Performance optimization of big data computing workflows for batch and stream data processing in multi-clouds

Huiyan Cao
New Jersey Institute of Technology

ABSTRACT

PERFORMANCE OPTIMIZATION OF BIG DATA COMPUTING WORKFLOWS FOR BATCH AND STREAM DATA PROCESSING IN MULTI-CLOUDS

by
Huiyan Cao

Workflow techniques have been widely used as a major computing solution in many science domains. With the rapid deployment of cloud infrastructures around the globe and the economic benefits of cloud-based computing and storage services, an increasing number of scientific workflows have migrated or are in active transition to clouds. As the scale of scientific applications continues to grow, it is now common to deploy various data- and network-intensive computing workflows such as serial computing workflows, MapReduce/Spark-based workflows, and Storm-based stream data processing workflows in multi-cloud environments, where inter-cloud data transfer oftentimes plays a significant role in both workflow performance and financial cost. Rigorous mathematical models are constructed to analyze the intra- and inter-cloud execution process of scientific workflows and a class of budget-constrained workflow mapping problems are formulated to optimize the network performance of big data workflows in multi-cloud environments. Research shows that these problems are all NP-complete and a heuristic solution is designed for each that takes into consideration module execution, data transfer, and I/O operations. The performance superiority of the proposed solutions over existing methods are illustrated through extensive simulations and further verified by real-life workflow experiments deployed in public clouds.

PERFORMANCE OPTIMIZATION OF BIG DATA COMPUTING WORKFLOWS FOR BATCH AND STREAM DATA PROCESSING IN MULTI-CLOUDS

by
Huiyan Cao

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

December 2020

APPROVAL PAGE

PERFORMANCE OPTIMIZATION OF BIG DATA COMPUTING WORKFLOWS FOR BATCH AND STREAM DATA PROCESSING IN MULTI-CLOUDS

Huiyan Cao

Dr. Chase Qishi Wu, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. Cristian M. Borcea, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Yi Chen, Committee Member — Date
Professor of Martin Tuchman School of Management, NJIT

Dr. Xiaoning Ding, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Senjuti Basu Roy, Committee Member — Date
Assistant Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Huiyan Cao

Degree: Doctor of Philosophy

Date: December 2020

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, USA, 2020
- Master of Information, Production and Systems Engineering,
  Waseda University, Tokyo, Japan, 2014
- Bachelor of Software Engineering,
  Xidian University, Xi'an, China, 2009

Major: Computer Science

Presentations and Publications:

Huiyan Cao, Chase Q. Wu, Liang Bao, Aiqin Hou and Wei Shen, "Throughput Optimization for Storm-based Processing of Stream Data on Clouds," *Future Generation Computer Systems*, Volume 112, Pages 567-579, November 2020.

Huiyan Cao and Chase Q. Wu, "Performance Optimization of Budget-Constrained MapReduce Workflows in Multi-Clouds," *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Washington, D.C., USA, May 1-4, 2018.

Huiyan Cao, Chase Q. Wu, Liang Bao, Aiqin Hou and Wei Shen, "Throughput Optimization for Storm-based Processing of Stream Data on Clouds," *Proceedings of the Workshop on Workflows in Support of Large-Scale Science in conjunction with Supercomputing*, Accepted for Lightning Talk, Dallas, TX, USA, November 11, 2018.

Chase Q. Wu and Huiyan Cao, "Optimizing the performance of big data workflows in multi-cloud environments under budget constraint," *Proceedings of the 13th IEEE International Conference on Services Computing*, San Francisco, USA, June 27 - July 2, 2016.

Qianwen Ye, Chase Q. Wu, Huiyan Cao, Nageswara S.V. Rao, and Aiqin Hou, "Storage-aware Task Scheduling for Performance Optimization of Big Data Workflows," *Proceedings of the 8th IEEE International Conference on Big Data and Cloud Computing*, Melbourne, Australia, December 11-13, 2018.

Meng Wang, Chase Q. Wu, Huiyan Cao, Yang Liu, Yongqiang Wang, and Aiqin Hou, "On MapReduce Scheduling in Hadoop Yarn on Heterogeneous Clusters," *Proceedings of the 12th IEEE International Conference on Big Data Science and Engineering*, New York City, USA, July 31 August 3, 2018.

Tao Wang, Chase Q. Wu, Yongqiang Wang, Aiqin Hou, and Huiyan Cao, "Multi-Path Routing for Maximum Bandwidth with K Edge-Disjoint Paths," *Proceedings of the 14th International Wireless Communications and Mobile Computing Conference*, Limassol, Cyprus, June 25-29, 2018.

Huiyan Cao and Chase Q. Wu, "Performance Optimization of Budget-Constrained Scientific Workflows in Multi-Cloud Environments," *IEEE Access*, under review, 2020.

I dedicate my dissertation work to my parents, Lianjun Yu and Huamin Cao.

ACKNOWLEDGMENT

I am very grateful to Prof. Chase Wu for his guidance.

I'd like to thank Prof. Cristian M. Borcea, Prof. Yi Chen, Prof. Xiaoning Ding and Prof. Senjuti Basu Roy for serving on my dissertation committee.

Also, I want to thank my parents and friends for their love and support.

Last, but not the least, I'd like to thank department of Computer Science at NJIT for the teaching assistantship and U.S. NSF for the research funding under Grant No. CNS-1560698 and No. CNS-1828123 with NJIT.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Next-generation scientific applications are producing colossal amounts of data, now frequently termed as "big data", on the order of terabytes nowadays and petabytes in the predictable future, which must be processed and analyzed in a timely manner for knowledge discovery and scientific innovation. In many of these scientific applications, computation and computing tasks for data generation, processing, and analysis are often assembled and constructed as workflows comprised of interdependent computing modules[1]. To some degree, workflows have become an indispensable enabling technology to meet various science missions in a wide spectrum of domains. In [41], stochastic optimization algorithms were utilized and Li *et al.* proposed a parameter tuning workflow which consists of two types of repeated tests. One type is multiple independent tests for a single event and the other involves tests of multiple events.

With the rapid deployment of cloud infrastructures around the globe and the economic benefit of cloud-based computing and storage services, an increasing number of scientific workflows have been shifted or are in active transition to clouds. As we enter the "big data" era, several efforts have been made to develop workflow engines for Hadoop ecosystem running on cloud platforms with virtual resources [20, 21, 22, 23, 24]. Due to the rapidly expanding scale of scientific workflows, it is now common to deploy data- and network-intensive computing workflows in multiple cloud sites. Real-life examples include a workflow deployment

[1]We refer to the smallest computing entity in a scientific workflow as a computing module, which represents either a serial computing task or a parallel processing job such as a typical MapReduce program in Hadoop.

across multiple Availability Zones within a certain Region in Amazon EC2, which are geographically distributed but connected through low-latency links [18]. Typically, within the same cloud, virtual machines (VMs) are organized into a virtual cluster and intra-cloud data transfer is performed through a shared storage system or network file system (NFS) without financial charge [1] [33]; while inter-cloud data transfer may constitute a significant part of both the execution time and the financial cost of a big data workflow due to the sheer volume of data being processed.

