Distributed Management of Scientific Workflows for High-Throughput Plant Phenotyping

Gaëtan Heidsieck

▶ **To cite this version:**

Gaëtan Heidsieck. Distributed Management of Scientific Workflows for High-Throughput Plant Phenotyping. Computer Science [cs]. Université de Montpellier (UM), FRA, 2020. English. NNT : . tel-03089552v1

HAL Id: tel-03089552

https://hal.science/tel-03089552v1

Submitted on 28 Dec 2020 (v1), last revised 16 Apr 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITE DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

Gestion distribuée de workflows scientifiques pour le phénotypage des plantes à haut débit

Présentée par Gaëtan Heidsieck

Le 9/12/2020

Sous la direction de Pacitti Esther et Tardieu François

Devant le jury composé de

Antoniu Gabriel, Directeur de recherche, Irisa Université de Rennes	Rapporteur
Ruiz Manuel, Directeur de recherche, Cirad	Rapporteur
Huchard Marianne, Professeur, LIRMM	Examinatrice
Pradal Christophe, Chargé de recherche, Cirad	Co-encadrant
Tardieu François, Directeur de recherche, Inrae	Co-encadrant
Esther Pacitti, Professeur, Université de Montpellier	Directrice

UNIVERSITÉ DE MONTPELLIER

Résumé

Dans de nombreux domaines scientifiques, les expériences numériques nécessitent généralement de nombreuses étapes de traitement ou d'analyse sur d'énormes ensembles de données. Elles peuvent être représentées comme des workflows scientifiques. Ces workflows facilitent la modélisation, la gestion et l'exécution d'activités de calcul liées par des dépendances de données. Comme la taille des données traitées et la complexité des calculs ne cessent d'augmenter, ces workflows deviennent orientés-données. Afin d'exécuter ces workflows dans un délai raisonnable, ils doivent être déployés dans un environnement informatique distribué à haute performance, tel que le cloud.

Le phénotypage des plantes vise à capturer les caractéristiques des plantes, telles que les caractéristiques morphologiques, topologiques et phénologiques. Des plateformes de phénotypage à haut débit ont vu le jour pour accélérer l'acquisition de données de phénotypage dans des conditions contrôlées (par exemple en serre) ou en plein champ. Ces plates-formes génèrent des téraoctets de données utilisées en sélection et en biologie végétale. Ces ensembles de données sont stockés dans différents sites géo-distribués. Les scientifiques peuvent utiliser un système de gestion de workflow scientifique (SWMS) pour gérer l'exécution du workflow sur un cloud multisite.

Dans le domaine des sciences biologiques, il est courant que les utilisateurs des workflows réutilisent d'autres analyses ou des données générées par d'autres utilisateurs. L'adaptation et la réutilisation des workflows permet à l'utilisateur de développer de nouvelles analyses plus rapidement. En outre, un utilisateur peut avoir besoin d'exécuter un workflow plusieurs fois avec différents ensembles de paramètres et de données d'entrée pour analyser l'impact d'une étape expérimentale quelconque, représentée comme un fragment du workflow. Dans les deux cas, certains fragments du workflow peuvent être exécutés plusieurs fois, ce qui peut être très consommateur de ressources et inutilement long. La ré-exécution du workflow peut être évitée en stockant les résultats intermédiaires de ces fragments et en les réutilisant dans des exécutions ultérieures.

Dans cette thèse, nous proposons une solution de mise en cache adaptative pour l'exécution efficace de workflows orientés-données dans des clouds monosites et multisites. En s'adaptant aux variations des temps d'exécution des tâches, notre solution peut maximiser la réutilisation des données intermédiaires produites par les workflows de plusieurs utilisateurs. Notre solution est basée sur une nouvelle architecture de SWMS qui gère automatiquement le stockage et la réutilisation des données intermédiaires. La gestion du cache intervient au cours de deux étapes principales : le prétraitement des workflows, pour supprimer tous les fragments du workflow qui n'ont pas besoin d'être exécutés ; et le provisionnement du cache, pour décider au moment de l'exécution quelles données intermédiaires doivent être mises en cache. Nous proposons un algorithme adaptatif de mise en cache qui tient compte des

variations des temps d'exécution des tâches et de la taille des données. Nous avons évalué notre solution en l'implémentant dans OpenAlea et en réalisant des expériences approfondies sur des données réelles avec une application complexe orientés-données de phénotypage de plantes.

Nos principales contributions sont i) une architecture SWMS pour gérer les algorithmes d'ordonnancement utilisant le cache lors de l'exécution de workflows dans des clouds monosites et multisites, ii) un modèle de coût qui inclut les coûts financiers et temporels, iii) deux algorithmes d'ordonnancement adapté au cache, en monosite et multisite clouds, et iv) une validation expérimentale sur une application de phénotypage de plantes orienté-données.

Abstract

In many scientific domains, such as bio-science, complex numerical experiments typically require many processing or analysis steps over huge datasets. They can be represented as scientific workflows. These workflows ease the modeling, management, and execution of computational activities linked by data dependencies. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive. In order to execute such workflows within a reasonable timeframe, they need to be deployed in a high-performance distributed computing environment, such as the cloud.

Plant phenotyping aims at capturing plant characteristics, such as morphological, topological, phenological features. High-throughput phenotyping (HTP) platforms have emerged to speed up the phenotyping data acquisition in controlled conditions (*e.g.* greenhouse) or in the field. Such platforms generate terabytes of data used in plant breeding and plant biology to test novel mechanisms. These datasets are stored in different geodistributed sites (data centers). Scientists can use a Scientific Workflow Management System (SWMS) to manage the workflow execution over a multisite cloud.

In bio-science, it is common for workflow users to reuse other workflows or data generated by other users. Reusing and re-purposing workflows allow the user to develop new analyses faster. Furthermore, a user may need to execute a workflow many times with different sets of parameters and input data to analyze the impact of some experimental step, represented as a workflow fragment, *i.e.*, a subset of the workflow activities and dependencies. In both cases, some fragments of the workflow may be executed many times, which can be highly resource-consuming and unnecessary long. Workflow re-execution can be avoided by storing the intermediate results of these workflow fragments and reusing them in later executions.

In this thesis, we propose an adaptive caching solution for efficient execution of data-intensive workflows in monosite and multisite clouds. By adapting to the variations in tasks' execution times, our solution can maximize the reuse of intermediate data produced by workflows from multiple users. Our solution is based on a new SWMS architecture that automatically manages the storage and reuse of intermediate data. Cache management is involved during two main steps: workflows preprocessing, to remove all fragments of the workflow that do not need to be executed; and cache provisioning, to decide at runtime which intermediate data should be cached. We propose an adaptive cache provisioning algorithm that deals with the variations in task execution times and the size of data. We evaluated our solution by implementing it in OpenAlea and performing extensive experiments on real data with a complex data-intensive application in plant phenotyping.

Our main contributions are i) a SWMS architecture to handle caching and cache-aware scheduling algorithms when executing workflows in both

monosite and multisite clouds, ii) a cost model that includes both financial and time costs for both the workflow execution, and the cache management, iii) two cache-aware scheduling algorithms one adapted for monosite and one for multisite cloud, and iv) and an experimental validation on a data-intensive plant phenotyping application.

Résumé Étendu

Introduction

Dans de nombreux domaines scientifiques, tels que la bioscience [130], la physique [8], l'astronomie [154], les expériences numériques complexes nécessitent généralement de nombreuses étapes de traitement ou d'analyse sur d'énormes ensembles de données. Elles peuvent être représentées comme des workflow scientifiques. Ces workflows facilitent la modélisation, la gestion et l'exécution d'activités de calcul liées par des dépendances de données. Les workflows sont généralement représentés sous la forme d'un graphe dirigé. Pendant l'exécution d'un workflow, les activités sont instanciées avec des données à traiter et des paramètres fixes, ce qui produit plusieurs tâches. Comme la taille des données traitées et la complexité du calcul ne cessent d'augmenter, ces workflows deviennent orientés données. Afin d'exécuter ces workflows dans un délai raisonnable, ils doivent être déployés dans un environnement informatique distribué à haute performance, tel que le cloud.

Le cloud computing offre un accès fiable et facile à la demande à des ressources informatiques, de stockage et de réseau virtuellement infinies. Les utilisateurs utilisent des services web pour accéder, gérer et déployer des tâches, telles que le stockage de données, le déploiement d'applications ou le calcul, sur de très grands centres de données. Un cloud est généralement composé de plusieurs sites géographiquement répartis (ou centres de données), chacun ayant ses propres ressources et données, et appelé cloud multisite. Cela permet d'exécuter des workflows orientés données sur plusieurs sites dans le cloud pour des raisons d'évolutivité ou de confidentialité.

Les scientifiques peuvent utiliser un système de gestion workflow scientifique (SWMS) pour gérer les exécutions de workflow sur un cloud monosite et multisite. De nombreux algorithmes d'ordonnacement ont été adaptés aux workflows orientés données executés sur un cloud monosite, notamment des heuristiques basées sur une matrice [170], des optimisations d'essaim de particules [121], et des optimisations basées sur des approches évolutives [87]. De nombreux algorithmes [114, 95] ont été proposés pour ordonnancer efficacement un workflow avec différentes granularités (tâche, activité, fragment). En général, ces solutions tiennent compte de l'emplacement des données, de la taille et de la disponibilité du stockage des sites, et des capacités de calcul pour ordonnancer le workflow en minimisant le coût global.

Il est courant que les utilisateurs de workflows réutilisent des fragments ou des données provenant d'autres workflows. L'adaptation et la réutilisation des workflows permet à l'utilisateur de développer plus rapidement de nouvelles analyses. En outre, un utilisateur peut avoir besoin d'exécuter un workflow plusieurs fois avec différents ensembles de paramètres et de données d'entrée pour analyser l'impact d'une étape expérimentale, représentée comme un

fragment de workflow, *i.e.*, un sous-ensemble des activités et des dépendances du workflow. Dans les deux cas, certains fragments du workflow peuvent être exécutés plusieurs fois, ce qui peut être très consommateur de ressources et inutilement long. La ré-exécution du workflow peut être évitée en mettant en cache les résultats intermédiaires generés par ces fragments et en les réutilisant dans des exécutions ultérieures.

Dans les systèmes informatiques, la mise en cache signifie le stockage de données dans un support de stockage dont l'accès est sensiblement plus rapide (*e.g.*, mémoire principale) que de recalculer les données ou de lire à partir d'un support de stockage plus lent (*e.g.*, disque). Habituellement, pour les exécutions de workflow, le cache doit être persistant et accessible par plusieurs utilisateurs. Dans le cloud, le stockage et le calcul ont tous deux un coût, et le défi de la mise en cache consiste à déterminer le meilleur compromis entre le coût du cache et le coût de la ré-exécution. Dans le cas d'un cloud multisite, le défi devient plus complexe car le cache peut être réparti entre les sites.

En raison de la forte latence du transfert de données entre sites, les données sont généralement traitées dans le site où elles se trouvent. Pourtant, il est fréquent que les scientifiques travaillant sur d'énormes ensembles de données effectuent des expériences en utilisant des données provenant de plusieurs centres de données. Par exemple, les données climatiques de la grille du système terrestre [161], les grandes quantités de données brutes de la chromodynamique quantique (QCD) [124] et les données du projet ALICE [1] sont toutes réparties géographiquement. Dans le cadre de la recherche sur le phénotypage, les données sont également stockées dans des centres de données du monde entier (comme le projet Phemome [2]). En outre, les données intermédiaires générées par des workflows orientés données peuvent être énormes et sont généralement stockées sur le site où elles ont été générées. En conséquence, les données (tant en entrée qu'en cache) nécessaires à l'exécution d'un workflow peuvent être géo-distribuées. Pour bénéficier efficacement de la mise en cache et de la réutilisation des données intermédiaires, un SWMS doit être adapté pour exploiter un cache distribué sur un cloud multisite.

Cette thèse a été réalisée dans le cadre d'un projet national, l'Institut de Convergences #DigitAg [3] sur l'agriculture numérique, et deux équipes associées de l'Inria avec le Brésil (LNCC, UFRJ, UFF et CEFET à Rio de Janeiro) : SciDISC [4] sur l'analyse des données scientifiques à l'aide de l'informatique évolutive à forte intensité de données et HPDaSc [5] sur la science des données de haute performance. Dans le projet #DigitAg, nous relevons le défi de l'exploitation des technologies de données géo-distribuées et à grande échelle pour le phénotypage des plantes. Le phénotypage des plantes vise à capturer les caractéristiques des plantes, telles que les caractéristiques morphologiques, topologiques, phénologiques [158]. La capture manuelle du phénotype d'une

[1] http://aliceinfo.cern.ch/Public/Welcome.html
[2] https://www.phenome-fppn.fr/
[3] https://www.hdigitag.fr/
[4] https://team.inria.fr/zenith/scidisc/
[5] https://team.inria.fr/zenith/hpdasc/

plante prend beaucoup de temps et produit une quantité limitée de données. Des plateformes de phénotypage à haut débit (HTP) ont vu le jour pour accélérer l'acquisition des données de phénotypage dans des conditions contrôlées (*par exemple* en serre) ou en champ [147]. Ces plateformes génèrent des téraoctets de données utilisées en sélection végétale et en biologie végétale pour tester de nouveaux mécanismes.

Chaque dataset est coûteux, long à générer et unique. En effet, il dépend de la croissance des plantes, des capacités de la plate-forme et de nombreux paramètres externes. Ainsi, les datasets sont très précieux et de nombreuses équipes de recherche différentes travaillent sur les mêmes ensembles de données, qui servent de référence. Ces équipes peuvent être situées sur le même site local ou sur des sites géographiques différents.

Les nouvelles expériences peuvent partager des données communes avec les précédentes et peuvent bénéficier de la réutilisation des données intermédiaires lorsqu'elles sont mises en cache. Cependant, comme les utilisateurs peuvent être situés sur des sites différents, les données en cache qu'ils produisent peuvent être géographiquement distribuées. Comme les expériences sont orientés données et qu'elles peuvent être exécutées sur un cloud monosite ou multisite, la gestion du cache et de l'odonnancement du workflow devient lié, ce qui devient un problème difficile.

Les objectifs de cette thèse sont les suivants :

1. proposer des stratégies efficaces de mise en cache et d'ordonnancement pour permettre le partage de données intermédiaires de grande taille entre les utilisateurs dans des architectures de cloud monosite et multisite, afin de réduire les temps d'exécution des workflows.

2. montrer les coûts et les avantages du partage de données intermédiaires volumineuses plutôt que la ré-exécution des activités qui les ont produites, entre plusieurs utilisateurs.

Cette thèse est composée de quatre chapitres principaux: état de l'art, cas d'utilisation en phenotypage à haut débit, gestion adaptative du cache en cloud monosite, et ordonnancement de workflow compatible avec le cache en cloud multisite. Un dernier chapitre conclut le manuscrit, résume les contributions et propose des directions de recherche futures.

État de l'art

Nous avons discuté de l'état actuel des connaissances en matière de gestion workflow scientifique sur le cloud, en mettant l'accent sur la mise en cache et l'ordonnancement. Nous avons tout d'abord présenté les concepts de base de la gestion des workflows, ainsi qu'une architecture fonctionnelle des SWMS. Ensuite, nous avons discuté du déploiement, de l'ordonnancement et de l'exécution des workflows sur un cloud monosite et multisite. Nous avons montré comment l'exécution des workflows peut bénéficier du cloud computing avec son coût réduit, sa facilité d'accès et d'utilisation, sa qualité de service et son élasticité. Dans un cloud multisite, le SWMS peut être

déployé à deux niveaux (intra-site et inter-site) pour utiliser efficacement les ressources des sites. Nous avons présenté les techniques de stockage des données utilisées sur le cloud, qui comprend des systèmes de fichiers à disque partagé et des systèmes de fichiers distribués pour le stockage des données d'exécution ainsi que des bases de données pour le stockage des métadonnées. Les techniques de base pour exécuter un workflow sur des ressources distribuées dans un cloud inclue la parallélisation et l'ordonnancement. Les différents types de techniques de parallélisation peuvent être exploités par le SWMS pour paralléliser les workflows. Les algorithmes d'ordonnancement peuvent être statiques, dynamiques ou hybrides.

Par la suite, nous avons discuté de la mise en cache des données et de l'ordonnancement des workflows avec des données mises en cache sur le cloud. Nous avons présenté une architecture SWMS de base qui comprend un gestionnaire de cache et une mémoire cache, qui sont nécessaires pour que le SWMS puisse utiliser un cache. Nous avons examiné les quatre principales approches de l'exploitation d'un cache par un SWMS. La première consiste à réutiliser les données générées au cours de l'exécution d'un seul workflow (workflows d'exploration de paramètres). La seconde consiste à stocker des données intermédiaires générées par un workflow pour sa ré-exécution future (smart rerun). La troisième consiste à stocker et à partager les données intermédiaires générées par des fragments de workflows pour de nouveaux workflows (workflows évolutifs). Le quatrième consiste à partager les données intermédiaires d'un workflows qui est exécuté par plusieurs utilisateurs en même temps. Enfin, nous avons analysé les techniques existantes pour l'ordonnancement et la gestion du cache SWMS sur un cloud monosite et multisite.

Cas d'utilisation: phenotypage à haut débit

Nous avons discuté des opportunités et des défis qui apportent le partage et la réutilisation des données intermédiaires pour la recherche sur le phénotypage à haut débit (HTP). Nous avons tout d'abord présenté le HTP et donné un aperçu des applications HTP avec des analyses orientés données et calcul. Nous avons présenté certaines des plateformes HTP, dont PhenoArch et les images de plantes qu'il génère. Nous avons présenté une infrastructure de calcul typique où les ensembles de données brutes sont stockés, analysés et utilisés dans des applications de phénotypage.

Ensuite, nous avons présenté le workflow Phenomenal, un workflow de phénotypage qui permet la reconstruction automatique de l'architecture végétale en 3D pour une série d'espèces et des mesures quantitatives sur les plantes reconstruites. Nous avons présenté OpenAlea, un SWMS largement utilisé pour la modélisation et l'analyse des plantes. Nous avons donné plus de détails sur la bibliothèque Phenomenal qui a été développée dans OpenAlea pour analyser et reconstruire la géométrie et la topologie de milliers de plantes à travers le temps dans diverses conditions. Cette bibliothèque est utilisée pour construire le workflow Phenomenal, qui sert de référence pour nos validations expérimentales. Enfin, nous avons montré comment

des ensembles de données intermédiaires peuvent être utiles dans d'autres applications, principalement la binarisation et la reconstruction 3D.

Gestion adaptative d'un cache pour SWMS en cloud monosite

Nous nous sommes attaqués au problème de la détermination des données intermédiaires à mettre en cache pendant l'exécution du workflow sur un cloud monosite. Dans ce travail, nous avons proposé une solution de mise en cache adaptative pour l'exécution efficace des workflows à orientés données sur un cloud monosite. Tout d'abord, nous avons proposé une nouvelle architecture SWMS qui gère automatiquement le stockage et la réutilisation des données intermédiaires sur un cloud monosite. Cette architecture est une extension d'une architecture de pointe, avec de nouveaux composants pour le stockage et la réutilisation des données mises en cache pendant l'exécution des workflows. L'architecture peut être décomposée de deux façons : i) en termes de couches fonctionnelles qui montrent les différentes fonctions et composants; et ii) en termes de nœuds et de composants qui interviennent dans le traitement des workflows. Les deux composants ajoutés aux couches fonctionnelles sont un *gestionnaire de cache* et un *index de cache*. Ces deux composants sont utilisés par le SWMS au cours de deux étapes de l'exécution des workflows : 1) le prétraitement du workflow, pour supprimer tous les fragments du workflow qui ne doivent pas être exécutés ; et 2) le provisionnement du cache, pour décider au moment de l'exécution quelles données intermédiaires doivent être mises en cache. Ensuite, pour déterminer quelles données intermédiaires seront mises en cache, nous avons conçu un modèle de coût à objectifs multiples qui inclut les coûts financiers et de temps. Le modèle de coût calcule un compromis entre le coût de stockage et le coût de régénération des données. Il inclut les temps d'écriture des données, les temps d'exécution, la topologie du workflow, la taille des données et le coût financier. Ensuite, sur la base du modèle de coût, nous avons proposé un algorithme qui gère dynamiquement le stockage et la réutilisation des données intermédiaires pendant l'exécution du workflow. L'algorithme est adaptatif en termes de variations des temps d'exécution des tâches et de la taille des données de sortie.

Nous avons mis en œuvre notre solution dans le SWMS OpenAlea et avons effectué des expériences approfondies sur les données de la plate-forme PhenoArch avec le workflow Phenomenal. Nous avons comparé trois méthodes d'approvisionnement du cache : sans cache, gourmande et adaptative. Notre évaluation expérimentale montre que la méthode adaptative permet de mettre en cache uniquement les données de sortie pertinentes pour les réexécutions ultérieures par d'autres utilisateurs, sans entraîner de coûts de stockage élevés pour le cache. Les résultats montrent que la mise en cache adaptative peut apporter des gains de performance importants, jusqu'à un facteur de 3,5 avec 6 réexécutions du workflow.

Ordonnancement de workflow compatible avec le cache en cloud multisite

Nous avons abordé le problème de l'ordonnacement et de la gestion du cache pour l'exécution de workflow sur cloud multisites. L'objectif principal était d'exécuter efficacement des workflow orientés données sur un cloud multisite, en utilisant la mise en cache des données intermédiaires produites par les workflows précédents. Tout d'abord, nous avons adapté l'architecture SWMS pour un cloud multisite. L'architecture est légèrement différente de celle proposée pour un cloud monosite. L'architecture pour les cloud multisites permet de prendre des décisions à la fois dans un même site et entre plusieurs sites. Au niveau inter-site, l'ordonnanceur global et le gestionnaire de cache utilisent la localité des données, les différentes ressources des sites et la bande passante pour leurs décisions. Au niveau intra-site, le SWMS gère l'ordonnancement et l'exécution des fragments de workflow. Ensuite, nous avons détaillé notre modèle de coût pour représenter les différents coûts en temps lors de l'exécution d'un workflow avec réutilisation de données et mise en cache sur un cloud multisite. Le modèle de coût comprend le coût en temps nécessaire pour écrire, lire, mettre en cache les données d'entrée et les données intermédiaires, et le temps nécessaire pour calculer les fragments du workflow sur chaque site. Sur la base du modèle de coût, nous avons conçu une décision globale qui minimise le coût total de l'exécution d'un fragment du workflow, en sélectionnant les données qui seront mises en cache, le site où elles le seront et le site où le fragment sera exécuté. Troisièmement, nous avons présenté un algorithme d'ordonnancement dynamique qui utilise la décision globale pour ordonnancer les fragments du workflow sur différents sites.

Nous avons évalué notre solution en exécutant le workflow Phenomenal sur un cloud multisite composé de trois sites en France : Montpelier, Lyon et Lille. Nous avons mis en œuvre notre solution dans le SWMS OpenAlea. Nous avons comparé notre solution avec deux algorithmes de base que nous avons étendus pour exploiter notre architecture de mise en cache. Les résultats expérimentaux montrent que notre solution peut apporter des gains de performance importants, réduisant le temps total jusqu'à 42 % avec 60 % de données d'entrée identiques pour chaque nouvelle exécution.

Conclusion

Dans cette thèse, nous avons abordé le problème de la gestion distribuée des workflows scientifiques pour le phénotypage à haut débit (HTP) des plantes, avec pour objectif de réduire le temps d'exécution. À cette fin, nous avons proposé de nouvelles architectures SWMS avec des stratégies efficaces de mise en cache et d'ordonnancement pour permettre le partage de données intermédiaires importantes entre les utilisateurs dans des environnements de cloud monosite et multisite. Les architectures et les algorithmes que nous avons proposés sont différents sur un cloud monosite et multisite. Dans un cloud

monosite, le SWMS comprend un composant de gestion du cache qui gère la mise en cache et la réutilisation des données intermédiaires. Avant l'exécution du workflow, le gestionnaire de cache simplifie le workflow en fonction de sa topologie et des données de cache existantes. Pendant l'exécution, le gestionnaire de cache gère les décisions concernant les données intermédiaires qui seront mises en cache à l'aide d'un algorithme adaptatif. L'algorithme utilise un modèle de coût pour déterminer le meilleur compromis entre le coût de stockage des données en cache et le coût de ré-exécution. Dans un cloud multisite, le SWMS comprend à la fois un gestionnaire de cache et un algorithme d'ordonnancement qui équilibre la charge de travail et les données du cache entre les sites. L'ordonnancement et la gestion du cache sont basées sur un algorithme dynamique qui minimise le coût total en temps de l'exécution du workflow et des transferts de données du cache. Pour évaluer notre solution de gestion du cache et de l'ordonnancement du workflow, nous avons implémenté nos algorithmes dans OpenAlea. Pour les expériences, nous avons utilisé le workflow Phenomenal, un workflow de phénotypage complexe orienté données.

Nous avons comparé nos algorithmes avec des algorithmes de référence sur un cloud monosite et multisite. Nos résultats montrent que nos approches réduisent le temps total (temps d'exécution et temps de transfert des données) par rapport aux algorithmes de référence. De plus, l'architecture que nous proposons est capable de mettre en cache et de partager automatiquement les données intermédiaires entre les utilisateurs.

Nous présentons plusieurs direction de recherche futures qui permettraient d'améliorer les travaux sur la mise en cache de données avec des exécutions de workflows scientifiques:

- **Apprentissage automatique pour la gestion du cache.** L'apprentissage automatique (ML) fournit des techniques qui permettent d'"apprendre" des calculs et des décisions du cache précédents afin de générer de nouvelles décisions plus précises et plus raffinées. Les techniques de ML proposent des solutions qui prennent plus d'informations que notre compromis pour la décision de mise en cache, telles que la topologie du workflow, et les métadonnées du workflow, y compris la provenance. Ces techniques présentent également de bons résultats dans la gestion du cache [133, 152, 139]. Ainsi, le ML pourrait améliorer les décisions relatives au cache lors de l'exécution du workflow.

- **Cache chaud et froid.** Le gestionnaire de cache proposé dans cette thèse ne prend en compte qu'un cache sur disque. Une partie des données du cache n'est presque jamais réutilisée mais vaut quand même le coût de rester dans le cache. D'autre part, certaines des données mises en cache sont très souvent réutilisées par plusieurs utilisateurs. Ces données "chaudes" gagneraient à être plus accessibles, dans un cache "plus rapide", tel qu'un cache en mémoire. Plusieurs travaux dans la communauté des bases de données ont étudié les avantages de la mise en cache à chaud et à froid, qui pourrait être utilisée avec la mise en cache de données de workflow [29, 23, 90]. Par exemple, nous pourrions

stocker le cache "chaud" dans la mémoire principale, et le cache "froid" dans le stockage sur disque.

- **Workflows cycliques et opérateurs algébriques.** Le travail présenté dans cette thèse est adapté aux workflows acycliques, qui sont les plus courants. Cependant, dans plusieurs expériences où les analyses de plantes sont mélangées à des simulations, les workflows sont cycliques ou utilisent des opérateurs algébriques [128], ce qui apporte de nouveaux défis en matière de mise en cache des données. En effet, dans les workflows algébriques, la même activité produira des données de sortie différentes selon l'état du cycle, ce qui rend difficile la mise en cache et la réutilisation des données.

- **Coûts environnementaux.** Notre modèle de coût tient compte de deux objectifs : la minimisation du coût de fabrication et le coût financier. Minimiser le coût environnemental devient une question essentielle et pourrait être intégré à la décision de mise en cache. Dans un cloud hétérogène et multisite, chaque site a des coûts environnementaux différents tant pour le stockage que pour le calcul. Le coût environnemental de l'écriture et de la lecture d'un disque est généralement beaucoup moins élevé que les coûts de calcul et d'allocation de la mémoire. Pourtant, le cache en mémoire peut être extrêmement coûteux et le gain d'accélération lié à la réutilisation des données rend complexe l'estimation du compromis entre le coût de régénération des données et le coût de stockage. Ainsi, le coût environnemental pourrait fournir une solution différente pour l'ordonnancement des workflows et la distribution des données du cache, qui devrait être envisagée pour des expériences plus écologiques.

Contents

Résumé iii

Abstract v

Résumé Étendu vii

1 Introduction 1
1.1 Motivations . . . 1
1.2 Thesis Context . . . 2
1.3 Contributions . . . 4
1.4 Organization of the Thesis . . . 7

2 State of the Art 9
2.1 Overview and Motivations . . . 9
2.2 Workflow Management . . . 11
2.2.1 Basic Concepts . . . 11
2.2.2 Architecture of SWMSs . . . 12
2.3 Scientific Workflow Management in the Cloud . . . 17
2.3.1 Cloud Computing . . . 18
2.3.2 Multisite Cloud . . . 20
2.3.3 Data Storage . . . 21
2.3.4 Workflow Scheduling in the Cloud . . . 25
2.4 Data Caching for Workflows . . . 29
2.4.1 Data Caching in Computer Systems . . . 30
2.4.2 Data Caching in SWMSs . . . 32
2.4.3 Scheduling and Caching in SWMSs . . . 39
2.5 Conclusion . . . 42

3 Use Case in Plant Phenotyping 45
3.1 High-throughput Plant Phenotyping . . . 45
3.1.1 Context . . . 45
3.1.2 High-Throughput Phenotyping Platforms . . . 47
3.1.3 Infrastructures . . . 48
3.2 Automatic Phenotyping in OpenAlea . . . 50
3.2.1 OpenAlea . . . 51
3.2.2 Phenomenal Library . . . 51
3.2.3 Applications with Common Data . . . 55
3.3 Conclusion . . . 55

4 Adaptive Caching for Workflows in Monosite Cloud **59**
4.1 Introduction 59
4.2 Related works 61
4.3 Monosite Cloud SWMS Architecture 62
4.4 Cost Model 65
4.5 Cache Management 67
4.6 Experimental Validation 69
4.6.1 Experimental Setup 69
4.6.2 Experiments 70
4.6.3 Discussion 79
4.7 Conclusion 80

5 Cache-Aware Scheduling for Workflows in Multisite Cloud **83**
5.1 Introduction 84
5.2 Use Case: Intermediate Data Reuse in Geo-dis- tributed Clouds 85
5.3 Related Works 86
5.4 Multisite SWMS Architecture 89
5.5 Cache-aware Workflow Execution 91
5.5.1 Distributed Workflow Execution Overview 92
5.5.2 Cache Data Selection 93
5.5.3 Cache Site Selection 93
5.5.4 Execution Site Selection 94
5.5.5 Global Decision 94
5.5.6 Cache-Aware Scheduling 95
5.6 Experimental Validation 97
5.6.1 Experimental Setup 97
5.6.2 Experiments 98
5.6.3 Discussion 108
5.7 Conclusion 109

6 Conclusion **111**
6.1 Contributions 111
6.2 Directions for Future Work 114

Bibliography **117**

List of Figures

2.1 Generic architecture of a SWMS [93]. 13
2.2 Multisite workflow execution architecture. Dotted lines represent inter-site interactions [94]. 20
2.3 Strategies for geographically distributed metadata management [126]. 25
2.4 Different types of parallelism. 27
2.5 Generic architecture of a SWMS with cache management. . . . 33
2.6 Parameter Sweep Workflow. 34
2.7 Smart rerun: a workflow executed twice, the second execution benefits from cached data generated by the first execution. . . 35
2.8 Evolving workflow: execution of two workflows *Wf1* and *Wf2*, with *Wf2* being designed on top of *Wf1*. 37
2.9 Multi-user SWMS architecture with a *Data management* module that enable intermediate data reuse between users [32]. 38
2.10 Kepler architecture in monosite cloud with cache feature [32]. 40
2.11 Three level cache architecture in multisite cloud [132]. 42

3.1 Map of Emphasis project platforms across Europe. 48
3.2 Time series of images for one plant, generated during one HTP experiment. 49
3.3 PhenoArch platform. 50
3.4 Phenomenal workflow and its generated intermediate data. . 52
3.5 Intermediate data generated by each fragment of the Phenomenal workflow on one genotype through time. 53
3.6 3D reconstruction of (A) Cotton (Gossypium); (B) Apple tree (Malus pumila); (C) Sorghum (Sorghum bicolor). 53
3.7 Order of magnitude of intermediate data size and execution time of each activity of the Phenomenal workflow executed on one HTP experiment dataset. 54
3.8 Four different workflows that use components of the Phenomenal library. 1) the Phenomenal workflow [12]; 2) a light competition workflow [26]; 3) a light interception and radiation workflow [31]; and 4) a maize ear detection workflow [20] . . 56

4.1 SWMS Functional Architecture 63
4.2 SWMS Technical Architecture 64
4.3 DAG of tasks before preprocessing (left) and the selected fragments that need to be executed (right). 67
4.4 Speedup versus number of vCPUs: without cache (orange), greedy caching (blue), and adaptive caching (green). 71

4.5 Execution time of each activity's task. 73
4.6 p_{min} of each activity's task. 74
4.7 Monetary cost depending on the number of users that execute the workflow with three different cache strategies with the execution cost (blue) and the storage cost (red). 74
4.8 Monetary cost of scenario S1: Each user executes the last activity of the workflow, for three values of p_{tresh}. 75
4.9 Monetary cost of scenario S2: Each user executes all the activities of the workflow, for three values of p_{tresh}. 77
4.10 Monetary cost of scenario S3: Each user executes the activities based on the use case (A1, A3, A7, A9), for three values of p_{tresh}. 77

5.1 Two workflows in plant analysis 86
5.2 Phenomenal plant analysis workflow 87
5.3 Workflow System Functional Architecture 90
5.4 Multisite Workflow System Architecture 91
5.5 Total time of Phenomenal workflow execution in four cases . . 99
5.6 Centralized versus distributed cache in terms of execution time for three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*) 102
5.7 Total times for multiple users (60% of same raw data per user) for three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*) 103
5.8 Execution for one user (60% of same raw data used) on heterogeneous sites with three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*) 105
5.9 Four subworkflows derived from the Phenomenal workflow . 106
5.10 Total times for executing the four subworkflows by two users (with 60% of same raw data for second user) 107

List of Tables

4.1 Percentage of caching decision per task 76
5.1 Scheduling algorithms and their main dimensions 98

Chapter 1

Introduction

1.1 Motivations

In many scientific domains, such as bio-science [130], physics [8], astronomy [154], complex numerical experiments typically require many processing or analysis steps over huge datasets. They can be represented as scientific workflows. These workflows ease the modeling, management, and execution of computational activities linked by data dependencies. Workflows are typically represented as a directed graph. During a workflow execution, the activities are instantiated with some data to process and some fixed parameters, producing several tasks. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive. To execute such workflows within a reasonable timeframe, they need to be deployed in a high-performance distributed computing environment, such as the cloud.

Cloud computing provides reliable and on-demand easy access to virtually infinite computing, storage, and networking resources. Users use web services to access, manage, and deploy tasks, such as data storage, application deployment, or computation, over very large data centers. A cloud is typically composed of several geo-distributed sites (or data centers), each with its own resources and data, and called multisite cloud. This makes it possible to execute data-intensive workflows on several cloud sites for scalability or privacy reasons.

Scientists can use a Scientific Workflow Management System (SWMS) to manage workflow executions over a monosite and multisite cloud. Many scheduling algorithms have been adapted for data-intensive workflows in monosite cloud, including matrix-based heuristics [170], particle swarm optimizations [121], and optimizations based on evolutionary approaches [87]. Many algorithms [114, 95] have been proposed to efficiently schedule a workflow with different granularities (task, actity, fragment). In general, these solutions consider the data location, the sites' storage size and availability, and computing capacities to schedule the workflow minimizing the overall cost.

It is common for workflow users to reuse other workflows or data generated by other workflows. Reusing and re-purposing workflows allow the user to develop new analyses faster. Furthermore, a user may need to execute a workflow many times with different sets of parameters and input data to

analyze the impact of some experimental step, represented as a workflow fragment, *i.e.*, a subset of the workflow activities and dependencies. In both cases, some fragments of the workflow may be executed many times, which can be highly resource-consuming and unnecessary long. Workflow re-execution can be avoided by caching the intermediate results of these workflow fragments and reusing them in later executions.

In computer systems, caching means storing data in a storage media whose access is significantly faster (*e.g.*, main memory) than recomputing the data or reading from a slower storage media (*e.g.*, disk). Usually, for workflow executions, the cache has to be persistent and accessible by multiple users. In a cloud environment, both the storage and computation have a cost, and a caching challenge is to determine the best trade-off between cache cost and re-execution cost. In the case of a multisite cloud, the challenge becomes more complex as the cache can be distributed among the sites.

Due to high latency in inter-site data transfer, usually, the data is processed on the site where it is located. Yet, it is common for scientists working on huge datasets to perform experiments using data from several data centers. For instance, the climate data in the Earth System Grid [161], large amounts of raw data from Quantum Chromodynamics (QCD) [124] and the data of the ALICE project [1] are all geographically distributed. In phenotyping research, the data is also stored in data centers all across the world (such as the Phemome project [2]). Moreover, the intermediate data generated by data-intensive workflows can be huge and usually stored on the site where it was generated. As a consequence, the data (both input and cached) required for a workflow execution can be geo-distributed. To efficiently benefit from caching and reusing intermediate data, a SWMS should be adapted to exploit a distributed cache over a multisite cloud.

1.2 Thesis Context

This thesis has been done in the context of one national project, the Institut de Convergences #DigitAg [3] on digital agriculture, and two Inria associated teams with Brazil (LNCC, UFRJ, UFF and CEFET in Rio de Janeiro): SciDISC [4] on scientific data analysis using data-intensive scalable computing and HPDaSc [5] on high performance data science.

In the #DigitAg project, we address the challenge of exploiting geo-distributed and big data technologies for plant phenotyping. Plant phenotyping aims at capturing plant characteristics, such as morphological, topological, phenological features [158]. Manually capturing the phenotype of a plant is time-consuming and produces a limited amount of data. High-throughput phenotyping (HTP) platforms have emerged to speed up the phenotyping data acquisition in controlled conditions (*e.g.* greenhouse) or in the field [147].

[1] http://aliceinfo.cern.ch/Public/Welcome.html
[2] https://www.phenome-fppn.fr/
[3] https://www.hdigitag.fr/
[4] https://team.inria.fr/zenith/scidisc/
[5] https://team.inria.fr/zenith/hpdasc/

Such platforms generate terabytes of data used in plant breeding and plant biology to test novel mechanisms.

Each dataset is expensive and time consuming to generate and it is unique. Indeed, it depends on the plant growth, platform capacities, and many external parameters. Thus, the datasets are very valuable and many different research teams (users) work on the same datasets, which serve as baselines. These teams can be located at the same local site or at different geographically sites.

New experiments may share common data with previous executions, and can benefit from reusing intermediate data whenever it is cached. However, since the users may be located at different sites, the cache data they produce may be geo-distributed. As the experiments are data-intensive and can be executed in either monosite or multisite cloud, the management of the cached data involves scheduling workflow fragments or activities, which becomes a challenging problem.

The objectives of this thesis are to:

1. propose efficient caching and scheduling strategies to enable sharing of intermediate big data among users in both monosite and multisite cloud architectures, to reduce workflow execution times.

2. show the costs and benefits of sharing intermediate big data rather than re-executing the activities that produced it, between multiple users.

In particular, we consider the following questions for executing workflows in monosite and multisite cloud using a cache:

- **How to determine which data should be stored?** Caching intermediate data requires both CPU time, to serialize the data on disk, and storage, to maintain the data on disk. As the latency to access data increases with the cache size, the cache can become inefficient, with slow access to cache data. Thus, saving all the intermediate data that is generated would be both expensive in terms of storage and inefficient in terms of access time. Furthermore, due to their dependencies with in a workflow, all intermediate data do not have the same importance in a workflow re-execution. The data generated by the input activities can be reused in more cases than the one generated by the output activities. Thus, to determine which data generated should be stored, the SWMS should take into account many parameters including the data size, its generation time, and its efficiency for latter execution in order to find the best trade-off between the costs to store data and the benefits of reusing it.

- **How to determine where to store the cached data?** A multisite cloud is said homogeneous if the sites (data centers) have same computing and storage capabilities. In an heterogeneous multisite cloud, with much variation in the sites's capabilities, determining where to store the cached data is difficult. A site with a lot of computing resources is likely to be used a lot in the workflow execution, leading to a lot of the intermediate data being generated at this site. In that case, as the

intermediate data is huge, the time to transfer the newly generated data to another site should be taken into consideration when determining the best site to cache the data. Furthermore, as data transfers within a site (across cluster nodes) are significantly faster than across the sites, the future workflow executions will be privileged to be executed on the sites that store the cached data. Thus, simply storing all the cached data at the cheapest site would result in a bottleneck in both computation and bandwidth to access the cached data at that site.

- **How to efficiently schedule a workflow to benefit from the cached data?** Reusing previously generated data instead of regenerating it can save time and resources. But due to the complexity and size of the data processed, this is not always true. Indeed, transferring huge cached data to another site for computation might be more time consuming that regenerating the data at that other site. However, due to the time of transferring cached data, it can be more time-efficient to schedule an activity execution at a site with less computing power than at a site that could not be able to store the cached data. Thus, in order to schedule the workflow in a way that minimizes the overall cost, the SWMS should use the cached data location, the decision on which data should be stored, and the sites' storage and computing capacities.
- **Which SWMS architecture should be used for both cache management and workflow scheduling in clouds?** The SWMS is managing the workflow scheduling and execution in the cloud. The management of the cached data impacts the workflow execution both when the cached data is written and read. Thus, the SWMS should integrate cache management with the workflow scheduling and execution. A new SWMS architecture is required for workflow execution with a cache in a cloud environment. Moreover, in a multisite cloud, both the cloud sites' resources and the cached data are distributed, making the workflow scheduling problem more complex. Thus, in a multisite cloud, the SWMS should additionally consider the distribution of the resources and the cached data location.

