduce operator reduces several basic data chunks to one basic data chunk. The Filter operator removes useless data chunks. SRQuery and MRQuery are traditional relational algebra expressions. Each activity corresponds to an operator. These operators are able to parallelize SWf execution onto the distributed computation resources.

Taverna utilizes a simple conceptual unified flow language (Scufl) to represent SWfs [141]. Scufl is an XML-based language, which consists of three main entities: processors, data links, and coordination constraints. Processors represent a computational activity in a SWf. Data links and coordination constraints separately represent the data dependencies and control dependencies between two activities.

SWfMSs such as Galaxy [82], Taverna [141] and Kepler [21] offer a GUI for SWf design. The GUI simplifies the designing process of a SWf for the SWfMS users to assemble the components described as icons through drag-and-drop functionality. Graphical SWfMSs combine the efficiency of SWf design and the ease of SWf representation. Desktop-based graphical SWfMSs are typically installed either in a local computer or in a remote server that is accessible through network connection. The local computer or remote server can be connected to large computing and storage resources for large-scale SWf execution. Some graphical SWfMSs such as Galaxy are web-portal-based, which makes it easy to share SWf information among the SWfMS users. With these SWfMSs, a SWf is generally designed in a browser on the client side but executed in a private or public web server. Some of the graphical SWfMSs take textual languages as inner representation of a SWf. For instance, Taverna utilizes Scufl within the SWfMS while Galaxy represents workflows in JSON format [16].

2.2.2.2 User Services Layer

The user services layer is responsible for supporting user functionality, *i.e.* SWf monitoring and steering, SWf information sharing and providing SWf provenance data.

SWf monitoring makes it possible to get real-time execution status for SWfMS users. Since SWf execution may take a long time, dynamic monitoring and steering of the execution are important to control SWf execution [57]. SWf monitoring tracks the execution status and displays this information to users during SWf execution [43]. Through SWf monitoring, a scientist can verify if the result is already enough to prove her hypothesis [46]. SWf monitoring remains an open challenge as it is hard to fully support. However, it can be achieved based on log data (in log files) or more general provenance data, typically in a database [129]. Gunter *et al.* [89] and Samak *et al.* [159] propose the Stampede monitoring infrastructure for real-time SWf monitoring and troubleshooting. This infrastructure takes a common data model to represent SWf execution and utilizes a high-performance loader to normalize the log data. It offers a query interface for extracting data from the normalized data. It has been initially integrated with Pegasus SWfMS and then adapted in Triana SWfMS [181]. Horta *et al.* [95] propose a provenance interface to describe the production and consumption relationships between data artifacts such as output data files and computational activities at runtime for SWf monitoring. This interface can be used to select the desired output data to monitor the SWf execution for SWfMS users through browsers or a high-resolution tiled display. This interface is based on on-line provenance query supported by algebraic approach. The on-line provenance query is different from the provenance collected at runtime, but made available only after the execution, where monitoring is no longer possible. This interface is available for Chiron [139] or SciCumulus [51] that store all the provenance data in a relational database. SciCumulus is an extension of Chiron for cloud environments.

SWf steering is the interaction between a SWfMS and a user to control the workflow execution progress or configuration parameters [129]. Thus, through SWf steering, a scientist can control SWf execution dynamically so that she does not need to continue unnecessary execution or execute a SWf again when an error occurs [46, 84]. SWf steering, which still remains an open issue, saves much time for SWfMS users.

Information sharing functionality enables SWf information sharing for SWf reusing. Through SWf information sharing, SWfMS users of the same SWfMS environments or different SWfMS environments can share SWf information including SWf design, the input data or the output data. Since designing a SWf is challenging work, SWf information sharing is useful to reduce repetitive work between different scientist groups. A SWfMS can directly integrate SWf repositories to support SWf information sharing. A SWf repository is a SWf information pool, where SWf data (input data and output data), SWf designs (structures) and available programs (to be used in activities of workflows) are stored. The SWf repository can contain shared SWfs for the same SWfMS environments, *e.g.* " the myExperiment" social network [191] for Taverna, the web-based sharing functionality of Galaxy [82], the SWf hosting environment for Askalon [91], or for different SWfMS environments, *e.g.* SHIWA Repository [174]. The SWf repositories should support SWf uploading, publishing, download or searching. The SWfs for different SWfMS environments can be adapted to an intermediate representation [151] to compose a meta-workflow, which can be executed by execution platforms such as SHIWA [175], with corresponding SWfMS engines for the sub-workflows. Except for SHIWA, the information sharing indicates SWf information sharing among the users of the same SWfMS environment.

Provenance data in SWfs is important to support reproducibility, result interpretation and problem diagnosis. Provenance data management concerns the efficiency and effectiveness of collecting, storing, representing and querying provenance data. Different methods have been proposed for different SWfMSs. Gadelha *et al.* [76] develop MTCProv, a provenance component for the Swift SWfMS. Swift optionally produces provenance information in its log files while this data is exported to relational databases by MTCProv. MTCProv supports a data model for representing provenance and provides a provenance query interface for the visualization of provenance graphs and querying of provenance information. Kim *et al.* [108] present a semantic-based approach to generate provenance information in the Wings/Pegasus framework. Wings is a middleware that supports the creation of SWf templates and instances, which are then submitted to Pegasus. This approach produces activity-level provenance through the semantic representations used in Wings, and execution provenance through Pegasus' task scheduling and execution process. SPARQL (SPARQL Protocol and RDF Query Language), a semantic query language, is used for querying provenance data. Costa *et al.* [46] propose PROV-Wf, a practical approach for capturing and querying provenance data for SWfs. PROV-Wf gathers provenance data in different granularities based on PROV recommendation [26]. The PROV-Wf contains three main parts: the structure of the experiment, execution of the experiment and environment configuration. PROV-Wf supports prospective and retrospective provenance data allowing for on-line provenance queries through SQL. The provenance database of this approach acts as a statistics catalog from DBMS. Altintas *et al.* [19] present a provenance information collection framework for Kepler. This framework can collect provenance information thanks to its implementation of event listener interfaces. Moreover, Crawl *et al.* [48] introduce a provenance system that manages provenance data from Kepler. This system records both data and dependencies of tasks executing on the same computing node. The provenance data is stored in a MySQL database. The Kepler Query API is used to retrieve provenance information and to display provenance graphs of SWf execution.

2.2.2.3 WEP Generation Layer

The WEP generation layer is responsible for generating a WEP according to a SWf design as shown in Figure 2.4. This layer contains three processes, *i.e.* SWf refactoring, SWf parallelization and optimization.

The SWf refactoring module refines the SWf structure for WEP generation. For instance, Ogasawara *et al.* [138, 139] take advantage of a workflow algebra to generate equivalent expressions, which are transformed into WEPs to be optimized. When a SWf representation is given, it is generally not adapted for an execution environment or a SWfMS. Through SWf refactoring, a SWfMS can transform the SWf into a simpler one, *e.g.* by removing redundant or useless activities, and partition it into several pieces, called fragments (by analogy with program fragments), to be executed separately by different nodes or sites. A SWfMS can schedule fragments to reduce scheduling complexity [38]. Thus, SWf partitioning is the process of decomposing a SWf into (connected) SWf fragments to yield distributed [122] or parallel processing. A SWf fragment (or fragment for short) can be defined as a subset of activities and data dependencies of the original SWf (see [138] for a formal definition). Note that the term SWf fragment is different from the term sub-workflow, although they are sometimes confused. However, the term sub-workflow is used to refer to the relative position of a SWf in a SWf composition hierarchy [180]. SWf partitioning is addressed in [38] for multiple execution sites (computer clusters explained in Section 2.2.2.5) with storage constraints. A method is proposed to partition a big SWf into small fragments, which can be executed in an execution site with moderate storage resources. In addition, Deelman *et al.* [60] propose an approach to remove SWf activities for SWf refactoring. This approach reduces redundant computational activities based on the availability of the intermediate data produced by previous execution. Tanaka and Tatebe [171] use a Multi-Constraint Graph Partitioning (MCGP) algorithm [107] to partition a SWf into fragments, which has equal weight value in each dimension while the weight of the edges crossing between fragments is minimized. In this method, each activity is defined as a vector of multiple values and each dependency between different activities has a value. This method balances the activities in each fragment while minimizing associated edges between different fragments. Moreover, SWf refactoring can also reduce SWf structure complexity. Cohen-Boulakia *et al.* [44] present a method to automatically detect over-complicated structures and replace them with easier equivalent structures to reduce SWf structure complexity.

SWf parallelization exploits different types of parallelism to generate concrete executable tasks for the WEP. The parallelization can be performed at sub-workflow level and activity, task level. The parallelization at sub-workflow level is realized by executing different sub-workflows in corresponding SWf execution engines in parallel. The parallelization at activity or task level encapsulates the related data, *i.e.* input, instruction and parameter data, into a task; Then, an activity may correspond to several tasks that can be executed in parallel. Swift [201], Pegasus [60], Chiron [139] and some other SWfMSs can achieve SWf parallelization using Message Passing Interface (MPI) [169] (or an MPI-like language) or a middleware within their execution engine. Since they have full control over the parallel SWf execution, these SWfMSs can leverage parallelism at different levels and yield the maximum level of performance. Some other SWfMSs outsource parallelization and SWf scheduling (see Section 2.2.2.4) to external execution tools, *e.g.* web services or Hadoop MapReduce systems. These SWfMSs can achieve activity parallelism but data parallelism (see Section 2.3.1) is generally realized in the external execution tools. The SWfMSs that outsource parallelization to a Hadoop MapReduce system adapt a data analysis process to a MapReduce workflow, composed of Map and Reduce activities. These SWfMSs generate corresponding MapReduce tasks and submit the tasks to the MapReduce system. Wang *et al.* [183] propose an architecture that combines the Kepler SWf engine with the Hadoop MapReduce framework to support the execution of MapReduce workflows. Delegating parallelization and parallel execution to an external engine makes it easy for the SWfMS to deal with very large data-intensive tasks. However, this approach is not as efficient as direct support of parallelism in the SWfMS. In particular, it makes the SWfMS loose control over the entire SWf execution, so that important optimizations, *e.g.* careful placement of intermediate data exchanged between tasks, cannot be realized.

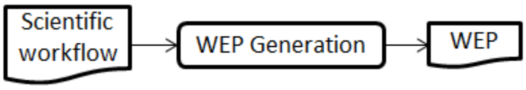


Figure 2.4: **WEP generation.**

Furthermore, provenance management becomes almost impossible as the external tools typically do not support provenance.

SWf optimization captures the results of SWf refactoring and SWf parallelization and inserts additional instructions for SWf scheduling to generate a WEP. The additional instructions describe multiple objectives for SWf execution, such as minimizing execution time, meeting security restrictions and reducing resource cost. The multiple objectives are mainly attained by adjusting SWf scheduling at the WEP execution layer. Having an algebra and dataflow-oriented execution engine opens up interesting opportunities for optimization [64, 41]. For example, it allows for user interference on the execution plan, even during the execution.

2.2.2.4 WEP Execution Layer

The WEP execution is managed at the WEP execution layer. This layer handles workflow scheduling, task execution and fault-tolerance.

Through SWf scheduling, a SWfMS produces a Scheduling Plan (SP), which aims at making good use of computing resources and preventing execution stalling [31]. A SWfMS can schedule SWf fragments, bags of tasks or individual tasks into an execution site (computer clusters explained in Section 2.2.2.5) or a computing node according to different task scheduling methods. The scheduling methods are presented in Section 2.3.2. Some SWfMSs outsource SWf scheduling to external tools (see Section 2.2.2.3). Even though these SWfMSs can achieve parallelism at the task level, they cannot optimize SPs in external tools, which are generally not data-flow aware, according to the entire structure of the SWf [64].

During task execution, the input data is transferred to the computing nodes and the output data is produced. Generally, the provenance data is also generated at this time. SWfMSs can execute tasks either directly in their execution engine (*e.g.* Kepler, Galaxy, Pegasus, Chiron) or using an external tool (*e.g.* web service, MapReduce system). To enable parallel task execution, SWfMSs may exploit MPI (or an MPI-like language), SSH commands, web services, Hadoop or other middlewares. MPI and SSH allow the SWfMS to have full control of task execution. However, MPI requires using a shared file system while SSH does not. Using web services, Hadoop or other middlewares, parallel task execution moves outside the direct control of SWfMS.

The SWf fault tolerance mechanism deals with failures or errors of task execution and guarantees the availability and reliability of SWf execution. According to Ganga and Karthik [77], fault-tolerance techniques can be classified into proactive and reactive. Proactive fault tolerance avoids faults and errors by predicting the failure and proactively replacing the suspected components from other working components. Reactive fault-tolerance reduces the effect of failures after perceiving failures, using check pointing/restart, replication and task resubmission techniques. Ganga and Karthik [77] propose a task replication technique based on the idea that a replication of size *r* can tolerate *r-1* failed tasks while keeping the impact on the execution time minimal. Costa *et al.* [45] introduce heuristics based on real-time provenance data for detecting task execution failure and re-executing failed tasks. This heuristic re-executes failed tasks during SWf execution using extra computing resources in the cloud to reduce bad influences on SWf execution from the task failures.

2.2.2.5 Infrastructure Layer

The limitations of computing and storage resources of one computer force SWfMS users to use multiple computers in a cluster, grid or cloud infrastructure for SWf execution. This layer provides the interface between the SWfMS and the infrastructure.

Cluster computing provides a paradigm of parallel computing for high performance and availability. A computer cluster, or cluster for short, consists of a set of interconnected computing nodes [42]. A cluster is typically composed of homogeneous physical computers interconnected by a high speed network, *e.g.* Fast Ethernet or Infiniband. A cluster can consist of computer nodes in the grid or supercomputers [49]. In addition, the cluster can also consist of virtual machines (VMs) in the cloud. In the cloud, a VM is a virtualized machine (computer), *i.e.* a software implementation of a computer that executes programs (like a real computer) while abstracting away the details of physical hardware [200]. Cluster users can rely on message passing protocols, such as MPI for parallel execution.

According to Foster and Kesselman [73], a grid is a hardware and software infrastructure that manages distributed computers to provide good quality of service through standard protocols and interfaces with a decentralized control mechanism. A grid federates geographical distributed sites that are composed of diverse clusters belonging to different organizations through complex coordinating mechanisms to serve as a global system. Furthermore, it has rules to define who can use what resources for what destination [73]. In addition, a particular grid, *i.e.* desktop grid, exists for SWf execution [157]. Compared to cluster computing, grid computing gathers heterogeneous computer resources to provide more flexible services to diverse users by inner resource allocation mechanisms.

Cloud computing provides computing, storage and network resources through infrastructure, platform and software services, with the illusion that resources are unlimited. Although sometimes confused, there are five main differences between cloud computing and grid computing. The first one is that the cloud uses virtualization techniques to provide scalable services that are independent of the physical infrastructure. The second one is that it not only provides infrastructure services such as computing resources or storage resources but also platform and software services. In the cloud, we can configure and use a cluster composed of VMs. Moreover, database management systems can be offered as platform in the cloud. The third difference is the possibility of dynamic provisioning. In a grid environment, a list of resources is generally fixed during the entire execution of the SWf. Thus, it is very unusual to use dynamic provisioning in the grid as it does not provide any benefit. In contrast, in cloud environments, we have a list of resource types from which we can provision a potentially unlimited number of resource instances. Such dynamic provisioning can provide much more benefits, in particular better performance, and reduced financial cost. The fourth difference is in the use of service-level agreement (SLA), which defines the quality of service provided to users by service providers [187]. Cloud SLA includes relatively full performance metrics for on-demand services [32] while grid SLA is relatively informal. The fifth difference is that the cloud provides support for pricing and accounting services, which is not necessary in the grid. Some grids evolve towards the cloud. For instance, Grid'5000 [13] is a grid in France which provides virtualized resources and services for big data (*e.g.* Hadoop).

The operational layer is also in charge of provisioning, which can be static or dynamic. Static provisioning can provide unchangeable resources for SWfMSs during SWf execution while dynamic provisioning can add or remove resources for SWfMSs at runtime. Based on the types of resources, provisioning can be classified into computing provisioning and storage provisioning. Computing provisioning means offering computing nodes to SWfMSs while storage provisioning means providing storage resources for data caching or data persistence. However, most SWfMSs are just adapted to static computing and storage provisioning.

The data storage module generally exploit database management systems and file systems to manage the data during SWf execution. Some SWfMSs such as Taverna put intermediate data and output data in a database. As proposed in [202], some SWfMSs such as Pegasus and Chiron utilize a shared-disk file system (see Section 2.4.3.1). Some SWfMSs such as Kepler [183] can exploit distributed file systems (see Section 2.4.3.1). Some SWfMSs such as Pegasus can directly take advantage of the local file systems in each computing node. Generally, the file systems and the database management systems take advantage of computing nodes and storage resources provided by the provisioning module. In a multisite environment, SWfMSs can cluster the data and place each data set at a single site, distribute the newly generated data to multiple sites at runtime and adjust data among multiple sites [197]. SWfMSs can also put some data in the disk cache of one computing node to accelerate data access during SWf execution [165]. However, existing SWfMSs just use few types of storage resources, some other types of storage resources, *e.g.* cache for a single site, cache in one computing node etc., can be also exploited.

2.2.3 Techniques for Data-intensive Scientific Workflows

Because they deal with big data, data-intensive SWfs have some features that make them more complicated to handle, compared with traditional SWfs. From the existing solutions, we can observe three main features, which we briefly discuss.

The first feature is the diversity of data sources and data formats in data-intensive SWfs. The data can consist of the input data stored in a shared-disk file system and the intermediate data stored in a database. The data of various data sources differ in data transfer rate, data processing efficiency and data transfer time. These differences have a strong influence on SWf design and execution. However, the SWf representation method composed of activities and dependencies for general SWfs can only depict different computational components and the data dependencies among them. Thus, the data-intensive SWf representation should be adapted to be able to depict the diverse types of data.

The second feature is that moving some program code (*i.e.* instruction data) to where the input data is can be more efficient than moving the data to the code. This is true when the input data sets are very big while the corresponding instruction data is small. The decision of moving code to where the data is should be done during the deployment phase in the SWf life cycle (see Section 2.2.1.2), in order to optimize SWf execution according to the characteristics of the input data. However, moving program codes across computing nodes is not always possible, for instance, because of proprietary rights or runtime compatibility.

The third feature is that not all the data needs to be kept all along SWf execution [61]. In particular, given fixed constraints on storage capacity allocated to SWf execution, the intermediate data may be too big to be kept. Thus, it is important to discover and keep only the necessary data, to remove redundant data and to compress the data that is not used frequently.

There have been several studies that propose techniques for data-intensive SWf representation, data processing and redundant data removing, which we discuss below.

Albrecht *et al.* [18] propose a makeflow method for representing and running a data-intensive SWf. In their system, the input data of each activity should be explicitly specified for SWf representation or the SWf description will be regarded as incorrect. An example is shown in Figure 2.5 with a BLAST SWf (from bioinformatics) that has four types of activities. The first activity takes input data and splits the data into several files. The second type of (BLAST) activities searches for similarities between a short query sequence and a set of DNA or amino acid sequences [112]. The third type of activities (cat) regroups the output and errors of BLAST activities into two files. The last activity (finish) generates the final results. In Figure 2.5, the input data and the intermediate data are represented explicitly for further SWf textual description and execution.

Deng *et al.* [62] propose a task duplication approach in SWfMS for scheduling tasks onto multiple data centers. They schedule the tasks by comparing the task computational time and output data transmission time. For instance, let us consider two data centers as depicted in Figure 2.6. Tasks t_1 and t_3 and the corresponding input data d_1, d_3 are located at data center dc_1. Task t_2 at data center dc_2 needs to take the output of task t_3 as input data d_2. We note as T_1 the time to transfer the data d_2 from data center dc_1 to data center dc_2. We note as T_2, the sum of the time to transfer the input data d_3 from data center dc_1 to data center dc_2 and the time to execute task t_3. If time T_1 is longer than time T_2, the SWfMS will duplicate task t_3 from data center dc_1 to data center dc_2 to reduce execution time as shown in Figure 2.6 (b). If not, the SWfMS executes the tasks as they are and transfers the output of task t_3 to data center dc_2 as shown in Figure 2.6 (a). Furthermore, Raicu *et al.* [155] propose a data-aware scheduling method for data-intensive application scheduling. This approach schedules the tasks according to data location and available

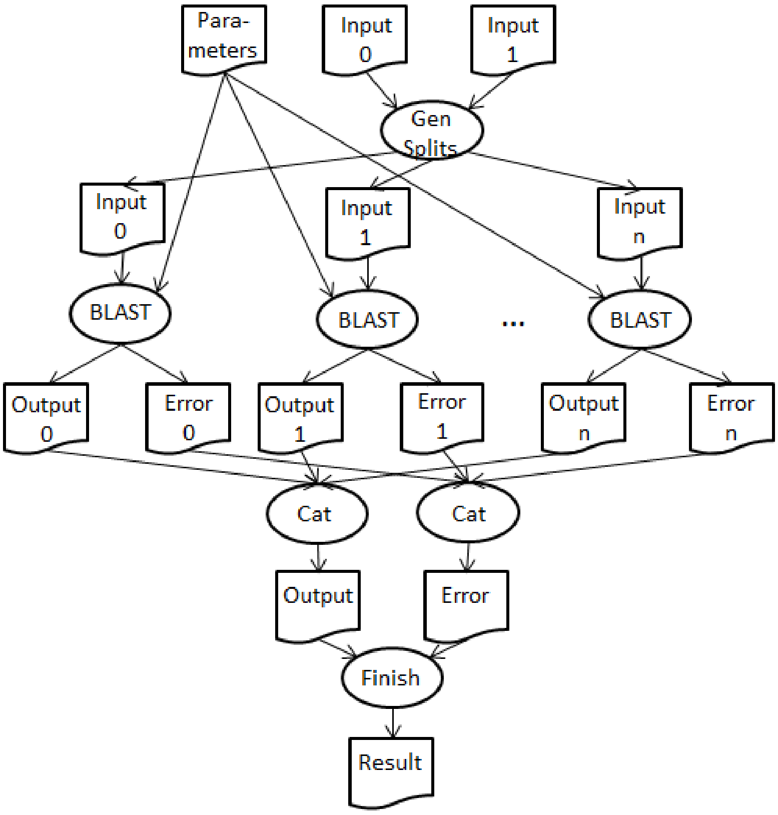


Figure 2.5: **BLAST SWf [18].**

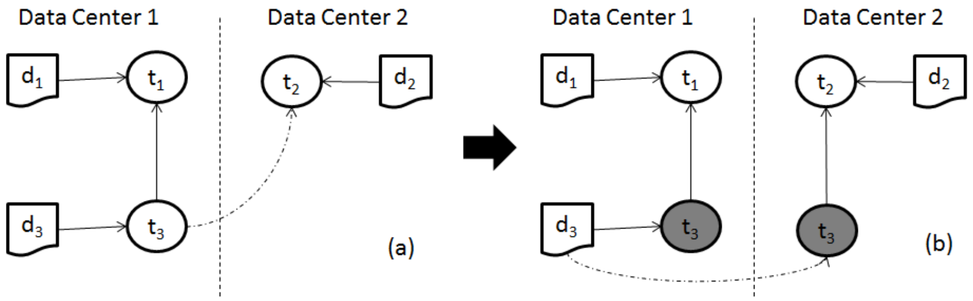


Figure 2.6: **Task duplication [62].**

execution computing resources.

Yuan *et al.* [197] build an Intermediate data Dependency Graph (IDG) from data provenance of SWf execution. Based on IDG, a novel intermediate data storage strategy is developed to store the most appropriate intermediate datasets instead of all the intermediate data to reduce the storage cost during execution. Ramakrishnan *et al.* [156] propose an approach for scheduling data-intensive SWfs onto storage-constrained distributed resources. This approach minimizes the amount of data by removing needless data and scheduling tasks according to the storage capacity on individual resources.

There are some other techniques for data-intensive SWfs, in particular, algebraic optimization and data transfer optimization. Dias *et al.* [64] discuss several performance advantages of having an algebra and dataflow-oriented execution engine for data-intensive applications. They argue that current main approach that statically generates a WEP or an execution plan for Hadoop leaves no room for dynamic runtime changes. They propose that dataflow-based data-intensive SWfs can be executed by algebraic SWfMS, such as Chiron and Scicumulus, with efficient algebraic optimizations. Moreover, we can take advantage of the former data transfer orders or current data location to control data transfer for reducing the makespan of data-intensive SWf execution. Chervenak *et al.* [40] describe a policy service that provides advice on data transfer orders and parameters based on ongoing and recent data transfers and current allocation of resources for data staging.

2.3 Parallel Execution in SWfMSs

Since data-intensive SWfs handle large input or intermediate datasets in large scale experiments, SWfMSs rely on the parallel techniques on multiple computers to accelerate the execution. SWf parallelization is the process of transforming and optimizing a (sequential) SWf into a parallel WEP. WEP allows the SWfMS to execute the SWf in parallel in a number of computing nodes, *e.g.* in a cluster. It is similar to the concept of Query Execution Plan (QEP) in distributed database systems [146].

This section introduces the basic techniques for the parallel execution of SWfs in SWfMSs: SWf parallelism techniques; SWf scheduling algorithms; and parallelization in the cloud. The section ends with concluding remarks.

2.3.1 Scientific Workflow Parallelism

SWf parallelization identifies the tasks that can be executed in parallel in the WEP. Similar to parallel query processing [146], whereby a QEP can be parallelized based on data and operator dependencies. There are two parallelism levels: coarse-grained parallelism and fine-grained parallelism. Coarse-grained parallelism can achieve parallelism by executing sub-workflows or fragments in parallel. Fine-grained parallelism realizes parallelism by executing different activities in parallel. If a SWf is composed of sub-workflows, it can be executed in parallel at coarse-grained level to parallelize the execution of sub-workflows and then, executed in parallel at fine-grained level to parallelize the execution of activities within sub-workflows. If a SWf is directly composed of activities, it can just be executed at fine-grained parallelism level to parallelize the execution of activities.

According to the dependencies defined in a SWf, different parallelization techniques can result in various execution plans. Some parameters can be used to evaluate the efficiency of each technique. An important parameter of parallelization is the degree of parallelism, which is defined as the number of concurrently running computing nodes or threads at any given time and that can vary for a given SWf depending on the type of parallelism [33].

2.3.1.1 Coarse-Grained Parallelism

Coarse-grained parallelism is performed at workflow level, which is critical to the execution of meta-workflows or parameter sweep workflows. To execute a meta-workflow, the independent sub-workflows are identified as SWf fragments to be submitted to corresponding SWf execution engine [174]. The execution of a parameter sweep SWf corresponds to the execution of the SWf with different sets of input parameter values. The combination of SWf and each set of input parameter values is viewed as independent sub-workflows, which can be run in parallel [104]. In addition, a SWf can achieve coarse-grained parallelism by SWf partitioning. The coarse-grained parallelism for parallel execution of sub-workflows resembles to the independent activity parallelism presented in the next section.

2.3.1.2 Fine-Grained Parallelism

The fine-grained parallelism is realized within a SWf, a sub-workflow (for meta-workflows) or a SWf fragment. Three types of parallelism exist at this level: data parallelism, independent parallelism and pipeline parallelism. Data parallelism deals with the parallelism within an activity while independent parallelism and pipeline parallelism handle the parallelism between different activities.

Data Parallelism

Data parallelism, similar to intra-operator parallelism in [146], is obtained by having multiple tasks performing the same activity, each on a different data chunk. As shown in Figure 2.7(b), data parallelism happens when the input data of an activity can be partitioned into different chunks and each chunk is processed independently by a task in a different computing node or processor. As the input data needs be partitioned, *e.g.* by a partitioning task, the activity result is also partitioned. Thus, the partitioned output data can be the base for data parallelism for the next activities. However, to combine the different results to produce a single result, *e.g.* the final result to be delivered to the user, requires special processing, *e.g.* by having all the tasks writing to a shared disk or sending their results to a task that produces the single result.

Data parallelism can be static, dynamic or adaptive [148]. If the number of data chunks is indicated before execution and fixed during execution, the data parallelism is static. If the number of data chunks is determined at run-time, the data parallelism is dynamic. In addition, if the number is automatically adapted to the execution environment, the data parallelism is adaptive. The adaptive data parallelism can determine the best number of data chunks to balance SWf execution, to increase parallelism degree while maintaining a reasonable overhead.

Independent Parallelism

Different activities of a SWf can be executed in parallel over several computing nodes. Two activities can be either independent, *i.e.* the execution of any activity does not depend on the output data of the other one, or dependent, *i.e.* there is a data dependency between them. Independent parallelism exploits independent activities while pipeline parallelism (see next subsection) deals with dependent activities.

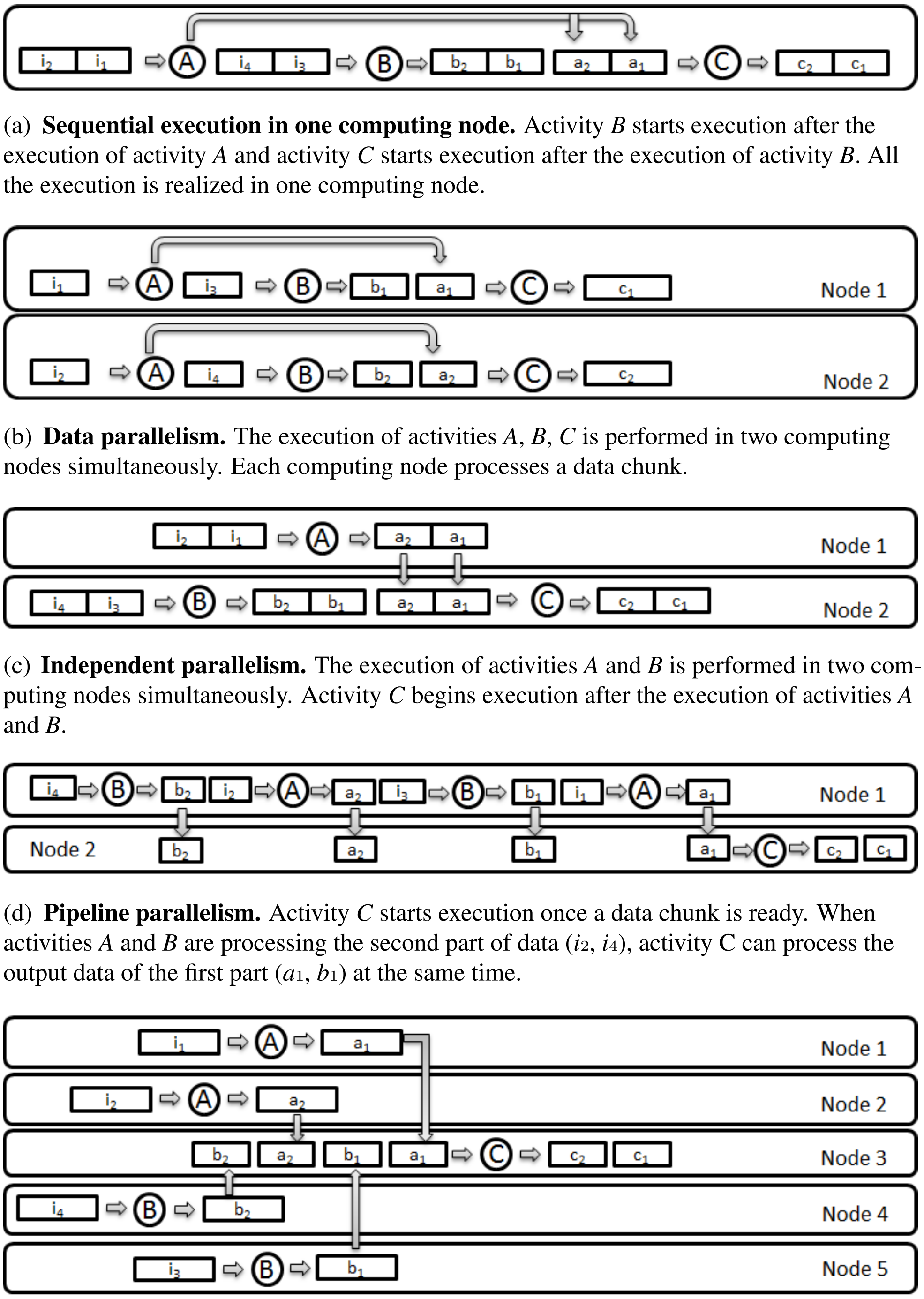
Independent parallelism is achieved by having tasks of different independent activities executed simultaneously. As shown in Figure 2.7(c), independent parallelism occurs when a SWf has more than one independent part in the graph and the activities in each part have no data dependencies with those in another part. To achieve independent parallelism, a SWfMS should identify activities that can be executed in parallel. SWfMSs can partition the SWf into independent parts (or SWf fragments) of activities to achieve independent parallelism.

Pipeline Parallelism

With pipeline parallelism (see Figure 2.7(d)), several dependent activities with a producer-consumer relationship are executed in parallel by different tasks. One output data chunk of one activity is consumed directly by the next dependent activities in a pipeline fashion. The advantage of pipeline execution is that the result of the producer activity does not need to be entirely materialized. Instead, data chunks can be consumed as soon as they are ready, thus saving memory and disk accesses.

Hybrid Parallelism

As shown in Figure 2.7(e), the three basic types of parallelism can be combined to achieve higher degrees of parallelism. A SWfMS can first perform data parallelism within each



(a) **Sequential execution in one computing node.** Activity *B* starts execution after the execution of activity *A* and activity *C* starts execution after the execution of activity *B*. All the execution is realized in one computing node.

(b) **Data parallelism.** The execution of activities *A*, *B*, *C* is performed in two computing nodes simultaneously. Each computing node processes a data chunk.

(c) **Independent parallelism.** The execution of activities *A* and *B* is performed in two computing nodes simultaneously. Activity *C* begins execution after the execution of activities *A* and *B*.

(d) **Pipeline parallelism.** Activity *C* starts execution once a data chunk is ready. When activities *A* and *B* are processing the second part of data (i_2, i_4), activity C can process the output data of the first part (a_1, b_1) at the same time.

(e) **Hybrid parallelism.** Activity *A* is executed through data parallelism at nodes 1 and 2. Activity *B* is executed through data parallelism at nodes 4 and 5. Activities *A* and *B* are also executed through independent parallelism. Activities *A* and *C*, respectively *B* and *C*, are executed through pipeline parallelism between nodes (1, 2) and 3, respectively nodes (4, 5) and 3.

Figure 2.7: **Different types of parallelism.** Circles represent activities. There are three activities: *A*, *B* and *C*. *C* processes the output data produced by *A* and *B*. Rectangles represent data chunks. "i_1" stands for the first part of input data. "a_1" stands for the output data corresponding to the first part of input data after being processed by activity *A*.

activity. Then, it can partition the SWf into independent parts or fragments for independent activities, *e.g.* with each part or fragment for execution in a different computing node. Finally, pipeline parallelism can be applied for executing dependent activities in parallel. In addition, the parallelism strategies may also be changed at runtime, according to the parallel computing environment behavior [50]. For the activities that process output data produced by more than one activity, the data is generally merged for the follow-up activity. This merging operation can also be found in the shuffle phase of the MapReduce program execution. By combining these mechanisms, the degree of parallelism can be maximized at different execution layers.

We illustrate different types of parallelism, including their combination in hybrid parallelism, with the example shown in Figure 2.7. In this figure, one task consists of one activity and the related input data. Figure 2.7(a) presents the sequential execution of Activity A, B, C and two parts of input data, *i.e.* i_1 and i_2. Since there is no parallelization, the corresponding tasks, *i.e.* Activity A and Data i_1, Activity A and Data i_2, Activity B and Data i_3, Activity B and Data i_4, Activity C and Data a_1, b_1, Activity C and Data a_2 , b_2, are executed one after another. Figure 2.7(b) describes the execution with data parallelism. The processing of each part of input data is done in different computing nodes in parallel, *i.e.* the processing of input data i_1, i_3 and that of input data i_2, i_4 are done at the same time. Figure 2.7(c) shows independent parallelism. Activity A and B are executed at different computing nodes at the same time. Figure 2.7(d) shows pipeline parallelism, *i.e.* the parallel execution of Activity A (or B) and Activity C. Activity A (or B) can be done at the same time as Activity C while processing the different parts of input data. While Activity C is processing Data a_1 and b_1 at Node 2, which corresponds to the input data i_1 and i_3, Activity A (or B) can process the input data i_2 (or i_4). Figure 2.7(e) shows hybrid parallelism. Thus, the tasks, *i.e.* Activity A and Data i_1, Activity A and Data i_2, Activity B and Data i_3, Activity B and Data i_4 can be executed in parallel in different computing nodes. Activity C can begin execution once both Data a_1 and b_1 (or both Data a_2 and b_2) are ready. This parallelism combines data parallelism, independent parallelism and pipeline parallelism.

2.3.2 Scientific Workflow Scheduling

SWf scheduling is a process of allocating concrete tasks to computing resources (*i.e.* computing nodes) to be executed during SWf execution [33]. The goal is to get an efficient Scheduling Plan (SP) that minimizes a function based on resource utilization, SWf execution cost and makespan. Since a SWfMS can schedule bags of tasks, there may be a task clustering phase to generate task bags. Moreover, scheduling methods can be static, dynamic or hybrid.

The SWfMSs that schedule tasks without external tools choose computing nodes to execute tasks without constraints. The SWfMSs that outsource SWf parallelization or SWf scheduling may relay on external tools to schedule tasks. The following scheduling methods focus on the SWfMSs that manage SWf scheduling by themselves.

2.3.2.1 Task Clustering

A SWfMS can schedule bags of tasks to a computing nodes or multiple computing nodes to reduce the scheduling overhead so that the execution time can be reduced. A bag of tasks contains several tasks to be executed in the same computing node. Note that bags of tasks are different from fragments. Fragments are generated by analyzing activities while bags of tasks are produced by grouping executable tasks. Several studies have been done for generating bags of tasks. Deng *et al.* [62] present a clustering method for efficient SWf execution. They use a k-means clustering method to group the tasks into several task bags according to different dependencies: data-data dependency, task-task dependency, and task-data dependency. These three types of dependencies are used to measure the correlations between datasets and tasks in a SWf. W. Chen *et al.* [39] present a balanced task clustering approach for SWf execution. They cluster the tasks by balancing total execution of each bag of task.

2.3.2.2 Static Scheduling

Static scheduling generates a SP that allocates all the executable tasks to computing nodes before execution and the SWfMS strictly abides the SP during the whole SWf execution [33]. Because it is before execution, static scheduling yields little overhead at runtime. It is efficient if the SWfMS can predict the execution load of each task accurately, when the execution environment varies little during the SWf execution, and when the SWfMS has enough information about the computing and storage capabilities of the corresponding computers. However, when the execution environment experiences dynamical changes, it is very difficult to achieve load balance. The static task scheduling algorithms have two kinds of processor selection methods [177]: heuristic-based and guided random search based. The heuristic-based method schedules tasks according to a predefined rule while the random search based method schedules tasks randomly. Static task scheduling algorithms can also be classified between task-based and workflow-based [30]. The task-based method directly schedules tasks into computing nodes while the workflow-based method schedules a fragment into computing nodes. Since the workflow-based method transfers the data with less overhead compared to the task-based method, it is better for data-intensive applications.

Topcuouglu *et al.* [177] propose two static scheduling algorithms: Heterogeneous Earliest-Finish-Time (HEFT) and Critical-Path-on-a-Processor (CPOP). Both algorithms contain a task prioritizing phase and a processor selection phase. The task prioritizing phase is for ranking tasks while the processor selection phase is for scheduling a selected task on a "best" computing node, which minimizes the total execution time. We note the average computation cost of a task as *CPC* and the average communication cost of the current task to a succeed task as *CMC*. The $rank_u$ is the rank that is based on *CPC* and *CMC*. The $rank_d$ indicates the rank based on *CPC* and *CMC* consisting of the communication from a preceding task to the current task. In the task prioritizing phase, HEFT ranks tasks based on $rank_u$. In the processor selection phase, HEFT selects a computing node, which finishes its current task firsts. HEFT can also insert a task in a computing node when there is idle time between the execution of two consecutive tasks. The CPOP algorithm uses a $rank_c$ that combines both $rank_u$ and $rank_d$ in the task prioritizing phase. It utilizes a critical path in the processor selection phase. A critical path is a pipeline of tasks, in which a task has no more than one input dependency and no more than one output dependency. Each task in the critical path has the highest priority value (in $rank_c$) in all the tasks that have the input data dependencies from the same parent task. CPOP chooses a computing node as a critical-path processor and schedules the tasks in the critical path to the critical-path processor. It schedules the other tasks to the other computing nodes with the same mechanism as HEFT.

2.3.2.3 Dynamic Scheduling

Dynamic scheduling produces SPs that distribute and allocate executable tasks to computing nodes during SWf execution [33]. This kind of scheduling is appropriate for SWfs, in which the workload of tasks is difficult to estimate, or for environments where the capabilities of the computers varies a lot during execution. Dynamic scheduling can achieve load balancing while it takes time to dynamically generate SPs during execution. The scheduling algorithms can be based on the queue techniques in a publish/subscribe model with different strategies such as First In First Out (FIFO), adaptive and so on. SWfMSs such as Swift [188], Chiron [139], and Galaxy [106] exploit dynamic scheduling algorithms.

Dynamic SPs may be generated by adapting a static scheduling method to dynamic environment. Yu and Shi [196] introduce an HEFT-based dynamic scheduling algorithm. This algorithm is suited to the situation where a SWf has been executed partially before scheduling. It schedules the tasks by applying an HEFT-based algorithm according to a dynamically generated rank of tasks.

There are some original approaches to generate dynamic SPs. Maheswaran *et al.* [125] present a min-min algorithm that is designed as a batch mode scheduling of two steps. First, a list of tasks ready to be executed is created. This phase is called “task prioritizing” phase. Then, the tasks in the list are scheduled to computing nodes based on a heuristic. The heuristic maps the task T to the computing node M such that T is the task that has minimum expected execution time in the non-mapped tasks and that M is the computing node that is executing a task having minimum expected execution time in the mapped tasks. This phase is called the “resource selection” phase.

Anglano and Canonico [22] present several knowledge-free scheduling algorithms that are able to schedule multiple bags of tasks. A knowledge-free algorithm does not require any information on the resources for scheduling. These algorithms implement different policies: First Come First Served – Exclusive (FCFS-Excl), First Come First Served – Shared (FCFS-Share), Round Robin (RR), Round Robin –No Replica First (RR-NRF) and Longest Idle. With the FCFS-Excl policy, bags of tasks are scheduled in order of arrival. Different from FCFS-Excl, FCFS-Share can allocate more than one bag of tasks to a computing node. The RR policy schedules bags of tasks in a fixed circular order while all the bags have the same probability to be scheduled. With the RR-NRF policy, a bag of tasks that does not have any task executed will be given priority. In this policy, all the bags of tasks have at least a task running. The longest idle policy tries to reduce waiting time by giving preference to the bag of tasks hosting the task that exhibits the largest waiting time. The paper shows that the FCFS-based policy performs better for small task granularity scheduling.

2.3.2.4 Hybrid Scheduling

Both of static and dynamic scheduling have their own advantages and they can be combined as a hybrid scheduling method to achieve better performance than just using one or the other. For example, a SWfMS might schedule a part of the tasks of a SWf, *e.g.* those tasks for which there is enough information, using static scheduling and schedule the other part during execution with dynamic scheduling [62].

Oliveira *et al.* [50] propose a hybrid scheduling method with several algorithms: greedy scheduling, task grouping, task performing, and load balancing. The greedy scheduling algorithm produces static WEPs to choose the most suitable task to execute for a given idle VM based on a proposed cost model. The task grouping algorithm produces new tasks by encapsulating two or more tasks into a new one. The task performing algorithm sets up the granularity factor for each VM in the system and modifies the granularity according to the average execution time. The load balancing algorithm is a dynamic scheduling algorithm that adjusts the number of VMs and static WEP in order to meet the deadline of execution time and the budget limit.

2.3.2.5 Scheduling Optimization Algorithms

Since there are many criteria to measure SWf execution, SWfMS users may have multiple objectives for SWf execution, such as reducing execution time, minimizing monetary cost etc. Therefore, SPs should also be optimized to attain multiple objective in a given context (cluster, grid, cloud). Unlike query optimization in database, however, this optimization phase is often not explicit and mixed with the scheduling method. Though there are some existing scheduling optimization algorithms [50, 88, 70, 65], they do not consider a multisite environment with distributed data at each site. We present [88] and [70] as examples.

Gu *et al.* [88] address the scheduling optimization problem of mapping distributed SWf to maximize the throughput in unstable networks where nodes and links are subject to probabilistic failures. And they propose a mapping algorithm to maximize both throughput and reliability for SWf scheduling. They consider a network where the failure occurrences follow a Poisson distribution with a constant parameter. The mapping algorithm includes three algorithms: disLDP-F, Greedy disLDP-F and decentralized Greedy disLDP-F. The disLDP-F algorithm schedules the tasks by identifying and minimizing the global bottleneck time based on the rank of computational requirements of tasks. Greedy disLDP-F reduces the search complexity of disLDP-F by selecting the best node for each type of requirement of the current task and then generates a best computing node for the current task. The decentralized Greedy disLDP-F algorithm decentralizes the disLDP-F algorithm by storing all the parameters of each individual node locally and selecting the node through the communication between individual nodes instead of centralized control.

Fard *et al.* [70] propose a multi-objective scheduling method of SWf execution in distributed computing infrastructures. Their approach generates a best scheduling strategy, which is a Pareto optimality (no other strategy can achieve a result of a better component while ensuring the other components of the result as well as this strategy), to achieve 4 objectives, *i.e.* execution time, monetary cost, energy consumption and reliability. Nevertheless, this approach does not consider data distribution in a multisite environment.

2.3.2.6 Conclusion

Data-intensive SWfs need to process big data, which may take a very long time with sequential execution. Parallel execution is therefore necessary to reduce execution time on parallel computers. SWf parallel execution exploits a WEP that includes parallel execution decisions, which achieves SWf parallelism. The parallel execution also schedules execution tasks to computing nodes with optimization instructions.

SWf parallelism consists of two lelvels, *i.e.* fine-grained and coarse-grained. Fine-grained parallelism is for parallelizing the execution of sub-workflows and fragments. Coarse-grained parallelism parallelizes the execution of activities in three basic types: data parallelism, independent parallelism and pipeline parallelism. Data parallelism is fine-grained parallelism within one activity and can yield a very high degree of parallelism on big datasets. Independent parallelism and pipeline parallelism exploit the parallelism between different activities. These are coarse-grained and the degree of parallelism is bound by the maximum of activities. Therefore, the highest levels of parallelism can be achieved by combining these three types of parallelism into hybrid parallelism.

SWf scheduling is a process of allocating concrete tasks to computing node during SWf execution. Static scheduling method generates a SP prior to SWf execution and thus SWf execution is very fast, but it makes SWfMSs difficult to achieve load balancing at a dynamically changing environment. Dynamic scheduling can better achieve load balancing but takes more time to generate SPs at run-time. Hybrid scheduling can combine the best of both static and dynamic scheduling. SWf scheduling performs some optimization, trying to reach multiple objectives such as minimizing the makespan of SWf execution or reducing monetary expenses.