The current cloud services are categorized into: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS clouds provide virtualized hardware resources for users to deploy their own systems and applications, and therefore are most suitable for executing scientific workflows developed on various programming platforms. For example, Amazon EC2 provides VM instances with different CPU, memory, disk, and bandwidth capacities to meet the varying resource demands of different applications [18]. Such VM instances are usually priced according to their processing power and storage space but not necessarily in a linear manner [43], and charged by the provisioned time units, such as hours. Note that any partial hours are often rounded up in the cost evaluation of workflow execution as in the case of EC2 [37], but such rounding may be negligible in big data applications with a long workflow execution time. In many real-life applications, budget constraint is a major factor that affects the deployment of data-intensive workflows.

As we enter the big data era, several efforts have been made to develop workflow engines for Hadoop ecosystem in clouds with virtual resources. We expand our work to Hadoop environments.

Recently, there is an increasing need to process and analyze datasets as they are generated and transferred in real time for various purposes such as stock prediction, malfunction detection, social network analysis, and log data processing. To meet such demands, a wide range of computing engines have been developed and deployed for streaming data processing, including Apache Storm [24], Apache Flink [2], Apache Spark (Spark Streaming) [3], Apache Samza [4], Apache Apex [5], and Google Cloud Dataflow [6]. For example, Yahoo adopted Apache Storm to replace the internally developed S4 platform [46]; JStorm [7], now being merged into Apache Storm, and Heron [8] are heavily used by Alibaba Inc. and Twitter Inc., respectively; Spark Streaming and Flink are also gaining a widespread adoption in industry. In fact, real-time streaming data processing systems have become an indispensable building block in the entire big data ecosystem. As one of the most commonly used systems for streaming data processing, Apache Storm provides a workflow-based mechanism to execute directed acyclic graph (DAG)-structured topologies[2]. In recent years, we have witnessed a rapid deployment of cloud infrastructures around the globe and great economic benefits brought by cloud-based computing and storage services. As a result, many such Storm workflows have been shifted or are in active transition to cloud environments. As most public clouds adopt a pay-as-you-go service model, one additional constraint on financial budget must be considered in addition to traditional performance optimization goals. However, efforts in improving the performance of streaming data processing in clouds are still very limited.

To summarize, in this work, we focus on improving performance of big data computing workflows for batch and stream data processing in multi-cloud environment. We have worked on (1) serial-computing workflows, (2) MapReduce workflows and (3) Storm-based streaming processing workflows. The framework of the dissertation research consisting of various technical components is illustrated in Figure 1.1.

[2]The workflow structure in Storm is referred to as a topology, and hereafter, these two terms are used interchangeably.

For serial computing workflows, considering two scenarios: 1) mapping using dedicated VM instances, and 2) mapping reusing VM instances. In Chapter 4, we solved scenario 1) and analyzed both the time cost and the financial cost of intra- and inter-cloud execution of big data scientific workflows and formulate a Budget-Constrained workflow mapping problem for Minimum End-to-end Delay in IaaS Multi-Cloud environments, referred to as BCMED-MC. We show BCMED-MC to be NP-complete and design a heuristic solution that considers cloud service types and wide-area data transfer cost in the provisioning of VM instances and the selection of cloud sites. In Chaper 5, we tackle BCMED-MC problem for scenario 2). We designed a new heuristic solution that considers VM reuse.

As we enter the big data era, several efforts have been made to develop workflow engines for Hadoop ecosystem in clouds with virtual resources. We expanded our work to Hadoop environments in Chapter 6, we dived into the execution dynamics of MapReduce-based scientific workflows and formulated a budget-constrained workflow mapping problem for minimum makespan in IaaS multi-cloud environments, referred to as MinMRW-MC. We show MinMRW-MC to be NP-complete and design a heuristic solution.

Also, streaming data processing has become increasingly important due to its impacts on a wide range of use cases, such as real-time trading analytics, malfunction detection, campaign, social network, log processing and metrics analytics. To meet the demands of streaming data processing, many new computing engines have emerged, including Apache Storm and Apache Spark (Spark Streaming). In Chapter 7, we analyze both the time and financial cost of Storm-based workflow execution and formulate a Storm Topology Mapping problem for maximum throughput in clouds under Budget Constraint, referred to as STM-BC. We show

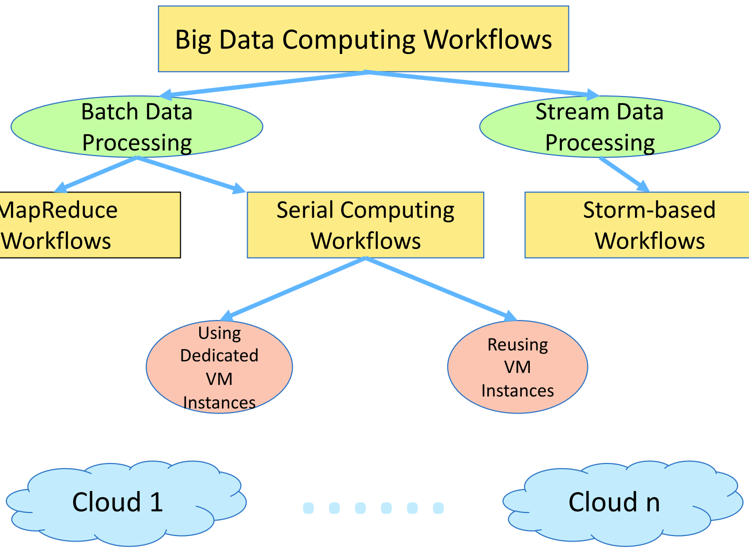


Figure 1.1 The framework of the dissertation research.

STM-BC to be NP-complete and design a heuristic solution that takes into consideration the parallelism of each task (spout/bolt) in the topology.

The performance superiority of all proposed solutions over existing methods are illustrated through extensive simulations and further verified by real-life workflow experiments deployed in public clouds.

Our workflow mapping solutions offer IaaS providers an economical resource allocation scheme to meet the budget constraint specified by the user, and meanwhile also serve as a cloud resource provisioning reference for scientific users to make proactive and informative resource requests. When try to locate the cooresponding chapter in this dissertation for the suitable workflow optimization solution of different types, please refer to the roadmap shown in Figure 1.2.