1.3 Contributions

The main objective of this thesis is to efficiently execute several workflows, that share common fragments, by using a cache system in either monosite or multisite cloud.

The main contributions are:

- **A SWMS architecture to handle caching and cache-aware scheduling algorithms when executing workflows in either monosite or multisite cloud.** The SWMS architecture is an extension of a state-of-the-art architecture, with new components for storing and reusing cached data during workflow execution. The architecture includes a *cache manager*

and a *cache index*. The cache manager determines which intermediate data will be stored. In multisite cloud, it also determines at which site the data should be stored. The cache manager also selects which intermediate data will be reused instead of be re-generated. The cache index is a metadata store on the cached data.

- **A cost model that includes both financial and time costs for both workflow execution and cache management.** In an heterogeneous multisite cloud, the cost of task execution and cache storage may greatly vary between the sites. The cost model includes data transfer times, execution times, workflow topology, data sizes, sites' resources and availability, and financial cost. The cost model is used to compute a trade-off between storage cost and data re-generation cost, which is used in turn for the decisions on caching and scheduling at the monosite and multisite levels. Yet, the cost model is different in the case of monosite versus multisite cloud. In monosite cloud, it combines financial and execution time cost. In multisite cloud, the time to read, write, and transfer the cached data is added to the execution time.

- **Cache-aware scheduling algorithms for monosite and multisite clouds.** To optimize workflow execution with cached data, we propose two scheduling algorithms. The first one is designed for monosite cloud and optimizes the scheduling depending on the cache decisions made by the cache manager. This algorithm dynamically adapts to the tasks' variation in execution time and data sizes. The second algorithm is for multisite cloud. It makes scheduling decisions based on three aspects: which data should be cached, where the cached data should be stored, and where the workflow's tasks should be executed. It optimizes the overall cost of the workflow execution with cached data.

- **OpenAlea in the cloud.** OpenAlea was not adapted for cloud execution. We developed an extension to OpenAlea which enables the execution of workflows in monosite and multisite cloud. It now considers persistent caching and reusing data when executing workflows. In monosite cloud, the extension manages the workflow scheduling over the cluster nodes at a site. The cached data is reused whenever possible and stored in the site nodes. In multisite cloud, both the input and cached data can be distributed in the cloud sites. The extension manages the workflow execution at two levels: inter-site level (between the sites), and intra-site level (inside a site). At the inter-site level, OpenAlea manages the partitioning, cache decisions, and scheduling of workflow fragments. At the intra-site level, OpenAlea manages the execution of workflow fragments and storage of new cached data in the site nodes.

- **An experimental validation on a data-intensive plant phenotyping application.** To evaluate our solution for cache management and workflow scheduling, we implemented our algorithms in OpenAlea. For

the experiments, we use the Phenomenal workflow, a complex data-intensive phenotyping workflow, on real data. We measure the execution time, data transfer time, latency, the time to feed and use the cache. We compare our algorithms with baseline scheduling algorithms in monosite and multisite cloud. Our results show that our approaches reduce the total time (execution time and data transfer time) compared with baseline algorithms. Moreover, our proposed architecture is able to successfully cache and share the intermediate data automatically between users.

All these contributions have been published in the following publications:

- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Cache-aware Scheduling of Scientific Workflows in Multisite Cloud. Future Generation Computer Systems, under revision, 2020.
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Efficient Execution of Scientific Workflows in the Cloud Through Adaptive Caching. Transactions on Large-Scale Data-and Knowledge-Centered Systems (pp. 41-66), 2020.
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Distributed Caching of Scientific Workflows in Multisite Cloud. DEXA 2020 : International Conference on Database and Expert Systems Applications (pp. 51-65). **Best Paper Award.**
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Cache-aware scheduling of scientific workflows in multisite cloud. BDA 2020 : Gestion de Données – Principes, Technologies et Applications.
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Efficient Execution of Scientific Workflows in the Cloud Through Adaptive Caching. BDA 2019 : Gestion de Données – Principes, Technologies et Applications (pp. 41-66).
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Adaptive Caching for Data-Intensive Scientific Workflows in the Cloud. DEXA 2019 : International Conference on Database and Expert Systems Applications (pp. 452-466).
- C. Pradal, S. Cohen-Boulakia, G. Heidsieck, E. Pacitti, C. Pradal, F. Tardieu, & P. Valduriez. Distributed Management of Scientific Workflows for High-Throughput Plant Phenotyping. ERCIM News 2018, Smart Farming (pp.36-37)

And the following posters:

- item G. Heidsieck. Smart Reuse of High-Throughput Phenotyping Data using Scientific Workflows on the Cloud. ICROPM 2020: International Crop Modelling Symposium, France, Montpellier, 03 Feb. 2020.

- G. Heidsieck. Efficient Execution of Scientific Workflows in the Cloud through Adaptive Caching. ICROPM 2019: International Crop Modelling Symposium, France, Montpellier, 04 July 2019.
- G. Heidsieck. Smart rerun of data intensive scientific workflow in distributed environment - Rencontre Annuelle MaDICS, France, Strasbourg, 20 Nov. 2018.
- G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, & F. Tardieu. Efficient Re-execution of Data Intensive Scientific Workflows for High-throughput Phenotyping. BDA 2018 : Gestion de Données – Principes, Technologies et Applications, Romania, Bucharest

1.4 Organization of the Thesis

The rest of the thesis is organized as follows:

Chapter 2: Survey of scientific workflow management in the cloud. This chapter surveys scientific workflow management in the cloud, with a focus on data caching and workflow scheduling. We start by introducing the basic concepts of workflow management, within a functional architecture of SWMSs. Then, we present the general techniques for workflow parallelisation, *i.e.* coarse-grained parallelism and fine-grained parallelism, and workflow scheduling, *i.e.* static, dynamic, or hybrid. Then, we focus on scientific workflow management in both monosite and multisite cloud, including data storage and workflow execution. Based on an extension of the general architecture for SWMSs to deal with data caching, we introduce techniques for scheduling workflows with cached data in monosite and multisite cloud. Finally, we analyze the limitations of the existing approaches.

Chapter 3: Use case in plant phenotyping. In this chapter, we present a real use case in plant phenotyping based on the Phenomenal workflow from the OpenAlea SWMS. This use case is the basis for our motivations in this thesis and will be used in our experimental validation. We start with an overview of HTP and present some challenges. Then, we introduce the phenotyping modules of OpenAlea, in particular, the Phenomenal library, a set of components for automatic plant phenotyping from images. Then, we describe the Phenomenal workflow, a workflow created from the Phenomenal library components. Finally, we present different published analyses that use Phenomenal components, and discuss the potential benefits of caching intermediate data.

Chapter 4: Adaptive caching in monosite cloud.
In this chapter, we propose an adaptive caching solution for data-intensive workflows in monosite cloud. Our solution is based on a new SWMS architecture that automatically manages the storage and reuse of intermediate data and adapts to the variations in task execution times and output data size. We

evaluated our solution by implementing it in OpenAlea and performing extensive experiments on real data from the Phenomenal workflow. The results show that adaptive caching can yield major performance gains, e.g., up to a factor of 4.5 with 6 workflow re-executions.

Chapter 5: Cache-aware scheduling in multisite cloud. In this chapter, we propose a solution for cache-aware scheduling of scientific workflows in multisite cloud. Our solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection, and dynamic workflow scheduling. We implemented our solution in OpenAlea , together with cache-aware distributed scheduling algorithms. Our experimental evaluation in a three-site cloud with real data from the Phenomenal workflow shows that our solution can yield majors performance gains, reducing total time up to 42% with 60% of the same input data for each new execution.

Chapter 6: Conclusion. This chapter concludes the thesis, summarizing and discussing our contributions. We also give some future directions of research, based on the results of this thesis.

Chapter 2

State of the Art

In many scientific domains, complex experiments typically require many processing or analysis steps over huge quantities of data. They can be represented as scientific workflows, or workflows for short, which ease the modeling, management, and execution of computational activities linked by data dependencies. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive [81], thus requiring execution in a high-performance distributed and parallel environment, *e.g.*, a large-scale virtual cluster in a cloud [80]. These experiments often require executing again the same workflows for various reasons, including changing workflow parameters, sharing intermediate results, and upgrading an existing analysis. The intermediate data generated by workflow executions can be shared and/or reused by users using data caching in order to reduce the amount of recomputation during new workflow executions. For data-intensive workflows, caching intermediate data can require much time to save and load. And reusing some existing cached data is not always cost-effective. Thus, there is a difficult trade-off between reusing cached data and recomputing intermediate data. Furthermore, in a distributed environment where cached data need be transferred across sites, it becomes crucial to schedule the workflow based on cache location. The workflow execution and design are usually managed by a Scientific Workflow Management System (SWMS), which becomes then in charge of the caching.

This chapter gives the state of the art in scientific workflow management in the cloud, with a focus on caching and scheduling.

Section 2.1 presents the motivations of the chapter. Section 2.2 gives an overview of scientific workflow management, including basic concepts and architectures. Section 2.3 focuses on workflow management in both monosite and multisite clouds. Section 2.4 discusses data caching in SWMSs. Section 2.4.3 discusses data caching and workflow scheduling in SWMSs. Section 2.5 concludes this chapter.

2.1 Overview and Motivations

In many scientific domains, *e.g*, bio-science [81], complex numerical experiments typically require many processing or analysis steps over huge datasets. They can be represented as workflows. These workflows ease the modeling,

management, and execution of computational activities linked by data dependencies. Workflows are typically represented as a directed graph. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive. In order to execute such workflows within a reasonable time, they need to be deployed in a high-performance distributed computing environment, such as the cloud.

A SWMS is a tool to manage workflows, its execution, the datasets consumed and produced. SWMSs have been developed in various computing environments, both centralized and distributed. SWMSs such as Pegasus [47], Swift [178], Kepler [8], Taverna [112], Galaxy [61], Chiron [111] or OpenAlea [129] are used intensively by various research communities, *e.g.* astronomy, biology, computational engineering.

SWMSs can be represented with a five layer architecture [93], where a scientific workflow is first transformed into a workflow execution plan (WEP) to be scheduled and executed. These layers are: user interface layer, user services layer, WEP generation layer, WEP execution layer, and infrastructure layer. These layers represent the common features between most SWMSs throughout the workflow life cycle. The related works presented in this thesis are based on this architecture.

In a distributed environment, it is critical to exploit parallelism in order to execute data- and compute-intensive workflows in a reasonable time. Thus, the SWMS needs to manage workflow parallelization and scheduling using various types of parallelism and scheduling strategies.

Cloud computing provides reliable and on-demand easy access to virtually infinite computing, storage, and networking resources. Users use web services to access, manage, and deploy tasks, such as data storage, application deployment, or computation, over very large data centers. A cloud is typically composed of several geo-distributed sites (or data centers), each with its own resources and data. This makes it possible to execute data-intensive workflows on several cloud sites, for scalability or privacy reasons.

There are several surveys on workflow management. Bux *et al.* [24] focus on the parallelization techniques and their implementations in distributed environments: cluster, grid and cloud. Bux *et al.* also provide an overview of how SWMSs manage parallelism, and propose to group SWMSs into three classes. The first one is textual workflow languages, which can distribute workflow tasks over external resources, but lack support for data parallelism and they are not user friendly for the domain scientists. The second one is graphical standalone systems. They are easy for scientists to use but lack efficiency when dealing with parallelism. The third is life science enactment portals, that ease the design and sharing of workflows over a web browser. Yet, they are not optimized for workflow parallelism.

Rodriguez *et al.* [136] propose a taxonomy to focus on the features of workflows related to clouds and provide an overview of the scheduling algorithms for workflows in a cloud. The taxonomy is based on three categories: application model, scheduling model, and resource model. The application model defines the ability to schedule either a single or multiple workflows. The scheduling model defines the features of mapping workflow tasks to

virtual machines, the resources provisioning, the scheduling objective, and the optimization strategy. Finally, the resource model defines the cloud environment resources, the cloud provider features, the storage and networks, and the virtual machines. This taxonomy aims at describing all useful features for SWMSs execution in a cloud.

Mattoso *et al.* [103] propose a taxonomy to identify the main concepts related to address dynamic workflow steering, *i.e.* dynamically monitoring the execution of the workflow. The three essential categories for workflow steering are: monitoring the workflow execution, data analysis at runtime, and dynamic workflow adaptation. The taxonomy proposed considers three more categories. Thus, it is composed of six categories: monitoring, analysis, adaptation, notification, interface for interaction, and computing model. These categories consider all the parameters to include the user in the workflow execution. Mattoso *et al.* also present an overview of how the main SWMSs deal with each category of the taxonomy.

Liu *et al.* [93] provide an overview of data-intensive workflow management in SWMSs and their parallelization techniques. The overview includes a functional architecture that represents the common features between most SWMSs, and a comparative analysis of the existing solutions based on the proposed architecture. The architecture will be described in Section 2.2.2 as it serves as a baseline for our SWMS architecture.

These surveys provide a good basis to understand workflow management and execution in general. In this chapter, we introduce workflow management in distributed environments, with a focus on caching and scheduling of data-intensive workflows.

2.2 Workflow Management

This section introduces the basic concepts of workflows and SWMSs used throughout this thesis. First, we define scientific workflows and SWMSs. Then, we present a functional architecture for SWMSs.

2.2.1 Basic Concepts

This section introduces the concepts of scientific workflows and SWMSs.

Scientific Workflows

A workflow consists of a set of processes, called activities, that are linked by data or logical dependencies according to a set of semantic rules. Workflows are either business workflows or scientific workflows.

Scientific workflows are used to manage, model, and execute scientific experiments. A scientific workflow is the assembly of complex sets of scientific data processing activities with data dependencies between them [49]. Workflows can be combined, thus a scientific workflow can be composed of sub-workflows. Sub-workflows are composed of a subset of activities and dependencies of the scientific workflow. Sub-workflows typically represent

specific steps in data processing. Workflows can be represented in various ways. A common one is a directed graph, in which nodes represent the processing activities, and the edges represent the dependencies. In many applications, the graph is acyclic (DAG), or even a pipeline (sequence of activities). Directed Cyclic Graphs (DCG) present a challenge in scientific workflow executions since it brings logical dependencies or activities, *e.g.* with a whiledo construct [166].

An activity is a description of a piece of work and can be a computational script (computational activity), some data (data activity), or some set-oriented algebraic operator like map or filter [110]. During the execution of a scientific workflow, an activity is instantiated with some data to process and some fixed parameters, producing several tasks. A task is the one-time execution of an activity, processing some data. Sometimes, "jobs" are used to represent the meaning of tasks [24] or activities [33, 46].

Business workflows focus on procedural rules that generally represent the control flows while scientific workflows highlight data flows that are depicted by data dependencies [14]. Business workflows present differences from scientific workflows: 1) scientific workflows exploit tools with a higher level of abstraction; 2) business workflows rely more on the users' interactions; 3) business workflows are based on control flow, whereas scientific workflows use data flow. In the rest of the manuscript, we focus on scientific workflows, or workflows for short when there is no ambiguity.

Scientific Workflow Management System

A Scientific Workflow Management System (SWMS) is a tool to design, manages, share, and executes a scientific workflow. It enables the generation of a Workflow Execution Plan (WEP), which captures optimization decisions and execution directives, typically the result of compiling and optimizing workflows, before execution [93].

SWMSs can also support functionalities to capture and share workflow information, such as workflow provenance. Workflow provenance is the metadata that captures the generation history of a dataset, of a workflow and of an execution. Provenance data is used for workflow analyses and workflow reproducibility. [41, 114, 9, 77]

2.2.2 Architecture of SWMSs

Liu *et al.* [93] propose a generic architecture for SWMSs composed of five layers: user interface, user services, WEP generation, WEP execution, and infrastructure. These layers represent the common features of SWMSs [47, 178, 8, 111]. Figure 2.1 shows the generic architecture of a SWMS. The upper layers perform more abstract functionalities and rely on the lower layers. Users interact with a SWMS through the user interface to access the user services. When executed, workflows are processed by the WEP generation to produce a WEP, which is executed at the WEP execution layer. The SWMS accesses the physical resources through the infrastructure layer for workflow executions.

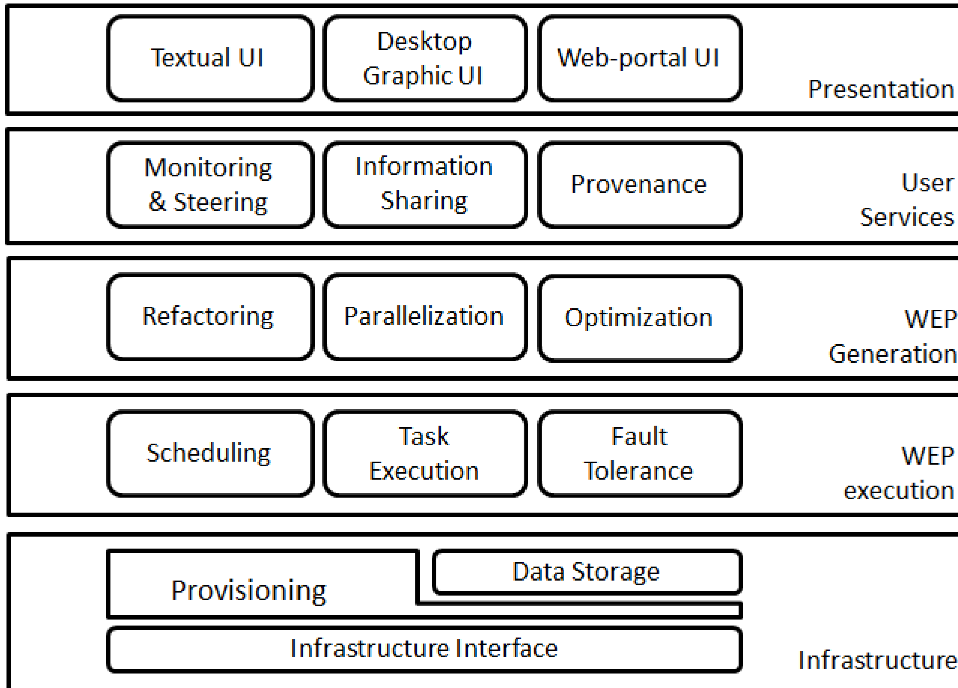


FIGURE 2.1: Generic architecture of a SWMS [93].

The combination of the WEP generation layer, the WEP execution layer, and the infrastructure layer corresponds to a workflow execution engine.

This architecture will be the basis for the architecture in our work. We focus on the user service layer for workflow information sharing and on the WEP generation and WEP execution layers for workflow scheduling and cache data management. For more details see Section 2.2.2 Functional Architecture of SWfMSs [92].

User Interface

The user interface layer is the interface for the interactions between the user and the SWMS. It is through this interface that the user design and manage workflows. The user interface can be either textual or graphical. This interface also shows results of the user services, such as execution status, workflow steering representation, and information sharing commands.

Some SWMSs have a textual interface [160, 60, 111]. Wilde *et al.*[160] propose Swift, a distributed scripting language. 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.

Pegasus relies on Wings to create workflows [60]. Wings is used for the workflow design and the mapping of the data to the workflow, while Pegasus manages the workflow execution through a textual interface. Chiron represents workflows as a DAG in textual XML format [111]. It enables Chiron to use algebraic operators to optimize workflow execution.

Other SWMSs propose a graphical interface [61, 8, 129]. The graphical interface simplifies the designing and management processes of the workflow

by the user, offering drag and drop functionality. Galaxy [61], is a web-portal-based SWMS. The workflow is designed through a browser on the client side, then it can be executed on a remote cloud. Taverna and OpenAlea [112, 129] enable workflow design and sharing through packages accessible on the graphical interface. The workflow design can be performed through a drag and drop feature, where the user interacts directly with the DAG. These SWMSs use textual languages for the inner representation of the workflow, such as JSON in Galaxy [61].

User Service

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

Workflow steering is the interaction between a SWMS and a user to control the workflow execution progress or configuration parameters [103]. Through workflow steering, a user may change parameters, stop unnecessary execution, or re-execute workflows if errors occur. The user may even benefit from the workflow execution before its completion to prove her hypothesis [41, 9]. The user can also monitor the workflow executions. Workflow monitoring consists of tracking the workflow execution status and displaying it for the user. Dynamic monitoring and steering are important to control workflow executions, especially when the execution time is long [49].

Information sharing consists of sharing data including input data, output data, metadata, or even workflow design between users for reusing workflows. It can be done through the same SWMS of different SWMS environments. SWMSs may enable workflow repositories, where users can upload and download activities and workflows, thus reducing repetitive work between scientist groups. A workflow repository can be shared in the same SWMS environment, *i.e.*, Taverna enables data and workflow sharing through the “ myExperiment” social network [162]; Galaxy has a web-based shared repository [61]; OpenAlea propose an integrated repository to share workflows [129]. Terstyanszky *et al.* [148] propose the SHIWA repository, an inter SWMS sharing solution.

Provenance data in workflows is important to support reproducibility, result interpretation, and problem diagnosis. Data provenance captures the metadata on the generation and dependencies of a given dataset. Execution provenance captures the metadata on the tasks execution and execution environment. Provenance data management concerns the efficiency and effectiveness of collecting, storing, representing, and querying provenance data [93]. Although some workflow provenance recommendations exist, PROV [106], their implementation varies in SWMSs. Gadelha *et al.* [55] propose MTCProv, implemented in the SWift SWMS, that enable to store the provenance data in a relational database. The provenance data can be queried for a graph representation trough a data model or to query information. Kim *et al.* [82] propose a semantic-based approach to generate provenance information in Pegasus SWMS. The approach saves the semantic workflow definition

given by Wings, with the execution provenance generated by Pegasus during a workflow execution. Altintas *et al.* [7, 43] propose a framework to collect provenance data for Kepler SWMS. The framework collects both data provenance and execution provenance, and store them in a MySQL database. The provenance data can be retrieved and visualized through the Kepler API. Callahan *et al.* [28] propose to store provenance data with intermediate data generated with workflow executions implemented in VisTrails. It strengthens reproducibility creating "strong links" between the provenance and execution [85]. VisTrails also enables visualization of provenance data evolution with workflow evolution.

WEP Generation

The WEP generation layer is responsible for generating a WEP from the workflow design and optimization, *i.e.*, workflow refactoring, workflow parallelization, and optimization.

Workflow refactoring consists of simplifying the workflow, *i.e.*, by removing redundant activities, or by partitioning it in workflow fragments (a subset of activities and data dependencies of the workflow [110]), to ease the execution. Cohen *et al.* [40] present a method that automatically detects workflow structures and replaces them by equivalent structures of lower complexity when possible. The study provides a set of anti-patterns, *i.e.*, idiomatic forms that lead to over-complicated design. Then, the algorithm automatically detects derivations of such anti-patterns in the workflow and replace them. It conserves the workflow semantic while reducing the redundancy and finding patterns that simplify the workflow. Deelman *et al.* [47] present a caching approach, where redundant activities are removed when refactoring. Before the workflow execution, the SWMS check each activity with the cache metadata. The intermediate data generated by previous executions are reused when available. Thus, the executable workflow generated is modified to take the cached data into account. Section 2.4.2 provides more details on data caching and reuse in SWMS.

Workflow partitioning consists of splitting a workflow into workflow fragments. Ogasawara *et al.* [110] present a partitioning method based on a workflow algebra, where the workflow is fragmented into equivalent expressions. The fragments are then transformed into multiple WEPs to be executed. The workflow can be partitioned to reduce the storage required for the execution of each fragment or to reduce the scheduling complexity [33].

Workflow parallelization consists in converting the workflow into sets of executable tasks for the WEP. It is based on different types of parallelism, *e.g.*, data parallelism, workflow parallelism. The parallelization can be performed at different levels, *i.e.*, workflow fragment, activity, or task level. Some SWMSs can manage the parallelization at the activity level, *i.e.*, Swift [178], Pegasus [47], or OpenAlea [130], perform the parallelization within their execution engine. The parallelization can be outsourced using web services or Hadoop MapReduce systems [157]. Outsourcing the parallelization prevents

the SWMS to manage some optimizations on the workflow execution, to manage the data placement on a distributed environment and make it harder to capture the provenance data [93].

Workflow optimization generates the WEP based on the workflow refactoring and parallelization with additional instructions. The additional instructions describe scheduling objectives for workflow executions, *i.e.*, minimizing execution costs, meeting deadline constraints, or following security constraints.

WEP Execution

The WEP execution layer manages the workflow scheduling, task execution, and fault-tolerance.

Workflow scheduling is the process of mapping and managing execution of inter-dependent tasks on distributed resources [167]. The scheduling can be performed at the workflow fragment level, activity, or task level. Some SWMSs outsource workflow scheduling as well as its execution, it is performed at the same time as the workflow parallelization [157]. Wu *et al.* [164] propose a taxonomy to describe workflow scheduling in a cloud and compare the main scheduling algorithms. The taxonomy is composed of three categories: static scheduling, dynamic scheduling and hybrid scheduling. This taxonomy aims at comparing scheduling algorithms and workflow execution objectives (such as robustness, reducing costs, constrains). We present more in detail workflow scheduling in cloud in Section 2.3.4.

The task execution consists of processing some input data at computing nodes to produce the output data. Some SWMSs, *e.g.*, Galaxy, Pegasus, OpenAlea, manage the data transfers and task execution within their execution engine. It enables them to take into account the distribution of the resources, the data location, the dynamic variation of resources, to optimize the execution. The provenance data of the execution is generated at this point.

The goal of fault tolerance management is to handle failures during task execution and resource provisioning. Ganga *et al.* [56] classify the fault tolerance technique into two groups: proactive and reactive. Proactive fault tolerance aims at predicting the failures and handle the failure before it happens. Reactive fault tolerance act after a failure occurs and aims at reducing its impact while continuing the execution.

Infrastructure

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

Cluster computing is 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 [38]. 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

or virtual machines (VMs) in the cloud. In the cloud, a VM is a virtualized machine (computer). Cluster users can rely on message passing protocols, such as MPI for parallel execution.

Cloud computing provides computing, storage and network resources through infrastructure, platform and software services, with the illusion that resources are unlimited. The cloud uses virtualization techniques to provide scalable services that are independent of the physical infrastructure. 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. There is also the possibility of dynamic provisioning. 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 many benefits, in particular better performance, and reduced financial cost.

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

The data storage module generally exploit database systems and file systems to manage the data during workflow execution. Generally, the file systems and the database systems take advantage of computing nodes and storage resources provided by the provisioning module. In a multisite environment, SWMSs can cluster the data and place each dataset at a single site, distribute the newly generated data to multiple sites at runtime and adjust data among multiple sites [169]. SWMSs can also put some data in the disk cache of one computing node to accelerate data access during workflow execution [143]. 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.

2.3 Scientific Workflow Management in the Cloud

The cloud provides a scalable, cost-effective solution to deploy and execute data-intensive workflows. A cloud is usually composed of multiple data centers located at different sites, which allows to scale up compute and storage resources. Yet, most of the applications can be run on a single site, which is often enough. However, a workflow may require multiple sites to be executed in order to use more resources than what is available at one site. An execution needs more than one site when the datasets are stored at specific sites, and cannot be transferred due to access rights or privacy issues. Therefore, workflow management on a multisite cloud is an important problem.

In this section, we focus on workflow deployment and execution in the cloud, including parallelization and scheduling.

2.3.1 Cloud Computing

Cloud computing is a natural evolution, and combination, of different computing models proposed for supporting applications over the web: service oriented architectures for high-level communication of applications through web services, utility computing for packaging computing and storage resources as services, cluster and virtualization technologies to manage lots of computing and storage resources, autonomous computing to enable self-management of complex infrastructure, and grid computing to deal with distributed resources over the network. However, what makes cloud computing unique is its ability to provide various levels of functionality such as infrastructure, platform, and application as services that can be combined to best fit the users' requirements. From a technical point of view, the grand challenge is to support in a cost-effective way, the very large scale of the infrastructure that has to manage lots of users and resources with high quality of service.

Agreeing on a precise definition of cloud computing is difficult as there are many different perspectives (business, market, technical, research, etc.). However, a good working definition is that a "cloud provides on demand resources and services over the Internet, usually at the scale and with the reliability of a data center" [65]. This definition captures well the main objective (providing on-demand resources and services over the Internet) and the main requirements for supporting them (at the scale and with the reliability of a data center). 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. However, implementing a pricing model is complex as users should be charged based on the level of service actually delivered, e.g., in terms of service availability or performance. To govern the use of services by customers and support pricing, cloud providers use the concept of Service Level Agreement (SLA), which is critical in the services industry (e.g., in telecoms), but in a rather simple way. The SLA (between the cloud provider and any customer) typically specifies the responsabilities, guarantees and service commitment. For instance, the service commitment might state that the service uptime during a billing cycle (e.g., a month) should be at least 99a service credit.

According to Buyya *et al.* [25], a cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreement. Cloud computing enables on-demand access to a shared pool of configurable computing resources, *e.g.*, storage, services, servers, that can be easily provisioned and deployed. Cloud services can be divided in three broad categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

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 server as virtual machines, i.e., as if they would run on distinct physical servers. Customers can then requisition computing instances as virtual machines and add and attach storage as needed. An example of popular IaaS is Amazon web Services.

SaaS is the delivery of application software as a service. It generalizes the earlier Application Service Provider (ASP) model whereby the hosted application is fully owned, operated and maintained by the ASP. With SaaS, the cloud provider allows the customer to use hosted applications (as with ASP) but also provides tools to integrate other applications, from different vendors or even developed by the customer (using the cloud platform). 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. An example of popular SaaS is Safesforce CRM system.

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 virtual machines, and integrate them with applications provided as SaaS. An example of popular PaaS is Google Apps.

By using a combination of IaaS, SaaS and PaaS, customers could move all or part of their information technology (IT) services to the cloud, with the following main benefits:

- **Cost.** The cost for the customer can be greatly reduced since the IT infrastructure does not need to be owned and managed; billing is only based only on resource consumption. For the cloud provider, using a consolidated infrastructure and sharing costs for multiple customers reduces the cost of ownership and operation.

- **Ease of access and use.** The cloud hides the complexity of the IT infrastructure and makes location and distribution transparent. Thus, customers can have access to IT services anytime, and from anywhere with an Internet connection.

- **Quality of Service (QoS).** The operation of the IT infrastructure by a specialized provider that has extensive experience in running very large infrastructures (including its own infrastructure) increases QoS.

- **Elasticity.** The ability to scale resources out, up and down dynamically to accommodate changing conditions is a major advantage. In particular, it makes it easy for customers to deal with sudden increases in loads by simply creating more virtual machines.

SWMSs can use SaaS and IaaS to deploy Workflow as a Service (WfaaS) software [86, 39]. This enables the user to deploy and run workflows in a cloud from the SWMS interface. In the rest of the manuscript, we focus on IaaS, which enables the SWMS to optimize workflow execution on the infrastructure.

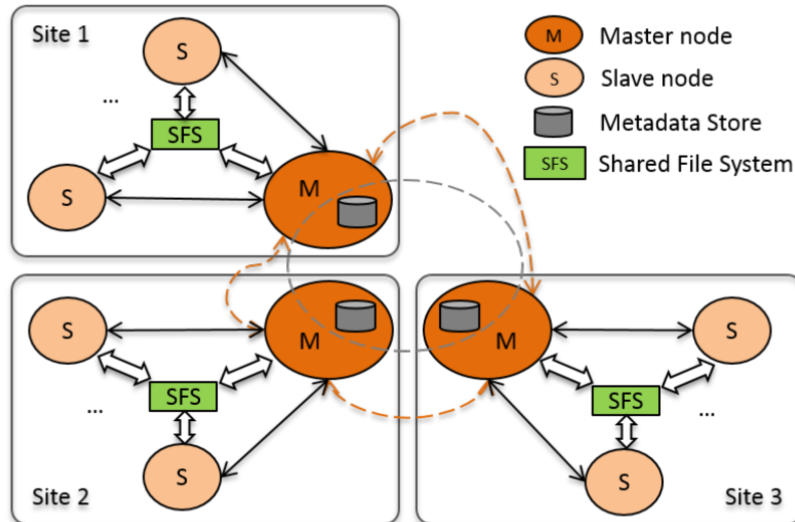


FIGURE 2.2: Multisite workflow execution architecture. Dotted lines represent inter-site interactions [94].

2.3.2 Multisite Cloud

For scalability, data access or data privacy reasons, an application may require to be executed at more than one site. Most public cloud providers, such as Amazon or Microsoft, have many geographically distributed sites. Amazon for example has dozens of sites distributed on each continent.

Liu *et al.* [97] define 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. Thus, the user can access and deploy data and applications at each site, taking advantage of the data location when using computing and storage resources. If each site has the same resource capabilities as the others, the multisite cloud is said to be homogeneous. Otherwise, it is heterogeneous.

Petri *et al.* [125] present a framework to manage workflow executions in a multisite cloud. The user submits a workflow to be executed in a shared multisite management space. Then, the framework automatically distributes the tasks on the sites. The framework is based on Building Information Modelling to manage "shared" execution spaces. The shared execution spaces are created on-demand at a cloud site, composed of one master node and several worker nodes. One site can host several shared execution spaces. Executions are managed by the master node at each share execution space. The framework enables users to share their data in the multisite cloud while maintaining their data on their infrastructure (local server, private cloud). It also allows executing a workflow using the data shared by other users.

Liu *et al.* [94] propose a multisite workflow execution architecture (see Figure 2.2). The architecture is based on the two-level management of workflows: inter-site and intra-site. At the inter-site level, the communication and synchronization are handled by the master nodes (M), one per site. At the intra-site level, the workflow execution is managed with a master/ slave model, where the slave nodes (S) execute the tasks. At each site, the processed data (input, output, and intermediate data) is shared through a shared file

system and metadata is stored at the master node. This two-level management enables multisite load balancing and data location awareness while having a low latency at the inter-site execution between the master nodes and the slave nodes.

Other common frameworks such as Hadoop and MapReduce have been extended to work in multisite clouds. Tudoran *et al.* [151] propose a method to distribute MapReduce applications on the sites depending on the input data, assuming that the input data is distributed among the sites. The Map tasks are scheduled at the sites where the input data is stored. Each site has a Reducer, to execute Reduce operations. At the multisite level, the output data is aggregated by a MetaReducer. The solution efficiently reduces the latency of the Reduce tasks and reduces data transfer by exploiting input data locality.

Luo *et al.* [98] improve the previous MapReduce multisite solution by proposing two static scheduling algorithms for MapReduce multisite execution. The first algorithm schedules the tasks according to computing nodes, enabling task balancing among the sites. However, it does not consider data transfers between sites, so it does not perform well in scenarios where the input data distribution over multiple sites in unbalanced. The second algorithm schedules the tasks according to the location of their input data, reducing data transfers between the sites. However, the algorithm does not consider the workload when scheduling tasks, thus, hurting load balancing of compute-intensive workflows. Both algorithms first schedule the tasks at the sites, then, the output data at each site is aggregated by a global reducer, such as the MetaReducer. These two algorithms enable MapReduce workflow execution in multisite cloud, taking into account either data-intensive or compute-intensive workflows but not both.

2.3.3 Data Storage

Data storage in the cloud is critical for the performance of data-intensive workflows. The data processed by a workflow during execution, including input data, intermediate data and output data, is usually stored in a file system while the metadata (provenance data, or metadata on the processed data) needed for workflow execution is usually stored in a database, including SQL and NoSQL. In this section, we discuss the techniques for file management and metadata management that can be used in the cloud.

File Management

A file system is in charge of controlling how information is stored and retrieved in a computer or a computer cluster [11]. 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 (See Section 2.4.3 Data Storage in the Cloud [92]).

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) [141], Global File System (GFS) [131] and Network File System (NFS) [140].

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 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 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 partitioned 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) [59], 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. GFS also provides snapshot support, garbage collection, fault-tolerance and diagnosis. For high availability, GFS supports master replication and data chunk replication.

BlobSeer [109] is another distributed file system optimized for Binary Large OBjects (BLOBs). 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 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 data replication, consistency semantics and scalability based on several versioning mechanisms. A first performance comparison of BlobSeer with HDFS 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.

Metadata Management

SWMSs rely on databases to manage provenance and file metadata. The metadata is usually structured and can be stored in a relational database with good performance in a centralized environment [91]. However, in a distributed environment such as the cloud, NoSQL databases are more efficient for storing and querying metadata [19]. NoSQL databases enable exponential growth with no clear schema definitions, that suits graph provenance data model. Moreover, NoSQL databases enable horizontal scaling, *i.e.*, it considers

documents as independent objects, the documents can be stored on different servers without worrying about joining rows from multiple servers, as it is the case with the relational model. Thus, the NoSQL databases scale better than relational databases in distributed environments.

Notice that, common metadata management systems rely on an assumption that the average file size is very large, while the number of unique files and directories is comparatively small. Yet, in many workflow applications, the workflow execution generates a huge number of tasks that consume and generate small files. Thus, the replication, synchronization, and latency accesses of the metadata lead to a metadata management bottleneck [149]. In that case, efficient metadata handling plays a key role in workflow execution performance [94].

Wercelens *et al.* [159] propose a comparison of different NoSQL systems to manage metadata for bio-informatics workflows. Their workflow provenance model is PROV-DM, a standard provenance model for workflows. The bio-informatics workflows considered are not data-intensive but generates a huge number of tasks, thus leading to a huge number of accesses to the provenance database. The workflows are executed in a cloud, with several provenance nodes, and one computing node. NoSQL databases, such as Cassandra or MongoDB, appear to efficiently handle provenance data, as they scale up well in distributed environments and match the provenance model well. Especially, graph NoSQL databases, such as OrientDB, are well adapted for provenance as they conform to the PROV-DM model.

Pineda Morales *et al.* [127] propose a hybrid centralized/decentralized model to handle metadata in a multisite cloud. Figure 2.3 shows the four metadata management strategies they propose. The strategies are based on the assumption that local metadata operations are significantly faster than remote operations. However, to maximize the local operations, the metadata have to be fully replicated at all sites. The different strategies explore the trade-off between latency for metadata access and the replication cost of the metadata. Each strategy is shown to be suited for a different scenario. In particular, the analysis shows that the centralized approach with replicated metadata registries is best suited for data-intensive workflows. while decentralized approaches are more adapted for workflows with a large number of tasks that process small files.

Chebotko *et al.* [30] propose an RDF data store based on SPARQL to manage and store the provenance data in centralized storage (in a local server). The solution includes two storage models: a database schema with a slower data mapping strategy but with faster query response time, and a faster data mapping strategy can be chosen to speed up data writing. The first model is adapted for long duration tasks, where provenance access performance is less important. The second model is adapted for short duration tasks, where provenance access performance becomes critical. The solution enables an efficient trade-off between data mapping performance and query performance but only in a centralized environment.

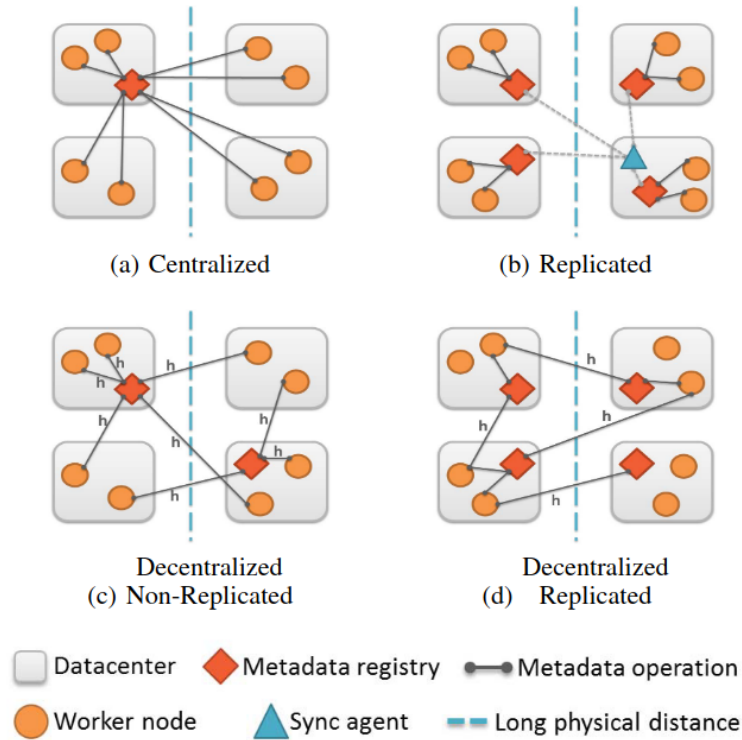


FIGURE 2.3: Strategies for geographically distributed metadata management [126].

2.3.4 Workflow Scheduling in the Cloud

To be executed in distributed environments, workflows require to be parallelized and scheduled. Parallelization consists of generating parallel executable tasks from the workflow definition. Workflow scheduling is the process of mapping and managing executable tasks to computing resources (*i.e.*, computing nodes) during workflow execution. The goal is to get an efficient Scheduling Plan (SP) that minimizes a function based on resource utilization, execution cost and makespan. The section focus on SWMSs that perform both parallelization and scheduling.

The cloud, with its elasticity, is cost-effective to execute workflows. The cloud also enables a multisite approach, where the workflow execution may benefit from data location to avoid data transfers.

In this section, we introduce workflow parallelization and workflow scheduling in general. Then, we discuss workflow management in monosite and multisite clouds.

Parallelization

Workflow parallelization analyzes the workflow to identify the tasks that can be executed in parallel. Parallelism can be coarse-grained and fine-grained. Coarse-grained parallelism is performed at the workflow level, where fragments and sub-workflows are parallelized [148]. Fine-grained parallelism is

performed at the activity level, determining which tasks from each activity can be performed in parallel [24].