2.4 SWfMS in a Multisite Cloud

The cloud, which provides virtually infinite computing and storage resources, appears as a cost-effective solution to deploy and to run data-intensive SWfs. For scalability and high availability, large cloud providers such as Amazon and Microsoft typically have multiple data centers located at different sites. In general, a user uses a single site, which is sufficient for most applications. However, there are important cases where SWfs will need to be deployed at several sites, *e.g.* because the data accessed by the SWf is in different research groups' databases at different sites or because SWf execution needs more resources than those at one site. Therefore, multisite management of data-intensive SWfs in the cloud becomes an important problem.

This section introduces cloud computing and discusses the basic techniques for the parallel execution of SWfs in the cloud, including multisite management and data storage. This section ends with concluding remarks.

2.4.1 Cloud Computing

Cloud computing encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage and networking resources. These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model, in which guarantees are offered by the cloud provider by means of customized Service-Level Agreements (SLAs). SLA is a part of a service contract where a service is formally defined [187]. SLA defines the quality of service provided to users by the cloud providers. One of the major differences between grid and cloud is the quality of service as Grid computing offers only best effort service. In addition, clouds provide support for pricing, accounting and SLA management.

Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, system administration, or application deployment, to very large data centers operated by cloud providers. Thus, the complexity of managing the software/hardware infrastructure gets shifted from the users' organization to cloud providers.

Cloud services can be divided into three broad categories: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). SaaS is the delivery of application software as a service. Hosted applications can range from simple ones such as email and calendar to complex applications such as Customer Relationship Management (CRM), data analysis or even social networks. Examples of popular SaaS are Safesforce CRM system and Microsoft Office 365.

PaaS is the delivery of a computing platform with development tools and APIs as a service. It enables developers to create and deploy custom applications directly on the cloud infrastructure, in VMs, and integrate them with applications provided as SaaS. Examples of popular PaaS are Google App Engine and Windows Azure Platform.

IaaS is the delivery of a computing infrastructure (*i.e.* computing, networking and storage resources) as a service. It enables customers to scale up (add more resources) or scale down (release resources) as needed (and only pay for the resources consumed). This important capability is called *elasticity* and is typically achieved through *server virtualization*, a technology that enables multiple applications to run on the same physical computer as VMs, *i.e.* as if they would run on distinct physical computers. Customers can then require computing instances as VMs and attach storage resources as needed. Because it is cost-effective, all cloud providers use computer clusters for their data centers, and often shared-nothing clusters with commodity computers. Examples of popular IaaS are Amazon Elastic Compute Cloud (EC2) and Microsoft Azure.

SaaS, PaaS and IaaS can be useful to develop, share and execute SWf components as cloud services. However, in the rest of this report, we will focus on IaaS, which will allow running existing SWfs in the cloud.

2.4.2 Multisite Management in the Cloud

One site in the cloud may not be big enough for providing unlimited computing and storage capability for the world. Big cloud providers such as Microsoft and Amazon typically have many geographically distributed sites. For instance, Microsoft Azure separates the world into six regions and Amazon has three sites in the USA, one site in Europe, three sites in Asia and one site in South America.

We can define a multisite cloud as a cloud composed of several sites (or data centers), each from the same or different providers and explicitly accessible to cloud users [134]. Explicitly accessible has two meanings. The first one is that each site is separately visible and directly accessible to cloud users. The second one is that the cloud users can decide to deploy their data and applications at specific sites while the cloud providers will not change the location of their data. The computing resources providers include grid computing and computer cluster providers.

In a multisite cloud environment, cloud users must take care of the location of their data, which can be difficult. A multisite cloud platform is a solution that can manage several sites (or data centers) of single or multiple cloud providers, with a uniform interface for cloud users.

BonFIRE [97] is a European Research project[5] that develops a multisite cloud platform for applications, services and systems experimentation. It adopts a federated multi-platform approach, providing interconnection and interoperation between service and networking testbeds. As an IaaS, it provides large-scale, virtualized computing, storage and networking resources with full control of the user on resource deployment. It also provides in-depth monitoring and logging of physical and virtual resources and ease of use of experimentation. BonFIRE currently comprises several (7 at the time of this writing) geographically distributed testbeds across Europe. Each testbed provides its computing, storage and network resources and can be accessed seamlessly with a single experiment descriptor through the BonFIRE API, which is based on the Open Cloud Computing Interface.

U-chupala *et al.* [179] propose a multisite cloud platform based on a Virtual Private Network (VPN) and a smart VM scheduling mechanism. It is composed of a virtual infrastructure layer, an overlay network layer and a physical resource layer. The VM containers lie in the physical resource layer. The overlay network connects all the physical resources together and enables the virtual infrastructure layer to use a cloud framework that gives the illusion of a single pool of resources. This pool can provide scalable resources to users while hiding the complexity of the physical infrastructure underneath.

[5] http://www.bonfire-project.eu

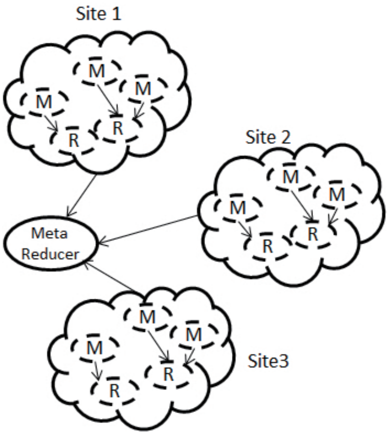


Figure 2.8: **Running MapReduce across a multisite cloud [178].**

A multisite cloud platform may contain modules coming from existing frameworks. Mandal *et al.* [127] have implemented the Hadoop framework in a multisite cloud through a cloud control framework that can gather computing and storage resources from multiple sites and offer a common interface for resource provisioning. They conclude that deploying Hadoop in networked clouds is not difficult but low quality of network yields poor performance.

MapReduce is also extended to deal with a multisite cloud. In [178], a MapReduce application is partitioned for several sites according to the available data chunks distributed at each site. In this architecture (see Figure 2.8), a MetaReducer is implemented as an independent service and built on top of a pool of reducers distributed over multiple sites. It is used to generate the final result by aggregating all the intermediate results generated by the reducers at different sites. In [124], Hadoop is also extended to multiple sites with partitioning and a global reducer (similar to the MetaReducer) while two multisite scheduling algorithms are proposed. The first scheduling algorithm is to schedule the tasks according to computing capacity at each site and the second algorithm is to schedule tasks according to the data location. In [178] and [124], the granularity of scheduling is a fragment, which contains both Map and Reduce operations. Hadoop is also extended to multiple sites at task level, *i.e.* the scheduling granularity can be either Map operation or Reduce operation, while there is no optimized task scheduling algorithms for the inter-site or intra-site data transfer [184].

2.4.3 Data Storage in the Cloud

Data storage in the cloud is critical for the performance of data-intensive SWfs. It can be done using different file systems. In this section, we discuss the techniques for file systems that can be used in the cloud.

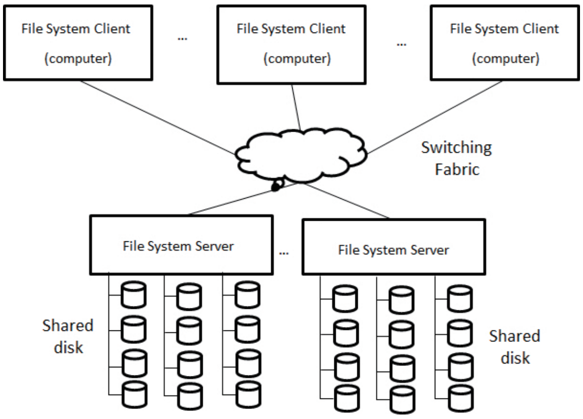


Figure 2.9: **GPFS architecture.**

2.4.3.1 File Systems

A file system is in charge of controlling how information is stored and retrieved in a computer or a computer cluster [23]. In the cloud, IaaS users need a file system that can be concurrently accessible for all the VMs. This can be achieved through a shared-disk file system or a distributed file system.

Shared-disk file systems

In a shared-disk file system, all the computing nodes of the cluster share some data storage that are generally remotely located. Examples of shared-disk file systems include General Parallel File System (GPFS) [162], Global File System (GFS) [152] and Network File System (NFS) [160].

A shared-disk file system is composed of data storage servers, a Storage Area Network (SAN) with fast interconnection (*e.g.* Infiniband or Fiber (GPFS), Channel) and is accessible to each computing node. The data storage servers offer block data level storage that is connected to each computing node by storage are network. The data in data storage servers can be read or written as in the local file system. The shared-disk file system handles the issues of concurrent access to file data, fault-tolerance at the file level and big data throughput.

Let us illustrate with General Parallel File System (GPFS), IBM's shared-disk file system. GPFS provides the behavior of a general-purpose POSIX file system running on a single computing node. GPFS's architecture (see Figure 2.9) consists of file system clients, a fast interconnection network and file system servers, which just serve as an access interface to the shared disks. The file system clients are the computer nodes in a cluster that need to read or write data from the shared disk for their installed programs. The interconnection network connects the file system clients to the shared disks through a conventional block I/O interface.

GPFS provides fault-tolerance in large-scale clusters in three situations. Upon a node failure, GPFS will restore metadata updated by the failed node to a consistent state and release lock tokens in the failed node and appoint others nodes for special roles played by the failed node. Upon a communication failure, the mechanism for one node lost is handled as the node failures while a network equipment failure causes a network partition. In the case of partition, the nodes in the partition that has the highest number of nodes have access to the shared disks. GPFS uses data replication across multiple disks to deal with disk failures.

Cloud users can deploy a shared-disk file system by installing the corresponding frameworks (*e.g.* GPFS framework) in the VMs with the cloud storage resources such as Microsoft Blob Storage and Amazon Elastic Block Store (EBS). Alternatively, cloud users can mount Amazon Simple Storage Service (S3) into all the Linux-based VMs to realize the functionality that all the VMs can have access to the same storage resource, as with a shared-disk file system.

Distributed file systems

A distributed file system stores data directly in the file system that is constructed by gathering storage space in each computing node in a shared-nothing architecture. The distributed file system integrates solutions for load balancing among computing nodes, fault-tolerance and concurrent access. Files must be partitionned into chunks, *e.g.* through a hash function on records' keys, and the chunks are distributed among computing nodes. Different from the shared-disk file system, computing nodes have to load the data chunks from the distributed file system to the local system before local processing.

Let us illustrate with Google File System (GFS) [80], which had a major impact on cloud data management. For instance, Hadoop Distributed File System (HDFS) is an open source framework based on GFS. GFS is designed for a shared-nothing cluster made of commodity computers, and applications with frequent read operations while write operations mainly consist of appending new data. GFS is composed of a single GFS master node and multiple GFS chunk servers. The GFS master maintains all the file system metadata while GFS chunk servers store all the real data. The master can send instruction information to the chunk servers while the chunk server can send chunk server status information to the master. A GFS client can get the data location information from the file namespace of the GFS master. Then it can write data to the GFS chunk servers at this data location or get the data chunks from a corresponding GFS chunk server according to the data location information and required data size. This mechanism is shown in Figure 2.10. GFS also provides snapshot support, garbage collection, fault-tolerance and diagnosis. For high availability, GFS supports master replication and data chunk replication.

BlobSeer [135] is another distributed file system optimized for Binary Large OBjects (BLOBs). The architecture of BlobSeer is shown in Figure 2.11. Data providers physically store the data in the storage resources (data providers) while physical storage resources can be inserted or removed dynamically in the data providers. The provider manager tracks the information about the storage resources and schedules the placement of newly generated data. All the stored data has a version. Metadata providers store the metadata for identifying data chunks that make up a snapshot version. The version manager assigns new snapshot version numbers to writers and appenders and reveals new snapshots to readers. The write operation is performed in parallel on data chunks and

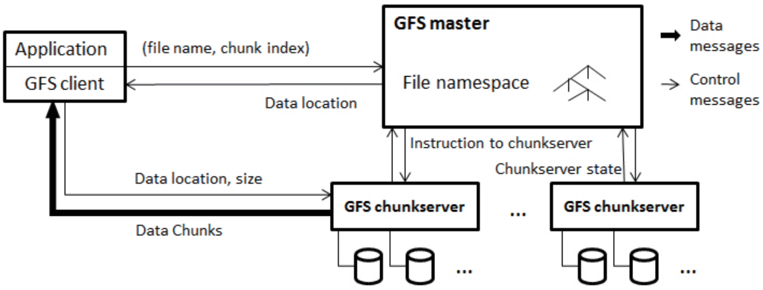


Figure 2.10: **GFS architecture [80].**

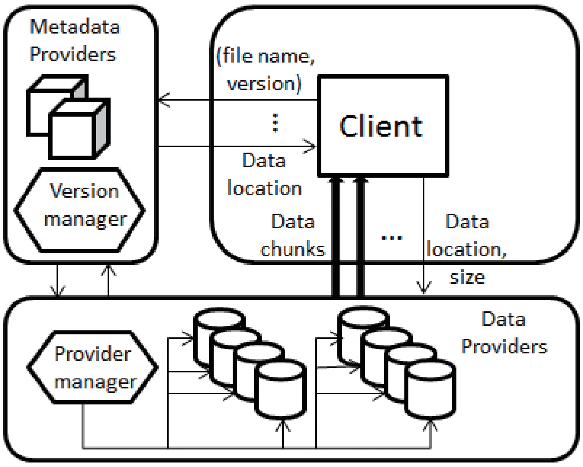


Figure 2.11: **BlobSeer Architecture [135].**

creates a new version of the data. Because of data versioning, read and write operations can be asynchronous and thus improve the read and write throughput. The client can get the data location of the required files corresponding to the file name and the required version when the required version is equal or inferior to the latest snapshot version. Then it can write data to the data providers or get the corresponding data chunks from the data providers by the data location and desired data size. BlobSeer also provides fault-tolerance through replication, consistency semantics and scalability based on several versioning mechanisms. Nicolae *et al.* [135] made a first performance comparison of Blobster with HDFS, which shows important improvements in read and write thoughput, because of versioning.

Cloud users can deploy a distributed file system by installing corresponding frameworks (*e.g.* HDFS) of the aforementioned systems in available VMs to gather storage resources in each VM for executing applications in the Cloud.

2.4.4 Scientific Workflow Execution in the Cloud

The cloud has some useful features to execute SWfs. In particular, the quality of service guaranteed by SLA can yield more stable performance. Juve *et al.* [103] compare the performance of an astronomy application with the Pegasus SWfMS in the grid, the commercial cloud and the academic cloud. They conclude that the performance is the least stable in grid and more stable in commercial cloud than academic cloud. We present the adaptations of SWfMSs for cloud environments, including SWf execution in a single site cloud and that in multiple cloud sites.

2.4.4.1 Execution at a Single Cloud Site

In a single site cloud environment, SWfMSs can be directly installed in the VMs and exploit services deployed in the cloud [101, 191, 58]. Existing parallelization techniques, *e.g.* parallelism techniques (see Section 2.3.1), scheduling techniques (see Section 2.3.2), existing execution execution models in grid [58], can be used to execute a data-intensive SWf in the environment. SWfMSs can exploit some middleware to create or remove VMs and enable the communication between VMs in order to execute data-intensive SWfs in the cloud [90, 182, 17], such as Coasters [90] in Swift, Kepler EC2 actors [182] for Kepler, CloudMan [17] for Galaxy and RabbitMQ12 for Triana[6] SWfMS. These tools can provide computing or storage provisioning for SWf execution or communication between VMs. However, they cannot take advantage of the dynamic provisioning features of the cloud.

Some SWfMSs can take advantage of the scalability of cloud to provision VMs and storage for SWf execution. For instance, Afgan *et al.* [17] propose CloudMan that permits Galaxy to make use of Amazon EC2 and EBS for computing and storage provisioning for SWf execution. Some other SWfMSs are optimized for the cloud by supporting dynamic resource provisioning under budget and time limits, such as Pegasus [126, 133] with Wrangler [102] (a dynamic provisioning system in the cloud) and Askalon [145, 144, 69] SWfMS. Since high parallelization degree can lead to less execution time, SWfMSs can dynamically create new VMs in order to reduce execution time under monetary cost constraint. But if the estimated monetary cost of SWf execution with current number of VMs exceeds the monetary cost constraint, SWfMSs can remove some VMs.

Chiron is adapted to the cloud through its extension, Scicumulus [51, 52]. The architecture of Scicumulus contains three layers and four corresponding tiers: desktop layer for client tier, distribution layer for distribution tier, execution layer for execution tier and data tier. The desktop layer is to compose and execute SWfs. The distribution layer is responsible for parallel execution of activities in the cloud. The execution layer manages activity execution in VM instances. Finally, the data tier manages the related data during SWf execution. Scicumulus exploits hybrid scheduling approaches with dynamic computing provisioning support. Furthermore, Scicumulus uses services such as SciDim [53] to determine an initial virtual cluster size through a multi-objective cost function and provenance data under budget and time limits. Moreover, Scicumulus can be coupled with SciMultaneous, which is used to manage fault tolerance in the cloud [45].

[6] Triana in cloud: `http://www.trianacode.org/news.php`

2.4.4.2 Execution in a Multisite Cloud

Clouds are independent of geographical distribution of physical resources by nature. However, it is possible to control the location of deployed services for better performance in some cloud environments, *e.g.* Microsoft Azure cloud [14] and Amazon cloud [12]. In general, a user uses a single site, which is sufficient for most applications. However, there are important cases where SWfs will need to be deployed at several sites, *e.g.* because the data accessed by the SWf is in different research groups' databases in different sites or because SWf execution needs more resources than those at a single site. Big cloud providers such as Microsoft and Amazon typically have multiple geographically distributed data centers located at different sites. In some cases, a SWf must be executed at multiples sites. There are two approaches to do this. The first approach is to deploy a SWfMS in a multisite cloud platform as discussed in Section 2.4.2. The second approach is to make a SWfMS multisite-aware and capable to utilize computing and storage resources distributed at different sites. This is the approach we now focus on.

In a multisite cloud, we can execute a SWf with or without SWf partitioning. SWf execution without partitioning is to schedule all the tasks into all the VMs in the multiple sites. This centralized method makes it hard to realize load balancing, incurs much overhead for each task and makes scheduling very complicated. With partitioning, a SWf is divided in fragments (see Section 2.3.1) and each fragment is scheduled at a specific site and its tasks scheduled within the VMs at this site. This method can reduce the overhead of task scheduling, which is done in parallel at multiple sites, and realize load balancing at two levels: inter-site and intra-site. Inter-site load balancing is realized by scheduling fragments, with a global scheduler, and intra-site load balancing is realized by local task scheduling. This two-level approach makes the scheduling operation easier.

Swift and Pegasus achieve multisite execution by SWf partitioning. Swift performs SWf partitioning by generating corresponding abstract WEPs for each site [201]. Pegasus realizes partitioning through several methods [38, 39]. As discussed in Section 2.2.2.3, Chen and Deelman [38] propose a SWf partitioning method under storage constraints at each site. This SWf partitioning method is used in a multisite environment with dynamic computing provisioning as explained in [37]. Another method is balanced task clustering [39]. The SWf is partitioned into several fragments which have almost the same workload. This method can realize load balancing for homogeneous computing resources. Askalon can execute SWf in a federated multisite cloud [145], *i.e.* a multisite cloud composed of resources from different providers. Nevertheless, it schedules tasks in computing nodes without considering the organization of computing resources, *i.e.* which VMs are at the same site, for optimization. This method just takes the VMs as the grid computing nodes without considering the features of multisite resources, *e.g.* the difference of data transfer rate, resource sharing for intra-site and inter-site, etc.

2.4.5 Conclusion and Remarks

There are important cases where SWfs will need to be deployed at several data centers in the cloud, either from the same or different cloud providers, thus making multisite management an important problem. Although some SWfMSs such as Swift and Pegasus provide some functionality to execute SWfs in the multisite environment, this is generally done by simply reusing the techniques from a grid environment or simple dynamic provisioning and scheduling mechanisms, without exploiting new data storage and data transfer capabilities provided by multisite clouds.

We believe that much more work is needed to improve the execution of data-intensive SWfs in a multisite cloud. First, the co-scheduling of tasks and data should be exploited. Most SWfMSs make use of a shared file system to store data but do not care about where the data is stored for task scheduling. We believe that the co-scheduling of tasks and data can be efficient at maximizing local data processing. Second, SWfMSs could optimize the task scheduling while ensuring provenance support with the consideration of different bandwidths among different sites. Third, the communication between two sites is generally achieved by having two nodes, each at one of the two sites, communicating directly, which is not efficient in a multisite cloud. For instance, selecting several nodes at one site to send or receive data to or from several nodes at another site could exploit parallel data transfer and make it more efficient.

2.5 Overview of Existing Solutions

In this section, we illustrate SWf parallel execution solutions in existing SWfMSs. This section starts by a short presentation of parallel processing frameworks such as MapReduce. Although they are not full-fledged SWfMS, they do share techniques in common and are often used for complex scientific data analyses, or in conjunction with SWfMS to deal with big data [183]. Then, the section introduces eight widely used SWfMSs and a science gateway platform. Finally, the section ends with concluding remarks.

2.5.1 Parallel Processing Frameworks

Parallel processing frameworks enable the programming and execution of big data analysis applications in massively parallel computing infrastructures.

MapReduce [55] is a popular parallel processing framework for shared-nothing clusters, *i.e.* highly-scalable clusters with no sharing of either disk or memory among computers. MapReduce was initially developed by Google as a proprietary product to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or URL access frequencies. Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop [185].

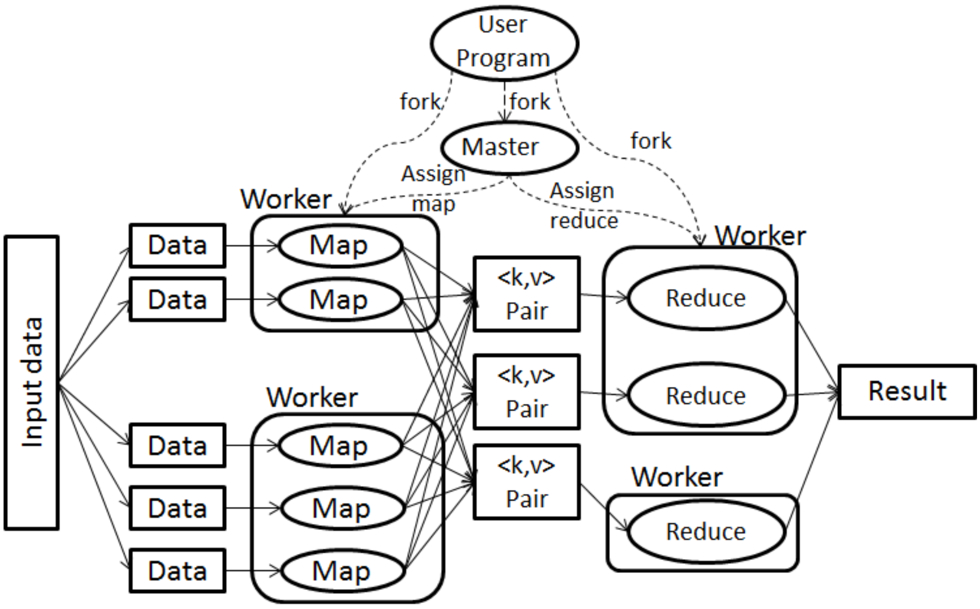


Figure 2.12: **MapReduce execution process.**

MapReduce includes only two types of operations, *map* and *reduce*. The Map operation is applied to each record in the input data set to compute one or more intermediate (key,value) pairs. The Reduce operation is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, Map and Reduce can be automatically processed in parallel, on different data chunks using many computer nodes in a cluster.

MapReduce execution proceeds as follows (see Figure 2.12). First, the users submit their jobs composed of MapReduce functions to a scheduling system. When the user program calls the MapReduce job, the MapReduce library in the user program splits the input data into several chunks. A MapReduce job consists of one Map function and one Reduce function. Then, the library makes several copies of the functions and distribute the copies into available computers. One copy is the master while the others are workers that are assigned tasks by the master. The master attempts to schedule a Map task, which is composed of a copy of the map function and corresponding input data chunks, to an idle worker. The worker that is assigned a Map task processes the (key,value) pairs of input data chunks and puts the intermediate (key,value) pairs in memory. The intermediate data is written to local disk periodically after being partitioned into several regions and the location information of this data is passed to the master. The combination of a copy of Reduce function and related intermediate data chunks is a Reduce task. The worker, which is assigned a Reduce task, reads the corresponding intermediate (key,value) data and sorts the data by grouping the data of the same key together. Then the sorted data is passed to Reduce tasks, which process the data and append their output data to a final output file. When all the map tasks and reduce tasks are completed, the master wakes up the user program.

Hadoop is an open source framework that supports MapReduce in a shared-nothing cluster. It uses Hadoop Distributed File System (HDFS) as storage layer (see Section 2.4.2). In Hadoop, MapReduce programs take input data from HDFS and put the final result and execution logs back to HDFS. Using Hadoop framework for SWf parallel execution can facilitate the implementation of SWfMSs and offer good compatibility for MapReduce programs. Hadoop is extended to multiple sites while the existing approaches do not consider the provenance support or different bandwidths among different sites for the task scheduling [184]. However, Hadoop can be used for executing SWfs by with combination of SWfMSs. For instance, Wang *et al.* [183] propose an architecture that combines Kepler with Hadoop so that Kepler can represent an activity as a MapReduce program and exploit the Hadoop framework to execute tasks. While designing a SWf with MapReduce activities, the input path, output path and result for the MapReduce activities should be specified through the Kepler GUI. Inside of the MapReduce activity, the input key, input value (input list) and output list (or output value) for the Map function (or Reduce function) should be specifed through the GUI. During the execution of a MapReduce activity in the Kepler/Hadoop system, Kepler first transfers all the input data into HDFS. Then, it runs the Map function followed by the Reduce function in the Hadoop system. Finally, it retrieves the output data from HDFS and stores it to the local file system. This approach enables Kepler to outsource data parallelism of a MapReduce activity to Hadoop, yet loosing control of activity execution.

Pig [143] is an interactive, or script-based, execution environment atop MapReduce. It supports *Pig Latin*, a declarative worflow language to express large dataset analysis. PigLatin resembles SQL, with a more procedural style, and allows expressing sequences of activities that get translated in MapReduce jobs. Pig Latin can be extended using user-defined functions written in different languages like Java, Python or JavaScript. Pig programs can be run in three different ways: with a script interpreter, with a command interpreter or embedded in a Java program. Pig performs some logical optimization, by grouping activities into MapReduce jobs. For executing the activities, Pig relies on Hadoop to schedule the corresponding Map and Reduce tasks. Hadoop provides the functionality such as load-balancing and fault-tolerance. However, task scheduling and data dispatching in Hadoop is not optimized for the entire SWf.

Dryad [98] is another parallel processing framework developed by Microsoft (which eventually adopted Hadoop MapReduce). Similar to a SWf, a Dryad job is represented as a DAG. To compose a Dryad job, the users can extend Dryad by implementing new composition operations based on two standard compositions: $A >= B$ and $A >> B$ (see Figure 2.13). During job execution, Dryad refines the job graph in order to reduce network consumption. Once all the input data of one program (vertex) is ready, the corresponding programs (vertices) are put into a scheduling queue, which applies a greedy scheduling strategy. Then, Dryad re-executes corresponding failed programs several times for fault tolerance.

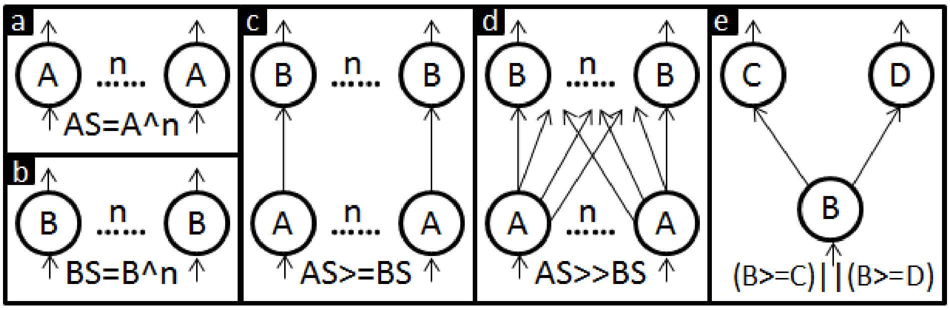


Figure 2.13: **Dryad operations [98].** Circles represent programs and arrows represent data dependencies. Box (a) and box (b) illustrate program cloning with the ^ operator. We note each program P of type x as P_x. The operation $AS >= BS$ in (c) means that each P_a has an input data flow to each P_b. The operation $AS >> BS$ in (d) expresses complete bipartite composition. Box (e) shows an operation by combining the data from Pb to P_c and P_d.

2.5.2 SWfMS

Most SWfMSs implement the five layer architecture discussed in Section 2.2.2. We selected eight typical SWfMSs and a gateway framework to illustrate their techniques: Pegasus, Swift, Kepler, Taverna, Chiron, Galaxy, Triana [173], Askalon [68] and WS-PGRADE/gUSE [105]. Pegasus and Swift have excellent support for scalability and high-performance of data-intensive SWfs, with reported results using more than a hundred thousand of cores and terabytes of data during SWf execution [61, 201]. Kepler, Taverna, Triana have a GUI for desktop computers. Chiron is widely used because of a powerful algebraic approach for SWf parallelization. Galaxy integrates a GUI that can be accessed through web browsers. Triana is able to use P2P services. Askalon implements both desktop and web GUI and has been adapted to cloud environments. WS-PGRADE/gUSE is a widely used gateway framework, which enables SWf execution in Distributed Computing Infrastructures (DCI) with a web interface for users.

Pegasus, Swift, Kepler, Taverna and WS-PGRADE/gUSE are widely used in astronomy, biology, and so on while Galaxy can only execute bioinformatics SWfs. Pegasus, Swift and Chiron design and execute a SWf through a textual interface while Kepler, Taverna, Galaxy, Triana, Askalon and WS-PGRADE/gUSE integrate a GUI for SWf design. All of the eight SWfMSs and the gateway framework support SWf specification in a DAG structure while Swift, Kepler, Chiron, Galaxy, Triana and Askalon also support SWfs in a DCG structure [195]. Users can share SWf information from Taverna, Galaxy, Askalon and WS-PGRADE/gUSE. All of them support independent parallelism. All of them support dynamic scheduling and three of them (Pegasus, Kepler and WS-PGRADE/gUSE) support static scheduling. All the eight SWfMSs and the gateway framework support SWf execution in both grid and cloud environments. A brief comparison of these eight SWfMSs and the gateway framework is given in Table 2.1.

2.5.2.1 Pegasus

Pegasus[7] [61] is widely used in multiple disciplines such as astronomy, bioinformatics, climate modeling, earthquake science, genome analysis, etc. Pegasus has interesting features: portability on different infrastructures such as grid and cloud, optimized scheduling algorithms; good scalability, support for provenance data that can be used for debugging, data transfer support for data-intensive SWfs, fault-tolerance support, detailed user guide [15] and available package in the Debian repository.

Pegasus consists of five components, *i.e.* mapper, local execution engine, job scheduler, remote execution engine and monitoring component. The mapper generates an executable SWf and partitions SWf to fragments based on an abstract SWf provided by the users. The local execution engine submits the fragments to execution engines according to dependencies. The job scheduler schedules the fragments to available remote execution engines. The remote execution engine manages the execution of the tasks of the fragments. Finally, the monitoring component monitors the execution of the SWf.

The process of executing SWfs in Pegasus is shown in Figure 2.14. In the presentation layer, Pegasus takes an abstract SWf represented as a DAX (DAG in an XML file). Pegasus provides programmatic APIs in Python, Java, and Perl for DAX generation[15]. Pegasus exploits a lightweight web dashboard to monitor and the execution of SWfs for a user. In the user services layer, Pegasus supports SWf monitoring through Stampede monitoring infrastructure [89, 159]. Pegasus also support provenance data gathering and

[7] Pegasus: http://pegasus.isi.edu/

Table 2.1: **Comparison of SWfMSs.** A categorization of SWfMSs based on supported SWf structures, SWf information sharing, UI types, parallelism types and scheduling methods. “activity” means that this SWfMS supports both independent parallelism and pipeline parallelism. WPg represents WS-PGRADE/gUSE. WP indicates that the interface is a web-portal.

SWfMS	structures	SWf sharing	UI type	parallelism	scheduling
Pegasus	DAG	not supported	GUI & textual	data & independent	static & dynamic
Swift	DCG	not supported	textual	activity	dynamic
Kepler	DCG	not supported	GUI	activity	static & dynamic
Taverna	DAG	supported	GUI	data & activity	dynamic
Chiron	DCG	not supported	textual	data & activity & hybrid	dynamic
Galaxy	DCG	supported	GUI (WP)	independent	dynamic
Triana	DCG	not supported	GUI	data & activity	dynamic
Askalon	DCG	supported	GUI	activity	dynamic & hybrid
WPg	DAG	supported	GUI (WP)	data & independent & hybrid	static & dynamic

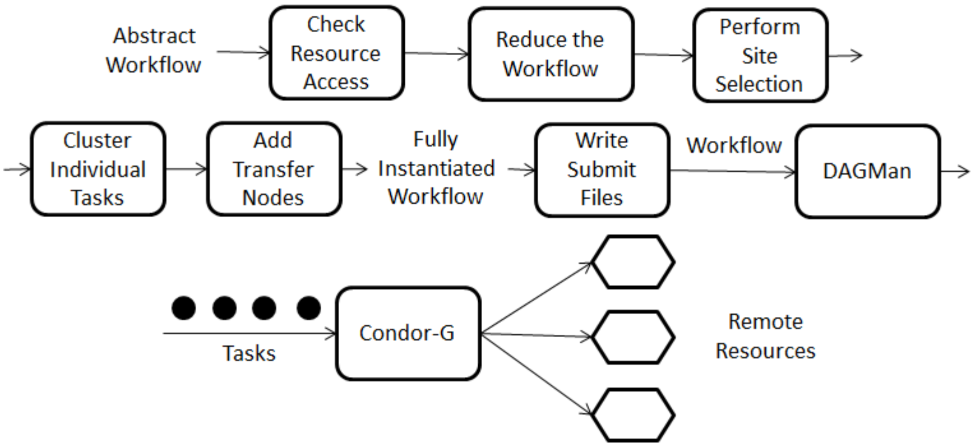


Figure 2.14: **Pegasus SWf execution process [60].**

querying through a Pegasus/Wings framework [108]. The provenance data or monitoring data come from the log data gathered during SWf execution.

In the WEP generation layer, the mapper reduces the abstract SWf by checking available intermediate data in the available computing nodes. The intermediate data can come from the previous execution of the same SWf or the execution of other SWfs that contain several common activities. In addition, Pegasus inserts the data transfer activities, *e.g.* data stage-in, in the DAG for SWf execution. The mapper component can realize SWf partitioning through three methods [38, 39, 61]. As discussed in Section 2.2.2.3, Chen and Deelman [38] propose a SWf partitioning method under storage constraints at each site. This partitioning method is used in a multisite environment with dynamic computing provisioning as explained in [37]. Another method is balanced task clustering [39]. The SWf is partitioned into several fragments which have almost the same workload. This method can realize load balancing for homogeneous computing resources. The last method is to cluster the tasks of the same label [61]. To use this method, the tasks should be labeled by users. In the WEP execution layer, the job scheduler may perform site execution based on standard algorithms (random, round-robin and min-min), data location and the significance of computation and data in SWf execution. For example, the job scheduler moves computation to the data site where big volume of data is located and it sends data to compute site if computation is significant. At this point, Pegasus schedules the execution of tasks within a SWf engine such as DAGMan. In Pegasus, DAGMan sends the concrete executable tasks to Condor-G, a client tool that can manage the execution of a bag of related tasks on grid-accessible computation nodes in the selected sites. Condor-G has a queue of tasks and it schedules a task in this queue to a computing node in the selected site once this computing node is idle [75, 115]. Pegasus handles task failures by retrying the corresponding part of SWfs or transfer the data again with a safer data transfer method. Through these mechanisms, Pegasus hides the complex scheduling, optimization and data transmission of SWfs from SWfMS users.

In the infrastructure layer, Pegasus is able to use computing cluster, grid (including desktop grids) and cloud to execute a SWf. It can exploit a shared file system, local storage resources at each computing node or cloud storage, *e.g.* Amazon S3, for data storage and it provides static computing and storage provisioning for SWf execution. Pegasus can be directly executed in a virtual cluster in cloud [101] while it can also use dynamic scheduling algorithms [126, 133] for budget constraint and time limit through Wrangler [102], a dynamic provisioning system in the cloud.

2.5.2.2 Swift

Similar to Pegasus, Swift [201] has been used in multiple disciplines such as biology, astronomy, economics, neuroscience, etc. Swift grew out of the GriPhyN Virtual Data System (VDS) whose objective is to express, execute, track the results of SWfs through program optimization and scheduling, task management, and data management. Swift has been revised and improved its (already) large-scale performance into the Turbine system [192].

Swift executes data-intensive SWfs through five functional phases: program specification, scheduling, execution, provenance management and provisioning. In the presentation layer, Swift takes a SWf specification that can be described in two languages: XDTM and SwfitScript. XDTM is an interface to map the logical structure of data to physical resources. SwiftScript defines the sequential or parallel computational procedures that operate on the data defined by XDTM. In the user services layer, provenance data is available for the users.

In the WEP generation layer, the SwiftScript is compiled to an abstract computation specification. Swift performs partitioning by generating corresponding abstract WEPs for each site [201]. In the WEP execution layer, the abstract WEPs are scheduled to execution sites. The Karajan SWf execution engine is used by Swift to realize the functions such as data transfer, task submission, grid services access, task instantiation, and task schedule. Swift runtime callouts provide the information for task and data scheduling and offer status reporting, which shows the SPs. During SWf execution, provenance data is gathered by a launcher program (*e.g.* kickstart). Swift achieves fault tolerance by retrying the failed tasks and provides a restart log when the failures are permanent.

In the infrastructure layer, the provisioning phase of Swift provides computing resources in a computer cluster, grid, and cloud through a dynamic resource provisioner for each execution site. In the cloud, Swift takes advantage of Coasters [90] to manage communication, task allocation and data stage for scientific SWf execution while it is not optimized for dynamic provisioning of VMs and storage. Figure 2.15 depicts the Swift system architecture.

2.5.2.3 Kepler

Kepler [21, 20] is a SWfMS built upon the Ptolemy II system from the Kepler[8] project. It allows to plug in different execution models into SWfs. Kepler is used in many projects of

[8] Kepler project: https://kepler-project.org/

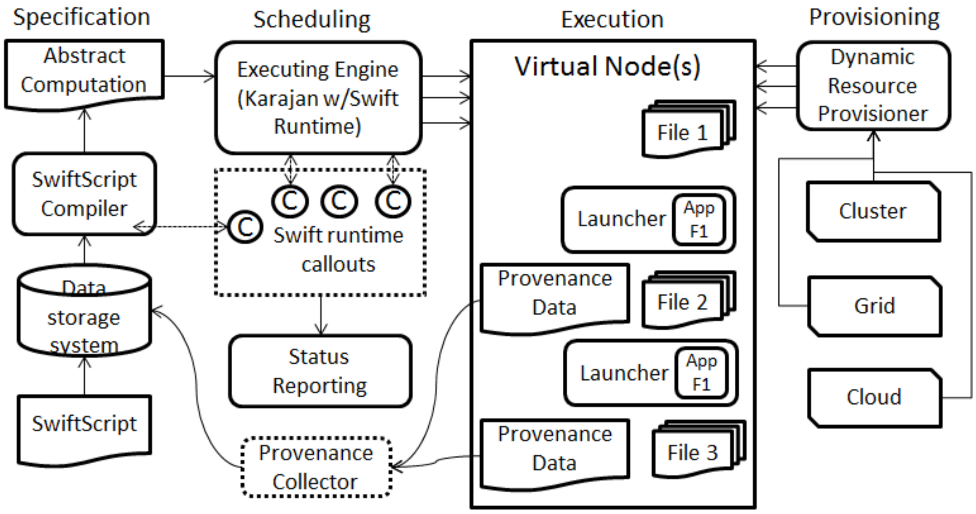


Figure 2.15: **Swift system architecture [201].**

various disciplines such as oceanography[9], data management[10], and biology[11] etc. Kepler integrates a powerful graphical workbench (shown in Figure 2.16). In the presentation layer, each individual reusable SWf step is implemented as an actor that can be signal processing, statistical operations, etc. SWf activities are associated to different actors as shown in Figure 2.16.

In the user services layer, the provenance functionality in Kepler is realized by corresponding actors such as Provenance Recorder (PR) [19]. PR records the information of SWf execution such as context, input data, associated metadata, SWf outputs, etc.

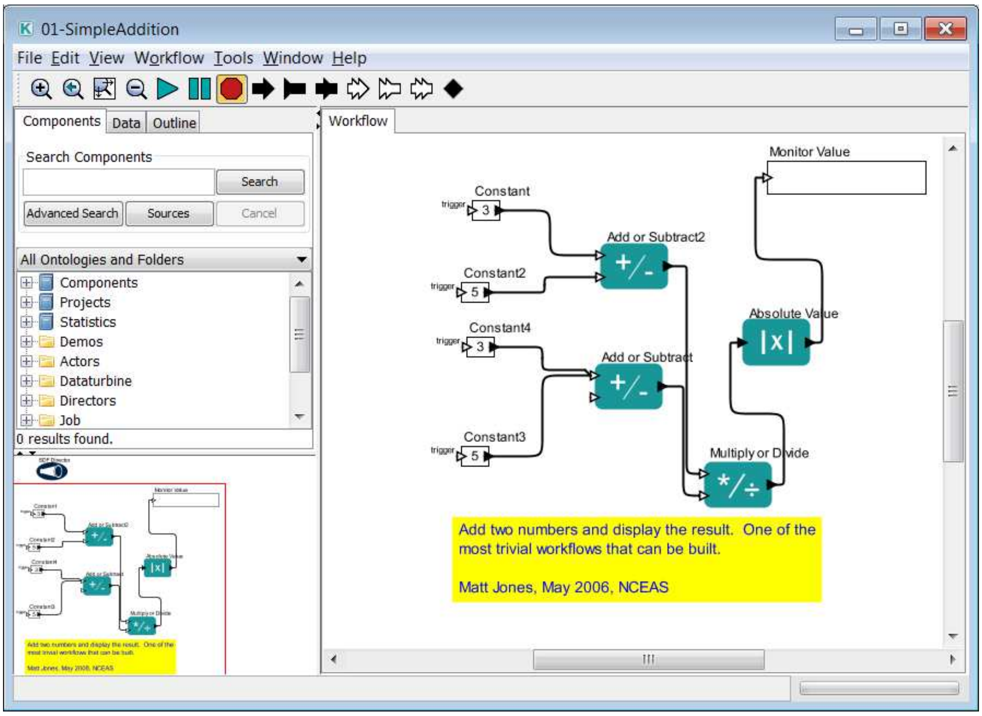
In the WEP generation layer, the SWf is handled by a separate component named director. Kepler supports several directors and each director corresponds to a unique model of execution, which is a model of WEP. The director generates executable tasks to achieve activity parallelism (pipeline parallelism and independent parallelism).

In the WEP execution layer, Kepler exploits static or dynamic scheduling according to the director that is used during SWf execution [123, 33]. The fault tolerance functionality of Kepler can be achieved by a framework that provides three complementary mechanisms. The first mechanism is a forward recovery mechanism that retries the failed tasks. The second mechanism offers a check-pointing mechanism that resumes the execution in case of a failure at the last saved state. The last one is a watchdog process that analyzes the SWf execution based on provenance data and sends an appropriate signal and possible course of action to the SWf engine to handle it. Kepler executes SWfs in parallel through web services, grid-based actors or Hadoop framework. Kepler can execute SWfs by using external execution environments such as SAS, Matlab, Python, Perl, C++ and R (S+) using corresponding actors.

[9] REAP project: https://kepler-project.org/users/projects-using-kepler-1/reap-project

[10] Scientific Data Management Center: https://sdm.lbl.gov/sdmcenter/

[11] Clotho project: http://www.clothocad.org/



Add two numbers and display the result. One of the most trivial workflows that can be built.

Matt Jones, May 2006, NCEAS

Figure 2.16: **Kepler workbench.**

In the infrastructure layer, Kepler can achieve data access through an OpenDBConnection actor for data in a database and an EMLDataSource actor for ecological and biological datasets. Kepler is compatible with the cloud through Kepler EC2 actors, which can directly create a set of EC2 VMs and attach Elastic Block Store volumes to running VMs [182].

2.5.2.4 Taverna

Taverna [132] is an open-source SWfMS from the ^{my}Grid project to support workflow-based biological experiments. Taverna is used in multiple areas such as astronomy, bioinformatics, chemistry etc. In the presentation layer, Taverna takes a GUI for designing SWfs and showing monitoring information while it uses a textual language to represent a SWf as a DAG [195]. The SWfs can be designed in Taverna installed in the user's computer or an online web server. Moreover, this GUI can be installed in an Android mobile [199]

In the user services layer, Taverna uses a state machine for the activities to achieve SWf monitoring [141]. The SWfs designed through Taverna can be shared through "myExperiment" social network [191]. It gathers provenance data from local execution information and the remotely invoked web services [140].

In the WEP generation layer, Taverna automatically optimizes the SWf structure by identifying complex parts of structures and simplifies them for easier design and parallelization [44]. Taverna links the invocation of web services and the activities and checks the availability of the needed web services for generating a WEP. In the WEP execution layer, Taverna relies on web and grid services for task execution.

In the infrastructure layer, Taverna is able to use the computing resources from grid or cloud. It also stores execution data in a database.

2.5.2.5 Chiron

Chiron exploits a database approach [146] to manage the parallel execution of data-intensive SWfs. In the presentation layer, it uses an algebraic data model to express all data as relations and represent SWf activities as algebraic expressions in the presentation layer. A relation contains sets of tuples composed of basic attributes such as integer, float, string, and file references, etc. An algebraic expression consists of algebraic activities, additional operands, operators, input relations and output relations. An algebraic activity contains a program or an SQL expression, and input and output relation schemas. An additional operand is the side information for the algebraic expression, which can be relations or a set of grouping attributes. There are six operators: Map, SplitMap, Reduce, Filter, SRQuery and MRQuery (see Section 2.2.2.1 for the function of each operator). In the user services layer, Chiron supports SWf monitoring, steering and gathers provenance data based on algebraic approach.

In the WEP generation layer, a SWf is wholly expressed in an XML file called conceptual model. Chiron supports all types of parallelism (data parallelism, independent parallelism, pipeline parallelism, hybrid parallelism) and optimizes SWf scheduling by distinguishing between blocking activities, *i.e.* activities that require all their input data to proceed, and non blocking, *i.e.* that can be pipelined. Chiron generates concrete executable tasks for each activity and schedules the tasks of the same fragments to multiple computing nodes. Chiron uses two scheduling policies, called blocking and pipeline in [64]. Let *A* be a task that produces data consumed by a task *B*. With the blocking policy, *B* can start only after all the data produced by *A* are ready. Hence, there is no parallelism between *A* and *B*. With the pipeline policy, *B* can start as soon as some of its input data chunks are ready. Hence, there is pipeline parallelism. This pipeline parallelism is inspired by DBMS pipeline parallelism in [146]. Moreover, Chiron takes advantage of algebraic approach for SWf execution optimization to generate a WEP.