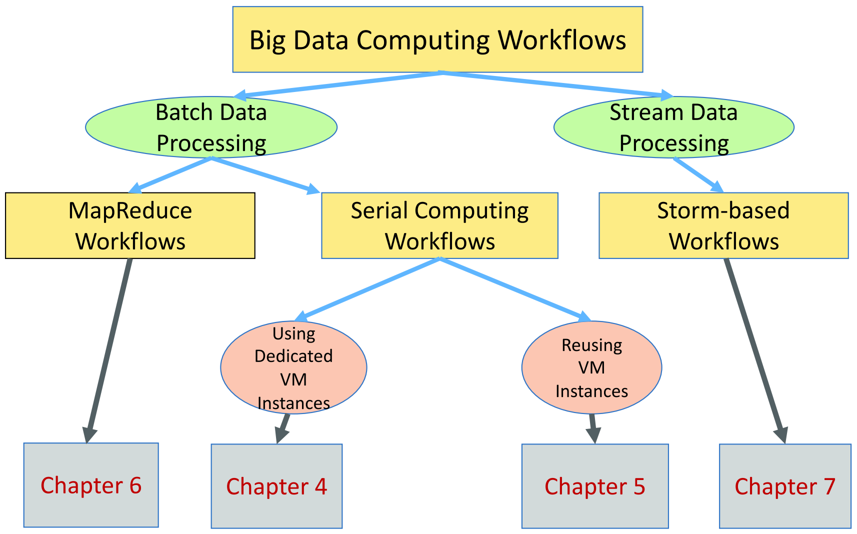


Figure 1.2 The roadmap of the dissertation research.

CHAPTER 2

RELATED WORK

In recent years, an increasing number of efforts have been made on resource provisioning and workflow mapping in clouds where both financial cost and workflow performance must be taken into consideration. Zeng *et al.* proposed a budget-conscious scheduler to minimize many-task workflow execution time within a certain budget [58]. Their scheduling strategy starts with tasks labeled with an average execution time on several VMs, and then sorts the tasks such that those on the initial critical path will be rescheduled first based on an objective function defined as Comparative Advantage (CA). In [15], Abrishami *et al.* designed a QoS-based workflow scheduling algorithm based on Partial Critical Paths (PCP) in SaaS clouds to minimize the cost of workflow execution within a user-defined deadline. As many existing critical-path heuristics, they schedule modules on the critical path first to minimize the cost without exceeding their deadline. PCP are then formed ending at those scheduled modules, and each PCP takes the start time of the scheduled critical module as its deadline. This scheduling process continues recursively until all modules are scheduled. In [44, 43], Mao *et al.* investigated the automatic scaling of clouds with budget and deadline constraints and proposed Scaling-Consolidation-Scheduling (SCS) with VMs as basic computing elements. In [32], Hacker proposed a combination of four scheduling policies based on an on-line estimation of physical resource usage. Rodriguez *et al.* proposed a combined resource provisioning and scheduling strategy for executing scientific workflows on IaaS clouds to minimize the overall execution cost while meeting a user-defined deadline [47]. They designed a Particle Swam Optimization-based approach that incorporates basic IaaS cloud principles such as a pay-as-you-go model, without considering the data transfer cost between data centers. And in [60], Wang *et al.* proposed a dynamic group learning distributed particle swarm optimization (DGLDPSO) for large-scale optimization and extends it for the large-scale cloud workflow scheduling. In [57], a novel directional and non-local-convergent particle swarm optimization (DNCPSO) was proposed and it employs non-linear inertia weight with selection and mutation operations by directional search process, which can reduce the makespan and cost dramatically and obtain a compromising result. Jiang *et al.* addressed two main challenges in executing large-scale workflow ensembles in public clouds, i.e., execution coordination and resource provisioning [36]. They developed DEWE v2, a pulling-based workflow management system with a profiling-based resource provisioning strategy, and demonstrated the effectiveness of their provisioning method in terms of both cost and deadline compliance. Also, while centralized datacenter schedulers can make high-quality placement decisions when scheduling tasks in a cluster, several efforts have been made to address the response time for interactive tasks and cluster utilization challenges. In [34], Gog *et al.* proposed Firmament, a centralized scheduler that scales to over ten thousand machines at sub-second placement latency even though it continuously reschedules all tasks via a min-cost max-flow (MCMF) optimization.

Most of the existing efforts are focused on workflow execution in a single cloud site. As scientific workflows are increasingly deployed across multiple clouds or data centers with network infrastructures to support inter-cloud data transfers, there is a need to revisit the workflow mapping problems in multi-cloud environments. For VM placement in such environments, cloud service providers must optimize the use of physical resources by a careful allocation of VMs to hosts, continuously balancing between the conflicting requirements on performance and operational costs. In recent years, several algorithms have been proposed for this important problem [42].

In [31], Abazari *et al.* tackled the Multi-objective workflow scheduling problem by considering both tasks security demands and interactions in secure tasks placement in the cloud. in [52], inspired by the hybrid chemical reaction optimization (HCRO) algorithm, the proposed workflow scheduling scheme is shown to be energy efficient under a deadline constraint. In [55], Ma *et al.* also considers the deadline as a constraint and proposed a scheduling algorithm that minimizes the execution cost of a workflow by dividing tasks into different levels according to the topological so that no dependency exists between tasks at the same level. In [38], Zhou *et al.* proposed two efficient workflow scheduling approaches for hybrid clouds that both consider makespan and monetary cost. In [29], besides scheduling strategy, Oliveira *et al.* tried to adress the failure of components as failures in a cloud system can simultaneously affect several users and depreciate the number of available computing resources. In [39], Bao *et al.* worked with microservice-based applications in clouds by performance modeling and prediction methods.

There also exist several efforts on MapReduce optimization. In [45], Moon *et al.* explored to optimize the MapReduce framework with solid-state drives in terms of performance, cost, and energy consumption within Hadoop. In [51], Tian *et al.* designed a job scheduler called "HScheduler" to minimize the makespan of multiple MapReduce jobs, which can be considered as a special case of MapReduce workflows with independent modules. In [49], Shi *et al.* designed an elastic resource provisioning and task scheduling mechanism in clouds to complete as many high-priority workflow jobs as possible under budget and deadline constraints. In [17], Alshammari *et al.* developed an enhanced MapReduce architecture to reduce the number of blocks to be read for computation and tested it with DNA datasets. These efforts are focused on independent MapReduce jobs in a single cloud.

As streaming data processing gains more attention, we also conduct a survey of related work on streaming data processing in various computing environments. Note that different from batch data processing, the scheduling problem in a streaming data processing framework processes a continuous flow of input instances upon their arrival.

Many existing efforts have been focused on workflow mapping or job scheduling in grid environments under different mapping and resource constraints. Agarwalla *et al.* proposed *Streamline*, a workflow scheduling scheme for streaming data, which places a coarse-grain dataflow graph on available grid resources [16]. mapping problem we consider: while each module in Condor Similar mapping problems are also studied in the context of sensor networks. Sekhar *et al.* proposed an optimal algorithm for mapping subtasks onto a large number of sensor nodes based on an A^* algorithm [48].