Fine-grained parallelism is of three types: data parallelism, independent parallelism and pipeline parallelism [93] (See Figure 2.4). Data parallelism is the generation of multiple tasks from one activity to be executed on different data chunks (See Figure 2.4b). Independent parallelism is the parallelization of "independent" activities, *i.e.*, activities that have data dependencies between them (See Figure 2.4c). Pipeline parallelism consists of sending the output data of an activity to be consumed directly by the next dependent activity. The producer activity does not need to be entirely completed, thus saving memory and disk access (See Figure 2.4d). Hybrid parallelism is the combination of the three other types (See Figure 2.4e).

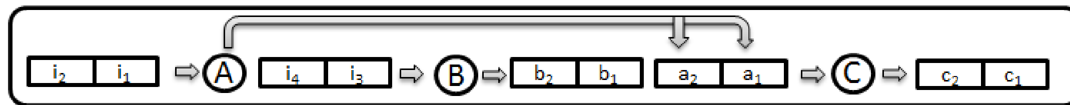
Scheduling

Workflow scheduling consists of mapping the workflow tasks, generated by workflow parallelization, to the physical resources. Scheduling algorithms can focus on one or more objectives (such as makespan, financial cost, energy consumption). Finding a schedule for any DAG of tasks is an NP-hard problem [57]. Thus, the scheduling algorithms presented in the manuscript are heuristics-based algorithms.

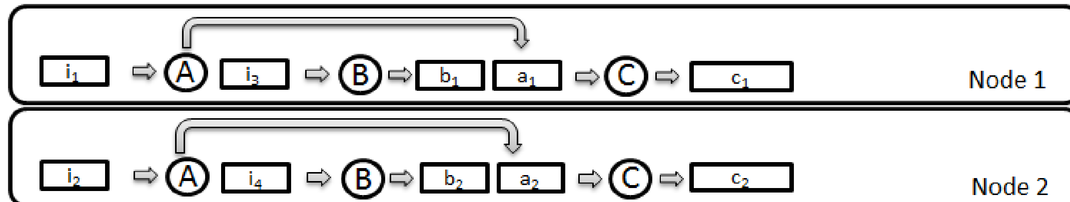
The tasks can be clustered in bags to be scheduled, which reduces the scheduling overhead. A bag of tasks is executed on the same computing node. Chen *et al.* [34] propose three balanced task clustering algorithms. They are fine-grained approaches that reduce the imbalance of runtime and data dependencies among the bags of tasks. The three algorithms cluster the tasks, focusing on one of the following objectives: balancing the tasks runtime variation, balancing the data dependencies, or balancing the impact factor (which is a metric that represents the similarity between tasks).

Workflow scheduling can be static, dynamic, or hybrid. Static scheduling generates an SP that allocates all the executable tasks to computing nodes before execution and the SP is strictly applied during execution [24]. Static scheduling adds very little overhead at runtime because it is computed before. It is efficient when the SWMS can accurately predict the workload and the size of data generated by each task. Topcuoglu *et al.* [150] propose a static scheduling algorithm, Heterogeneous Earliest-Finish-Time (HEFT), which rank tasks by their expected completion time. Then, it schedules each task to minimize the expected completion time. HEFT is widely used in SWMSs as it efficiently schedules workflows in environments with little variation (such as a grid). Rodriguez *et al.* [137] propose a static cost minimization algorithm based on the meta-heuristic optimization technique particle swarm optimization (PSO). PSO finds a solution within a deadline constraint. It considers elastic provisioning and VM performance variation, which make it efficient for execution in a cloud.

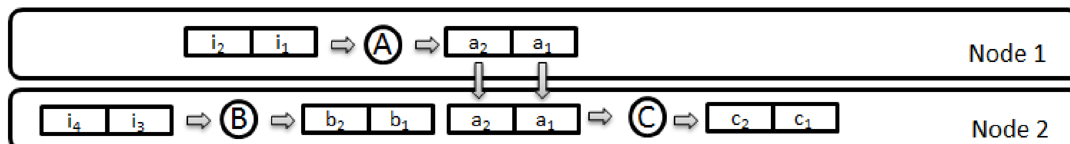
Dynamic scheduling generates SPs during workflow execution, dynamically assigning executable tasks to computing nodes. Dynamic scheduling is efficient in environments where the infrastructure capabilities (*e.g.*, bandwidth,



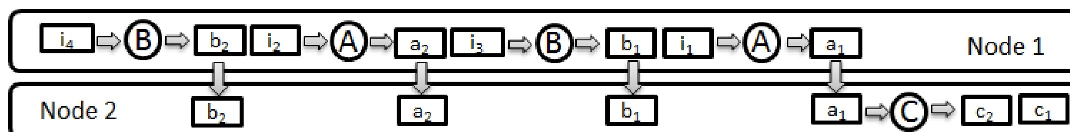
(A) **Sequential execution in one computing node.** Activity B starts execution after the execution of activity A and activity C starts execution after the execution of activity B. All the execution is realized in one computing node.



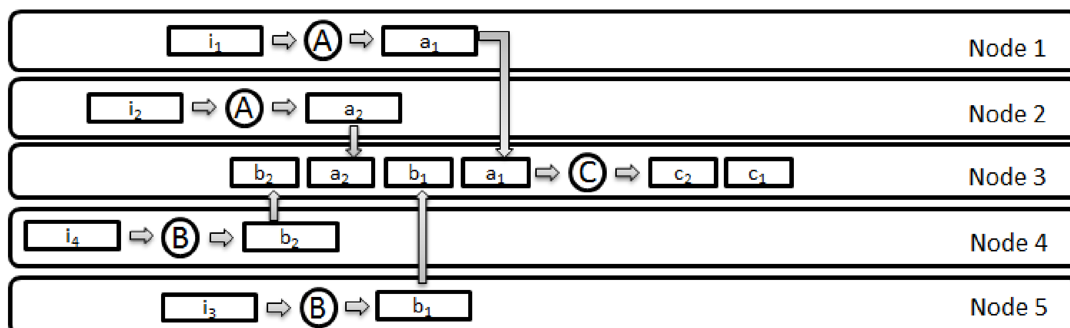
(B) **Data parallelism.** The execution of activities A, B, C is performed in two computing nodes simultaneously. Each computing node processes a data chunk.



(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 2.4: **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. [93]

computing power) vary a lot during execution or when the workflow tasks' behavior is unpredictable. Some dynamic scheduling algorithms are based on static algorithms. Yu *et al.* [168] propose a dynamic scheduling algorithm based on HEFT which takes into account the tasks that are already executed to adjust the ranking among the remaining tasks. The algorithm performs better when the workflow has many similar tasks. Thus, it is efficient for data-intensive workflows with a high degree of parallelism.

Hensgen *et al.* [76] propose min-min, a dynamic scheduling algorithm that 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 M is the computing node that is executing a task having minimum expected execution time in the mapped tasks.

Hybrid scheduling takes advantage of both static and dynamic scheduling approaches. The SWMS first generates a SP before the execution for tasks with enough information. Then, it can adapt the SP during execution. De Oliveira *et al.* [114] propose four hybrid scheduling algorithms: greedy scheduling, task grouping, task performing, and load balancing. The greedy scheduling algorithm produces static SPs to choose the most suitable task to execute for a given idle VM based on a 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 SP in order to meet the deadline of execution time and the budget limit.

Monosite Cloud

In a monosite cloud, SWMSs rely on the VMs to execute workflows and exploit cloud services [45]. Some SWMSs use a middleware to manage the VMs in order to execute the workflow in a cloud. This midleware handles the creation and removal of VMs as well as the resources provisioning and the communications between VMs. Some SWMS examples that use a middleware are Swift with Coasters [70], Kepler with its Amazon EC2 actors [156], Galaxy with CloudMan [5]. Using a middleware to manage VMs prevents taking advantage of the dynamic provisioning aspect of the cloud.

Some SWMSs can exploit dynamic provisioning of VMs and storage in a cloud for workflow executions. The SWMS can scale up or down the resources allocated to adapt the environment to the workflow execution. Nagavaram *et al.* [100, 107] propose Wrangler, a dynamic provisioning system in a cloud for Pegasus. It can create VMs dynamically to reduce execution time under monetary cost constraints. If the monetary cost exceeds the constraint, the system removes VMs. Other examples that perform dynamic provisioning are Askalon [53], or Chiron with Scicumulus [114]. Dynamic provisioning enables to manage changes in resource demand during workflow execution at the expense of a runtime overhead.

Multisite Cloud

Workflows can require execution in a geo-distributed multisite cloud for different reasons, *e.g.*, 1) when the input data is distributed on the sites (for example when several research teams generated them), it might be too costly to transfer the data among the sites; 2) some datasets cannot be transferred due to privacy requirement or access rights, a part of the computations needs to be performed at the site where the data is located; 3) to scale up the computing environment, workflow executions may require more resources than the one available at one site. A multisite cloud is composed of several data centers that are located at different geographically sites. A SWMS can run a workflow on a multisite cloud by either 1) deploying the SWMS on top of a multisite cloud platform; 2) directly executing the SWMS on the multisite cloud [44].

When the SWMS is deployed in a multisite cloud, the cloud platform manages the multisite execution (see Section 2.3.2 for examples of cloud platforms). As resources management is performed by the cloud platform, the SWMS cannot optimize workflow execution using infrastructure and data location information.

SWMSs are directly executed in the multisite cloud when the SWMS has access to the resources of each site. The SWMS becomes multisite aware and can adapt the workflow execution to the distributed environment. Most SWMSs that manage multisite workflow scheduling, adopt a two-level scheduling: inter-site and intra-site. At the inter-site level, the SWMS may partition the workflow and schedule the fragments at each site, balancing the workflow execution. At the intra-site level, the workflow fragments are scheduled on the computing nodes by monosite scheduling methods.

Chen *et al.* [33, 34] propose two workflow partitioning methods for multisite execution in Pegasus. The first method partitions the workflow under storage constraints at each site. It aims at minimizing the overall execution cost by computing a trade-off between the increased data transfer and the increased parallelism. The second method partitions the workflow by generating fragments with similar workloads, balancing the execution time at each site. These two methods are efficient in homogeneous multisite clouds only.

Liu *et al.* [95] propose three multisite scheduling algorithms based on a multi-objective cost function. The algorithms take into account the resources of each site, and inter-site latency to minimize the monetary cost and makespan of the workflow execution. The algorithms partition the workflow at the inter-site level and schedule each fragment at the site that minimizes the global cost. They are adapted for data-intensive workflows in heterogeneous multisite cloud. These algorithms will be presented in more detail in Section 2.4.3.

2.4 Data Caching for Workflows

To save execution time and resource utilization, a SWMS can cache and reuse some data. The data can be cached in memory to be reused during the same workflow execution or made persistent for future workflow executions. In

this section, we first introduce data caching in computer systems. Then, we present data caching in SWMSs with an overview of the existing solutions.

2.4.1 Data Caching in Computer Systems

In computer systems, caching means storing data in a storage media whose access is significantly faster (*e.g.*, main memory) than recomputing the data or reading from a slower storage media (*e.g.*, disk). To be efficient, a cache must exploit temporal locality and spatial locality. The temporal locality represents the likelihood of cached data to be frequently accessed in the near future. The spatial locality represents the likelihood of cached data to be physically close to the location where it is accessed.

In computer systems, caching is used in both hardware and software. At the hardware layer, cache memories (small and fast storage) hold some instruction items and referenced blocks of data, enabling high-speed access to main memory. At the operating system, the most recently used blocks can be stored in main memory. At the layer of application programs, a cache speeds up requests from users when they access the same data several times.

Cache implementation and contents vary between applications. As an example, Web browsers cache the data of the most recently visited sites in a local disk [176]. At the network level, the Web servers store the recently requested web data in a front-end disk cache. In a relational database, the query plans and the pages can be cached to speed up queries over big tables.

In distributed databases, an important caching technique is materialized views (See Section 3.1.1 Maintenance of Materialized Views in [119]) A materialized view stores the tuples of a view in a database relation, like the other database tuples, possibly with indices. Thus, access to a materialized view is much faster than deriving the view, in particular, in a distributed DBMS where base relations can be remote. Introduced in the early 1980s [4], materialized views have since gained much interest in the context of data warehousing to speed up On Line Analytical Processing (OLAP) applications [67]. Materialized views in data warehouses typically involve aggregate (such as SUM and COUNT) and grouping (GROUP BY) operators because they provide compact database summaries. An important aspect is query rewriting using materialized views, which requires translating a query on base tables into an equivalent query on materialized views.

A materialized view is a copy of some base data and thus must be kept consistent with that base data which may be updated. View maintenance is the process of updating (or refreshing) a materialized view to reflect the changes made to the base data. The issues related to view materialization are somewhat similar to those of database replication. However, a major difference is that materialized view expressions, in particular, for data warehousing, are typically more complex than replica definitions and may include join, group by and aggregate operators. Another major difference is that database replication is concerned with more general replication configurations, *e.g.*, with multiple copies of the same base data at multiple sites.

A view maintenance policy allows a database administrator to specify when and how a view should be refreshed. The first question (when to refresh) is related to consistency (between the view and the base data) and efficiency. A view can be refreshed in two modes: immediate or deferred. With the immediate mode, a view is refreshed immediately as part as the transaction that updates base data used by the view. If the view and the base data are managed by different DBMSs, possibly at different sites, this requires the use of a distributed transaction, for instance, using the two-phase commit protocol [119]. The main advantages of immediate refreshment are that the view is always consistent with the base data and that read-only queries can be fast. However, this is at the expense of increased transaction time to update both the base data and the views within the same transactions. Furthermore, using distributed transactions may be difficult.

In practice, the deferred mode is preferred because the view is refreshed in separate (refresh) transactions, thus without performance penalty on the transactions that update the base data. The refresh transactions can be triggered at different times: lazily, just before a query is evaluated on the view; periodically, at predefined times, *e.g.*, every day; or forcedly, after a predefined number of updates to the base data. Lazy refreshment enables queries to see the latest consistent state of the base data but at the expense of increased query time to include the refreshment of the view. Periodic and forced refreshment allow queries to see views whose state is not consistent with the latest state of the base data. The views managed with these strategies are also called snapshots [3, 18].

The second question (how to refresh a view) is an important efficiency issue. The simplest way to refresh a view is to recompute it from scratch using the base data. In some cases, this may be the most efficient strategy, *e.g.*, if a large subset of the base data has been changed. However, there are many cases where only a small subset of view needs to be changed. In these cases, a better strategy is to compute the view incrementally, by computing only the changes to the view.

Efficient techniques have been devised to perform incremental view maintenance using both the materialized views and the base relations. The techniques essentially differ in their views' expressiveness, their use of integrity constraints, and the way they handle insertion and deletion. Gupta and Mumick [69] classify these techniques along the view expressiveness dimension as non-recursive views, views involving outerjoins, and recursive views. For non-recursive views, *i.e.*, select-project-join (SPJ) views that may have duplicate elimination, union and aggregation, an elegant solution is the counting algorithm [68]. One problem stems from the fact that individual tuples in the view may be derived from several tuples in the base relations, thus making deletion in the view difficult. The basic idea of the counting algorithm is to maintain a count of the number of derivations for each tuple in the view, and to increment (resp. decrement) tuple counts based on insertions (resp. deletions); a tuple in the view of which count is zero can then be deleted.

Self-maintainability depends on the views' expressiveness and can be defined with respect to the kind of updates (insertion, deletion or modification)

[66]. Most SPJ views are not self-maintainable with respect to insertion but are often self-maintainable with respect to deletion and modification. For instance, an SPJ view is self-maintainable with respect to deletion of relation R if the key attributes of R are included in the view.

Although the concept of materialized view is very powerful, it is simply not applicable in the context of workflows for the following reasons. First, a SWMS does not have a data model, unlike the relational model behind materialized views, and the workflow data is stored in various data sources, such as files or NoSQL databases. Thus, one cannot define what a workflow materialized view would mean. Second, materialized views are useful to speed up queries, by rewriting queries that bear on base tables on equivalent queries on materialized views. This is made possible because there is a high-level query language like SQL. In SWMSs, there is no such query language and workflow queries are made by workflow activities in different ways depending on the data stores that are accessed (*e.g.*, databases or files). Third, there is no equivalent of updating base data through views in SWMSs, as base data are directly accessed by activities.

In distributed environments, systems such as "Memcached" [1] can be used to extend data-intensive applications. "Memcached" is a key-value distributed memory object caching system. It is used widely in the data-center environment for caching results of database calls, API calls or any other data. It is used by several big data applications such as Facebook, Netflix, and Twitter. Some SWMSs [51, 22] use Memcached in multisite execution for both managing the intermediate data distributed on the nodes, and for caching intermediate data during execution. However, Memcached is adapted for small data caching in memory, thus, it does not manage intermediate data generated by data-intensive workflows.

Data caching is a common, efficient feature in computer systems. Some existing caching methods such as Memcached can be applied to SWMSs. Yet, to fully benefit from caching, a SWMS has to deal with both data caching and cache-aware scheduling. Moreover, in SWMSs, the requirements for caching depends on the workflow under consideration. As an example, depending on the time range of re-executions, a workflow re-executed over a long period would benefit more from a persistent cache.

2.4.2 Data Caching in SWMSs

In SWMSs, caching requires to be managed at different layers of the architecture we introduced in Section 2.2.2. Figure 2.5 shows how this architecture should be adapted for cache management. The two required components are cache manager and cache storage. The cache manager manages the cached data during the workflow execution. The cache manager mostly focuses on two decisions: 1) which intermediate data generated by the cache should be stored, and 2) which existing cached data should be reused. The cache manager is accessed at the user service layer for the cached data to be shared

[1] https://memcached.org/

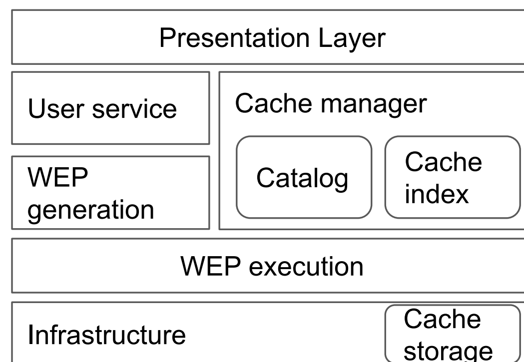


FIGURE 2.5: Generic architecture of a SWMS with cache management.

between users. Data sharing can be either manual, *i.e.*, the user needs to retrieve cached data information through a repository such as "MyExperiment", or it can be automated, in which case the SWMS automatically adds and retrieves cache data information. Cached data information is stored in the cache index, and the information about users (access rights, etc.) is stored in the catalog.

The cache manager is also accessed by the WEP generation layer when a workflow is executed. Workflow refactoring and optimization use the cache information to modify the workflow and generate a WEP accordingly. The decisions on which intermediate data will be added to the cache is performed at the WEP generation layer. Finally, at the infrastructure layer, the cache storage manages the cache replacement policies (*i.e.* when some cached data should be remove from the cache), and its provisioning.

In this section, we present four main approaches to exploit caching in SWMSs: parameter sweep, smart rerun, evolving workflows, and multiuser workflow sharing.

Parameter Sweep

A parameter sweep workflow (PS workflow) is a workflow with multiple input parameter sets, which needs to be executed for each input parameter set [36, 63]. Figure 2.6 shows a PS workflow. The workflow is executed with each set of input data (p_1 to p_n), which generates a set of output data (r_1 to r_n). A set of parameters p_i is composed of the input parameters for each activity (there can be several parameters for one activity). A set of parameters p_i can have common input parameters with another set p_j. When the workflow is executed with p_i and p_j, some tasks may consume the same intermediate data. Structure patterns in the workflow show which fragments are executed several times. When executing PS workflows, finding structure patterns in the workflow enables to optimize the number of tasks that will be executed. Workflow structure patterns can be patterns for parallelization, *e.g.* representing workflows as algebraic expressions [111], or component structure patterns, *e.g.* single activities with one or more input/output dependencies, sequential control and sequential/concurrent data, synchronization of sequential

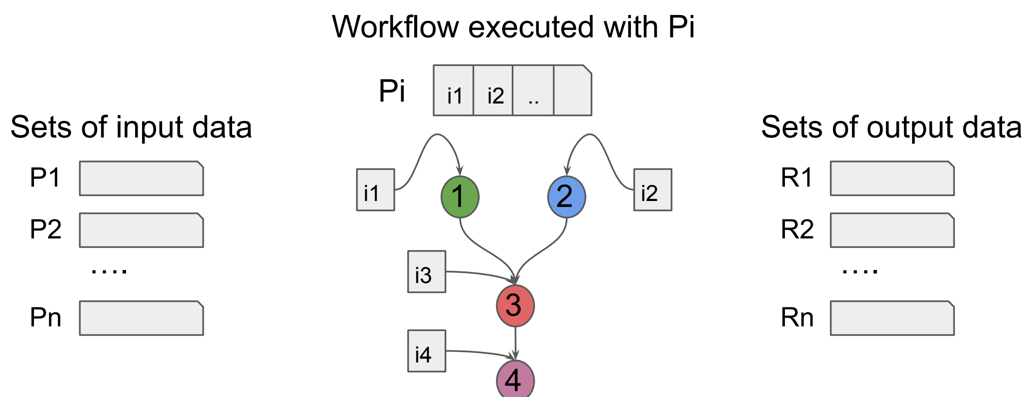


FIGURE 2.6: Parameter Sweep Workflow.

data, data duplication [165]. Similar structure patterns of workflows can be found based on a similarity model of nodes and edges in the workflow DAG [16]. Identifying workflows patterns or common activities enables workflow information sharing and reuse (see Section 2.2.2) among users [165].

PS workflows are used in many domains such as modelling [99], bioscience [144] or chemistry [146]. The techniques to optimize PS workflows executions are mostly based on parallelization techniques. However, some tasks generate data that can be used by each set of input data. These activities can be determined by the workflow structure patterns. The PS workflow execution can be optimized by minimizing the number of times the output data of these specific activities is generated, by replicating this data or sharing it.

De Oliveira *et al.* [115] propose an adaptive approach to execute PS workflows, implemented in SciCumulus. The approach is based on identifying the data that will be consumed by several tasks and storing them in a centralized shared file system. During workflow execution, the tasks reuse the output data whenever possible, which reduces task re-computations. The scheduling algorithm proposed in [115] is based on a 3-objective weighted cost model that considers total execution time, financial cost, and reliability. Scheduling is done by an adaptive greedy algorithm that explores the space of alternative schedules, considering the cost model and possible changes in the cloud environment.

Owsiak *et al.* [118] improve Nimrod, the PS workflow framework of the Kepler SWMS, by enabling caching and reusing of data. To handle PS workflows, Kepler generates multiple instances of the SWMS, each executing the workflow with one set of parameters. One instance of Kepler is the coordinator of the other instances. The solution proposed by Owsiak *et al.* includes a cache storage accessible by each Kepler instance. The coordinator instance manages what cached data can be reused by the other instances. The cache data is stored in a relational database, which is hosted on the user device. However, the solution is not adapted for data-intensive workflows and distributed environments.

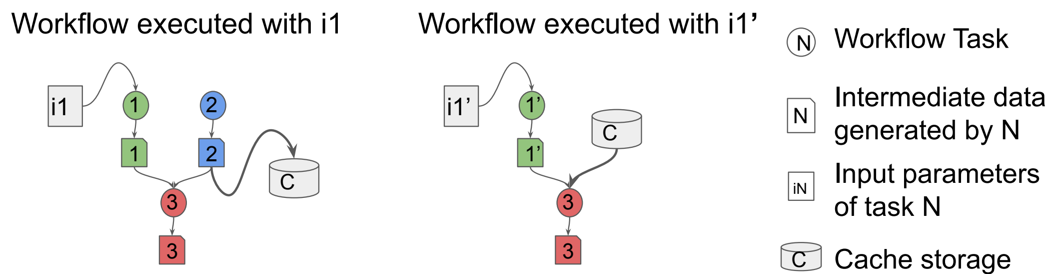


FIGURE 2.7: Smart rerun: a workflow executed twice, the second execution benefits from cached data generated by the first execution.

Smart Rerun

Smart rerun is the re-execution of a workflow that has already been executed with the same or similar input data and input parameters by a single user in most cases. A user needs to rerun a workflow for routine jobs or analyses over multiple workflows.

Figure 2.7 shows a workflow executed twice. The first execution of the workflow (left hand-side) is with input parameter *i1* for task 1. After execution, the intermediate data generated by task 2 is stored in the cache. The workflow is re-executed (right hand-side) with a different parameter for task 1. Task 1 with parameter *i1′* generates the intermediate data *1′* which is different from the data generated from the first execution. Task 2 however is the same and the intermediate data *2* can be reused instead of recomputing task 2.

Several SWMSs, such as Kepler, VisTrails, OpenAlea, exploit intermediate data for workflow re-execution. Each of these systems has its unique way of addressing data reuse. OpenAlea [130] uses a cache that captures the intermediate results in memory and dynamic scheduling algorithms that exploit the cache. The scheduling algorithm traverses the workflow from the bottom activities to the input activities. For each activity, the algorithm checks the activity input data and if it is unchanged, accesses the cached data. The remaining activities (those that still require to be executed) are greedily scheduled on a single VM. Whenever a task is executed, it is stored in the memory of the VM. The scheduling and cache are centralized, thus not adapted for data-intensive workflows.

VisTrails provides a cache when executing a workflow [15]. The intermediate data generated is stored and can be reused in future executions of the workflow. The intermediate data is cached in a provenance database and is only accessible for the user that produced it. As the intermediate data is linked to the provenance data, whenever the workflow is re-executed, it only requires one query per activity for both the provenance and the cached databases. This approach allows the user to change parameters or activities in the workflow and efficiently re-execute each workflow. The database is hosted on the user computer, thus not adapted for data- or compute-intensive workflows.

Chen *et al.* [32] propose an algorithm for Kepler to select the best datasets to store in order to enable "smart rerun" of workflows. It aims at minimizing the monetary cost for workflow execution, taking into account the trade-off between storage and computation costs. The trade-off is computed from the provenance data by an heuristic based on ant colony system optimization, which finds a near optimum solution. Then, the SWMS can statically schedule workflows using this cache. The cache is hosted on a centralized server, and can be accessed for cloud computation. The intermediate data is persistent, which enables the user to benefit from cached data generated a long time ago. However, the solution does not consider evolving workflows, *i.e.* the workflow that is re-executed has to be the same. It also does not take data transfer into account, and the cached data is transferred to the cloud before execution.

Evolving Workflows

It is common for workflow users to reuse code or data from other workflows and from previous executions of the same workflow [58]. For instance, the workflow has already been executed but some activities have changed, *e.g.*, by new versions of the code they represent. Or the workflow is reused in another scientific application, and some of the activities have already been executed. The SWMS can cache some intermediate data to save time and money when the activities are re-executed. The evolving workflows require that the SWMS keeps track of workflow evolution and can involve several users among different teams. Figure 2.8 shows the execution of two workflows *Wf1* and *Wf2*. When *Wf1* is executed, its provenance and intermediate data generated are cached. *Wf2* is an new workflow based on *Wf1* with two new activities. When *Wf2* is executed, the provenance evolution shows that the intermediate data previously generated can be reused.

Joeris *et al.* [78] present a versioning model for workflows that captures workflow evolution and workflow configurations. The model is based on schema versions and considers five operations to merge, propagate and share schemas. A schema is associated with each workflow and captures the modifications on the workflow activities, even when only fragments of the workflow are reused by another workflow. When a workflow is executed, its schema is analyzed to find the activities that are not changed (in "lazy" state) from previous executions. The "lazy" activities are not re-executed.

Freire *et al.* [54] present the evolving workflow provenance in Vistrails. VisTrails provides visual analyses of workflow results and captures the evolution of workflow provenance, *i.e.*, the steps of the workflow at each execution, as well as the intermediate data from each execution [28]. The intermediate data can be stored in either a relational database or in a file system. Either way, the storage is centralized and can be shared between users. This approach allows the user to efficiently re-execute a workflow when she changes or updates some activities. Yet, the reuse of cached data is only available in centralized computing environments. Thus, it is not suited for data-intensive workflows as it does not scale in distributed environments.

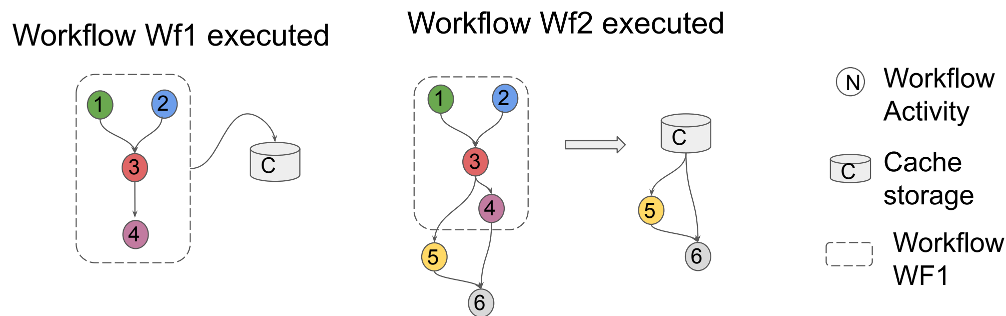


FIGURE 2.8: Evolving workflow: execution of two workflows *Wf1* and *Wf2*, with *Wf2* being designed on top of *Wf1*.

The VisTrails solution for caching intermediate data has been extended to generate "strong links" between provenance and execution [85] [50]. The intermediate data cache is associated with provenance to enhance reproducibility, and the intermediate data that has been cached is always reused. Cala *et al.* [27] propose a reusable provenance-based approach to optimize workflow re-executions in the case of evolving versions of the software used. The model computes a "restart tree" that captures the evolution of the workflow when executed. Then, it uses *ReComp* [105] a meta-process that enables re-computation of processes such as workflows. The intermediate data can be stored in a distributed data store over a single cloud site and the solution is adapted for data-intensive workflows.

Multiuser Workflow Sharing

During scientific experiments, multiple users may work on the same workflow, on the same dataset, or both. Some of the workflow fragments or activities may be common between users. When the workflow is executed, each user re-execute the activities common with the other. Multiple users can also work on sub-workflows that have some common activities, or fragments. Moreover, apart from intermediate data, users can also share workflow information through the User Service Layer (see Section 2.2.2), including workflow design, input data, provenance data.

Figure 2.9 shows the architecture of Kepler that enables intermediate data sharing among users. Users communicate to the SWMS through a Web Portal (the user interface), then all the workflow information, input, intermediate and output data is handled by the data management module. All the data is stored in a centralized database. The intermediate data is made persistent and accessible by other users, given access rights.

Zhang *et al.* [172] propose *Confucius*, a tool for collaborative workflow management. The framework is based on a service-oriented collaboration model to enhance single user SWMS environments into collaborative environments. User information is stored in a centralized database on a server. Then, users can join in to a peer-to-peer collaboration when executing and managing workflows. Whenever a user wants to change the workflow, she requests a token that prevents concurrent changes from multiple users. The user also

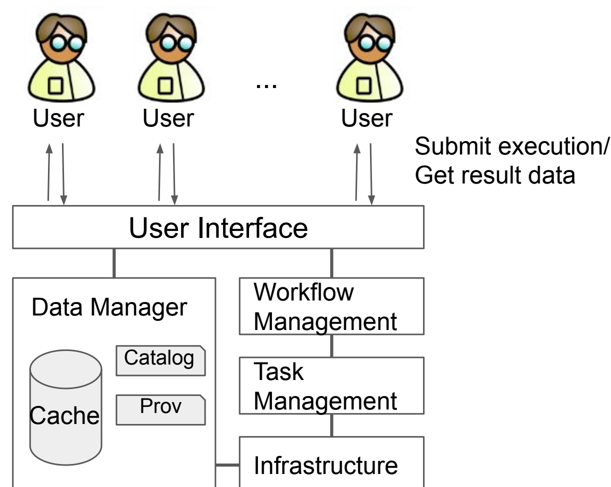


FIGURE 2.9: Multi-user SWMS architecture with a *Data management* module that enable intermediate data reuse between users [32].

takes a token when she executes the workflow. After the workflow changes or the workflow execution, the user releases the token. Once she releases the token, each user can access the workflow and the data stored in the shared database. The solution only provides sharing of small intermediate data, workflow design, and provenance data, thus is not adapted for data-intensive workflows re-execution.

Mates *et al.* [102] propose *CrowdLabs*, a system that can be added to an existing SWMS to enable workflow information sharing (workflow design, provenance data, architecture, input data). It is presented as a "Social Web Site" where users can share provenance data, workflow architecture, and intermediate data. It has been implemented and tested on VisTrails. Users access the system through a web client API hosted on a CrowdLabs server. Whenever a user submits a workflow execution, the CrowdLabs server checks for data that can be reused, then, it transfers to a Vistrails server the WEP that the SWMS executes. The workflow execution generates data, stored in a remote cache server. The CrowdLabs server also hosts a social data database which is a catalog of user information and access rights on the cached data. The cache is stored in a centralized MySQL database hosted in a server, thus is not adapted for data-intensive workflows.

Galaxy [61] provides an interface for the user to share intermediate data generated by workflow executions. After a workflow execution, a user can decide to upload the intermediate data that has been generated. Then, from the Galaxy interface, users can access a library of datasets, workflows, and intermediate data. Users have to manually find and select the cached data they want to reuse. Once a user finds cached data she wants to reuse, she adds to her workflow a reference to the cached data. Then, when the workflow is executed, it will fetch the cached data automatically.

2.4.3 Scheduling and Caching in SWMSs

To efficiently execute a data-intensive workflow in the cloud, the SWMS should strive to exploit data locality, provenance data, and resource capacities. Since the intermediate data generated by a data-intensive workflow is also data-intensive, cache management is critical to take advantage of caching. Thus, when using cached data, the SWMS needs to add one more dimension to the scheduling problem. In this section, we discuss existing solutions to perform workflow scheduling and workflow data caching in both monosite and multisite clouds.

Monosite Cloud

In a monosite cloud environment, during a workflow execution the input data and intermediate data do not require to be transferred among data centers. Yet, data-intensive workflow executions benefit from data location aware scheduling [37]. Many scheduling algorithms have been adapted for data-intensive workflows in monosite cloud execution, including matrix-based heuristics [170], particle swarm optimizations [121], and optimizations based on evolutionary approaches [87]. Yet, these approaches do not consider reusing cached data during workflow execution.

In monosite cloud, the cost of adding a cache for workflow data includes the storage cost, the writing and reading cost and the latency. Having a cache storage and caching data adds a cost (both in terms of time and money) to the workflow execution. Yet, it reduces the execution cost as the intermediate data reused is not re-executed. But the benefit of caching intermediate data depends on the number of times the cached data will be reused, which is usually not easy to predict. Thus, the trade-off between the costs of re-executing tasks and the costs of caching intermediate data is not easy to estimate [2, 48].

Yuan *et al.* [169] propose an algorithm based on the ratio between re-computation cost and storage cost at the task level. The algorithm is based on a graph of dependencies between the intermediate datasets, generated from the provenance data. Then, the cost of caching each intermediate dataset is weighted by the number of dependencies in the graph. The algorithm computes the optimized set of intermediate datasets that need to have minimum cost. The algorithm is improved in [171] to take into account workflow fragments. Both algorithms are used before the workflow execution, using the provenance data of the intermediate datasets. They provide near optimal caching intermediate datasets selection. Yet, both algorithms do not take data transfer into account and focus on centralized storage.

Kepler [7] provides intermediate data caching for single-site cloud workflow executions. It uses a remote centralized relational database where intermediate data is stored after workflow execution. Two steps are added when executing a workflow. First, the cache database is checked and all intermediate cached data is sent to a specific cloud site before execution. To reduce storage cost, the intermediate data that need to be cached is determined based on how many times the workflow will be re-executed in a given period of time [32].

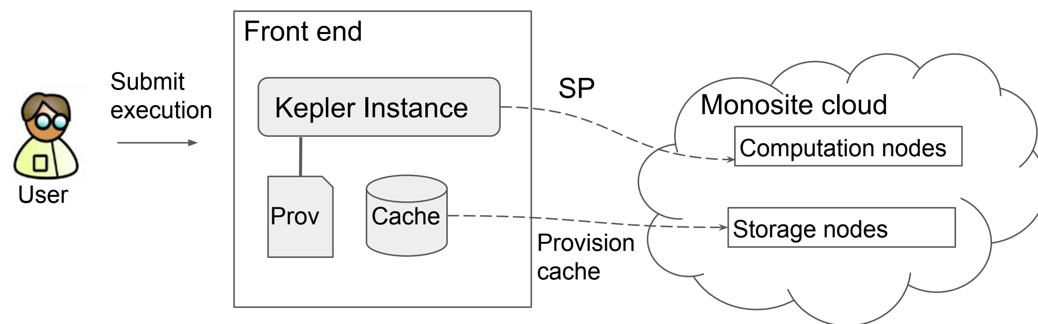


FIGURE 2.10: Kepler architecture in monosite cloud with cache feature [32].

Figure 2.10 shows the architecture of Kepler in the cloud using a cache. A user submits a workflow execution to the Kepler front-end. Kepler's Caching/ Reuse Decision manager checks the provenance and generate a scheduling plan accordingly. Then, before the execution starts, the cached data is transferred to the cloud site storage. The cache management is static, then, the scheduling can be dynamic.

Other approaches propose solutions for caching data in MapReduce workflows. Zhang *et al.* [177] use the *Memcached* distributed memory caching system to cache the intermediate data between Map and Reduce operations. This approach focuses on a single MapReduce task, and the cached data is not persistent and reused across executions. This solution is improved by Elghandour *et al.* in [52]. They propose ReStore, a system to manage and cache intermediate data of MapReduce tasks for future reuse. Whenever a workflow is executed, the intermediate data of MapReduce tasks are stored in a distributed file system (in this case the cloud where the MapReduce is executed). The intermediate data generated is associated with a unique name, which enables ReStore to identify the input data and Map task that produced the data. Then, before executing a new workflow, ReStore checks the existing cached data and adapts the MapReduce workflow to the cached data. As the cached data is stored at the task level, when executing a MapReduce workflow on a different set of input data, if some subset of input data has already been computed, the workflow can reuse it. However, task level caching requires that for each task, the cache is queried, which can lead to cache latency.

Olston *et al.* [117] propose a caching strategy on top of the Pig language to execute workflows in a monosite cloud. The strategy consists of replicating and storing on a compute node any new input data that is transferred for a task execution, as well as the intermediate data generated by the execution. After several executions, some tasks have been executed on different nodes, thus, generating the same cached data replicas on each of these nodes. Then, based on an eviction policy, the less useful replicas are removed from the cache. This approach aims at dynamically balancing the amount of data cached. Whenever some popular tasks are executed, more compute nodes will save a replica of the intermediate data. Then, once these tasks are no longer re-executed, most of the replicas are deleted. The solution however

does not consider the workflow scheduling as a part of cache management. Thus, executing the same data-intensive workflow several times will lead to either a bottleneck of the execution on the compute node with the cached data, or huge data transfer times between the nodes.

Multisite Cloud

In a multisite cloud, the latency between the sites is significantly higher than the latency of transfer inside a cloud site. Thus, to efficiently share workflow information and intermediate data in a multisite cloud, data location, data accessibility, and latency become critical.

Zhang *et al.* [175] propose a static data-oriented scheduling method for data-intensive workflows in multisite cloud. The input data is distributed among the sites. It first determines the best location for the intermediate data that minimizes the data transfers from the initial data location and the data dependencies. Then, it schedules the tasks on the sites and finds the tasks that need to be replicated to minimize the volume of intermediate data transferred. The algorithm does not consider the dependencies between the tasks when scheduling them, which may result in more data transfer. The algorithm is adapted for homogeneous multisite clouds.

Ji *et al.* [95] propose three dynamic multisite greedy scheduling algorithms for data-intensive workflows. The algorithms are based on a multi-objective cost function, that consider time costs and financial costs. The three algorithms are: a data location based scheduling (*LocBased*), a site greedy scheduling (*SGreedy*) and an activity greedy scheduling (*AGreedy*). *LocBased* schedules the activities on the site that minimize the data transfers. It is an efficient algorithm for workflow that are not compute intensive as the data transfers represents most of the total cost. *SGreedy* dynamically schedules the activities on the first site with available computation resources. This algorithm is most adapted for compute-intensive workflows, as it minimize the idleness of computing nodes. Yet, it leads to many inter-site data transfers. *AGreedy* dynamically schedules the activities on the site that minimize the cost function. *AGreedy* efficiently schedules data- and compute-intensive workflows, as it find a near optimal scheduling plan. The solution is also very scalable in the number of activities in the workflow.

In the two multisite scheduling solutions [175][95], the SWMS uses data locality to efficiently schedule the workflow by scheduling the activities where the data is. Intermediate data can be reused by several tasks during workflow execution. Yet, the intermediate data is not cached and the workflow execution does not benefit from cached data.

Vulimiri *et al.* [155] propose *WANalytics*, an Hadoop based system that manages data-intensive workflows in a multisite cloud. The system is available to be installed on top of any SWMS. It analyzes workflows before their execution and minimizes data transfers during execution by taking into account the workflow workload and the dataset sizes. The solution enables to automatically cache all intermediate data produced at the site where it is generated. It also sends a message to all other sites where this data is stored,

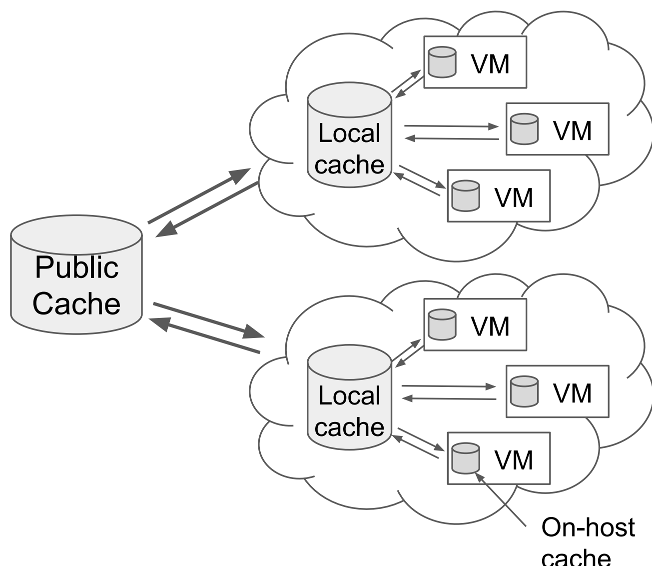


FIGURE 2.11: Three level cache architecture in multisite cloud [132].

so each site that already has the data can update its cache metadata. To reduce inter-site transfers, once some intermediate data is regenerated, the site only computes a *diff* between the old and new intermediate data. Then, only the *diffs* are transferred between the sites. The solution store cached data replicated among the site for efficient workflow rerun, but it does not consider the storage costs, that can be huge when dealing with data-intensive workflows.