In the WEP execution layer, Chiron uses an execution module file to specify the scheduling method, database information and input data information. Chiron exploits dynamic scheduling method for task execution. Chiron gathers execution data, light domain data and provenance data into a database structured by following the PROV-Wf [46] provenance model. The execution of tasks in Chiron is based on MPJ [35], an MPI-like message passing system. In the infrastructure layer, Chiron exploits a shared-disk file system and database for data storage.

Chiron is adapted to the cloud through its extension, Scicumulus [51, 52], which supports dynamic computing provisioning [50]. The architecture of Scicumulus contains three layers and four corresponding tiers: desktop layer for client tier, distribution layer for distribution tier, execution layer for execution tier and data tier. The desktop layer is to compose and execute SWfs. The distribution layer is responsible for parallel execution of activities in the cloud. The execution layer manages activity execution in VM instances. Finally, the data tier manages the related data during SWf execution. Scicumulus exploits hybrid scheduling approaches with dynamic computing provisioning support. Furthermore, Scicumulus uses services such as SciDim [53] to determine an initial virtual cluster size through a multi-objective cost function and provenance data under budget and time limits. Moreover, Scicumulus can be coupled with SciMultaneous, which is used to manage fault tolerance in the cloud [45].

2.5.2.6 Galaxy

Galaxy is a web-based SWfMS for genomic research. In the presentation layer, Galaxy provides a GUI for designing SWfs through browsers. It can be installed in a public web server (https://usegalaxy.org/) or a private server to address specific needs.

In the user services layer, users can upload data from a user's computer or online resources and share SWf information including SWfs, SWf description information, SWf input data and SWf provenance data in a public web site. Moreover, users can import SWfs from the "myExperiment" [191] social network [83].

In the WEP generation layer, Galaxy manages the dependencies between each activity for parallelization. In the WEP execution layer, Galaxy generates concrete tasks for each activity, puts the tasks in a queue to be submitted, and monitors the task status (in queue, running or completion) [106]. Through this mechanism, Galaxy exploits dynamic scheduling to dispatch executable tasks. Galaxy uses Gridway to execute tasks in the Grid. Gridway manages a task queue and the tasks in a queue are executed in an available computing node that is selected according to a greedy approach, *i.e.* requests are sent to all the available computing nodes while the node that has minimum response time is selected [96].

In the infrastructure layer, Galaxy can exploit Globus [116] and CloudMan [17] to achieve dynamic computing and storage provisioning such as dynamic VM inserting and removing and shared-disk file system construction across computing nodes. Galaxy is adapted to cloud environment by CloudMan [17] middlewear, which can create Amazon EC2 clusters based on a Bio-Linux machine image, dynamically change cluster size and attach S3 storage to the clusters.

2.5.2.7 Triana

Triana [173] is a SWfMS initially developed as a data analysis tool within the GEO 600 project[12]. It provides a GUI in the presentation layer. In the user services layer, it implements the Stampede monitoring infrastructure [181] (see Section 2.2.2.2).

In the WEP generation layer, Triana exploits components to realize different data processing functions similar to Kepler actors. In the WEP execution layer, Triana supports the grid Application Toolkit (GAT) API for developing grid-oriented components. Triana

[12] http://www.geo600.org/

also uses the Grid Application Prototype (GAP) as an interface to interact with service-oriented networks. The GAP contains three bindings, *i.e.* implemented GAP, such as P2PS and JXTA to use P2P network and Web services binding to invoke Web services.

In the infrastructure layer, Triana can employ computing resources in the grid or cloud. Triana uses RabbitMQ12 [13], a message broker platform, to realize the communication among different VMs in order to run SWfs in the cloud environment.

2.5.2.8 Askalon

Askalon [68] is also a SWfMS initially designed for a grid environment. In the presentation layer, it provides a GUI, through which a SWf can be modeled using Unified Modeling Language. It also exploits an XML-based language to model workflows. In the user services layer, it provides on-line SWf execution monitoring functionality through SWf execution monitoring and dynamic workflow steering to deal with exceptions in dynamic and unpredictable execution environments [153].

In the WEP generation layer, Askalon optimizes the SWf representation with loops, *i.e.* within DCG structures, to a DAG SWf structure. In the WEP execution layer, Askalon exploits an execution engine to provide fault-tolerance at the levels of workflow, activity and control-flow. It can exploit static and hybrid scheduling, *e.g.* rescheduling because of unpredictable changes in the execution environment.

In the infrastructure layer, Askalon uses a resource manager to discover and reserve available resources and to deploy executable tasks in the grid environment. Askalon is able to execute SWfs in cloud environment by dynamic creation of VMs with available cloud images [145]. In the cloud environment, Askalon can estimate the cost of SWf execution by simulation [144] and provide dynamic resource provisioning and task scheduling under budget constraint [69].

Moreover, Askalon can execute SWfs in a federated multisite cloud [145], *i.e.* a multisite cloud composed of resources from different providers. Nevertheless, it schedules tasks in computing nodes without considering the organization of computing resources, *i.e.* which VMs are at the same site, for optimization. This method just takes the VMs as the grid computing nodes without considering the features of multisite resources, *e.g.* the difference of data transfer rate, resource sharing for intra-site and inter-site, etc.

2.5.2.9 WS-PGRADE/gUSE

WS-PGRADE/gUSE is a science gateway framework widely used in various disciplines such as biology [79], seismology [111], astronomy [163], and neuroscience [164]. It is an open source software [105] used for teaching [142], research [109] and commercial activities [110].

The architecture of WS-PGRADE/gUSE framework is shown in Figure 2.17. WS-PGRADE portal is a web-portal interface to help the users designing SWfs. The grid and cloud User Support Environment (gUSE) is a middle layer for different user services. The

[13] Triana in cloud: `http://www.trianacode.org/news.php`

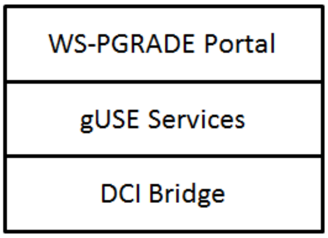


Figure 2.17: **Architecture of WS-PGRADE/gUSE [87].**

DCI bridge is a web service based application that provides access to divers infrastructures such as grid and cloud [105].

In the presentation layer, the WS-PGRADE portal [104] has a web browser based interface, which supports the definition of different kinds of SWfs, including meta-workflows and parameter sweep workflows. The meta-workflows can contain embedded SWfs, which are supported by the SHIWA repository, as sub-workflows. In the user services layer, WS-PGRADE supports SWf information sharing between the users of WS-PGRADE/gUSE through a build-in SWf repository, which enables publishing, searching and downloading programs, SWf designs and templates [24]. A SWf template is an available SWf pattern that can be changed to other SWfs by modifying corresponding parameters. In addition, it can exploit the SHIWA repository [151] to support SWf sharing between the users of different SWfMSs. The monitoring functionality is supported by the gUSE services. However, the framework lacks provenance support.

In the WEP generation layer, a SWf is represented in XML format. The framework may use data and independent parallelism according to the structure of the SWf [24]. The DCI bridge dynamically schedules the tasks through a submit queue. The task execution is handled by web services enabled by a web container, *e.g.* Tomcat or Glassfish.

Provisioning and data storage are provided by the DCI Bridge [113] and CloudBroker framework. The DCI Bridge can dynamically create VMs through existing images and cloud configurations. The CloudBroker is able to exploit resources distributed in multiple clouds [71], including major cloud types , *e.g.* Amazon, OpenStack [99] and OpenNebula [131]. Moreover, it can take advantage of GPUs to execute the SWfs that invoke GPU programs [24].

The SHIWA simulation platform enables reusing of SWfs in ten different SWfMS environments [174]. First, the users can search for SWfs in the SHIWA repository, where SWf developers can upload available programs or SWfs. Then, the SWfs can be downloaded and adjusted to an Interoperable Workflow Intermediate Representation (IWIR) language to compose a meta-workflow [151]. Then, when the meta-workflow is submitted in the platform, each sub-workflow of the meta-workflow can be scheduled to the appropriate SWfMS execution environment.

2.5.3 Concluding Remarks

We observed that some SWfMSs take advantage of parallel processing frameworks such as Hadoop as lower-level tools to parallelize SWf execution and schedule tasks. This is a straightforward approach to extend a SWfMS with parallel processing capabilities. However, it lacks the capability to perform parallelization according to the entire SWf structure. Our comparative presentation of eight SWfMSs showed that most SWfMSs do not exploit hybrid parallelism (only Chiron does) and hybrid scheduling methods (only Askalon does), which may bring the highest degrees of parallelism and good load balancing.

Although there has been much work on data-intensive SWf management, we believe there is a lot of room for improvement. First, input data staging needs more attention. Most SWfMSs just do this as a preprocessing step before actual SWf execution. For data-intensive SWfs, this step may take a very long time, for instance, to transfer several gigabytes to a computing node. Integrating this step as part of the WEP can help optimize it, based on the activities and their execution at computing nodes. Second, SWf partitioning strategies should pay attention to the computing capabilities of the resources and data to be transferred across computing nodes, as this is a major performance and cost factor, and not focus only on one constraint, *e.g.* storage limitation. Third, although there is already study on the VM provisioning techniques, it remains open problem to execute SWfs in a multisite cloud with the consideration of distributed data, features of the cloud sites. In addition, a cost model to estimate the execution of SWfs in a multisite cloud and corresponding VM provisioning methods are also critical to the multisite SWf execution in the clouds. Fourth, the structure of SWfMSs is generally centralized (the new version of Swift is not centralized). In this structure, a master node manages all the optimization and scheduling processes. This master node becomes a single point of failure and performance bottleneck. Distributed and P2P techniques [147] could be applied to address this problem. Fourth, although most SWfMSs are capable to produce provenance data, they lack integrated UI with provenance data which is very useful for SWf steering.

2.6 Conclusion

In this chapter, we discussed the current state of the art of the SWfMSs, parallel execution of data-intensive SWfs in different infrastructures, especially in the cloud.

First, we introduced the definitions in SWf management, including SWfs and SWfMSs. In particular, we illustrated the representation of SWfs with real examples from astronomy and biology. Then, we presented in more details a five-layer functional architecture of SWfMSs and the corresponding functions. Special attention has been paid to data-intensive SWfs by identifying their features and presenting the corresponding techniques.

Second, we presented the basic techniques for the parallel execution of SWfs in SWfMSs: parallelization and scheduling. We showed how different kinds of parallelism (coarse-grained parallelism, data parallelism, independent parallelism and pipeline parallelism) can be exploited for parallelizing SWfs. The scheduling methods to allocate tasks to computing resources can be static or dynamic, with different trade-offs, or hybrid to combine the advantages of static and dynamic scheduling methods. SWf scheduling may include an optimization phase to minimize a multi-objective function, in a given context (cluster, grid, cloud). However, unlike in database query optimization, this scheduling optimization phase is often not explicit and mixed with the scheduling method.

Third, we discussed cloud computing and the basic techniques for parallel execution of SWfs in the cloud, including single site cloud and multisite cloud. We discussed three categories of cloud computing, multisite management in the cloud and data storage in the cloud. The data storage techniques includes shared-disk file systems and distributed file systems. Then, we analyzes the parallelization techniques of SWfs in both single site cloud and multisite cloud.

Fourth, to illustrate the use of the techniques, we introduced the recent parallelization frameworks such as MapReduce and gave a comparative analysis of eight popular SWfMSs (Pegasus, Swift, Kepler, Taverna, Chiron, Galaxy, Triana and Askalon) and a science gateway framework (WS-PGRADE/gUSE).

The current solutions for the parallel execution of SWfMSs are appropriate for static computing and storage resources in a grid environment. They have been extended to deal with more elastic resources in a cloud, but only with single site. Although some SWfMSs such as Swift and Pegasus provide some functionality to execute SWfs in the multisite environment, this is generally done by reusing techniques from grid computing or simple dynamic provisioning and scheduling mechanisms, without exploiting new data storage and data transfer capabilities provided by a multisite cloud. Our analysis of the current techniques of SWf parallelization and SWf execution has shown that there is a lot of room for improvement. And we proposed research directions addressed in this thesis (the first four points) and for future research, which we summarize as follows:

1. SWf partitioning: To partition a SWf into several parts based on resources in each site is also a difficult optimization problem in a multisite environment.

2. VM provisioning: In order to achieved multiple objectives, *e.g.* reducing execution time and monetary cost, a cost model and corresponding VM provisioning method is critical to SWf execution in a single cloud site or a multisite cloud. The cost model could be optimized by considering the sequential workload and the cost to initiate the cloud sites, *i.e.* creation and configuration of VMs.

3. Fragment scheduling: In order to execute a SWfs in a multisite cloud, it is also important to consider the distributed data at each site and different prices to use VMs at each site. A cost model and scheduling methods remain open problem for executing SWfs in a multisite with multiple objectives, *e.g.* reducing execution time and monetary cost.

4. Task scheduling and data location: most SWfMSs do not take data location into consideration during task scheduling period. For data-intensive SWfs, a uniform scheduling method is needed to handle task and data scheduling at the same time.

Furthermore, SWfMSs should also consider provenance support for SWf execution with different bandwidths among different sites.

5. Multisite Data transfer: the current data transfer methods between two sites are realized by two nodes, each of which located at a site. We believe that using multiple nodes at each site can achieve bigger bandwidths. However, the problem of mapping heterogeneous computing nodes at a site to the computing nodes at another site requires better algorithms.

Chapter 3

Scientific Workflow Partitioning in a Multisite Cloud

As the scale of the data increases, SWfMSs need to support SWf execution in High Performance Computing (HPC) environments. Because of various benefits, cloud emerges as an appropriate infrastructure for SWf execution. However, it is difficult to execute some SWfs at a cloud site because of geographical distribution of scientists, data and computing resources. Therefore, a SWf often needs to be partitioned and executed in a multisite environment. This chapter proposes a non-intrusive approach to execute SWfs in a multisite cloud with three SWf partitioning techniques. This chapter is based on [122].

Section 3.3 introduces our system model based on an adaptation of Chiron for multisite cloud. Then, Section 3.4 presents the Buzz SWf we use for experimentation. Section 3.5 details our three SWf partitioning techniques and Section 3.6 presents our experimental validation in Microsoft Azure. The experimental validation used an adaptation of Chiron SWfMS for Microsoft Azure multisite cloud. The experiment results reveal the efficiency of our partitioning techniques, and their superiority in different environments.

3.1 Overview of the Proposal and Motivations

Scientific experiments generally contain multiple computational activities to process experimental data and these activities are related by data or control dependencies. SWfs enable scientists to model these data processing activities together to be automatically executed. In a SWf, one activity may consist of several executable tasks for different parts of experimental data during SWf execution. SWfs exploit SWfMSs to manage SWf representation, execution and data sets in various computing environments. SWfMSs may exploit High Performance Computing (HPC) to execute SWfs within reasonable time. The HPC environment may be provided by a cluster, grid or cloud. Cloud computing, which promises virtually infinite resources, scalable services, stable service quality and flexible payment policies, has recently become a viable solution for SWf execution.

In general, one cloud site is sufficient for executing one user application. However, in the case of SWfs, some important restrictions may force their execution in a multisite cloud, *i.e.* a cloud with multiple distributed data centers, each being explicitly accessible to cloud users. For instance, some activities need to be executed at a cloud site trusted by the scientists for SWf monitoring without malicious attack, *i.e.* scientist security restriction; some activities need to be moved to another cloud site because of stored big input data at that site and the cost of transferring this big data to another site is very high, *i.e.* data transfer restriction; some activities have to be executed at a site where more computing resources are available, *i.e.* computing capacity restriction; some other activities may invoke special programs (instruction data), which are located at a specific cloud site and cannot be moved to another site because of proprietary reasons, *i.e.* proprietary restriction. The configuration data, which includes SWf representation files or SWf parameters, can be located at one site or distributed at different sites. In this situation, multisite cloud is appealing for data-intensive SWfs.

For a given application, a multisite cloud configuration, which is the configuration of Virtual Machines (VMs) at each site, can be homogeneous, with homogeneous computing capacity at each site, *e.g.* 8 VMs at sites 1 and 2, or heterogeneous, *e.g.* 8 VMs at site 1 and 2 VMs at site 2. The homogeneous configuration is obviously easier to deal with in terms of SWf partitioning and execution. However, even the homogeneous configuration makes it difficult to reproduce experiments as the allocation of VMs to real resources is typically controlled at runtime by the cloud provider. For instance, at time t_1, one VM may be allocated to a processor that is already very busy (*e.g.* running 16 other VMs) while at time t_2, the same VM may be allocated to an underloaded processor (*e.g.* running 2 other VMs).

In order to execute SWfs in a multisite cloud environment, a SWfMS can generate a Workflow Execution Plan (WEP) for SWf execution. Similar to the concept of Query Execution Plan (QEP) in distributed database systems [146], the WEP is a program that captures optimization decisions and execution directives, typically the result of compiling and optimizing a SWf. Since the multiple sites are interconnected but share nothing, the WEP includes a SWf partitioning result, which is the decision of partitioning a SWf into SWf fragments for independent execution. A SWf fragment (or fragment for short) can be defined as a subset of activities and data dependencies of the original SWf (see [138] for a formal definition). In order to execute a fragment within reasonable time at one site, the WEP generally contains a SWf parallelization plan, which parallelizes SWf execution.

We formulate the problems addressed in this chapter as follows. A SWf W = $\{V$,$E\}$ consists of a set of activities V and a set of dependencies E. A multisite cloud MS = $\{S_1$, S_2, ..., $S_n\}$ is composed of multiple cloud sites, each of which has multiple computing nodes and stores its own data (input data, instruction data or configuration data of W). The SWf execution time is the entire time to execute a SWf at a given execution environment. Given a SWf W and a multisite cloud MS, the multisite cloud execution problem is how to execute W in MS in a way that reduces execution time while respecting restrictions.

We propose a non-intrusive approach to execute a SWf in a multisite cloud. We propose three partitioning techniques with the consideration of restrictions. We validate our approach with a data-intensive SWf using Chiron [139] SWfMS in Microsoft Azure [5] cloud. The experiment results reveal that our approach can reduce execution time. Since the occupied computing resources do not change, the reduction of execution time may lead to less lease time, which corresponds to less monetary cost.

3.2 Related Work

SWf partitioning and execution in a multisite cloud remains a challenge and little work has been done. Deng *et al.* [62] adopt a clustering method based on data-data, data-activity and activity-activity dependencies. This method is adapted for SWf structures, but it may have big amount of data to be transferred among fragments. Chen *et al.* [37] present SWf partitioning based on storage constraints. Since a cloud environment can offer big storage capacity and the VMs can be mounted additional storage resources before or during SWf exeuciton, the storage capacity limitation is not general in a cloud environment. In addition, this method do not take data transfer cost and different computing capacity at each site into consideration. Tanaka and Tatebe [171] use a Multi-Constraint Graph Partitioning (MCGP) algorithm [107] to partition a SWf. This approach partitions a SWf by minimizing the removed dependency and balancing the activities in each fragment. However, this approach is appropriate only for homogeneous execution sites. In this chapter, we propose several partitioning techniques to address data transfer restriction and computing capacity restriction in the multisite cloud. Because of SWf partitioning, distributed provenance data is supported in the multisite cloud. In addition, data compression and file archiving is proposed to accelerate the data transfer between different cloud sites.

3.3 System Model

In this section, we present our system model based on Chiron SWfMS, its adaptation for multisite cloud and a SWf partitioner.

Chiron implements an algebraic approach for data-intensive SWfs proposed by Ogasawara *et al.* [139], to perform SWf parallelization and scheduling. This approach associates each activity with an operator, which has a semantic meaning for parallel execution. Since it models SWf data as relations similar to relational database management systems, this approach can optimize the entire SWf parallel execution based on well-founded relational algebra query optimization models [138].

The algebraic approach also allows online provenance data to be managed (and stored in a database by Chiron) for SWf activity monitoring [46]. Provenance data is the metadata that captures the derivation history of a dataset, including the original data sources, intermediate datasets, and the SWf computational steps that were applied to produce this dataset [46].

Chiron was initially developed for a one site execution environment as shown in Fig. 3.1-A. In a one site environment, a database is installed in a master node and all the computing nodes share storage resources through Network File System. Chiron achieves activity parallelism, data parallelism and dynamic scheduling for SWf parallelization as explained in Section 3.2. Chiron was modified to gather necessary produced data at the end of SWf execution at one site.

In a multisite cloud environment (see Fig. 3.1-B), all the sites have the same configuration as one site environment, *i.e.* a database installed in a master node and shared storage resources, while each site can have different numbers of slave computing nodes. We developped a SWf partitioner to automatically partition a processed SWf representation file into SWf fragment representation files when the first activity in each fragment is given. All the activities in a fragment are placed together in the processed SWf representation file. The SWf partitioner removes the dependencies in the original SWf and generates corresponding configuration files for each fragment. The execution of the generated fragments should respect the dependencies removed from the original SWf. Let us suppose that, in a SWf, activity A_2 consumes the output data produced by activity A_1. If these two activities are allocated to different fragments, their data dependencies will be removed from the explicit data dependencies. In this case, the fragment that contains activity A_2 should be executed after the execution of the fragment that contains activity A_1.

In order to reduce data transfer volume between different sites, we can use data compression and file archiving techniques. Data compression can just reduce the volume of transferred data to reduce transmission time. Through file archiving, we can also transfer the data at a relatively high speed to reduce transmission time. When transferring one file between two sites and the default transfer speed is less than the average transfer speed between two sites, the file transfer is accelerated (accelerating phase) at the beginning and decreased (decreasing phase) at the end. The data transfer rate remains high in the middle (high speed transfer phase). If we transfer several small files, there will be many accelerating and decreasing phases. But if we transfer a big file of the same data volume, there will be an accelerating phase and a decreasing phase while the high speed transfer phase will be longer. Therefore, the transmission speed of a big file is higher than that of several small files of the same data volume. In the remainder of the chapter, we note data refining as the combination of file archiving and data compression.

3.4 Use Case: Buzz Workflow

This section presents Buzz SWf [64], a data-intensive SWf, as a use case to illustrate our partitioning techniques. Buzz SWf is modeled and executed using Chiron. Buzz SWf searches for trends and measures correlations in published papers from scientific publications. This SWf uses data collected from bibliography databases such as the DBLP Computer Science Bibliography (DBLP) [4] or the U.S. National Institutes of Health's National Library of Medicine (PubMed). We used a DBLP 2013 XML file of $1,29$GB as input for Buzz SWf in our experiments.

Buzz SWf has thirteen activities (Fig. 3.2(a)). Each activity has a specific operator according to the algebraic approach. Boxes in the figure represent activities together with the

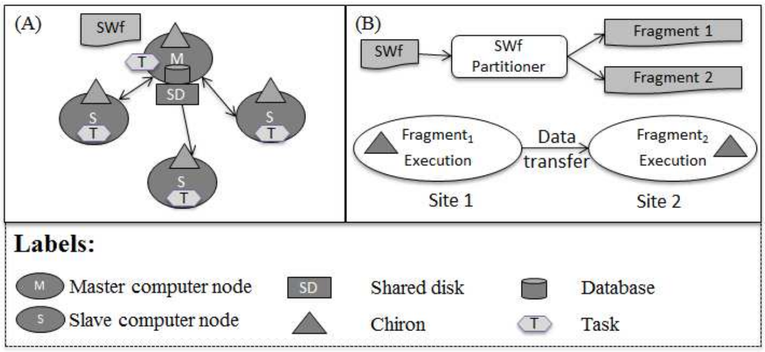
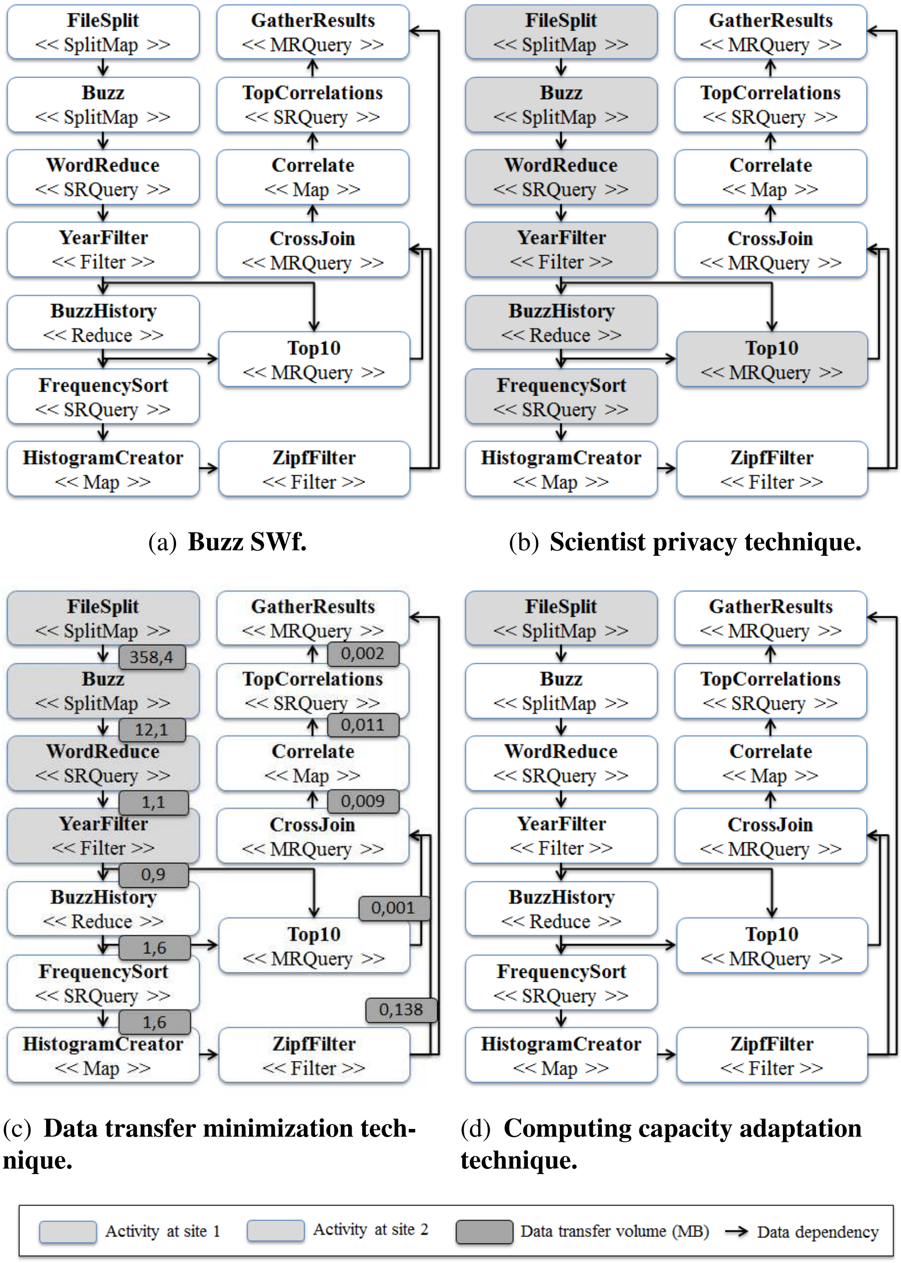


Figure 3.1: **The architecture of SWf Execution in Chiron.** Fig. A presents SWf execution in one site using Chiron before modification. Fig. B shows the multisite SWf execution using SWf partitioner and modified Chiron.

involved algebraic operators. *FileSplit* activity is responsible for gathering all scientific publications from bibliography databases. *Buzz* activity uses these publications to identify buzzwords (a word or phrase that can become popular for a specific period of time). *WordReduce* activity organizes these publications according to buzzword and publication year, and it also computes occurrences of identified words. Furthermore, *YearFilter* activity selects buzzwords that appeared in the publications after 1991, while *BuzzHistory* activity and *FrequencySort* activity create a history for each word and compute its frequency. With this information, *HistogramCreator* activity generates some histograms with word frequency varying the year. On the other hand, *Top*10 activity selects ten of the most frequent words in recent years, whilst *ZipfFilter* activity selects terms according to a Zipf curve that is specified by word frequency values [172]. Moreover, *CrossJoin* activity merges results from *Top*10 activity and *ZipfFilter* activity. *Correlate* activity computes correlations between the words from *Top*10 activity and buzzwords from *ZipfFilter* activity. Using these correlations, *TopCorrelations* activity takes the terms that have a correlation greater than a threshold and *GatherResults* activity presents these selected words with the histograms.

In the remainder of the chapter, we assume that there are two cloud sites (S_1 and S_2) to execute Buzz SWf. A fixed activity is located at a specific site and cannot be moved to another site because of additional restrictions, *i.e.* scientist security, data transfer, computing capacity and proprietary issues. We also assume that the first activity (*FileSplit*) is a fixed activity at S_1 since the input data located at S_1 is very big. In addition, we assume that the last activity (*GatherResults*) is a fixed activity at S_2 because of proprietary issues. Finally, scientists at S_2 need to monitor the execution of activity *HistogramCreator* without malicious attack and thus, *HistogramCreator* becomes a fixed activity at S_2, which is trusted by scientists.



(a) **Buzz SWf.**

(b) **Scientist privacy technique.**

(c) **Data transfer minimization technique.**

(d) **Computing capacity adaptation technique.**

Figure 3.2: **Buzz SWf partitioning.**

3.5 Workflow Partitioning Techniques

In this section, we present the approaches to partition a SWf into fragments. SWf partitioning is the process of dividing a SWf and input data into several fragments, so that each fragment can be executed at a site. It can be performed by DAG partitioning or data partitioning. DAG partitioning transforms a DAG composed of activities into a DAG composed of fragments while each fragment is a DAG composed of activities and dependencies. Data partitioning divides the input data of a fragment generated by DAG partitioning into several data sets, each of which is encapsulated in a newly generated fragment. This chapter focuses on the DAG partitioning.

The smallest granularity of fragment is an activity. Thus, we can encapsulate each activity in one fragment. We call this method activity encapsulation partitioning. However, this method is very simple and not optimized for the execution environment. In this chapter, we propose to partition the SWf according to the structure of the SWf or the execution environment. We propose three techniques for SWf partitioning, *i.e.* scientist privacy, data transfer minimization and computing capacity adaptation.

The first technique, Scientist Privacy (SPr), is for better supporting SWf activity monitoring under scientist security restriction. When a SWf contains an activity that needs to be monitored by scientists, this activity is defined as a locking activity to be executed at a trusted cloud site to avoid malicious attack during SWf execution. A locking activity implies that this activity and all the following activities should be assigned to a same fragment, in order to provide further activity monitoring. The following activities represent the activities that process the output data or the data produced from the output data of the locking activity. In order to partition a SWf based on scientist privacy technique, a SWfMS identifies the locking activity. Then it can partition the SWf by putting the locking activity and its available following activities (the following activities that are not fixed activities) into a fragment. According to this technique, Buzz SWf is partitioned into two fragments as shown in Fig. 3.2(b). As scientists need to analyze some histogram files produced by *HistogramCreator* activity at runtime at S_2 (trusted by scientists), *HistogramCreator* activity is handled as a locking activity. This activity and the following activities (*ZipfFilter*, *CrossJoin*, *Correlate*, *TopCorrelations* and *GatherResults*) are assigned to the same fragment while the other activities stay in another fragment.

The second technique is Data Transfer Minimization (DTM), which minimizes the volume of data to be transferred between different fragments. It is based on the fact that it takes much time to transfer certain amount of data from one site to another site. If the amount of data to be transferred between fragments is minimized, the time to transfer data between different sites can be reduced so as to reduce the entire execution time. During SWf design, the ratio between the volume of input data and output data can be offered. The scientists can estimate the data transfer for each data dependency based on the volume of input data of the SWf. The corresponding algorithm is shown in Algorithm 1.

In Algorithm 1, a set of data dependencies DS are chosen to be removed from the original SWf in order to partition the SWf. In this algorithm, the input data is viewed as a fixed activity. Lines $2 - 10$ select the dependencies to be removed in order to partition the

```
Input: $SWf$: a SWf; $S$: a set of sites
Output: $DS$: a set of dependencies to cut in order to partition the SWf
begin
 1: $DS \leftarrow \emptyset$
 2: for each $s \in S$ do
 3:     $A \leftarrow$ fixedActvities$(SWf, s)$
 4:     $A' \leftarrow$ fixedOutsideUnprocessedActivities$(SWf, s)$
 5:     for each $a \in A$ do
 6:         for each $a' \in A'$ do
 7:             $paths \leftarrow$ findPaths$(a, a')$
 8:             for each $path \in paths$ do
 9:                 $ds \leftarrow$ minData$(path)$
10:                 $DS \leftarrow DS \bigcup ds$
11:             end for
12:         end for
13:     end for
14: end for
15: $DS \leftarrow sort(DS)$
16: for each $ds \in DS$ do
17:     if $SWf$ can be partitioned by $DS$ without $ds$) then
18:         $DS \leftarrow DS - ds$
19:     end if
20: end for
end
```

algorithm 1: Data transfer minimization SWf partitioning

activities of the SWf at each site. Line 4 selects the fixed activities that are outside of Site s and that the corresponding sites are not processed. If one site is processed in the loop of Lines $5 - 11$, it is marked as processed. For the two functions $fixedActivities$ and $fixedOutsideUnprocessedActivities$, each activity is checked to know if the activity is to be selected. Lines $7 - 10$ choose the dependencies to be removed so that the fixed activities at Site s are not connected with the fixed activities at other sites. Line 7 finds all the paths that connect two activities fixed at different sites. A path has a set of data dependencies that can connect two activities without consideration of direction. In order to find all the paths, a depth-first search algorithm can be used. For each path (Line 8), the data dependency that has the least amount of data to be transferred is selected (Line 9) to DS (Line 10). At the same time, the input activity and output activity of the selected data dependency are marked. Line 11 sorts the data dependencies according to the amount of data to be transferred in descending order. If two or more data dependencies have the same amount of data, they are sorted according to the amount of output data of the following activity of each data dependency in descending order. This order allows the activities that have bigger data dependencies with their following activities to be connected with their following activities, after the removing of dependencies (Lines $12-14$), in order to reduce the amount of data to be transferred among different fragments. Lines $12-14$ remove data dependencies from DS while ensuring that the SWf can always be partitioned with DS. Checking if SWf can be partitioned by a set of dependencies can be realized by checking if the corresponding input and output activities of ds are connected with a depth-first search algorithm (Line 13).

With this algorithm, Buzz SWf is partitioned as shown in Fig. 3.2(c). The dark gray boxes represent the data transfer volume for the corresponding dependencies. The data dependencies of each possible route between *FileSplit* and *HistogramCreator* is analyzed. The dependencies (*YearFilter* to *BuzzHistory* and *YearFilter* to *Top10*) are put in DS to be removed. Finally, Buzz SWf is partitioned by removing the selected dependencies (*YearFilter* to *BuzzHistory* and *YearFilter* to *Top10*).

The third technique is Computing Capacity Adaptation (CCA), which adapts SWfs partitioning to the computing capacity at different cloud sites. This technique is for the heterogeneous multisite cloud configurations, which may be incurred by the different configurations of different groups of scientists. If a SWf is partitioned into two fragments that are sequentially executed, *i.e.* one fragment begins execution after the execution of another one, a SWfMS can put all the possible activities to one fragment while leaving fixed activities in another fragment. As an example, Buzz SWf is partitioned into two fragments (WF_1 and WF_2) as shown in Fig. 3.2(d) . Since the input data of activity *FileSplit* is relatively big and located at a specific site, we keep this activity in the gray fragment. Then, the white fragments can be scheduled to a cloud site that has more computing capacity.

3.6 Validation

In this section, we present experiments to validate our approach, by executing Buzz SWf using our partitioning techniques in Microsoft Azure cloud. The VMs are distributed at two cloud sites: Western Europe (Amsterdam, Netherlands) and Eastern US (North Virginia). In the first experiment, we use a homogeneous configuration by creating two A4 VMs at both of Western Europe site and Eastern US site. In the second experiment, we use a heterogeneous configuration by creating two A4 VMs at the Western Europe site and eight A4 VMs at the Eastern US site. Each A4 VM has 8 CPU cores, 14 GB of RAM memory, 127 GB of instance storage and a network of 800 Mbps [11, 10].

We executed Buzz SWf with Chiron and the SWf partitioner. We used Linux *tar* command and Linux *scp* command for data refining and data transfer. We launched the fragment execution by hand at each cloud site, which resembles to the cooperation between two scientist group. In our execution, Chiron exploits data parallelism and the dynamic scheduling method for SWf parallel execution within one site. Table 3.1 shows the experimental results. Elapsed time 1 represents the execution time without considering data transfer time. Elapsed time 2 shows the execution time plus the data transfer time. Elapsed time 3 is the execution time plus data transfer time with data refining. Data

Table 3.1: **Experimental Results.**

Approach (Time in minutes)	Execution time 1	Transmission time 1	Execution time 2	Transmission time 2	Execution time 3
1^{st} experiment	2*A4 VM (EU) + 2*A4 VM (US)				
SPr technique	199	38	237	0	199
DTM technique	186	0	186	0	186
CCA technique	209	35	245	0.5	209
1^{st} experiment	2*A4 VM (EU) + 8*A4 VM (US)				
SPr technique	198	38	236	0	198
DTM technique	182	0	182	0	182
CCA technique	169	35	201	0.5	169

transfer 1 reveals the data transfer time without data refining. Data transfer 2 presents the data refining and data transfer time. The three techniques correspond to the three partitioning techniques as explained in Section 5.

In the first experiment, the SWf execution time of the three techniques without considering data transfer time is different because there are different amounts of data loaded into RAM memory for the white SWf fragment execution. Since the DTM technique minimizes the data transfer between two fragments, it also reduces the data to be transferred from disk to RAM at the beginning of the second fragment (white fragment) execution. When the data is transferred without data refining, the execution time of the DTM technique is 21.5% and 24.1% less than the SPr and the CCA technique. When the data is transferred with data refining, the DTM technique saves 6.5% and 11.0% of the execution time compared to the SPr and the CCA technique. As the two sites have the same computing capacity and it incurs big data transfer volume, the CCA is the least efficient technique. In this experiment, the SWf execution with the best partitioning technique (DTM with data refining) takes 24.1% less time than the least efficient technique (CCA without data refining). In the second experiment, because of the difference of computing capacity at two cloud sites, the execution of the CCA technique takes the least amount of time without considering data transfer. When the data is transferred without data refining, the DTM technique is still the best performance because of the minimized data transfer cost. This technique yields a gain of 22.9% and 10.4% of the execution time compared to the SPr and CCA technique. However, when we use data refining techniques, the third technique yields the best performance because of the adaptation of SWf partitioning to the computing capacity at each cloud site. In this case, the SWf execution of the CCA technique takes 14.6% and 7.1% less time compared to the SPr and DTM technique. In this experiment, the best partitioning technique (CCA with data refining) saves 28.4% time compared to the least efficient technique (SPr without data refining).

The experiments reveal that the DTM technique with data refining is the best for a homogeneous configuration and that the CCA technique with data refining has better performance for a heterogeneous configuration.

3.7 Conclusion

The cloud is emerging as an appropriate infrastructure for SWf execution. However, because of different restrictions, a SWf often needs to be partitioned and executed in parallel in a multisite environment. In this chapter, we proposed a non-intrusive approach to execute SWfs in a multisite cloud with three SWf partitioning techniques. We proposed a system model based on Chiron SWfMS and its adaptation to multisite cloud and a SWf partitioner. We presented the scheduling of fragment execution by respecting all the data dependencies in the original SWf. We described an experimental validation using an adaptation of Chiron SWfMS for Microsoft Azure multisite cloud. The experiments experiments reveal the efficiency of our partitioning techniques, and their superiority in different environments. The experiment results show that data transfer minimization technique with data refining, *i.e.* file archiving and data compression, has better performance (24.1% of time saved compared to computing capacity adaptation technique without data refining) for homogeneous configurations while computing capacity adaptation technique with data refining (28.4% of time saved compared to scientist privacy technique without data refining) is appropriate to heterogeneous configurations.

Chapter 4

VM Provisioning of Scientific Workflows in a Single Site Cloud

A Cloud provide diverse computing resources and appear as appropriate infrastructures for executing Scientific Workflows (SWfs). However, the problem of how to provision Virtual Machines (VMs) is critical for SWf execution. In addition, since SWf execution takes much time and money, it is important to achieve multi-objectives, *i.e.* reducing both execution time and monetary cost. In this chapter, we address a problem of how to provision VMs to execute a SWf in a multisite cloud, while reducing execution time and monetary costs. The solution consists of a multi-objective cost model including execution time and monetary costs and a Single Site VM Provisioning approach (SSVP). We present an experimental evaluation, based on the execution of the SciEvol SWf in Microsoft Azure cloud. This chapter is based on [117].

Section 4.2 propose our multi-objective cost model. Then, Section 4.3 describes our SSVP algorithm including the SciEvol SWf use case. Section 4.5 presents our experimental evaluation in Microsoft Azure cloud [5]. The results reveal that our cost model is accurate and that SSVP can generate better VM provisioning plans compared with an existing approach.

4.1 Motivations and Overview

SWfs are used to model large-scale *in silico* scientific experiments to process big amounts of data. In order to process big data, the execution of SWfs generally takes much time. As a result, it is important to use parallelism techniques to execute SWfs within a reasonable time. Some SWf Management Systems (SWfMSs) with parallelism techniques already exist, *e.g.* Pegasus [60, 61], Swift [201], Chiron [139], which can take advantage of clusters, grids, and clouds to execute SWfs.

Since a cloud offers diverse resources, virtually infinite computing and storage resources, it becomes an interesting infrastructure for SWf execution. For instance, Infrastructure-as-a-Service, *i.e.* IaaS, providers offer VMs to the general public, including scientists, through the Internet [85]. Diverse types of VMs are available. The VM type determines some parameters such as the number of virtual CPUs, the size of memory and the default storage size of hard disk. The VMs can be used to create a cluster in the cloud.

However, the problem of choosing the number and the type of VMs remains a critical problem for SWf execution in the cloud since the estimation involves various parameters, such as VM types, different SWfs [47]. Although some existing VM provisioning solutions exist, some [66] focus on a single objective and some generate provisioning plans without consideration of the workload of SWfs [166, 47] or just simulates SWf execution to validate the approach [193]. SciDim [54] is proposed to generate a provisioning plan based on a heuristic, *i.e.* genetic algorithm. However, it depends on the provenance data and needs to dynamically modify the provisioning plans, which are not supported by most SWfMSs. Coutinho *et al.* [47] propose a GraspCC algorithm to generate a provisioning plan for SWf execution. However, GraspCC relies on the strong assumption that the entire workload of SWfs can be executed in parallel. Furthermore, it cannot reuse existing started VMs and its cost model is too simple, *e.g.* does not consider the cost of starting VMs, which may be high with many VMs to provision. As a result, the real execution time of two SWfs (SciEvol and SciPhylomics [53]) in Amazon EC2 [12] is two times the estimated time. In addition, some cost models [52][161] for SWf scheduling cannot be used for generating a proper number of virtual CPUs (CPUs designed to VMs) to instantiate for SWf execution at a multisite cloud. In our VM provisioning approach, we use a more precise cost model to calculate a proper number of virtual CPUs, assuming that part of the workload can be executed only sequentially. We also consider the existing started VMs and the cost to start VMs before SWf execution. In addition, we use a real-life SWf to validate our approach.

In this chapter, we propose a Single Site VM Provisioning (SSVP) approach based on a multi-objective cost model. The cost model is used to estimate the cost of the execution of SWfs [52], which includes two objectives, namely reducing execution time and monetary costs. A VM provisioning plan defines how to provision VMs. SSVP generates VM provisioning plans for the execution of SWfs with minimum cost for SWf execution at a single cloud site. We consider a single cloud site, *i.e.* from a single provider and in the same data center. The case of a multisite cloud (with single or multiple cloud providers) is beyond the scope of this chapter. The main contributions of this chapter are:

1. The design of a multi-objective cost model that includes execution time and monetary costs, to estimate the cost of executing SWfs at a single cloud site.

2. A single site VM provisioning approach (SSVP), to generate VM provisioning plans to execute SWfs at a single site.

3. An extensive experimental evaluation, based on the implementation of our approach in Microsoft Azure, and using a real SWf use case (SciEvol [137], a bioinformatics SWf for molecular evolution reconstruction) that shows the advantages of our approach, compared with baseline algorithms.

4.2 Multi-objective Cost Model

This section focuses on multi-objective cost model, used to estimate the cost of executing SWfs at a single cloud site. We propose a multi-objective cost model, which is composed of time cost, *i.e.* execution time, and monetary cost for the execution of SWfs. In order to generate a VM provisioning plan, we need a cost model to estimate the cost of executing a SWf with the VMs to provision. A cost model is composed of a set of formulas to estimate the cost of the execution of SWfs [52]. Our proposed cost model is an extension of the model proposed in [52] and [161].

The cost of executing a SWf can be defined by:

$$Cost(SWf, PL) = \omega_t * Time_n(SWf, PL) + \omega_m * Money_n(SWf, PL) \tag{4.1}$$

, ω_t and ω_m represent the weights for execution time and monetary costs, which are positive. $Time_n(wf, s)$ and $Money_n(wf, s)$ are normalized values that are defined in Sections 4.2.1 and 4.2.2. Since the value of time and money is normalized, the cost has no unit. In the rest of this chapter, cost represents the normalized cost, which has no real unit. PL is the provisioning plan, which defines the number of virtual CPU cores to instantiate.

4.2.1 Time Cost

In this section, we present the method to estimate the time to execute SWf. The normalized time cost used in Formula 4.1 can be defined as:

$$\boxed{Time_n(SWf, PL) = \frac{Time(SWf, PL)}{DesiredTime}} \tag{4.2}$$

, where $Time(SWf, PL)$ represents the entire time for the execution of SWf with the VM provisioning plan PL and $DesiredTime$ is the user defined desired time to execute SWf. Both $DesiredTime$ and $DesiredMoney$ (see Section 4.2.2) are configured by the users. Note that these may be unfeasible to obtain for the execution of the SWf. We take the desired execution time and monetary costs into consideration in the cost model while the real execution time and monetary costs may be bigger or smaller depending on the real execution environment.

In order to execute SWf, the system needs to initialize the corresponding execution environment and to run the program in the VMs. The initialization deploys and initializes VMs for the execution of SWfs. The deployment of a VM is to create a VM under a user account in the cloud. The deployment of the VM defines the type and location, namely the cloud site, of the VM. The initialization of a VM is the process of starting the VM, installing programs and configuring parameters of the VM, so that the VM can be used for executing the tasks of SWfs. This way, the entire time for the execution of SWf can be estimated by the following formula:

$$\boxed{\begin{aligned}Time(SWf, PL) =&InitializationTime(PL)\\ &+ ExecutionTime(SWf, PL)\end{aligned}} \tag{4.3}$$

$InitializationTime$ represents the time to initialize the environment and $ExecutionTime$ is the time to execute the SWf. The time to provision the VMs is estimated by Formula 4.4.

$$\boxed{InitializationTime(PL) = m * InitializationTime} \tag{4.4}$$

$InitializationTime$ represents the average time to provision a VM. The value of $InitializationTime$ can be configured by users according to the cloud environment, which can be obtained by measuring the average time to start, install the required programs and configure 2 - 3 VMs. In the rest of the chapter, we assume that the provisioning plan PL corresponds to m VMs and that there is only one VM being started at a Web domain at the same time, which is true in Azure.