More recently, as Storm gains its popularity for streaming data processing in big data systems, a number of improvements have been made to Storm in either physical or cloud-based clusters. The current Storm platform employs a pseudo-random round-robin task scheduling and placement scheme without considering resource availability in the underlying cluster. This default scheme is simple but does not always yield the best performance in terms of workflow throughput and resource utilization. Many efforts have been made to improve throughput performance using resource-aware scheduling. In [25], Peng *et al.* proposed R-Storm (Resource-Aware Storm), a system that implements resource-aware scheduling within Storm, which is designed to increase overall throughput by maximizing resource utilization while minimizing network latency. In [30], Eskandari *et al.* considered the data transfer rate and traffic pattern between Storm's tasks and assign task pairs with heavy communication to the same node by dynamically employing two phases of graph partitioning. In [27], Chen *et al.* presented the design, implementation, and evaluation of G-Storm, a GPU-enabled parallel system based on Storm, which harnesses the massively parallel computing power of GPUs for high-throughput stream data processing.

There also exists some work on Storm scheduling that considers various application features such as data transfer, workflow topology, and QoS. In [56], Xu *et al.* proposed T-Storm, a traffic-aware online scheduler in Storm, to minimize inter-node and inter-process traffic for better performance with even fewer worker nodes. In [19], Aniello *et al.* proposed two schedulers for Storm to improve performance by adapting the deployment to application topologies, and by rescheduling the deployment at runtime based on traffic information. In [26], Cardellini *et al.* extended the Storm architecture by designing and implementing the support for distributed QoS-aware scheduling and run-time adaptivity.

Different from the aforementioned work that considers resources or application features, we take an orthogonal approach to maximize the throughput performance of Storm workflows in clouds by deciding an appropriate degree of parallelism for each component task of the topology and selecting a suitable virtual machine (VM) type for each component task processing an input instance.

CHAPTER 3

COST MODEL CONSTRUCTION

We construct analytical models to quantify workflow performance and cost, which facilitate a rigorous formulation of a budget-constrained optimization problem for big data computing workflows in multi-cloud environments.

3.1 A Three-layer Workflow Execution Architecture

As illustrated in Figure 3.1, we consider a three-layer architecture for workflow execution in multi-cloud environments. This architecture is generic and could be used to model many computing systems that utilize virtualized resources.

i) the top layer defines a workflow structure comprised of various computing modules with inter-module data transfer and execution dependency;

ii) the bottom layer defines a number of distributed data centers, which are organized as clusters of physical machines (PMs) and connected via high-speed networks;

iii) the middle layer defines a cloud-based network of VMs provisioned on the PMs located in different data centers.

Depending on the workflow mapping scheme, the execution of a workflow may incur an intra-cloud data transfer if two adjacent modules are mapped to the VMs within the same data center and an inter-cloud data transfer, otherwise.

As in the majority of the previous work, in the top layer, we model a scientific workflow as a weighted directed acyclic graph (DAG) $G_w(V_w, E_w)$, with $|V_w| = m$ modules and $|E_w|$ directed edges $e_{p,q}$, which represents the execution dependency

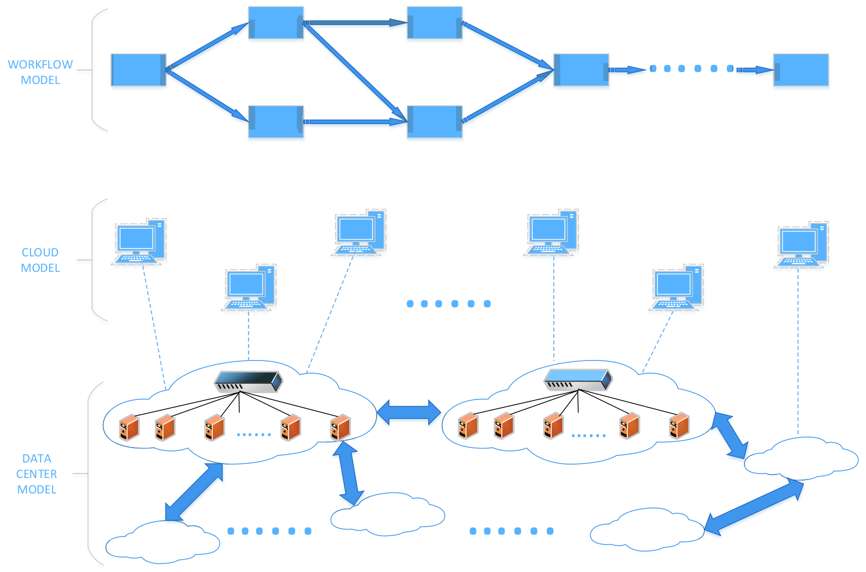


Figure 3.1 A three-layer architecture for workflow execution in multi-clouds.

between two adjacent modules w_p and w_q, and whose weight represents the data size DS transferred between them. In the bottom physical layer, we model the cloud infrastructure of each data center as a complete weighted graph $G_c(V_c, E_c)$ consisting of a limited number of PMs (nodes) connected via a high-speed switch organized as a typical PC cluster. Each PM is associated with a resource profile that defines the CPU frequency f_{CPU}, memory size s_{RAM}, I/O speed $r_{I/O}$, disk capacity c_{disk}, and a Network Interface Card (NIC) with uplink bandwidth BW_{up}, and downlink bandwidth BW_{down}. We also model a set of multiple interconnected data centers as a complete weighted graph $G_{mc}(V_{mc}, E_{mc})$. In the middle virtual layer, we model a set of fully connected VMs as a complete weighted graph $G'_{mc}(V'_{mc}, E'_{mc})$ consisting of $|V'_{mc}|$ VMs interconnected by $|E'_{mc}|$ virtual links. Note that the weight of each link in the above cloud models represents the corresponding link bandwidth BW.

We consider a set of n available VM types $VT = \{vt_1, vt_2, ..., vt_n\}$, each of which is associated with a set of performance attributes including CPU frequency

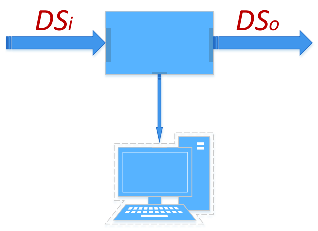


Figure 3.2 Data storage.

f'_{CPU}, I/O speed $r'_{I/O}$, and disk capacity c'_{disk}, as well as a VT pricing model $p(vt) = f(f'_{CPU}, r'_{I/O}, c'_{disk})$, which determines the financial cost per time unit for using a VM instance of type vt. Similarly, we consider a set of k available bandwidth types $BT = \{bt_1, bt_2, \ldots, bt_k\}$ to support inter-cloud data transfer and a bandwidth pricing model $p(dt) = f(bt)$. Note that we ignore the financial cost for intra-cloud data transfer, as followed by widely adopted cloud services in real-life scientific applications.