Qasha *et al.* [132] propose a framework to execute and share workflows in a multisite cloud. The framework enables users to automatically deploy and provision workflows in the cloud. It proposes a three-level cache for tasks. Figure 2.11 shows the architecture proposed. The first level cache is on-host within Docker containers. The second level cache is the local cache, shared within an organization. The third level cache is the public cache, accessible by everybody. The cache store workflow and task information, but no input, output, or intermediate data.

2.5 Conclusion

In this chapter, we discussed the current state of the art in scientific workflow management in the cloud, with a focus on caching and scheduling. First, we introduced the basic concepts of workflow management, including a functional architecture of SWMSs. Then, we discussed deployment, scheduling, and execution of workflows in both monosite and multisite cloud. Finally, we discussed data caching and scheduling of workflows with cached data in the cloud.

Most SWMSs (such as SciCumulus, Chiron, Pegasus) that manage data-intensive workflow executions in multisite cloud, focus on workflow scheduling. The solutions include data location aware, multi-objective, dynamic scheduling approaches. Yet, they do not consider caching and reusing intermediate data during execution.

One SWMS (WANalytics) considers caching and sharing data for a single user in a distributed environment. The solution schedules workflow tasks in a way that minimizes inter-site data transfers. However, the data is always stored where it is generated. Thus, this caching solution does not scale up with multiple users. The cached data is also associated to a single workflow.

Some SWMSs (OpenAlea, Vistrails, Kepler) manage a cache for intermediate data using a centralized storage. They can execute workflows in distributed environments, but typically ignore data transfers.

The approaches that focus on multiuser platforms, such as MyExperiment), enables sharing workflow's intermediate data and execution between users. Yet, these solutions are based on users' interactions when handling cached intermediate data. The solutions are not adapted for data-intensive workflow scheduling.

Finally, there are studies on the trade-off between storage and computation costs for workflows but they do not consider data transfers and data location. Thus, they are not adapted for multisite cloud.

To the best of our knowledge, none of the works presented in this chapter address cache and scheduling management for data-intensive workflows in monosite clouds and multisite clouds in a distributed approach.

Chapter 3

Use Case in Plant Phenotyping

Plant phenotyping aims at capturing plant characteristics, such as morphological, topological and phenological features. Manually capturing the phenotype of a plant is time consuming and yields low throughput [158]. High-throughput phenotyping (HTP) platforms have emerged to speed up phenotyping data acquisition in controlled conditions (*e.g.* in a greenhouse) or in the field [147]. Such platforms generate terabytes of data that are used in plant breeding and plant biology to test novel mechanisms [147]. These datasets need to be curated, analyzed, and visualised to extract knowledge using data-intensive and computational intensive analyses and simulations. To be processed in a relatively short time, such computational analyses require very large distributed computational infrastructures. However, they produce huge quantities of data that are hard to manage [84]. For this purpose, scientific workflows provide a convenient solution to execute, share, reproduce such analyses and simulations.

In this chapter, we present a real use case in plant phenotyping based on the Phenomenal workflow [13] from OpenAlea [1], a widely used scientific workflow management system (SWMS) for plant science. This use case is the basis for our motivations in this thesis and will be used in our experimental validation. The use case includes data generated by the HTP platform PhenoArch in Montpellier, in the context of the French Phenome project [2] with data- and compute- intensive analyses.

Section 3.1 gives an overview of HTP and presents some challenges. Section 3.2 presents the phenotyping modules of OpenAlea, in particular, the Phenomenal library, a set of components for automatic plant phenotyping from images. Then, we present the Phenomenal workflow, a workflow created from the Phenomenal library components. Finally, section 3.3 concludes.

3.1 High-throughput Plant Phenotyping

3.1.1 Context

Plant breeding is one of the oldest agricultural techniques used to increase crop performance [6]. The phenotype of a plant concerns the macroscopic features determined by both genetic makeup and external environment. The

[1] http://openalea.gforge.inria.fr/dokuwiki/doku.php
[2] https://www.phenome-emphasis.fr/phenome_eng/

genotype of a plant is the sum of all its genes. The plant breeding process consists in selecting seed according to their phenotype in order to choose the most suited plant in a particular environment [6]. At organ and plant scale, there are many indicators linked to the phenotype, such as the topological and geometrical description of the plant such as the number of leaves and tillers, leaf geometry (*i.e.* shape, dimension and orientation), the number of organs (leaves, branch), plant height and phenology. Originally, these indicators were measured manually, which was a very long process [145]. The number of individual plants measured in various conditions was limited. Plant breeders use phenotype observations at the field scale to select varieties that are productive or resistant in a given environment. However, the plant breeding techniques are starting to show some limits: progress in yields of some species stagnates (*e.g.* wheat [21], corn [135], or rice [89]).

New approaches, such as genomic selection, try to link the genotype of a plant to the crop performance. One of the difficulties is to link the sets of genomic markers to phenotypic responses [71]. The current method consists in monitoring the phenotype of a plant knowing its genotype. As the phenotype also depends on the environmental influence, the experiment must include many plants and have to be performed under the total control of the climatic conditions.

During the last decade, genotyping data has been collected and analyzed at faster and faster rates with the improvement of computing platforms. The research on selective breeding, based on phenotype and genotype analyses, becomes limited by the bottleneck of phenotyping data generation [10]. High-throughput phenotyping (HTP) platforms have emerged to speed up phenotyping data acquisition. They have allowed the acquisition of quantitative data on thousands of plants in well-controlled environmental conditions. These platforms produce huge datasets (hundreds of Terabytes a year) of heterogeneous data (images, environmental conditions, and sensor outputs) and generate complex elaborated variables with in-silico data analyses. Data from fields can be added to the one produced by these platforms.

The data generated by HTP platforms is used in various analyses, such as plant modeling, plant growth prediction, or plant-plant interactions study. The lifecycle of such analyses is a loop of reuse and redo as they are used for further analyses on different input data, and parameters [79]. Through time, as new research questions arise, these analyses are updated with new models, new implementations, and new data. Some unchanged resulting data of the oldest analyses can be reused in new ones to reduce execution time, financial cost or energy consumed.

Phenotyping analyses are both data- and compute-intensive and thus require to be executed in high-performance computing environments. The user needs to be able to transparently manage the execution of the analysis on such infrastructures. A common solution is to use a SWMS to manage and perform analyses [128]. These analyses are scientific computational experiments used to decipher genetic and environmental impact on plant growth and functioning. Provenance data describes the chain of reasoning used to derive the results, which provides additional insights when the findings are reproduced

or extended for other experiments [35]. Thus, as a scientific artifact, the reproducibility of the analysis results is critical. The provenance data eases the reproducibility of the analyses. Yet, it is often not sufficient to reproduce the results. Indeed, other elements, such as database updates, distributed environments, or different code versioning, make the reproducibility challenging. As sharing intermediate results is increasing the link between provenance and execution data, it increases the reproducibility of the experiment [85]. Thus, tracking, saving, and sharing intermediate data improve existing analyses and ease the development of new ones.

3.1.2 High-Throughput Phenotyping Platforms

A High-throughput Plant Phenotyping (HTP) platform is a general denomination for an automated tool that enables the capture of phenotyping data of a large number of plants. There are platforms to study plants in controlled conditions (*e.g.* greenhouse platforms) or in the field, where environmental conditions are difficult to control (field platforms). Field platforms are usually mobile vehicles that go across the field and capture data (such as the field platform at ETH, Zurich [83]), but they can also be hard structures that monitor a part of the field (such as Field Scanalyzer [153] or Heliaphen [62]). The field based platforms capture data on plants in a more realistic environment than in greenhouses. However, the environment in greenhouse platforms is more controlled and monitored.

Greenhouse platforms enable the production of reproducible and precise phenotypes of early traits data [104]. The greenhouse controlled environment (soil and air) enables experiments on specific scenarios (such as water deficit, or nutriments deficit) while capturing precise plant responses. Such experiments enable scientists to study the phenotype responses of selected genotype plants [123].

HTP projects are federated into national or regional networks all over the world, including the European Network of Pilot Production Facilities (EPPN) in Europe, the North American Plant Phenotyping Network (NaPPN) in North America, the Latin American Plant Phenomics Network (LatPPN) in South America, and the Australian Plant Phenomics Facility (APPF) in Australia. These international projects include many different HTP platforms. As an example, Emphasis [3], a European project of EPPN, federates platforms and data repositories from dozens of European countries. Figure 3.1 shows the different platforms across Europe. The platforms are grouped as: field platforms in green, greenhouses in yellow, and intensive fields in gray (which are field with hard structure platforms).

All these platforms use different sensors, including lidar, sonar, camera, and chemical sensor. They generate huge datasets of a range of very different data types. The data generated also covers different scales: temporal and spatial. The data cover many temporal scales with hourly interactions, such as water interaction on a plant, to seasonal interactions, such as field evolution. The data also cover different temporal scales, from milliseconds with cell

[3] https://emphasis.plant-phenotyping.eu/

FIGURE 3.1: Map of Emphasis project platforms across Europe.

interactions to hours or months with global field interactions. The diversity of the data produced adds information on the phenotyping but also difficulties in the analyses.

The generation of data by the HTP platforms is expensive and time-consuming. Indeed, it is limited by the plant growth, platform capacities, and is dependent on many external parameters. Thus, the datasets are very valuable and many teams (users) work on the same datasets, that serve as baselines.

3.1.3 Infrastructures

Phenotyping projects include data centers and computational infrastructures. The data generated by a platform is stored in a geographically close data center to minimize data transfer time, which can be huge in HTP. For instance, the seven facilities of the French Phenome project [4] produce each year 200 Terabytes of data. The raw data includes various data types (images, environmental conditions, and sensor outputs). The raw data is multiscale (at the plant level, molecular level, or field level) and comes from different data centers.

Figure 3.3 shows the PhenoArch platform in Montpellier [5]. The plants are grown in pots and the platform capacity is 1600 plants at same the time.

[4] https://www.phenome-emphasis.fr/phenome_eng/

[5] https://www6.montpellier.inrae.fr/lepse/M3P/Plateformes/PHENOARCH

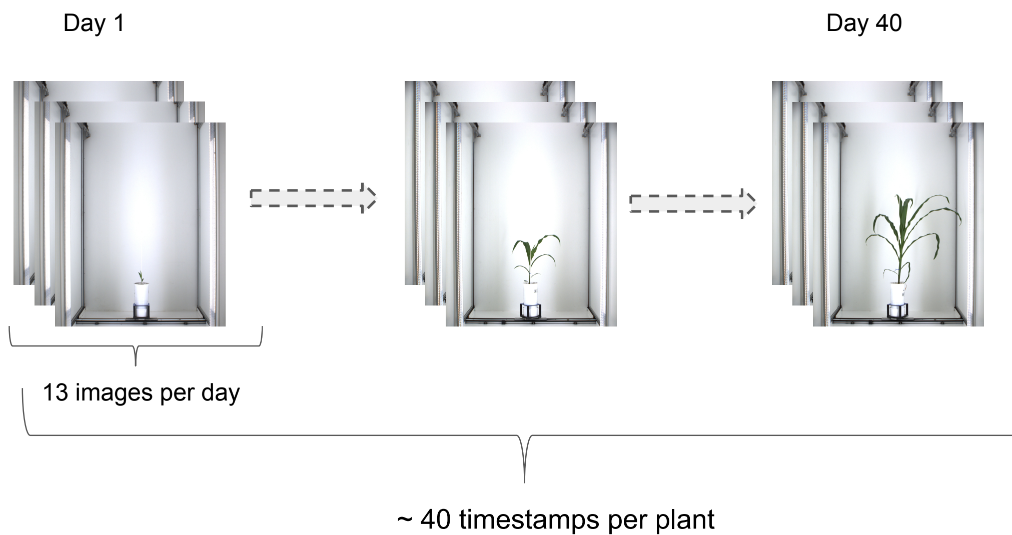


FIGURE 3.2: Time series of images for one plant, generated during one HTP experiment.

The pots are conveyed in a small room, where they are pictured. Each pot is photographed 13 times per day. Moreover, the pot is weighted and the amount of water given is captured. Figure 3.2 shows the raw data for one plant generated by the platform during one HTP experiment. The platform generates data during experiments that last for two to three months, where plants are grown from the seed to a given state of growth. There can be one or two experiments per year. During one experiment, each plant is pictured every day from 13 angles. For one plant, the data is a time series of sets of 13 images for each day of the experiment. The total size of the raw image dataset for one experiment is 11 Terabytes, which represents about 80000 time series of plants and about 1040000 images.

The data generated is stored in data centers and usually follow ontology standards (such as the Breeding Application Programming Interface (BrAPI) [142] for genomics/ phenotypes, or the Minimum Information About a Plant Phenotyping Experiment (MIAPPE) for phenotyping experiments [122]). Some ontologies are both multiscale and multisource, such as in the Phenotyping Hybrid Information System (PHIS) [108] that enriches datasets with knowledge and metadata from integrating and managing data from multiple experiments and platforms. Thus, integrating phenotyping data from different sources (platforms, and teams) is difficult because the data is geodistributed over several data centers. Analyzing such massive, geodistributed datasets is an open, yet important, problem for biologists [147].

A classical execution environment is data centers with raw HTP data and a private computing platform (such as a department cluster or a private cloud). Usually, the data generated by the PhenoArch platform is computed in a cluster hosted by the research team that manages the platform. Scientists rely on frameworks to schedule and execute their experiments, when they use workflows the SWMS handles it. In the case of the PhenoArch data, OpenAlea

FIGURE 3.3: PhenoArch platform.

is the most common SWMS with InfraPhenogrid, its distributed execution engine [128]. However, the local cluster has limited computing resources and some experiments require to be computed on other environments. The phenomenal projects usually enable access to computational infrastructures from associated projects, such as Grid'5000 [6] or IFB-cloud [7].

Data captured by the HTP platforms evolves, as the sensors and platforms do. The data gain in precision, quantity, and metadata. The analyses also evolve with models and implementations. Yet, both of these evolutions can be independent [138]. Usually, the improvement of a model is tested on a sub-dataset, and datasets are updated as the plant experiments are performed. To be kept up to date, the model evolutions must be executed on new datasets. But the acquisition of new datasets does not imply that all data has changed and most of the previous data can remain up to date. However, determining what new model should be executed on which new data is not obvious. Provenance on the executions can link the model improvements and data improvements. Data ontology and annotations can be a solution to improve provenance data as they add information to the metadata.

3.2 Automatic Phenotyping in OpenAlea

Retrieving phenotyping data of a plant from raw HTP data is not a trivial problem. In the case of images, some computation is required to determine useful phenotype features. The Phenomenal library [12] [8] proposes an automatic phenotyping workflow that reconstructs 3D plant architecture from

[6]https://www.grid5000.fr/w/Grid5000:Home
[7]https://www.france-bioinformatique.fr/cloud-ifb/
[8]https://github.com/openalea/phenomenal

multi-view images. The library is developed for OpenAlea. Several phenotype features can be extracted using this workflow from raw images such as plant height, number of leaves, and their shapes.

In this section, we present the OpenAlea SWMS. Then, we present in detail the Phenomenal library, the analyses it enables, and the intermediate data it generates. Finally, we present some workflows that use some activities or fragments proposed by the Phenomenal library, highlighting the opportunities for data sharing.

3.2.1 OpenAlea

OpenAlea has been in constant use since 2004 by users of the plant science community both in France and in other countries. The system has been downloaded 820 000 times and the web site has about 10 000 unique visitors a month according to the OpenAlea web repository [9].

The OpenAlea tools (*e.g.*, models, workflows, components) are published and shared on the web both through the OpenAlea web repository and through the web sites of groups that use and contribute to OpenAlea without being formal partners of the OpenAlea project. As a result, more than 60 researchers have contributed to OpenAlea packages, in France and internationally, published through large meta-packages (*e.g.*, Alinea to simulate ecophysiological and agronomical processes and VPlants to analyze, model and simulate plant architecture and its development) to ease the installation for end-users. End-users can also create new tools or packages by implementing them in Python, and then share them with the OpenAlea community. It is also possible to wrap existing functions into an OpenAlea activity.

End-user access the OpenAlea libraries and workbench through visual programming. They manage the workflow activities and dependencies by drag and drop. They can specify execution parameters for each activity through this interface, before starting execution. The user also selects an OpenAlea Evaluator (a model of computation), which is used by the execution engine to interpret the workflow [130].

Once the workflow is ready to be executed, the user selects the activity whose output data she requests. Depending on the Evaluator, the workflow is traversed and a specific scheduling plan is generated. Once the scheduling plan is executed, *i.e.* the requested data is generated, the user is notified. OpenAlea also enables the user to label activity as *blocked* or *lazy* to optimize workflow executions. If an activity is blocked, the execution is not propagated to the upstream sub-workflow. If the activity is lazy, the execution is performed only if the activity's inputs have not changed compared to its previous execution.

3.2.2 Phenomenal Library

The Phenomenal library [12] has been developed in OpenAlea to analyze and reconstruct the geometry and topology of thousands of plants through time

[9] https://gforge.inria.fr

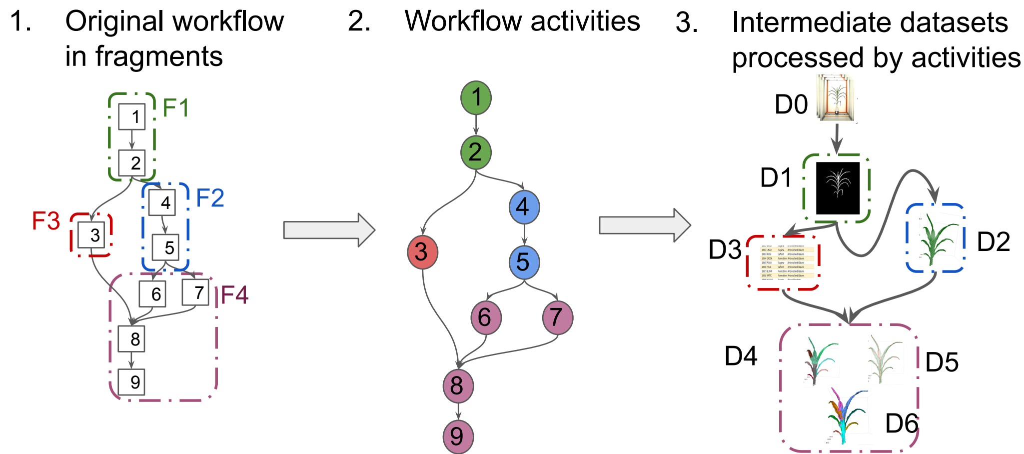


FIGURE 3.4: Phenomenal workflow and its generated intermediate data.

in various conditions. The Phenomenal library provides 1) a completely automatic analysis including data import, reconstruction of 3D plant architecture for a range of species and quantitative measurements on the reconstructed plants; 2) an open source library for the development and comparison of new algorithms to perform 3D shoot reconstruction; and 3) an integration framework to couple workflow outputs with existing models towards model-assisted phenotyping. The library is continuously evolving with the addition of new state-of-the-art methods, thus yielding new biological insights.

The Phenomenal workflow is constructed from the components of the Phenomenal library [12], shown in Figure 3.4. It automatically reconstructs 3D model and organ segmentation from plant images. The workflow (Figure 3.4.1) is composed of four different fragments that implement various algorithms: binarization (circled in green), 3D volume reconstruction (blue), image calibration (red), and organ segmentation (purple). Figure 3.4.2 gives an abstract representation of the workflow, in terms of activities where the activities from the same fragment have the same color. The intermediate datasets, processed by the workflow activities during execution, are shown in Figure 3.4.3. Dataset $D0$ is the dataset of raw data that serves as input for the first fragment. Dataset $D1$ is generated by activity 2 and is the input of fragment 2 as it is processed by activity 4. The datasets are grouped by fragments.

During its execution, the workflow generates heterogeneous intermediate data such as raw RGB (Red, Green, Blue) images, 3D plant volumes, tree skeleton, and segmented 3D mesh. Figure 3.5 shows the intermediate data generated by various activities applied on the raw images from a maize:

- The *Binarize* activity separates plant pixels from the background in each image. It produces a binary image from an RGB image. The binary images are compressed by several hundreds fold compared with raw images, thus analyses usually start from the binaries as they are easier to manage. This activity is often reused as most plant analyses start from it. The binarize activity generates the data $D1$ in Figure 3.4.

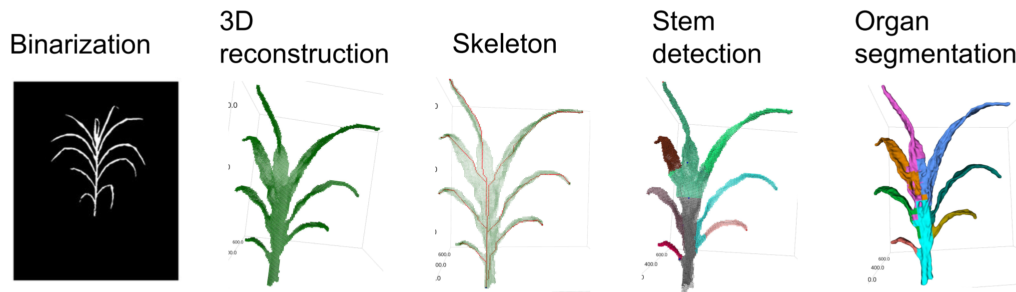


FIGURE 3.5: Intermediate data generated by each fragment of the Phenomenal workflow on one genotype through time.

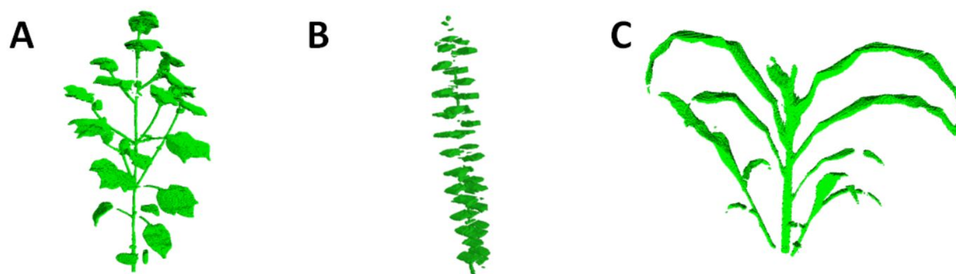


FIGURE 3.6: 3D reconstruction of (A) Cotton (Gossypium); (B) Apple tree (Malus pumila); (C) Sorghum (Sorghum bicolor).

- The *3D reconstruction* activity produces a 3D volume based on 12 side and 1 top binary images. It generates *D*2.
- The *Skeletonization* activity computes a skeleton inside the reconstructed 3D volume. It generates *D*4.
- The *Stem detection* activity computes a main path in the skeleton to identify the main stem of cereal plants (*e.g.*, maize, wheat, sorghum). It generates *D*5.
- The *Organ segmentation* segments the different organs on the skeleton after removal of the main stem. It generates *D*6.

The workflow has been designed to phenotype maize but also other plants grown in different conditions. Figure 3.6 shows the intermediate data produced by the workflow on (A) Cotton (*Gossypium*); (B) Apple tree (*Malus domestica*); (C) Sorghum (*Sorghum bicolor*). The workflow is used either on different genotypes of the same plant and on different plants. Thus, it increases the number of users that executes the workflow on different datasets, generating intermediate data that could be useful for other users.

Figure 3.7 shows the execution time and intermediate data size of the activities of the Phenomenal workflow when executed on one HTP experiment dataset. The intermediate data generated by the execution of the workflow on one raw dataset represents almost 10 terabytes of data. The total execution time (on one CPU) is around 1500 hours.

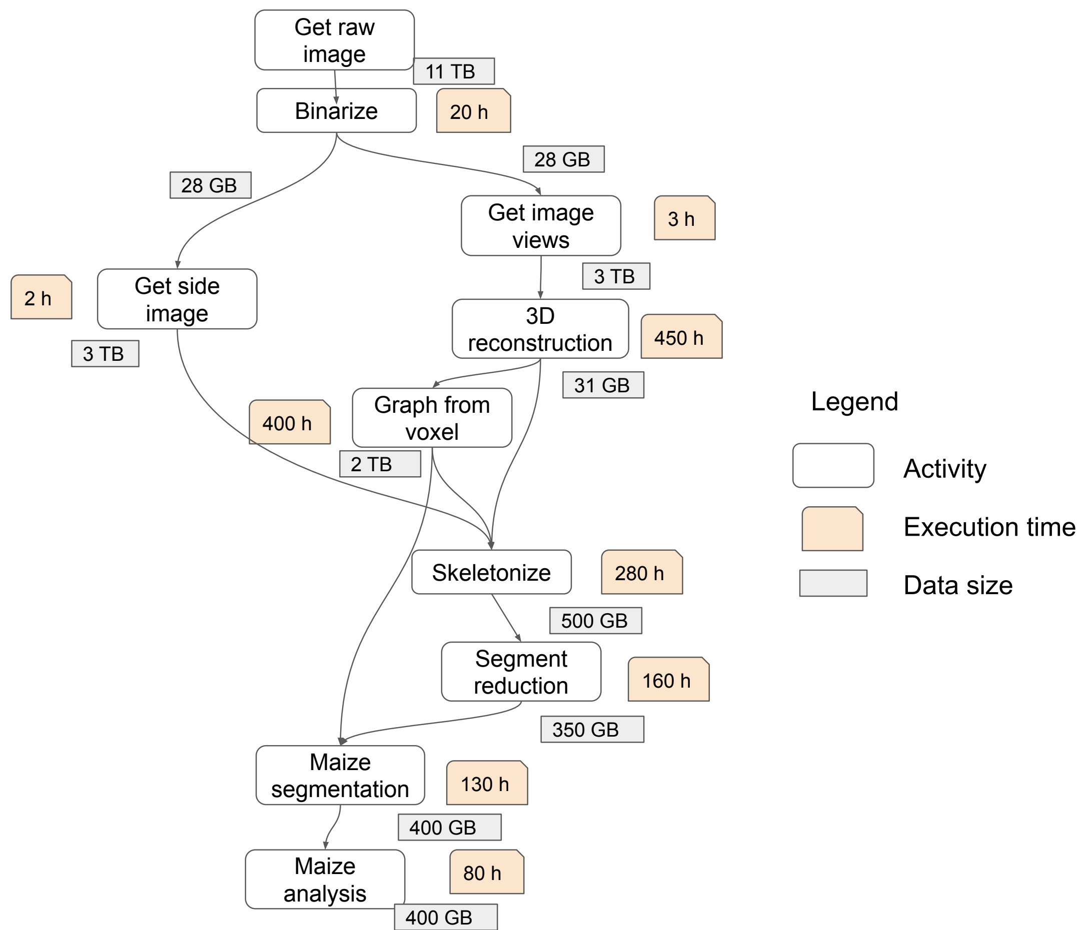


FIGURE 3.7: Order of magnitude of intermediate data size and execution time of each activity of the Phenomenal workflow executed on one HTP experiment dataset.

3.2.3 Applications with Common Data

It is common for biological analyses to use the same raw datasets, as they are costly and slow to produce. Moreover, some plant genotypes and species (such as maize, sorghum) serve as a baseline for analyses, which leads to a wide usage of datasets on such plants. At least, scientists want to compare their results on previous datasets and extend the existing workflows with their own developed activities or fragments. In the case of the Phenomenal workflow, several datasets of intermediate data can be useful in other applications, mostly the binarization and the 3D reconstruction. Several published analyses share common intermediate datasets with the Phenomenal workflow [26, 20, 128, 88, 31].

Figure 3.8 shows the Phenomenal workflow with three variants used for different analyses that have been published in [26, 31, 20], with the intermediate data generated by the workflow fragments. The workflows share common fragments and use the same intermediate data, namely binarization and 3D reconstruction. Then, each of them uses the intermediate data for a different purpose, and the workflows generate data of different scales. Indeed, $Wf1$ and $Wf4$ can be applied at a single plant level, while $Wf3$ and $WF4$ model thousands of plants at the field level. The three applications are: $Wf2$, a light competition workflow [26]; $Wf3$, a light interception and radiation workflow [31]; and $Wf4$, a maize ear detection workflow [20].

These workflows are executed with the same datasets, which are raw HTP images generated by the PhenoArch platform. They share some intermediate data, which has been re-generated for each of these analyses. Based on the execution times presented in 3.7, executing the fragments $F1$ three times and $F2$ once, 512 hours of computations could have been saved by reusing cached data.

3.3 Conclusion

In this chapter, we presented a real use case in HTP based on the Phenomenal workflow from the OpenAlea scientific workflow management system (SWMS) We first introduced HTP and gave an overview of HTP applications with data-intensive and computational intensive analyses. Then, we presented the Phenomenal workflow, a phenotyping workflow that enables the automatic reconstruction of 3D plant architecture for a range of species and quantitative measurements on the reconstructed plants. Finally, we presented the intermediate data generated by the Phenomenal workflow and how they can be reused in other published analyses.

In this thesis, we use the OpenAlea SWMS, is widely used in the plant analysis community, to implement and validate all approaches to intermediate data caching and workflow scheduling. The Phenomenal workflow, which is both data- and compute- intensive, is representative of many workflows used for plant analyses as it is composed of various activities that: compress the data, expand the data and require lots of memory and CPU. Thus, we use it as a basis for our experimental validations. The experiments are run

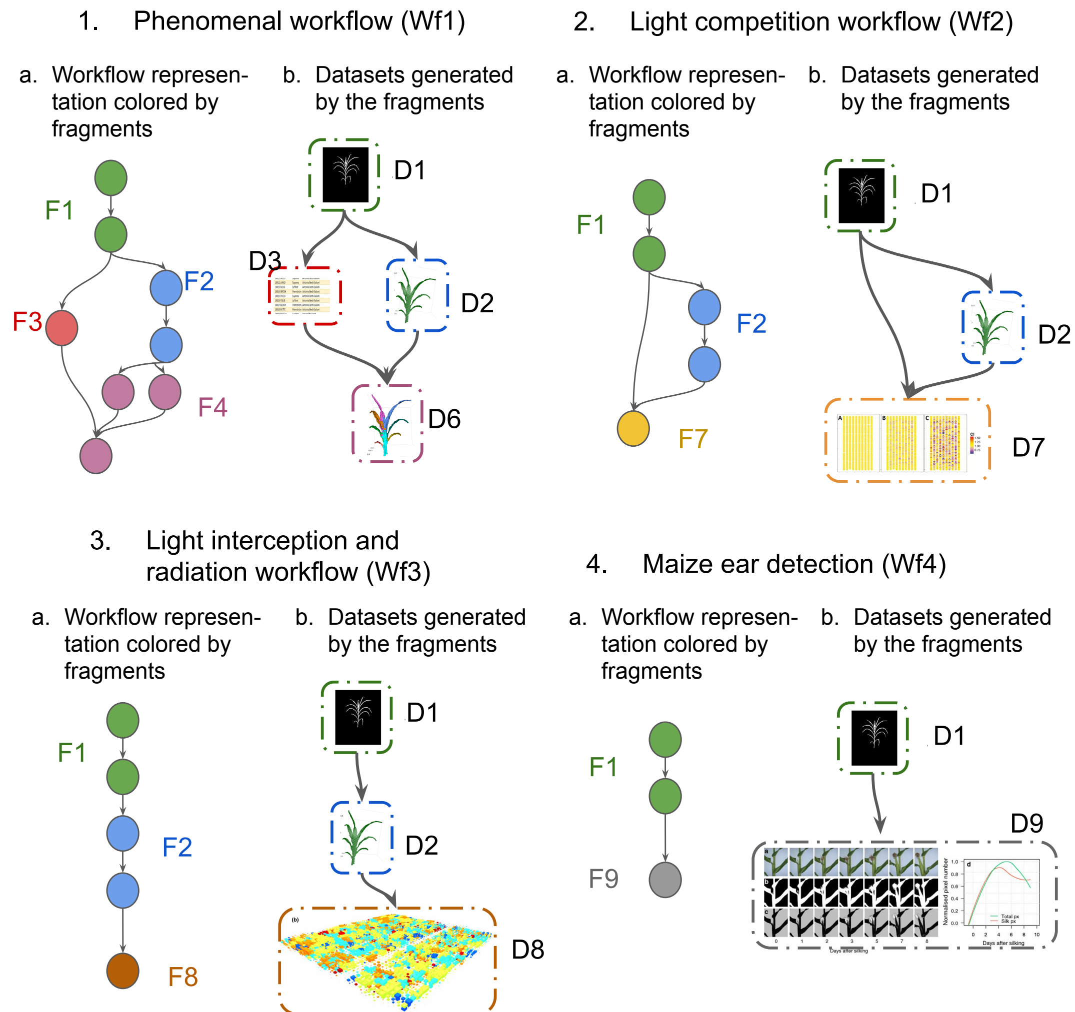


FIGURE 3.8: Four different workflows that use components of the Phenomenal library. 1) the Phenomenal workflow [12]; 2) a light competition workflow [26]; 3) a light interception and radiation workflow [31]; and 4) a maize ear detection workflow [20]

with one input dataset from the PhenoArch platform. For the experiments, we use an environment configuration similar to the one used by the PhenoArch scientists, *i.e.* a cloud with one site with huge data storage, where the raw data is stored, and one (in Chapter 4) or several (in Chapter 5) other sites for workflow execution.

There are many opportunities for data reuse in the HTP domain for several reasons:

1. The number of raw datasets is limited and unique (it depends on plant growth).
2. To compare their analyses, scientists work on the same species and genotypes for baseline comparison.
3. Once some efficient algorithms are implemented, many scientists use them for their analyses.

Yet, they bring several challenges:

1. There is no standard to index and classify intermediate data, which makes the intermediate data difficult to retrieve within the huge amount of data produced and shared.
2. There is no infrastructure to automatically share the intermediate data produced, which requires the user to manually find and select the data she wants to reuse.
3. There are no scheduling techniques adapted to deal with cached data in a multisite environment.

Chapter 4

Adaptive Caching for Scientific workflows in Monosite Cloud

A Cloud provides diverse storing and computing resources and is well adapted for workflow execution. However, workflow executions are both time and resource consuming. These costs adds up when multiple users execute the workflow on the cloud. In this chapter we propose an cache architecture and an adaptive scheduling algorithm to enable the workflow to reuse intermediate data during its execution. This solution automatically share the most suited intermediate data between users' executions. The algorithm is based on a cost model to find a good trade-off between storage cost and computation cost. This chapter is based on [74]

In this chapter, we propose an adaptive algorithm to select the intermediate data generated that should be stored. Section 4.1 presents an overview and the motivations. Section 4.2 presents related works. Section 4.3 describes our proposed system architecture for executing workflows with cache on monosite cloud. Section 4.4 describes our cost model used to minimize monetary costs. Section 4.5 describes the adaptive cache management. Section 4.6 presents our experimental evaluation based on the execution of the Phenomenal workflow on the IFB-cloud. The results reveal that our algorithm significantly reduce monetary costs with regards to two baselines. Section 4.7 concludes.

4.1 Introduction

In many scientific domains, *e.g.*, health [75] or, bio-science [81], complex experiments typically require many processing or analysis steps over huge quantities of data. They can be represented as scientific workflows, which facilitate the modeling, management and execution of computational activities linked by data dependencies. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive [81], thus requiring execution in a high-performance distributed and parallel environment, *e.g.*, a large-scale virtual cluster in the cloud [80].

Most Scientific Workflow Management Systems (SWMSs) can now execute workflows in the cloud [113]. Some examples are Swift/T, Pegasus, SciCumulus, Kepler and OpenAlea [93]. Our work is based on OpenAlea [130], which is being widely used in plant science for simulation and analysis [129].

It is common for workflow users to reuse other workflows or data generated by other workflows. Reusing and re-purposing workflows allows for the user to develop new analyses faster [58]. Furthermore, a user may need to execute a workflow many times with different sets of parameters and input data to analyze the impact of some experimental step, represented as a workflow fragment, *i.e.*, a subset of the workflow activities and dependencies. In both cases, some fragments of the workflow may be executed many times, which can be highly resource consuming and unnecessarily long. Workflow re-execution can be avoided by storing the intermediate results of these workflow fragments and reusing them in later executions.

In OpenAlea, this is provided by a cache in memory, *i.e.* the intermediate data is simply kept in memory after the execution of a workflow. This allows for the user to visualize and analyze all the activities of a workflow without any re-computation, even with some parameter changes. Although cache in memory represents a step forward, it has some limitations, *e.g.*, it does not scale in distributed environments and requires much memory if the workflow is data-intensive.

From a single user perspective, the reuse of the previous results can be done by storing the relevant outputs of intermediate activities (intermediate data) within the workflow. This requires the user to manually manage the caching of the results that she wants to reuse. This can be difficult as the user needs to be aware of the data size, execution time of each task, *i.e.*, the instantiation of an activity during the execution of a workflow, or other factors that could allow deciding which data is best to be cached.

A complementary, promising approach is to reuse intermediate data produced by multiple executions of the same or different workflows. Some SWMSs support the reuse of intermediate data, yet with some limitations. VisTrails [28] automatically makes the intermediate data persistent with the workflow definition. Using a plugin [173], VisTrails allows workflow execution in HPC environments, but does not benefit from reusing intermediate data. Kepler [7] manages a persistent cache of intermediate data in the cloud, but does not take data transfers from remote servers into account. There is also a trade-off between the cost of re-executing tasks versus storing intermediate data that is not trivial [2, 48]. Yuan et al. [169] propose an algorithm to determine what data generated by the workflow should be cached, based on the ratio between re-computation cost and storage cost at the task level. The algorithm is improved in [171] to take into account workflow fragments. Both algorithms are used before the execution of the workflow, using the provenance data of the intermediate datasets, *i.e.*, the metadata that traces their origin. However, these two algorithms are static and cannot deal with variations in tasks' execution times. In both cases, such variations can be very important depending on the input data, *e.g.*, data compression tasks can be short or long depending on the data itself, regardless of size. For instance, an image of a given resolution can contain more or less information.

In this chapter, we propose an adaptive caching solution for efficient execution of data-intensive workflows in the cloud. By adapting to the variations in tasks' execution times, our solution can maximize the reuse of intermediate

data produced by workflows from multiple users. Our solution is based on a new SWMS architecture that automatically manages the storage and reuse of intermediate data. Cache management is involved during two main steps: workflow preprocessing, to remove all fragments of the workflow that do not need to be executed; and cache provisioning, to decide at runtime which intermediate data should be cached. We propose an adaptive cache provisioning algorithm that deals with the variations in task execution times and output data. We evaluated our solution by implementing it in OpenAlea and performing extensive experiments on real data with a complex data-intensive application in plant phenotyping.

4.2 Related works

Storing and reusing intermediate data in workflow executions can be found in several SWMSs [28, 130]. However, there is no definitive solution for two important problems: 1) how to automatically reuse workflow fragments in multiple workflow's executions. 2) what intermediate data to cache if there is not provenance data available. The related works either focus on an optimized solution for selecting a specific portion of data to cache when all provenance and reuse information are known, or automatic caching for the same workflow.

Different SWMSs, such as Kepler, VisTrails, OpenAlea, exploit intermediate data for workflow re-execution. Each of these systems has its unique way of addressing data reuse. OpenAlea [130] uses a cache that captures the intermediate results in main memory. When a workflow is executed, it first accesses the cached data. However, the OpenAlea cache is local and main memory-based, while the approach proposed in this work is distributed and persistent. VisTrails provides visual analysis of workflow results and captures the evolution of workflow provenance, *i.e.*, the steps of the workflow at each execution, as well as the intermediate data from each execution [28]. The intermediate data are then reused when previous tasks are re-executed. This approach allows the user to change parameters or activities in the workflow and efficiently re-execute each workflow activity to analyze the different results. This solution for caching intermediate data has been extended to generate "strong links" between provenance and execution [85] [50]. The intermediate data cache is associated with provenance to enhance reproducibility, and the intermediate data that has been cached is always reused. However, VisTrails does not take distribution into account when storing and using the cache, and the selection of the data to be stored becomes manual as the size of the intermediate data increases. Our approach is different as it works in a distributed environment where data transfer costs may be significant.

Storing intermediate data in the cloud may be also beneficial. However, the trade-off between the cost of re-executing tasks and the costs of storing intermediate data is not easy to estimate [2, 48]. Yuan et al. [169] propose an algorithm based on the ratio between re-computation cost and storage cost at the task level. The algorithm is based on a graph of dependencies between the intermediate data sets, generated from the provenance data. Then, the cost of

storing each intermediate data set is weighted by the number of dependencies in the graph. The algorithm computes the optimized set of intermediate data sets that need to have minimum cost. The algorithm is improved in [171] to take into account workflow fragments. Both algorithms are used before the workflow execution, using the provenance data of the intermediate datasets. They provide near optimal caching intermediate datasets selection. However, this approach requires global knowledge of executions, such as the execution time of each task, the size of each data set and the number of incoming re-executions. This optimization is also based on a single workflow and is not adapted to changing workflows. Our approach is different as it provides efficient caching of intermediate data in evolving workflows.