Assuming that the provisioning plan corresponds to n virtual CPU cores to execute a SWf, according to Amdahl's law [170], the execution time can be estimated by Formula 4.5.

$$\boxed{\begin{aligned}&ExecutionTime(SWf, PL)\\ &\quad= \frac{(\frac{\alpha}{n} + (1-\alpha)) * Workload(SWf, InputData)}{ComputingSpeedPerCPUCore}\end{aligned}} \tag{4.5}$$

α[1] represents the percentage of the workload that can be executed in parallel. $ComputingSpeedPerCPUCore$[2] represents the average computing performance of each virtual CPU core, which is measured by FLOPS (FLoating-point Operations Per Second) [47]. $Workload$ represents the workload of SWf with specific amounts of input data $InputData$, which can be measured by the number of FLOP (FLoat-point Operations) [47]. α, the function of $Workload$ and the parameter $ComputingSpeedPerCPUCore$ should be configured by the user according to the features of the cloud and the SWf to be executed.

[1] α can be obtained by measuring the execution time of executing the SWf with a small amount of input data two times with different numbers of virtual CPUs. For instance, assume that we have t_1 for n virtual CPUs and t_2 for m virtual CPUs,

$$\alpha = \frac{m * n * (t_2 - t_1)}{m * n * (t_2 - t_1) + n * t_1 - m * t_2} \tag{4.6}$$

[2] According to [3], we use the following formula to calculate the computing speed of a virtual CPU core. The unit of CPU Frequency is GHz and the unit of Computing speed is GFLOPS.

$$ComputingSpeedPerCPU = 4 * CPUFrequency \tag{4.7}$$

In this chapter, we calculate the workload of a SWf by the following function:

$$Workload(SWf, InputData) = \sum_{act_j \in wf} workload(act_j, inputData) \tag{4.8}$$

The workload of an activity with a specific amount of input data is estimated according to the SWf.

4.2.2 Monetary Cost

In this section, we present the method to estimate the monetary cost to execute SWf with a provisioning plan PL. The normalized monetary cost used in Formula 4.1 can be defined by the following formula:

$$\boxed{Money_n(SWf, PL) = \frac{Money(SWf, PL)}{DesiredMoney}} \tag{4.9}$$

Let us assume that each activity has a user defined workload $Workload(act, inputData)$ similar to that of time cost. Similar to Formula 4.3 for estimating the time cost, the monetary cost also contains two parts, *i.e.* initialization and SWf execution, as defined in Formula 4.10.

$$\boxed{\begin{aligned} Money(SWf, PL) =& InitializationMoney(PL) \\ &+ ExecutionMoney(SWf, PL) \end{aligned}} \tag{4.10}$$

where $InitializationMoney$ represents the monetary cost to provision the VMs for SWf execution and $ExecutionMoney$ is the monetary cost to execute the SWf.

The monetary cost to initialize the execution environment is estimated by Formula 4.11, *i.e.* the sum of the monetary cost of provisioning each VM.

$$\boxed{\begin{aligned} &InitializationMoney(PL) = \\ &\sum_{i=1}^{m} (MonetaryCost(VM_i) * \frac{(m-i) * InitializationTime}{TimeQuantum}) \end{aligned}} \tag{4.11}$$

$MonetaryCost(VM_i)$ is the monetary cost to use a VM VM_i per time quantum at Site s. $InitializationTime$ represents the average time to provision a VM. $TimeQuantum$ is the time quantum in the cloud, which is the smallest possible discrete unit to calculate the cost of using a VM. For instance, if the time quantum is one minute and the price of a VM is 0.5 dollar per hour, the cost to use the VM for the time period of T ($T \geq N - 1$ minutes and $T < N$ minutes) will be $\frac{N*0.5}{60}$ dollars. m (determined by SSVP) represents that there are m VMs to execute SWf. Similar to the time cost estimation, we assume that there is only one VM being started at a Web domain at the same time. In addition, during the provisioning of VMs, the VM that has less virtual CPU cores is provisioned first in order to reduce the monetary cost for waiting for the provisioning of other VMs. Thus, the order of VM_i is also in this order in Formula 4.11, *i.e.* VM_i begins with the VM that has less virtual CPU cores.

The monetary cost for SWf execution can be estimated by Formula 4.12, *i.e.* the monetary cost of using n virtual CPU cores during SWf execution.

$$\boxed{\begin{aligned} &ExecutionMoney(SWf, PL) \\ &\quad = n * MCostPerCPU * \lfloor\frac{ExecutionTime(SWf, PL)}{TimeQuantum}\rfloor \end{aligned}} \tag{4.12}$$

$ExecutionTime(SWf, PL)$ is defined in Formula 4.5. The parameter $MCostPerCPU$ represents the average monetary cost to use one virtual CPU core in one time quantum in the cloud, which can be the price of VMs divided by the number of virtual CPU cores. We assume that the monetary cost of each virtual CPU in the VMs of available different types are the same in the cloud. $TimeQuantum$ represents the time quantum in the cloud.

4.3 Single Site VM Provisioning

We propose a single site VM provisioning algorithm, called SSVP, to generate VM provisioning plans. In order to execute a SWf at a single site cloud, the SWfMS system needs to provision a set of VMs to construct a cluster at a site. The problem of how to provision VMs, *i.e.* to determine the number, type and order of VMs to provision, is critical to the cost of SWf execution.

Based on the aforementioned formulas, we can calculate the execution cost to execute SWf without considering the cost of site initialization according to Formula 4.13. This formula is used to calculate an optimal number, which is used to generate a VM provisioning plan in SSVP, of virtual CPU cores to instantiate for the execution of SWfs.

$$\begin{aligned} ExecutionCost(SWf, PL) = \omega_t * \frac{ExecutionTime(SWf, PL)}{DesiredTime} \\ + \omega_m * \frac{ExecutionMoney(SWf, PL)}{DesiredMoney} \end{aligned} \tag{4.13}$$

In Formula 4.13, $ExecutionTime(SWf, PL)$ is defined in Formula 4.5, $ExecutionMoney(SWf, PL)$ is defined in Formula 4.12 and $DesiredTime$ and $DesiredMoney$ are defined by users. In order to get a general formula to calculate the optimal number of virtual CPUs, we use Formula 4.14, which has no floor function, for $ExecutionMoney(SWf, PL)$.

$$\begin{aligned} &ExecutionMoney(SWf, PL) \\ &\quad = n * MCostPerCPU * \frac{ExecutionTime(SWf, PL)}{TimeQuantum} \end{aligned} \tag{4.14}$$

Finally, the execution cost can be expressed as Formula 4.15 with the parameters defined in Formula 4.16.

$$ExecutionCost(SWf, PL) = a * n + \frac{b}{n} + c \tag{4.15}$$

where

$$\begin{aligned}
a =& \frac{\omega_m * MCostPerCPU * Workload(SWf, InputData) * (1-\alpha)}{ComputingSpeedPerCPUCore * TimeQuantum * DesiredMoney} \\
b =& \frac{\omega_t * \alpha * Workload(SWf, InputData)}{ComputingSpeedPerCPU(s) * DesiredTime} \\
c =& (\frac{\omega_m * \alpha * MCostPerCPU}{DesiredMoney * TimeQuantum} + \frac{\omega_t * (1-\alpha)}{DesiredTime}) \\
& * \frac{Workload(SWf, InputData)}{ComputingSpeedPerCPU(s)}
\end{aligned} \tag{4.16}$$

Based on Formula 4.15, we can calculate a minimal execution cost $Cost_{min}$ and an optimal number of virtual CPUs, *i.e.* n_{opt}, according to Formula 4.17 and Formula 4.18.

$$Cost_{min}(wf, s) = 2 * \sqrt{a * b} + c \tag{4.17}$$

$$\boxed{N_{opt} = \sqrt{\frac{b}{a}}} \tag{4.18}$$

When the system provisions VMs of n_{opt} virtual CPUs, the cost is the minimal[3] based on Formula 4.13, namely $Cost_{min}$, for the execution of SWf.

In order to provision VMs in a cloud, the system can exploit Algorithm 2 to generate a provisioning plan, which minimizes the cost based on the cost model and n_{opt}. In Algorithm 2, Line 2 calculates the optimal number of virtual CPUs to instantiate according to Formulas 4.16 and 4.18. Since the number of virtual CPUs should be a positive integer, we take $\lceil\sqrt{\frac{b}{a}}\rceil$ as the optimal number of virtual CPUs to instantiate. Lines $4-9$ optimize the provisioning plan to reduce the cost to execute SWf. Lines 4 and 6 calculate the cost to execute the SWf based on Formulas 4.1, 4.3 and 4.10. Line 5 improves the provi-

[3] Considering that a, b and n are positive numbers, we can calculate the derivative of function 4.15 as:

$$ExecutionCost'(N, wf, s) = \frac{d}{dn}ExecutionCost(n, wf, s) = a - \frac{b}{n^2} \tag{4.19}$$

When n is smaller than $\sqrt{\frac{b}{a}}$, $ExecutionCost'(n, wf, s)$ is negative and $ExecutionCost(n, wf, s)$ declines when n grows. When n is bigger than $\sqrt{\frac{b}{a}}$, $ExecutionCost'(n, wf, s)$ is positive and $ExecutionCost(n, wf, s)$ increases when n grows. So $ExecutionCost(n, wf, s)$ has a minimum value when $ExecutionCost'(n, wf, s)$ equals zero, *i.e.* $n = \sqrt{\frac{b}{a}}$. And we can calculate the corresponding value of $ExecutionCost'(n, wf, s)$ as shown in Formula 4.17.

```
Input: $SWf$: the SWf to execute; $m$: the number of existing virtual CPUs; $EVM$: existing VMs; $limit$: the maximum number of virtual CPU cores to instantiate at Site $s$;
Output: $PL$: provisioning plan of VMs
begin
 1: $PL \leftarrow \emptyset$
 2: $CPUNumber \leftarrow$ CalculateOptimalNumber($SWf$)
 3: do
 4:     $CurrentCost \leftarrow$ CalculateCost($SWf, m, EVM, PL$)
 5:     $PL' \leftarrow$improve($PL, m, EVM, limit, CPUNumber$)
 6:     $Cost \leftarrow$ CalculateCost($SWf, m, EVM, PL'$)
 7:     if $Cost < CurrentCost$ then
 8:         $PL \leftarrow PL'$
 9:     end if
10: while $Cost < CurrentCost$
end
```

algorithm 2: Single Site VM Provisioning (SSVP)

sioning plan by inserting a new VM, modifying an existing VM or removing an existing VM. If the optimal number of virtual CPUs $CPUNumber$ is bigger than the number $ExistingCPUNumber$ of virtual CPU cores with the consideration of current provisioning plan, and existing virtual CPU cores, a VM is planned to be inserted in the provisioning plan. The VM is of the type that can reduce the difference between $CPUNumber$ and $ExistingCPUNumber$. If $CPUNumber$ is smaller than $ExistingCPUNumber$, the difference between $CPUNumber$ and $ExistingCPUNumber$ is not big and the difference can be reduced by modifying the type of an existing VM, the type of the VM is planned to be modified in the provisioning plan. Otherwise, an existing VM is planned to be removed in the provisioning plan. The VM to be removed is selected among all the existing VMs in order to reduce the most the difference between $CPUNumber$ and $ExistingCPUNumber$. If the cost to execute SWf can be reduced by improving the provisioning plan, the provisioning will be updated (Line 8), and the improvement of provisioning plan continues (Line 9). Note that the direction in the *improve* function of SSVP is determined by comparing $CPUNumber$ and $ExistingCPUNumber$, while the function in GraspCC [47] compares the current provisioning plan with all the possible solutions by changing one VM in the provisioning plan, which has no direction, *i.e.* add, modify or remove, and which is less efficient. While choosing the type of VM to be added, modified or removed, storage constraints[4], specifying that the scheduled site should have enough storage resources for executing the SWf. If the storage constraint is not met, more storage resources are planned to be added to the file system of the VM

[4] All the types (A1, A2, A3 and A4) of VMs mentioned in Section 4.5 can execute the activities of SciEvol in terms of memory.

cluster[5] at the site. Note that the number of virtual CPU cores to instantiate in the provisioning plan, generated by Algorithm 2, may be smaller than n_{opt} because the cost (time and monetary costs) to initialize the site is considered.

4.4 Use Case

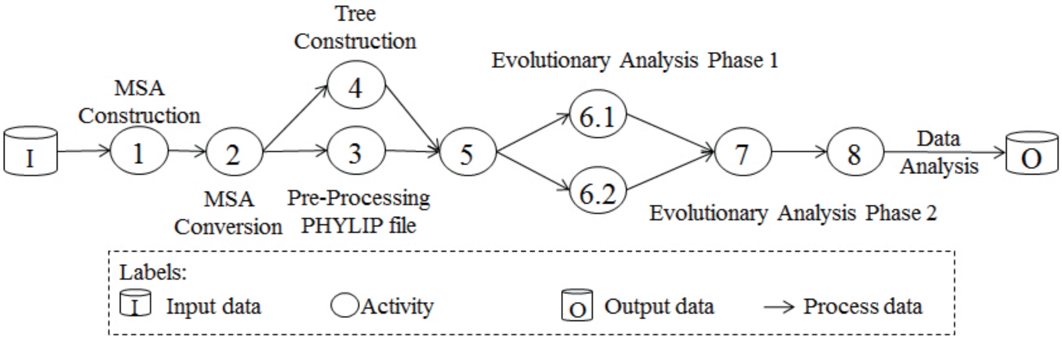


Figure 4.1: **SciEvol Scientific Workflow.**

In this section, in order to validate the SSVP approach, we present a use case, *i.e.* SciEvol with two analysis phase. As presented in Section 2.2.1.4, SciEvol [137] is a SWf for molecular evolution reconstruction that aims at inferring evolutionary relationships, namely to detect positive Darwinian selection, on genome data. It has data and compute intensive activities with data constraints. These characteristics are important to evaluate our scheduling approaches. Figure 4.1 shows the conceptual structure of SciEvol used for this chapter, which is composed of 9 activities.

4.5 Experimental Evaluation

In this section, we present an experimental evaluation of the SSVP algorithm by comparing it with GraspCC. The experiments show the advantages of SSVP over GraspCC in two aspects. The first aspect is that SSVP can estimate cost more accurately based on our proposed cost model than GraspCC. The second aspect is that the provisioning plans generated by SSVP incur less cost than that generated by GraspCC. All experiments are based on the execution of the SciEvol SWf in the Japan East region of Microsoft Azure cloud. During the experiments, the life circle of VM is composed of creation, start, configuration, stop and deletion. The creation, start, stop and deletion of a VM is managed by using Azure CLI. The configuration of VM is realized by Linux *SSH* command. In the experiments, the execution of SWfs is performed by Chiron [139]. The goal is to show

[5] We assume that a VM cluster exploits a shared file system for SWf execution. In a shared file system, all the computing nodes in the cluster share some data storage that is generally remotely located [119].

Table 4.1: **Parameters of different types of VMs.** Type represents the type of VMs. vCPUs represents the number of virtual CPUs in a VM. RAM represents the size of memory in a VM. Disk represents the size of the hard disk in a VM. CC represents the computing capacity of VMs. MC represents Monetary Cost.

Type	vCPUs	RAM	Disk	CC	MC
A1	1	1.75	70	9.6	0.0604
A2	2	3.5	135	19.2	0.1208
A3	4	7	285	38.4	0.2416
A4	8	14	605	76.8	0.4832

that SSVP is suitable to dynamic provisioning of VMs by making a good trade-off among different objectives for the execution of SWfs. Microsoft Azure provides 5 tiers of VM, which are basic tier, standard tier, optimized compute, performance optimized compute and compute intensive. Each tier of VM contains several types of VMs. In one Web domain, users can provision different types of VMs at the same tier. In our experiments, we consider 4 types, namely $A1$, $A2$, $A3$, and $A4$, in the standard tier. The features of the VM types are summarized in Table 4.1. In Azure, the time quantum is one minute. In addition, the average time to provision a VM is estimated as 2.9 minutes. Each VM uses Linux Ubuntu 12.04 (64-bit), and is configured with the necessary software for SciEvol. All VMs are configured to be accessed using Secure Shell (SSH).

Table 4.2: **Workload Estimation.**

Activity	Number of Fasta Files		
	100	500	1000
	Estimated Workload (in GFLOP)		
1	1440	10416	20833
2	384	2778	5556
3	576	4167	8333
4	1440	10416	20833
6.1	5760	41667	83334
6.2	10560	76389	152778
6.3	49920	361111	722222
6.4	59520	430556	861111
6.5	75840	548611	1097222
6.6	202560	1465278	2930556
8	6720	48611	97222

In the experiments, we use 100, 500, 1000 fasta files generated from the data stored in a genome database [7][9]. The programs used are: mafft (version 7.221) for Activity 1, ReadSeq $2.1.26$ for Activity 2, raxmhpc ($7.2.8$ alpha) for Activity 4, pamlX$1.3.1$ for Activities $6.1 - 6.2$, in-house script for Activity 3 and Activity 8, and Activity 5 and Activity

7 exploit a PostgreSQL database management system to process data. The percentage of the workload, *i.e.* α in Formula 4.5, that can be parallelized is 96.43%. In addition, the input data of the SWf is stored at a data server of Site 3, which is accessible to all the sites in the cloud using *SCP* command (a Linux command). The estimated workload (in GFLOP) of each activity of SciEvol SWf for different numbers of input fasta files is shown in Table 4.2.

In the tables and figures, the unit of time is minute, the unit of monetary cost is Euro, the unit of RAM and Disk is Gigabytes, the unit of data is MegaByte (MB), the computing capacity of VMs is GigaFLOPS (GFLOPS) and the unit of workload is GigaFLOP (GFLOP). ω_t represents the weight of time cost. $A1$, $A2$, $A3$ and $A4$ represent the types of VMs in Azure. [Type of VM] * [number] represents provisioning [number] of VMs of [Type of VM] type, *e.g.* A1 * 1 represents provisioning one VM of A1 type. WE represents West Europe; JW Japan West and JE Japan East. The cost corresponds to the price in Euro of Azure on July 27, 2015.

Table 4.3: **VM Provisioning Results.**

Algorithm		SSVP			GraspCC		
ω_t		0.1	0.5	0.9	0.1	0.5	0.9
Provisioning Plan		$A3 * 1$	$A4 * 1$	$A4 * 3$	$A1 * 6$	$A2 * 3$	
Estimated	Execution Time	95	55	34	60		
	Monetary Cost	0.38	0.44	0.75	0.36		
	Cost	1.3094	1.1981	0.7631	1.1882	1.104	1.0208
Real	Execution Time	98	54	35	113	100	
	Monetary Cost	0.40	0.43	0.81	0.64	0.60	
	Cost	1.3472	1.1748	0.7879	2.1181	1.8199	1.6973

We execute the SciEvol SWf with 100 fasta files for different weights of execution time and monetary costs. We assume that the limitation of the number of virtual CPU cores is 32. The estimated workload of this SWf is $192,000$ GFLOP. The desired execution time is set to 60 minutes and the maximum execution time is defined as 120 minutes. The desired monetary cost is configured as 0.3 Euros and the maximum monetary cost is 0.6 Euros. The deployment plans presented in Table 4.3 are respectively generated by SSVP, and GraspCC [47]. Table 4.3 shows the result of the experiments to execute the SWf with different weights of execution time and monetary costs. The execution time, monetary cost and cost is composed of the time or the cost of VM provisioning and SWf execution. For SSVP, the difference between estimated and real execution time ranges from 1.9% to 3.1%, the difference for monetary cost ranges from 2.0% to 6.5% and the difference for the cost is between 2.0% to 3.2%. In fact, the difference between the estimated and real values also depends on the parameters configured by the users. Table 4.3 shows that SSVP can make an acceptable estimation based on different weights of objectives, *i.e.* time and monetary costs.

GraspCC is based on two strong assumptions. The first assumption is that the entire workload of each activity can be executed in parallel, which may not be realistic since

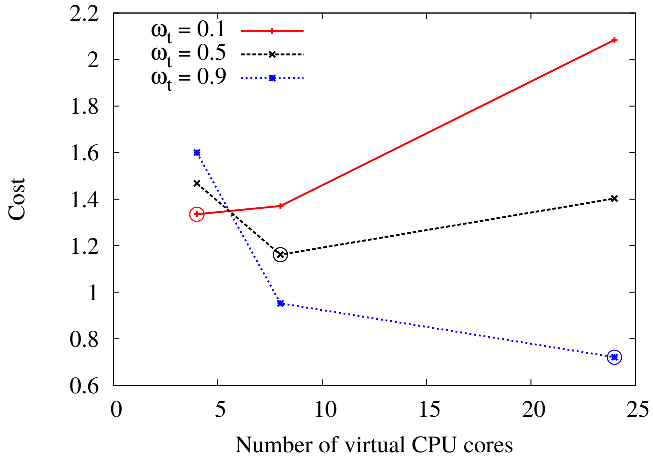


Figure 4.2: **Cost for different values of ω_t with three provisioning plans.** The circled points represent the number of virtual CPU cores corresponding to the provisioning plan generated by SSVP and the corresponding cost of the execution of SciEvol SWf.

some activities cannot be parallelized. The second one is that more VMs can reduce execution time without any bad impact on the cost, *e.g.* higher monetary cost, for the whole execution of a SWf. These two assumptions lead to inaccuracies of the estimation of execution time and monetary costs. In addition, as it is only designed for the time quantum of one hour, GraspCC always generates a provisioning plan that contains the possible largest number of virtual CPUs to reduce the execution time to one time quantum, namely one hour. In Azure, since the time quantum is one minute, it is almost impossible to reduce the execution time to one time quantum, namely one minute. In order to use GraspCC in Azure, we take the time quantum of one hour for GraspCC. GraspCC does not take into consideration the cost (time and monetary costs) to provision VMs, which also brings inaccuracy to the estimated time. Moreover, GraspCC is not sensitive to different values of weight, which are ω_t and ω_m. But SSVP is sensitive to different values of weight because of using the optimal number of virtual CPUs calculated based on the cost model. The final provisioning plan of GraspCC is listed in Table 4.3. GraspCC generates the same provisioning plan for different values of ω_t (0.5 and 0.9). In addition, the difference between the estimated time and the real time is 88.3% ($\omega_t = 0.1$) and 66.7% ($\omega_t = 0.5$ and $\omega_t = 0.9$). However, the difference corresponding to the cost model of SSVP is under 3.1%. Finally, compared with SSVP, the corresponding real cost of the GrapsCC algorithm is 57.2% ($\omega_t = 0.1$), 54.9% ($\omega_t = 0.5$) and 115.4% ($\omega_t = 0.9$) bigger.

Figure 4.2 shows the cost for different provisioning plans and different weights of execution time and monetary costs. According to the provisioning plan generated by SSVP, 4, 8 and 24 virtual CPU cores are instantiated when ω_t is 0.1, 0.5 and 0.9. The corresponding cost is the minimum value in each polyline. The three polylines show that SSVP can generate a good VM provisioning plan, which reduces the cost based on the cost model. The differences between the highest cost and the cost of corresponding good

Table 4.4: **Setup Parameters.** “Number” represents the number of input fasta files. “Limit” represents the maximal number of virtual CPUs that can be instantiated in the cloud. Maximum values are twice the desired values.

Number		500	1000
Desired	Execution Time	60	60
	Monetary Cost	2	6
Maximum	Execution Time	120	120
	Monetary Cost	4	12
Limit		64	128
Estimated Workload		$1,401,600$	$2,803,200$

provisioning plans are: 56.1% ($\omega_t = 0.1$), 26.4% ($\omega_t = 0.5$) and 122.1% ($\omega_t = 0.9$).

Table 4.5: **SSVP VM Provisioning Results.** “Number” represents the number of input fasta files. The provisioning plan represents the plan generated by the corresponding algorithms. "ET" represents execution time and "MC" represents monetary cost.

Number		500			1000		
ω_t		0.1	0.5	0.9	0.1	0.5	0.9
Provisioning Plan		$A2 * 1, A4 * 1$	$A4 * 3$	$A4 * 7$	$A4 * 2$	$A4 * 6$	$A4 * 11$
Estimated	ET	328	194	150	473	290	260
	MC	3.29	4.60	7.93	7.59	13.62	21.70
	Cost	2.0263	2.7640	2.6419	1.9271	3.5462	4.2602
Real	ET	299	177	136	424	294	244
	MC	2.99	4.42	8.34	6.90	14.71	23.21
	Cost	1.8438	2.5800	2.4572	1.7417	3.6758	4.0468

We also execute SciEvol with 500 and 1000 fasta files. The setup parameters are listed in Table 4.4 and the results are shown in Tables 4.5 and 4.6. Since it needs bigger computing capacity to process more input fasta files, we increase the limitation of the number of virtual CPUs, *i.e.* 64 virtual CPUs for 500 fasta files and 128 virtual CPUs for 1000 fasta files. From the tables, we can see that as the estimated workload and desired monetary cost of the SWf grow, more virtual CPUs are planned to be deployed in the cloud. SSVP generates different provisioning plans for each weight of time cost. However, for the same number of input fasta files, GraspCC generates the same provisioning plan for different weights of time cost, namely $A1*1$, $A3*10$ for 500 fasta files and $A2*1$, $A4*10$ for 1000 fasta files. The execution time corresponding to both SSVP and GraspCC, exceeds the maximum execution time. However, SSVP has some important advantages, *e.g.* precise estimation of execution time and smaller corresponding cost.

The difference between estimated time and real time is calculated based on Formula 4.20. As shown in Figure 4.3, the difference between estimated execution time and real execution time corresponding to GraspCC is much higher than that corresponding to the

Table 4.6: **GraspCC VM Provisioning Results.** "Number" represents the number of input fasta files. The provisioning plan represents the plan generated by the corresponding algorithms.

Number		500			1000		
Provisioning Plan		$A1 * 1, A3 * 10$			$A2 * 1, A4 * 10$		
Estimated	Execution Time	60			60		
	Monetary Cost	2.48			4.95		
	Cost	1.2144	1.1191	1.0238	0.8429	0.9127	0.9825
Real	Execution Time	166			257		
	Monetary Cost	6.19			22.43		
	Cost	3.06	2.93	2.80	3.79	4.01	4.23

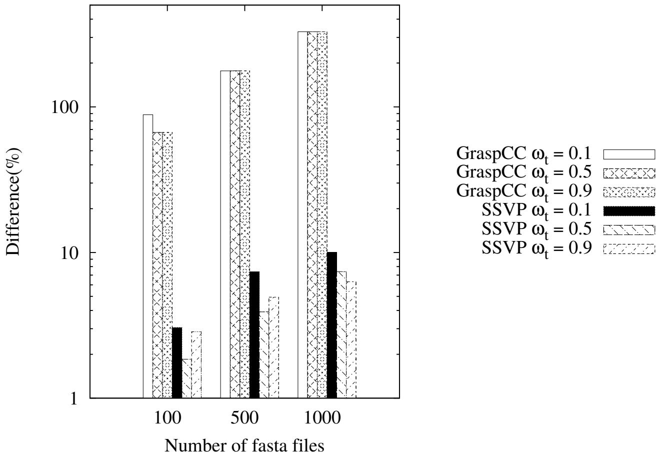


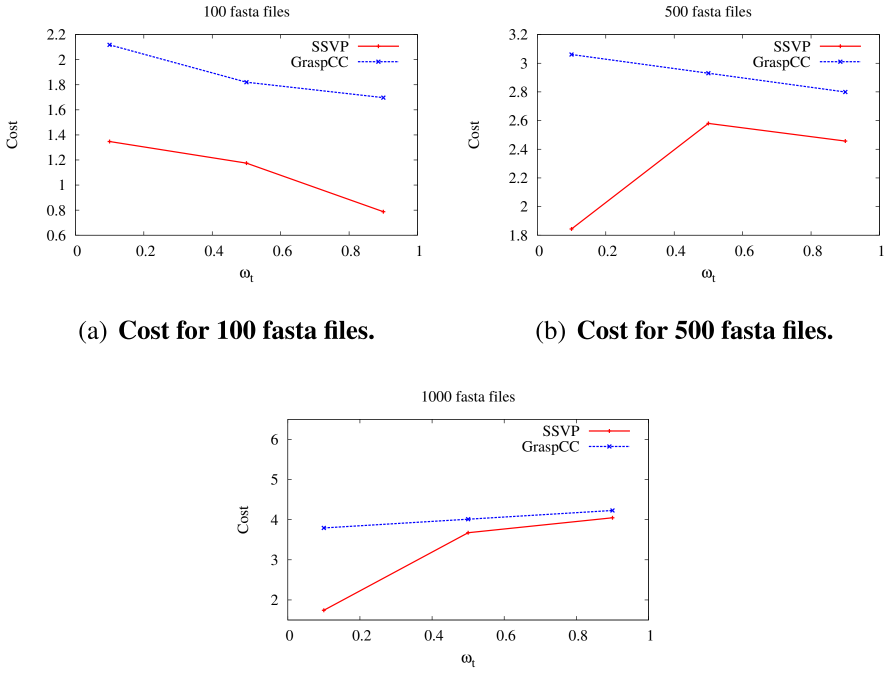
Figure 4.3: **Difference between estimated time and real time.**

cost model of SSVP, which ranges between 66.7% and 328.3%. This result reveals that our cost model can be up to 76.7% more precise than that of GraspCC. As the number of fasta files increases, the difference goes up, *i.e.* it is more difficult to estimate the time. However, the difference corresponding to the cost model of SSVP is always under 11%.

$$Difference = \frac{EstimatedTime - RealTime}{RealTime} * 100\% \tag{4.20}$$

The cost corresponding to different numbers of fasta files is shown in Figure 4.4. It can be seen from Figure 4.4(a), Figure 4.4(b) and Figure 4.4(c) that the cost corresponding to GraspCC is always higher than that corresponding to SSVP with different amounts of input data because SSVP is based on a more accurate cost model and is designed for the quantum of one minute. Based on Formula 4.21, compared with GraspCC, the cost corresponding to SSVP is up to 53.6% smaller. The cost for GraspCC is a line in Figures 4.4(b) and 4.4(c), since GraspCC is not sensitive to the weights of time cost and it generates the same VM provisioning plans, the cost of which is a line. However, since SSVP is sensitive to different values of the weights of execution time, it can reduce the cost at large.

$$Difference = \frac{Cost(GraspCC) - Cost(SSVP)}{Cost(SSVP)} * 100\% \tag{4.21}$$



(a) **Cost for 100 fasta files.**

(b) **Cost for 500 fasta files.**

(c) **Cost for 1000 fasta files.**

Figure 4.4: **Cost for different numbers of fasta files.**

From the experimental results, we can get the conclusion that SSVP can generate better VM provisioning plans than GraspCC because of accurate cost estimation of the cost model.

4.6 Conclusion

In this chapter, we proposed a new VM provisioning approach, namely SSVP, to generate VM provisioning plans for SWf execution with multiple objectives in a single site cloud. The cost model aims at minimizing two costs: execution time and monetary costs. We used a real SWf that is SciEvol, with real data from the bioinformatics domain as a use case. We evaluated our approaches by executing SciEvol in Microsoft Azure cloud. The results show the provisioning approach (SSVP) generates better provisioning plans for different weights of time cost to execute a SWf at a site, compared with other existing approaches, namely GraspCC. The advantage of SSVP can be up to 53.6%. In addition, our cost model can estimate the cost within an acceptable error limit and it is 76.7% more precise than that of GraspCC.

Chapter 5

Multi-Objective Scheduling of Scientific Workflows in a Multisite Cloud

Clouds appear as appropriate infrastructures for executing Scientific Workflows (SWfs). A cloud is typically made of several sites (or data centers), each with its own resources and data. Thus, it becomes important to be able to execute some SWfs at more than one cloud site because of the geographical distribution of data or available resources among different cloud sites. Therefore, a major problem is how to execute a SWf in a multisite cloud, while reducing execution time and monetary costs. This chapter is based on [117] and [118].

Section 5.3 defines the problems for SWf scheduling. In this chapter, we propose a general solution based on multi-objective scheduling in order to execute SWfs in a multisite cloud. The solution consists of a multi-objective cost model including execution time and monetary costs and ActGreedy, a multisite scheduling approach. Section 5.4 describes the system architecture for SWf execution in a multisite cloud. Section 5.5 describes our multi-objective optimization approach and Section 5.6 describes our scheduling approaches including the SciEvol SWf use case, the approaches for SWf partitioning and three scheduling approaches, *i.e.* ActGreedy, LocBased and SGreedy. Then, Section 5.7 is our experimental evaluation based on the execution of the SciEvol SWf in Microsoft Azure cloud. The results reveal that our scheduling approach significantly outperforms two adapted baseline algorithms (which we propose by adapting two existing algorithms) and the scheduling time is reasonable compared with genetic and brute-force algorithms.

5.1 Overview and Motivations

Large-scale *in silico* scientific experiments typically take advantage of SWfs to model data operations such as loading input data, data processing, data analysis, and aggregating output data. SWfs enable scientists to model the data processing of these experiments as a graph, in which vertices represent data processing activities and edges represent dependencies between them. A SWf is the assembly of scientific data processing activities with data dependencies among them [57]. An activity is the description of a piece of work that forms a logical step within a SWf representation [120]. Since SWf activities may process big data, we can exploit data parallelism whereby one activity corresponds to several executable tasks, each working in parallel on a different part of the input data.

In order to execute SWfs efficiently, SWfMSs typically exploit High Performance Computing (HPC) resources in a cluster, grid or cloud environment. Because of virtually infinite resources, diverse scalable services, stable quality of service and flexible payment policies, clouds have become an interesting solution for SWf execution. In particular, the user of Virtual Machines (VMs) makes it easy to deal with elasticity and workloads that change rapidly. A cloud is typically made of several sites (or data centers), each with its own resources and data. Thus, in order to use more resources than available at a single site or to access data at different sites, SWfs could also be executed in a distributed manner at different sites. Nowadays, the computing resources or data of a cloud provider such as Amazon or Microsoft are distributed at different sites and should be used during the execution of SWfs. As a result, a multisite cloud is an appealing solution for large scale SWf execution. As defined in [119], a multisite cloud is a cloud with multiple data centers, each at a different location (possibly in a different region) and being explicitly accessible to cloud users, typically in the data center close to them for performance reasons.

To enable SWf execution in a multisite cloud, the execution of each activity should be scheduled to a corresponding cloud site (or site for short). Then, the scheduling problem is to decide where to execute the activities. In general, to map the execution of activities to distributed computing resources is an NP-hard problem [195]. The objectives can be to reduce execution time or monetary cost, to maximize performance, reliability *etc.* Since SWf execution may take a long time and cost much money, the scheduling problem may have multiple objectives, *i.e.* multi-objective. Thus, the multisite scheduling problem must take into account the impact of resources distributed at different sites, *e.g.* different bandwidths and data distribution at different sites, and different prices for VMs.

In this chapter, we propose a general solution based on multi-objective scheduling in order to execute SWfs in a multisite cloud. The solution includes a multi-objective cost model for multisite SWf execution and ActGreedy, a multisite scheduling approach. The cost model includes two objectives, namely reducing execution time and monetary costs, under stored data constraints, which specify that some data should not be moved, because it is either too big or for proprietary reasons. Although useful for fixing some activities, these constraints do not reduce much the complexity of activity scheduling. We consider a homogeneous cloud environment, *i.e.* from single provider. The case of federated clouds (with multiple cloud providers) is beyond the scope of this chapter and relatively new to cloud users [176]. ActGreedy handles multiple objectives, namely reducing execution time and monetary costs. In order to schedule a SWf in a multisite cloud, the SWf should be partitioned to SWf fragments, which can be executed at a single site. Each fragment can be scheduled by ActGreedy to the site that yields the minimum cost among all available sites. When a fragment is scheduled to a site, the execution of its associated activities is scheduled to the site. ActGreedy is based on our dynamic VM provisioning algorithm, *i.e.* SSVP (see Section 4.3), which generates VM provisioning plans for the execution of fragments with minimum cost at the scheduled site based on a cost model. The cost model is used to estimate the cost of the execution of SWfs [52] according to a scheduling plan, which defines the schedule of fragments to execution sites. A VM provisioning plan defines how to provision VMs. For instance, it determines the types, corresponding number and the order of VMs to provision, for the execution of a fragment. The VM type determines some parameters such as the number of virtual CPUs, the size of memory and the default storage size of hard disk. The main contributions of this chapter are:

1. The design of a multi-objective cost model that includes execution time and monetary costs, to estimate the cost of executing SWfs in a multisite cloud.
2. ActGreedy multisite scheduling algorithm that uses the cost model and SSVP to schedule and execute SWfs in a multisite cloud.
3. An extensive experimental evaluation, based on the implementation of our approach in Microsoft Azure, and using a real SWf use case (SciEvol [137], a bioinformatics SWf for molecular evolution reconstruction) that shows the advantages of our approach, compared with baseline algorithms.

5.2 Related Work

To the best of authors' knowledge, there is no solution to execute SWfs in a multisite cloud environment that takes into account both multiple objectives and dynamic VM provisioning. The related work either focuses on static VM provisioning [65], single objective [22, 125, 154, 168, 177, 196, 186, 67, 125, 122] or single site execution [52, 70, 158]. Static VM provisioning refers to the use of the existing VMs (before execution) for SWf execution without changing the types of VMs during execution. However, existing cost models are not suitable for the SWfs that have a big part of the sequential workload. For instance, the dynamic approach proposed in [47] ignores the sequential part of the SWf and the cost of provisioning VMs, which may generate VM provisioning plans that yield high cost.

Many solutions for SWf scheduling [22, 125, 154, 168, 177, 196] focus on a single objective, *i.e.* reducing execution time. These solutions address the scheduling problem in a single site cloud. Classic heuristics have been used in scheduling algorithms, such as HEFT [186], min-min [67], max-min [67] and Opportunistic Load Balancing (OLB) [125], but they only address the single objective. Furthermore, they are designed for static computing resources in grid or cluster environments. In contrast, our algorithm handles multiple objectives, which are reducing execution time and monetary costs, with dynamic VM provisioning support. Although some general heuristics, *e.g.* genetic algorithms [186], can generate near optimal scheduling plans, it is not always feasible to design algorithms for every possible optimization problem [186] and it is not trivial to configure parameters for the problem. In addition, it may take much time to generate scheduling plans. A brute-force method can generate an optimal scheduling plan, but its complexity is very high.

Some multi-objective scheduling techniques [52, 70, 158] have been proposed. However, they do not take the distribution of resources at different sites into consideration, so they are not suitable for a multisite environment. De Oliveira *et al.* [52] propose a greedy scheduling approach for the execution of SWfs at a single site. However, this approach is not appropriate for multisite execution of SWfs as it schedules the most suitable activities to each VM, which may incur transferring of big data. Rodriguez and Buyya [158] introduce an algorithm for scheduling dynamic bags of tasks and dynamic VM provisioning for the execution of SWfs with multiple objectives in a single site cloud. Rather than using real execution, they simulate the execution of SWfs , thus missing the heterogeneity among the activities of the same SWf, to evaluate their proposed approaches. In real SWf execution, the activities generally correspond to different programs to process data. However, in simulations of SWf execution, the activities are typically made homogeneous, namely, they correspond to the same program. Different from the existing approaches, our approach is suitable for multisite execution and is evaluated by executing a real-life SWf on a multisite cloud (Azure).

Some scheduling techniques have been proposed for the multisite cloud, yet focusing on a single objective, *i.e.* reducing execution time. For instance, Liu *et al.* [122] present a workflow partitioning approach and data location based scheduling approach. But this approach does not take monetary cost into consideration. Our approach uses an *a priori* method, where preference information is given by users and then the best solution is produced. Our approach is based on a multi-objective scheduling algorithm focusing on minimizing a weighted sum of objectives. The advantage of such approach is that it is automatically guided by predetermined weights while the disadvantage is that it is hard to determine the right values for the weights [29]. In contrast, *a posteriori* methods produce a Pareto front of solutions without predetermined values [29]. Each solution is better than the others with respect to at least one objective and users can choose one from the produced solutions. However, this method requires users to pick the most suitable solution. In this chapter, we assume that users have a clear idea of the importance of objectives, and they can determine the value for the weight of each objective. One advantage of using *a priori* method is that we can produce optimal or near optimal solutions without user interference at run-time. When we are using the method of Pareto front, several solutions may be produced to be chosen by the user. Finally, when the weight of each objective is positive, the minimum of the sum is already a Pareto optimal solution [198] [128] and our proposed approach can generate a Pareto optimal or near-optimal solution with the predefined weights. Therefore, we do not consider *a posteriori* methods.

The existing cost models for generating VM provisioning plans [47] are not suitable for SWfs in multisite environments [52, 161] and they do not consider sequential workload in SWfs. Our cost model is based on the cost model presented in Section 4.2, which does consider the cost to provision VMs and the sequential parts of the workload in SWf execution. Furthermore, the cost model also works for multisite SWf execution.

Duan *et al.* [65] propose a multisite multi-objective scheduling approach with consideration of different bandwidths in a multisite environment. However, it is only suitable for static computing resources. In our approach, our scheduling approach is based on a more precise cost model and our SSVP algorithmfor dynamic VM provisioning.

5.3 Problem Definition

This section introduces some important terms, *i.e.* SWf, SWf fragment and multisite cloud and defines the scheduling problem in the multisite cloud.

A *SWf* is described as a Directed Acyclic Graph (DAG) denoted by $W(V,E)$. Let $V = \{v_1, v_2, ..., v_n\}$ be a set of vertices, which are associated with the scientific data processing activities and $E = \{e_{i,j}$: $v_i, v_j \in V$ and v_j consumes the output data of v_i $\}$ be a set of edges that correspond to dependencies between activities in V. Activity v_j is the following activity of Activity v_i and Activity v_i is a preceding activity of Activity v_j. The dependencies can be data or control dependencies. Compared to data dependencies, fewer data are transferred in control dependencies. The transferred data in a control dependency is the configuration parameters for activity execution while the transferred data in a data dependency is the input data to be processed by the following activity. The activity that processes control parameters is a control activity. Since the control activity takes little time to execute, we assume that a control activity has no workload. In addition, we assume that the data stored at a specific site may not be allowed to be transferred to other sites because of proprietary or big amounts of data, which is denoted as *stored data constraint*. If an activity needs to read the data from the stored data located at a specific site, this activity is denoted as *fixed activity*.

A large-scale SWf and its input data can be partitioned into several fragments [39] [122]. Thus, a SWf can be described as the assembly of fragments and fragment dependencies, *i.e.* $W(WF, FE)$ where $WF = \{wf_1, wf_2, ..., wf_n\}$ represents a set of fragments connected by dependencies in the set $FE = \{fe_{i,j}$: $wf_i, wf_j \in WF$ and wf_j consumes the output data of $wf_i\}$. A fragment dependency $fe_{i,j}$ represents that fragment wf_j processes the output data of fragment wf_i. $fe_{i,j}$ is the input dependency of wf_j and output dependency of wf_i. A fragment can be denoted by $wf(V, E, D)$. V represents the activities, E represents the dependencies and D represents the input data of the SWf fragment.

We denote the SWf execution environment by a configured multisite cloud[1] $MS(S)$, which consists of a set of sites S. A multisite cloud configuration defines the instances of VMs and storage resources for cloud users in a multisite cloud. One site $s_i \in S$ is composed of a set of Web domains. A Web domain contains a set of VMs, shared storage resources, and stored data. In this chapter, we assume that one site contains only one Web domain for the execution of a SWf. We assume that the available VMs for the execution of SWfs in a multisite cloud have the same virtual CPUs, *i.e.* the virtual CPUs have the same computing capacity, but the number of virtual CPUs in each VM may be different.

[1] The multisite cloud environment configured for the quota of resources that can be used by a cloud user.

In addition, we assume that the price to instantiate VMs of the same type are the same at the same site while the prices at different sites can be different. The price is the monetary cost to use a VM during a time quantum (the quantum varies according to the cloud provider, *e.g.* one hour or one minute). Time quantum is the smallest possible discrete unit to calculate the cost of using a VM. For instance, if the time quantum is one minute and the price of a VM is 0.5 dollar per hour, the cost to use the VM for the time period of T ($T \geq N - 1$ minutes and $T < N$ minutes) will be $\frac{N*0.5}{60}$ dollars.

Scheduling fragments requires choosing a site to execute a fragment, *i.e.* mapping each fragment to an execution site. A fragment scheduling plan defines the map of fragments and sites. When a fragment is scheduled at a site, the activities of the fragment are also scheduled at that site. Based on a multi-objective cost model, the problem we address has the following form [146]:

min($Cost(Sch(SWf, S))$)
subject to
stored data constraint

The decision variable is $Schedule(wf, s)$, which is defined as

$$Schedule(wf, s) = \begin{cases} 1 \text{ if Fragment } wf \text{ is scheduled at Site } s \\ 0 \text{ otherwise} \end{cases}$$

Thus, the scheduling problem is, given a multi-objective cost model, how to generate a fragment scheduling plan $Sch(SWf, S)$, for which the corresponding SWf execution has minimum $Cost(Sch(SWf, S))$ while respecting the stored data constraints $Const(data)$ with $data \in input(SWf)$. The cost is the value calculated based on formulas defined in a cost model, *e.g.* Formulas 5.1 and 5.2, which depends on a scheduling plan and VM provisioning plans at scheduled sites. In the scheduling plan, for each Fragment wf and site s, only if Fragment wf is scheduled at Site s, the decision variable is 1; otherwise, the variable is 0. One fragment can be scheduled at only one site. The search space of scheduling plans contains all the possible scheduling plans, *i.e.* for any combination of wf and s, we can find a scheduling plan in the search space that contains the decision variable $Schedule(wf, s) = 1$. If the cost is composed of just one objective, the problem is a single objective optimization problem. Otherwise, the problem is a multi-objective problem. The cost model is detailed in Section 5.5.1. In this chapter, we use SSVP (see Section 4.3) to generate VM provisioning plans. The stored data constraints can be represented as a matrix (as the one presented below), and its cell values are known before execution, where each row a_i represents an activity, each column s_j represents a cloud site, and $(a_i, s_j) = 1$ means that a_i needs to read the data stored at s_j.

$$\begin{array}{cccc} & s_1 & s_2 & s_3 \\ a_1 & 1 & 0 & 0 \\ a_2 & 0 & 0 & 1 \\ a_3 & 0 & 1 & 0 \end{array}$$

5.4 Multisite SWfMS Architecture

In this section, we present the architecture of a multisite SWfMS. This architecture (see Figure 5.1) has four modules: workflow partitioner, multisite scheduler, single site initialization, and single site execution. The workflow partitioner partitions a SWf into fragments (see Section 3.5). After SWf partitioning, the fragments are scheduled to sites by the multisite scheduler. After scheduling, in order to avoid restarting VMs for the execution of continuous activities, all the activities scheduled at the same site are grouped as a fragment to be executed. Then, the single site initialization module prepares the execution environment for the fragment, using two components, *i.e.* VM provisioning and multisite data transfer. At each site, the VM provisioning component deploys and initializes VMs for the execution of SWfs. The deployment of a VM is to create a VM under a user account in the cloud. The deployment of the VM defines the type and location, namely the cloud site, of the VM. The initialization of a VM is the process of starting the VM, installing programs and configuring parameters of the VM, so that the VM can be used for executing the tasks of fragments. The multisite data transfer module transfers the input data of fragments to the site. Finally, the single site execution module starts the execution of the fragments at each site. This can be realized by an existing single site SWfMS, *e.g.* Chiron [139]. Within a single site, when the execution of its fragment is finished and the output data is moved to other sites, the VMs are shut down. When the execution of the fragment is waiting for the output data produced by other sites and the output data at this site are transferred to other corresponding sites, the VMs are also shut down to avoid the useless monetary cost. When the necessary data is ready, the VMs are restarted to continue the execution of the fragment.