To run a computing module on a VM instance of a certain type, the disk capacity of the selected VM type must be large enough to meet the storage requirement of the module. Depending on the nature of the data processing, modules may have different storage requirements. As shown in Figure 3.2, if a module requires both input data DS_i and output data DS_o to be stored at the same time, i.e., the input data cannot be deleted before the output data is produced, the minimum required storage size is $DS_i + DS_o$; otherwise, the minimum required storage size is $\max(DS_i, DS_o)$. A feasible VM type $vt \in VT$ must meet the minimum storage requirement for a given computing module w, i.e.,

$$c'_{disk}(vt) \geq \begin{cases} DS_i(w) + DS_o(w), & \text{if } DS_i \text{ and } DS_o \text{ coexist;} \\ \max(DS_i(w), DS_o(w)), & \text{otherwise.} \end{cases} \tag{3.1}$$

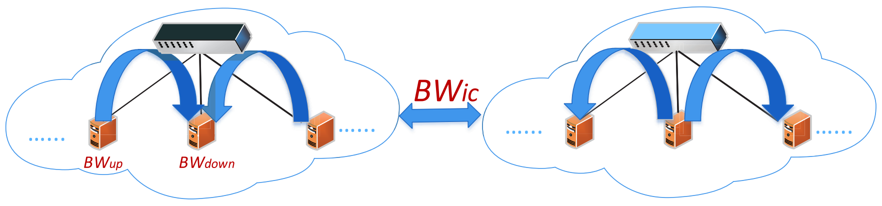


Figure 3.3 Intra-cloud and inter-cloud data transfer.

3.2 Inter-Module Transfer Time Cost Models

Time Cost of Data Transfer As shown in Figure 3.3, on each PM, the uplink bandwidth is equally shared if the PM sends data concurrently to multiple other PMs; similarly, the downlink bandwidth is equally shared if the PM receives data concurrently from multiple other PMs. The inter-cloud connection bandwidth is also equally shared by multiple concurrent data transfers between two data centers at different geographical locations. The time cost T_{dt} of an intra- or inter-cloud data transfer is determined by both the data size DS and the sharing dynamics of bandwidth BW, i.e.,

$$T_{dt} = \frac{DS}{\min(\frac{BW_{up}(PM_s)}{n_s}, \frac{BW_{down}(PM_r)}{n_r}, \frac{BW_{ic}}{n_{ic}})}, \tag{3.2}$$

where n_s, n_r, and n_{ic} is the number of concurrent data transfers from a sender PM_s, to a receiver PM_r, and over an inter-cloud connection, respectively.

Equation.

CHAPTER 4

SERIAL COMPUTING WORKFLOWS USING DEDICATED VM

4.1 Problem Formulation

In addition to the common cost model provided in Chapter 3, we construct a set of additional cost models specific to the execution of the serial computing workflows using dedicated VM instances.

4.1.1 Time and Financial Cost of Module Execution

A computing module may contain multiple code segments that are either CPU bound or I/O bound. In a CPU-bound segment, the program running time T_{CPU} is mainly determined by the workload WL of the module w and the CPU speed f'_{CPU} of the mapped VM type vt, i.e.,

$$T_{CPU} = \frac{WL(w)}{f'_{CPU}(vt)}. \tag{4.1}$$

In an I/O-bound segment of module w, the time $T_{I/O}$ spent on I/O operations depends on the input data size DS_i, the output data size DS_o, and the I/O speed $r'_{I/O}$ of the mapped VM type vt, i.e.,

$$T_{I/O} = \frac{DS_i(w) + DS_o(w)}{r'_{I/O}(vt)}. \tag{4.2}$$

The total execution time T_{me} of a module on a mapped VM is:

$$T_{me} = T_{CPU} + T_{I/O}. \tag{4.3}$$

End-to-end Delay of a Workflow The end-to-end delay ED of a workflow G_w is determined by the critical path (CP), which contains critical modules along the longest execution path, and is calculated as

$$ED = \sum_{\text{for all } w \in CP} (T_{me}(w) + T_{wait}(w)) + \sum_{\text{for all } e_w \in CP} (T_{dt}(e_w)), \tag{4.4}$$

where $T_{wait}(w)$ is the time a critical module w spends in waiting for its input data from all of its preceding modules.

Financial Cost Models The total financial cost C_{Total} of an entire workflow execution process is calculated as

$$\begin{aligned} C_{Total} = \sum_{\text{for all } w \in V_w} &((T_{me}(w, vt) + T_{wait}(w, vt)) \cdot p(vt)) \\ &+ \sum_{\text{for all } e_w \in E_w} T_{dt}(e_w, bt) \cdot p(dt), \end{aligned} \tag{4.5}$$

where the first term calculates the financial cost of all module executions, and the second term calculates the financial cost of all inter-cloud data transfers using selected bandwidth type bt over inter-cloud network links l_{ic}.

4.1.2 Problem Formulation

We first define a workflow mapping scheme $\mathcal{M}$ as

$$\mathcal{M} : v_w \rightarrow VM(PM, vt), \text{ for all } w \in V_w, \tag{4.6}$$

where $VM(PM, vt)$ represents a VM instance of type vt provisioned on a PM.

Based on the above mathematical models, we formulate a Budget-Constrained workflow mapping problem for Minimum End-to-end Delay in Multi-Cloud environments, referred to as BCMED-MC, as follows.

Definition 1. Given a DAG-structured workflow graph $G_w(V_w, E_w)$, a complete weighted multi-cloud network graph $G_{mc}(V_{mc}, E_{mc})$, a set BT of available inter-cloud bandwidth types, a set VT of available VM types, and a fixed financial budget B, we wish to find a workflow mapping scheme $\mathcal{M}$ to achieve the minimum end-to-end delay:

$$MED = \min_{all\, possible\, \mathcal{M}} ED, \tag{4.7}$$

while satisfying the following financial constraint:

$$C_{Total} \leq B. \tag{4.8}$$

The formulated BCMED-MC problem considers inter-cloud data transfers and I/O operations for big data workflows in multi-cloud environments. This is a generalized version of the MED-CC problem in [40], which only considers the execution time of CPU-bound modules in a single cloud site. Since MED-CC, which is a special case of BCMED-MC, has been proved to be NP-complete and non-approximable, so is BCMED-MC.

4.2 Algorithm Design

Our work targets data-, network-, and compute-intensive workflows deployed in multiple cloud-based data centers, where the inter-cloud data transfer time and financial cost could be significant and must be explicitly accounted for.

4.2.1 Multi-Cloud Workflow Mapping Algorithm

We design a Multi-Cloud Workflow Mapping (MCWM) algorithm to solve BCMED-MC. The pseudocode of MCWM is provided in Algorithm. 1, which consists of three key steps.

Step 1) We first sort the available VM types VT according to the disk space, CPU frequency, and I/O speed, and then call Function $AssignVT()$ in Algorithm. 2, which in turn calls Function $FindCP()$[1] to compute an initial critical path

[1]There exist efficient algorithms for finding a critical path in a weighted DAG-structured network.

(CP) based on the data size and computing workload of each module, and decides the worst possible VM type vt that meets the minimum storage requirement defined in Equation. 3.1 for each module.

Step 2) We check the financial cost against the budget: i) if it is already over the budget, there does not exist a feasible mapping scheme; ii) otherwise, we first upgrade the VM types for the critical modules prioritized by their execution time up to the best possible VM types and then the non-critical modules until the budget is exhausted.