Kepler [7] provides intermediate data caching for single-site cloud workflow execution. It uses a remote relational database where intermediate data is stored after workflow execution. Two steps are added when executing a workflow. First, the cache database is checked and all intermediate cached data is sent to a specific cloud site before execution. To reduce storage cost, the intermediate data that need to be cached are determined based on how many times the workflow will be re-executed in a given period of time [32]. Finally, reuse is done at the entire workflow level, whereas our solution is finer grain, working at the activity level.

Other approaches propose solutions for caching data in MapReduce workflows. Zhang et al. [177] use the *Memcached* distributed memory caching system to cache the intermediate data between Map and Reduce operations. This approach focuses on a single MapReduce job, and the cached data is not persistent and reused across executions. Elghandour et al. [52] propose a system to manage and cache intermediate data of MapReduce jobs for future reuse. Olston et al. [117] propose two caching strategies on top of the Pig language and propose different methods to manage persistent intermediate data. The problem of this approach is that it is static, *i.e.*, they do not consider automatic caching. Gottin et al. [64] propose an algorithm that finds an optimized cache decision plan for a dataflow execution in Apache Spark. The approach is based on a cost model that uses provenance data, and tries the possible combinations of caching selection in order to select the best one. This approach does not scale with the size of the workflow, and the caching decision still falls in the hands of the user.

4.3 Monosite Cloud SWMS Architecture

In this section, we present the proposed SWMS architecture that integrates caching and reuse of intermediate data in the cloud. We motivate our design decisions and describe our architecture in two ways: i) in terms of functional layers (see Figure 4.1), which shows the different functions and components; and ii) in terms of nodes and components (see Figure 4.2), which are involved in the processing of workflows.

Our architecture capitalizes on the latest advances in distributed and parallel computing to offer performance and scalability [120]. We consider a distributed architecture with on premise servers, where raw data is produced

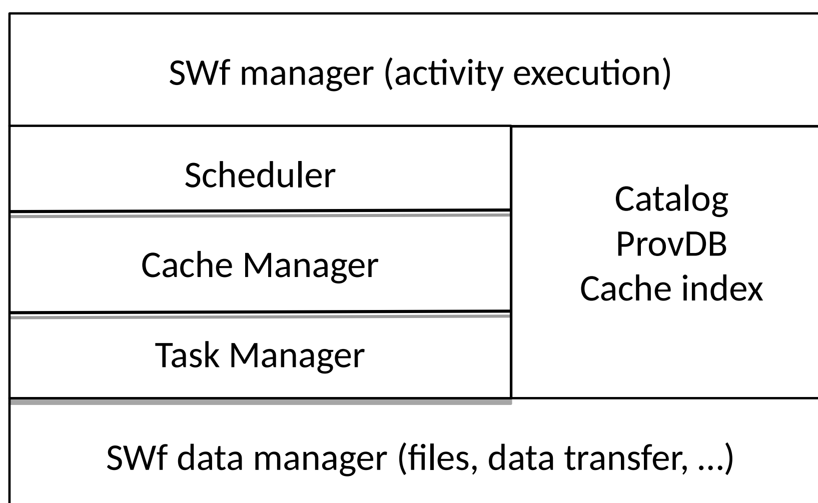


FIGURE 4.1: SWMS Functional Architecture

(*e.g.*, by a phenotyping experimental platform in our use case), and a cloud site, where the workflow is executed. The cloud site (data center) is a shared-nothing cluster, *i.e.*, a cluster of server machines, each with processor, memory and disk. We adopt shared-nothing as it is the most scalable and cost-effective architecture for big data analysis.

In the cloud, metadata management has a critical impact on the efficiency of workflow scheduling as it provides a global view of data location, *e.g.*, at which nodes some raw data is stored, and enables task tracking during execution [94]. We organize the metadata in three repositories: catalog, provenance database and cache index. The catalog contains all information about users (access rights, etc.), raw data location and workflows (code libraries, application code). The provenance database captures all information about workflow execution. The cache index contains information about tasks and cache data produced, as well as the location of files that store the cache data. Thus, the cache index itself is small (only file references) and the cached data can be managed using the underlying file system. A good solution for implementing these metadata repositories is a key-value store, such as Cassandra (`https://cassandra.apache.org`), which provides efficient key-based access, scalability and fault-tolerance through replication in a shared-nothing cluster [1].

The raw data (files) are initially produced at some servers, *e.g.*, in our use case, at the phenotyping platform and get transferred to the cloud site. The server associated with the phenotyping platform is using iRODS [134] to grant access to the data generated. The intermediate data is placed on the node that execute the task, and is produced and processed through memory. It is only written on disk if it is added to the cache. The cache data (files) are produced at the cloud site after workflow execution. A good solution to store these files in a cluster is a distributed file system like Lustre (`http://lustre.org`) which is used a lot in HPC as it scales to high numbers of files.

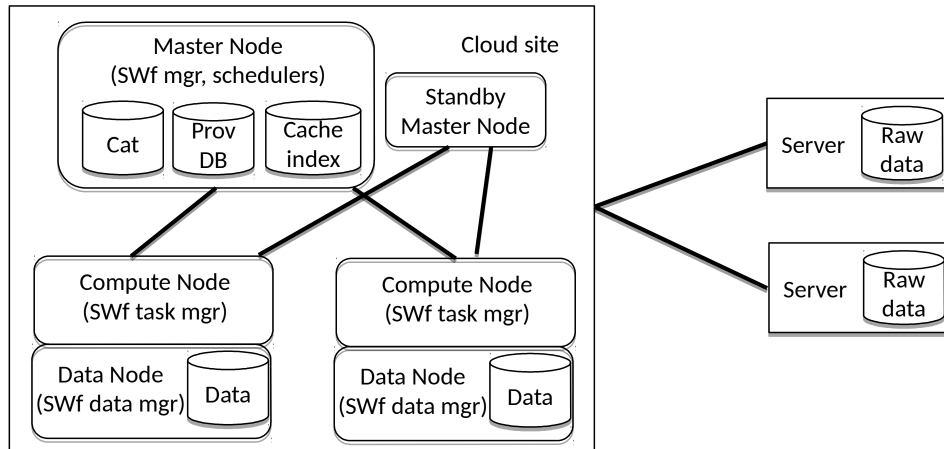


FIGURE 4.2: SWMS Technical Architecture

Figure 4.1 extends the SWMS architecture proposed in [93], which distinguishes various layers, to support intermediate data caching. The workflow manager is the component that the user clients interact with to develop, share and execute workflows, using the metadata (catalog, provenance database and cache index). It determines the workflow activities that need to be executed, and generates the associated tasks for the scheduler. It also uses the cache index for workflow preprocessing to identify the intermediate data to reuse and the tasks that need not be re-executed.

The scheduler exploits the catalog and provenance database to decide which tasks should be scheduled to cloud sites. The task manager controls task execution and uses the cache manager to decide whether the task's output data should be placed in the cache. The cache manager implements the adaptive cache provisioning algorithm described in Section 4.5. The workflow data manager deals with data storage, using a distributed file system.

Figure 4.2 shows how these components are involved in workflow processing, using the traditional master-worker model. There are three kinds of nodes, master, compute and data nodes, which are all mapped to cluster nodes at configuration time, *e.g.*, using a cluster manager like Yarn (`http://hadoop.apache.org`). The master node includes the workflow manager, scheduler and cache manager, and deals with the metadata. The worker nodes are either compute or data nodes. The master node is lightly loaded as most of the work of serving clients is done by the compute and data nodes (or worker nodes), that perform task management and execution, and data management, respectively. Therefore, the master node is not a bottleneck. However, to avoid any single point of failure, there is a standby master node that can perform failover upon the master node's failure and provide high availability.

Let us now illustrate briefly how workflow processing works. User clients connect to the cloud site's master node. Workflow execution is controlled by the master node, which identifies, using the workflow manager, which activities in the fragment can take advantage of cached data, thus avoiding task reexecution. The scheduler schedules the corresponding tasks that need

to be processed on compute nodes which in turn rely on data nodes for data access. It also adds the transfers of raw data from remote servers that are needed for executing the workflow. For each task, the task manager decides whether the task's output data should be placed in the cache taking into account storage costs, data size, network costs. When a task terminates, the compute node sends to its master the task's execution information to be added in the provenance database. Then, the master node updates the provenance database and may trigger subsequent tasks.

4.4 Cost Model

In this section, we present our cost model. We start by introducing some terms and concepts. A workflow $W(A, D)$ is the abstract representation of a directed acyclic graph (DAG) of computational activities A and their data dependencies D. There is a dependency between two activities if one consumes the data produced by the other. An activity is a description of a piece of work and can be a computational script (computational activity), some data (data activity) or some set-oriented algebraic operator like map or filter [110]. The parents of an activity are all activities directly connected to its inputs. A task t is the instantiation of an activity during execution with specific associated input data. The input $In(t)$ of t is the data needed for the task to be computed, and the output $Out(t)$ is the data produced by the execution of t. Whenever necessary, for clarity, we alternatively use the term intermediate data instead of output data. Execution data corresponds to the input and output data related to a task t. For the same activity, if two tasks t_i and t_j have equal inputs, then they produce the same output data, *i.e.*, $In(t_i) = In(t_j) \Rightarrow Out(t_i) = Out(t_j)$. A workflow's input data is the raw data generated by an experimental platform, *e.g.*, a phenotyping platform.

Our approach focuses on the trade-off between execution time and cache size. In order to compare the execution time and cache size, we use a monetary cost approach, which we will also use in the experimental evaluation in Section 4.6. All the costs are compared at the task level and are expressed in USD. For a task t, the total cost of n executions according to the caching decision can be defined by:

$$Cost(t, n) = \omega_t * TimeCost(t, n) + \omega_c * CacheCost(t, n) \quad (4.1)$$

where $TimeCost(t, n)$ is the cost associated with the execution time and $CacheCost(t,n)$ is the cost associated with caching. They represent the amount of USD spent in order to obtain the output of a task, n times. ω_t and ω_c represent the weights of the two cost components, which are positive.

The execution time cost of a task depends on whether or not the output data of the task is added to the cache. If the output of t is not added to the cache, the execution time cost $Cost_{nocache}(t, n)$ is the sum of the costs associated with getting *In(t)* and executing t, n times. Otherwise, *i.e.*, *Out(t)* is added to the cache, the execution time cost $Cost_{cache}(t, n)$ is composed of the cost of the first execution of t, the cost to provision the cache with $Out(t)$ and the cost of

retrieving $Out(t)$, *n-1* times. $TimeCost(t, n)$ can be defined by:

$$TimeCost(t,n) = \begin{cases} Cost_{nocache}(t,n), & \text{if Out(t) not in cache.} \\ Cost_{cache}(t,n), & \text{otherwise.} \end{cases} \quad (4.2)$$

During workflow execution, the execution time of each task t, denoted by $Time_{exec}(t)$, is stored in the provenance database. If t has already been executed, $Time_{exec}(t)$ is already known and can be retrieved from the provenance database. When t is re-executed, its execution time is recomputed and $Time_{exec}(t)$ is updated as the average of all execution times. The access times to read and write in the cache are $Time_{read}$ and $Time_{write}$. Here it will be applied to input $In(t)$ and output $Out(t)$ data. This time mostly dependent on the data size. $Cost_{nocache}(t, n)$ and $Cost_{cache}(t, n)$ are then given by:

$$Cost_{nocache}(t,n) = Cost_{cpu} * n * [Time_{read}(In(t)) + Time_{exec}(t)] \quad (4.3)$$

$$\begin{aligned} Cost_{cache}(t,n) = Cost_{nocache}(t,1) \\ +Cost_{cpu} * (n-1) * Time_{read}(Out(t)) \end{aligned} \quad (4.4)$$

where $Cost_{cpu}$ represents the average monetary cost to use virtual CPUs in one determined time interval.

The cost associated with the size of the cache can be defined by:

$$CacheCost(t,n) = Cost_{disk} * size(Out(t)) \quad (4.5)$$

where $Cost_{disk}$ represents the monetary cost of storing data in one specific time interval, determined by the user, and $size(Out(t))$ is the real size of the output data generated by t execution.

The caching decision depends on the trade-off between the execution time cost and the storage cost. For some tasks, the output data is either much bigger in size or much complex than their input data, in this case, it is more time consuming to retrieve data from the cache than re-executing the task (see Equation 4.6). This is the case for most of the tasks on plant graph generation in our workflow's use case. In this case, no matter what is the storage cost, it is less costly to simply re-execute t. The output data generated is then not added to the cache.

$$Time_{read}(In(t)) + Time_{exec}(t) \leq Time_{read}(Out(t)) \quad (4.6)$$

In other cases, *i.e.*, when it is time saving to retrieve the output data of a task t instead of re-executing t, the execution time cost and caching cost are compared. The output data of the task t is worth putting in the cache if for n executions of t, the cost of adding the data into the cache is smaller than the cost of an execution without cache, *i.e.*:

$$Cost_{cache}(t,n) + CacheCost(t,n) \leq Cost_{nocache}(t,n) \quad (4.7)$$

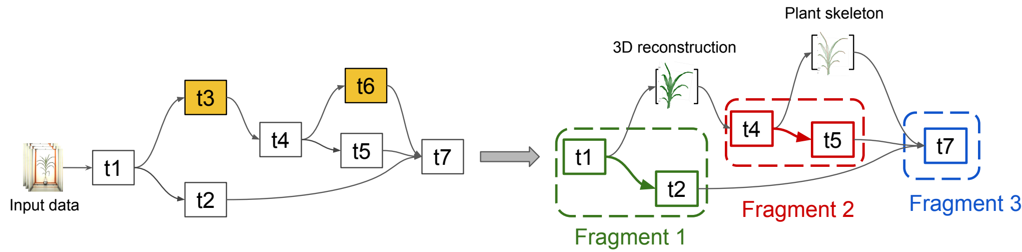


FIGURE 4.3: DAG of tasks before preprocessing (left) and the selected fragments that need to be executed (right).

From Equations ((4.3)), (4.4), (4.5) and (4.7), we can now get the minimal number of times denoted by $n_{min}(t)$, which the task t needs to be executed that it is cost effective to add its output into the cache. $n_{min}(t)$ is given by:

$$n_{min}(t) = 1 + \frac{Time_{write}(Out(t)) + \dfrac{Cost_{disk} * size(Out(t))}{Cost_{cpu}}}{Time_{read}(In(t)) + Time_{exec}(t) - Time_{read}(Out(t))} \tag{4.8}$$

We introduce $p(t)$, the probability that t be re-executed. There is then a limit value $p_{min}(t)$ that represents the minimum value of $p(t)$ from which the output of t is worth to add in the cache. Based on Equation (4.8), $p_{min}(t)$ can be defined as:

$$p_{min}(t) = n_{min}(t) - 1 \tag{4.9}$$

The value $p_{min}(t)$ is a ratio between the cost of adding the output data of the task t into a cache and the possible cost saved if this cached data is used instead of re-executing the task and its parents.

In the case of multiple users, the exact probability $p(t)$ or the number of times the task t will be re-executed is not known when the workflow is executed. We then introduce a threshold p_{tresh} arbitrarily picked by the user. This threshold will be the limit value to decide whether a task output will be added to the cache.

During the execution of each task, the real values of the execution time and data size related to t are known. Thus, the caching decision is made from the Equations (4.6) and (4.9).

4.5 Cache Management

This section presents in detail our techniques for cache management. In our solution, cache management is involved during two main steps: *workflow preprocessing* and *cache provisioning*. The preprocessing step transforms the workflow based on the cache by replacing workflow fragments by already computed output data stored in the cache. The preprocessing step occurs just before execution and is done by the workflow manager using the cache index. The workflow manager transform the workflow $W(A, D)$ into an executable workflow $W_{ex}(A, D, T, Input)$, where T is a DAG of tasks corresponding to the

activities in A and $Input$ is the input data. The goal of workflow preprocessing is to transform an executable workflow $W_{ex}(A, D, T, Input)$ into an equivalent, simpler subworkflow $W'_{ex}(A', D', T', Input')$, where A' is a subgraph of A with dependencies D', T' is a subgraph of T corresponding to A' and $Input'$ is a subset of $Input$. The preprocessing step uses a recursive algorithm that traverses the DAG T starting from the sink tasks to the source ones. The algorithm marks each task whose output is already in the cache. Then, the subgraphs of T that have each of their sink tasks marked are removed, and replaced by the associated data from the cache. The remaining graph is T'. Finally, the algorithm determines the fragments of T': subgraphs that still need to be executed.

Figure 4.3 illustrates the preprocessing step on the Phenomenal workflow. The yellow tasks have their output data stored in the cache. They are replaced by the corresponding data as input for the subgraphs of tasks that need to be executed.

The second step, cache provisioning, is performed during workflow execution. Traditional (in memory) caching involves deciding, as data is read from disk into memory, which data to replace to make room, using a cache replacement algorithm, *e.g.*, Least Recently Used (LRU). In our context, using a disk-based cache, the question is different. Unlike memory cache, disk-based cache makes it possible to cache the Terabytes of data generated by the workflow's execution. Caching huge datasets has a cost and the question is to decide which task output data to place in the cache using a cache provisioning algorithm, in order to limit execution costs. This algorithm is implemented by the cache manager and used by the task manager when executing a task.

A simple cache provisioning algorithm, which we will use as baseline in the experimental evaluation, is to use a *greedy* method that simply stores all tasks' output data in the cache. However, since workflow executions produce huge quantities of output data, this approach would incur high storage costs. Worse, for some short duration tasks, accessing cache data from disk may take much more time than re-executing the corresponding task subgraph from the input data in memory.

Thus, we propose a cache provisioning algorithm with an adaptive method that deals with the variations in task execution times and output data complexity and sizes. The principle is to compute, for each task t, the value $p_{min}(t)$ defined in Section 4.4, called cache score of t, which is based on the sizes of the input and output data it consumes and produces, and the execution time of t. Depending on this value, after each task execution, the cache manager decides on whether the output data is added to the cache or not.

The cache score reveals the relevancy of caching the output data of t and takes into account the compression ratio and execution time as well as the caching costs. According to the weights provided by the user, she may prefer to give more importance to the compression ratio or executions time, depending on the storage capacity and available computational resources.

Then, during the execution of each task t, the task manager calls the cache manager to compute $p_{min}(t)$. If the computed value is smaller than the threshold p_{tresh} provided by the user, then t's output data will be cached. This

threshold is arbitrarily chosen based on the probability of the workflow being re-executed.

4.6 Experimental Validation

In this section, we first present our experimental setup. Then, we present our experiments and experimental comparisons of different caching methods in terms of speedup and monetary cost in single user and multiuser scenarios. Finally, we give concluding remarks.

4.6.1 Experimental Setup

Our experimental setup includes the cloud infrastructure, a workflow implementation and an experimental dataset.

The cloud infrastructure is composed of one site with one data node (*N1*) and two identical compute nodes (*N2*, *N3*). The raw data is originally stored in an external server. During computation, raw data is transferred to N1, which contains Terabytes of persistent storage capacities. Each compute node has much computing power, with 80 vCPUs (virtual CPUs, equivalent to one core each of a 2.2GHz Intel Xeon E7-8860v3) and 3 Terabytes of RAM, but less persistent storage (20 Gigabytes).

We implemented the Phenomenal workflow (see Section 3.2.2) using OpenAlea and deployed it on the different nodes using the Conda multi-OS package manager. The master node is hosted on one of the compute nodes (*N2*). The metadata repositories are stored on the same node (*N2*) using the Cassandra key-value store. Files for raw and cached data are shared between the different nodes using the Lustre file system. File transfer between nodes is implemented with ssh.

The Phenoarch platform has a capacity of 1,680 plants with 13 images per plant per day. The size of an image is 10 Megabytes and the duration of an experiment is around 50 days. The total size of the raw image dataset represents 11 Terabytes for one experiment. The dataset is structured as 1,680 time series, composed of 50 time points (one per plant and per day).

We use a version of the Phenomenal workflow composed of 9 main activities. We execute it on a subset of the use case dataset, which is $\frac{1}{25}$ of the size of the full dataset, or 440 Gigabytes of raw data, which represents the execution of 30,240 tasks.

The time interval considered for the caching time (see Section 4.4) is 30 days, *i.e.*, the workflow re-executions are done within one month. The user can select longer or shorter time intervals depending on the application.

For the comparison of different cost-based caching methods, we use cost models defined in Section 4.4. To set the price parameters, we use prices from Amazon AWS, *i.e.*, $Cost_{disk}$ is \$0.1 per Gigabyte per month for storage and two instances at \$5.424 per hour for computation, *i.e.*, $Cost_{cpu}$ is \$10.848 per hour. We set the user's parameters ω_t and ω_c at 0.5.

The caching methods we compare, defined in Section 4.5 are noted as:

- *M1* for the execution without cache.
- *M2* for the greedy method where all the created intermediate data produced is cached.
- $M3_X$ for the adaptive method, with X as the p_{tresh} value. In our experiments, X vary between 10, 40 and 160.

4.6.2 Experiments

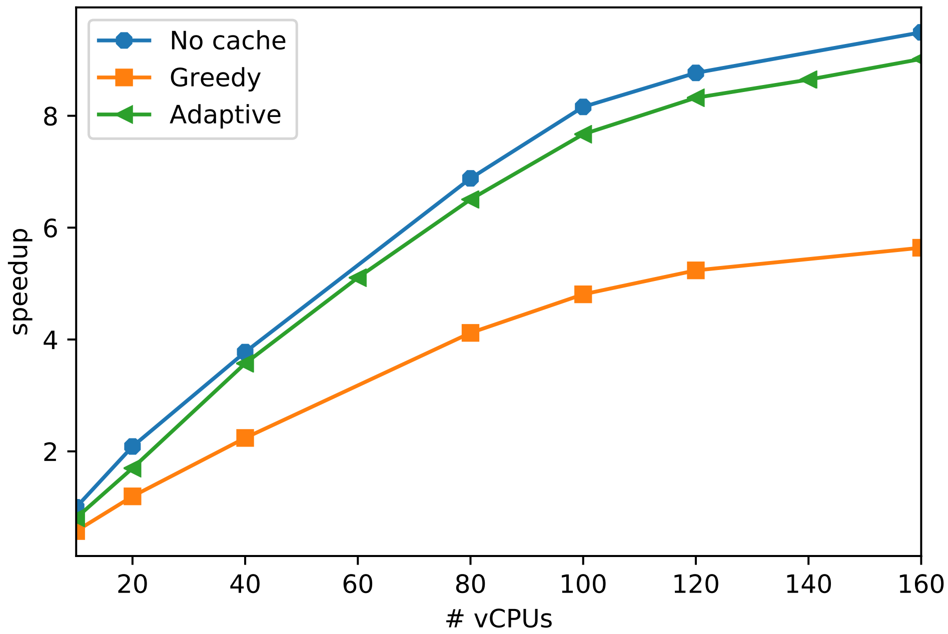
We consider three experiments, based on the use case in order to analyze our caching method under different conditions:

1. This experiment aims at evaluating the scalability and speedup of the caching methods. In this experiment, we assume that the same workflow is computed three times in a month, at different times (one user at a time). This experiment is based on the workflow Phenomenal, *i.e.*, the maize analysis (see Section 3.2.2). The scalability of the workflow execution is studied using different numbers of vCPUs from 10 to 160.
2. This experiment aims at analysing the impact from the variability in execution time and data size of the tasks from each activity, on the components of the proposed cost function.
3. In this experiment, the same workflow is executed with an adaptive cache strategy with different monetary costs. We assume that the same workflow is executed up to six times in a month, starting from an empty cache. This experiment shows the trade-off between better re-execution time and smaller cache size.
4. In this experiment, different users execute different workflows that reuse sub-parts of the complete Phenomenal workflow. Depending on the caching strategy and cache size, the result of some tasks may already be present in the cache. We show the impact of the value p_{tresh} (see Section 4.4) on execution time, cache size and overall monetary cost depending on the user executions.

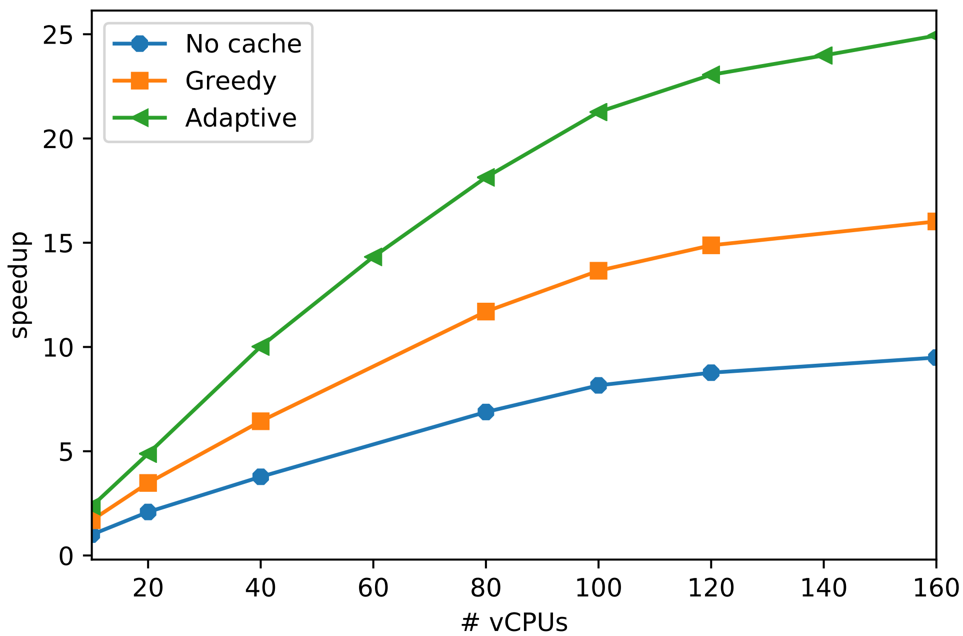
Except for Experiment 1 where the number of vCPUs varies, it is set at 160 for the three other experiments. The execution time corresponds to the time to transfer the raw data files from the remote servers, the time to run the workflow and the time to provision the cache.

Workflow executions of the different users are serial, thus we do not consider concurrency when accessing cached data. Moreover, we assume that there are no execution or data transfer failures.

The raw data is retrieved on the data node as follows: a first file is retrieved from the remote data servers and stored in one cluster's data node. Then, execution starts using this first file while the next files are retrieved in parallel. As executing the workflow on the first file takes longer than transferring one more raw data file, we only count the time of transferring the first chunk in the execution time.



(A) *For one execution*



(B) *For three executions*

FIGURE 4.4: Speedup versus number of vCPUs: without cache (orange), greedy caching (blue), and adaptive caching (green).

Speedup.

In Experiment 1, we compare the speedup of the three caching methods with a threshold $p_{tresh} = 40$, which is optimal in this case. We define the speedup as $speedup(n) = \frac{T_n}{T_{10}}$, where T_n is the execution time on n vCPUs and T_{10} is the execution time of method *M1* on 10 vCPUs.

The workflow execution is distributed on nodes *N2* and *N3*, for different numbers of vCPUs. For one execution, Figure 4.4.a shows that the fastest method is *M1* (orange curve). This is expected as there is no extra time spent to make data persistent and provision the cache. However, the overhead of cache provisioning with method $M3_{40}$ is very small, less than 6% (green curve in Figure 4.4.a) compared with method *M2*, up to 40% (blue curve in Figure 4.4.a) where all the output data are saved in the cache.

For the first execution, method $M3_{40}$'s overhead is only 5,6% compared to method *M1*, while method *M2*'s overhead goes up to 40.1%. For instance, with 80 vCPUs, the execution time of method $M3_{40}$ (*i.e.*, 3,714 seconds) is only 5.8% higher than execution time of method *M1* (*i.e.*, 3,510 seconds). This is much faster than method *M2*, which adds 2,152 seconds (1.58 times longer) of computation time in comparison with method $M3_{40}$. In both cases, any re-execution is much faster than the first execution. Method *M2* re-execution time is the fastest, with a speedup gain of factor that is 102 times (*i.e.*, 34 seconds) better compared with method *M1*, because all the output data is already cached. Furthermore, while only the master node is working when no computation is done, the re-execution time is independent of the number of vCPUs and can be computed from a personal computer with limited vCPUs. Method $M3_{40}$ re-execution time is 12.6 times (*i.e.*, 258 seconds) better compared to method *M1*'s re-execution time. With method $M3_{40}$, some computation still needs to be done when the workflow is re-executed, but such re-execution on the whole dataset can be done in a bit more than a day (*i.e.*, 28.7 hours) on a 10 vCPUs machine, compared with 7.2 days with method *M1*.

For three executions starting without cache, Figure 4.4.b shows that method $M3_{40}$ is much faster than the other methods (about 2.5 and 1.5 times faster of 3 executions compared to methods *M1* and *M2* on 80 vCPUs). Method *M2* is faster than method *M1* in this case, because the additional time for cache provisioning is compensated by the very short re-execution times of method *M2*. With 80 vCPUs, the speedup of method $M3_{40}$ (*i.e.*, 18.1) is 54.70% better than that the speedup of method *M2* (*i.e.*, 11.7) and 162.31% better than that of method *M1* (*i.e.*, 6.9). Method $M3_{40}$ is faster than the other methods on three executions, despite having a re-execution time higher than method *M2*, because the overhead of the cache provisioning is 57% smaller.

Analysis of tasks variability.

The Phenomenal workflow is composed of nine activities (see Section 3.2.2), which we denote by A1, A2, ..., A9. During its execution, thousands of tasks are executed that belong to the same activities. In order to assess the behavior of a task with respect to its activity, we analyze the execution time of each task

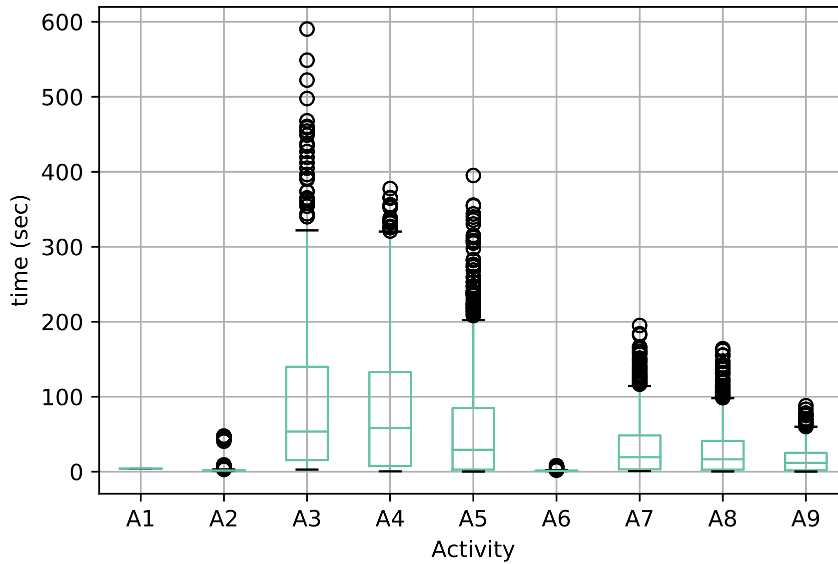


FIGURE 4.5: Execution time of each activity's task.

per activity (see Figure 4.5) and the cost model through their p_{min} value (see Figure 4.6).

In Figure 4.5, execution times of tasks that belong to activities A1, A2 and A6 have few variations. The tasks of such activities have predictable execution times and this information can be used to make decisions about static caching. However, the execution times of A3, A4 and A5 have high variability, which makes them unpredictable.

Figure 4.6 shows that the variability of the p_{min} value is reduced, compared to the variability of the execution times for activities A3 and A7. For activity A9, this is the opposite: the p_{min} values for the tasks of A9 have high variability. Note that the values of p_{min} shown on the figure are limited at 500 for visibility and A9 values are not entirely visible. For activities A2 and A4, the p_{min} value is not computed as it is always more time consuming to get their output data from the cache than recomputing them. This case is explained in Section 4.4 with Equation 4.6.

If the variance in the behavior of an activity's tasks is small, then the behavior of the whole workflow execution is predictable, *i.e.*, the tasks' execution times and intermediate data sizes are predictable. In this case, the caching decision can be static, and done prior to execution. However, in our case, there are significant variations in the task behaviors, so we adopt an adaptive approach.

Monetary cost evaluation.

The first experiment shows that method $M3_{40}$ scales and reduces re-execution time. However, method *M2* enables faster re-executions despite a longer first execution time, but it also generates more cached data. In this experiment,

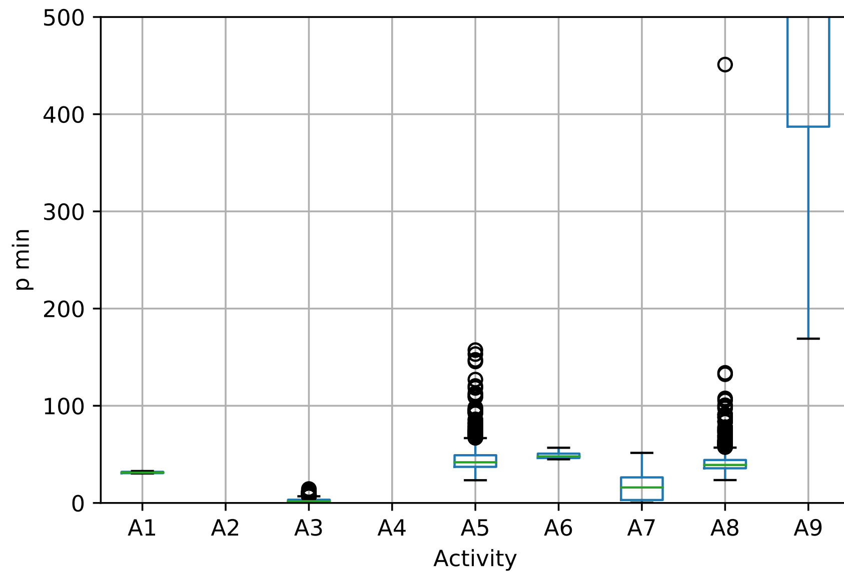


FIGURE 4.6: p_{min} of each activity's task.

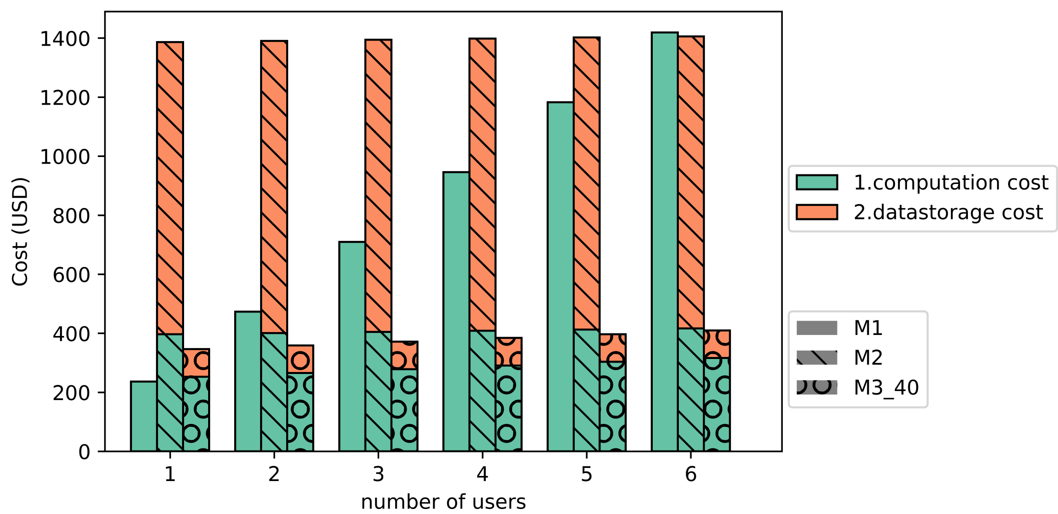


FIGURE 4.7: Monetary cost depending on the number of users that execute the workflow with three different cache strategies with the execution cost (blue) and the storage cost (red).

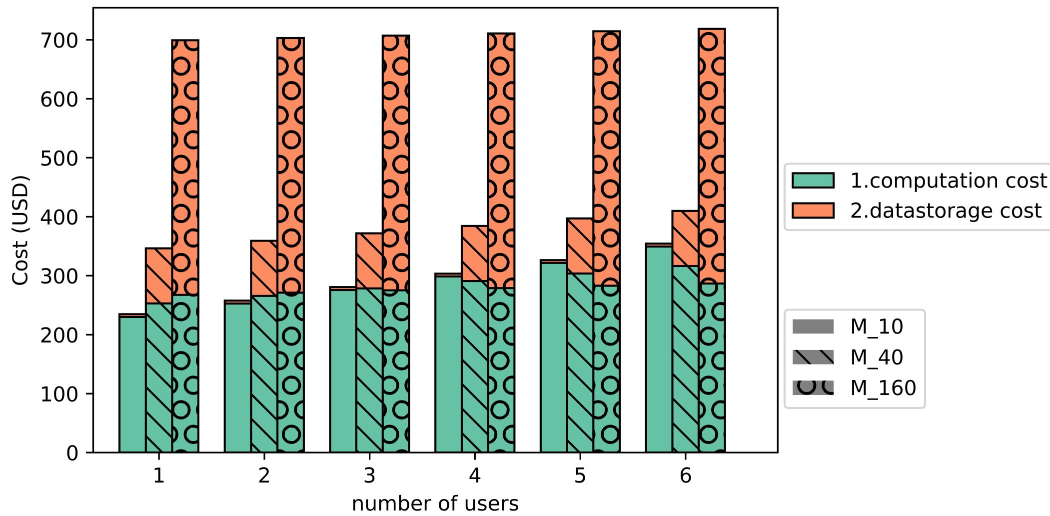


FIGURE 4.8: Monetary cost of scenario S1: Each user executes the last activity of the workflow, for three values of p_{tresh}.

we evaluate the monetary costs of the various methods for the executions of workflow (see Figure 4.7).

The cost of method *M1* comes only from the computation, as no data is stored. The whole workflow is completely re-executed so the cost increases linearly with the number of executions and ends at a total of USD 1419 for six executions. Method *M2* has computation and data storage costs higher than the other two methods for the first execution. The amount of intermediate data added to the cache is huge and the total cost for the first execution is 5.96 times higher than method *M1* (*i.e.*, 1405$). However, the very small computation cost from re-execution (7.73$) compensates for the data storage cost in comparison with the method *M1* after the sixth executions.

For the first execution, method $M3_{40}$ adds 6% overhead in regards to method *M1*'s execution cost because it populates the cache with a total of 934 Gigabytes. For any future re-executions, the decrease in computation time for method $M3_{40}$ makes it less expensive than method *M1*. For six executions, the cost gain is a factor of 3.5 (the total cost of method $M3_{40}$ is 409$). Method $M3_{40}$ also has a cost gain of a factor of 3.5 compared to *M2* for six executions. The amount of intermediate data added to the cache is almost 10 times smaller for method $M3_{40}$ than for method *M2*. Thus, the data storage cost of method *M2* is not worth the decrease in the computation cost compared with method $M3_{40}$.

This shows that method $M3_{40}$ efficiently selects the intermediate data to be added to the cache in order to reduce the cache size significantly while also reducing the re-execution time.

Adding activities.

In this experiment, we evaluate how the parameters of our approach impact the re-execution time, cache size and monetary cost in three different scenarios from the use case, where different workflows are executed independently but

Caching method	Percentage of tasks cached									Cache size (GB)	Re-execution time (hours)		
	A1	A2	A3	A4	A5	A6	A7	A8	A9		S1	S2	S3
No cache	0	0	0	0	0	0	0	0	0	0	21.8	103.4	69.4
Greedy	100	100	100	100	100	100	100	100	100	9894.9	0.04	0.43	0.13
p_{tresh} 10	0	0	98	0	0	0	41	0	0	49.1	1.6	22.4	9.6
p_{tresh} 40	100	0	100	0	39	0	96	55	0	934.3	0.71	5.5	3.6
p_{tresh} 160	100	0	100	0	100	100	100	99	0	4318.4	0.31	2.4	1.2

TABLE 4.1: Caching decision per task and total cache size and re-execution time for different caching methods.

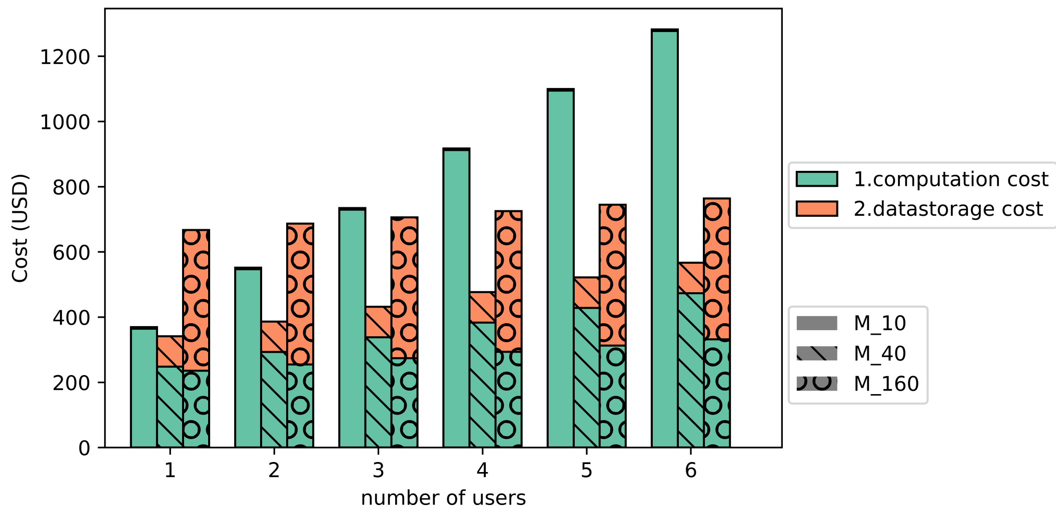


FIGURE 4.9: Monetary cost of scenario S2: Each user executes all the activities of the workflow, for three values of p_{tresh}.