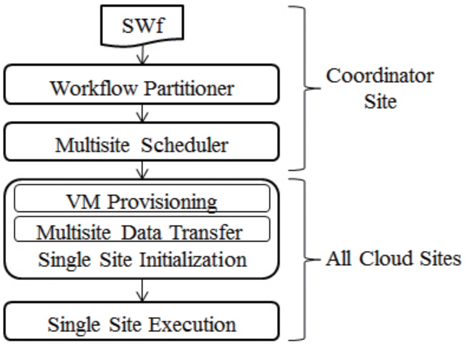


Figure 5.1: **System Architecture.**

In a multisite cloud, there are two types of sites, *i.e.* coordinator and participant. The coordinator is responsible for coordinating the execution of fragments at different participants. Two modules, namely workflow partitioner and multisite scheduler, are implemented at the coordinator site. Both the coordinator and participants execute the scheduled fragments. The initialization module and single site execution module are implemented at both the coordinator and participants.

5.5 Multi-objective Optimization

This section focuses on multi-objective optimization, which is composed of a multi-objective cost model, used to estimate the cost of executing SWfs in a multisite cloud and a cost estimation method for the scheduling process.

5.5.1 Multi-objective Cost Model

We propose a multi-objective cost model, which is composed of time cost, *i.e.* execution time, and monetary cost for the execution of SWfs. In order to choose a good scheduling plan, we need a cost model to estimate the cost of executing a SWf in a multisite cloud. A cost model is composed of a set of formulas to estimate the cost of the execution of SWfs [52] according to a scheduling plan. It is generally implemented in the scheduling module and under a specific execution environment. In the case of this chapter, the execution environment is a multisite cloud. Our proposed cost model is an extension of the model proposed in [52] and [161]. In addition, the cost model is also used to calculate the real cost by replacing estimated parameters by real values obtained from the real execution in the evaluation part, *i.e.* Section 5.7.

The cost of executing a SWf can be defined by:

$$\begin{aligned} Cost(Sch(SWf,S)) = & \omega_t * \frac{Time(Sch(SWf,S))}{DesiredTime} \\ & + \omega_m * \frac{Money(Sch(SWf,S))}{DesiredMoney} \end{aligned} \tag{5.1}$$

, where $DesiredTime$ represents the desired execution time to execute the SWf and $DesiredMoney$ is the desired monetary cost for the execution. Both $DesiredTime$ and $DesiredMoney$ are configured by the users. Note that these may be unfeasible to obtain for the execution of the SWf. We take the desired execution time and monetary costs into consideration in the cost model while the real execution time and monetary costs may be bigger or smaller depending on the real execution environment. $Time(SWf)$ and $Money(SWf)$ is the real execution time and real monetary cost for the execution of the SWf. ω_t and ω_m represent the weights for execution time and monetary costs, which are positive.

However, it is difficult to estimate the execution time and monetary costs for the whole SWf even with a scheduling plan according to Formula 5.1 since it is hard to generate a VM provisioning plan for each site with global desired execution time and monetary costs. As shown in Formula 5.2, we decompose the cost model as the sum of the cost of executing each fragment.

$$Cost(Sch(SWf,S)) = \sum_{wf_i \in SWf}^{Schedule(wf_i,s_j)=1} Cost(wf_i, s_j) \tag{5.2}$$

The cost of executing a fragment at a site can be defined as:

$$\boxed{Cost(wf, s) = \omega_t * Time_n(wf, s) + \omega_m * Money_n(wf, s)} \tag{5.3}$$

The box represents that the formula is referred in the following sections and the meaning of boxes of other formulas is the same. ω_t and ω_m, which are the same as that in Formula 5.1, represent the weights for the execution time and the monetary cost to execute Fragment wf at Site s. $Time_n(wf, s)$ and $Money_n(wf, s)$ are normalized values that are defined in Sections 5.5.1.1 and 5.5.1.2. Since the value of time and money is normalized, the cost has no unit. In the rest of this chapter, cost represents the normalized cost, which has no real unit. And we use SSVP (see Section 4.3) to generate VM provisioning plans at each site for the execution of SWf fragments.

5.5.1.1 Time Cost

In this section, we present the method to estimate the time to execute Fragment wf at Site s with scheduling plan SP. The normalized time cost used in Formula 5.3 can be defined as:

$$\boxed{Time_n(wf, s) = \frac{Time(wf, s)}{DesiredTime(wf)}} \tag{5.4}$$

, where $Time(wf, s)$ represents the entire time for the execution of Fragment wf at Site s and $DesiredTime(wf)$ is the desired time to execute Fragment wf. Assuming that each activity has a user estimated workload $Workload(a, inputData)$ with a specific amount of input data $inputData$, we can calculate the desired execution time of Fragment wf with the user defined desired time for the whole SWf by Formula 5.5.

$$\boxed{\begin{aligned} DesiredTime(wf) =& \frac{\sum_{a_i \in CP(wf)} workload(a_i, inputData)}{\sum_{a_j \in CP(SWf)} workload(a_j, inputData)} \\ & * DesiredTime \end{aligned}} \tag{5.5}$$

In this formula, $CP(SWf)$ represents the critical path of Workflow SWf, which can be generated by the method proposed by Chang *et al.* [36]. A critical path is a path composed of a set of activities with the longest average execution time from the start activity to the end activity [36]. In a SWf, the start activity is the activity that has no input dependency and the end activity is the activity that has no output dependency. Similarly, $CP(wf)$ represents the critical path of Fragment wf. The workload $workload(a_i, input\text{-}Data)$ of an activity a_i with a specific amount of data $inputData$ is estimated by users according to the features of the SWf. $DesiredTime$ is the desired execution time for the whole SWf, defined by user. Since the time to execute a fragment or a SWf is similar to that of the executing the activities in the critical path, we calculate the desired time for a fragment as the part of the time to execute the same workload of activities in the critical path of the SWf as that of the fragment.

In order to execute Fragment wf at Site s, the system needs to initialize the corresponding execution site, to transfer the corresponding input data of Fragment wf to Site s and to run the program in the VMs of the site. The initialization of the site is explained in Section 5.4. This way, the entire time for the execution of Fragment wf at Site s can be estimated by the following formula:

$$\boxed{\begin{aligned} Time(wf,s) =& InitializationTime(wf,s) \\ &+ TransferTime(wf,s) \\ &+ ExecutionTime(wf,s) \end{aligned}} \tag{5.6}$$

, where s represents the site to execute Fragment wf according to the scheduling plan SP, $InitializationTime$ represents the time to initialize Site s, $TransferTime$ is the time to transfer input data from other sites to Site s and $ExecutionTime$ is the time to execute the fragment. In order to simplify the problem, we ignore the cost (both time cost and monetary cost) to restart VMs at a site to wait for the output data produced by other sites. In this formula, the time to wait for the input data produced by the activities executed at another site (Site s_o) is not considered since this time is considered in that of the fragment executed at Site s_o.

The multisite SWfMS needs to provision m (determined by SSVP) VMs to execute Fragment wf at a single site. As explained in Section 5.4, to provision a VM is to deploy and to initialize a VM at a cloud site. The time to provision the VMs at a single site is estimated by Formula 4.4.

The time for data transfer is the sum of the time to transfer input data stored in other sites to Site s. The data transfer time can be estimated by formula 5.7.

$$\boxed{TransferTime(wf,s) = \sum_{s_i \neq s} \frac{DataTransferAmount(wf,s_i)}{DataTransferRate(s_i,s)}} \tag{5.7}$$

$DataTransferAmount(wf,s_i)$ is the amount of input data of Fragment wf stored at Site s_i, which is defined later (see Formula 5.8). $DataTransferRate(s_i,s)$ represents the data transfer rate between Site s_i and Site s, which can be roughly obtained by measuring the amount of data transferred by Linux SCP command during a specific period of time between two VMs located at the two sites.

We assume that the amount of input data for each dependency is estimated by the user. The amount of data to be transferred from another site (s_i) to the site (s) to execute the fragment can be estimated by Formula 5.8.

$$\begin{aligned} &DataTransferAmount(wf,s_i) \\ &= \sum_{a_j \in wf} \sum_{a_k \in preceding(a_j)}^{a_k \in activities(s_i)} AmountOfData(e_{k,j}) \end{aligned} \tag{5.8}$$

where $preceding(a_j)$ represents the preceding activities of Activity a_j at Site s_o. s_i represents the site that originally stores a part of the input data of Fragment wf. $activities(s_i)$ represents the activities in the fragments that are scheduled at Site s_i. As defined in Section 5.3, $e_{k,j}$ represents the data dependency between Activity a_k and a_j.

Assuming that one site has n (determined by SSVP) virtual CPUs to execute a fragment, according to Amdahl's law [170], the execution time can be estimated by Formula 4.5. The parameters n and m can be determined by a dynamic VM provisioning algorithm, which is detailed in Section 4.3.

5.5.1.2 Monetary Cost

In this section, we present the method to estimate the monetary cost to execute Fragment wf at Site s with a scheduling plan SP. The normalized monetary cost used in Formula 5.3 can be defined by the following formula:

$$\boxed{Money_n(wf, s) = \frac{Money(wf, s)}{DesiredMoney(wf)}} \tag{5.9}$$

Let us assume that each activity has a user defined workload $Workload(a, inputData)$ similar to that of time cost. Inspired by Fard *et al.* [70], we calculate the desired monetary cost of executing a fragment wf by Formula 5.10, which is the part of the monetary cost to execute the workload of Fragment wf in the SWf. In Formula 5.10, a_i and a_j represent an activity.

$$\boxed{\begin{aligned} DesiredMoney(wf) =& \frac{\sum_{a_i \in wf} workload(a_i, inputData)}{\sum_{a_j \in SWf} workload(a_j, inputData)} \\ & * DesiredMoney \end{aligned}} \tag{5.10}$$

Similar to Formula 5.6 for estimating the time cost, the monetary cost also contains three parts, *i.e.* site initialization, data transfer and fragment execution, as defined in Formula 5.11.

$$\boxed{\begin{aligned} Money(wf, s) =& InitializationMoney(wf, s) \\ & + TransferMoney(wf, s) \\ & + ExecutionMoney(wf, s) \end{aligned}} \tag{5.11}$$

where s represents the site to execute Fragment wf. $InitializationMoney$ represents the monetary cost to provision the VMs at Site s, $TransferMoney$ is the monetary cost to transfer input data of Fragment wf from other sites to Site s and $ExecutionMoney$ is the monetary cost to execute the fragment. The monetary cost to initialize a single site is estimated by Formula 4.11, *i.e.* the sum of the monetary cost for provisioning each VM.

The monetary cost for data transfer should be estimated based on the amount of transferred data and the price to transfer data among different sites, which is defined by the cloud provider. In this chapter, the monetary cost of data transfer is estimated according to Formula 5.12, where $DataTransferUnitCost$ represents the monetary cost to trans-

Table 5.1: **Parameter summary.** Original represents where the value of the parameter comes from. UD: that the parameter value is defined by users; ESWf: that the parameter value is estimated according to the SWf; Measure: that the parameter value is measured by user with the SWf and in a cloud environment; Cloud: the parameter value is obtained from the cloud provider; Execution: measured during the execution of SWf in a multisite cloud.

Parameter	Meaning	Original
DesiredTime	Desired execution time	UD
DesiredMoney	Desired monetary cost	UD
workload	The workload of an activity	ESWf
AmountOfData	The amount of data in a data dependency	ESWf
InitializationTime	The time to initialize a VM	Measure
DataTransferRate	Data transfer rate between two sites	Measure
α	The percentage of the workload that can be executed in parallel	Measure
CPUFrequency	Computing performance of virtual CPUs	Cloud
MonetaryCost	Monetary cost of a VM	Cloud
TimeQuantum	The time quantum of a cloud	Cloud
DataTransferUnitCost	The monetary cost to transfer a unit of data between two sites	Cloud
MCostPerCPU	The monetary cost to use a virtual CPU at a site	Cloud
ExecutionTime	The execution time of a fragment at a site	Execution

fer a unit, *e.g.* gigabyte(GB), of data from the original site (s_o) to the destination site (s). $DataTransferUnitCost$ is provided by the cloud provider.

$$\boxed{\begin{aligned} TransferMoney(wf, s) = \sum_{s_i \neq s} (&DataTransferAmount(wf, s_i) \\ &* DataTransferUnitCost(s_i, s)) \end{aligned}} \tag{5.12}$$

$DataTransferAmount(wf, s_i, s)$ is defined in Formula 5.8. The monetary cost for the fragment execution can be estimated by Formula 4.12, *i.e.* the monetary cost of using n virtual CPUs during the Fragment execution.

The original parameters mentioned in this section are listed in Table 5.1. The other parameters that are not listed in Table 5.1 are derived based on the listed original parameters.

5.5.2 Cost Estimation

The cost estimation method is used to estimate the cost to execute a fragment at a site based on the cost model. First, SSVP (see Section 4.3) is used to generate a provisioning plan. Then, the number of virtual CPUs, *i.e.* n, and number of VMs to deploy, namely m, is known to estimate the time and monetary costs to initiate a site based on Formulas 4.4, 4.11. In addition, the time and monetary costs to execute the fragment are recalculated using Formulas 4.5 and 4.12. During scheduling, only available fragments are scheduled. An available fragment indicated that its preceding activities are already scheduled, which means that the location of the input data of the fragment is known. Thus, Formulas 5.7 and 5.12 are used to estimate the time and monetary costs to transfer the input data of Fragment wf to Site s. Afterwards, the total time and monetary costs can be estimated by Formulas 5.6 and 5.11. Finally, the cost of executing Fragment wf at Site s is estimated based on Formulas 5.3, 5.4, 5.9, 5.5, 5.10.

5.6 Fragment Scheduling

In this section, we present our approach for fragment scheduling, which is the process of scheduling fragments to sites for execution. First, we present a use case, *i.e.* the SciEvol SWf, which we will use to illustrate fragment scheduling. Then, we present an adaptation of two state-of-the-art algorithms (LocBased and SGreedy) and our proposed algorithm (ActGreedy).

5.6.1 Use Case

In this section, in order to illustrate partitioning and scheduling approaches, we use the SciEvol SWf use case presented in Chapter 2 with stored data at different cloud sites. In Figure 5.2, “read data” represents that one activity just reads the stored data without

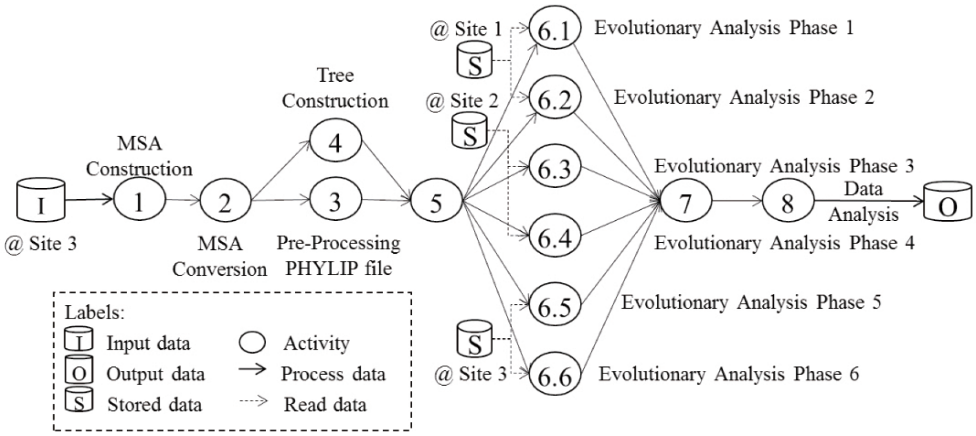

@ Site 1
6.1
Evolutionary Analysis Phase 1
S
Tree Construction
@ Site 2
6.2
Evolutionary Analysis Phase 2
S
MSA Construction
4
6.3
Evolutionary Analysis Phase 3
I
1
2
3
5
7
8
Data Analysis
O
@ Site 3
MSA Conversion
Pre-Processing PHYLIP file
6.4
Evolutionary Analysis Phase 4
Labels:
I Input data
Activity
O Output data
Process data
S Stored data
Read data
6.5
Evolutionary Analysis Phase 5
S
@ Site 3
6.6
Evolutionary Analysis Phase 6

Figure 5.2: **SciEvol SWf.**

modifying it and that the activity should be executed at the corresponding site to read data since the stored data is too big to be moved and can be accessed only within the corresponding site because of configurations, *e.g.* security configuration of a database. The stored data constraints are defined by the following matrix (the activities not listed in the matrix are not fixed).

$$\begin{array}{cccc}
 & s_1 & s_2 & s_3 \\
a_{6.1} & 1 & 0 & 0 \\
a_{6.2} & 1 & 0 & 0 \\
a_{6.3} & 0 & 1 & 0 \\
a_{6.4} & 0 & 1 & 0 \\
a_{6.5} & 0 & 0 & 1 \\
a_{6.6} & 0 & 0 & 1
\end{array}$$

5.6.2 Scheduling approaches

In this section, we propose three multisite scheduling algorithms. The first one, *LocBased* is adapted from the scheduling algorithm used in Chapter 3, which schedules a fragment to the site that stores the data while reducing data transfer among different sites. The second one, *SGreedy*, is adapted from the greedy scheduling algorithm designed for multi-objective single site scheduling in our previous work [52], which schedules the most suitable fragment to each site. The last one, *ActGreedy*, which combines characteristics of the two adapted algorithms, schedules the most suitable site to each fragment. In addition, we propose that a fixed activity can only be scheduled and executed at the site where the stored data is located. This is applied by analyzing the constraint matrix in all the three algorithms before other steps, which are not explicitly presented in the algorithms.

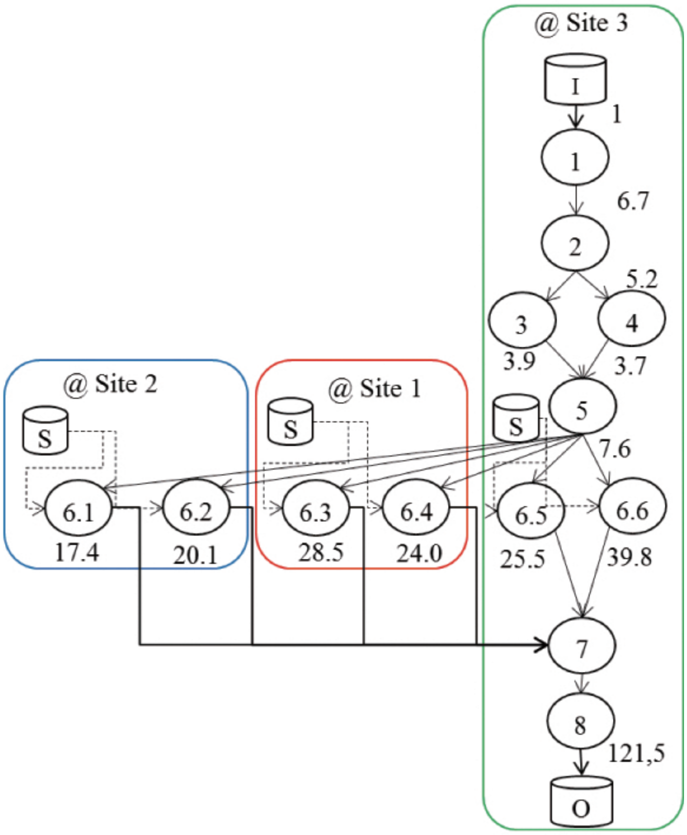


Figure 5.3: **SWf partitioning and data location based scheduling.** The number represents the relative (compared with the input data) amount of output data for corresponding activities.

```
Input: $swf$: a scientific workflow; $S$: a set of sites
Output: $SP$: scheduling plan for $swf$ in $S$
 1: $SP \leftarrow \emptyset$
 2: $WF \leftarrow partition(WF)$                    ▷ According to Algorithm 1
 3: for each $wf_i \in WF$ do
 4:     $s_j \leftarrow GetDataSite(wf_i, S)$        ▷ get Site $s_j$ that stores its required data or the
    biggest amount of input data
 5:     $SP \leftarrow SP \bigcup \{Schedule(wf_i, s_j)\}$
 6: end for
end
```

algorithm 3: Data location based scheduling

5.6.2.1 Data Location Based Scheduling

We adapt the scheduling approach proposed in [122] to the multisite cloud environment. Since this approach is based on the location of data, we call it LocBased (data location based) scheduling, which is given in Algorithm 3. Line 2 partitions a Fragment wf using the data transfer minimization algorithm (Algorithm 1 in Section 3.5) as shown in Figure 5.3. Then, each fragment wf (Line 3) is scheduled to a data site (Line $4-5$). If the fragment contains a fixed activity, the scheduled data site is the site that stores the required data (*i.e.* stored data) of the fixed activity. If the fragment does not contain a fixed activity, the scheduled data site is the site that stores the biggest part of the input data of the fragment.

Algorithm 3 schedules the fragment to the data site that stores the required data (*i.e.* stored data for fixed activity) or the biggest part of the input data (for normal activities) in order to reduce the time and monetary costs to transfer data among different sites. However, the granularity of this scheduling algorithm is relatively big and some activities are scheduled at a site that incurs high cost. For instance, the result of this algorithm is shown in Figure 5.3 while Activity $1, 2, 3, 4, 5, 7$ and 8 can be scheduled at Site 1, which is less expensive to use VMs than at other sites.

5.6.2.2 Site Greedy Scheduling

We adapt a Site Greedy (SGreedy) scheduling algorithm proposed by de Oliveira *et al.* [52], for multiple objectives in a multisite environment. Algorithm 4 describes SGreedy. When there is a fragment that is not scheduled (Line 3), for each site (Line 4), the fragment that takes the least cost is scheduled to each site (Line $5-10$). The fragments are selected from the available fragments (Line 5 and 12). Line $7-8$ estimate the total cost to execute Fragment wf at the site. Line 9 chooses the optimal fragment, *i.e.* wf_{opt}, that needs the smallest total cost to be executed for the site. Line 10 schedules the optimal fragment to the site. Line 11 updates the fragments that need to be scheduled. Line 12 prepares available fragments to be scheduled for the next site.

This algorithm intends to keep all the sites busy and chooses the best fragment for each

```
Input: $swf$: a scientific workflow; $S$: a set of sites
Output: $SP$: scheduling plan for $swf$ in $S$
$SP \leftarrow \emptyset$
$WF \leftarrow partition(WF)$   ▷ According to activity encapsulation partitioning method
while $WF \neq \emptyset$ do
    for each $s \in S$ do
        $WFA \leftarrow GetAvailableFragments(WF)$
        if $WFA \neq \emptyset$ then
            for each $wf \in WFA$ do
                $Cost[i] \leftarrow EstimateCost(wf, s)$   ▷ According to the cost estimation method
            end for
            $wf_{opt} \leftarrow GetFragment(s, Cost)$   ▷ Get the fragment that takes the minimal cost for the execution at Site $s$
            $SP \leftarrow SP \bigcup \{Schedule(wf_{opt}, s)\}$
            $WF \leftarrow WF - wf_{opt}$
            $WFA \leftarrow GetAvailableFragments(WF)$
        end if
    end for
end while
end
```

algorithm 4: Site greedy scheduling

site. However, as shown in Figure 5.4, this algorithm may break the dataflow between the parent activity and child activity, which may incur high cost to transfer the data among different sites, *e.g.* Activities $1, 2, 3, 4, 7, 8$. In addition, in order to avoid useless usage of VMs at Site 1 and Site 2 during the execution of the SWf, the VMs are shut down after finishing the execution of the corresponding activities, and restarted for the following activities when the input data is ready. After executing Activities 1, 6.3 and 6.4, the VMs at Site 1 are shut down. The VMs at Site 2 are shut down after executing Activity 2. Since Activity 5 is a control activity, which takes little time to be executed, the VMs at Site 1 and 3 are not shut down after executing Activities 4 and 3. When the execution of Activities 6.5 and 6.6 are to be finished, the VMs at Site 2 are restarted to continue the execution (since the execution of Activities 6.5 and 6.6 may take more time because of big workload). This process may also incur high cost when there are many VMs to restart.

5.6.2.3 Activity Greedy Scheduling

Based on LocBased and SGreedy, we propose the ActGreedy (Activity Greedy) scheduling algorithm, which is described in Algorithm 5. In this algorithm, all the fragments of a SWf are not scheduled, *i.e.* $Sch(SWf, S) = \emptyset$, at beginning. During the scheduling process, all the fragments are scheduled at a corresponding site, which takes the

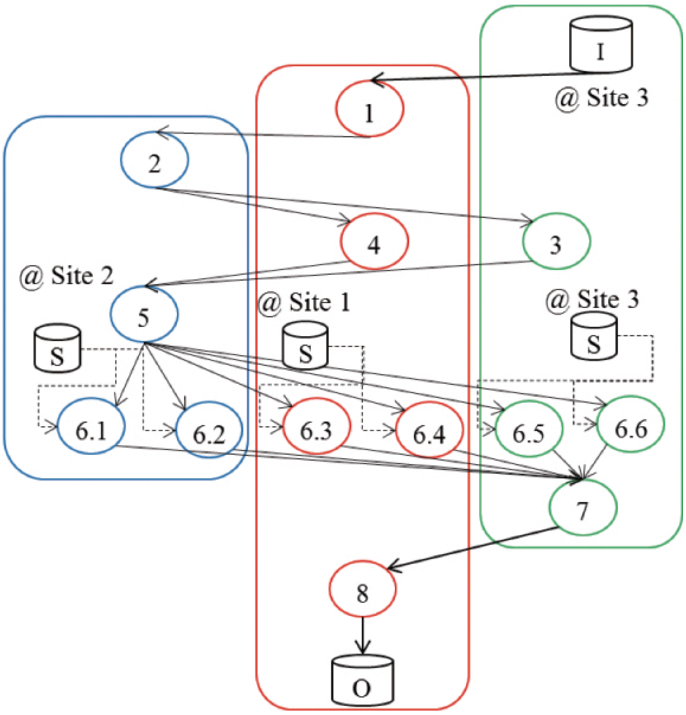


Figure 5.4: **Site greedy scheduling.**

minimum cost, namely $Sch(SWf, S) = \{Schedule(wf, s) | wf \in SWf, s \in S\}$ and $\forall\ wf \in SWf, \exists\ Schedule(wf, s) \in Sch(SWf, S)$ while cost $Cost(Schedule(wf, s))$ is the minimum compared to schedule Fragment wf to other sites. As a result, the cost $Cost(Sch(SWf, S))$ of executing a SWf in a multisite cloud is minimized. Similar to LocBased, ActGreedy schedules fragments of multiple activities. ActGreedy can schedule a pipeline of activities to reduce data transfer between different fragments, *i.e.* the possible data transfer between different sites. As formally defined in [158], a pipeline is a group of activities with a one-to-one, sequential relationship between them. However, ActGreedy is different from LocBased since it makes a trade-off between time and monetary costs. Similar to SGreedy, ActGreedy schedules the available fragments, while choosing the best site for an available fragment instead of choosing the best fragment for an available site.

ActGreedy chooses the best site for each fragment. First, it partitions a SWf according to the activity encapsulation partitioning method (Line 3). Then, it groups the fragments of three types into bigger fragments to be scheduled (Line 6). The first type is a pipeline of activities. We use a recursive algorithm presented in [158] to find pipelines. Then, the fragments of corresponding activities of each pipeline are grouped into a fragment. If there are stored activities of different sites in a fragment of a pipeline, the fragment is partitioned into several fragments by the data transfer minimization algorithm (Algorithm 1) in the $Group$ function. The second type is the control activities. If it has only one preceding activity, a control activity is grouped into the fragment of its preceding activity.

```
Input: $swf$: a scientific workflow; $S$: a set of sites
Output: $SP$: scheduling plan for $swf$ in $S$
 1: $SP \leftarrow \emptyset$
 2: $SWfCost \leftarrow \infty$
 3: $WF \leftarrow partition(swf)$
 4: do
 5:     $SP' \leftarrow \emptyset$
 6:     $WF \leftarrow Group(WF)$
 7:     do
 8:         $WFA \leftarrow GetAvailableFragments(WF, SP')$
 9:         if $WFA \neq \emptyset$ then
10:             for each $wf \in WFA$ do
11:                 $s_{opt} \leftarrow BestSite(wf, S)$
12:                 $SP' \leftarrow SP' \bigcup \{Schedule(wf, s_{opt})\}$
13:                 update $CurrentSWfCost$
14:             end for
15:         end if
16:     while not all the fragments $\in WF$ are scheduled
17:     if $CurrentSWfCost < SWfCost$ then
18:         $SP \leftarrow SP'$
19:         $SWfCost \leftarrow CurrentSWfCost$
20:     end if
21: while $CurrentSWfCost < SWfCost$
end
```

algorithm 5: Activity greedy scheduling

If it has multiple preceding activities and only one following activity, a control activity (Activity 7) is grouped into the fragment of its following activity (Activity 8). If it has multiple preceding activities and multiple following activities, a control activity (Activity 5) is grouped into the fragment of one of its preceding activities (Activity 3), which has the most data dependencies among all its preceding activities, *i.e.* the amount of data to be transferred in the data dependency is the biggest. It reduces data transfer among different fragments, namely the data transfer among different sites, to group the fragments for pipelines and control activities. The third type is the activities that are scheduled at the same site and that they have dependencies to connect each activity of them. Afterwards, Line 8 gets the available fragments (see Section 5.5.2) to be scheduled to the best site (Line $11 - 12$), which takes the minimal cost among all the sites to execute the fragment. The cost is estimated according to the method presented in Section 5.5.2. When estimating the cost, if the scheduled fragment has data dependencies with fixed activities, the cost to transfer the data in these data dependencies will be taken into consideration. The loop (Lines $7 - 14$) schedules each fragment to the best site while the big loop (Lines $4 - 18$) improves the scheduling plans by rescheduling the fragments after grouping the fragments at the same site, which ensures that the final scheduling plan corresponds to smaller cost to execute a SWf.

As shown in Figure 5.5, this algorithm has relatively small granularity compared with LocBased. ActGreedy exploits data location information to select the best site in order to make a trade-off between the cost for transferring data among different sites and another cost, *i.e.* the cost to provision the VMs and the cost to execute fragments. Figure 5.5 shows the scheduling result. If the amount of data increases and the desired execution time is small, Activity 7 and Activity 8 may be scheduled at Site 3, which takes less cost to transfer data. In order to avoid useless usage of VMs, the VMs at Site 1 are shut down when the site is waiting for the output data of the execution of Site 3, namely the execution of Activity 6.5 and Activity 6.6.

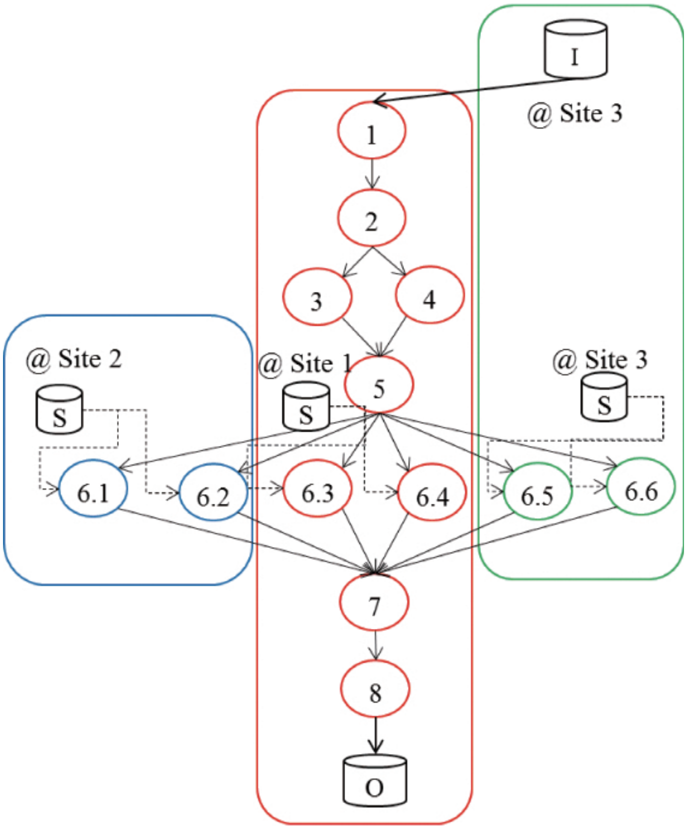


Figure 5.5: **Activity greedy scheduling.**

5.6.2.4 Solution analysis

Let us assume that we have n activities, s cloud sites and f fixed activities. The solution search space of a general scheduling problem is $\mathcal{O}(s^n)$. The solution search space of a scheduling problem after fixing the activities becomes $\mathcal{O}(s^{n-f})$. Even though the search space is reduced because of *stored data constraints*, the problem remains hard since the search space exponentially increases when n becomes bigger. For instance, assuming that

Table 5.2: **Parameters of different types of VMs.** Type represents the type of VMs. vCPUs represents the number of virtual CPUs in a VM. RAM represents the size of memory in a VM. Disk represents the size of hard disk in a VM. CC represents the computing capacity of VMs. MC represents Monetary Cost.

Type	vCPUs	RAM	Disk	CC	MC @ WE	MC @ JW	MC @ JE
A1	1	1.75	70	9.6	0.0447	0.0544	0.0604
A2	2	3.5	135	19.2	0.0894	0.1088	0.1208
A3	4	7	285	38.4	0.1788	0.2176	0.2416
A4	8	14	605	76.8	0.3576	0.4352	0.4832

we have a SWf with 77 activities (6 fixed activities) to be schedule at 3 sites, the search space of a general scheduling problem is $\mathcal{O}(3^{77})$, *i.e.* $\mathcal{O}(5.47 * 10^{36})$, and that of the scheduling problem with fixed activities is $\mathcal{O}(3^{71})$, *i.e.* $\mathcal{O}(7.51 * 10^{33})$. Some input or output activities may be related to the fixed activities, but they are free to be scheduled at any site. In general, the number of fixed activities is quite small compared with the number of other activities. The complexity of our proposed algorithm (ActGreedy) is $\mathcal{O}(s*(n-f))$, which is much smaller than $\mathcal{O}(s^{n-f})$. As a result, our solution can resolve the problem within reasonable scheduling time, *i.e.* the time to generate scheduling plans.

The knowledge of the location of stored data can be obtained by the metadata of files stored at each site, which is easy to get before SWf execution. Then, the knowledge of fixed activities can be generated with the dependencies between activities and data. Thus, knowing fixed activities is not a problem.

Our solution generates a scheduling plan that corresponds to the minimum cost to execute a SWf in a multisite cloud since all the fragments are scheduled to a site, which takes the minimum or near-minimum cost to execute them. The fragments of small granularity are scheduled to the best site, which takes the minimum cost to execute the fragments, by the small loop (Lines $7-14$) while the scheduling of fragments of big granularity, *i.e.* the activities of a site, is ensured by big loop (Lines $4 - 18$). Thus, our proposed algorithm can generate a scheduling plan which may minimize the cost. Since the weight of each objective is positive and the generated scheduling plan may minimize the sum function of multiple objectives, the solution may also be Pareto optimal [198] [128]. Although, in some rare cases, *e.g.* the cost to transfer data between different sites affects the scheduling plans, the cost corresponding to the generated scheduling plan is not minimum, our proposed solution generates a near-optimal scheduling plan. Since the experiments presented in this chapter are not rare cases, and that the scheduling plan generated by our algorithm is already Pareto optimal (no better scheduling plan can be found by estimating the cost of other scheduling plans), we do not compare it with another optimal solution, which may not even exist.

Note that the proposed algorithm and the results shown in Section 5.7 are sensitive to the cost model. Although the cost model is mentioned in other work [52], it is not used in a multisite environment with stored data constraints.

5.7 Experimental Evaluation

Table 5.3: **Estimated amount of data transferred in a dependency.** Input data represents the number of input fasta files for executing SciEvol SWf.

Dependency	Number of Fasta Files		
	100	500	1000
	Estimated Amount of Data		
Input Data	1	5	10
$e_{1,2}$	6	32	67
$e_{2,3}$	5	24	52
$e_{3,5}$	3	17	39
$e_{4,5}$	3	16	37
$e_{5,6.1}$	6	33	76
$e_{6.1,7}$	16	85	174
$e_{6.2,7}$	20	100	201
$e_{6.3,7}$	28	140	285
$e_{6.4,7}$	23	118	240
$e_{6.5,7}$	24	125	255
$e_{6.6,7}$	34	175	348
$e_{7,8}$	120	605	1215

In this section, we present an experimental evaluation of the fragment scheduling algorithms. All experiments are based on the execution of the SciEvol SWf in Microsoft Azure multisite cloud. We compare ActGreedy with LocBased and SGreedy, as well as with two general algorithms, *i.e.* Genetic and Brute-force. In the experiments, we consider three Azure [5] sites to execute SciEvol SWf, namely West Europe as Site 1, Japan West as Site 2, Japan East as Site 3. During the experiments, the the life circle of VM is composed of creation, start, configuration, stop and deletion. The creation, start, stop and deletion of a VM is managed by using Azure CLI. The configuration of VM is realized by Linux SSH command. In the experiments, workflow partitioner, multisite scheduler and single site initialization are simulated, but the execution of fragments is performed in a real environment by Chiron [139]. We conduct the experiments to show that ActGreedy takes the smallest cost (compared with LocBased and SGreedy) to execute a SWf in a multisite cloud within reasonable time (compared with Genetic and Brute-force) by making a trade-off of different objectives based on SSVP. Microsoft Azure provides 5 tiers of VM, which are basic tier, standard tier, optimized compute, performance optimized compute and compute intensive. Each tier of VM contains several types of VMs. In one Web domain, users can provision different types of VMs at the same tier. In our experiments, we consider 4 types, namely $A1$, $A2$, $A3$ and $A4$, in the standard tier. The features of the VM types are summarized in Table 5.2. In Azure, the time quantum is one minute. In addition, the average time to provision a VM is estimated as 2.9 minutes. Each VM uses

Linux Ubuntu 12.04 (64-bit), and is configured with the necessary software for SciEvol. All VMs are configured to be accessed using Secure Shell (SSH).

In the experiments, we use 100, 500, 1000 fasta files generated from the data stored in a genome database [7][9]. The programs used are similar to that presented in Section 4.5. The estimated workload (in GFLOP) of each activity of SciEvol SWf for different numbers of input fasta files is shown in Table 4.2. In Table 5.3, $e_{i,j}$ represents the data dependency between Activity i and Activity j while Activity j consumes the output data of Activity i. Let DataSize($e_{i,j}$) represent the estimated amount of data in dependency $e_{i,j}$. Then, we have DataSize($e_{2,3}$) = DataSize($e_{2,4}$); DataSize($e_{4,5}$) = DataSize($e_{3,5}$); DataSize($e_{5,6.1}$) = DataSize($e_{5,6.2}$) = DataSize($e_{5,6.3}$) = DataSize($e_{5,6.4}$) = DataSize($e_{5,6.5}$) = DataSize($e_{5,6.6}$).

In the tables and figures, the unit of time is minute, the unit of monetary cost is Euro, the unit of RAM and Disk is Gigabytes, the unit of data is MegaByte (MB), the computing capacity of VMs is GigaFLOPS (GFLOPS) and the unit of workload is GigaFLOP (GFLOP). ω_t represents the weight of time cost. $A1$, $A2$, $A3$ and $A4$ represent the types of VMs in Azure. [Type of VM] * [number] represents provisioning [number] of VMs of [Type of VM] type, *e.g.* A1 * 1 represents provisioning one VM of A1 type. WE represents West Europe; JW Japan West and JE Japan East. The cost corresponds to the price in Euro of Azure on July 27, 2015.

Table 5.4: **Setup parameters.** "Number" represents the number of input fasta files. "Limit" represents the maximal number of virtual CPUs that can be instantiated in the cloud. "DET" represents Desired Execution Time and "DMC" represents Desired Monetary Cost.

Number	100	500	1000
Limit	350	350	350
Estimated workload	$414,720$	$3,000,000$	$6,000,000$
DET	60	60	60
DMC	0.3	3	6

We present the experimental results to show that our proposed scheduling algorithm, *i.e.* ActGreedy, leads to the least cost for the execution of SWf in a multisite cloud environment. We schedule the fixed activities at the site where the data is stored and use the three scheduling algorithms, namely LocBased, SGreedy and ActGreedy, to schedule other activities of SciEvol at the three sites. In addition, we implemented a genetic algorithm and a brute-force algorithm that generate the best scheduling plans similar to those generated by ActGreedy. Brute-force measures the cost of all the possible scheduling plans and finds the optimal one, corresponding to the minimum cost to execute SWfs in a multisite cloud. The principle of a genetic algorithm [186] is to encode possible scheduling plans into a population of chromosomes, and subsequently to transform the population using standard operations of selection, crossover and mutation, producing successive generations, until the convergence condition is met. In the experiments, we set the convergence condition so that the cost of scheduling plans should be equal or smaller than

Table 5.5: **VM Provisioning Plans (100 fasta files).**

Algorithm	Site	ω_t		
		0.1	0.5	0.9
LocBased	WE	$A3*1$	$A4*1$	$A4*2$
	JW	$A3*1$	$A4*1$	$A4*1$
	JE	$A1*1, A2*1$	$A4*1$	$A4*3$
SGreedy	WE	$A3*1$	$A4*1$	$A4*2$
	JW	$A2*1, A3*1$	$A4*1$	$A4*1$
	JE	$A1*1, A2*1$	$A4*1$	$A4*3$
ActGreedy	WE	$A2*1$	$A2*1; A3*1$	$A4*2$
	JW	$A3*1$	$A4*1$	$A4*1$
	JE	$A1*1, A2*1$	$A4*1$	$A4*2$

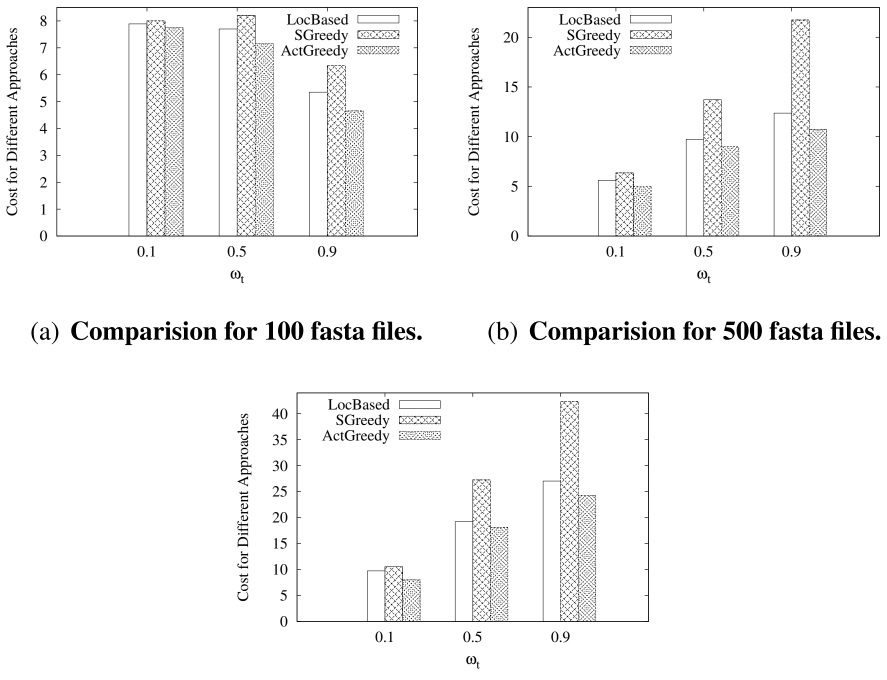
Table 5.6: **VM Provisioning Plans (500 fasta files).**

Algorithm	Site	ω_t		
		0.1	0.5	0.9
LocBased	WE	$A3*1$, $A4*1$	$A4*4$	$A4*7$
	JW	$A4*1$	$A4*2$	$A4*3$
	JE	$A1*1, A4*1$	$A4*3$, $A3*1$	$A4*8$
SGreedy	WE	$A3*1$, $A4*1$	$A4*4$	$A4*7$
	JW	$A4*1$	$A4*2$	$A4*3$
	JE	$A1*1, A3*1$	$A3*1$, $A4*3$	$A4*8$
ActGreedy	WE	$A4*1$	$A2*1; A4*2$	$A4*5$
	JW	$A4*1$	$A4*2$	$A4*3$
	JE	$A1*1, A4*1$	$A3*1$, $A4*3$	$A4*9$

Table 5.7: **VM Provisioning Plans (1000 fasta files).**

Algorithm	Site	ω_t		
		0.1	0.5	0.9
LocBased	WE	$A2*1$, $A4*1$	$A4*6$	$A4*9$
	JW	$A4*2$	$A4*3$	$A4*4$
	JE	$A2*1, A3*1, A4*1$	$A4*5$	$A4*11$
SGreedy	WE	$A4*2$	$A4*6$	$A4*10$
	JW	$A4*2$	$A4*3$	$A4*4$
	JE	$A2*1, A3*1, A4*1$	$A4*5$	$A4*11$
ActGreedy	WE	$A2*1, A4*1$	$A4*4$	$A4*8$
	JW	$A4*2$	$A4*3$	$A4*4$
	JE	$A2*1, A3*1, A4*1$	$A4*6$	$A4*12$

that generated by ActGreedy. We use 100 chromosomes and set the number of generations as 1 for the experiments of different numbers of input files and different values of α. We choose a random point for the crossover and mutation operation. The experimental results[2] are shown in Figure 5.6. The setup parameters are shown in Table 5.4 and provisioning plans, which are generated by SSVP, are listed in Table 5.5, Table 5.6 and Table 5.7. We assume that the data transfer rate between different sites is 2 MB/s. The monetary cost to transfer data from Site 1 to other sites is 0.0734 Euros/GB and the monetary cost for Site 2 and Site 3 is 0.1164 Euros/GB. In the experiments, the critical path of SciEvol SWf is composed of Activities 1, 2, 4, 5, 6.6, 7, 8.



(a) **Comparision for 100 fasta files.**

(b) **Comparision for 500 fasta files.**

(c) **Comparision for 1000 fasta files.**

Figure 5.6: **Cost for different scheduling algorithms.** The cost is calculated according to Formula 5.2.