Step 3) In Function $SelectPM()$ in Algorithm. 3, for each VM type vt, we select a PM in a cloud site to provision a corresponding VM instance. One basic guideline is to use the same PM in the same cloud whenever possible to avoid intra- and inter-cloud data transfer. If an inter-cloud data transfer is inevitable, we assign the smallest bandwidth type bt to the corresponding inter-cloud connection l_{ic}. Once a mapping scheme $\mathcal{M}$ is obtained, we calculate the total financial cost C_{Total} and check it against the budget B: i) if it is over the budget, we repeatedly downgrade the VM type vt for each module until the mapping scheme $\mathcal{M}$ becomes feasible; ii) otherwise, we repeatedly upgrade the bandwidth type bt for each inter-cloud data transfer until the budget is completely exhausted.

Step 4) If the workflow's total delay T_{Total} under the current mapping scheme $\mathcal{M}$ is shorter than that under the previous one, go back to Step 1) where $FindCP()$[2] computes a new CP based on the module execution/waiting and data transfer

[2]There exist efficient algorithms for finding a critical path in a weighted DAG-structured network.

time under the current mapping scheme; otherwise, terminate the mapping process.

In $SelectPM()$, we choose the PM and cloud site to provision a VM instance of a selected VM type vt for every module in the workflow. In every cloud, we check if it can provision the required VM instances one at a time. If some VM instances cannot be provisioned from this cloud, we move to another cloud. Since we consider homogeneous PMs in the same cloud, we try to provision the VM instances on the same PM whenever possible to avoid intra-cloud data transfer. However, the number and type of VM instances that can be provisioned on one PM is limited by the available physical resources of that PM. After finding all the cloud sites, we select the lowest bt between every two of those selected cloud sites for inter-cloud data transfer. We gradually upgrade the inter-cloud connection bandwidth and check it against the budget. If there is no budget to support even the worst bandwidth, we repeatedly downgrade the assigned VM type for each module.

The time complexity of each iteration in MCWM is $O(\max(m \cdot |E_w|, m \cdot n) + \max(|E_w|^2, |V_{mc}|^2 \cdot |E_w| \cdot m))$, where m is the number of modules in the workflow, n is the number of available VM types, and $|V_{mc}|$ is the number of clouds. is intra- or inter-cloud data transfer so that we can calculate the shared bandwidth.

4.3 Algorithm Implementation and Performance Evaluation

4.3.1 Simulation Settings

We implement the proposed MCWM algorithm in C++ and evaluate its performance in comparison with three algorithms, i.e., VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY [40]. VM-GREEDY is greedy on VM optimization. It assigns as many high-performance computers as possible within the budget and then randomly assigns the modules to the clouds. If the cost is over the budget, it downgrades the VM type of the modules until within the budget. On the contrary,

Algorithm 1: **MCWM**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a complete weighted multi-cloud network graph $G_{mc}(V_{mc}, E_{mc})$, a set of available VM types $VT[]$, the attributes $PM[][]$ of the PMs in each cloud, the workload $WL[\,]$ of each module, the inter-cloud bandwidth type $BT[]$, a fixed financial budget B.

Output: the workflow's end-to-end delay T_{Total}.

```
1: sort the VM types $VT[]$ in a decreasing order according to the disk space, $CPU$
   frequency, and $I/O$ speed;
2: $PrevT_{Total} = INF$;
3: **while** $(TRUE)$ **do**
4:    $AssignedVT[] = AssignVT(VT[], G_w, WL[], B)$;
5:    $C_{Total} = SelectPM(G_w, AssignedVT[], BT[], PM[][], VT[], B)$;
6:    **if** $(C_{Total} > B)$ **then**
7:       **if** $(PrevT_{Total} == INF)$ **then**
8:          **return** $-1$;
9:       **else**
10:         **return** $PrevT_{Total}$;
11:   use Equation. 6.15 to calculate $T_{Total}$;
12:   **if** $(T_{Total} < PrevT_{Total})$ **then**
13:      $PrevT_{Total} = T_{Total}$;
14:   **else**
15:      *break*;
16: **return** $PrevT_{Total}$;
```

Algorithm 2: **AssignVT**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a set of available VM types $VT[]$, the workload $WL[]$ of each module, and a fixed financial budget B.

Output: the VM type $AssignedVT[]$ assigned to each module.

```
1:  $CP[] = FindCP(G_w, VT[], WL[], AssignedVT[])$;
2:  for all (module $w_i \in V_w$) do
3:      $AssignedVT[i]$ = the worst possible $VT[]$;
4:      calculate module execution $T_{me}(w_i)$ and cost $C(T_{me}(w_i))$;
5:  calculate the total module execution cost $C(T_{me}(Total))$;
6:  for all (module $w_j \in CP[]$) do
7:      while ($C(T_{me}(Total)) \leq B$) do
8:          $AssignedVT[j]$ = the next better $VT[]$ than the one in the previous round;
            $C(T_{me}(w_j))$;
9:      downgrade $AssignedVT[j]$;
10: for all (module $w_k \notin CP[]$) do
11:     while ($C(T_{me}(Total)) \leq B$) do
12:         $AssignedVT[k]$ = the next better $VT[]$ than the one in the previous round;
            $C(T_{me}(w_k))$;
13:     downgrade $AssignedVT[k]$;
14: return $AssignedVT[]$;
```

Algorithm 3: **SelectPM**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, the VM type $AssignedVT[]$ assigned to each module, the inter-cloud bandwidth type $BT[]$, the attributes $PM[][]$ of the PMs in each cloud, a set of available VM types $VT[]$, a fixed financial budget B.

Output: the total cost C_{Total}.

```
1: for all (cloud $i$) do
2:   for all (module $w_j$) do
3:     if ($w_j$ has not been assigned to any cloud) then
4:       assign $w_j$ to cloud $i$;
5: update the BW and unit price on each edge;
6: calculate the total workflow execution cost $C_{Total}$;
7: if ($C_{Total} < B$) then
8:   if (there are inter-cloud data transfers) then
9:     for all (inter-cloud link $l_{ic}$) do
10:      while ($l_{ic}$'s BW type bt is not the best) do
11:        upgrade $l_{ic}$'s bt;
12:        calculate $C_{Total}$;
13:        if ($C_{Total} > B$) then
14:          if ($l_{ic}$'s bt is not the worst) then
15:            downgrade $l_{ic}$'s bt;
16:            break;
17: else
18:   for all (module $w_j$) do
19:     while ($w_j$'s vt is not the worst) do
20:       downgrade $w_j$'s vt;
21:       calculate $C_{Total}$;
22:       if ($C_{Total} < B$) then
23:         break;
24: return $C_{Total}$.
```

BW-GREEDY is greedy on data transfer optimization. It assigns each module with the worst possible VM type that satisfies its storage requirement, randomly deploys the modules in the clouds, and assigns the largest bandwidth for inter-cloud data transfer. CRITICAL-GREEDY was proposed in [40] to solve the MED-CC problem. We first use this algorithm to determine the VM type for each module, and then randomly deploy the modules in the clouds using the smallest bandwidth for any inter-cloud data transfer.