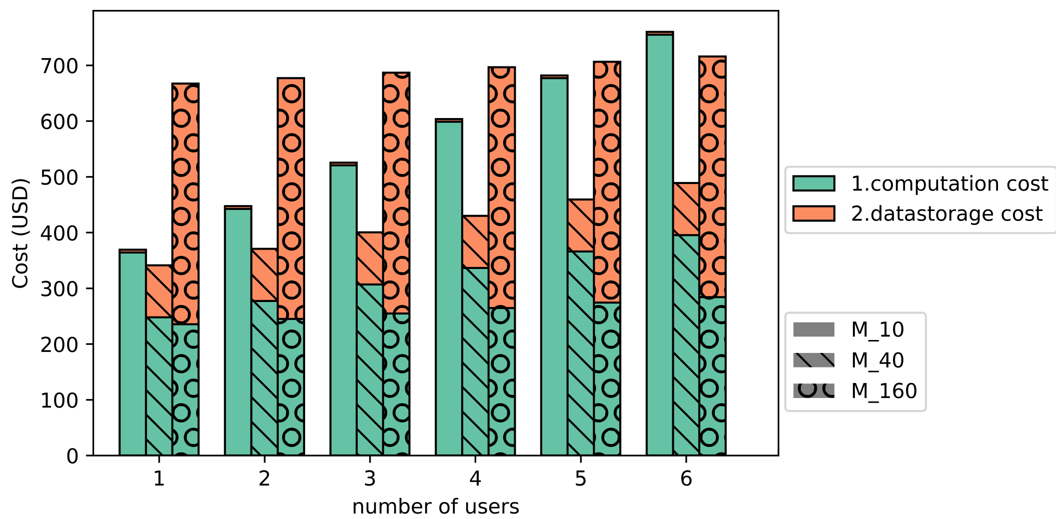


FIGURE 4.10: Monetary cost of scenario S3: Each user executes the activities based on the use case (A1, A3, A7, A9), for three values of p_{tresh}.

share activities. We say that a user executes an activity from a workflow if she executes the sub-part of the workflow that leads to this activity only. The three scenarios are as follows:

1. Scenario S1 is the one presented in the monetary cost evaluation: a single user executes the last activity, *i.e.*, A9: maize analysis, up to six times.

2. Scenario S2 involves nine users, that will each executing a different activity of the workflow up to six times.

3. Scenario S3 involves four users: one executes activity A1, *i.e.*, binarization, the second executes activity A3, *i.e.*, 3D reconstruction, the third executes activity A7, *i.e.*, maize segmentation, and the last one executes activity A9, *i.e.*, maize analysis.

In these scenarios, each user executes a part of the workflow different from the others. Figures 4.8 - 4.10 illustrate the monetary costs for three values of p_{tresh}: 10, 40 and 160. The p_{tresh} values are the threshold set by the user to manage the weight between cache size and re-execution time. With a small p_{tresh}, only a small portion of the output of the tasks is to be added to the cache. A larger p_{tresh} results in more intermediate data added to the cache.

In scenario S1, only one activity, the last one, is re-executed. As this activity is the last, without cache implies the whole workflow to be re-executed. However, as we can see in Figure 4.8, re-executions require little computation time even for the smallest p_{min} = 0.1. The re-execution times are respectively 1.6, 0.71 and 0.31 hours for methods $M3_{10}$, $M3_{40}$ and $M3_{160}$ (see Table 4.1) instead of 21.8 hours without cache. In this scenario, the overall monetary cost of method $M3_{160}$ is the highest, 49.1% higher than method $M3_{40}$, which we used as baseline in the previous section. Yet, the monetary cost of method $M3_{160}$ remains 57.1% smaller than that of method $M2$. This shows that adaptive method successfully selects the intermediate data that is the less costly to store and most worth for re-execution even in the case where a lot of intermediate data is cached.

The computation cost for the re-execution of method $M3_{10}$ is 3.6 and 6.2 times higher than for methods $M3_{40}$ and $M3_{160}$. However, it is the most cost-effective: 324.9$, *i.e.*, 19.7% less than method $M3_{40}$'s cost. In scenario S1 where a single activity is re-executed, a small cache size is the best option.

In scenario S2, each activity is re-executed, which represents the extreme opposite of scenario S1. In this scenario, the re-execution time for method $M3_{10}$ is much higher: 8.7 hours compared to 3.4 and 1.2 hours. The difference in the cache storage cost is not enough to compensate for the re-computation cost and method $M3_{10}$ ends up being 22.8% more costly than the method $M3_{40}$. The re-computation cost of method $M3_{40}$ is also higher than in S1, and the overall cost for six executions is only 32.4% smaller than that of method $M3_{160}$. In this scenario where a lot of different activities are re-executed, our method still successfully selects the right intermediate data for efficient re-execution, yet with a limited cache size.

Scenario S3 is the most representative of our use case, with different users working on specific activities, *i.e.*, A1, A3, A7 and A9. In this scenario, the re-execution time of method $M3_{10}$ is 4.4 times higher than that of method $M3_{40}$, so the computation cost of method $M3_{10}$ increases 4.4 times faster. Method $M3_{40}$ is still the cheapest one, being 8.8% and 46.4% cheaper than methods $M3_{10}$ and $M3_{160}$. However, the computation cost is almost the same as method $M3_{160}$, the cost difference coming mostly from storage. This demonstrates that our method efficiently selects the intermediate data to cache when sub-parts of the workflow are executed separately.

4.6.3 Discussion

The proposed adaptive method has better speedup compared to the no cache and greedy methods, with performance gains up to 162.31% and 54.70% respectively for three executions. The execution time gain for each re-execution goes up to a factor of 60 for the adaptive method in comparison to the no cache method (*i.e.*, 0.31 hours instead of 21.8). One requirement from the use case was to make workflow execution time shorter than half a day (12 hours). The adaptive method allows for the user to re-execute the workflow on the total dataset (*i.e.*, 11 Terabytes) in less than one hour in the cloud and still within a day on a 10 CPUs server. In terms of monetary cost, the adaptive method yields very good gains, up to 257.8% with 6 workflow re-executions in comparison to the no cache method and 229.2% in comparison to the greedy method, which represents up to 1000$.

The experiments on several fragments of the workflow as described in the use case, show that the adaptive method succeeds in picking the most worthy intermediate data to cache. The method does work, even though the structure of the workflow is changed across re-executions. Similar to what happens with re-execution of a single workflow, the monetary cost of the greedy method is higher than the no cache method for up to 6 executions with different fragments or different parameters. And the execution time of greedy is always better than no cache. The adaptive method is both faster and cheaper than both no cache and greedy.

The different values of the parameter p_{tresh} allow the user to adjust between a smaller cache size or smaller re-execution time. Table 4.1 shows the trade-off for three p_{tresh} values. Increasing the amount of intermediate data cached obviously decreases the re-execution time of the workflow in any scenario proposed. But the increase in cache size is not proportional to the decrease of re-execution time. The method first selects the most worthy intermediate data to add in the cache. Then, some intermediate data which is considered not beneficial, will never be added to the cache.

The method proposed in this chapter focuses on finding the most cost effective intermediate data to cache during workflow execution, depending on p_{tresh} and on the user's preferences (ω_t and ω_c), assuming the cache size is unlimited. However, in some applications and organizations, data storage may have some limitation. In this case, it could be interesting to get the optimal value of p_{tresh} for each task in order to minimize the re-execution time.

As the method is adaptive, the size of the total cached data is unknown until the end of the execution. However, the adaptive method could be coupled with other approaches that would approximate the final cache size. Indeed, even if tasks from the same activity have variations in their caching values and their output data sizes, an approximation could be done by taking the average value or the maximum value of some tasks for each activity, then making a static caching decision on all the rest of the tasks.

The Phenomenal workflow is data-intensive because some activities process/ generate huge datasets. Indeed, some activities are compressing the data by a significant factor, *i.e.*, the binarization is compressing the raw data by a factor 500. Other are expending the data, *i.e.*, the skeletonization is expanding the data by a factor 100 (it generates 2TB of data while consuming only 30GB). The Phenomenal workflow is also compute-intensive, as some activities require long computing time, *i.e.*, the 3D reconstruction require 1200 hours of total computing time. The Phenomenal workflow is representative of many other data science workflows, that perform long analyses on huge datasets. Thus, the method presented would work on data-intensive workflows where the execution time is significant with regards to the data transfers times However, the method is not suitable for any kind of application. It adds an overhead when the workflow is executed, thus it would be inefficient on workflows that are not data- or compute-intensive.

4.7 Conclusion

In this chapter, we proposed an adaptive caching solution for efficient execution of data-intensive workflows in the cloud. Our solution automatically manages the storage and reuse of intermediate data, and is adaptive in terms of variations in task execution times and output data size. The adaptive aspect of our solution is to take into account task compression behavior.

We implemented our solution in the OpenAlea SWMS and performed extensive experiments on real data with the Phenomenal workflow, a real big workflow that consumes and produces around 11 TB of raw data. We compared three methods : no cache, greedy, and adaptive. Our experimental evaluation shows that the adaptive method allows for caching only the relevant output data for subsequent re-executions by other users, without incurring a high storage cost for the cache. The results show that adaptive caching can yield major performance gains, *e.g.*, up to a factor of 3.5 with 6 workflow re-executions.

In this chapter, we focused on reducing the monetary cost of running multiple workflows by caching and reusing intermediate data. While our technique show an improvement with respect to greedy approaches, we notice that the scaling up is limited (see Figure 4.4). In the case of multiple users, the cloud computing and storage capacities might be a bottleneck to scale up workflow executions. In the use case, multiple cloud sites are available. A next step would be to extend our method to multisite clouds.

The architecture proposed is based on disk storage for data reuse. Writing and reading the cached data on disk adds an significant overhead. A next step

to improve our cache architecture would be to add an in memory cache for some of the most used cached data.

The contributions of this chapter represent an important step forward in experimental science like biology, where scientists extend existing workflows with new methods or new parameters to test their hypotheses on datasets that have been previously analyzed.

Chapter 5

Cache-Aware Scheduling for Scientific Workflows in Multisite Cloud

The cloud is a convenient infrastructure for handling workflows, as it allows leasing resources at a very large scale and relatively low cost. Clouds are typically composed of multiple geo-distributed data-centers (henceforth named sites). Thus, it becomes important to be able to execute workflow taking into account the geographical distribution of the data and resources among the cloud sites. Therefore, managing workflow executions on multisite cloud that can benefit from caching data is a major problem. This chapter is based on [73]

In this chapter, we propose both a technical and functional workflow architecture, as well as three scheduling algorithms. The algorithms enable workflow execution on multisite cloud with caching and reusing intermediate data. Section 5.1 presents an overview and the motivations. Section 5.2 presents some elements of context from our use case. It presents how users may benefit from sharing intermediate data distributed on a geo-distributed cloud. Section 5.3 presents related works. Section 5.4 describes our proposed system architecture for executing workflow with cached data on a multisite cloud. Section 5.5 describes both our cost model and our algorithms. The cost model is used to minimize the execution time, the intermediate data transfer time and the cache data transfer time. The scheduling algorithms efficiently select the execution site for each workflow fragment taken into account the cache management. Then, section 5.6 presents the experimental evaluation based on the execution of the Phenomenal workflow on the IFB-cloud [1] using a large amount of time series of raw images generated by the PhenoArch platform. The results reveal that our algorithm significantly outperforms two baseline algorithms (execution on multisite cloud without cache, and execution on monosite cloud with a centralized cache). Finally, section 5.7 resumes and concludes this chapter.

[1] https://www.france-bioinformatique.fr/cloud-ifb/

5.1 Introduction

In many scientific domains, *e.g*, bio-science [81], complex numerical experiments typically require many processing or analysis steps over huge datasets. They can be represented as workflow These workflows facilitate the modeling, management, and execution of computational activities linked by data dependencies. As the size of the data processed and the complexity of the computation keep increasing, these workflows become data-intensive [81], thus requiring high-performance computing resources.

Today, all popular public clouds, *e.g*, Microsoft Azure, Amazon EC2, and Google Cloud, provide a multisite option with the capability of using multiple sites with a single cloud account, The main reason for using multiple cloud sites to execute data-intensive workflows is that they often exceed the capabilities of a single site, either because the site imposes usage limits for fairness and security, or simply because the datasets are too large.

In scientific applications, there can be much heterogeneity in the storage and computing capabilities of the different sites, *e.g*, on premise servers, HPC platforms from research organizations or federated cloud sites at the national level [42]. As an example in plant phenotyping, greenhouse platforms generate terabytes of raw data from plants. Such data is typically stored at data centers geographically close to the greenhouse to minimize the impact of data transfers. However, the computation power of those data centers may be limited and fail to scale when the analyses become complex, such as in plant modeling or 3D reconstruction. In this case, other computation sites are then required.

Most workflow management systems, can execute workflows in the cloud [113]. Some examples are Swift/T [163], Pegasus [47], SciCumulus [116], Kepler [8] and OpenAlea [130]. Our work is based on OpenAlea [130], which is widely used in plant science for simulation and analysis [129]. Most existing systems use naive or manual approaches to distribute the tasks across sites. The problem of scheduling a workflow execution over a multisite cloud has started to be addressed in [101], using performance models to predict the execution time on different resources. In [95], it is proposed a solution based on multi-objective scheduling and a single site virtual machine provisioning approach, assuming homogeneous sites, as in public cloud.

Since it is common for workflow users to reuse code or data from other workflows and from previous executions of the same workflow [58], a promising approach for efficient workflow execution is to cache intermediate data in order to avoid re-executing entire workflows. Furthermore, a user may need to re-execute a workflow many times with different sets of parameters and input data depending on the previous results. Workflow fragments, *i.e.*, subsets of workflow activities and dependencies, can often be reused. Another important benefit of caching intermediate data is to make it easy to share with other research teams, thus fostering new analyses at low cost.

Caching has been supported by some SWMS, *e.g*, Kepler [7], VisTrails [28] and OpenAlea [128]. In [72], we proposed an adaptive caching method for OpenAlea that automatically determines the most suited intermediate data to

cache, taking into account workflow fragments, but only in the case of a single cloud site. Another interesting single site method, also exploiting workflow fragments, is to compute the ratio between re-computation cost and storage cost to determine what intermediate data should be stored [171]. All these methods are designed for a single site. The only distributed caching method for workflow execution in a multisite cloud we are aware of is restricted to hot metadata (frequently accessed metadata) and ignores intermediate data [94].

Caching data in a multisite cloud with heterogeneous sites is much more complex. In addition to the trade-off between re-computation and storage cost at single sites, there is the problem of site selection for placing cached data. The problem is more difficult than data allocation in distributed databases [120], which deals only with well-defined base data, not intermediate data produced. Furthermore, the scheduling of workflow executions must be cache-aware, *i.e.*, exploit the knowledge of cached data to decide between reusing and transferring cached data versus re-executing the workflow fragments.

In this chapter, we propose a solution for cache-aware scheduling of workflow in multisite cloud. Our solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection and dynamic workflow scheduling. We implemented our caching solution in OpenAlea. Based on a real data-intensive application in plant phenotyping, we provide an extensive experimental evaluation using a cloud with three heterogeneous sites.

5.2 Use Case: Intermediate Data Reuse in Geo-distributed Clouds

Different users can conduct different analyses by executing some workflow fragments on the same dataset to test different hypotheses [72]. To save both time and resources, it may be useful to reuse the corresponding intermediate data that has already been computed rather than recompute the fragments again. In our use case, we suppose that workflow executions are done in serial order and that workflows are submitted from the site where the raw data is produced.

Figure 5.1 shows two workflows used in plant analysis: the Phenomenal workflow (Wf1) and a workflow to simulate light competition for plants in greenhouse (Wf2). Both workflows use fragments F1 (binarization) and F2 (3D reconstruction), so the subsequent execution of Wf2 may benefit from reusing the data generated previously from the corresponding fragments in Wf1. Suppose for instance that Wf1 execution has generated some data that has been cached, as shown in Figure 5.2.4. Then, a user can reuse datasets D1 and D2 to speed up the execution of Wf2. Thus, the only fragment that requires to be executed is F7.

The raw data is produced by the Phenoarch platform, which has a capacity of managing 1,680 plants within a controlled environment (*e.g*, temperature, humidity, irrigation) and automatic imaging through time. The total size of the raw image dataset for one experiment is 11 Terabytes. The raw data is

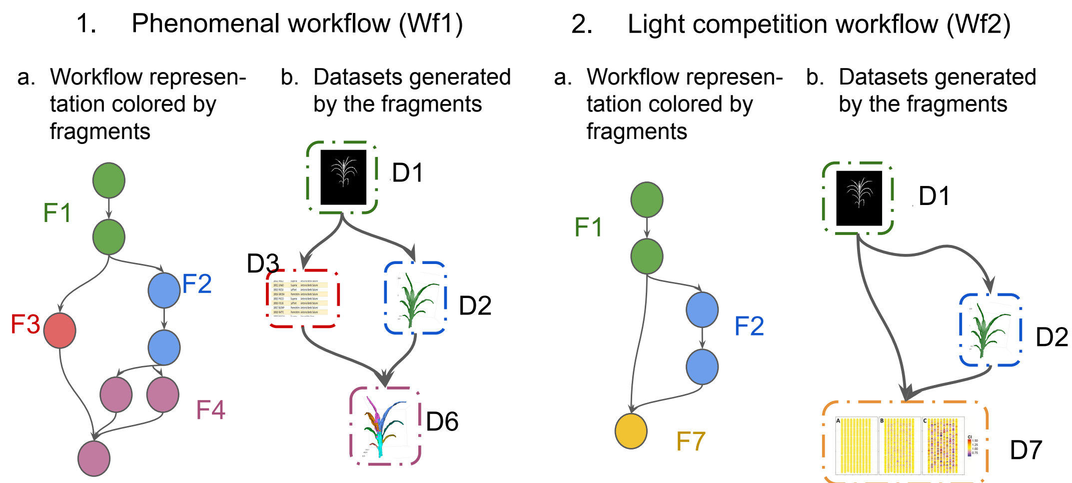


FIGURE 5.1: Two workflows in plant analysis and their intermediate data (the shared activities have same color)

stored on a server at the same location as the experimental platform. This server is considered as a cloud site and has both data storage and computing resources. However, these computing resources are not enough to process a full workflow analysis in a relatively short time. Thus, scientists who need to analyse the full workflow need more computational resources, requiring to share resources with existing distributed cloud sites. Therefore the execution of the workflow is distributed onto several cloud sites.

The multisite cloud architecture (see Figure 5.2.4) is composed of heterogeneous sites, in terms of computing and storage capacities. The site with the raw data is used to execute some Phenomenal fragments that do not require powerful resources. Whenever more computational resources are needed, it is necessary to choose between transferring the raw data or some intermediate data to a more powerful site, or re-executing some fragments locally before transferring intermediate data.

Figure 5.2.4 shows a case where a user submits the Phenomenal workflow on the multisite cloud. The workflow fragments are distributed and executed on different cloud sites depending on the site resources. At each site, some intermediate data generated by the fragment execution is stored in a cache to be reused in other fragment executions.

5.3 Related Works

Several workflow systems provide caching and reuse of intermediate data during workflow execution [28, 130]. However, no solution we are aware of takes into account the geo-distributed aspect of workflow execution when using cached data. There is no definitive solution for two important problems: 1) how to determine what intermediate data should be cached, taking into account data transfers; 2) how the workflow should be scheduled when the intermediate data and the cached data processed are distributed over a

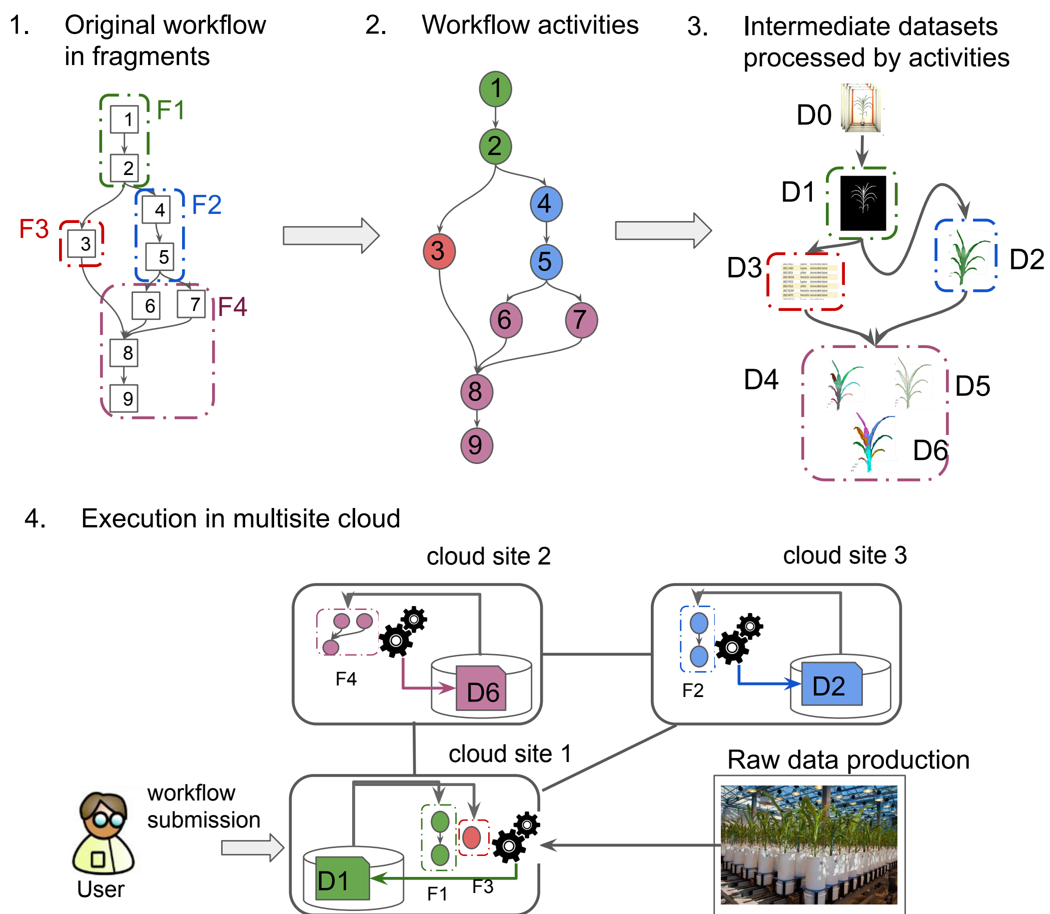


FIGURE 5.2: Phenomenal plant analysis workflow

multisite cloud. The related work provides two kinds of methods, either to reuse intermediate data in a monosite cloud or to optimize workflow execution in multisite cloud, but without any reuse of intermediate data.

Several workflow systems, such as Kepler, VisTrails, OpenAlea, exploit intermediate data for more efficient workflow execution. Each system has its unique way of addressing data reuse. VisTrails provides visual analysis of workflow results and provenance, *i.e.*, captures the graph of execution and the intermediate data generated [28]. The intermediate data stored is reused when tasks are re-executed on a local computer. The user can then change some activities and parameters in the workflow and efficiently re-execute each workflow activity to analyze the different results. The intermediate data cache is used to enhance reproducibitily when associated with provenance metadata [85]. Caching and reuse of intermediate data is done whenever it is possible, but does not scale up as the data size increases, *i.e.*, the data cannot be stored locally. Finally, VisTrails does not take distribution into account when storing and using the cache. Our approach is different as it works in a distributed environment where data transfer costs may be significant.

When storing intermediate data in the cloud, the trade-off between the cost of re-executing tasks and the costs of storing intermediate data is not easy to estimate [2, 48]. Yuan et al. [169] propose an algorithm based on the ratio between re-computation cost and storage cost at the task level. The algorithm uses the provenance data to generate a graph of the intermediate datasets dependencies. Then, the cost of storing each intermediate data set is weighted by the number of dependencies in the graph. The algorithm determines the optimized set of intermediate datasets to store. It is improved in [171] to take into account workflow fragments, yielding near optimal caching of intermediate datasets. However, this approach requires global knowledge of executions, such as the execution time of each task, the size of each dataset and the number of incoming re-executions, which is hard to monitor in practice. Furthermore, it does not take data transfer into account.

Kepler [7] provides intermediate data caching that can be used by workflows executed on a monosite cloud. The cache data is stored on a remote server. When a workflow is re-executed, the workflow is modified to access the cached data. Then, all the cached data that will be reused by the workflow is sent to the cloud where the workflow will be executed. This solution is improved in [32] to store the cache data on the same cloud site than the execution. This method select which intermediate data that will be cached by using an algorithm based on Ant Colony System optimization to find a near optimum data caching policy. This solution does not take data transfer into account and only works on monosite cloud. Our approach is different as it manages cache and workflow execution on multisite cloud.

OpenAlea [130] uses caching both in memory and on disk. In memory caching is used on a local computer for smaller workflow execution. Disk-based caching is based on an adaptive cache decision that automatically determines which intermediate data is to be stored [72]. This solution only works on monosite cloud and the cached data is always reused.

Multisite cloud scheduling algorithms have been proposed to allow distributed workflow execution on multiple sites. Liu et al. [96] propose a scheduling algorithm based on data location, that minimizes data transfer during workflow executions. The algorithm is further improved in [95] with a multi-objective cost function, which includes the time and monetary cost of workflow execution in a dynamic environment. Duan *et al.* [17] propose another interesting approach based on a multisite multi-objective cost function, considering a multisite environment with different bandwidths and processing power. These distributed scheduling approaches focus on optimizing workflow execution, but do not consider caching and reusing intermediate data.

5.4 Multisite SWMS Architecture

In this section, we present our workflow system architecture that integrates caching and reuse of intermediate data in a multisite cloud. We motivate our design decisions and describe our architecture in two ways: in terms of functional layers (see Figure 5.3), which shows the different functions and components; and in terms of nodes and components (see Figure 5.4), which are involved in the processing of workflows.

Our architecture capitalizes on the latest advances in distributed and parallel data management to offer performance and scalability [120]. We consider a distributed cloud architecture with on premise servers, where raw data is produced, *e.g*, by a phenotyping experimental platform in our use case, and remote sites, where the workflow is executed. The remote sites are data centers using shared-nothing clusters, *i.e.*, clusters of server machines, each with independent processors, disk and memory. We adopt shared-nothing as it is the most scalable and cost-effective architecture for big data analysis.

In the cloud, metadata management has a critical impact on the efficiency of workflow scheduling as it provides a global view of data location, *e.g*, at which nodes some raw data is stored, and enables task tracking during execution [94]. We organize the metadata in three repositories: catalog, provenance database and cache index. The catalog contains all information about users (access rights, etc.), raw data location and workflows (code libraries, application code). The provenance database captures all information about workflow execution and the workflow specification. The cache index contains information about tasks and cache data produced, as well as the location of files that store the cache data. Thus, the cache index itself is small (only file references) and the cached data can be managed using the underlying distributed file system. A good solution for implementing these metadata repositories is a key-value store, such as Cassandra[2], which provides efficient key-based access, scalability and fault-tolerance through replication in a shared-nothing cluster.

The raw data (files) are initially produced and stored at some cloud sites, *e.g*, in our use case, at the phenotyping platform. During workflow execution,

[2] `https://cassandra.apache.org`

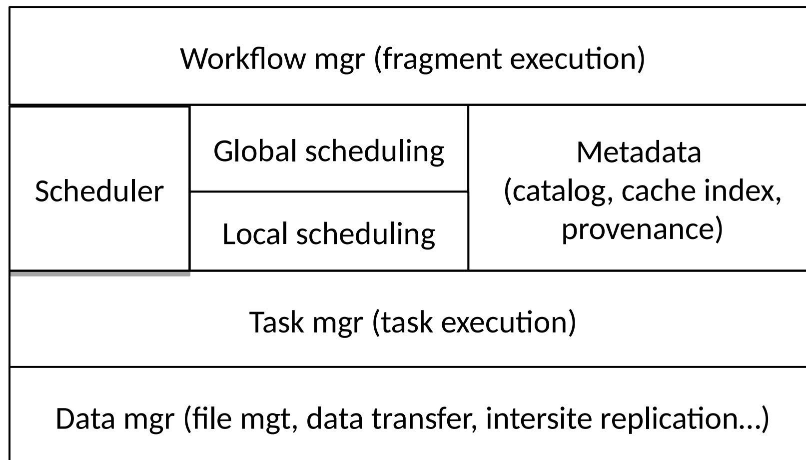


FIGURE 5.3: Workflow System Functional Architecture

the intermediate data is generated and consumed at one site's node in memory. It gets written to disk when it must be transferred to another node (potentially at the same site) or explicitly added to the cache. Later on, the cached data (files) can be replicated at other sites to minimize data transfers.

Figure 5.3 extends the workflow system architecture proposed in [93] for single site. It is composed of six modules: workflow manager, global scheduler, local scheduler, task manager, data manager and metadata manager, to support both execution and intermediate data caching in a multisite cloud. The workflow manager provides a user interface for workflow definition and processing. Before workflow execution, the user selects a number of virtual machines (VMs), given a set of possible instance formats, *i.e.*, the technical characteristics of the VMs, deployed on each site's nodes. When a workflow execution is started, the workflow manager simplifies the workflow by removing some workflow fragments and partitions depending on the raw input data and the cached data (see Section 5.5). The global scheduler uses the metadata (catalog, provenance database, and cache index) to schedule the workflow fragments of the simplified workflow. The VMs on each site are then initialized, *i.e.*, the programs required for the execution of the tasks are installed and all parameters are configured. The local scheduler schedules the workflow fragments received on its VMs.

The data manager module handles data transfers between sites during execution (for both newly generated intermediate data and cached data) and manages cache storage and replication. At a single site, data storage is distributed between cluster nodes. Finally, the task manager manages the execution of fragments on the VMs at each site. It exploits the provenance data to decide whether or not the task's output data should be placed in the cache, based on the cache provisioning algorithm described in Section 5.5. Local scheduling and execution can be performed as in [72].

Figure 5.4 shows how these components are organized, using the traditional master-worker model, in a multisite cloud. Each site provides the same functionality, *i.e.*, all the components in Figure 5.3. Thus, users can trigger a

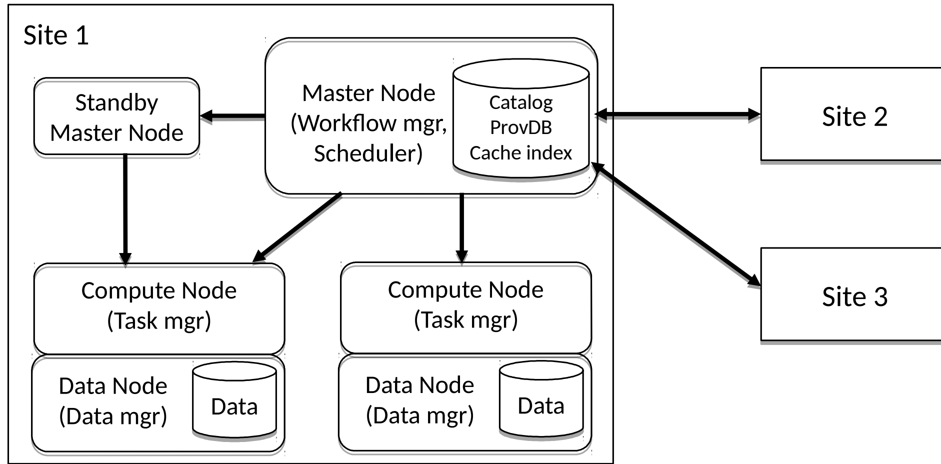


FIGURE 5.4: Multisite Workflow System Architecture

workflow execution at any site. However, for a given workflow execution, there is one coordinator site, where the execution is started. The coordinator site performs workflow management and global scheduling, and manages the execution with other participant sites. The workflow manager and the global scheduler modules are involved only on the coordinator site while all other modules are involved on all sites.

At each site, there are three kinds of nodes: master, compute and data nodes, which are mapped to cluster nodes at configuration time, *e.g*, using a cluster manager like Yarn (http://hadoop.apache.org). Each site has one active master node and a standby node to deal with master node failure. The master nodes are the only ones to communicate across sites. Each master node supports the top layers of the functional architecture: workflow manager, global scheduler, local scheduler and metadata management.

The master nodes are responsible for transferring data between sites during execution. They are lightly loaded as most of the work of serving clients is done by the compute and data nodes (or worker nodes), which perform local execution and data management, respectively.

5.5 Cache-aware Workflow Execution

In this section, we present in more details the cache-aware workflow execution in multisite cloud. In particular, the global scheduler must decide which data to cache (cache data selection) and where (cache site selection), and where to execute workflow fragments (execution site selection). Since these decisions are not independent, we propose a cost function to make a global decision, based on the cost components for individual decisions. We start by giving an overview of distributed workflow execution. Then, we present the methods and cost functions for cache data selection, cache site selection, execution site selection and global decision. Finally, we introduce our algorithms for cache-aware scheduling.

5.5.1 Distributed Workflow Execution Overview

We consider a multisite cloud with a set of sites S={$s_1, ..., s_n$}. A workflow $W(A, D)$ is a a directed acyclic graph (DAG) of computational activities A and their data dependencies D. A task t is the instantiation of an activity during execution with specific associated input data. A fragment f of an instantiated workflow is a subset of tasks and their dependencies.

The execution of a workflow $W(A, D)$ in S starts at a coordinator site s_c and proceeds in three main steps:

1. The global scheduler at s_c simplifies and partitions the workflow into fragments. Simplification uses metadata to decide whether a task can be replaced by corresponding cached data references. Partitioning uses the dependencies in D to produce fragments.

2. For each fragment, the global scheduler at s_c computes a cost function to make a global decision on which data to cache, where, and on which site to execute. Then, it triggers fragment execution and data placement in cache at the selected sites.

3. At each selected site, the local scheduler performs the execution of its received fragments using its task manager (to execute tasks) and data manager (to transfer the required input data). It also applies the decision of the global scheduler on storing new intermediate data into the cache.

We introduce basic cost functions to reflect data transfer and distributed execution. The time to transfer some data d from site s_i to site s_j, noted $T_{tr}(d, s_i, s_j)$, is defined by

$$T_{tr}(d, s_i, s_j) = \frac{Size(d)}{TrRate(s_i, s_j)} \tag{5.1}$$

where $TrRate(s_i, s_j)$ is the transfer rate between s_i and s_j.

The time to transfer input and cached data, $In(f)$ and $Cached(f)$ respectively, to execute a fragment f at site s_i is $T_{input}(f, s_i)$:

$$T_{input}(f, s_i) = \sum_{s_j}^{S}(T_{tr}(In(f), s_j, s_i) + T_{tr}(Cached(f), s_j, s_i)) \tag{5.2}$$

The time to compute a fragment f at site s, noted $T_{compute}(f, s)$, can be estimated using Amdahl's law [174]:

$$T_{compute}(f, s) = \frac{(\frac{\alpha}{n} + (1 - \alpha)) * W(f)}{CPU_{perf}(s)} \tag{5.3}$$

where $W(f)$ is the workload for the execution of f, $CPU_{perf}(s)$ is the average computing performance of the CPUs at site s and n is the number of CPUs at site s. We suppose that the local scheduler may parallelize task executions. Therefore, α represents the percentage of the workload that can be executed in parallel.

The expected waiting time to be able to execute a fragment at site s is noted $T_{wait}(s)$, which is the minimum expected time for s to finish executing the fragments in its queue.

The time to transfer the intermediate data generated by fragment f at site s_i to site s_j, noted $T_{write}(Output(f), s_i, s_j)$, is defined by:

$$T_{write}(Output(f), s_i, s_j) = T_{tr}(Output(f), s_i, s_j) \tag{5.4}$$

where $Output(f)$ is the data generated by the execution of f.

5.5.2 Cache Data Selection

To determine what new intermediate data to cache, we consider two different methods: greedy and adaptive. Greedy data selection simply adds all new data to the cache. Adaptive data selection extends our method proposed in [72] to multisite cloud. It achieves a good trade-off between the cost saved by reusing cached data and the cost incurred to feed the cache.

To determine if it is worth adding some intermediate data $Output(f)$ at site s_j, we consider the trade-off between the cost of adding this data to the cache and the potential benefit if this data was reused. The cost of adding the data to site s_j is the time to transfer the data from the site where it was generated. The potential benefit is the time saved from loading the data from s_j to the site of computation instead of re-executing the fragment. We model this trade-off with the ratio between the cost and benefit of the cache, noted $p(f, s_i, s_j)$, which can be computed from Equations 5.2, 5.3 and 5.4,

$$p(f, s_i, s_j) = \frac{T_{write}(Output(f), s_i, s_j)}{T_{input}(f, s_i) + T_{compute}(f, s_i) - T_{tr}(Output(f), s_j, s_i)} \tag{5.5}$$

In the case of multiple users, the probability that $Output(f)$ will be reused or the number of times fragment f will be re-executed is not known when the workflow is executed. Thus, we introduce a threshold *Threshold* (computed by the user) as the limit value to decide whether a fragment output will be added to the cache. The decision on whether $Output(f)$ generated at site s_i is stored at site s_j can be expressed by

$$d_{f,i,j} = \begin{cases} 1 & \text{if } p(f, s_i, s_j) < Threshold. \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

5.5.3 Cache Site Selection

Cache site selection must take into account the data transfer cost and the heterogeneity of computing and storage resources. We propose two methods to balance either storage load (*bStorage*) or computation load (*bCompute*) between sites. The *bStorage* method allows for preventing bottlenecks when loading cached data. To assess this method at any site s, we use a load indicator, noted $L_{bStorage}(s)$, which represents the relative storage load as the ratio between

the storage used for the cached data ($Storage_{used}(s)$) and the total storage ($Storage_{total}(s)$).

$$L_{bStorage}(s) = \frac{Storage_{used}(s)}{Storage_{total}(s)} \tag{5.7}$$

The *bCompute* method balances the cached data between the most powerful sites, *i.e.*, with more CPUs, to prevent computing bottlenecks during execution. Using the knowledge on the sites' computing resources and usage, we use a load indicator for each site s, noted $L_{bCompute}(s)$, based on CPUs idleness ($CPU_{idle}(s)$) versus total CPU capacity ($CPU_{total}(s)$).

$$L_{bCompute}(s) = \frac{1 - CPU_{idle}(s)}{CPU_{total}(s)} \tag{5.8}$$

The load of a site s, depending on the method used, is represented by $L(s)$, ranging between 0 (empty load) and 1 (full). Given a fragment f executed at site s_i, and a set of sites s_j with enough storage for $Output(f)$, the best site s^* to add $Output(f)$ to its cache can be obtained using Equation 5.1 (to include transfer time) and Equation 5.6 (to consider multiple users).

$$s^*(f)_{s_i} = \underset{s_j}{\text{argmax}}(d_{f,i,j} * \frac{(1 - L(s_j))}{T_{write}(Output(f), s_i, s_j)}) \tag{5.9}$$

5.5.4 Execution Site Selection

To select an execution site s for a fragment f, we need to estimate the execution time for f as well as the time to feed the cache with the result of f. The execution time f at site s ($T_{execute}(f, s)$) is the sum of the time to transfer input and cached data to s, the time to get computing resources and the time to compute the fragment. It is obtained using Equations 5.2 and 5.3.

$$T_{execute}(f, s) = T_{input}(f, s) + T_{compute}(f, s) + T_{wait}(s) \tag{5.10}$$

Given a fragment f executed at site s_i and its intermediate data $Output(f)$, the time to write $Output(f)$ to the cache (T_{feed_cache} (f, s_i, s_j)) can be defined as:

$$T_{feed_cache}(f, s_i, s_j, d_{f,i,j}) = d_{f,i,j} * T_{write}(Output(f), s_i, s_j) \tag{5.11}$$

where s_j is given by Equation 5.9.

5.5.5 Global Decision

At Step 2 of workflow execution, for each fragment f, the global scheduler must decide on the best combination of individual decisions regarding cache data, cache site, and execution site. These individual decisions depend on each other. The decision on cache data depends on the site where the data is generated and the site where it will be stored. The decision on cache site depends on the site where the data is generated and the decision of whether or not the data will be cached. Finally, the decision on execution site depends

on what data will be added to the cache and at which site. Using Equations 5.10 and 5.11, we can estimate the total time (T_{total}) for executing a fragment f at site s_i and adding its intermediate data to the cache at another site s_j:

$$T_{total}(f, s_i, s_j, d_{f,i,j}) = T_{execute}(f, s_i) + T_{feed_cache}(f, s_i, s_j, d_{f,i,j}) \tag{5.12}$$

Then, the global decision for cache data ($d_{f,i,j}$), cache site (s^*_{cache}) and execution site (s^*_{exec}) is based on minimizing the following equation for the n^2 pairs of sites s_i and s_j

$$(s^*_{exec}, s^*_{cache}, d_{f,i,j}) = \underset{s_i, s_j}{\operatorname{argmin}}(T_{total}(f, s_i, s_j, d_{f,i,j})) \tag{5.13}$$

This decision is performed by the coordinator site before each fragment execution and only takes into account the cloud site's status at that time. It is worth mentioning that s^*_{exec} and s^*_{cache} can be the same site, including the coordinator site.

5.5.6 Cache-Aware Scheduling

In this section, we present in details our solution to cache-aware scheduling in our architecture. We propose three algorithms: *CacheA*, *Sgreedy-CH* and *Fgreedy-CH*. *CacheA* is a new greedy algorithm that performs cache-aware scheduling. The two other algorithms extend distributed greedy scheduling algorithms [95] to become cache-aware. These three algorithms are dynamic, *i.e.*, they produce scheduling plans that distribute and allocate executable tasks to computing nodes during workflow execution [93]. This kind of scheduling is appropriate for our workflows, where the workload is difficult to estimate, or for environments where the capabilities of the computers varies much during execution.

CacheA.