LocBased is optimized for reducing data transfer among different sites. The scheduling plan generated by this algorithm is shown in Figure 5.3. However, the different monetary costs of the three sites are not taken into consideration. In addition, this algorithm directly schedules a fragment, which contains multiple activities. Some activities are scheduled at a site, which is more expensive to use VMs, *e.g.* Site 3. As a consequence, the scheduling plan may correspond to higher cost. SGreedy schedules a site to the available activity, which takes the least cost. The corresponding scheduling plan is shown in

[2] In the experiments, in order to facilitate the data transfer of many small files, we use the *tar* command to archive the small files into one big file before data transferring.

Figure 5.4. SGreedy does not take data location into consideration and may schedule two continuous activities, *i.e.* one preceding activity and one following activity, to two different sites, which takes time to transfer data and to provision VMs. As a result, this algorithm may lead to higher cost. ActGreedy can schedule each fragment to a site that takes the least cost to execute it, which leads to smaller cost compared with LocBased and SGreedy. In addition, ActGreedy can make adaptive modification for different numbers of input fasta files. For instance, there are three situations where Activity 7 and Activity 8 are scheduled at Site 3 to reduce the cost of data transfer while the other scheduling plans are the same as shown in Figure 5.5. The three situations are when there are 500 input fasta files and $\omega_t = 0.9$ and when there are 1000 input fasta files and $\omega_t = 0.5$ or $\omega_t = 0.9$. Note that SWf partitioning algorithms may also have impact on the performance of the scheduling algorithm. For instance, LocBased is based on a SWf partitioning algorithm to reduce data transfer among different SWf fragments.

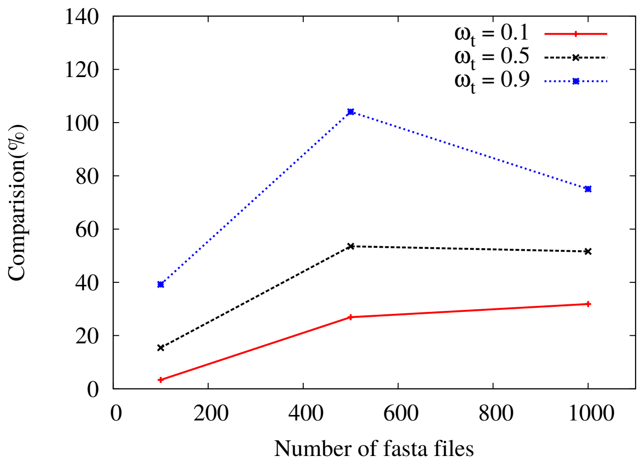
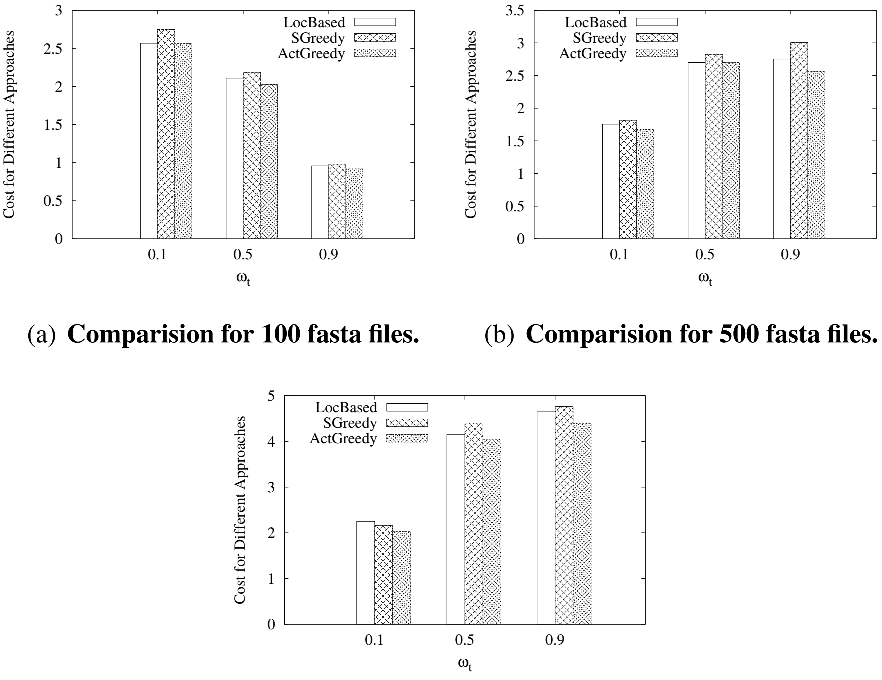


Figure 5.7: **Comparision of cost between SGreedy and ActGreedy for different number of input fasta files.** The cost is calculated according to Formula 5.2.

First, we analyze the cost based on Formula 5.2 and Formula 5.3. The time and monetary costs to execute a fragment at a site are measured during execution. Both the time and monetary costs are composed of three parts, *i.e.* site execution, data transfer and fragment execution. Based on Formulas 5.3, 5.4, 5.9, 5.5, 5.10, the cost to execute a fragment is calculated. Based on Formula 5.2 and Formula 5.3, the cost to execute a SWf is calculated. The cost corresponding to 100 fasta files is shown in Figure 5.6(a). In order to execute SciEvol SWf with 100 fasta files, ActGreedy can reduce 1.85% ($\omega_t = 0.1$), 7.13% ($\omega_t = 0.5$) and 13.07% ($\omega_t = 0.9$) of the cost compared with LocBased. Compared with SGreedy, ActGreedy can reduce 3.22% ($\omega_t = 0.1$), 12.80% ($\omega_t = 0.5$) and 26.60% ($\omega_t = 0.9$) of the cost. Figure 5.6(b) shows the cost for different values of ω_t for processing 500 fasta files. The experimental results show that ActGreedy is up to 13.15% ($\omega_t = 0.9$) better than LocBased and up to 50.57% ($\omega_t = 0.9$) better than SGreedy for 500 fasta files. In addition, Figure 5.6(c) shows the experimental results for 1000 fasta files. The results show that LocBased takes up to 21.75% ($\omega_t = 0.1$) higher cost than ActGreedy and that SGreedy takes up to 74.51% ($\omega_t = 0.9$) higher cost than ActGreedy when processing 1000 fasta files. Figure 5.7 describes the difference between the worst case (SGreedy) and the best case (ActGreedy). In Figure 5.7, the Y axis represents the advantage[3] of ActGreedy compared with SGreedy. It can be seen from Figure 5.7 that ActGreedy outperforms SGreedy and its advantage becomes obvious when ω_t grows. When there are more input fasta files, the advantage is bigger at first. But it decreases when the number of fasta files grows from 500 to 1000 since the cost corresponding to ActGreedy increases faster.



(a) **Comparision for 100 fasta files.**

(b) **Comparision for 500 fasta files.**

(c) **Comparision for 1000 fasta files.**

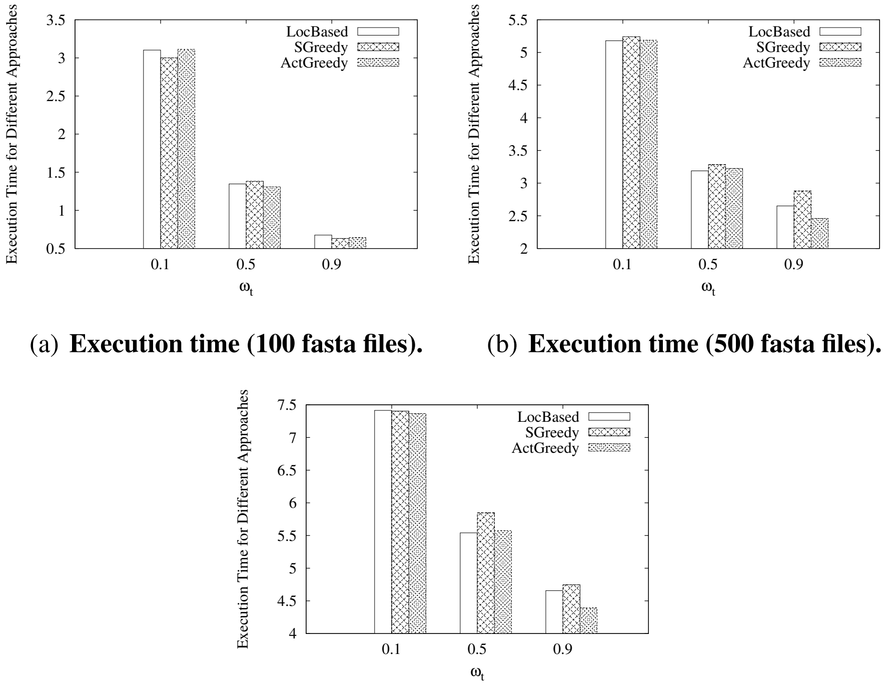
Figure 5.8: **Cost for different scheduling algorithms.** According to Formula 5.1.

Accordingly, the cost calculated according to Formula 5.1 is shown in Figure 5.8. In the real execution, the time and monetary costs for the whole execution of a SWf are measured and the real cost can be calculated by Formula 5.1. From Figures 5.8(a), 5.8(b) and 5.8(c), we can see that the cost corresponding to ActGreedy is smaller than that of SGreedy at all the situations. Except in one situation, ActGreedy performs better than LocBased. When the number of input fasta files is 500 and $\omega_t = 0.5$, the cost for the

[3] The advantage is calculated based on the following formula:

$$Advanage = \frac{Cost_{SGreedy}(\omega_t) - Cost_{ActGreedy}(\omega_t)}{Cost_{ActGreedy}(\omega_t)} * 100\% \qquad (5.13)$$

data transfer becomes important. In this case, LocBased performs slightly better than ActGreedy. However, the advantage of LocBased (0.03%) is very small and this may be because of the dynamic changing environment in the Cloud. As the number of input fasta files increases, the advantage of ActGreedy becomes obvious. Compared with LocBased, ActGreedy is up to 4.1% (100 fasta files and $\omega_t = 0.9$), 7.4% (500 fasta files and $\omega_t = 0.9$) and 10.7% (1000 fasta files and $\omega_t = 0.1$) better. Compared with SGreedy, the advantage of ActGreedy can be up to 7.5% (100 fasta files and $\omega_t = 0.5$), 17.2% (500 fasta files and $\omega_t = 0.9$) and 8.8% (1000 fasta files and $\omega_t = 0.5$).



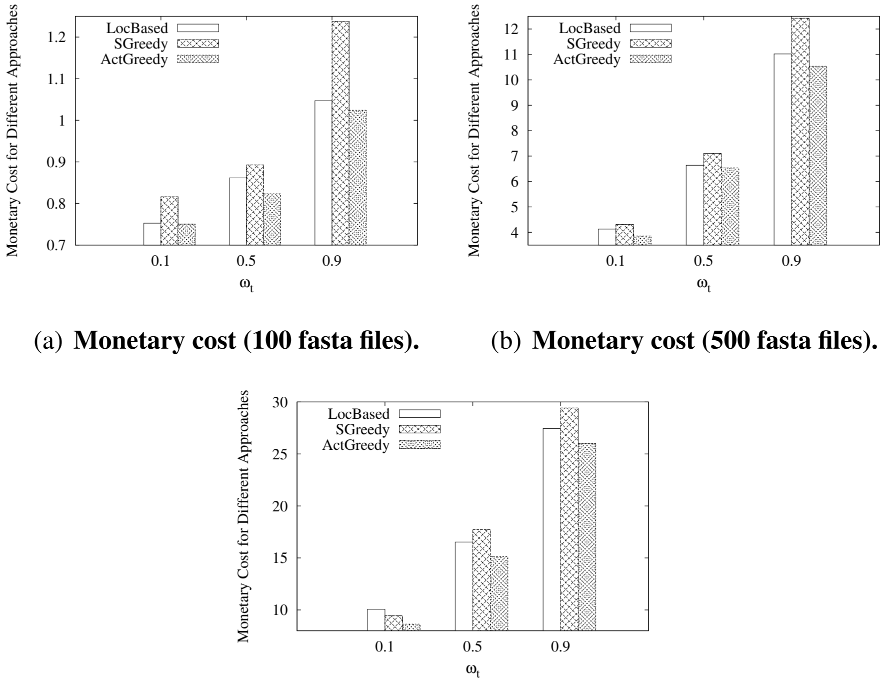
(a) **Execution time (100 fasta files).** (b) **Execution time (500 fasta files).**

(c) **Execution time (1000 fasta files).**

Figure 5.9: **Execution time of SciEvol with different scheduling approaches.**

Figures 5.9 and 5.10 show the execution time and the monetary costs for the execution of SciEvol with different amounts of input data and different values of ω_t. When ω_t increases, the execution time is largely reduced and the monetary cost increases. When the weight of execution time cost is low, *i.e.* $\omega_t = 0.1$, Compared with LocBased and SGreedy, ActGreedy may correspond to more execution time while it generally takes less monetary cost. When the weight of execution time cost is high, $\omega_t = 0.9$, ActGreedy corresponds to less execution time. The execution with ActGreedy always takes less monetary cost compared with LocBased (up to 14.12%) and SGreedy (up to 17.28%). The reason is that ActGreedy can choose a cheap site to execute fragments, namely the monetary cost to instantiate VMs at that site is low. As a result, ActGreedy makes a good trade-off between execution time and monetary costs for the execution of SciEvol at a multisite cloud.

Furthermore, we measured the amount of data transferred among different sites, which is shown in Figure 5.11. Since LocBased is optimized for minimizing data transferred between different sites, the amount of intersite transferred data with LocBased remains minimum when the number of input fasta files varies from 100 to 1000. The amount of transferred data corresponding to ActGreedy is slightly bigger than that of LocBased and the difference is between 1.0% and 13.4%. SGreedy has the biggest amount of intersite transferred data. Compared with ActGreedy, the amount of intersite transferred data of SGreedy is up to 122.5%, 139.2% and 148.1% bigger when the number of input fasta files is 10, 500 and 1000. In addition, the amount of data transfer with ActGreedy decreases for the three cases, *i.e.* 500 input fasta files with $\omega_t = 0.9$ and 1000 input fasta files with $\omega_t = 0.5$ or $\omega_t = 0.9$. The reason is that Activities 7 and 8 are scheduled at the same site as Activities 6.5 and 6.6, namely Site 3, which reduces data transfer. Furthermore, when the data transfer rate between different sites decreases, the performance of SGreedy will be much worse since the time to transfer big amounts of data will be much longer.



(a) **Monetary cost (100 fasta files).**

(b) **Monetary cost (500 fasta files).**

(c) **Monetary cost (1000 fasta files).**

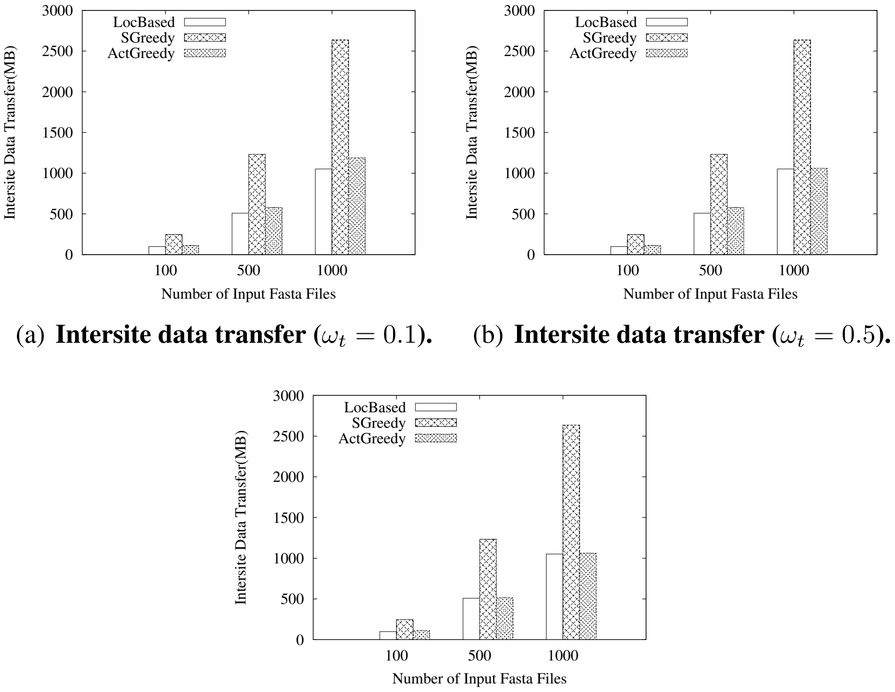
Figure 5.10: **Monetary cost of SciEvol execution with different scheduling approaches.**

In addition, we measure the idleness of the virtual CPUs according to following formula:

$$Idleness = \frac{\sum_{i=1}^{n} IdleTime(CPU_i)}{\sum_{i=1}^{n} TotalTime(CPU_i)} * 100\% \tag{5.14}$$

where n represents the number of virtual CPUs, $IdleTime$ represents the time when the virtual CPU is not working for the execution of the programs of SciEvol SWf. $TotalTime$ represents the total time that the virtual CPU is instantiated.

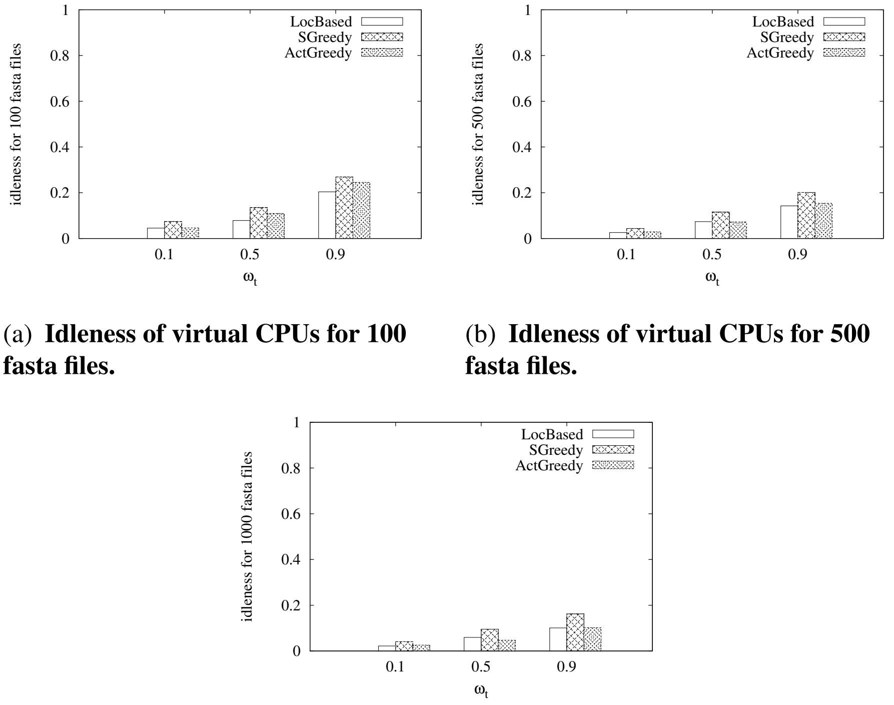
Figure 5.12 shows the idleness of virtual CPUs corresponding to different scheduling algorithms and different amounts of fasta files. From the figure, we can see that as ω_t increases, the idleness becomes bigger. When ω_t increases, the importance of execution time becomes bigger and more VMs are provisioned to execute fragments. In this case, the time to start the VMs becomes higher compared with execution time. As a result, the corresponding idleness goes up. In addition, when the amount of input files, *i.e.* fasta files, rises, the idleness decreases. The reason is that at this situation, more time is used for the execution of SWf for the increased workload. The figure also shows that LocBased has the smallest idleness while SGreedy has the biggest idleness. This is expected since the VMs at Sites 1 and 2 need to be shut down and restarted during the execution with SGreedy and that the VMs at Site 1 needs to be shut down and restarted during the execution with ActGreedy. The time to restart VMs at a site may consume several minutes while the virtual CPUs are not used for the execution of fragments. Figure 5.12(a), Figure 5.12(b) and Figure 5.12(b) show the experimental results for the corresponding idleness of virtual CPUs. From the figures, we can see that the idleness of ActGreedy is generally bigger than that of LocBased while it is always smaller than that of SGreedy. The idleness of ActGreedy is up to 38.2% ($\omega_t = 0.5$) bigger than that of LocBased and up to 37.8% ($\omega_t = 0.1$) smaller than that of SGreedy for 100 fasta files. The idleness of ActGreedy is



(a) **Intersite data transfer** ($\omega_t = 0.1$). (b) **Intersite data transfer** ($\omega_t = 0.5$).

(c) **Intersite data transfer** ($\omega_t = 0.9$).

Figure 5.11: **Intersite data transfer for different scheduling algorithms.**



(a) **Idleness of virtual CPUs for 100 fasta files.**

(b) **Idleness of virtual CPUs for 500 fasta files.**

(c) **Idleness of virtual CPUs for 1000 fasta files.**

Figure 5.12: **Idleness of virtual CPUs for different scheduling algorithms.** According to Formula 5.14.

up to 12.4% ($\omega_t = 0.1$) bigger than that of LocBased and up to 38.0% ($\omega_t = 0.5$) smaller than that of SGreedy for 500 fasta files. For 1000 fasta files, the idleness of ActGreedy is 18.6% ($\omega_t = 0.1$) and 1.0% ($\omega_t = 0.9$) bigger than that of LocBased. When $\omega_t = 0.5$, the idleness of ActGreedy is 20.9% smaller than that of LocBased. In addition, the idleness of ActGreedy is up to 50.9% ($\omega_t = 0.5$) smaller than that of SGreedy.

Finally, we study the scheduling time of different algorithms. In order to show the effectiveness of ActGreedy, we compare it with our two other algorithms, *i.e.* SGreedy and LocBased, and two more general algorithms, *i.e.* Genetic and Brute-force.

Table 5.8: **Number of generations.**

Number of sites	3	4	5	6	7	8	9	10	11		
Number of generations	1	1	3	10	28	71	162	337	657		
Number of activities	13	14	15	16	17	18	19	20	21	22	23
Number of generations	1	1	1	2	6	18	54	162	484	1450	4349

An example of scheduling time corresponding to 3 sites and 13 activities is shown in Table 5.9. This is a small example since the time necessary to schedule the activities may

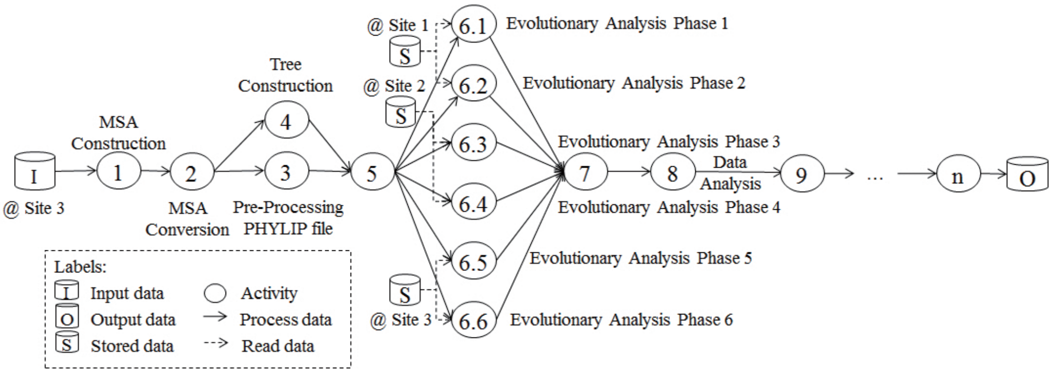


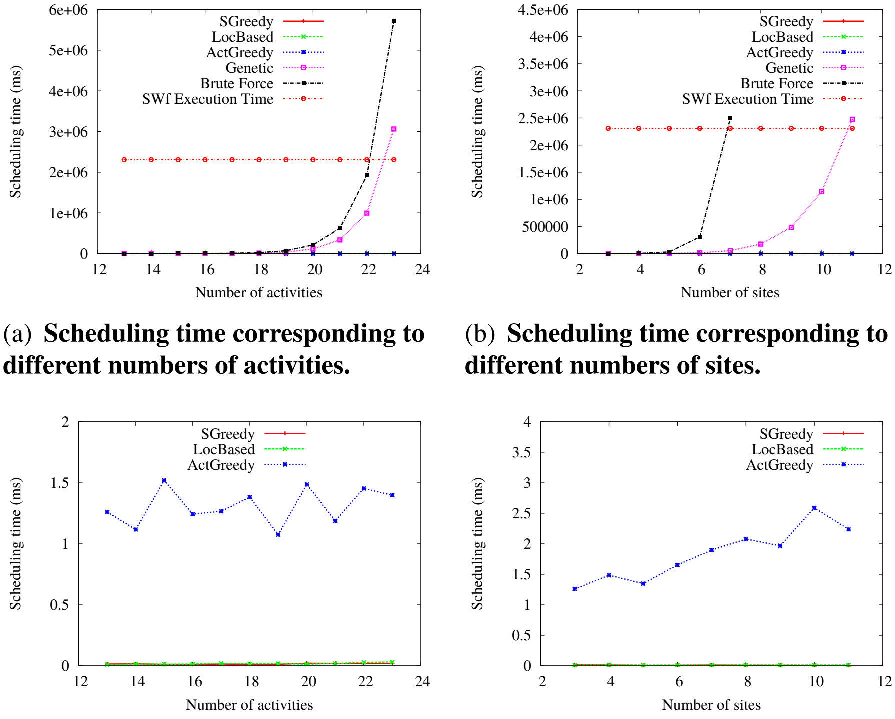
Figure 5.13: **SciEvol SWf.** Activities $9 - n$ are added control activities, which have no workload.

be unfeasible for Brute-force and Genetic when the numbers of activities or sites become high. Then, we vary the numbers of activities or sites. When we increase the number of sites, we fix the number of activities at 13 and when we increase the number of activities, we fix the number of sites to 3. The number of generations for different numbers of activities or different numbers of sites is shown in Table 5.8. Since the search space gets bigger when the number of activities or sites increases, we increase the number of generations in order to evaluate at least 30% of all the possible scheduling plans for Genetic. We add additional control activities in the SciEvol SWf, which have little workload but increase the search space of scheduling plans. The modified SciEvol SWf is shown in Figure 5.13 and the scheduling time corresponding to different numbers of activities is shown in 5.14(a). In addition, we measure the scheduling time corresponding to different numbers of sites while using the original SciEvol SWf, as shown in Figure 5.14(b). The unit of scheduling time is millisecond. The data constraint remains the same while the number of input files is 100 and α equals to 0.9. In the experiments, only ActGreedy generates the same scheduling plans as that of Brute-force. Since the point of mutation operation and the points of crossover operation of Genetic are randomly selected, the scheduling plans generated by Genetic may not be stable, *i.e.* the scheduling plans may not be the same for each execution of the algorithm. Both LocBased and SGreedy cannot generate the optimal scheduling plans as that of Brute-force.

Table 5.9: **Comparison of scheduling algorithms.**

Algorithms	Scheduling time (ms)
LocBased	0.010
SGreedy	0.014
ActGreedy	1.260
Genetic	727
Brute-force	161

Table 5.9 shows that the scheduling time of Genetic and Brute-force is much longer



(a) **Scheduling time corresponding to different numbers of activities.**

(b) **Scheduling time corresponding to different numbers of sites.**

(c) **Scheduling time corresponding to different numbers of activities (zoom on LocBased, SGreedy and ActGreedy).**

(d) **Scheduling time corresponding to different numbers of sites (zoom on LocBased, SGreedy and ActGreedy).**

Figure 5.14: **Scheduling time.**

than ActGreedy (up to 577 times and 128 times). Genetic may perform worse than Brute-force for a small number of activities or sites with the specific configuration. The scheduling time of Genetic is smaller than that of Brute-force when the number of activities or sites increases as shown in Figure 5.14(a) and 5.14(b). Figures 5.14(a) and 5.14(b) show that the scheduling time of ActGreedy, SGreedy and LocBased is much smaller than that of Genetic and Brute-force. The scheduling time of ActGreedy, SGreedy and LocBased is represented by the bottom line in Figures 5.14(a)5.14(b). Even though when the number of activities and the number of sites is small, the scheduling time is negligible compared with the execution time, it becomes significant when the numbers of activities or sites increase. For instance, with more than 22 activities or 6 sites, the scheduling time of Brute-force exceeds the execution while the scheduling time of ActGreedy remains small. This is because of the high complexity of Brute-force, which is $\mathcal{O}(s^{n-f})$. With more than 22 activities or 10 sites, the scheduling time of Genetic is bigger than the execution time. Because of long scheduling time, Genetic and Brute-force are not suitable for SWfs with a big number of activities, *e.g.* Montage [6] may have 77 activities. In addition, according to [114], 22 activities are below the average number of SWf activities. As a result, Genetic and Brute-force are unfeasible for multisite scheduling of most SWfs.

These two algorithms are not suitable for SWf scheduling with a big number of sites. For instance, Azure has 15 sites (regions). Although the scheduling time of ActGreedy is much bigger (see Figures 5.14(c)5.14(d) for details) than that of SGreedy and LocBased, it remains reasonable compared with the overall SWf execution time.

The experimental results show that although ActGreedy may yield more data transferred among different sites and higher idleness (compared with LocBased), it generally yields smaller cost compared with both LocBased and SGreedy and the scheduling time of ActGreedy is much lower than that of Genetic and Brute-force.

5.8 Conclusion

Scientists usually make intensive usage of parallelism techniques in HPC environments. However, it is not simple to schedule and manage executions of SWfs, particularly in multisite cloud environments, which present different characteristics in comparison with a single site cloud. To increase the uptake of the cloud model for executing SWfs that demand HPC capabilities provided by a multisite cloud and to benefit from data locality, new solutions have to be developed, especially for scheduling SWf fragments in cloud resources. In previous work [52] we have addressed SWf execution in a single site cloud using a scheduling algorithm but these solutions are not suitable for a multisite cloud.

In this chapter, we proposed a new multi-objective scheduling approach, *i.e.* ActGreedy, for SWfs in a multisite cloud (from the same provider). We first proposed a novel multi-objective cost model, which aims at minimizing two costs: execution time and monetary costs. Our proposed fragment scheduling approach that is ActGreedy, allows for considering stored data constraints while reducing the cost based on the multi-objective cost model to execute a SWf in a multisite cloud. We used a real SWf that is SciEvol, with real data from the bioinformatics domain as a use case. We evaluated our approaches by executing SciEvol in Microsoft Azure cloud. The results show that since ActGreedy makes a good trade-off between execution time and monetary costs, ActGreedy leads to the least total normalized cost, which is calculated based on the multi-objective cost model, than LocBased (up to 10.7%) and SGreedy (up to 17.2%) approaches. In addition, compared with LocBased (up to 14.12%) and SGreedy (up to 17.28%), ActGreedy always corresponds to less monetary cost since it can choose cheap cloud sites to execute SWf fragments. Furthermore, compared with SGreedy, ActGreedy corresponds to more than two times smaller amounts of transferred data. Additionally, ActGreedy scales very well, *i.e.* it takes a very small time to generate the optimal or near optimal scheduling plans when the number of activities or sites increases, compared with general approaches, *e.g.* Genetic and Brute-force.

Chapter 6

Task Scheduling with Provenance Support in Multisite Clouds

Recently, some Scientific Workflow Management Systems (SWfMSs) with provenance support (*e.g.* Chiron) have been deployed in the cloud. However, they typically use a single cloud site. In this chapter, we consider a multisite cloud, where the data and computing resources are distributed at different sites (possibly in different regions). Based on a multisite architecture of SWfMS, *i.e.* multisite Chiron, and its provenance model, we propose a multisite task scheduling algorithm that considers the time to generate provenance data. This thesis is based on [121].

Section 6.3 presents the problems for task scheduling of SWf execution in a multisite cloud environment. Then, Section 6.4 gives the design of a multisite SWfMS. Afterwards, Section 6.5 explains our proposed scheduling algorithm. Section 6.6 gives our our extensive experimental evaluation of the algorithm using Microsoft Azure multisite cloud and two real-life scientific workflows (Buzz and Montage). The results show that our scheduling algorithm is much better than baseline algorithms in terms of execution time and the amounts of intersite transferred data.

6.1 Proposal Overview and Motivations

SWfs are generally used to model the data processing of large scale *in silico* scientific experiments as a graph, in which vertices represent data processing activities and edges represent dependencies between them. Since SWf activities may process multiple data chunks, one activity can correspond to several executable tasks for different parts of input data during SWf execution. Thus, efficiently executing data-intensive SWfs, *e.g.* Montage [6] and Buzz [64], becomes an important issue.

Some implementations of SWfMSs are publicly available, *e.g.* Pegasus [60] and Chiron [139]. A SWfMS generally supports provenance data, which is the metadata that captures the derivation history of a dataset [120], during SWf execution. Provenance data, which is used for SWf analysis and SWf reproducibility, may be as important as the scientifc experiment itself [120]. The provenance data is typically stored in a database to provide on-line provenance query [129], and contains the information regarding activities, tasks and files. During the execution of a task, there may be multiple exchanges of provenance data between the computing node and the provenance database.

Recently, some SWfMSs with provenance support (*e.g.* Chiron) have been deployed in the cloud. However, they typically focus the execution of a SWf at a single cloud site or in even a single computing node [92][93]. Although there are some multisite solutions [65][150], they do not support provenance data, which is important for the analysis of SWf execution. However, the data and computing resources (including programs) necessary to run a SWf may well be distributed at different sites (possibly in different regions), *e.g.* because of collaboration between different groups of scientists. And it may not be always possible to move all the resources to a single site for execution. Chapter 5 focuses on the scheduling of fragments, which is coarse-grained and cannot address the problem of executing an activities at multiple sites. In this chapter, we consider a multisite cloud that is composed of several sites (or data centers) of the same cloud provider, each with its own resources and data for the execution of each activity. In addition, we also take into consideration of the influence of the functionality of provenance data on the SWf multisite execution.

To enable SWf execution in a multisite cloud with distributed input data, the execution of the tasks of each activity should be scheduled to a corresponding cloud site (or site for short). Then, the scheduling problem is to decide at which sites to execute the tasks in order to achieve a given objective, *e.g.* reducing execution time. Compared with the approach of scheduling activities at a single site [122], the task scheduling is fine-grained, which enables the execution of the same activity at different sites to deal with distributed data and programs. Furthermore, since it may take much time to transfer data between two different sites, the multisite scheduling problem should take into account the resources at different sites, *e.g.* different bandwidths.

We focus on the task scheduling problem to reduce the makespan, *i.e.* the execution time, of executing a SWf in a multisite cloud. We use a distributed SWfMS architecture with a master site that coordinates the execution of each site and that stores all the provenance data of SWf execution. In this architecture, the intersite transferred data can be intermediate data or provenance data produced by SWf execution. The intermediate data is the data generated by executing activities and can also be the input data for the tasks of following activities. In the multisite cloud, the bandwidth between two different sites (of different regions) may be small. For data-intensive SWfs, there may be many data, *e.g.* intermediate data and provenance data, to transfer across different sites for the execution of a task while the time to execute the task can be very small, *e.g.* a few seconds or even less than one second. As a result, the time to transfer intermediate data and the time to generate the provenance data connot be ignored in the scheduling process. Thus, we also consider the time to transfer both the intermediate data and the provenance data in the scheduling process in order to better reduce the overall execution time of SWf execution.

We make the following contributions. First, we propose multisite Chiron, with a novel architecture to execute SWfs in a multisite cloud environment while generating provenance data. Second, an extended multisite provenance model and global provenance management of distributed provenance data in multisite cloud. Third, we propose a novel multisite task scheduling algorithm, *i.e.* Data-Intensive Multisite task scheduling (DIM), for SWf execution with provenance support in multisite Chiron. Fourth, we make an extensive experimental evaluation, based on the implementation of multisite Chiron in Microsoft Azure, and using two real SWf use cases (Buzz and Montage).

6.2 Related Work

Classic scheduling algorithms, *e.g.* Opportunistic Load Balancing (OLB) [125], Minimum Completion Time (MCT) [125], min-min [67], max-min [67] and Heterogeneous Earliest Finish Time (HEFT) [186], address the scheduling problem for the objective of reducing execution time within a single site. The OLB algorithm randomly assigns each task to an available computing node without considering the feature of the task or the computing node. The MCT algorithm schedules each task to the computing node that can finish the execution first. HEFT gives the priority to each task according to the dependencies of tasks and the workload of the task. Then, it schedules the tasks with the highest priority to the computing node that can finish the execution first. The min-min algorithm schedules the task, which takes the least time to execute, to the computing node that can finish the execution first. The max-min algorithm schedules the task, which takes the biggest time to execute, to the computing node that can finish the execution first. For the tasks of the same activity, they are independent of each other and have the same estimated execution time. As a result, the HEFT, min-min and max-min algorithms degrade to the MCT algorithm for this kind of tasks. Some other solutions [168][177][196] for SWf scheduling also focus on single site execution. These techniques do not consider the time to generate provenance data. Dean and Ghemawat [55] propose to schedule tasks to where the data is. Although this method focuses on single site, it considers the cost to transfer data among different computing nodes. However, this algorithm depends on the location of data. When the data is not evenly distributed at each computing node, this algorithm may lead to unbalanced load at some computing nodes and long execution time of tasks. De Oliveira *et al.* [52] propose a provenance based task scheduling algorithm for single site cloud environments. Some adaptation of SWfMSs [28][34] in the cloud environment can provide the parallelism in workflow level or activity level, which is coarse-grained, at a single site cloud. These methods cannot perform parallelism of the tasks of the same activities and they cannot handle the distributed input data at different sites.

Duan *et al.* [65] propose a multisite multi-objective scheduling algorithm with consideration of different bandwidths in a multisite environment. However, they do not consider the input data distribution at different sites and do not provide provenance support, , which may incur much time for intersite provenance data transfer. In Chapter 5, we proposed a solution of multisite activity scheduling of SWfs according to the data location. However, the activity scheduling method is coarse-grained: it can schedule the execution of each activity to a site but cannot schedule different tasks of one activity to different sites. Thus, it

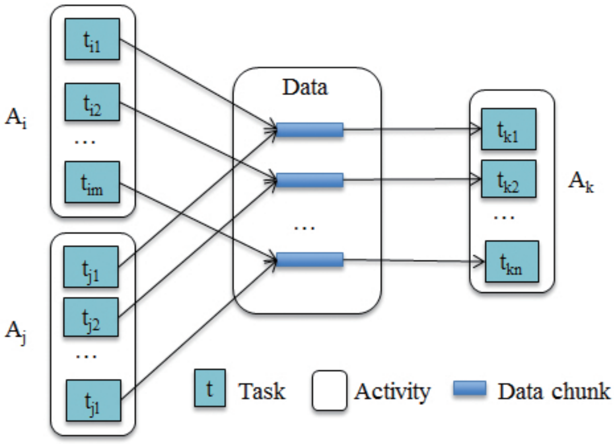


Figure 6.1: **Activity and tasks.**

cannot handle the distributed input data of a SWf in a multisite cloud environment. Luis *et al.* [150] propose caching metadata in the memory and replicating the metadata for SWf execution in a multisite cloud. The metadata is the description information of files at each site. In this method, data transfer is analyzed in multi-site SWf execution, stressing the importance of optimizing data provisioning. However, the metadata is not yet explored on task scheduling and, they just simulated SWf execution in the experiments. Their method can be used to optimize the metadata management in our multisite Chiron in the future. Hadoop [185] is extended to multiple sites while the the existing approaches do not consider the provenance support or different bandwidths among different sites for the task scheduling [184].

6.3 Problem Definition

This section introduces some important terms, *i.e.* SWf and multisite cloud, and formally defines the task scheduling problem we address.

A SWf can be described as a Directed Acyclic Graph (DAG) denoted by $W(V,E)$. Let V = $\{v_1, v_2, ..., v_n\}$ be a set of vertices, which represent the scientific data processing activities. E = $\{e_{i,j}$: $v_i, v_j \in V$ and Activity v_j consumes the output data of Activity v_i $\}$ represents a set of edges that correspond to dependencies between activities in V. Activity v_i is the parent activity of Activity v_j and Activity v_j is the child activity of Activity v_i. If it has no parent activity, an activity is a start activity. If it has no child activity, an activity is an end activity. If it has neither parent activity nor child activity, an activity is an intermediate activity. Since an activity may process big amount of data, it corresponds to multiple tasks. Thus, as shown in Figure 6.1, an activity A_k may have n tasks $\{t_1, t_2, ..., t_n\}$, each consuming a data chunk produced by the tasks of parent activities of Activity A_k, *i.e.* Activities A_i and A_j. A data-intensive SWf is the SWf that it is difficult to manage or transfer data compared to the data processing. For instance, for data-intensive SWfs, the time to transfer data cannot be ignored compared with the

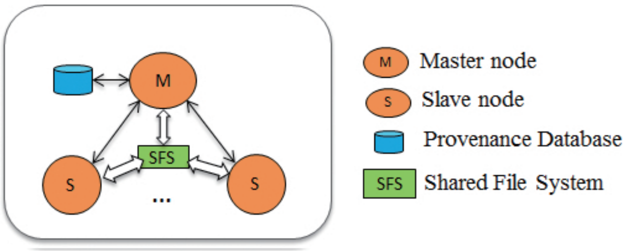


Figure 6.2: **Architecture of single site Chiron.**

time to process data. Different from data-intensive SWfs, the time to transfer data can be ignored for computing-intensive SWfs since it takes much more time to process data compared with the time to transfer data.

As defined in [119], a multisite cloud is a cloud with multiple distributed data centers of the same cloud provider, each being explicitly accessible to cloud users. A multisite cloud configuration defines the instances of Virtual Machines (VMs) and storage resources for cloud users at a multisite cloud. The configured[1] multisite cloud $MS(S)$ consists of a set of cloud sites S with instantiated VMs and data at each site. In this chapter, a cloud site corresponds to a cluster of VMs, data and cloud services, *e.g.* database and message queue service. In the cluster of VMs, each VM is a computing node.

We assume that the input data of the SWf cannot be moved across different sites. Thus, the tasks of the start activity should be scheduled at the site where the data is. We assume that the intermediate data can be moved across different sites. Thus, the tasks of the intermediate activities or end activities can be scheduled at any site. During SWf execution, the tasks of each activity are generated independently and the scheduling of the tasks of each activity is done independently. Thus, we need to group tasks of the same activities in bags. Then, a bag of tasks T is a set of tasks corresponding to the same activity. In addition, we assume that the time to transfer the input data of tasks between two different sites and the time to generate provenance data is non-negligible compared with the execution time of a task. Scheduling tasks is to choose the sites in S to execute a bag of tasks T, *i.e.* mapping each task to an execution site. In this chapter, we assume that the input data of the bag of tasks T is distributed at different sites. A scheduling plan defines the mapping of tasks to sites. Thus, the task scheduling problem is how to generate scheduling plans for a bag of tasks, which corresponds to an activity of a SWf, into sites while reducing the whole execution time of all the tasks in the bag with distributed input data and provenance support. The scheduling process is performed at the beginning of the execution of each activity when the tasks are generated and to be scheduled at each site.

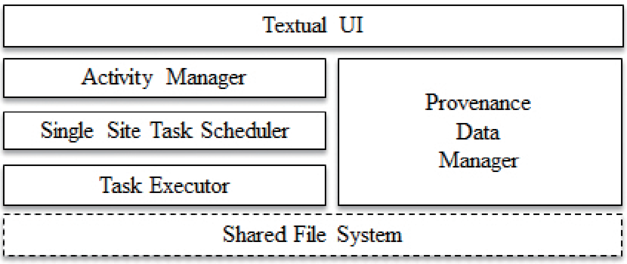


Figure 6.3: **Layered Architecture of Single Site Chiron.** Dashed box, *i.e.* shared file system, represents that the module exploits an external system, *e.g.* NFS, to realize the function.

6.4 System Design

Chiron [139] is a SWfMS for the execution of data-intensive SWfs at a single site, with provenance support. We adapt Chiron to a multisite cloud environment. In this section, we present the system architecture of single site Chiron and propose the adapted multisite Chiron.

6.4.1 Single Site Chiron

At a single site, Chiron takes one computing node as master node and the other nodes as slave nodes, as shown in Figure 6.2. In a cloud environment, a computing node is a VM. Designed for HPC environments, Chiron relies on a Shared File System [2], *e.g.* Network File System (NFS) [160], for managing data. All the computing nodes in the cluster can read or write the data stored in the shared file system. Chiron exploits a relational database, *e.g.* PosgreSQL, to store provenance data.

The layered architecture of single site Chiron is illustrated in Figure 6.3. There are six modules, *i.e.* textual UI, activity manager, single site task scheduler, task executor, provenance data manager and shared file system. As for the SWfMS functional architecture presented in Chapter 2, textual UI corresponds to the presentation layer; provenance data manager corresponds to the user services layer; activity manager and single site task scheduler correspond to the WEP generation layer; task executor is at the WEP execution layer and shared file system is at the infrastructure layer. The users can use a textual User Interface (UI) to interact with Chiron, in order to start an instance of Chiron at each computing node. During the execution of a SWf, each activity and its dependencies are analyzed by the activity manager to find executable activities, *i.e.* unexecuted activities, of which the input data is ready. In order to execute an activity, the corresponding tasks are generated by the activity manager. Afterwards, the task scheduler schedules each task to a computing node. Then, the task execution module at each computing node executes

[1] Configured for the quota of resources that can be used by a cloud user.

[2] In a shared file system, all the computing nodes of the cluster share some data storage that are generally remotely located [119].

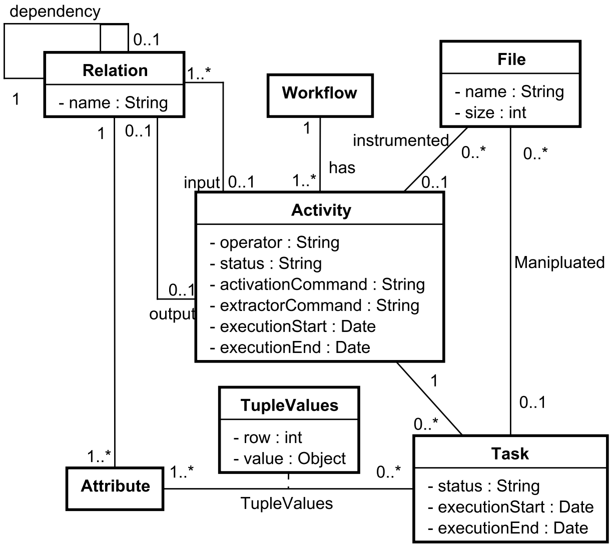


Figure 6.4: **Single Site Provenance Model [139].**

the corresponding scheduled tasks. When all the tasks of the executable activity are executed, the activity manager analyzes the activities to find new executable activities to execute. The process of activity analysis, task scheduling and task execution are repeated until all activities have been executed. Since the input data, intermediate data and output data of SWfs are stored in a shared file system, Chiron does not need to manage data transfer between different computing nodes. During SWf execution, the activity manager, the task scheduler and the task executor generate provenance data, which is gathered by the provenance data manager. The provenance data manager is located at the master node of the cluster.