To evaluate the algorithm performance, we consider different problem sizes from small to large scales. The problem size is defined as a 3-tuple $(m, |E_w|, n)$, where m is the number of workflow modules, $|E_w|$ is the number of workflow links, and n is the number of available VM types. We generate workflow instances with different scales in a random manner as follows: i) lay out all m modules sequentially along a pipeline, each of which is assigned a workload randomly generated within range $[5, 500]$, which represents the total number of instructions; ii) for each module, add an input edge from a randomly selected preceding module and add an output edge to a randomly selected succeeding module (note that the first module only needs output and the last module only needs input); iii) randomly pick up two modules from the pipeline and add a directed edge between them (from left to right) until we reach the given number of edges.

We compare our MCWM algorithm with VM-GREEDY, BW-GREEDY and CRITICAL-GREEDY [40], in terms of the total workflow execution time under the same budget constraint. The MED performance improvement of MCWM over the other algorithms in comparison is defined as:

$$Imp(Other) = \frac{MED_{Other} - MED_{MCWM}}{MED_{Other}} \times 100\%,$$

where MED_{Other} is the MED achieved by the other three algorithms, i.e., VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY, and MED_{MCWM} is the MED achieved by MCWM.

4.3.2 Simulation Results

Comparison with Optimal Solutions First we compare the performance of MCWM with optimal solutions in small-scale problems with 5, 6 and 7 modules and VM types. For each problem size, we randomly generate 50 problem instances with different module workloads and DAG topologies. For each problem instance, we specify 20 different budget levels. We run all these four algorithms on these instances and compare the MED results with the optimal ones computed by an exhaustive search-based approach. Figure 4.1 shows the number of optimal results among 1000 instances (50 workflow instances $\times$ 20 budget levels) achieved by MCWM, VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY under different problem sizes. and thus is not visible in the chart. We observe that MCWM is more likely to achieve the optimality than the other algorithms in a statistical sense, which indicates the efficacy of MCWM. We also observe that neither VM-GREEDY nor CRITICAL-GREEDY achieves any optimal results in these cases (no bars are produced for them in Figure 4.1). Also, the average gap from optimal results for MCWM, VM-GREEDY, BW-GREEDY and CRITICAL-GREEDY across all 1000 runs are 24%, 90%, 93% and 69%, respectively.

Comparison with VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY We further consider 20 problem sizes from small to large scales, indexed from 1 to 20. In each problem size, we randomly generate 20 problem instances, for each of which, we choose 20 budget levels with an equal interval of $\Delta B = (B_{max} - B_{min})/20$ within a certain budget range $[B_{min}, B_{max}]$. We provide in Table 4.1 the average MED

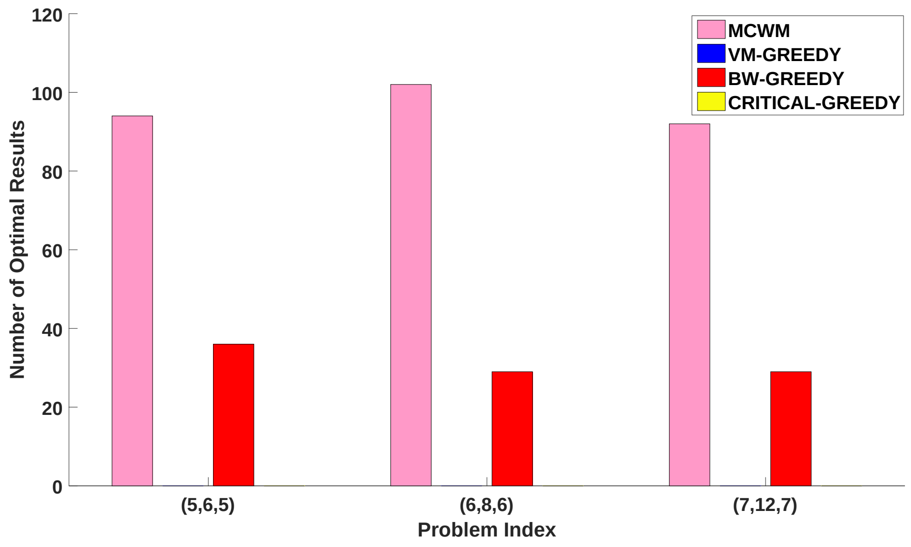


Figure 4.1 The number of optimal results among 1000 instances (50 workflow instances × 20 budget levels) produced by MCWM, VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY under different problem sizes.

improvement percentages together with standard deviations achieved by MCWM over VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY across all the 20 budget levels, which are further plotted in Figure 4.2 for a visual comparison.

Also, for each of the 20 budget levels from low to high values, indexed from 1 to 20, we run the scheduling simulation by iterating through 20 problem sizes from small to large scales. We provide in Table 4.2 the average MED improvement percentages together with standard deviations achieved by MCWM over VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY across all the 20 problem sizes, which are further plotted in Figure 4.3 for a visual comparison.

For a better illustration, we plot the overall performance improvement percentage of MCWM over VM-GREEDY in Figure 4.4, where x axis denotes the budget increment across 20 levels and y axis denotes the index of 20 problem sizes from small to large scales. Each point (x, y, imp) in the $3D$ plot represents the average performance improvement across all 20 problem instances of the same problem size under the same budget level (the actual budget values may be different in different instances). at different problem indices for a microscopic examination. These performance results show that MCWM achieves 50% performance improvement over VM-GREEDY on average. Similarly, we plot the overall performance improvement percentage of MCWM over BW-GREEDY in Figure 6.9. These performance results show that MCWM achieves 55% performance improvement over BW-GREEDY on average. Also, we plot the overall performance improvement percentage of MCWM over CRITICAL-GREEDY in Figure 4.6. These performance results show that MCWM achieves 50% performance improvement over CRITICAL-GREEDY on average. Such performance improvements are considered significant for workflow execution in large-scale scientific applications.