The *CacheA* algorithm (see Algorithm 1) is based on the global decision (see Equation 5.13) made by the coordinator site. It takes the simplified workflow graph as input and, starting from the root fragment, computes the global decision for each fragment. Recall that the global decision combines individual decisions regarding cache data, cache site, and execution site, before scheduling each fragment.

Algorithm 1 proceeds as follows. The workflow is partitioned into fragments (line 1), where F represents the set of all fragments of the workflow. Whenever a fragment is ready for execution, it is selected (line 3). Then (line 4), we compute the global decision using Equation 5.13 to determine the best execution site S_{exec}, cache placement site S_{cache} and cache decision $d_{f,i,j}$. At line 5, the fragment is transferred to the site S_{exec} to be executed. Recall that the cache decision $d_{f,i,j}$ determines if the intermediate data will be cached. Whenever the intermediate data is to be stored in the cache (lines 6-8), it is then transferred at site S_{cache} (line 7). At line 8, the *Cache Index* is updated locally and the update is propagated at all replicas at other sites. Finally (line 11), the fragment is removed from F.

Algorithm 1: CacheA

```
Input: WF: a workflow,
Cache index: the index of the placement of the data existing in the cache
1  F ← partition WF into fragment;
2  while F not empty do
3      f ← select a fragment of F that is next to be computed ;
4      S_exec, S_cache, d_{f,i,j} ← compute from Equation 5.13 ;
5      Schedule f execution on site S_exec ;
6      if d_{f,i,j} is True then
           /* The intermediate data is cached */
7          Place the intermediate data on site S_cache;
8          Update the Cache Index
9      else
           /* The intermediate data will not be cached */
10     end
11     Remove f from F;
12 end
```

Sgreedy-CH and Fgreedy-CH

.

Sgreedy-CH (site greedy with caching) extends the *SiteGreedy* algorithm presented in [95]. The scheduling decision of *SiteGreedy* works as follows. Let F be the set of workflow fragments. Whenever a cloud site s is available, it requests the execution of a ready fragment in F to the coordinator site. The selection of the fragment is based on a cost function that takes into account data transfer time and execution time. The idea of this scheduling is to keep cloud sites as busy as possible, scheduling a fragment whenever a site is available. The caching decision is done after the workflow fragment is scheduled for execution by the local scheduler at each site. Unlike *CacheA*, *Sgreedy-CH* does not make a global decision. Instead, the caching decision is done in two steps. First, the choice of the cloud site where the cached data should be stored is determined by Equation 5.9. Then, the decision on whether or not to store the intermediate data is determined by Equation 5.5, considering the execution time and the time to transfer the intermediate data. Note that Equation 5.5 considers both the execution site and the cache site, which are already determined when computed during the execution of *Sgreedy-CH*.

Fgreedy-CH (fragment greedy with caching) extends the *ActGreedy* algorithm presented in [95]. *Fgreedy-CH* schedules each workflow fragment at the site that minimizes a cost function based on execution time and input data transfer time. The cost function is the sum of the initialization time, expected execution time, and data transfer time for each fragment at each site. Then, after each fragment execution at the selected site, the local scheduler performs the cache site and cache data decisions based on Equations 5.9 and 5.5.

These two greedy algorithms generate a dynamic scheduling plan. After

each fragment execution, the decision concerning caching is made by the local scheduler. In contrast, *CacheA* makes a global decision for each fragment. Notice that *Sgreedy-CH* and *Fgreedy-CH* needs less operations to schedule the fragments.

5.6 Experimental Validation

In this section, we first present our experimental setup, which features a heterogeneous multisite cloud with multiple users who re-execute part of the workflow. Then, we compare the performance of our multisite cache scheduling algorithms against two baseline algorithms. We end the section with concluding remarks.

5.6.1 Experimental Setup

We use a real multisite cloud, with three sites, in France. *Site 1* in Montpellier is the raw data server of the Phenoarch phenotyping platform, with the smallest number of CPUs and largest amount of storage among the sites. The raw data is stored at this site. *Site 2* is the coordinator site, located in Lille. *Site 3*, located in Lyon, has the largest number of CPUs and the smallest amount of storage.

To model site heterogeneity in terms of storage and CPU resources, we use heterogeneity factor H in three configurations: $H = 0$, $H = 0.3$ and $H = 0.7$. For the three sites altogether, the total number of CPUs is 96 and the total storage 180 GB. With $H = 0$ (homogeneous configuration), each site has 32 CPUs and 60 GB. With $H = 0.3$, we have 22 CPUs and 83 GB for Site 1, 30 CPUs and 57 GB for Site 2 and 44 CPUs and 40 GB for Site 3. With $H = 0.7$ (most heterogeneous configuration), we have 6 CPUs and 135 GB for Site 1, 23 CPUs and 35 GB for Site 2 and 67 CPUs and 10 GB for Site 3.

The input dataset for the Phenomenal workflow is produced by the Phenoarch platform (see Section 5.2). Each execution of the workflow is performed on a subset of the input dataset, *i.e.*, 200 GB of raw data, which represents the execution of 15,000 tasks. For each user, 60% of the raw data is reused from previous executions. Thus, each execution requires only 40% of new raw data. For the first execution, no data is available in the cache.

We implemented our solution in OpenAlea and deployed it at each site using the Conda multi-OS package manager. The metadata distributed database is implemented using Cassandra. Communication between the sites is done using the protocol library ZeroMQ. Data transfer between sites is done through SSH.

We have also implemented two baseline algorithms: 1) *ActGreedy*, a multisite scheduling algorithm described in [95], and, 2) a centralized cache architecture for workflow execution. The algorithm *ActGreedy* minimize the execution time by scheduling the fragments at cloud site given a cost function based on execution time and input data transfer time. This algorithm does not reuse and store intermediate data in cache for future performance improvement.

Table 5.1 summarizes the different variants of the scheduling algorithms used in our experiments. Prefix "C-" indicates that the cache is centralized at a single site while prefix "D-" that it is distributed. For all algorithms that use cache, the cache index is fully replicated at all sites.

Algorithm	Cost function parameters	Cache decision	Cache placement
ActGreedy	Execution time of Act. & input transfer time	Local after execution	No cache
C-CacheA	Execution time, Frag. Input & Cache transfer time	Global per frag. before execution	Single cache site
C-Sgreedy-CH	Execution time & Frag. Input transfer time	Local after execution	Single cache site
C-Fgreedy-CH	Execution time & Frag. Input transfer time	Local after execution	Single cache site
D-CacheA	Execution time, Frag. Input & Cache transfer time	Global per frag. before execution	Distributed
D-Sgreedy-CH	Execution time & Frag. Input transfer time	Local after execution	Distributed
D-Fgreedy-CH	Execution time & Frag. Input transfer time	Local after execution	Distributed

TABLE 5.1: Scheduling algorithms and their main dimensions

5.6.2 Experiments

We compare the three algorithms we proposed (*CacheA*, *Sgreedy-CH*, *Fgreedy-CH*) in terms of execution time and amount of data transferred with two baselines. The total time is defined as the workflow execution time plus the transfer time. We consider different workflow executions: with and without caching (Experiment 1); on monosite or multisite cloud (Experiment 2); and using a centralized or distributed cache (Experiment 3). Then, we consider multiple users who execute the workflow in the following cases: on the same multisite configuration, where 60% of the data is the same (Experiment 4); on different multisite configurations (Experiment 5); and when adding or removing workflow fragments (Experiment 6).

Experiment 1: with and without caching.

In this experiment, we compare two workflow executions: with caching, using the *D-CacheA* scheduling algorithm and the *bStorage* load balancing method; and without caching, using the *ActGreedy* algorithm. We consider one re-execution of the workflow on different input datasets, from 0% to 60% of data reuse. *D-CacheA* outperforms *ActGreedy* from 20% of reused data. Below 20%, the overhead of caching outweighs its benefit. For instance, with no reuse (0%), the total time with *D-CacheA* is 16% higher than with *ActGreedy*. But with 30%, it is 11% lower, and with 60%, it is 42% lower.

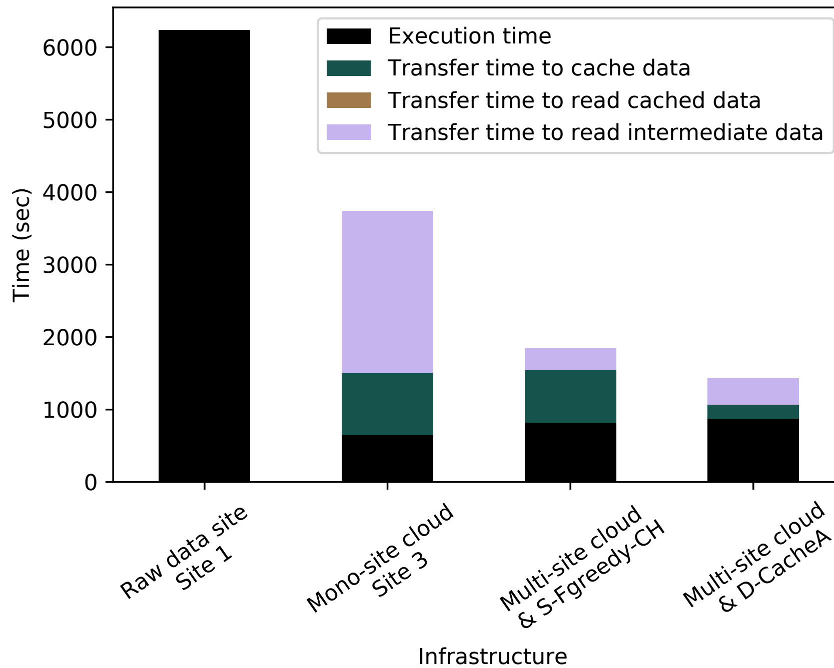


FIGURE 5.5: Total time of Phenomenal workflow execution in four cases

Experiment 2: single site versus multisite execution.

In this experiment, we compare the total time in four cases with monosite and multisite clouds:

1. raw data site, *i.e.*, a monosite cloud (Site 1) where the raw data is stored but with only 10 CPUs;
2. monosite cloud (Site 3) with 96 CPUs with the raw and cache data having to be transferred from the raw data site (that will not do computation);
3. multisite cloud composed of three sites with configuration $H = 0.7$, using *C-Fgreedy-CH*;
4. same multisite cloud but using *D-CacheA*.

Figure 5.5 shows the total time of the workflow for the different cases. When executing on the raw data site (first chart on Figure 5.5), all the input data is already stored on Site 1 and the cached data is stored on this site, thus there are no data transfer between sites during workflow execution. However, due to the reduced number of available CPUs, the total time is by far longer than on any other infrastructure (66% longer that of monosite (Site 3), 238% longer than on multisite with *C-Fgreedy-CH* and 334% longer than on multisite with *D-CacheA*. An execution of the workflow on the full raw dataset on Site 1

takes more than a week to compute. In practice, the raw dataset is first sent to a cloud site with more computation resources available before being executed.

The monosite 3 cloud yields the shortest execution time, outperforming the multisite cloud with *C-Fgreedy-CH* and *D-CacheA* in terms of execution time by 21% and 26% respectively. However, its data transfer time (for transferring the raw data) makes its total time much longer, so it is outperformed by the multisite cloud using *D-CacheA* by 61%.

The intermediate data transfer time on the multisite cloud is much smaller (83% smaller for the *D-CacheA*) than the raw data transfer time to execute on monosite 3. In multisite cloud, fragments can be executed on the site of their input data. In this case, the raw data is not transferred between sites, but locally processed on Site 1 by the first workflow fragment. The intermediate data generated by the first fragment is smaller than the raw data and is more easily transferred to other sites, where the other fragments are scheduled.

Experiment 3: centralized versus distributed cache.

Figure 5.6 shows the total time of the workflow for the three algorithms *Sgreedy-CH*, *Fgreedy-CH* and *CacheA*. The algorithms used with a centralized cache on Site 1 are *C-Sgreedy-CH*, *C-Fgreedy-CH* and *C-CacheA*. They are compared with *D-CacheA*, which uses a distributed cache in two configurations: (a) with two users; (b) with different site heterogeneity.

Let us first analyze the results of Figure 5.6.a, where two users execute the Phenomenal workflow with 60% of common raw data in two configurations: centralized cache on Site 1 and distributed cache with $H = 0.7$. For the first user execution, *D-CacheA* outperforms *C-Sgreedy-CH* in terms of total time by 43%. This is due to *D-CacheA* outperforming *C-Sgreedy-CH* in terms of intermediate and cache data transfer times by 66% and 61% respectively. *D-CacheA* outperforms *C-Fgreedy-CH* in terms of total time by 22%, even though *D-CacheA*'s execution time is lower than *C-Fgreedy-CH* (6%). This is due to *D-CacheA* outperforming *C-Fgreedy-CH* in terms of data transfer time by 45%. *D-CacheA* outperforms *C-CacheA* in terms of total time by 10%. The execution time and intermediate data transfer time are similar (7% shorter and 12% longer). Yet *D-CacheA* outperforms *C-CacheA* in terms of cache data transfer by 42%. For the first execution *D-CacheA* outperforms the three algorithms with centralized cache, mostly due to shorter data transfer times. This is because the distributed cache enables the workflow to be executed at a site with more computing resources and store the cached data on that site. For re-execution, *D-CacheA* outperforms the three algorithms with centralized cache, *C-Sgreedy-CH*, *C-Fgreedy-CH* and *C-CacheA*, in terms of total time by 63%, 46% and 21%, respectively.

Figure 5.6.b shows the total time of the workflow for the second user and the four different algorithms: *C-Sgreedy-CH*, *C-Fgreedy-CH*, *C-CacheA* and *D-CacheA*. The execution is performed on three values of H in two configurations: centralized cache on Site 1 and distributed cache. In any configuration, *D-CacheA* outperforms the three other algorithms with centralized cache, *C-Sgreedy-CH*, *C-Fgreedy-CH* and *C-CacheA*, in terms of total time by 43%, 31%

and 14%, respectively for $H = 0$, by 52%, 38% and 23%, respectively for $H = 0.3$, and by 60%, 42% and 19%, respectively for $H = 0.7$. The performance gain is due to less data transfers. For $H = 0$, *D-CacheA* outperforms *C-Sgreedy-CH* in terms of intermediate data, cached data transfer by 49% and 70%, respectively.

Experiment 4: multiple users.

Figure 5.7 shows the total time of the workflow for the three scheduling algorithms, four users, $H = 0.7$, and our two cache site selection methods: (a) *bStorage*, and (b) *bCompute*.

Let us first analyze the results in Figure 5.7.a (*bStorage* method). For the first user execution, *D-CacheA* outperforms *D-Sgreedy-CH* in terms of execution time by 8% and in terms of data and intermediate data transfer time by 51% and 63%, respectively. The reason *D-Sgreedy-CH* is slower is that it schedules some compute-intensive fragments at Site 1, which has the lowest computing resources. Furthermore, it does not consider data placement and transfer time when scheduling fragments.

Again for the first user execution, *D-CacheA* outperforms *D-Fgreedy-CH* in terms of total time by 24%, when considering the time to transfer data the cache. However, its execution time is a bit slower (by 9%). The reason that *D-Fgreedy-CH* is slower is that it does not take into account the placement of the cached data, which leads to larger amounts (by 67%) of cache data to transfer. For other users' executions (when cached data exists), *D-CacheA* outperforms *D-Sgreedy-CH* in terms of execution time by 29%, and for the fourth user execution, by 20%. This is because *D-CacheA* better selects the cache site in order to reduce the execution time of the future re-executions. In addition, *D-CacheA* balances the cached data and computations. It outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of intermediate data transfer time (by 59% and 15%, respectively) and cache data transfer time (by 82% and 74%, respectively).

Overall, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by 61% and 43%, respectively. The workflow fragments are not necessarily scheduled to the site with shortest execution time, but to the site that minimizes overall total time. Considering the multiuser perspective, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH*, reducing the total time for each new user (up to 6% faster for the fourth user compared to the second).

Let us now consider Figure 5.7.b (*bCompute* method). For the first user execution, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by 36% and 10% respectively. *bCompute* stores the cache data on the site with the most idle CPUs, which is often the site with the most CPUs. This leads the cached data to be stored close to where it is generated, thus reducing data transfers when adding data to the cache. For the second user, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by 46% and 21% respectively. The cached data generated by the first user is stored on the sites with more available CPUs, which minimizes the intermediate and

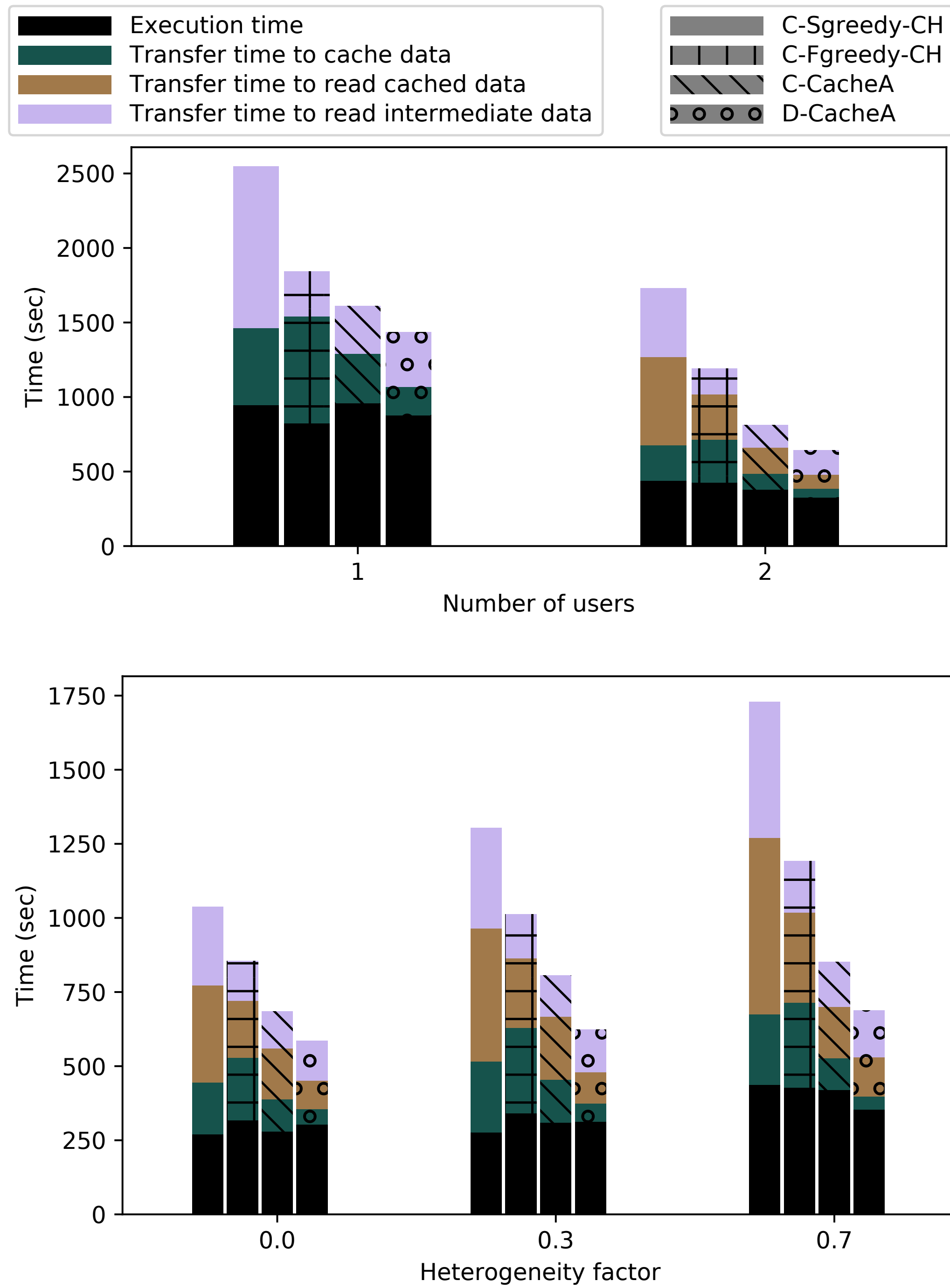
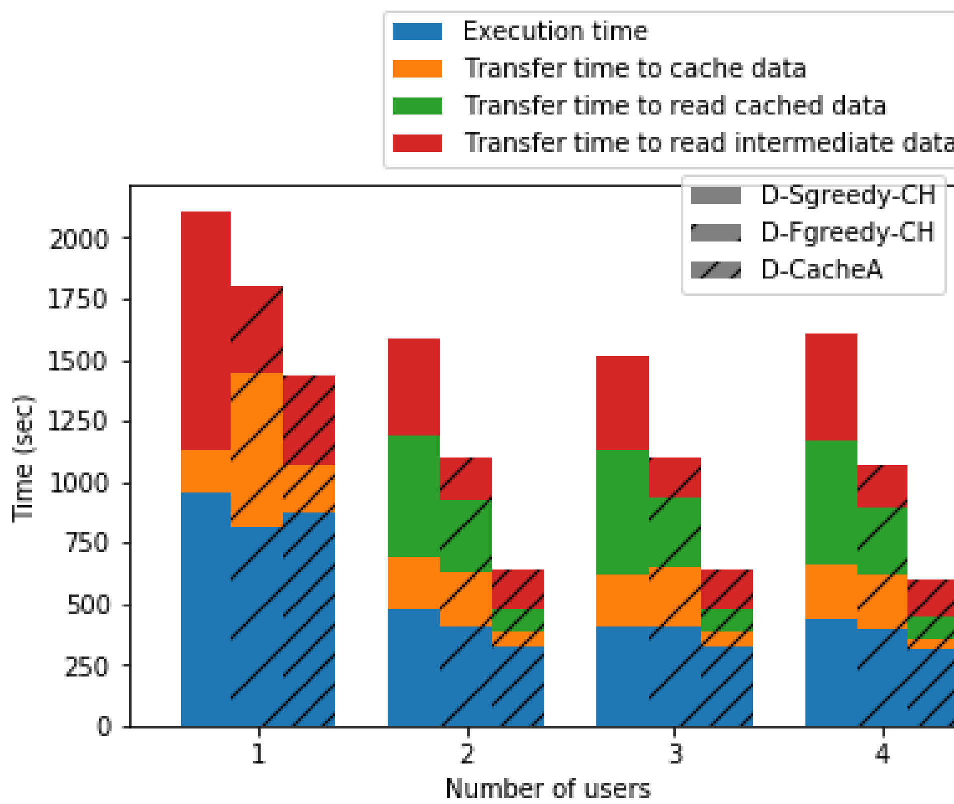
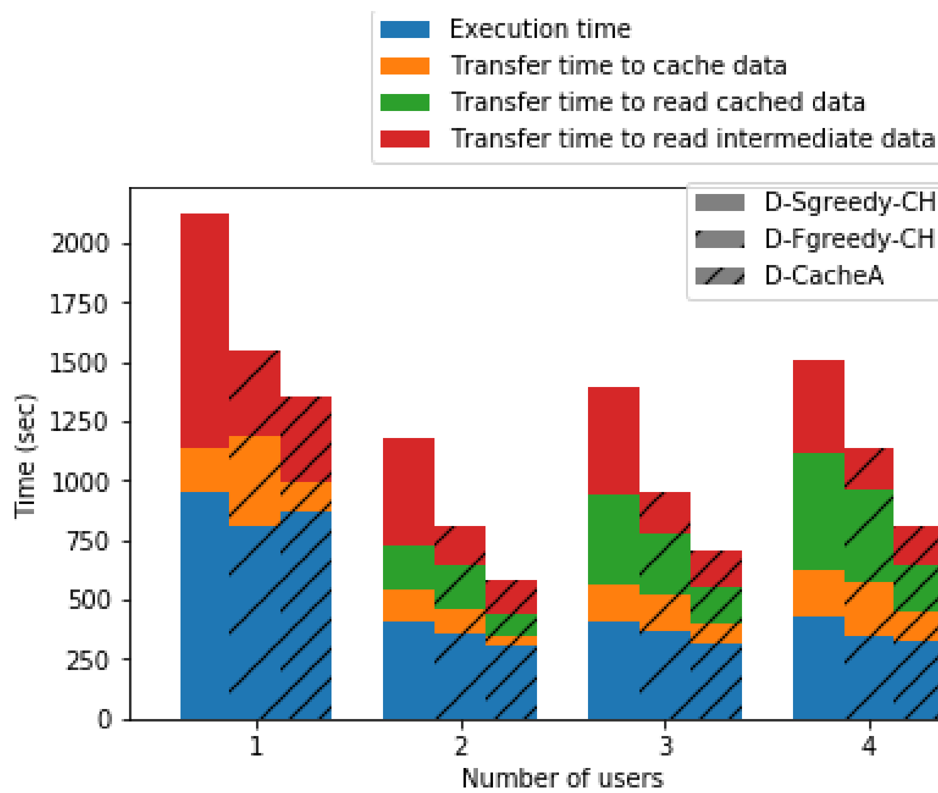


FIGURE 5.6: Centralized versus distributed cache in terms of execution time for three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*)



(A) *bStorage method*



(B) *bCompute method*

FIGURE 5.7: Total times for multiple users (60% of same raw data per user) for three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*)

reused cached data transfers. From the third user, the storage at some site gets full, *i.e.*, for the third user's execution, Site 3 storage is full and from the fourth user's execution, Site 2 storage is full. Thus, the performance of the three scheduling algorithms decreases due to higher cache data transfer time. Yet, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by 49% and 25% respectively.

Experiment 5: cloud site heterogeneity.

We now compare the three algorithms in the case of heterogeneous sites by considering the amount of data transferred and execution time. In this experiment (see Figure 5.8), we consider one user with the cache already provisioned by previous executions on 60% of the same raw data. We use the *bStorage* method for cache site selection.

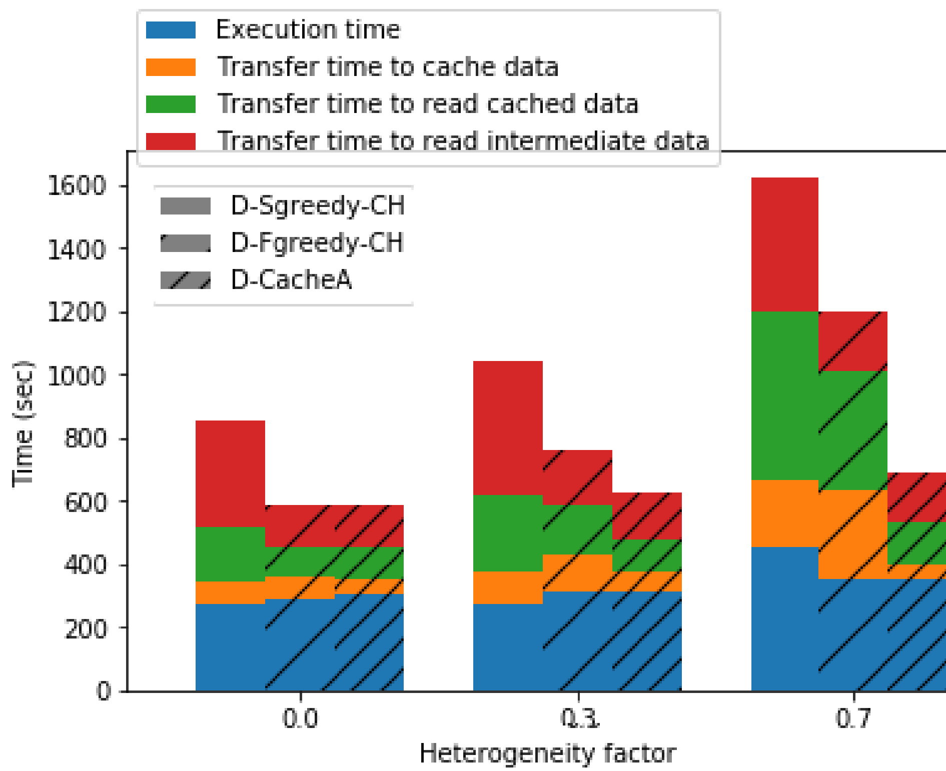
Figure 5.8 shows the execution times and the amount of data transferred using the three scheduling algorithms. With homogeneous sites ($H = 0$), the three algorithms have almost the same execution time. *D-CacheA* outperforms *D-Sgreedy-CH* in terms of amount of intermediate data transferred and total time by 44% and 26%, respectively. The execution time of *D-CacheA* is similar to *D-Fgreedy-CH* (3.1% longer). The cached data is balanced as the three sites have same storage capacities. Thus, total times of *D-CacheA* and *D-Fgreedy-CH* are almost the same.

With heterogeneous sites ($H > 0$), the sites with more CPUs have less available storage but can execute more tasks, which leads to a larger amount of intermediate and cached data being transferred between the sites. For $H = 0.3$, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time (by 40% and 18%, respectively) and amount of data transferred (by 47% and 21%, respectively).

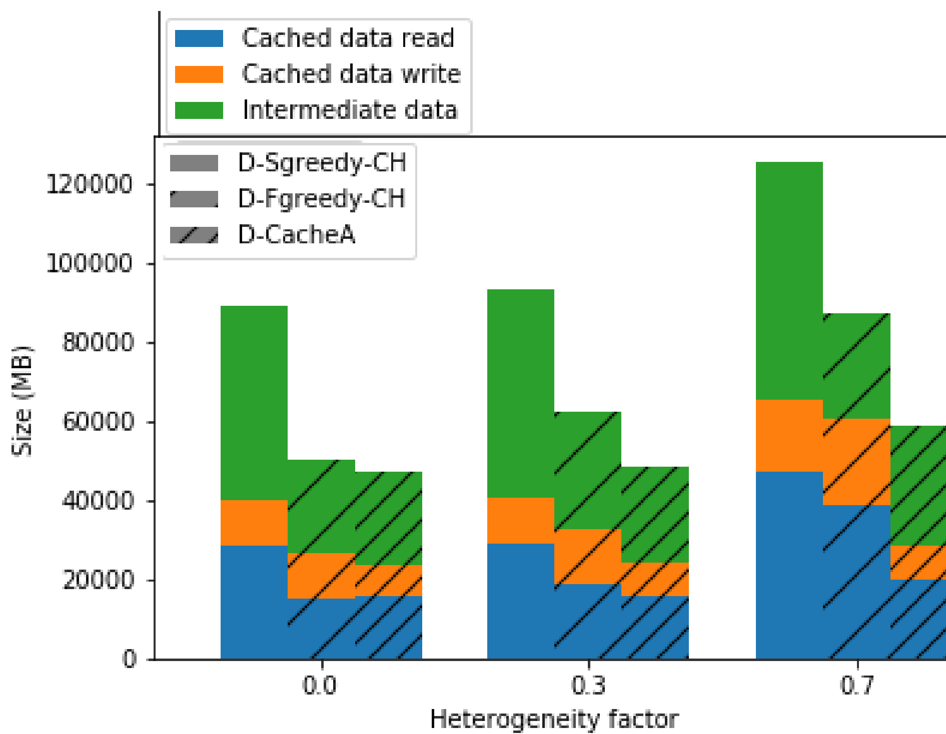
With $H = 0.7$, *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time (by 58% and 42%, respectively) and in terms of amount of data transferred (by 55% and 31%, respectively). *D-CacheA* is faster because its scheduling leads to a smaller amount of cached data transferred when reused (116% smaller than *D-Fgreedy-CH*) and added to the cache (47% smaller than *D-Fgreedy-CH*).

Experiment 6: adding and removing fragments.

In this experiment, we evaluate how our approach performs in terms of total time when subworkflows (with common fragments) derived from the Phenomenal workflow are executed independently. Figure 5.9 shows four subworkflows, each corresponding to a different analysis required by the user: WF1 performs image binarization, WF2 generates an analysis of the binary images, WF3 generates a 3D reconstruction of the plant and WF4 performs maize analysis. WF1 is mostly data-intensive, the image binarization fragment performing little computation but consuming Terabytes of data. WF2 requires more computational resources but is still mostly data-intensive. Fragment F3 in WF3 (composed of activities 3 and 4 in Figure 5.9) is mostly computation-intensive. Finally, WF4 is both data- and computation-intensive.



(A) Total time



(B) Amount of data transfer

FIGURE 5.8: Execution for one user (60% of same raw data used) on heterogeneous sites with three scheduling algorithms (*D-Sgreedy-CH*, *D-Fgreedy-CH* and *D-CacheA*)

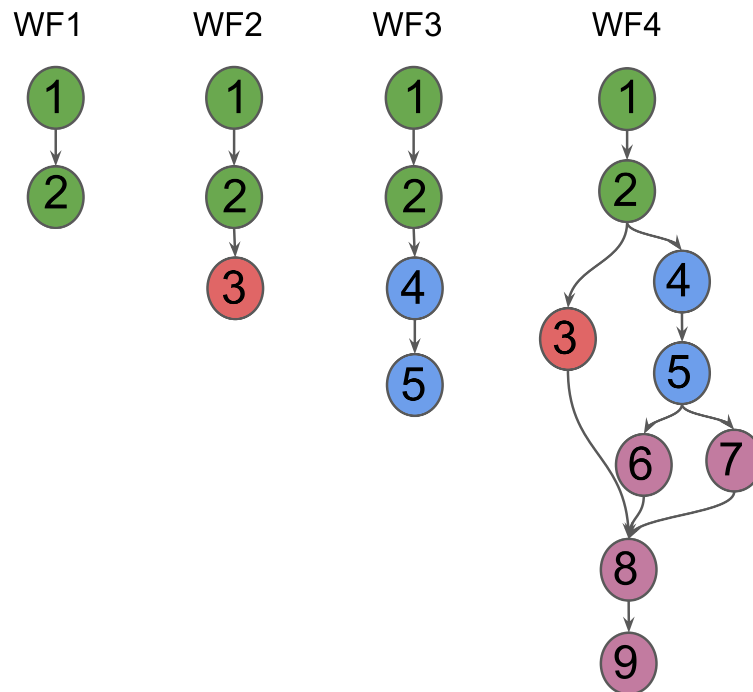
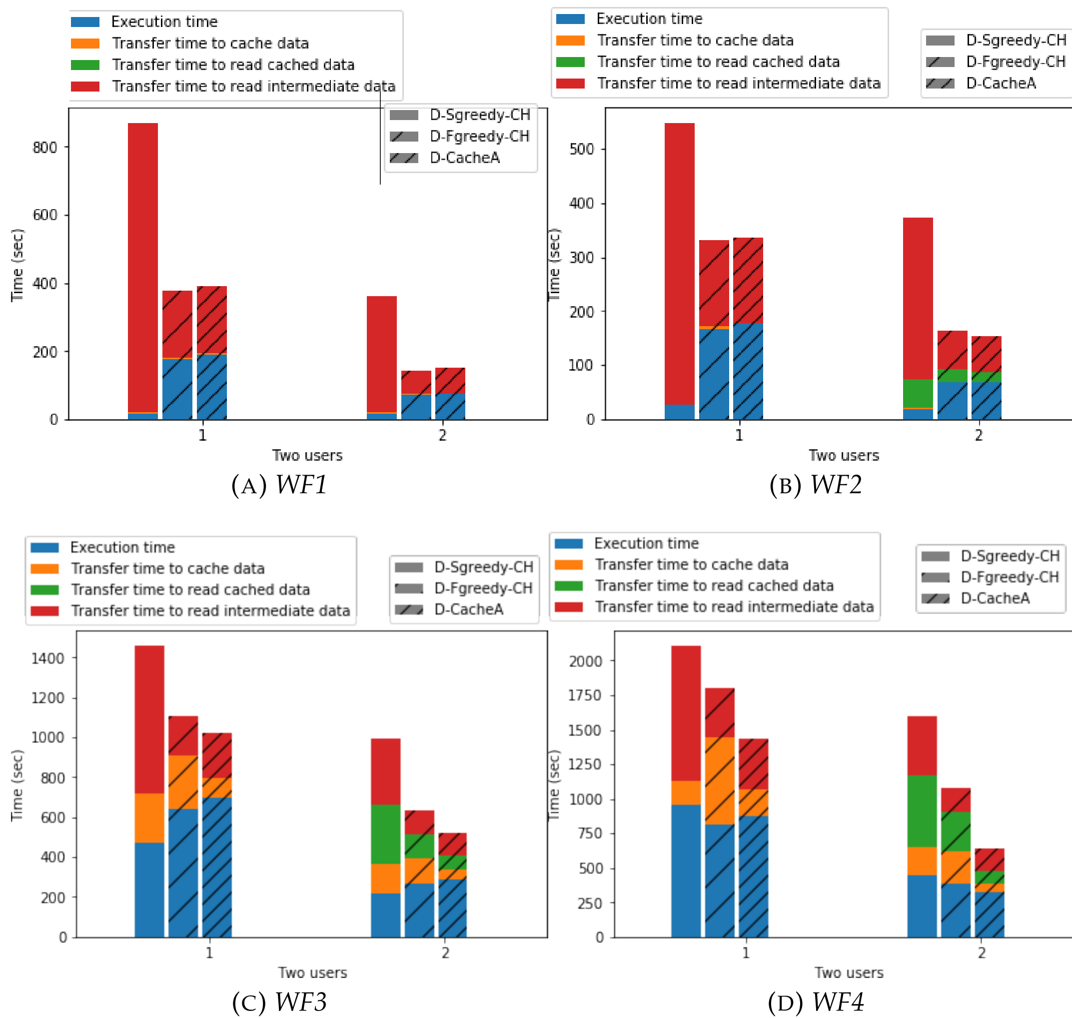


FIGURE 5.9: Four subworkflows derived from the Phenomenal workflow

The subworkflows are executed two times starting without cached data and 60% of the raw input data is common between the users. All executions use method *bStorage*.

Figure 5.10 shows the total times for executing WF1, WF2, WF3 and WF4 by two users, one after the other. The first user executes the subworkflow without existing cached data, then the second user executes the subworkflow using 60% of the same raw data. In the case of WF1 (see Figure 5.10a), *D-Sgreedy-CH* outperforms both *D-Fgreedy-CH* and *D-CacheA* in terms of execution times by 92% for the first user and by 83% asecond user. This is because*D-Sgreedy-CH* uses all CPUs at all sites, whereas *D-Fgreedy-CH* and *D-CacheA* almost only use the CPUs of Site 1 (where the raw input data is). However, *D-CacheA* transfers less intermediate data (70% less) during execution, which makes *D-CacheA* outperforming *D-Sgreedy* in terms of total time by 49%. *D-CacheA* and *D-Fgreedy-CH* have similar total times (*D-CacheA* is outperformed by only *D-Fgreedy-CH* by 2%). Since WF1 is mostly data-intensive, both methods *D-CacheA* and *D-Fgreedy-CH* try to execute the workflow at the site where the input data is.

In the case of WF2 (see Figure 5.10b), *D-Sgreedy-CH* also outperforms both *D-Fgreedy-CH* and *D-CacheA* in terms of execution times by 91% for the first user and by 79% for the second user. This is because the added fragment can be executed right after the execution of WF1 without delay as it does not required much computational resources. However, *D-CacheA* outperforms *D-Sgreedy-CH* in terms of total time by 39% due to longer data transfer times with *D-Sgreedy-CH*. *D-CacheA* and *D-Fgreedy-CH* also have similar total times,



(A) *WF1*

(B) *WF2*

(C) *WF3*

(D) *WF4*

FIGURE 5.10: Total times for executing the four subworkflows by two users (with 60% of same raw data for second user)

D-CacheA outperforms *D-Fgreedy-CH* by 4% for the second user. WF1 and WF2 are both data-intensive, not compute-intensive. Since the raw data is stored on Site 1, the site with the most storage capacities, it is the most likely to be used as cache site. For these subworkflows, the selection of the execution site by both algorithms *D-Fgreedy-CH* and *D-CacheA* depends mostly on the intermediate data location. In this case, they make similar decisions, and thus have similar performance.

In the case of WF3 (see Figure 5.10c), for the first user, *D-CacheA* outperforms both *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by 30% and 8% respectively. Then, for the second user, *D-CacheA* also outperforms both *D-Sgreedy-CH* and *D-Fgreedy-CH* by 47% and 18% respectively. This is due to *D-CacheA* outperforming *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of data transfers by 69% and 35% respectively. *D-Fgreedy-CH* selects the best sites that minimize execution times and intermediate data transfer times. In the case of WF3, which has a compute-intensive fragment, most of the computation will be scheduled on the site with the most computational resources. *D-CacheA*, however, will schedule some of the computation to the sites where the intermediate data will be cached and these sites may have less computational resources. This is why *D-Fgreedy-CH* has shorter execution time, but is outperformed by *D-CacheA* in terms of total time.

In the case of WF4 (see Figure 5.10d), *D-CacheA* outperforms both *D-Sgreedy-CH* and *D-Fgreedy-CH* by 32% and 24% for the first user and 60% and 40% for the second one respectively. In the case of WF4, the fragments are both data- and compute-intensive, thus the scheduling decision becomes more complex. *D-Fgreedy-CH* is scheduling fragments to minimize the execution and intermediate data transfer times, which leads *D-Fgreedy-CH* to outperform *D-CacheA* in terms of execution time and intermediate data transfer by 8%. However, the intermediate data that will be cached is bigger than in WF3, and the time to transfer the cached data becomes a major element of the total time. This is why *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time.

5.6.3 Discussion

The main result of this experimental evaluation is that *CacheA* always outperforms the two greedy algorithms *Sgreedy-CH* and *Fgreedy-CH*, both in the case of multiple users and heterogeneous sites.

The first experiment (with caching) shows that storing and reusing cached data becomes beneficial when 20% or more of the input data is reused. The second experiment (multiple users) shows that *D-CacheA* outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by up to 61% and 43%, respectively. It also shows that, with increasing numbers of users, the performance of the three scheduling algorithms decreases due to higher cache data transfer times. The third experiment (heterogeneous sites) shows that *D-CacheA* adapts well to site heterogeneity, minimizing the amount of cached data transferred and thus reducing total time. It outperforms *D-Sgreedy-CH* and *D-Fgreedy-CH* in terms of total time by up to 58% and 42% respectively.