The single site provenance model [139] is shown in Figure 6.4. In this model, a SWf is composed of several activities. An activity has an operator, *i.e.* the program for this activity. The status of the activity can be ready, running or finished. The $activationCommand$ of an activity is to execute the activity. The $extractorCommand$ is to generate provenance data for the corresponding tasks. The time at which the activity execution starts is $executionStart$ and the time at which it ends is $executionEnd$. One activity is related to several tasks, input relations and output relations. One relation is the input or output parameters for the activity. Each relation has its own attributes and tuples. The tasks of an activity are generated based on the input relation of the activity. A task processes the files associated with the corresponding activity. Each task has a status, *i.e.* ready, running or finished. In addition, the start time and end time of its execution is recorded as $ExecutionStart$ and $ExecutionEnd$. During execution, the corresponding information of activities, files and tasks are stored as provenance data.

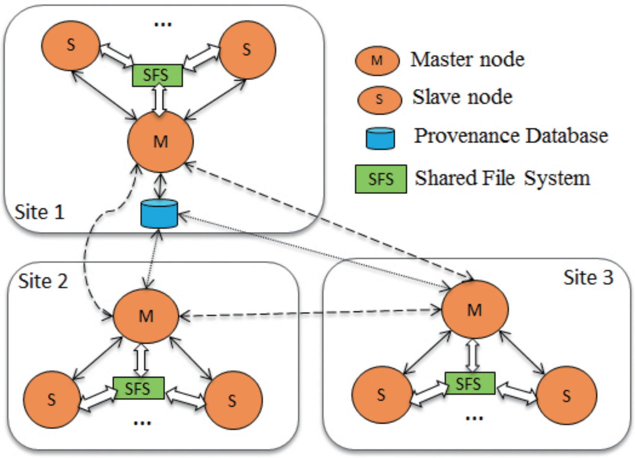


Figure 6.5: **Architecture of multisite Chiron.**

6.4.2 Multisite Chiron

In this section, we present the distributed architecture of multisite Chiron, with the modifications to adapt the single site Chiron to a multisite Cloud.

In the execution environment of multisite Chiron, there is a master site (site 1 in Figure 6.5) and several slave sites (Sites 2 and 3 in Figure 6.5). The master site is similar to the execution environment of a single site Chiron with computing nodes, shared file system and a provenance database. Moreover, a queuing service (see Section 6.6.2) is deployed at the master site. A slave site is composed of a cluster of VMs with a deployed shared file system. In addition, the master node of each site is configured to open the corresponding endpoints to enable the message communication and data transfer with other sites. In the cloud, an endpoint is a data communication tunnel, which maps a public port of a Web domain to a private port of a computing node within the Web domain. In this chapter, we assume that there is a Web domain at a cloud site, which is used for SWf execution and that all the resources related to SWf execution are in the Web domain at that site. A public port is accessible to all the devices on the Internet. A private port can only be recognized by the devices within the same Web domain. The endpoints can be configured by a user of the cloud.

The layered architecture of multisite Chiron is depicted in Figure 6.6. The textual UI is present at each node of each site to start an instance of Chiron. The activity manager is located at the master node of the master site to analyze the activities to find executable activities. The multisite task scheduler is also located at the master node of the master site, which schedules the tasks to be executed. The provenance data manager works at the master node of each site to gather provenance data for the tasks executed at each site and updates the provenance data in the provenance database. The task executor is present at each node of each site to execute tasks. The shared file system is deployed at the master node of each site and is accessible to all the nodes of the same site. The multisite file transfer and multisite message communication (corresponding to the infrastructure layer

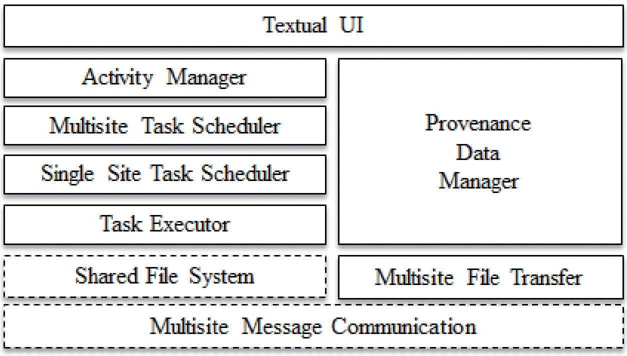


Figure 6.6: **Multisite layered Architecture.**

presented in Chapter 2) work at the master node of each site to enable the communication of different sites.

In order to extend Chiron to a multisite environment, three key aspects, i.e. provenance model, multisite communication and multisite scheduling, must be considered. First, we adapt the provenance model to the multisite environment. As shown in Figure 6.7, we add the information about site and computing node (VM) into the provenance model. A site has its own public IP address, public ports for the communication with other sites, number of virtual CPUs, bandwidth to transfer data to the provenance database and bandwidth to transfer data to other sites. A site can contain several VMs. Each VM has its private IP address (which can only be recognized by the devices deployed in the same Web domain), the type of VM, and the number of virtual CPUs. The type of a VM is configured by a cloud user. In a multisite environment, the provenance database is located at a master site. Since one task is executed at one computing node of a specific site, a task is related to one computing node and one site. A file can be stored at several sites. Since the input data of a task may be stored at one site (site s_1) and processed at another site (site s_2), it is transferred from s_1 to s_2 before being processed. As a result, the data ends up being stored at the two sites. Thus, one file is related to several sites. In addition, the provenance data can provide data location information for the scheduling process. Thus, users can also get execution information, *i.e.* which task is executed at which site, from the provenance database. The other objects and relationships remain the same as in the single site provenance model.

Second, to support communication between different sites, we add two modules, *i.e.* multisite message communication module and multisite file transfer module. The multisite message communication module is responsible for the exchange of control messages among different sites. The control messages are generated for synchronizing the execution of each site and sharing information among different sites. The multisite file transfer module transfers files to be processed by a task from the site where the files are stored to the site where the task is executed. The implementation techniques of the two modules are detailed in Section 6.6.2.

Third, we provide a multisite task scheduling module in multisite Chiron, which is

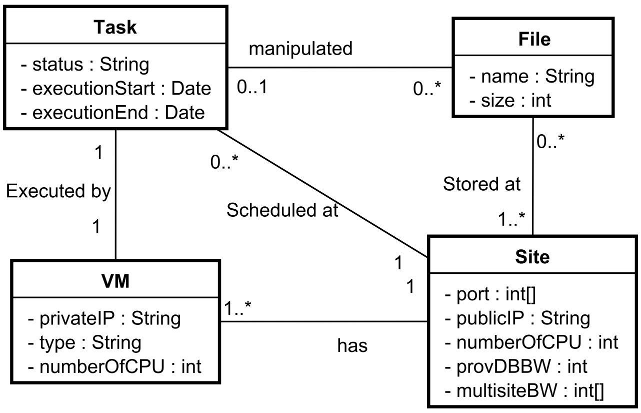


Figure 6.7: **Multisite Provenance Model.**

detailed in Section 6.5.2.

6.5 Task Scheduling

In this section, we present how single site task scheduling is done in Chiron and propose a multisite task scheduling algorithm, *i.e.* Data-Intensive Multisite task scheduling (DIM). Then, we analyze the complexity of the DIM algorithm. Finally, we present the method to estimate the execution time of a bag of tasks at a single site cloud, which is used in the DIM algorithm.

6.5.1 Single Site Task Scheduling

Currently, task scheduling in the single site implementation of Chiron is carried out in a simple way. Tasks are generated and published to a centralized provenance database. Each time a slave node is available, it requests new tasks from the master node, which in turn searches for unexecuted tasks and dispatches them to the slave. This approach is efficient for single site implementations, where communication latency is negligible and there exists an underlying shared file system. However, in multisite environments, scheduling has to be carefully performed to avoid costly and unnecessary intersite data transfers and to achieve good load balancing among sites. In a multisite environment, the different data transfer rates between different sites and computing resources should be considered to generate a good scheduling plan.

6.5.2 Multisite Task Scheduling

In this section, we propose our mutlisite scheduling algorithm, *i.e.* DIM. Multisite task scheduling is done with a two Level (2L) approach, which is shown in Figure 6.8. The first level performs multisite scheduling, where each task is scheduled to a site. Then, the second level performs single site scheduling, where each task is scheduled to a computing

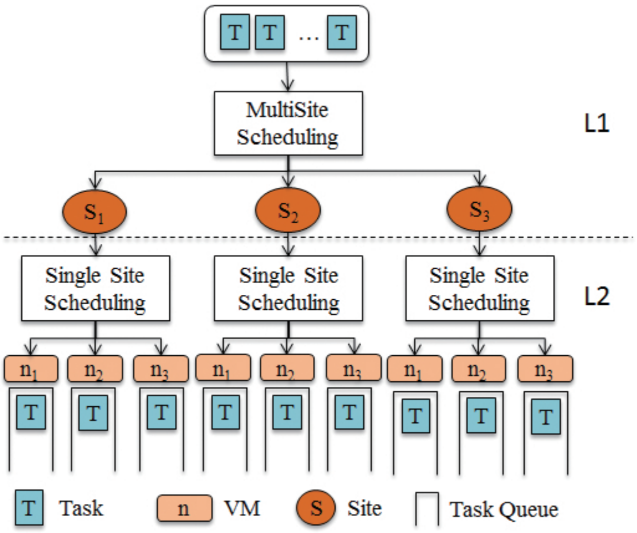


Figure 6.8: **MultiSite Scheduling.** The master node at the master site schedules tasks to each site. At each site, the master node schedules tasks to slave nodes.

node of the site. Compared with a one Level (1L) approach that schedules tasks directly to computing nodes at different cloud sites, this 2L approach may well reduce the multisite scheduling complexity. For instance, let us schedule N ($N >> 2$) tasks to M ($M > 2$) sites, each of which has K ($K > 2$) computing nodes. The complexity of the 2L approach is $M^N + K^N$, where M^N is the complexity of the multisite level and K^N is the complexity of the single site level. Assume that there are N_i tasks scheduled at site s_i while $\sum_{i=1}^{M} N_i = N$. Thus, the complexity of single site scheduling is:

$$\begin{aligned}\prod_{i=1}^{M} K^{N_i} &= K^{\sum_{i=1}^{M} N_i} \\ &= K^N\end{aligned} \tag{6.1}$$

Thus, the complexity of the single site scheduling of 2L approach is K^N. However, the complexity of the 1L approach is $(M * K)^N$. Let us assume that $N > 2, M > 2$ and $K > 2$.

$$\begin{aligned}M^N + K^N &< (\frac{1}{2} * M * K)^N + (\frac{1}{2} * M * K)^N \\ &< (\frac{1}{2})^{(N-1)} * (M * K)^N \\ &< (M * K)^N\end{aligned} \tag{6.2}$$

From Formula 6.2, we can conclude that $N^M + N^K < N^{M*K}$, *i.e.* the complexity of 2L scheduling approach is smaller than that of 1L scheduling approach. In addition, the 2L scheduling approach can exploit the existing scheduling solutions of single site

```
Input: $T$: a bag of tasks to be scheduled; $S$: a set of cloud sites
Output: $SP$: the scheduling plan for $T$ in $S$
 1: $SP \leftarrow \emptyset$
 2: for each $t \in T$ do
 3:     $s \leftarrow GetDataSite(t)$
 4:     $SP \leftarrow SP \cup \{Schedule(t, s)\}$
 5:     EstimateTime( $T$, $s$, $SP$ )
 6: end for
 7: while MaxunbalanceTime( $T$, $s$, $SP$ ) can be reduced do
 8:     $sMin < -MinTime(S)$
 9:     $sMax < -MaxTime(S)$
10:     ExchangeTasks($sMin$, $sMax$, $SP$)
11: end while
end
```

algorithm 6: Data-Intensive Multisite task scheduling (DIM)

SWfMSs. In this chapter, we focus on the multisite scheduling part, since we use the default scheduling solutions of Chiron for single site scheduling.

In our layered architecture (see Section 6.4.2), the multisite scheduling is performed at the master node of the master site. For the tasks of data-intensive SWfs, the time to transfer task input data and the time to generate provenance data should not be ignored, in particular in case of low bandwidth of intersite connection and big amounts of data in the files to be transferred between different sites. This is why we consider the time to transfer task input data and provenance data in the scheduling process. The method to estimate the execution time of a bag of tasks at a single site is detailed in Section 6.5.4. In addition, during the scheduling, if the data cannot be moved, the associated tasks are scheduled at the site where the data is stored.

The DIM algorithm schedules a bag of tasks onto multiple sites (see Algorithm 6). First, the tasks are scheduled according to the location of input data (Lines 2-5), which is similar to the scheduling algorithm of MapReduce [55]. Line 3 searches the site that stores the biggest part of input data corresponding to Task t. Line 4 schedules Task t at Site s. Line 5 estimates the execution time of all the tasks scheduled at Site s according to Formula 6.4. Then, the execution time at each site is balanced by adjusting the whole bag of tasks scheduled at that site (Lines 6-9). Line 6 checks if the maximum difference of the estimated execution time of tasks at each site can be reduced by verifying if the difference is reduced in the previous loop or if this is the first loop. While the maximum difference of execution time can be reduced, the tasks of the two sites are exchanged as described in Lines 7-9. Line 7 and 8 choose the site that has the minimal execution time and the site that has the maximum execution time, respectively. Then, the scheduler calls the function $ExchangeTasks$ to exchange the tasks scheduled at the two selected sites to reduce the maximum difference of execution time.

In order to achieve load balancing of two sites, we propose $ExchangeTasks$ algo-

```
Input: $s_i$: a site that has bigger execution time for its scheduled tasks; $s_j$: a site that has
    smaller execution time for its scheduled tasks; $SP$: original scheduling plan for a
    bag of tasks $T$
Output: $SP$: modified scheduling plan
 1: $Diff \leftarrow CalculateExecTimeDiff(s_i, s_j, SP)$
 2: $T_i \leftarrow GetScheduledTasks(s_i, SP)$
 3: for each $t \in T_i$ do
 4:     $SP' \leftarrow ModifySchedule(SP, \{Schedule(t, s_j)\}$
 5:     $Diff' \leftarrow CalculateExecTimeDiff(s_i, s_j, SP')$
 6:     if $Diff' < Diff$ then
 7:         $SP \leftarrow SP'$
 8:         $Diff \leftarrow Diff'$
 9:     end if
10: end for
end
```

algorithm 7: Exchange Tasks

rithm. Let us assume that there are two sites, *i.e.* Sites s_i and s_j. For the tasks scheduled at each site, we assume that the execution time of Site s_i is bigger than Site s_j. In order to balance the execution time at Sites s_i and s_j, some of the tasks scheduled at Site s_i should be rescheduled at Site s_j. Algorithm 7 gives the method to reschedule a bag of tasks from Site s_i to Site s_j in order to balance the load between the two sites. Line 1 calculates the difference of the execution time of two sites according to Formula 6.4 with a scheduling plan. Line 2 gets all the tasks scheduled at Site s_i. For each Task t in T_i (Line 3), it is rescheduled at Site s_j if the difference of execution time of the two sites can be reduced (Lines 4-8). Line 4 reschedules Task t at Site s_j. Line 5 calculates the execution time at Sites s_i and s_j. Lines 6-7 updates the scheduling plan if it can reduce the difference of execution time of the two sites by rescheduling Task t.

6.5.3 Complexity

In this section, we analyze the complexity of the DIM algorithm. Let us assume that we have n tasks to be scheduled at m sites. The complexity of the first loop (lines 2-5) of the DIM algorithm is $\mathcal{O}(n)$. The complexity of the $ExchangeTasks$ algorithm is $\mathcal{O}(n)$, since there may be n tasks scheduled at a site in the first loop (lines 2-5) of the DIM algorithm. Assume that the difference between the maximum execution time and the minimum execution time is T_{diff}. The maximum value of T_{diff} can be $n * avg(T)$ when all the tasks are scheduled at one site while there is no task scheduled at other sites. $avg(T)$ represents the average execution time of each task, which is a constant value. After m times of exchanging tasks between the site of maximum execution time and the site of minimum execution time, the maximum difference of execution time of any two sites should be reduced to less than $\frac{T_{diff}}{2}$. Thus, the complexity of the second loop (lines

6-9) of the DIM algorithm is $\mathcal{O}(m \cdot n \cdot \log n)$. Therefore, the complexity of the DIM algorithm is $\mathcal{O}(m \cdot n \cdot \log n)$. It is only a little bit higher than that of OLB and MCT, which is $\mathcal{O}(m \cdot n)$, but yields high reduction in SWf execution (see Section 6.6.3).

6.5.4 Execution Time Estimation

We now give the method to estimate the execution time of a bag of tasks at a single site, which is used in both the DIM algorithm and the MCT algorithm. Formula 6.3 gives the estimation of execution time without considering the time to generate provenance data, which is used in the MCT algorithm.

$$\begin{aligned} TotalTime(T, s) = & ExecTime(T, s) \\ & + InputTransTime(T, s) \end{aligned} \tag{6.3}$$

T represents the bag of tasks scheduled at site s. $ExecTime$ is the time to execute the bag of tasks T at site s, *i.e.* the time to run the corresponding programs. $InputTransTime$ is the time to transfer the input data of the tasks from other sites to site s. In the DIM algorithm, we use Formula 6.4 to estimate the execution time with the consideration of the time to generate provenance data.

$$\begin{aligned} TotalTime(T, s) = & ExecTime(T, s) \\ & + InputTransTime(T, s) \\ & + ProvTransTime(T, s) \end{aligned} \tag{6.4}$$

$ProvTransTime$ is the time to generate provenance data in the provenance database.

We assume that the workload of each task of the same activity is similar. The average workload (in GFLOP) of the tasks of each activity and the computing capacity of each VM at Site s is known to the system. The computing capacity (in GFLOPS) indicates the workload that can be realized per second. Then, the time to execute the tasks can be estimated by dividing the total workload by the total computing capacity of Site s, as shown in Formula 6.5.

$$ExecTime(T, s) = \frac{|T| * AvgWorkload(T)}{\sum_{VM_i \in s} ComputingCapacity(VM_i)} \tag{6.5}$$

$|T|$ represents the number of tasks in Bag T. $AvgWorkload$ is the average workload of the bag of tasks.

The time to transfer input data can be estimated as the sum of the time to transfer the input data from other sites to Site s as shown in Formula 6.6.

$$InTransTime(T, s) = \sum_{t_i \in T} \sum_{s_i \in S} \frac{InDataSize(t_i, s_i)}{DataRate(s_i, s)} \tag{6.6}$$

$InDataSize(t_i, s_i)$ represents the size of input data of Task t_i, which is stored at Site s_i.

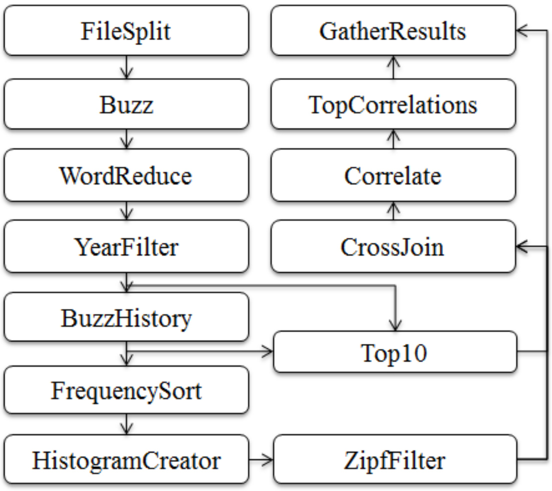


Figure 6.9: **Buzz Workflow.**

The size can be measured at runtime. $DataRate(s_i, s)$ represents the data transfer rate, which can be configured by users. S represents the set of sites.

Finally, the time to generate provenance data is estimated by Formula 6.7.

$$\begin{aligned} ProvTransTime(T, s) = & |T| * TransctionTimes(T) \\ & * AvgTransactionTime(s) \end{aligned} \tag{6.7}$$

$|T|$ represents the number of tasks in Bag T. We can estimate $AvgTransactionTime$ by counting the time to perform a data exchange to update the provenance data of a task in the provenance database from Site s. $TransctionTimes(T)$ represents the number of data exchanges to perform for generating the provenance data of each task in Bag T. It can be configured according to the features of the SWfMS.

6.6 Experimental Evaluation

In this section, we present an experimental evaluation of our DIM scheduling algorithm using Microsoft Azure multisite cloud [5]. First, we present two real-life SWfs, *i.e.* Buzz and Montage, as use cases. Then, we explain the techniques for the implementation of intersite communication of multisite Chiron in Azure. Afterwards, we show the experimental results of executing the two SWfs in Azure with different multisite scheduling algorithms.

6.6.1 SWf Use Cases

In this section, we present two SWfs, *i.e.* Buzz and Montage, to evaluate our proposed algorithms. The two SWfs have different structures, which can show that our proposed algorithm is suitable for different SWfs.

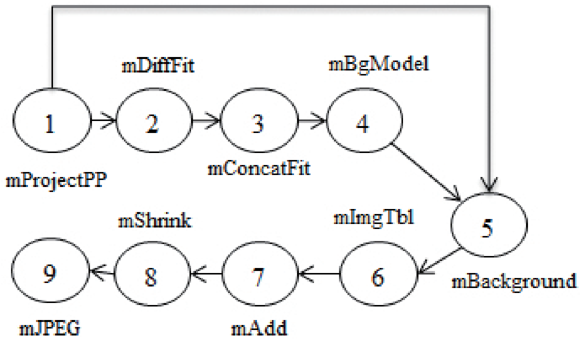


Figure 6.10: **Montage Workflow.**

6.6.1.1 Buzz Workflow

Buzz workflow (see Section 3.4 for details) is a data-intensive SWf that searches for trends and measures correlations in scientific publications as shown in Figure 6.9.

There are five activities, *i.e.* FileSplit, Buzz, BuzzHistory, HistogramCreator and Correlate, that correspond to multiple tasks. In our experiment, the tasks of the five activities are scheduled by the multisite scheduling algorithm. The other activities exploit a database management system to process data at the master site.

6.6.1.2 Montage Workflow

As presented in Section 2.2.1.4, Montage is a data-intensive SWf for computing mosaics of input images [59]. The input data and the intermediate data are of considerable size and require significant storage resources. However, the execution time of each task is relatively small, which can be at most a few minutes. The structure (at activity level) of the Montage SWf is shown in Figure 6.10. Activity 1, mProjectPP, reprojects single images to a specific scale. The mDiffFit activity performs a simple image difference between a single pair of overlapping images, which is generate by the mProjectPP activity. Then, the mConcatFit activity gathers the results of mDiffFit into a single file. Afterwards, mBgModel uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a “best” global fit. The mBackground activity removes a background from a single image. This activity takes the output data of the mProjectPP activity and that of the mBgModel activity. The mImgTbl activity prepares the information for putting the images together. The mAdd activity generates an output mosaic and the binning of the mosaic is changed by the mShrink activity. Finally, the mJPEG activity creates a JPEG image from the mosaic. In addition, Montage can correspond to different square degrees [59] (or degree for short), each of which corresponds to a different number of tasks.

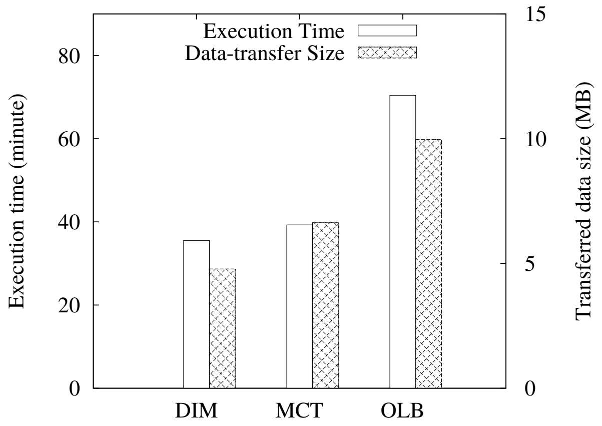


Figure 6.11: **Buzz SWf Execution time.** The amount of data is 60MB.

6.6.2 Intersite Communication

In this section, we present the detailed techniques for the multisite file transfer module and multisite message communication module. We choose Azure Service Bus [2] to realize the functionality of message communication module. Azure Service Bus is a generic, cloud-based messaging system for the communication among different devices. The communication can be based on the HTTP protocol, which does not need to maintain connection information (HTTP is stateless). Although this may bring more overhead for each message, the amount of control messages is low and this cost is negligible. The file transfer module is realized by Java TCP connections between two master nodes of two different sites. Since the idle intersite TCP connections may be cut down by the cloud operator, *e.g.* every $5 - 10$ minutes in Azure, the connections are maintained by sending $keepalive$ messages. For instance, two messages per time period. Before execution, a task is scheduled at a site by the multisite task scheduler. If they are not stored at the scheduled site, the input files of the task are transferred to the scheduled site by the multisite file transfer module.

6.6.3 Experiments

This section gives our experimental evaluation of the DIM algorithm, within Microsoft Azure. Azure [5] has multiple cloud sites, *e.g.* Central US (CUS), West Europe (WEU) and North Europe (NEU). We instantiated three A4 (the type of VMs in Azure [8]) VMs at each of the three site, *i.e.* CUS, WEU and NEU. We take WEU as a master site. We deploy an A2 VM at WEU and install PostgreSQL database to manage provenance data. We assume that the input data of the SWfs are distributed at the three sites. We compare our proposed algorithm with two representative baseline scheduling algorithms, *i.e.* Opportunistic Load Balancing (OLB) and Minimum Completion Time (MCT). In the multisite environment, OLB randomly selects a site for a task while MCT schedules a task to the site that can finish the execution first.

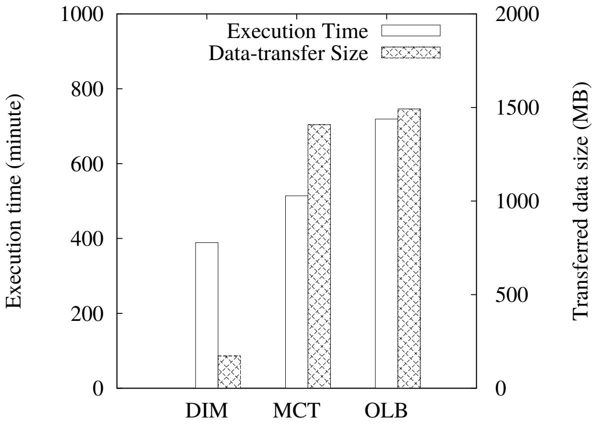


Figure 6.12: **Buzz SWf Execution time.** The amount of data is 1.29GB.

First, we used a DBLP 2013 XML file of 60MB as input data for Buzz SWf in our experiments. The input data is partitioned into three parts, which have almost the same amount of data, and each part is stored at a site while configuration files of Buzz SWf are present at all the three sites. We take WE as a master site. The provenance database and Azure Service Bus are also located at the WE site. The execution result corresponding to each scheduling algorithm is shown in Figure 6.11. During scheduling, if the data cannot be moved (for the start activity, *i.e.* FileSplit), the associated task is scheduled at the site where the data is stored.

Figure 6.11 shows that DIM is much better than MCT and OLB in terms of both execution time and transferred data size. The execution time corresponding to DIM is 9.6% smaller than that corresponding to MCT and 49.6% smaller than that corresponding to OLB. The size of the data transferred between different sites corresponding to MCT is 38.7% bigger than that corresponding to DIM and the size corresponding to OLB is 108.6% bigger than that corresponding to DIM.

Second, we performed an experiment using a DBLP 2013 XML file of 1.29GB as input data for Buzz SWf while configuration files of Buzz SWf are present at all the three sites. The other configuration is the same as the first one. The execution results are shown in Figure 6.12.

Figure 6.12 shows that the advantage of DIM in terms of both execution time and transferred data size compared with MCT and OLB increases with bigger amounts of input data. The execution time corresponding to DIM is 24.3% smaller than that corresponding to MCT and 45.9% smaller than that corresponding to OLB. The size of the data transferred between different sites corresponding to MCT is 7.19 times bigger than that corresponding to DIM and the size corresponding to OLB is 7.67 times bigger than that corresponding to DIM.

Since the DIM algorithm considers the time to transfer intersite provenance data and makes optimization for a bag of tasks, it can reduce the data transferred between different sites and the total execution time. MCT only optimizes the load balancing for each task

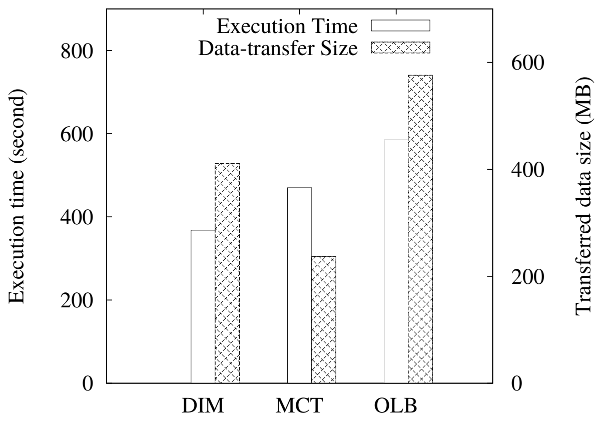


Figure 6.13: **Montage SWf Execution time.** 0.5 degree.

among different sites without consideration of the time to transfer intersite provenance data. It is a greedy algorithm that can reduce the execution time by balancing the execution time of each site while scheduling each task. However, it cannot optimize the scheduling for the whole execution of all the tasks of an activity. In addition, compared with OLB, MCT cannot reduce much the transferred data among different sites. Since OLB simply tries to keep all the sites working on arbitrary tasks, it has the worst performance.

Furthermore, we executed the Montage SWf with 0.5 degree with three sites, *i.e.* CUS, WEU and NEU. The size of input data is 5.5GB. The input data is evenly partitioned to three parts stored at the corresponding sites with configuration files stored at all the three sites. The execution time and amount of intersite transferred data corresponding to each scheduling algorithm are shown in Figure 6.13.

The execution results of Montage with 0.5 degree reveals that the execution time of DIM is 21.7% smaller than that of MCT and 37.1% smaller than that of OLB. This is expected since DIM makes optimization for a bag of tasks in order to reduce intersite transferred data with consideration of the time to transfer intersite intermediate data and provenance data. MCT is optimized for load balancing only with consideration of intermediate data. OLB has no optimization for load balancing. In addition, the intersite transferred data of DIM is 42.3% bigger than that of MCT. Since DIM is designed to achieve load balancing of each site to reduce execution time, it may yield more intersite transferred data in order to achieve load balance. However, the amount of intersite transferred data of DIM is 28.6% smaller than that of OLB. This shows the efficiency of the optimization for the data transfer of DIM. Moreover, when the degree (0.5) is low, there is less data to be processed by Montage, and the number of tasks to schedule is small. Since DIM is designed for high numbers of tasks, the amounts of intersite transferred data are not reduced very much in this situation.

Finally, we executed Montage SWf of 1 degree in the multisite cloud. We used the same input data as in the previous experiment, *i.e.* 5.5GB input data evenly distributed at

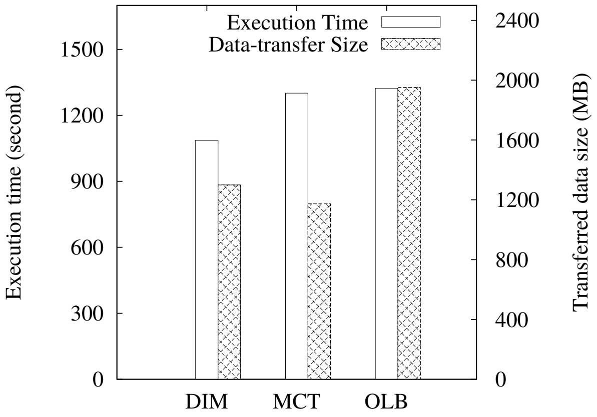


Figure 6.14: **Montage SWf Execution time.** 1 degree.

three sites. The execution time and the amount of intersite transferred data corresponding to each scheduling algorithm are shown in Figure 6.14.

The execution results of Montage with 1 degree reveals that the execution time of DIM is 16.4% smaller than that of MCT and 17.8% smaller than that of OLB. As explained before, this is expected since DIM can reduce the execution time by balancing the load among different sites compared with MCT and OLB. In addition, the intersite transferred data of DIM is 10.7% bigger than that of MCT. This is much smaller than the value for 0.5 degree (42.3%), since there are more tasks to schedule when the degree is 1 and DIM reduces intersite transferred data for a big amount of tasks. However, the amount of intersite transferred data is bigger than that of MCT. This happens since the main objective of DIM is to reduce execution time instead of reducing intersite transferred data. In addition, the amount of intersite transferred data of DIM is 33.4% smaller than that of OLB, which shows the efficiency of the optimization for the data transfer of DIM.

Table 6.1: **Scheduling Time.** The unit of time is second. The size of the input data of Buzz SWf is 1.2GB and the degree of Montage is 1^3.

Algorithm	DIM	MCT	OLB
Buzz	633	109	17
Montage	29.2	28.8	1.5

In addition, we measured the time to execute the scheduling algorithms to generate scheduling plans while executing Buzz and Montage. The scheduling time is shown in Table 6.1. The complexity of MCT is the same as that of OLB, which is $\mathcal{O}(m \cdot n)$. However, the scheduling time of MCT is much bigger than OLB. The reason is that MCT needs to interact with the provenance database to get the information of the files in order to estimate the time to transfer the files among different sites. The table shows that the time to execute DIM is much higher than OLB for both Buzz and Montage since the complexity of DIM is higher than that of OLB and that DIM has more interactions with

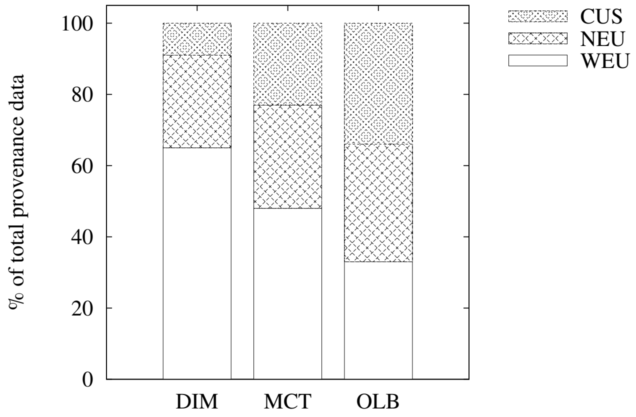


Figure 6.15: **Distribution of provenance during the execution of Buzz workflow.** The size of input data is 1.2GB.

the provenance database in order to estimate the execution time of the tasks at a site. When there are significant number of tasks to schedule (for the Buzz SWf), the time to execute DIM is much bigger than that of MCT because of higher complexity. However, when the number of tasks is not very big, the time to execute DIM is similar to that of MCT. The scheduling time of DIM and MCT is much bigger than that of OLB, since it takes much time to communicate with the provenance database for the estimation of the execution time of each site. The scheduling time of the three scheduling algorithms is always small compared with the total execution (less than 3%), which is acceptable for the task scheduling during SWf execution. Although the scheduling time of DIM is much bigger than MCT and OLB, the total execution time of SWfs corresponds to DIM is much smaller than that of MCT and OLB as explained in the four experiments. This means that DIM generates better scheduling plans compared with MCT and OLB.

Table 6.2: **Size of Provenance Data.** The unit of the data is MB. The size of the input data of Buzz SWf is 1.2GB and the degree of Montage is 1.

Algorithm	DIM	MCT	OLB
Buzz	301	280	279
Montage	10	10	10

Furthermore, we measured the size of provenance data and the distribution of the provenance data. As shown in Table 6.2, the amount of the provenance data corresponding to the three scheduling algorithms are similar (the difference is less than 8%). However, the distribution of the provenance data is different. In fact, the bandwidth between the

[3] The advantage of DIM over MCT and OLB is more obvious when the input data of Buzz is 1.2GB and the degree of Montage is 1 compared with the other cases in our experiments, *i.e.* when the input data of Buzz is 60MB and the degree of Montage is 0.5.

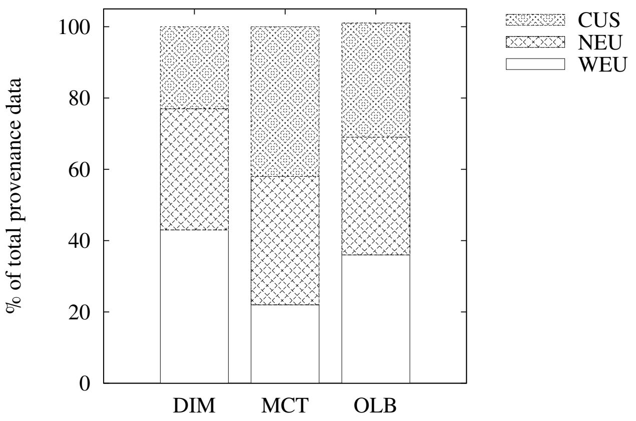


Figure 6.16: **Distribution of provenance during the execution of Montage workflow.** The degree is 1.

provenance database and the site is in the following order: WEU > NEU > CUS [4]. As shown in Figures 6.15 and 6.16, the provenance data generated at CUS site is much more than that generated at NEU site and WEU site for DIM algorithm. In addition, the percentage of provenance data at WEU corresponding to DIM is much bigger than MCT (up to 95% bigger) and OLB (up to 97% bigger). This indicates that DIM can schedule tasks to the site (WEU) that has bigger bandwidth with the provenance database (the database is at WEU site), which yields bigger percentage of provenance data generated at the site. This can reduce the time to generate provenance data in order to reduce the overall multisite execution time of SWfs. However, MCT and OLB is not sensitive to the centralized provenance data, which correspond to bigger multisite execution time.

From the experiments, we can see that DIM performs better than MCT (up to 24.3%) and OLB (up to 49.6%) in terms of execution time although it takes more time to generate scheduling plans. The scheduling time of MCT is always reasonable compared with the total SWf execution time (less than 3%). DIM can reduce the intersite transferred data compared with MCT (up to 719%) and OLB (up to 767%). As the amount of input data increases, the advantage of DIM becomes more important.

6.7 Conclusion

Although some SWfMSs with provenance support, *e.g.* Chiron, have been deployed in the cloud, they are generally designed for a single site. In this chapter, we proposed a solution based on multisite Chiron.

Multisite Chiron is able to execute SWfs in a multisite cloud with geographically distributed input data. We proposed the architecture of multisite Chiron, defined a new

[4] For instance, the time to execute "SELECT count(*) from eactivity" at the provenance database from each site: 0.0027s from WEU site, 0.0253s from NEU site and 0.1117s from CUS site.

provenance model for multisite SWf execution and a global method to gather the distributed provenance data in a centralized database. Based on this architecture, we proposed a new scheduling algorithm, *i.e.* DIM, which considers the latency to transfer data and to generate provenance data in multisite cloud. We analyzed the complexity of DIM ($\mathcal{O}(m \cdot n \cdot \log n)$), which is quite acceptable for scheduling bags of tasks. We used two real-life SWfs, *i.e.* Buzz and Montage to evaluate the DIM algorithm in Microsoft Azure with three sites. The experiments show that although its complexity is higher than that of OLB and MCT, DIM is much better than two representative baseline algorithms, *i.e.* MCT (up to 24.3%) and OLB (up to 49.6%), in terms of execution time. In addition, DIM can also reduce significantly data transfer between sites, compared with MCT (up to 719%) and OLB (up to 767%). Moreover, the scheduling time of MCT is always reasonable compared with the total SWf execution time (less than 3%). The advantage of DIM becomes important with high numbers of tasks.

Chapter 7

Conclusions

In this thesis, we addressed the problem of executing data-intensive SWfs in a multisite cloud, where the data and computing resources may be distributed in different cloud sites. To this end, we proposed a distributed and parallel approach that leverages the resources available at different cloud sites, based on a survey of existing techniques. In the survey, we proposed a functional SWfMS architecture by analyzing and categorizing the existing techniques. To exploit parallelism, we used an algebraic approach, which allows expressing SWf activities using operators and automatically transforming them into multiple tasks. We proposed SWf partitioning algorithms, a dynamic VM provisioning algorithm, an activity scheduling algorithm and a task scheduling algorithm. Different SWf partitioning algorithms partition a SWf to several fragments. The VM provisioning algorithm is used to dynamically create an optimal combination of VMs for executing SWf fragments at each cloud site. The activity scheduling algorithm distributes the SWf fragments to the cloud sites with the minimum cost based on a multi-objective cost model. The task scheduling algorithm directly distributes tasks among different cloud sites while achieving load balancing at each site based on a multisite SWfMS. We evaluated our proposed solutions by executing real-life SWfs in the Microsoft Azure cloud, the results of which shown the advantages of our solutions over the existing techniques. In this chapter, we summarize and discuss the contributions made in this thesis. Then, we give some research directions for future work.

7.1 Contributions

A survey of existing techniques for SWfs execution.

We discussed the existing techniques for parallel execution of data-intensive SWfs in different infrastructures, especially in the cloud. First, we introduced the definitions in SWf management, including SWfs and SWfMSs. Then, we presented in more details a five-layer functional architecture of SWfMSs and the corresponding functions. Special attention has been paid to data-intensive SWfs by identifying their features and presenting the corresponding techniques. Second, we presented the basic techniques for the parallel execution of SWfs in SWfMSs: parallelization and scheduling. We showed how different kinds of parallelism (coarse-grained parallelism, data parallelism, independent parallelism and pipeline parallelism) can be exploited for parallelizing SWfs. The scheduling methods to allocate tasks to computing resources can be static or dynamic, with different trade-offs, or hybrid to combine the advantages of static and dynamic scheduling methods. SWf scheduling may include an optimization phase to minimize a multi-objective function, in a given context (cluster, grid, cloud). Third, we discussed cloud computing and the basic techniques for parallel execution of SWfs in the cloud, including single site cloud and multisite cloud. We discussed three categories of cloud computing, multisite management in the cloud and data storage in the cloud. The data storage techniques include shared-disk file systems and distributed file systems. Then, we analyzed the parallelization techniques of SWfs in both single site cloud and multisite cloud. Fourth, to illustrate the use of the techniques, we introduced the recent parallelization frameworks such as MapReduce and gave a comparative analysis of eight popular SWfMSs (Pegasus, Swift, Kepler, Taverna, Chiron, Galaxy, Triana and Askalon) and a science gateway framework (WS-PGRADE/gUSE). Finally, we identified the limitations of existing techniques and proposed some research issues.

SWf Partitioning for the Execution in a Multisite Cloud.

We tackled the problem of SWf partitioning problem in order to execute SWfs in a multisite cloud. Our main objective was to enable SWf execution in a multisite cloud by partitioning SWfs into fragments while ensuring some activities executed at specific cloud sites. First, we presented the general SWf partitioning techniques, *i.e.* data partitioning and DAG partitioning. Then, we focused on DAG partitioning and mentioned activity encapsulation technique. Afterward, we proposed our SWf partitioning methods, namely Scientist Privacy (SPr), Data Transfer Minimization (DTM) and Computing Capacity Adaptation (CCA). SPr partitions SWfs by putting locking activities and its available following activities to a fragment, in order to better support execution monitoring under security restriction. DTM partitions SWfs with the consideration of locking activities while minimizing the volume of data to be transferred among different SWf fragments. CCA partitions SWfs according to the computing capacity of different cloud sites. This technique tries to put more activities to the fragment to be executed within a cloud site with bigger computing capacity. Our proposed partitioning techniques are suitable for different configurations of clouds in order to reduce SWf execution time. In addition, we also proposed to use data refining techniques, namely, file combining and data compression, to reduce the time to transfer data among different sites.

We evaluated extensively our proposed partitioning techniques by executing the Buzz SWf at two sites, *i.e.* Western Europe and Eastern US, of the Azure cloud with different configurations. All the sites have the same amounts and types of VMs correspond to the homogeneous configuration while the sites have different amounts or types of VMs correspond to the heterogeneous configuration. The experimental results show that DTM with data refining techniques is suitable (24.1% of time saved compared to CCA without data refining) for executing SWfs in a multisite cloud with a homogeneous configuration, and that CCA performs better (28.4% of time saved compared to SPr technique without data refining) with a heterogeneous configuration. In addition, the results also reveal that data refining techniques can significantly reduce the time to transfer data between two different sites.

VM Provisioning in a Single Site Cloud.

We handled the problem of generating VM provisioning plans for SWf execution within a single cloud site for multiple objectives. Our main contribution was to propose a cost model and an algorithm in order to generate VM provisioning plans to reduce both execution time and monetary cost for SWf execution in a single site cloud. To address the problem, we designed a multi-objective cost model for the execution of SWfs within a single cloud site. The cost model is a weighted function with the objectives of reducing execution time and monetary cost. Our cost model takes the sequential workload and the cost to start VMs into consideration, which is more precise compared with existing cost models. Then, based on the cost model, we proposed Single Site VM Provisioning (SSVP) algorithm to generate provisioning plans for SWf execution within a single cloud site. The SSVP first calculates an optimal number of CPU cores for SWf execution. Then, it generates a provisioning plan and iteratively improves the provisioning plan in order to reduce the cost based on the cost model and the optimal number of CPU cores.

We made extensive evaluations to compare our cost model and algorithm with an existing approach, *i.e.* GraspCC. We executed SciEvol with different amounts of input data and different weights of execution time at the Japan East site of Azure cloud. The experimental results show that our algorithm can adapt VM provisioning plans to different configurations, *i.e.* different weights of execution time and generate better (53.6%) VM provisioning plans compared with GraspCC. The results also reveal that our cost model is more (76.7%) precise to estimate the execution time and the monetary cost compared with the existing approach.

Multi-Objective SWf Fragment Scheduling in a Multisite cloud.

We addressed the problem of SWf fragment scheduling for multiple objectives in order to enable SWf execution in a multisite cloud with a stored data constraint. In this work, we took into consideration of different prices to use VMs and stored data at different sites. We formally defined the scheduling problem of executing SWfs in a multisite cloud for multiple objectives with the stored data constraint. Then, we presented the system model for multisite SWf execution. Afterward, we detailed our multi-objective cost model composed of a weighted function with two objectives, *i.e* reducing execution time and monetary cost. In addition, the cost model considers different costs of using VMs at different cloud sites. Finally, we presented two adapted scheduling algorithms, *i.e.* data location based scheduling (LocBased) and site greedy scheduling (SGreedy), and our proposed algorithm, namely activity greedy scheduling (ActGreedy). LocBased exploits DTM to partition SWfs and schedules the fragments to the site where its input data is stored. This algorithm does not take the monetary cost into consideration and may incur a big cost to execute SWfs. SGreedy takes advantage of the activity encapsulation technique to partition SWfs and schedules the best fragment for each site. SGreedy schedules the activities of a pipeline of activities to different sites, which leads to bigger intersite data transfer and execution time. ActGreedy partitions SWfs with the activity encapsulation technique and groups small fragments to bigger fragments to reduce data transfer among different sites and schedules each fragment to the best site. This algorithm can reduce the overall execution time by comparing the cost to execute fragments at each site, which is generated based on the SSVP algorithm.

We evaluated our scheduling algorithm by executing SciEvol with different amounts of input data and different weights of objectives at three cloud sites of Azure. The three cloud sites are West Europe, Japan West, and Japan East and the costs of using VMs at each site are different. We used SSVP to generate VM provisioning plans to execute SWf fragments at each site. The experimental results shown that ActGreedy performs better in terms of the weighted cost to execute SWfs in a multisite cloud compared with LocBased (up to 10.7%) and SGreedy (up to 17.2%). In addition, the results also reveal that the scheduling time of ActGreedy is reasonable compared with two general approaches.

Multisite Chiron and Multisite Task Scheduling with Provenance Support.

We dealt with task scheduling problem for multisite SWf execution with provenance support. The main goal was to enable SWf execution with the distributed input data at different sites within the minimum time with provenance support while the bandwidths among different sites are different. In this work, we formally defined the task scheduling problem for multisite SWf execution, including the support for provenance data, different bandwidths among different sites and the distribution of input data. Then, we proposed multisite Chiron, which enables scheduling and executing tasks of one activity at different sites with a centralized provenance database. We also detailed the modifications made to adapt single site Chiron to multisite. Then, we proposed our two level scheduling and our intersite task scheduling algorithm, *i.e.* Data-Intensive Multisite task scheduling (DIM). DIM considers the data locality and different bandwidths among different sites while transferring intersite provenance data. DIM also achieves load balance among different sites for the task execution based on an execution time estimation method.

We evaluated our algorithms and multisite Chiron by executing Buzz and Montage in three Azure cloud sites, *i.e.* Central US, West Europe and North Europe. We executed Buzz with different amounts of input data and Montage with different degrees using the multisite Chiron. The experimental results show that DIM performs much better than two existing scheduling algorithms, *i.e.* MCT (up to 24.3%) and OLB (up to 49.6%), in terms of execution time. Moreover, DIM can also reduce significantly (up to more than 7 times) data transfer between sites, compared with MCT and OLB. In addition, the results also reveal that the scheduling time of DIM is reasonable compared with the overall execution time of SWfs (less than 3%). In particular, the experiments show that the distribution of tasks is adapted according to different bandwidths among different sites for the generation of provenance data.

7.2 Directions for Future Work

Our contributions can be used as a starting point for future research. With *big data* being produced and to be processed at different sites of the cloud, multisite management of SWf execution in the cloud becomes more and more important. Based on this thesis' results, we can propose some future research directions:

- **Provenance distribution.** The existing SWfMSs generally store the provenance data in a centralized way and stores the data at a centralized site. This may incur network congestion when large amounts of tasks are executed in parallel. Thus, we believe that distributed provenance data management can reduce the time to generate or retrieve data at each site in order to reduce the overall SWf execution time in a multisite cloud. In addition, with data replication approaches in the distributed provenance data architecture, we believe that this approach can also help fault-tolerance at the multisite level, *i.e.* the situation where a cloud site is down.

- **Data transfer.** The data transfer between two sites is generally achieved by having two nodes, each at one of the two sites, communicating directly. This method is not efficient in a multisite cloud. One possible solution is to select several nodes at each site to send or receive data, by exploiting parallel data transfer and making the data transfer more efficient. Then, the problem of matching the nodes at two cloud sites is critical for the multisite data transfer rate.

- **Multisite Spark.** Because of in-memory data processing, Spark has become a major framework for big data processing. Spark can be used as an engine to execute SWfs. However, Spark is designed for a single cluster environment, where the bandwidth among different computing nodes are high and similar. Thus, it is interesting to propose multisite scheduling algorithms and optimizations to use Spark for multisite SWf execution. In addition, the problem of VM provisioning for Spark clusters in the cloud also remains critical to use Spark in a multisite cloud.

- **Architecture.** The structure of SWfMSs is generally centralized, with a master node, which is a single point of failure and performance bottleneck, managing all the optimization and scheduling processes. In addition, the system models (see Chapter 3, 5, 6) presented in this thesis are based on a master-worker model. A peer-to-peer architecture can be used in order to achieve fault-tolerance during the execution of SWfs both within a single site cloud or a multisite cloud. With a peer-to-peer architecture and data replication approaches, we believe the multisite SWfMSs are robust enough to address the situation where one computing node is down or even a cloud site is down.

- **Dynamic Scheduling.** The scheduling algorithms generally schedule activities or tasks according to predefined parameters with a static scheduling approach. However, the workload or the features of VMs are dynamically varying at each site of the cloud. We believe that dynamic scheduling of activities or tasks can perform better with the parameters measured during SWf execution in terms of execution time, monetary cost, energy consumption, and others for SWf execution in a multisite cloud.

Bibliography

[1] Alice Collaboration. http://aliceinfo.cern.ch/general/index.html.

[2] Azure service bus. http://azure.microsoft.com/en-us/services/service-bus/.

[3] Computing capacity for a CPU. http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329.

[4] DBLP Computer Science Bibliography. http://dblp.uni-trier.de/.

[5] Microsoft Azure. http://azure.microsoft.com.

[6] Montage. http://montage.ipac.caltech.edu/docs/gridtools.html.

[7] Oma genome database. http://omabrowser.org/All/eukaryotes.cdna.fa.gz.

[8] Parameters of different types of vms in microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/.

[9] Sequence identifier. http://omabrowser.org/All/oma-groups.txt.gz.

[10] VM bandwidth in Azure. http://windowsazureguide.net/tag/auzre-virtual-machines-sizes-bandwidth/.

[11] VM parameters in Azure. http://msdn.microsoft.com/en-us/library/azure/dn197896.aspx.

[12] Amazon cloud. http://aws.amazon.com/, 2015.

[13] Grid’5000 project. https://www.grid5000.fr/mediawiki/index.php, 2015.

[14] Microsoft Azure cloud. http://azure.microsoft.com/, 2015.

[15] Pegasus 4.4.1 user guide. `https://pegasus.isi.edu/wms/docs/latest/`, 2015.

[16] ABOUELHODA, M., ISSA, S., AND GHANEM, M. Tavaxy: Integrating taverna and galaxy workflows with cloud computing support. *BMC Bioinformatics 13*, 1 (2012), 77.

[17] AFGAN, E., BAKER, D., CORAOR, N., CHAPMAN, B., NEKRUTENKO, A., AND TAYLOR, J. Galaxy cloudman: delivering cloud compute clusters. *BMC Bioinformatics 11*, Suppl 12 (2010), S4.

[18] ALBRECHT, M., DONNELLY, P., BUI, P., AND THAIN, D. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies* (2012), pp. 1:1–1:13.

[19] ALTINTAS, I., BARNEY, O., AND JAEGER-FRANK, E. Provenance collection support in the kepler scientific workflow system. In *Int. Conf. on Provenance and Annotation of Data* (2006), pp. 118–132.

[20] ALTINTAS, I., BERKLEY, C., JAEGER, E., JONES, M., LUDASCHER, B., AND MOCK, S. Kepler: an extensible system for design and execution of scientific workflows. In *16th Int. Conf. on Scientific and Statistical Database Management (SSDBM)* (2004), pp. 423–424.

[21] ALTINTAS, I., BERKLEY, C., JAEGER, E., JONES, M., LUDÄSCHER, B., AND MOCK, S. Kepler: Towards a Grid-Enabled system for scientific workflows. *The Workflow in Grid Systems Workshop in GGF10-The 10th Global Grid Forum* (2004).

[22] ANGLANO, C., AND CANONICO, M. Scheduling algorithms for multiple bag-of-task applications on desktop grids: A knowledge-free approach. In *22nd IEEE Int. Symposium on Parallel and Distributed Processing (IPDPS)* (2008), pp. 1–8.

[23] ARKOUDAS, K., ZEE, K., KUNCAK, V., AND RINARD, M. Verifying a file system implementation. In *6th Int. Conf. on Formal Engineering Methods (ICFEM)*, vol. 3308. 2004, pp. 373–390.

[24] BALASKÓ, Á. Workflow concept of ws-pgrade/guse. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 33–50.

[25] BARKER, A., AND HEMERT, J. V. Scientific workflow: A survey and research directions. In *7th Int. Conf. on Parallel Processing and Applied Mathematics* (2008), pp. 746–753.

[26] Belhajjame, K., Cresswell, S., Gil, Y., Golden, R., Groth, P., Klyne, G., McCusker, J., Miles, S., Myers, J., and Sahoo, S. The prov data model and abstract syntax notation. `http://www.w3.org/TR/2011/WD-prov-dm-20111215/`, 2011.

[27] Bergmann, R., and Gil, Y. Retrieval of semantic workflows with knowledge intensive similarity measures. In *19th Int. Conf. on Case-Based Reasoning Research and Development* (2011), pp. 17–31.

[28] Bhuvaneshwar, K., Sulakhe, D., Gauba, R., Rodriguez, A., Madduri, R., Dave, U., Lacinski, L., Foster, I., Gusev, Y., and Madhavan, S. A case study for cloud based high throughput analysis of {NGS} data using the globus genomics system. *Computational and Structural Biotechnology Journal 13* (2015), 64 – 74.

[29] Blagodurov, S., Fedorova, A., Vinnik, E., Dwyer, T., and Hermenier, F. Multi-objective job placement in clusters. In *Proceedings of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC* (2015), pp. 66:1–66:12.

[30] Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., and Kennedy, K. Task scheduling strategies for workflow-based applications in grids. In *5th IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid)* (2005), pp. 759–767.

[31] Bouganim, L., Fabret, F., Mohan, C., and Valduriez, P. Dynamic query scheduling in data integration systems. In *International Conference on Data Engineering (ICDE)* (2000), pp. 425–434.

[32] Brandic, I., and Dustdar, S. Grid vs cloud - A technology comparison. *it - Information Technology 53*, 4 (2011), 173–179.

[33] Bux, M., and Leser, U. Parallelization in scientific workflow management systems. *The Computing Research Repository (CoRR) abs/1303.7195* (2013).

[34] Cala, J., Xu, Y., Wijaya, E., and Missier, P. From scripted hpc-based NGS pipelines to workflows on the cloud. In *14th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (2014), pp. 694–700.

[35] Carpenter, B., Getov, V., Judd, G., Skjellum, A., and Fox, G. Mpj: Mpi-like message passing for java. *Concurrency and Computation: Practice and Experience 12*, 11 (2000), 1019–1038.

[36] Chang, D., Son, J. H., and Kim, M. Critical path identification in the context of a workflow. *Information & Software Technology 44*, 7 (2002), 405–417.

[37] CHEN, W., AND DEELMAN, E. Integration of workflow partitioning and resource provisioning. In *IEEE/ACM Int. Symposium on Cluster Computing and the Grid (CCGRID)* (2012), pp. 764–768.

[38] CHEN, W., AND DEELMAN, E. Partitioning and scheduling workflows across multiple sites with storage constraints. In *9th Int. Conf. on Parallel Processing and Applied Mathematics - Volume Part II* (2012), vol. 7204, pp. 11–20.

[39] CHEN, W., SILVA, R. D., DEELMAN, E., AND SAKELLARIOU, R. Balanced task clustering in scientific workflows. In *IEEE 9th Int. Conf. on e-Science* (2013), pp. 188–195.

[40] CHERVENAK, A. L., SMITH, D. E., CHEN, W., AND DEELMAN, E. Integrating policy with scientific workflow management for data-intensive applications. In *Supercomputing (SC) Companion: High Performance Computing, Networking Storage and Analysis* (2012), pp. 140–149.

[41] CHIRIGATI, F., SILVA, V., OGASAWARA, E., DE OLIVEIRA, D., DIAS, J., PORTO, F., VALDURIEZ, P., AND MATTOSO, M. Evaluating parameter sweep workflows in high performance computing. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies* (2012), pp. 2:1–2:10.

[42] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications 41*, 4 (2011), 98–109.

[43] COALITION, W. M. Workflow management coalition terminology and glossary, 1999.

[44] COHEN-BOULAKIA, S., CHEN, J., MISSIER, P., GOBLE, C. A., WILLIAMS, A. R., AND FROIDEVAUX, C. Distilling structure in taverna scientific workflows: a refactoring approach. *BMC Bioinformatics 15*, S-1 (2014), S12.

[45] COSTA, F., DE OLIVEIRA, D., OCALA, K., OGASAWARA, E., DIAS, J., AND MATTOSO, M. Handling failures in parallel scientific workflows using clouds. In *Supercomputing (SC) Companion: High Performance Computing, Networking Storage and Analysis* (2012), pp. 129–139.

[46] COSTA, F., SILVA, V., DE OLIVEIRA, D., OCAÑA, K. A. C. S., OGASAWARA, E. S., DIAS, J., AND MATTOSO, M. Capturing and querying workflow runtime provenance with prov: a practical approach. In *EDBT/ICDT Workshops* (2013), pp. 282–289.

[47] Coutinho, R., Drummond, L., Frota, Y., de Oliveira, D., and Ocana, K. Evaluating grasp-based cloud dimensioning for comparative genomics: A practical approach. In *2014 IEEE Int. Conf. on Cluster Computing (CLUSTER)* (2014), pp. 371–379.

[48] Crawl, D., Wang, J., and Altintas, I. Provenance for mapreduce-based data-intensive workflows. In *6th Workshop on Workflows in Support of Large-scale Science* (2011), pp. 21–30.

[49] Critchlow, T., and Jr., G. C. Supercomputing and scientific workflows gaps and requirements. In *World Congress on Services* (2011), pp. 208–211.

[50] de Oliveira, D., Ocaña, K. A. C. S., Baião, F., and Mattoso, M. A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds. *Journal of Grid Computing 10*, 3 (2012), 521–552.

[51] de Oliveira, D., Ogasawara, E., Baião, F., and Mattoso, M. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In *3rd Int. Conf. on Cloud Computing (CLOUD)* (2010), pp. 378–385.

[52] de Oliveira, D., Ogasawara, E., Ocaña, K., Baião, F., and Mattoso, M. An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation: Practice & Experience 24*, 13 (2012), 1531–1550.

[53] de Oliveira, D., Viana, V., Ogasawara, E., Ocana, K., and Mattoso, M. Dimensioning the virtual cluster for parallel scientific workflows in clouds. In *4th ACM Workshop on Scientific Cloud Computing* (2013), pp. 5–12.

[54] de Oliveira, D., Viana, V., Ogasawara, E. S., Ocaña, K. A. C. S., and Mattoso, M. Dimensioning the virtual cluster for parallel scientific workflows in clouds. In *ScienceCloud'13, Proceedings of the 4th ACM HPDC Workshop on Scientific Cloud Computing* (2013), pp. 5–12.

[55] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)* (2004), pp. 137–150.

[56] Deelman, E., Callaghan, S., Field, E., Francoeur, H., Graves, R., Gupta, N., Gupta, V., Jordan, T. H., Kesselman, C., Maechling, P., Mehringer, J., Mehta, G., Okaya, D., Vahi, K., and Zhao, L. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *Second Int. Conf. on e-Science and Grid Technologies* (2006), p. 14.

[57] DEELMAN, E., GANNON, D., SHIELDS, M., AND TAYLOR, I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems 25*, 5 (2009), 528–540.

[58] DEELMAN, E., JUVE, G., AND BERRIMAN, G. B. Using clouds for science, is it just kicking the can down the road? In *Cloud Computing and Services Science (CLOSER), 2nd Int. Conf. on Cloud Computing and Services Science* (2012), pp. 127–134.

[59] DEELMAN, E., SINGH, G., LIVNY, M., BERRIMAN, B., AND GOOD, J. The cost of doing science on the cloud: The montage example. In *ACM/IEEE Conf. on High Performance Computing* (2008), pp. 1–12.

[60] DEELMAN, E., SINGH, G., SU, M.-H., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BERRIMAN, G. B., GOOD, J., LAITY, A., JACOB, J. C., AND KATZ, D. S. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming 13*, 3 (2005), 219–237.

[61] DEELMAN, E., VAHI, K., JUVE, G., RYNGE, M., CALLAGHAN, S., MAECHLING, P. J., MAYANI, R., CHEN, W., D. SILVA, R. F., LIVNY, M., AND WENGER, K. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems* (2014).

[62] DENG, K., KONG, L., SONG, J., REN, K., AND YUAN, D. A weighted k-means clustering based co-scheduling strategy towards efficient execution of scientific workflows in collaborative cloud environments. In *IEEE 9th Int. Conf. on Dependable, Autonomic and Secure Computing (DASC)* (2011), pp. 547–554.

[63] DIAS, J., DE OLIVEIRA, D., MATTOSO, M., OCANA, K. A. C. S., AND OGASAWARA, E. Discovering drug targets for neglected diseases using a pharmacophylogenomic cloud workflow. In *IEEE 8th Int. Conf. on E-Science (e-Science)* (2012), pp. 1–8.

[64] DIAS, J., OGASAWARA, E. S., DE OLIVEIRA, D., PORTO, F., VALDURIEZ, P., AND MATTOSO, M. Algebraic dataflows for big data analysis. In *IEEE Int. Conf. on Big Data* (2013), pp. 150–155.

[65] DUAN, R., PRODAN, R., AND LI, X. Multi-objective game theoretic schedulingof bag-of-tasks workflows on hybrid clouds. *IEEE Transactions on Cloud Computing 2*, 1 (2014), 29–42.

[66] EMEAKAROHA, V. C., MAURER, M., STERN, P., LABAJ, P. P., BRANDIC, I., AND KREIL, D. P. Managing and optimizing bioinformatics workflows for data analysis in clouds. *Journal of Grid Computing 11*, 3 (2013), 407–428.

[67] ETMINANI, K., AND NAGHIBZADEH, M. A min-min max-min selective algorihtm for grid task scheduling. In *The Third IEEE/IFIP Int. Conf. in Central Asia on Internet (ICI 2007)* (2007), pp. 1–7.

[68] FAHRINGER, T., PRODAN, R., DUAN, R., HOFER, J., NADEEM, F., NERIERI, F., PODLIPNIG, S., QIN, J., SIDDIQUI, M., TRUONG, H., VILLAZON, A., AND WIECZOREK, M. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[69] FARD, H. M., FAHRINGER, T., AND PRODAN, R. Budget-constrained resource provisioning for scientific applications in clouds. In *IEEE 5th Int. Conf. on Cloud Computing Technology and Science (CloudCom)* (2013), vol. 1, pp. 315–322.

[70] FARD, H. M., PRODAN, R., AND FAHRINGER, T. Multi-objective list scheduling of workflow applications in distributed computing infrastructures. *Journal of Parallel and Distributed Computing 74*, 3 (2014), 2152–2165.

[71] FARKAS, Z., HAJNAL, Á., AND KACSUK, P. Ws-pgrade/guse and clouds. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 97–109.

[72] FELSENSTEIN, J. Phylip - phylogeny inference package (version 3.2). *Cladistics 5* (1989), 164–166.

[73] FOSTER, I., AND KESSELMAN, C. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.

[74] FREIRE, J., KOOP, D., SANTOS, E., AND SILVA, C. T. Provenance for computational tasks: A survey. *Computing in Science and Engineering 10*, 3 (2008), 11–21.

[75] FREY, J., TANNENBAUM, T., LIVNY, M., FOSTER, I., AND TUECKE, S. Condor-g: a computation management agent for multi-institutional grids. In *10th IEEE Int. Symposium on High Performance Distributed Computing* (2001), pp. 55–63.

[76] GADELHA JR., L. M. R., WILDE, M., MATTOSO, M., AND FOSTER, I. Provenance traces of the swift parallel scripting system. In *EDBT/ICDT Workshops* (2013), pp. 325–326.

[77] GANGA, K., AND KARTHIK, S. A fault tolerent approach in scientific workflow systems based on cloud computing. In *Int. Conf. on Pattern Recognition, Informatics and Mobile Engineering (PRIME)* (2013), pp. 387–390.

[78] GARIJO, D., ALPER, P., BELHAJJAME, K., CORCHO, Ó., GIL, Y., AND GOBLE, C. A. Common motifs in scientific workflows: An empirical analysis. *Future Generation Computer Systems 36* (2014), 338–351.

[79] Gesing, S., Krüger, J., Grunzke, R., de la Garza, L., Herres-Pawlis, S., and Hoffmann, A. Molecular simulation grid (mosgrid): A science gateway tailored to the molecular simulation community. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 151–165.

[80] Ghemawat, S., Gobioff, H., and Leung, S. The google file system. In *19th ACM Symposium on Operating Systems Principles* (2003), pp. 29–43.

[81] Gil, Y., Kim, J., Ratnakar, V., and Deelman, E. Wings for pegasus: A semantic approach to creating very large scientific workflows. In *OWLED*06 Workshop on OWL: Experiences and Directions* (2006), vol. 216.

[82] Goecks, J., Nekrutenko, A., and Taylor, J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology 11*, 8 (2010), 1–13.

[83] Goecks, J., Nekrutenko, A., and Taylor, J. Lessons learned from galaxy, a web-based platform for high-throughput genomic analyses. In *IEEE Int. Conf. on E-Science, e-Science* (2012), pp. 1–6.

[84] Gonçalves, J. A. R., Oliveira, D., Ocaña, K., Ogasawara, E., and Mattoso, M. Using domain-specific data to enhance scientific workflow steering queries. In *Provenance and Annotation of Data and Processes*, vol. 7525. 2012, pp. 152–167.

[85] Gonzalez, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. A. A break in the clouds: towards a cloud definition. *Computer Communication Review 39*, 1 (2009), 50–55.

[86] Görlach, K., Sonntag, M., Karastoyanova, D., Leymann, F., and Reiter, M. Conventional workflow technology for scientific simulation. In *Guide to e-Science*. 2011, pp. 323–352.

[87] Gottdank, T. Introduction to the ws-pgrade/guse science gateway framework. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 19–32.

[88] Gu, Y., Wu, C., Liu, X., and Yu, D. Distributed throughput optimization for large-scale scientific workflows under fault-tolerance constraint. *Journal of Grid Computing 11*, 3 (2013), 361–379.

[89] Gunter, D., Deelman, E., Samak, T., Brooks, C., Goode, M., Juve, G., Mehta, G., Moraes, P., Silva, F., Swany, M., and Vahi, K. Online workflow management and performance analysis with stampede. In *7th Int. Conf. on Network and Service Management (CNSM)* (2011), pp. 1–10.

[90] HATEGAN, M., WOZNIAK, J., AND MAHESHWARI, K. Coasters: Uniform resource provisioning and access for clouds and grids. In *4th IEEE Int. Conf. on Utility and Cloud Computing* (2011), pp. 114–121.

[91] HERNÁNDEZ, F., AND FAHRINGER, T. Towards workflow sharing and reusein the askalon grid environment. In *Proceedings of Cracow Grid Workshops (CGW)* (2008), p. 111–119.

[92] HIDEN, H., WATSON, P., WOODMAN, S., AND LEAHY, D. e-science central: cloud-based e-science and its application to chemical property modelling. Tchnical Report CS-TR-1227, 2010.

[93] HIDEN, H., WOODMAN, S., WATSON, P., AND CALA, J. Developing cloud applications using the e-science central platform. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 371*, 1983 (2012).

[94] HOLL, S., ZIMMERMANN, O., AND HOFMANN-APITIUS, M. A new optimization phase for scientific workflow management systems. In *8th IEEE Int. Conf. on E-Science* (2012), pp. 1–8.

[95] HORTA, F., DIAS, J., OCANA, K., DE OLIVEIRA, D., OGASAWARA, E., AND MATTOSO, M. Abstract: Using provenance to visualize data from large-scale experiments. In *Supercomputing (SC): High Performance Computing, Networking Storage and Analysis* (2012), pp. 1418–1419.

[96] HUEDO, E., MONTERO, R. S., AND LLORENTE, I. M. A framework for adaptive execution in grids. *Software - Practice and Experience (SPE) 34*, 7 (2004), 631–651.

[97] HUME, A. C., AL-HAZMI, Y., BELTER, B., CAMPOWSKY, K., CARRIL., L. M., CARROZZO, G., ENGEN, V., GARCÍA-PÉREZ, D., PONSATÍ, J. J., UBERT, R. K., LIANG, Y., ROHR, C., AND SEGHBROECK, G. Bonfire: A multi-cloud test facility for internet of services experimentation. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, vol. 44. 2012, pp. 81–96.

[98] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems* (2007), pp. 59–72.

[99] JACKSON, K. *OpenStack Cloud Computing Cookbook.* Packt Publishing, 2012.

[100] JACOB, J. C., KATZ, D. S., BERRIMAN, G. B., GOOD, J. C., LAITY, A. C., DEELMAN, E., KESSELMAN, C., SINGH, G., SU, M.-H., PRINCE, T. A., AND WILLIAMS, R. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Int. Journal of Computational Science and Engineering 4*, 2 (2009), 73–87.

[101] JUVE, G., AND DEELMAN, E. Scientific workflows in the cloud. In *Grids, Clouds and Virtualization*. Springer, 2011, pp. 71–91.

[102] JUVE, G., AND DEELMAN, E. Wrangler: Virtual cluster provisioning for the cloud. In *20th Int. Symposium on High Performance Distributed Computing* (2011), pp. 277–278.

[103] JUVE, G., RYNGE, M., DEELMAN, E., VOCKLER, J.-S., AND BERRIMAN, G. Comparing futuregrid, amazon ec2, and open science grid for scientific workflows. *Computing in Science Engineering 15*, 4 (2013), 20–29.

[104] KACSUK, P. P-grade portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience 23*, 3 (2011), 235–245.

[105] KACSUK, P., FARKAS, Z., KOZLOVSZKY, M., HERMANN, G., BALASKO, A., KAROCZKAI, K., AND MARTON, I. Ws-pgrade/guse generic dci gateway framework for a large variety of user communities. *Journal of Grid Computing 10*, 4 (2012), 601–630.

[106] KARUNA, K., MANGALA, N., JANAKI, C., SHASHI, S., AND SUBRATA, C. Galaxy workflow integration on garuda grid. In *IEEE Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (2012), pp. 194–196.

[107] KARYPIS, G., AND KUMAR, V. Multilevel algorithms for multi-constraint graph partitioning. In *ACM/IEEE Conf. on Supercomputing* (1998), pp. 1–13.

[108] KIM, J., DEELMAN, E., GIL, Y., MEHTA, G., AND RATNAKAR, V. Provenance trails in the wings-pegasus system. *Concurrency and Computation: Practice and Experience 20* (2008), 587–597.

[109] KISS, T., KACSUK, P., LOVAS, R., BALASKÓ, Á., SPINUSO, A., ATKINSON, M., D'AGOSTINO, D., DANOVARO, E., AND SCHIFFERS, M. Ws-pgrade/guse in european projects. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 235–254.

[110] KISS, T., KACSUK, P., TAKÁCS, E., SZABÓ, Á., TIHANYI, P., AND TAYLOR, S. Commercial use of ws-pgrade/guse. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 271–286.

[111] KOCAIR, Ç., ŞENER, C., AND AKKAYA, A. Statistical seismology science gateway. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 167–180.

[112] KORF, I., YANDELL, M., AND BEDELL, J. A. *BLAST - an essential guide to the basic local alignment search tool.* O'Reilly, 2003.

[113] KOZLOVSZKY, M., KARÓCZKAI, K., MÁRTON, I., KACSUK, P., AND GOTTDANK, T. Dci bridge: Executing ws-pgrade workflows in distributed computing infrastructures. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 51–67.

[114] LITTAUER, R., RAM, K., LUDÄSCHER, B., MICHENER, W., AND KOSKELA, R. Trends in use of scientific workflows: Insights from a public repository and recommendations for best practice. *International Journal of Digital Curation (IJDC) 7*, 2 (2012), 92–100.

[115] LITZKOW, M. J., LIVNY, M., AND MUTKA, M. W. Condor-a hunter of idle workstations. In *8th Int. Conf. on Distributed Computing Systems* (1988), pp. 104–111.

[116] LIU, B., SOTOMAYOR, B., MADDURI, R., CHARD, K., AND FOSTER, I. Deploying bioinformatics workflows on clouds with galaxy and globus provision. In *Supercomputing (SC) Companion: High Performance Computing, Networking, Storage and Analysis (SCC)* (2012), pp. 1087–1095.

[117] LIU, J., PACITTI, E., VALDURIEZ, P., DE OLIVEIRA, D., AND MATTOSO, M. Multi-objective scheduling of scientific workflows in multisite clouds. *Future Generation Computer Systems 63* (2016), 76–95.

[118] LIU, J., PACITTI, E., VALDURIEZ, P., DE OLIVEIRA, D., AND MATTOSO, M. Multi-objective scheduling of scientificworkflows in multisite clouds. In *BDA* (2016). To appear.

[119] LIU, J., PACITTI, E., VALDURIEZ, P., AND MATTOSO, M. Parallelization of scientific workflows in the cloud. Research Report RR-8565, 2014.

[120] LIU, J., PACITTI, E., VALDURIEZ, P., AND MATTOSO, M. A survey of data-intensive scientific workflow management. *Journal of Grid Computing* (2015), 1–37.

[121] LIU, J., PACITTI, E., VALDURIEZ, P., AND MATTOSO., M. Scientific workflow scheduling with provenance support in multisite cloud. In *High Performance Computing for Computational Science VECPAR* (2016). To appear.

[122] LIU, J., SILVA, V., PACITTI, E., VALDURIEZ, P., AND MATTOSO, M. Scientific workflow partitioning in multisite cloud. In *BigDataCloud'2014: 3rd Workshop on Big Data Management in Clouds in conjunction with Euro-Par* (2014), pp. 105–116.

[123] LUDÄSCHER, B., ALTINTAS, I., BERKLEY, C., HIGGINS, D., JAEGER, E., JONES, M. B., LEE, E. A., TAO, J., AND ZHAO, Y. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience 18*, 10 (2006), 1039–1065.

[124] LUO, Y., AND PLALE, B. Hierarchical mapreduce programming model and scheduling algorithms. In *Proceedings of 12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (Ccgrid)* (2012), pp. 769–774.

[125] MAHESWARAN, M., ALI, S., SIEGEL, H. J., HENSGEN, D., AND FREUND, R. F. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th Heterogeneous Computing Workshop* (1999), pp. 30–.

[126] MALAWSKI, M., JUVE, G., DEELMAN, E., AND NABRZYSKI, J. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *Supercomputing (SC) Conf. on High Performance Computing Networking, Storage and Analysis* (2012), pp. 1–11.

[127] MANDAL, A., XIN, Y., BALDINE, I., RUTH, P., HEERMAN, C., CHASE, J., ORLIKOWSKI, V., AND YUMEREFENDI, A. Provisioning and evaluating multi-domain networked clouds for hadoop-based applications. In *Cloud Computing Technology and Science (CloudCom), IEEE 3rd Int. Conf. on Cloud Computing Technology and Science* (2011), pp. 690–697.

[128] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26*, 6 (2004), 369–395.

[129] MATTOSO, M., DIAS, J., OCAÑA, K. A., OGASAWARA, E., COSTA, F., HORTA, F., SILVA, V., AND DE OLIVEIRA, D. Dynamic steering of HPC scientific workflows: A survey. *Future Generation Computer Systems*, 0 (2014).

[130] MATTOSO, M., WERNER, C., TRAVASSOS, G., BRAGANHOLO, V., OGASAWARA, E., OLIVEIRA, D., CRUZ, S., MARTINHO, W., AND MURTA, L. Towards supporting the life cycle of large scale scientific experiments. In *Int. J. Business Process Integration and Management*, vol. 5. 2010, pp. 79–82.

[131] MILOJICIC, D. S., LLORENTE, I. M., AND MONTERO, R. S. Opennebula: A cloud management tool. *IEEE Internet Computing 15*, 2 (2011), 11–14.

[132] MISSIER, P., SOILAND-REYES, S., OWEN, S., TAN, W., NENADIC, A., DUNLOP, I., WILLIAMS, A., OINN, T., AND GOBLE, C. Taverna, reloaded. In *Int. Conf. on Scientific and Statistical Database Management* (2010), pp. 471–481.

[133] NAGAVARAM, A., AGRAWAL, G., FREITAS, M. A., TELU, K. H., MEHTA, G., MAYANI, R. G., AND DEELMAN, E. A cloud-based dynamic workflow for mass spectrometry data analysis. In *IEEE 7th Int. Conf. on E-Science (e-Science)* (2011), pp. 47–54.

[134] NGUYEN, D., AND THOAI, N. Ebc: Application-level migration on multi-site cloud. In *Int. Conf. on Systems and Informatics (ICSAI)* (2012), pp. 876–880.

[135] Nicolae, B., Antoniu, G., Bougé, L., Moise, D., and Carpen-Amarie, A. Blobseer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing 71*, 2 (2011), 169–184.

[136] Ocaña, K. A., Oliveira, D., Ogasawara, E., Dávila, A. M., Lima, A. A., and Mattoso, M. Sciphy: A cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes. In *Advances in Bioinformatics and Computational Biology*, vol. 6832. 2011, pp. 66–70.

[137] Ocaña, K. A. C. S., Oliveira, D., Horta, F., Dias, J., Ogasawara, E., and Mattoso, M. Exploring molecular evolution reconstruction using a parallel cloud based scientific workflow. In *Advances in Bioinformatics and Computational Biology*, vol. 7409. 2012, pp. 179–191.

[138] Ogasawara, E. S., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., and Mattoso, M. An algebraic approach for data-centric scientific workflows. *Proceedings of the VLDB Endowment (PVLDB) 4*, 12 (2011), 1328–1339.

[139] Ogasawara, E. S., Dias, J., Silva, V., Chirigati, F. S., de Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience 25*, 16 (2013), 2327–2341.

[140] Oinn, T., Li, P., Kell, D. B., Goble, C., Goderis, A., Greenwood, M., Hull, D., Stevens, R., Turi, D., and Zhao, J. Taverna/mygrid: Aligning a workflow system with the life sciences community. In *Workflows for e-Science*. 2007, pp. 300–319.

[141] Oinn, T. M., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, R. M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics 20*, 17 (2004), 3045–3054.

[142] Olabarriaga, S., Benabdelkader, A., Caan, M., Jaghoori, M., Krüger, J., de la Garza, L., Mohr, C., Schubert, B., Danezi, A., and Kiss, T. Ws-pgrade/guse-based science gateways in teaching. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 223–234.

[143] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)* (2008), pp. 1099–1110.

[144] Ostermann, S., Plankensteiner, K., Prodan, R., and Fahringer, T. Groudsim: An event-based simulation framework for computational grids and clouds. In *European Conf. on Parallel Processing (Euro-Par) Workshops* (2011), pp. 305–313.

[145] OSTERMANN, S., PRODAN, R., AND FAHRINGER, T. Extending grids with cloud resource management for scientific computing. In *10th IEEE/ACM Int. Conf. on Grid Computing* (2009), pp. 42–49.

[146] ÖZSU, M. T., AND VALDURIEZ, P. *Principles of Distributed Database Systems*. Springer, 2011.

[147] PACITTI, E., AKBARINIA, R., AND DICK, M. E. *P2P Techniques for Decentralized Applications*. Morgan & Claypool Publishers, 2012.

[148] PAUTASSO, C., AND ALONSO, G. Parallel computing patterns for grid workflows. In *Workshop on Workflows in Support of Large-Scale Science* (2006), pp. 1–10.

[149] PERRY, J., SMITH, L., JACKSON, A. N., KENWAY, R. D., JOO, B., MAYNARD, C. M., TREW, A., BYRNE, D., BECKETT, G., DAVIES, C. T. H., DOWNING, S., IRVING, A. C., MCNEILE, C., SROCZYNSKI, Z., ALLTON, C. R., ARMOUR, W., AND FLYNN, J. M. Qcdgrid: A grid resource for quantum chromodynamics. *Journal of Grid Computing 3*, 1 (2005), 113–130.

[150] PINEDA-MORALES, L., COSTAN, A., AND ANTONIU, G. Towards multi-site metadata management for geographically distributed cloud workflows. In *2015 IEEE Int. Conf. on Cluster Computing, CLUSTER* (2015), pp. 294–303.

[151] PLANKENSTEINER, K., PRODAN, R., JANETSCHEK, M., FAHRINGER, T., MONTAGNAT, J., ROGERS, D., HARVEY, I., TAYLOR, I., BALASKÓ, Á., AND KACSUK, P. Fine-grain interoperability of scientific workflows in distributed computing infrastructures. *Journal of Grid Computing 11*, 3 (2013), 429–455.

[152] PRESLAN, K. W., BARRY, A. P., BRASSOW, J. E., ERICKSON, G. M., NYGAARD, E., SABOL, C. J., SOLTIS, S. R., TEIGLAND, D. C., AND O'KEEFE, M. T. A 64-bit, shared disk file system for linux. In *16th IEEE Symposium on Mass Storage Systems* (1999), pp. 22–41.

[153] PRODAN, R. Online analysis and runtime steering of dynamic workflows in the askalon grid environment. In *7th IEEE Int. Symposium on Cluster Computing and the Grid (CCGRID)* (2007), pp. 389–400.

[154] RAHMAN, M., HASSAN, M. R., RANJAN, R., AND BUYYA, R. Adaptive workflow scheduling for dynamic grid and cloud computing environment. *Concurrency and Computation: Practice and Experience 25*, 13 (2013), 1816–1842.

[155] RAICU, I., ZHAO, Y., FOSTER, I. T., AND SZALAY, A. S. Data diffusion: Dynamic resource provision and data-aware scheduling for data intensive applications. *The Computing Research Repository (CoRR) abs/0808.3535* (2008).

[156] RAMAKRISHNAN, A., SINGH, G., ZHAO, H., DEELMAN, E., SAKELLARIOU, R., VAHI, K., BLACKBURN, K., MEYERS, D., AND SAMIDI, M. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *7th IEEE Int. Symposium on Cluster Computing and the Grid (CCGRID)* (2007), pp. 401–409.

[157] REYNOLDS, C. J., WINTER, S. C., TERSTYÁNSZKY, G., KISS, T., GREENWELL, P., ACS, S., AND KACSUK, P. Scientific workflow makespan reduction through cloud augmented desktop grids. In *IEEE 3rd International Conference on Cloud Computing Technology and Science* (2011), pp. 18–23.

[158] RODRIGUEZ, M. A., AND BUYYA, R. A responsive knapsack-based algorithm for resource provisioning and scheduling of scientific workflows in clouds. In *44th Int. Conf. on Parallel Processing, ICPP* (2015).

[159] SAMAK, T., GUNTER, D., GOODE, M., DEELMAN, E., JUVE, G., MEHTA, G., SILVA, F., AND VAHI, K. Online fault and anomaly detection for large-scale scientific workflows. In *13th IEEE Int. Conf. on High Performance Computing and Communications (HPCC)* (2011), pp. 373–381.

[160] SANDBERG, R., GOLGBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Innovations in internetworking. 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[161] SARDIÑA, I., BOERES, C., AND DE A. DRUMMOND, L. An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. In *Euro-Par 2009 – Parallel Processing Workshops*, vol. 6043. 2010, pp. 102–111.

[162] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *1st USENIX Conf. on File and Storage Technologies* (2002).

[163] SCIACCA, E., VITELLO, F., BECCIANI, U., COSTA, A., AND MASSIMINO, P. Visivo gateway and visivo mobile for the astrophysics community. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 181–194.

[164] SHAHAND, S., JAGHOORI, M., BENABDELKADER, A., FONT-CALVO, J., HUGUET, J., CAAN, M., VAN KAMPEN, A., AND OLABARRIAGA, S. Computational neuroscience gateway: A science gateway based on the ws-pgrade/guse. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 139–149.

[165] SHANKAR, S., AND DEWITT, D. J. Data driven workflow planning in cluster management systems. In *16th International Symposium on High-Performance Distributed Computing (HPDC-16)* (2007), pp. 127–136.

[166] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing in conjunction with SOSP* (2011), p. 5.

[167] SINGH, G., SU, M.-H., VAHI, K., DEELMAN, E., BERRIMAN, B., GOOD, J., KATZ, D. S., AND MEHTA, G. Workflow task clustering for best effort systems with pegasus. In *15th ACM Mardi Gras Conf.: From Lightweight Mash-ups to Lambda Grids: Understanding the Spectrum of Distributed Computing Requirements, Applications, Tools, Infrastructures, Interoperability, and the Incremental Adoption of Key Capabilities* (2008), pp. 9:1–9:8.

[168] SMANCHAT, S., INDRAWAN, M., LING, S., ENTICOTT, C., AND ABRAMSON, D. Scheduling multiple parameter sweep workflow instances on the grid. In *5th IEEE Int. Conf. on e-Science* (2009), pp. 300–306.

[169] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, 1998.

[170] SUN, X., AND CHEN, Y. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing 70*, 2 (2010), 183–188.

[171] TANAKA, M., AND TATEBE, O. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (Ccgrid)* (2012), pp. 65–72.

[172] TARAPANOFF, K., QUONIAM, L., DE ARAÚJO JÚNIOR, R. H., AND ALVARES, L. Intelligence obtained by applying data mining to a database of french theses on the subject of brazil. *Information Research 7*, 1 (2001).

[173] TAYLOR, I., SHIELDS, M., WANG, I., AND HARRISON, A. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*. Springer, 2007, pp. 320–339.

[174] TERSTYÁNSZKY, G., KUKLA, T., KISS, T., KACSUK, P., BALASKÓ, Á., AND FARKAS, Z. Enabling scientific workflow sharing through coarse-grained interoperability. *Future Generation Computer Systems 37* (2014), 46–59.

[175] TERSTYÁNSZKY, G., MICHNIAK, E., KISS, T., AND BALASKÓ, Á. Sharing science gateway artefacts through repositories. In *Science Gateways for Distributed Computing Infrastructures*, P. Kacsuk, Ed. Springer International Publishing, 2014, pp. 123–135.

[176] TOOSI, A. N., CALHEIROS, R. N., AND BUYYA, R. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Computing Surveys 47*, 1 (2014), 7:1–7:47.

[177] TOPCUOUGLU, H., HARIRI, S., AND WU, M. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems 13*, 3 (2002), 260–274.

[178] TUDORAN, R., COSTAN, A., ANTONIU, G., AND SONCU, H. Tomusblobs: Towards communication-efficient storage for mapreduce applications in azure. In *12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (2012), pp. 427–434.

[179] U-CHUPALA, P., UTHAYOPAS, P., ICHIKAWA, K., DATE, S., AND ABE, H. An implementation of a multi-site virtual cluster cloud. In *10th Int. Joint Conf. on Computer Science and Software Engineering (JCSSE)* (2013), pp. 155–159.

[180] V. D. AALST, W. M. P., WESKE, M., AND WIRTZ, G. Advanced topics in workflow management: Issues, requirements, and solutions. *Transactions of the SDPS 7*, 3 (2003), 49–77.

[181] VAHI, K., HARVEY, I., SAMAK, T., GUNTER, D., EVANS, K., ROGERS, D., TAYLOR, I., GOODE, M., SILVA, F., AL-SHAKARCHI, E., MEHTA, G., JONES, A., AND DEELMAN, E. A general approach to real-time workflow monitoring. In *Supercomputing (SC) Companion: High Performance Computing, Networking, Storage and Analysis (SCC)* (2012), pp. 108–118.

[182] WANG, J., AND ALTINTAS, I. Early cloud experiences with the kepler scientific workflow system. In *Int. Conf. on Computational Science (ICCS)* (2012), vol. 9, pp. 1630–1634.

[183] WANG, J., CRAWL, D., AND ALTINTAS, I. Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In *4th Workshop on Workflows in Support of Large-Scale Science* (2009), pp. 12:1–12:8.

[184] WANG, L., TAO, J., RANJAN, R., MARTEN, H., STREIT, A., CHEN, J., AND CHEN, D. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *Future Generation Comp. Syst. 29*, 3 (2013), 739–750.

[185] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[186] WIECZOREK, M., PRODAN, R., AND FAHRINGER, T. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Record 34*, 3 (2005), 56–62.

[187] WIEDER, P., BUTLER, J. M., THEILMANN, W., AND YAHYAPOUR, R. *Service Level Agreements for Cloud Computing*. Springer, 2011.

[188] WILDE, M., HATEGAN, M., WOZNIAK, J. M., CLIFFORD, B., KATZ, D. S., AND FOSTER, I. Swift: A language for distributed parallel scripting. *Parallel Computing 37*, 9 (2011), 633–652.

[189] WILLIAMS, D. N., DRACH, R., ANANTHAKRISHNAN, R., FOSTER, I. T., FRASER, D., SIEBENLIST, F., BERNHOLDT, D. E., CHEN, M., SCHWIDDER, J., BHARATHI, S., CHERVENAK, A. L., SCHULER, R., SU, M., BROWN, D., CINQUINI, L., FOX, P., GARCIA, J., MIDDLETON, D. E., STRAND, W. G., WILHELMI, N., HANKIN, S., SCHWEITZER, R., JONES, P., SHOSHANI, A., AND SIM, A. The earth system grid: Enabling access to multimodel climate simulation data. *Bulletin of the American Meteorological Society 90*, 2 (2009), 195–205.

[190] WOITASZEK, M., DENNIS, J. M., AND SINES, T. R. Parallel high-resolution climate data analysis using swift. In *ACM Int. Workshop on Many Task Computing on Grids and Supercomputers* (2011), pp. 5–14.

[191] WOLSTENCROFT, K., HAINES, R., FELLOWS, D., WILLIAMS, A. R., WITHERS, D., OWEN, S., SOILAND-REYES, S., DUNLOP, I., NENADIC, A., FISHER, P., BHAGAT, J., BELHAJJAME, K., BACALL, F., HARDISTY, A., DE LA HIDALGA, A. N., VARGAS, M. P. B., SUFI, S., AND GOBLE, C. A. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research 41*, Webserver-Issue (2013), 557–561.

[192] WOZNIAK, J. M., ARMSTRONG, T. G., MAHESHWARI, K., LUSK, E. L., KATZ, D. S., WILDE, M., AND FOSTER, I. T. Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies* (2012), pp. 5:1–5:12.

[193] XU, L., ZENG, Z., AND YE, X. Multi-objective optimization based virtual resource allocation strategy for cloud computing. In *IEEE/ACIS 11th Int. Conf. on Computer and Information Science* (2012), pp. 56–61.

[194] YILDIZ, U., GUABTNI, A., AND NGU, A. H. H. Business versus scientific workflows: A comparative study. In *IEEE Congress on Services, Part I, Services I* (2009), pp. 340–343.

[195] YU, J., AND BUYYA, R. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing 3* (2005), 171–200.

[196] YU, Z., AND SHI, W. An adaptive rescheduling strategy for grid workflow applications. In *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)* (2007), pp. 1–8.

[197] YUAN, D., YANG, Y., LIU, X., AND CHEN, J. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *IEEE Int. Symposium on Parallel Distributed Processing (IPDPS)* (2010), pp. 1–12.

[198] ZADEH, L. Optimality and non-scalar-valued performance criteria. *IEEE Transactions on Automatic Control 8*, 1 (1963), 59–60.

[199] ZHANG, H., SOILAND-REYES, S., AND GOBLE, C. A. Taverna mobile: Taverna workflows on android. *The Computing Research Repository (CoRR) abs/1309.2787* (2013).

[200] ZHANG, Q., CHENG, L., AND BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications 1* (2010), 7–18.

[201] ZHAO, Y., HATEGAN, M., CLIFFORD, B., FOSTER, I., VON LASZEWSKI, G., NEFEDOVA, V., RAICU, I., STEF-PRAUN, T., AND WILDE, M. Swift: Fast, reliable, loosely coupled parallel computation. In *IEEE Int. Conf. on Services Computing - Workshops (SCW)* (2007), pp. 199–206.

[202] ZHAO, Y., RAICU, I., AND FOSTER, I. T. Scientific workflow systems for 21st century, new bottle or new wine? In *IEEE Congress on Services, Part I, Services I* (2008), pp. 467–471.