Both cache site selection methods *bCompute* and *bStorage* have their own advantages. *bCompute* outperforms *bStorage* in terms of data transfer time by 13% for the first user and up to 17% for the second user. However, it does not scale with the number of users, and the limited storage capacities of Site 2 and 3 lead to a bottleneck. On the other hand, *bStorage* balances the cached data among sites and prevents the bottleneck when accessing the cached data, thus reducing re-execution times In summary, *bCompute* is best suited for compute-intensive workflows that generate smaller intermediate datasets while *bStorage* is best suited for data-intensive workflows where executions can be performed at the site where the data is stored.

5.7 Conclusion

In this chapter, we considered the efficient execution of data-intensive scientific workflows in multisite cloud, using caching of intermediate data produced by previous workflows. However, caching intermediate data and scheduling workflows to exploit such caching is complex, because of the heterogeneity of the cloud data centers. In particular, workflow scheduling must be cache-aware, in order to decide whether reusing cached data or re-executing workflows.

We proposed a solution for cache-aware scheduling of scientific workflows in multisite cloud. Our solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection and dynamic workflow scheduling. We implemented our solution in the OpenAlea workflow system and performed an extensive experimental evaluation in a three-site cloud with a real application in plant phenotyping (Phenomenal). We compared our solution with baseline algorithms which we extended to exploit our caching architecture. The experimental results show that our solution can yield major performance gains, reducing total time up to 42% with 60% of same input data for each new execution.

Chapter 6

Conclusion

In this thesis, we addressed the problem of distributed management of scientific workflows for high-throughput phenotyping (HTP) for plants, with the objective of reducing workflow execution time. To this end, we proposed new SWMS architectures with efficient caching and scheduling strategies to enable sharing of intermediate big data among users in both monosite and multisite cloud environments. The architectures and algorithms we proposed are different in monosite and multisite cloud. In monosite cloud, the SWMS includes a cache manager component that handles the caching and reusing of intermediate data. Before workflow execution, the cache manager simplifies the workflow according to the workflow topology and the existing cache data. During execution, the cache manager manages the decisions on which intermediate data will be cached using an adaptive algorithm. The algorithm uses a cost model to determine the best trade-off between cache data storage cost and re-execution cost. In multisite cloud, the SWMS includes both a cache manager and a scheduling algorithm that balances the workload and cache data between sites. The scheduling and cache management are based on a dynamic algorithm that minimizes the total time cost of workflow execution and cache data transfers. To evaluate our solution for cache management and workflow scheduling, we implemented our algorithms in OpenAlea. For the experiments, we used the Phenomenal workflow, a complex data-intensive phenotyping workflow, from the PhenoArch HTP platform.

We compared our algorithms with baseline algorithms in monosite and multisite cloud. Our results show that our approaches reduce the total time (execution time and data transfer time) compared with baseline algorithms. Moreover, our proposed architecture is able to successfully cache and share the intermediate data automatically between users.

In this chapter, we summarize and discuss the contributions made in this thesis. Then, we give some research directions for future work.

6.1 Contributions

A SWMS architecture to handle caching and cache-aware scheduling algorithms when executing workflows in either monosite or multisite cloud. The architecture is an extension of a state-of-the-art architecture, with new components for storing and reusing cached data during workflow execution. The architecture can be decomposed in two ways: i) in terms of functional

layers which shows the different functions and components; and ii) in terms of nodes and components which are involved in the processing of workflows. The two components added to the functional layers are a *cache manager* and a *cache index*. These two components are used by the SWMS during two steps of workflow execution: 1) workflow preprocessing, to remove all fragments of the workflow that do not need to be executed (producing a simplified workflow); and 2) cache provisioning, to decide at runtime which intermediate data should be cached. In monosite cloud, the *cache manager* and *cache index* focus on the decision to cache or not the intermediate data, based a trade-off between the storage costs and the data re-generation costs. In multisite cloud, the architecture is slightly different and enables both intra-site (inside a site) and inter-site (between the sites) decisions. At the inter-site level, the global scheduler and the cache manager uses data locality, the sites' different resources, and bandwidth for their decisions. At the intra-site level, the SWMS manages the scheduling and the execution of workflow fragments.

A cost model that includes both financial and time costs for both workflow execution and cache management. We tackled the problem of cache data selection, to determine which intermediate data should be cached during workflow execution. In monosite cloud, we designed a multi-objective cost model that includes financial costs and time costs. The cost model computes a trade-off between storage cost and the data regeneration cost. It includes the data writing times, data loading time, execution times, data sizes, and financial costs of storing and computing. In multisite cloud, in addition to cache data selection, we considered the problems of determining at which site the workflow fragments should be executed (execution site selection) and at which site the cached data should be stored (cache site selection). In multisite cloud, our cost model focuses on time costs only. Different from the monosite cloud where our model is designed for task granularity, we make the scheduling decision at workflow fragment granularity, which enables to reduce the scheduling decisions latency and the data transfers between sites. Our cost model is adapted for a heterogeneous multisite cloud, where the cost of fragment execution and cache storage may greatly vary between the sites. The cost model also includes data transfer times, workflow topology, site resources, and availability. We designed a cost function for each of the three decisions: cache data selection, execution site selection, and cache site selection. Since these decisions are not independent, we proposed a cost function to make a global decision, based on the cost components for individual decisions.

Cache-aware scheduling algorithms for monosite and multisite clouds. We addressed the problem of efficiently scheduling a workflow, taking into account cache management. To optimize workflow execution with cached data, we proposed two scheduling algorithms. The first algorithm is designed for monosite cloud and optimizes scheduling depending on the cache decisions made by the cache manager. It combines both workflow scheduling and cache decisions based on our cost model, and dynamically adapts to the tasks' variation in execution time and data sizes. The second algorithm is

for multisite cloud and is based on the global decision from our cost model. It takes the simplified workflow graph as input and, starting from the root fragment, computes the global decision for each fragment. Recall that the global decision combines individual decisions regarding cache data, cache site, and execution site, before scheduling each fragment. It optimizes the overall cost of the workflow execution with cached data. This algorithm is performed at the inter-site level, by the global scheduler. Then, at the intra-site level, the local scheduler manages the execution of the workflow fragments using the cache decisions made by the global scheduler.

OpenAlea in the cloud. OpenAlea, which is widely used by plant analysis scientists, was not adapted for cloud execution. To enable these scientists to benefit from our caching solutions without changing their SWMS, we developed an extension to OpenAlea which enables the execution of workflows in monosite and multisite cloud. The extension is based on the SWMS architectures proposed. It now considers persistent caching and reusing data when executing workflows. In monosite cloud, the extension manages the workflow scheduling over the cluster nodes at a site. The cached data is reused whenever possible and stored in the site nodes. In multisite cloud, the extension manages the workflow execution at two levels: inter-site level, and intra-site level. At the inter-site level, OpenAlea manages the partitioning, cache decisions, and scheduling of workflow fragments. At the intra-site level, OpenAlea manages the execution of workflow fragments and storage of new cached data in the site nodes.

An experimental validation on a data-intensive plant phenotyping application. To evaluate our solution for cache management and workflow scheduling, we implemented our algorithms in OpenAlea and performed experiments with the Phenomenal workflow, on real data. The Phenomenal workflow, which is both data- and compute- intensive, is representative of many workflows used for plant analyses as it is composed of various activities that: compress the data, expand the data, and require lots of memory and CPU. The input dataset used is composed of time series of images from the PhenoArch platform. For the cloud environment, we used the IFB-cloud with one site composed of one data node and two identical compute nodes. We compared three methods for cache provisioning: no cache, greedy, and adaptive (the one we propose). We evaluated the speedup, and the financial cost in two scenarios. First, we evaluated the speedup. Caching data adds an overhead to the execution time. For the first execution, the adaptive method only adds an overhead of 5,6% compared to the no cache method. While the greedy method adds 40% of overhead. For three executions starting without cache, the adaptive method is much faster than the other methods (about 2.5 and 1.5 times faster of 3 executions compared to the no cache method and the greedy method on 80 vCPUs). Second, we evaluated the financial gain in the scenario of multiple users that execute the workflow. The results show that adaptive caching can yield major performance gains, *e.g.*, up to a factor of 3.5 with 6 workflow re-executions. Third, we evaluated how the parameters

of our approach impact the re-execution time, cache size and monetary cost in the scenario where different workflows are executed independently but share activities. Our experimental evaluation shows that the adaptive method allows for caching only the relevant output data for subsequent re-executions by other users, without incurring a high storage cost for the cache.

In multisite cloud, both the input and cached data can be distributed in the cloud sites. For the cloud environment, we used the IFB-cloud with three sites in France: Montpelier, Lyon, and Lille. We compared our solution with four baselines. The first two baselines are: execution on multisite cloud without cache, and execution on monosite cloud with a centralized cache, on which we use a baseline multisite scheduling algorithm. Compared with a re-execution without caching, our algorithm outperforms the baseline from 20% of reused data. Below 20%, the overhead of caching outweighs its benefit. Compared with a centralized cache, our algorithm outperforms the baseline in terms of total time up to 60% in a heterogeneous multisite. Then, we compared our approach with two baselines multisite scheduling which we extended to exploit our caching architecture. The experimental results show that our solution can yield major performance gains, reducing total time up to 42% with 60% of the same input data for each new execution. Moreover, our proposed architecture is able to successfully cache and share the intermediate data automatically between users.

6.2 Directions for Future Work

HTP applications generate more and more data that tend to be geo-distributed. Experimental analyses performed on HTP datasets are resource-intensive, thus sharing and reusing intermediate data becomes essential. Our contributions represent an important step forward in experimental science such as biology, where scientists extend existing workflows with new methods or new parameters to test their hypotheses on datasets that have been previously analyzed. Yet, the problem of caching data for workflows in the cloud still raises important challenges. In this section, we propose some future research directions:

- **Machine learning for cache management.** We used a cost model to compute a trade-off between cache and re-execution costs to determine which data should be cached. This method efficiently decides for the tasks' data that are either worth caching or not at all. However, for many tasks whose behavior is unpredictable, the caching decision is not always precise. Machine learning (ML) provides techniques that learn from previous computations and cache decisions to generates new decisions that are more precise and refined. ML techniques propose solutions that take more information than our trade-off for caching decision, such as the workflow topology, and workflow metadata including provenance. These techniques also present good results in cache management [133, 152, 139]. Vietri *et al.* [152] propose an ML technique of

regret minimization to enhance cache replacement. Qin *et al.* [133] propose an internet-scale cache with prefetching, driven by data analytics and ML techniques, able to predict future requests and preemptively place data close to the users that are more likely to request it in the future. The method presented by Qin *et al.* could be used to better select the sites to cache the data, or to move cache data before workflow execution. Sadeghi *et al.* [139] propose an efficient caching policy that leverages deep reinforcement learning and is capable of learning-and-adapting to dynamic evolutions of file requests, and caching policies of cloud sites.

- **Hot and cold cache.** The cache manager proposed in this thesis only considers an on-disk cache. Some of the cache data is almost never reused but is still worth staying in the cache. On the other hand, some of the cached data is reused often by multiple users. Such "hot" data would benefit to be more accessible, in a "faster" cache, such as a memory cache. Several works in the database community have studied the benefit of hot and cold caching, that could be used with workflow caching [29, 23, 90]. For example, we could store the "hot" cache in main memory, and the "cold" cache in on-disk storage.

- **Cyclic workflows and algebraic operators.** The work presented in this thesis is adapted for acyclic workflows, which are the most common. Yet, in several experiments where plant analyses are mixed with simulations, workflows are cyclic or use algebraic operators [128], which brings new challenges with data caching. Indeed, in algebraic workflows the same activity will produce different output data depending on the state of the cycle, which makes it difficult to cache and reuse data.

- **Environmental cost model.** Our cost model takes into account two objectives: minimizing the makespan and the financial costs. Some other objectives, such as meeting deadline constraints, or following security constraints would change the decisions on scheduling and data caching. Moreover, the objective of minimizing environmental cost becomes essential and could be integrated with the cache decision. In an heterogeneous multisite cloud, each site has different environmental costs for both storage and computation. The environmental cost of writing and reading from a disk is usually much cheaper than the computation and the memory allocation costs. Yet, the in-memory cache can be extremely costly and the speed up gain from reusing data makes the trade-off between data regeneration cost and storage cost complex to estimate. Thus, the environmental cost could provide a different solution for workflow scheduling and cache data distribution which should be considered for greener experiments.

Bibliography

[1] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. "Testing cloud benchmark scalability with cassandra". In: *2014 IEEE World Congress on Services*. IEEE. 2014, pp. 434–441.

[2] Ian F Adams et al. "Maximizing Efficiency by Trading Storage for Computation." In: *HotCloud*. 2009.

[3] Michel E Adiba. "Derived relations: a unified mechanism for views, snapshots, and distributed data". In: *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*. 1981, pp. 293–305.

[4] Michel E Adiba and Bruce G Lindsay. "Database snapshots". In: *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*. 1980, pp. 86–91.

[5] Enis Afgan et al. "Galaxy CloudMan: delivering cloud compute clusters". In: *BMC bioinformatics*. Vol. 11. 12. BioMed Central. 2010, pp. 1–6.

[6] Robert W Allard. *Principles of plant breeding*. John Wiley & Sons, 1999.

[7] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. "Provenance collection support in the kepler scientific workflow system". In: *International Provenance and Annotation Workshop*. 2006, pp. 118–132.

[8] Ilkay Altintas et al. "Kepler: an extensible system for design and execution of scientific workflows". In: *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE. 2004, pp. 423–424.

[9] João Carlos de AR Gonçalves et al. "Using domain-specific data to enhance scientific workflow steering queries". In: *International Provenance and Annotation Workshop*. Springer. 2012, pp. 152–167.

[10] José Luis Araus and Jill E Cairns. "Field high-throughput phenotyping: the new crop breeding frontier". In: *Trends in plant science* 19.1 (2014), pp. 52–61.

[11] Konstantine Arkoudas et al. "Verifying a file system implementation". In: *International Conference on Formal Engineering Methods*. Springer. 2004, pp. 373–390.

[12] Simon Artzet et al. *OpenAlea.Phenomenal: A Workflow for Plant Phenotyping*. Sept. 2018. DOI: 10.5281/zenodo.1436634.

[13] Simon Artzet et al. "Phenomenal: An automatic open source library for 3D shoot architecture reconstruction and analysis for image-based plant phenotyping". In: (2020).

[14] Adam Barker and Jano Van Hemert. "Scientific workflow: a survey and research directions". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2007, pp. 746–753.

[15] Louis Bavoil et al. "Vistrails: Enabling interactive multiple-view visualizations". In: *VIS 05. IEEE Visualization, 2005.* IEEE. 2005, pp. 135–142.

[16] Ralph Bergmann and Yolanda Gil. "Retrieval of semantic workflows with knowledge intensive similarity measures". In: *International Conference on Case-Based Reasoning*. Springer. 2011, pp. 17–31.

[17] Sergey Blagodurov et al. "Multi-objective job placement in clusters". In: *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2015, pp. 1–12.

[18] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently updating materialized views". In: *ACM SIGMOD Record* 15.2 (1986), pp. 61–71.

[19] Alexandre G de Brevern et al. "Trends in IT innovation to build a next generation bioinformatics solution to manage and analyse biological big data produced by NGS technologies". In: *BioMed research international* 2015 (2015).

[20] Nicolas Brichet et al. "A robot-assisted imaging pipeline for tracking the growths of maize ear and silks in a high-throughput phenotyping platform". In: *Plant Methods* 13.1 (2017), pp. 1–12.

[21] Nadine Brisson et al. "Why are wheat yields stagnating in Europe? A comprehensive data analysis for France". In: *Field Crops Research* 119.1 (2010), pp. 201–212.

[22] Piotr Bryk et al. "Storage-aware algorithms for scheduling of workflow ensembles in clouds". In: *Journal of Grid Computing* 14.2 (2016), pp. 359–378.

[23] Omran Bukhres and Jin Jing. "Performance analysis of adaptive caching algorithms in mobile environments". In: *Information Sciences* 95.1-2 (1996), pp. 1–27.

[24] Marc Bux and Ulf Leser. "Parallelization in scientific workflow management systems". In: *arXiv preprint arXiv:1303.7195* (2013).

[25] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[26] Llorenç Cabrera-Bosquet et al. "High-throughput estimation of incident light, light interception and radiation-use efficiency of thousands of plants in a phenotyping platform". In: *New Phytologist* 212.1 (2016), pp. 269–281.

[27] Jacek Cała and Paolo Missier. "Provenance Annotation and Analysis to Support Process Re-computation". In: *International Provenance and Annotation Workshop*. Springer. 2018, pp. 3–15.

[28] Steven P Callahan et al. "VisTrails: visualization meets data management". In: *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*. 2006, pp. 745–747.

[29] Michael J Carey et al. "Data caching tradeoffs in client-server DBMS architectures". In: *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*. 1991, pp. 357–366.

[30] Artem Chebotko et al. "RDFProv: A relational RDF store for querying and managing scientific workflow provenance". In: *Data & Knowledge Engineering* 69.8 (2010), pp. 836–865.

[31] Tsu-Wei Chen et al. "Genetic and environmental dissection of biomass accumulation in multi-genotype maize canopies". In: *Journal of experimental botany* 70.9 (2019), pp. 2523–2534.

[32] Wanghu Chen et al. "Enhancing smart re-run of Kepler scientific workflows based on near optimum provenance caching in cloud". In: *IEEE World Congress on Services (SERVICES)*. 2014, pp. 378–384.

[33] Weiwei Chen and Ewa Deelman. "Partitioning and scheduling workflows across multiple sites with storage constraints". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2011, pp. 11–20.

[34] Weiwei Chen et al. "Balanced task clustering in scientific workflows". In: *2013 IEEE 9th International Conference on e-Science*. IEEE. 2013, pp. 188–195.

[35] Fernando Chirigati and Juliana Freire. *Provenance and Reproducibility*. 2018.

[36] Fernando Chirigati et al. "Evaluating parameter sweep workflows in high performance computing". In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. 2012, pp. 1–10.

[37] Jieun Choi, Theodora Adufu, and Yoonhee Kim. "Data-locality aware scientific workflow scheduling methods in HPC cloud environments". In: *International Journal of Parallel Programming* 45.5 (2017), pp. 1128–1141.

[38] Mosharaf Chowdhury et al. "Managing data transfers in computer clusters with orchestra". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 98–109.

[39] Philip Church, Andrzej Goscinski, and Christophe Lefèvre. "Exposing HPC and sequential applications as services through the development and deployment of a SaaS cloud". In: *Future Generation Computer Systems* 43 (2015), pp. 24–37.

[40] Sarah Cohen-Boulakia et al. "Distilling structure in Taverna scientific workflows: a refactoring approach". In: *BMC bioinformatics* 15.S1 (2014), S12.

[41] Flavio Costa et al. "Capturing and querying workflow runtime provenance with PROV: a practical approach". In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 2013, pp. 282–289.

[42] Steve Crago et al. "Heterogeneous cloud computing". In: *2011 IEEE International Conference on Cluster Computing*. IEEE. 2011, pp. 378–385.

[43] Daniel Crawl, Jianwu Wang, and Ilkay Altintas. "Provenance for mapreduce-based data-intensive workflows". In: *Proceedings of the 6th workshop on Workflows in support of large-scale science*. 2011, pp. 21–30.

[44] Daniel CM De Oliveira, Ji Liu, and Esther Pacitti. "Data-intensive workflow management: for clouds and data-intensive and scalable computing environments". In: *Synthesis Lectures on Data Management* 14.4 (2019), pp. 1–179.

[45] Ewa Deelman, Gideon Juve, and G Bruce Berriman. "Using Clouds for Science, is it just Kicking the Can down the Road?." In: *CLOSER*. 2012, pp. 127–134.

[46] Ewa Deelman et al. "Pegasus, a workflow management system for science automation". In: *Future Generation Computer Systems* 46 (2015), pp. 17–35.

[47] Ewa Deelman et al. "Pegasus: A framework for mapping complex scientific workflows onto distributed systems". In: *Scientific Programming* 13.3 (2005), pp. 219–237.

[48] Ewa Deelman et al. "The cost of doing science on the cloud: the montage example". In: *International Conference forHigh Performance Computing, Networking, Storage and Analysis*. 2008, pp. 1–12.

[49] Ewa Deelman et al. "Workflows and e-Science: An overview of workflow system features and capabilities". In: *Future generation computer systems* 25.5 (2009), pp. 528–540.

[50] Saumen C Dey et al. "UP & DOWN: Improving Provenance Precision by Combining Workflow-and Trace-Level Information". In: *USENIX Workshop on the Theory and Practice of Provenance (TAPP)*. 2014.

[51] Francisco Rodrigo Duro et al. "Exploiting data locality in Swift/T workflows using Hercules". In: *Proc. NESUS Workshop*. 2014.

[52] Iman Elghandour and Ashraf Aboulnaga. "ReStore: reusing results of MapReduce jobs". In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 586–597.

[53] Hamid Mohammadi Fard, Thomas Fahringer, and Radu Prodan. "Budget-constrained resource provisioning for scientific applications in clouds". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE. 2013, pp. 315–322.

[54] Juliana Freire et al. "Managing rapidly-evolving scientific workflows". In: *International Provenance and Annotation Workshop*. Springer. 2006, pp. 10–18.

[55] Luiz MR Gadelha Jr et al. "Provenance traces of the swift parallel scripting system". In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 2013, pp. 325–326.

[56] K Ganga and S Karthik. "A fault tolerent approach in scientific workflow systems based on cloud computing". In: *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*. IEEE. 2013, pp. 387–390.

[57] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[58] Daniel Garijo et al. "Common motifs in scientific workflows: An empirical analysis". In: *Future Generation Computer Systems (FGCS)* 36 (2014), pp. 338–351.

[59] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43.

[60] Yolanda Gil et al. "Wings for Pegasus: A Semantic Approach to Creating Very Large Scientific Workflows." In: *OWLED*. 2006.

[61] Jeremy Goecks et al. "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences". In: *Genome biology* 11.8 (2010), R86.

[62] Florie Gosseau et al. "Heliaphen, an outdoor high-throughput phenotyping platform for genetic studies and crop modeling". In: *Frontiers in plant science* 9 (2019), p. 1908.

[63] Tibor Gottdank. "Introduction to the ws-pgrade/guse science gateway framework". In: *Science Gateways for Distributed Computing Infrastructures*. Springer, 2014, pp. 19–32.

[64] Vinicius M Gottin et al. "Automatic Caching Decision for Scientific Dataflow Execution in Apache Spark". In: *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM. 2018, p. 2.

[65] Robert L Grossman et al. "Compute and storage clouds using wide area high performance networks". In: *Future Generation Computer Systems* 25.2 (2009), pp. 179–183.

[66] Ashish Gupta, Hosagrahar V Jagadish, and Inderpal Singh Mumick. "Data integration using self-maintainable views". In: *International Conference on Extending Database Technology*. Springer. 1996, pp. 140–144.

[67] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[68] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. "Maintaining views incrementally". In: *ACM SIGMOD Record* 22.2 (1993), pp. 157–166.

[69] Himanshu Gupta and Inderpal Singh Mumick. “Selection of views to materialize under a maintenance cost constraint”. In: *International Conference on Database Theory*. Springer. 1999, pp. 453–470.

[70] Mihael Hategan, Justin Wozniak, and Ketan Maheshwari. “Coasters: uniform resource provisioning and access for clouds and grids”. In: *2011 Fourth IEEE International Conference on Utility and Cloud Computing*. IEEE. 2011, pp. 114–121.

[71] Elliot L Heffner, Mark E Sorrells, and Jean-Luc Jannink. “Genomic selection for crop improvement”. In: *Crop Science* 49.1 (2009), pp. 1–12.

[72] Gaëtan Heidsieck et al. “Adaptive Caching for Data-Intensive Scientific Workflows in the Cloud”. In: *Int. Conf. on Database and Expert Systems Applications (DEXA)*. 2019, pp. 452–466.

[73] Gaëtan Heidsieck et al. “Distributed Caching of Scientific Workflows in Multisite Cloud”. In: *International Conference on Database and Expert Systems Applications*. Springer. 2020, pp. 51–65.

[74] Gaëtan Heidsieck et al. “Efficient Execution of Scientific Workflows in the Cloud Through Adaptive Caching”. In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLIV*. Springer, 2020, pp. 41–66.

[75] R Heidsieck et al. “Dual target x-ray tubes for mammographic examinations: Dose reduction with image quality equivalent to that with standard mammographic tubes”. In: *Radiology* 181 (1991), p. 311.

[76] D Hensgen et al. “Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems”. In: *Proc. Heterogeneous Computing Workshop*. 1999.

[77] Felipe Horta et al. “Using provenance to visualize data from large-scale experiments”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 1418–1419.

[78] Gregor Joeris and Otthein Herzog. “Managing evolving workflow specifications”. In: *Proceedings. 3rd IFCIS International Conference on Cooperative Information Systems (Cat. No. 98EX122)*. IEEE. 1998, pp. 310–319.

[79] Astrid Junker et al. “Optimizing experimental procedures for quantitative evaluation of crop plant performance in high throughput phenotyping systems”. In: *Frontiers in plant science* 5 (2015), p. 770.

[80] Gideon Juve and Ewa Deelman. “Scientific workflows in the cloud”. In: *Grids, clouds and virtualization*. Springer, 2011, pp. 71–91.

[81] Steve Kelling et al. “Data-intensive science: a new paradigm for biodiversity studies”. In: *BioScience* 59.7 (2009), pp. 613–620.

[82] Jihie Kim et al. “Provenance trails in the wings/pegasus system”. In: *Concurrency and Computation: Practice and Experience* 20.5 (2008), pp. 587–597.

[83] Norbert Kirchgessner et al. "The ETH field phenotyping platform FIP: a cable-suspended multi-sensor system". In: *Functional Plant Biology* 44.1 (2017), pp. 154–168.

[84] James E Koltes et al. "A vision for development and utilization of high-throughput phenotyping and big data analytics in livestock". In: *Frontiers in Genetics* 10 (2019).

[85] David Koop et al. "Bridging workflow and data provenance using strong links". In: *Int. Conf. on Scientific and Statistical Database Management (SSDBM)*. 2010, pp. 397–415.

[86] Prakashan Korambath et al. "Deploying kepler workflows as services on a cloud infrastructure for smart manufacturing". In: *Procedia Computer Science* 29 (2014), pp. 2254–2259.

[87] Yu-Kwong Kwok and Ishfaq Ahmad. "Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm". In: *Journal of Parallel and Distributed Computing* 47.1 (1997), pp. 58–77.

[88] Sebastien Lacube et al. "Distinct controls of leaf widening and elongation by light and evaporative demand in maize". In: *Plant, Cell & Environment* 40.9 (2017), pp. 2017–2028.

[89] JK Ladha et al. "How extensive are yield declines in long-term rice–wheat experiments in Asia?" In: *Field Crops Research* 81.2-3 (2003), pp. 159–180.

[90] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 26–37.

[91] Chunhyeok Lim et al. "Storing, reasoning, and querying OPM-compliant scientific workflow provenance using relational databases". In: *Future Generation Computer Systems* 27.6 (2011), pp. 781–789.

[92] Ji Liu. "Gestion multisite de workflows scientifiques dans le cloud". PhD thesis. Montpellier, 2016.

[93] Ji Liu et al. "A survey of data-intensive scientific workflow management". In: *Journal of Grid Computing* 13.4 (2015), pp. 457–493.

[94] Ji Liu et al. "Efficient Scheduling of Scientific Workflows using Hot Metadata in a Multisite Cloud". In: *IEEE Trans. on Knowledge and Data Engineering* (2018), pp. 1–20.

[95] Ji Liu et al. "Multi-objective scheduling of Scientific Workflows in multisite clouds". In: *Future Generation Computer Systems(FGCS)* 63 (2016), pp. 76–95.

[96] Ji Liu et al. "Scientific workflow partitioning in multisite cloud". In: *European Conf. on Parallel Processing (Euro-Par)*. 2014, pp. 105–116.

[97] Ji Liu et al. "Scientific Workflow Scheduling with Provenance Data in a Multisite Cloud". In: *Trans. on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)* 33 (2017), pp. 80–112.

[98] Yuan Luo and Beth Plale. "Hierarchical mapreduce programming model and scheduling algorithms". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE. 2012, pp. 769–774.

[99] Amanda H Lynch et al. "Influence of savanna fire on Australian monsoon season precipitation and circulation as simulated using a distributed computing environment". In: *Geophysical Research Letters* 34.20 (2007).

[100] Malawski Maciej. "Cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press*. 2012.

[101] Ketan Maheshwari et al. "Improving multisite workflow performance using model-based scheduling". In: *2014 43rd International Conference on Parallel Processing*. IEEE. 2014, pp. 131–140.

[102] Phillip Mates et al. "Crowdlabs: Social analysis and visualization for the sciences". In: *International conference on scientific and statistical database management*. Springer. 2011, pp. 555–564.

[103] Marta Mattoso et al. "Dynamic steering of HPC scientific workflows: A survey". In: *Future Generation Computer Systems* 46 (2015), pp. 100–113.

[104] Reyazul Rouf Mir et al. "High-throughput phenotyping for crop improvement in the genomics era". In: *Plant Science* 282 (2019), pp. 60–72.

[105] Paolo Missier and Jacek Cala. "Efficient Re-computation of Big Data Analytics Processes in the Presence of Changes: Computational Framework, Reference Architecture, and Applications". In: *2019 IEEE International Congress on Big Data (BigDataCongress)*. IEEE. 2019, pp. 24–34.

[106] L Moreau et al. "The PROV data model and abstract syntax notation". In: *W3C Working Draft.(Work in progress.)* (2011).

[107] Ashish Nagavaram et al. "A cloud-based dynamic workflow for mass spectrometry data analysis". In: *2011 IEEE Seventh International Conference on eScience*. IEEE. 2011, pp. 47–54.

[108] Pascal Neveu et al. "Dealing with multi-source and multi-scale information in plant phenomics: the ontology-driven Phenotyping Hybrid Information System". In: *New Phytologist* 221.1 (2019), pp. 588–601.

[109] Bogdan Nicolae et al. "BlobSeer: Next-generation data management for large scale infrastructures". In: *Journal of Parallel and distributed computing* 71.2 (2011), pp. 169–184.

[110] Eduardo Ogasawara et al. "An algebraic approach for data-centric scientific workflows". In: *Proc. of the VLDB Endowment (PVLDB)* 4.12 (2011), pp. 1328–1339.

[111] Eduardo Ogasawara et al. "Chiron: a parallel engine for algebraic scientific workflows". In: *Concurrency and Computation: Practice and Experience* 25.16 (2013), pp. 2327–2341.

[112] Tom Oinn et al. "Taverna: a tool for the composition and enactment of bioinformatics workflows". In: *Bioinformatics* 20.17 (2004), pp. 3045–3054.

[113] Daniel de Oliveira, Fernanda Araujo Baião, and Marta Mattoso. "Towards a taxonomy for cloud computing from an e-science perspective". In: *Cloud Computing. Computer Communications and Networks.* Springer, 2010, pp. 47–62.

[114] Daniel de Oliveira et al. "A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds". In: *Journal of grid Computing* 10.3 (2012), pp. 521–552.

[115] Daniel de Oliveira et al. "An adaptive parallel execution strategy for cloud-based scientific workflows". In: *Concurrency and Computation: Practice and Experience* 24.13 (2012), pp. 1531–1550.

[116] Daniel de Oliveira et al. "Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows". In: *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE. 2010, pp. 378–385.

[117] Christopher Olston et al. "Automatic Optimization of Parallel Dataflow Programs." In: *USENIX Annual Technical Conference*. 2008, pp. 267–273.

[118] Michal Owsiak et al. "Running simultaneous kepler sessions for the parallelization of parametric scans and optimization studies applied to complex workflows". In: *Journal of Computational Science* 20 (2017), pp. 103–111.

[119] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Vol. 2. Springer, 1999.

[120] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Fourth Edition*. Springer, 2020.

[121] Suraj Pandey et al. "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments". In: *2010 24th IEEE international conference on advanced information networking and applications*. IEEE. 2010, pp. 400–407.

[122] Evangelia A Papoutsoglou et al. "Enabling reusability of plant phenomic datasets with MIAPPE 1.1". In: *New Phytologist* 227.1 (2020), pp. 260–273.

[123] Laura S Peirone et al. "Assessing the efficiency of phenotyping early traits in a greenhouse automated platform for predicting drought tolerance of soybean in the field". In: *Frontiers in plant science* 9 (2018), p. 587.

[124] James Perry et al. "QCDGrid: A grid resource for Quantum Chromodynamics". In: *Journal of Grid Computing* 3.1-2 (2005), p. 113.

[125] Ioan Petri et al. "Coordinating multi-site construction projects using federated clouds". In: *Automation in Construction* 83 (2017), pp. 273–284.

[126] Luis Pineda-Morales, Alexandru Costan, and Gabriel Antoniu. "Towards multi-site metadata management for geographically distributed cloud workflows". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2015, pp. 294–303.

[127] Luis Pineda-Morales et al. "Managing hot metadata for scientific workflows on multisite clouds". In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 390–397.

[128] Christophe Pradal et al. "InfraPhenoGrid: a scientific workflow infrastructure for plant phenomics on the grid". In: *Future Generation Computer Systems (FGCS)* 67 (2017), pp. 341–353.

[129] Christophe Pradal et al. "OpenAlea: a visual programming and component-based software platform for plant modelling". In: *Functional plant biology* 35.10 (2008), pp. 751–760.

[130] Christophe Pradal et al. "OpenAlea: scientific workflows combining data analysis and simulation". In: *Int. Conf. on Scientific and Statistical Database Management (SSDBM)*. 2015, 11:1–11:6.

[131] Kenneth W Preslan et al. "A 64-bit, shared disk file system for Linux". In: *16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (Cat. No. 99CB37098)*. IEEE. 1999, pp. 22–41.

[132] Rawaa Qasha et al. "Sharing and performance optimization of reproducible workflows in the cloud". In: *Future Generation Computer Systems* 98 (2019), pp. 487–502.

[133] Yubo Qin et al. "Towards a smart, internet-scale cache service for data intensive scientific applications". In: *Proceedings of the 10th Workshop on Scientific Cloud Computing*. 2019, pp. 11–18.

[134] Arcot Rajasekar et al. "iRODS primer: integrated rule-oriented data system". In: *Synthesis Lectures on Information Concepts, Retrieval, and Services* 2.1 (2010), pp. 1–143.

[135] Deepak K Ray et al. "Recent patterns of crop yield growth and stagnation". In: *Nature communications* 3.1 (2012), pp. 1–7.

[136] Maria Alejandra Rodriguez and Rajkumar Buyya. "A taxonomy and survey on scheduling algorithms for scientific workflows in IaaS cloud computing environments". In: *Concurrency and Computation: Practice and Experience* 29.8 (2017), e4041.

[137] Maria Alejandra Rodriguez and Rajkumar Buyya. "Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds". In: *IEEE transactions on cloud computing* 2.2 (2014), pp. 222–235.

[138] Thomas Roitsch et al. "New sensors and data-driven approaches—A path to next generation phenomics". In: *Plant Science* 282 (2019), pp. 2–10.

[139] Alireza Sadeghi, Gang Wang, and Georgios B Giannakis. "Adaptive Caching via Deep Reinforcement Learning." In: (2019).

[140] Russel Sandberg et al. "Innovations in internetworking. chapter Design and Implementation of the Sun Network Filesystem". In: *Artech House, Inc., Norwood, MA, USA* 2 (1988), pp. 3–3.

[141] Frank B Schmuck and Roger L Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *FAST*. Vol. 2. 19. 2002.

[142] Peter Selby et al. "BrAPI—an application programming interface for plant breeding applications". In: *Bioinformatics* 35.20 (2019), pp. 4147–4155.

[143] Srinath Shankar and David J DeWitt. "Data driven workflow planning in cluster management systems". In: *Proceedings of the 16th international symposium on High performance distributed computing*. 2007, pp. 127–136.

[144] Anna Sher et al. "Incorporating Local Ca 2+ Dynamics into Single Cell Ventricular Models". In: *International Conference on Computational Science*. Springer. 2008, pp. 66–75.

[145] Hervé Sinoquet, Pierre Rivet, Christophe Godin, et al. "Assessment of the three-dimensional architecture of walnut trees using digitising." In: (1997).

[146] Wibke Sudholt et al. "Parameter scan of an effective group difference pseudopotential using grid computing". In: *New Generation Computing* 22.2 (2004), pp. 137–146.

[147] François Tardieu et al. "Plant phenomics, from sensors to knowledge". In: *Current Biology* 27.15 (2017), R770–R783.

[148] Gabor Terstyanszky et al. "Enabling scientific workflow sharing through coarse-grained interoperability". In: *Future Generation Computer Systems* 37 (2014), pp. 46–59.

[149] Alexander Thomson and Daniel J Abadi. "CalvinFS: Consistent {WAN} Replication and Scalable Metadata Management for Distributed File Systems". In: *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 2015, pp. 1–14.

[150] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[151] Radu Tudoran et al. "Tomusblobs: Towards communication-efficient storage for mapreduce applications in azure". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE. 2012, pp. 427–434.

[152] Giuseppe Vietri et al. "Driving cache replacement with ml-based lecar". In: *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. 2018.

[153] Nicolas Virlet et al. “Field Scanalyzer: An automated robotic field phenotyping platform for detailed crop monitoring”. In: *Functional Plant Biology* 44.1 (2017), pp. 143–153.

[154] Jens-Sönke Vöckler et al. “Experiences using cloud computing for a scientific workflow application”. In: *Proceedings of the 2nd international workshop on Scientific cloud computing*. 2011, pp. 15–24.

[155] Ashish Vulimiri et al. “WANalytics: Analytics for a Geo-Distributed Data-Intensive World.” In: *CIDR*. 2015.

[156] Jianwu Wang and Ilkay Altintas. “Early cloud experiences with the kepler scientific workflow system”. In: *Procedia Computer Science* 9 (2012), pp. 1630–1634.

[157] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. “Kepler+ Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems”. In: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. 2009, pp. 1–8.

[158] Xu Wang et al. “Field-based high-throughput phenotyping of plant height in sorghum using different sensing technologies”. In: *Plant Methods* 14.1 (2018), p. 53.

[159] Polyane Wercelens et al. “Bioinformatics Workflows With NoSQL Database in Cloud Computing”. In: *Evolutionary Bioinformatics* 15 (2019), p. 1176934319889974.

[160] Michael Wilde et al. “Swift: A language for distributed parallel scripting”. In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[161] Dean N Williams et al. “The Earth System Grid: Enabling access to multimodel climate simulation data”. In: *Bulletin of the American Meteorological Society* 90.2 (2009), pp. 195–206.

[162] Katherine Wolstencroft et al. “The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud”. In: *Nucleic acids research* 41.W1 (2013), W557–W561.

[163] Justin M Wozniak et al. “Swift/t: Large-scale application composition via distributed-memory dataflow processing”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 95–102.

[164] Fuhui Wu, Qingbo Wu, and Yusong Tan. “Workflow scheduling in cloud: a survey”. In: *The Journal of Supercomputing* 71.9 (2015), pp. 3373–3418.

[165] Ustun Yildiz, Adnene Guabtni, and Anne HH Ngu. “Business versus scientific workflows: A comparative study”. In: *2009 Congress on Services-I*. IEEE. 2009, pp. 340–343.

[166] Jia Yu and Rajkumar Buyya. “A taxonomy of workflow management systems for grid computing”. In: *Journal of Grid Computing* 3.3-4 (2005), pp. 171–200.

[167] Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. "Workflow scheduling algorithms for grid computing". In: *Metaheuristics for scheduling in distributed computing environments*. Springer, 2008, pp. 173–214.

[168] Zhifeng Yu and Weisong Shi. "An adaptive rescheduling strategy for grid workflow applications". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8.

[169] Dong Yuan et al. "A cost-effective strategy for intermediate data storage in scientific cloud workflow systems". In: *IEEE Int. Symp. on Parallel & Distributed Processing (IPDPS)*. 2010, pp. 1–12.

[170] Dong Yuan et al. "A data placement strategy in scientific cloud workflows". In: *Future Generation Computer Systems* 26.8 (2010), pp. 1200–1214.

[171] Dong Yuan et al. "A highly practical approach toward achieving minimum data sets storage cost in the cloud". In: *IEEE Trans. on Parallel and Distributed Systems* 24.6 (2013), pp. 1234–1244.

[172] Jia Zhang, Daniel Kuc, and Shiyong Lu. "Confucius: A tool supporting collaborative scientific workflow composition". In: *IEEE Transactions on Services Computing* 7.1 (2012), pp. 2–17.

[173] Jia Zhang et al. "Bridging vistrails scientific workflow management system to high performance computing". In: *2013 IEEE Ninth World Congress on Services*. IEEE. 2013, pp. 29–36.

[174] Jinghui Zhang, Junzhou Luo, and Fang Dong. "Scheduling of scientific workflow in non-dedicated heterogeneous multicluster platform". In: *Journal of Systems and Software* 86.7 (2013), pp. 1806–1818.

[175] Jinghui Zhang et al. "Towards optimized scheduling for data-intensive scientific workflow in multiple datacenter environment". In: *Concurrency and Computation: Practice and Experience* 27.18 (2015), pp. 5606–5622.

[176] Kaimin Zhang et al. "Smart caching for web browsers". In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 491–500.

[177] Shubin Zhang et al. "Accelerating MapReduce with distributed memory cache". In: *2009 15th International Conference on Parallel and Distributed Systems*. IEEE. 2009, pp. 472–478.

[178] Yong Zhao et al. "Swift: Fast, reliable, loosely coupled parallel computation". In: *2007 IEEE Congress on Services (Services 2007)*. IEEE. 2007, pp. 199–206.