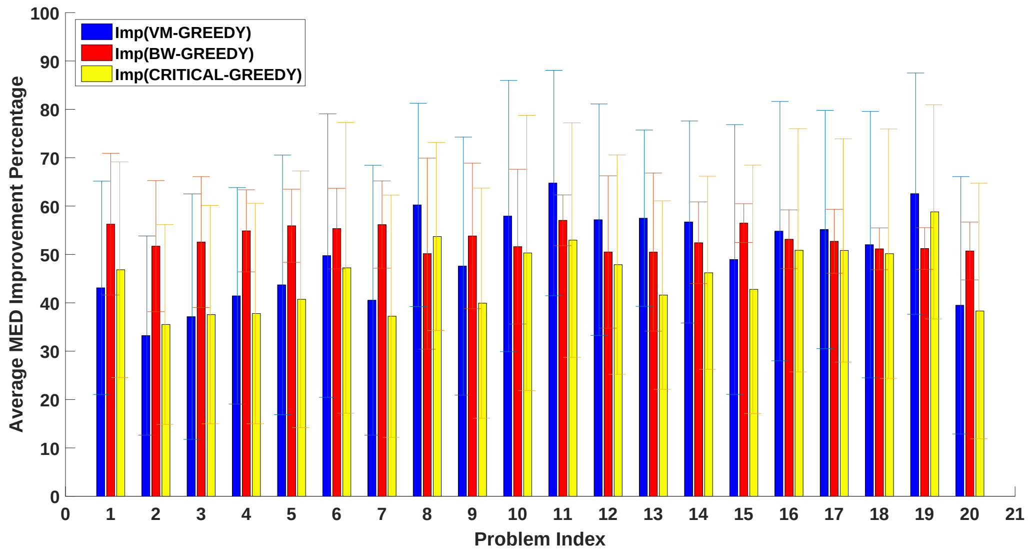


Figure 4.2 The average MED performance improvement percentages (%) with standard deviations across 400 instances (20 budget levels $\times$ 20 random workflow instances) for each problem size.

Algorithm.

Table 4.1 The Average MED Improvement Percentages α = Imp(VM-GREEDY), β = Imp(BW-GREEDY), and γ = Imp(CRITICAL-GREEDY) of MCWM and the Corresponding Standard Deviations Across 400 Instances (20 Budget Levels $\times$ 20 Random Workflow Instances) at Different Problem Sizes

Idx	$(m, \|E_w\|, n)$	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	$(5, 7, 5)$	43.10	22.06	56.26	14.66	46.84	22.31
2	$(8, 14, 6)$	33.23	20.58	51.72	13.56	35.52	20.66
3	$(10, 18, 6)$	37.14	25.37	52.56	13.54	37.56	22.57
4	$(13, 30, 7)$	41.44	22.38	54.88	8.48	37.78	22.79
5	$(15, 60, 7)$	43.72	26.83	55.93	7.56	40.73	26.53
6	$(18, 80, 8)$	49.76	29.31	55.35	8.33	47.24	30.06
7	$(20, 100, 8)$	40.55	27.90	56.18	9.02	37.22	25.04
8	$(23, 150, 9)$	60.25	21.02	55.17	19.75	53.71	19.45
9	$(25, 200, 9)$	47.60	26.67	53.83	15.05	39.92	23.79
10	$(28, 250, 10)$	57.94	28.05	51.61	16.00	50.31	28.45
11	$(30, 300, 10)$	64.77	23.31	57.06	5.25	52.96	24.24
12	$(33, 380, 11)$	57.18	23.93	50.51	15.75	47.89	22.68
13	$(35, 400, 11)$	57.49	18.22	50.48	16.35	41.60	19.50
14	$(38, 450, 12)$	56.71	20.88	52.42	8.45	46.21	19.97
15	$(40, 500, 12)$	48.95	27.89	56.48	4.01	42.78	25.68
16	$(43, 550, 13)$	54.82	26.80	53.14	6.07	50.86	25.15
17	$(45, 580, 13)$	55.17	24.64	52.72	6.62	50.83	23.09
18	$(48, 600, 14)$	52.02	27.54	51.16	4.32	50.16	25.79
19	$(50, 650, 14)$	62.57	24.95	51.22	4.31	58.81	22.15
20	$(53, 700, 15)$	39.49	26.61	50.71	5.98	38.31	26.42

Table 4.2 The Average MED Improvement Percentages α = Imp(VM-GREEDY), β = Imp(BW-GREEDY), and γ = Imp(CRITICAL-GREEDY) of MCWM and the Corresponding Standard Deviations Across 400 Instances (20 Problem Sizes $\times$ 20 Random Workflow Instances) at Difference Budget Levels

Budget Level	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	47.93	20.53	54.47	8.17	47.45	24.01
2	46.95	24.86	54.27	8.45	46.89	24.28
3	47.26	23.85	54.47	8.50	46.73	23.43
4	47.87	25.17	54.83	8.21	47.77	24.73
5	46.15	24.04	53.82	9.09	45.68	23.34
6	47.70	25.09	54.25	8.24	46.99	24.73
7	48.73	24.96	55.04	8.25	49.15	24.42
8	47.64	25.55	55.45	7.86	47.47	24.38
9	47.27	25.31	53.80	8.53	47.26	25.12
10	47.92	24.53	54.46	8.67	47.97	24.36
11	47.15	25.84	54.79	7.89	46.08	25.65
12	48.75	24.97	54.65	8.52	49.14	24.69
13	45.07	23.85	53.96	8.62	44.74	23.58
14	46.43	24.71	54.63	8.25	45.85	24.83
15	48.76	25.61	54.57	8.62	48.89	25.34
16	46.51	23.87	54.47	8.62	46.52	23.62
17	47.37	24.01	54.28	9.10	47.61	23.81
18	48.68	24.56	55.23	8.49	48.48	24.50
19	46.13	24.52	54.10	7.87	45.96	24.42
20	47.19	23.81	54.92	8.17	47.42	23.65

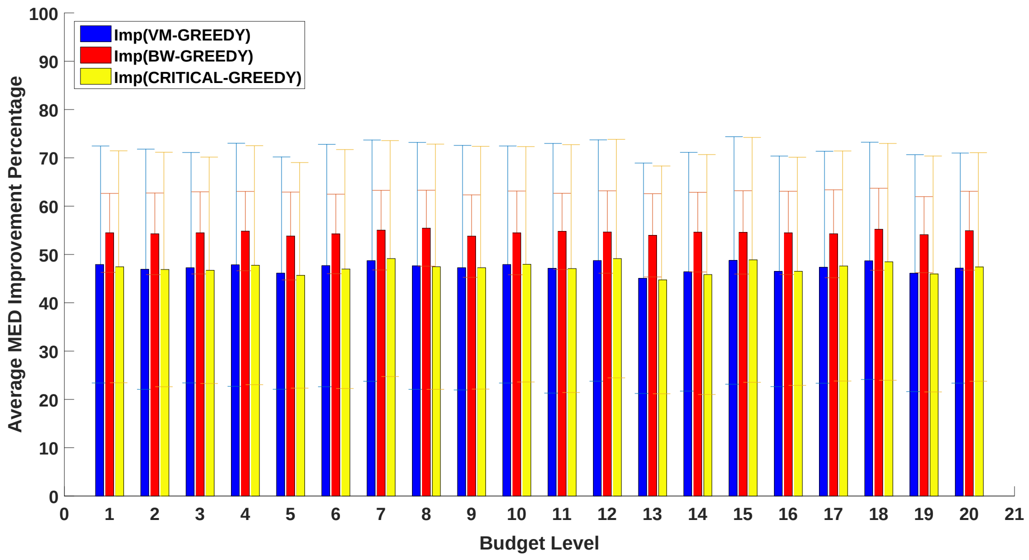


Figure 4.3 The average performance improvement percentages with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) for each budget level.

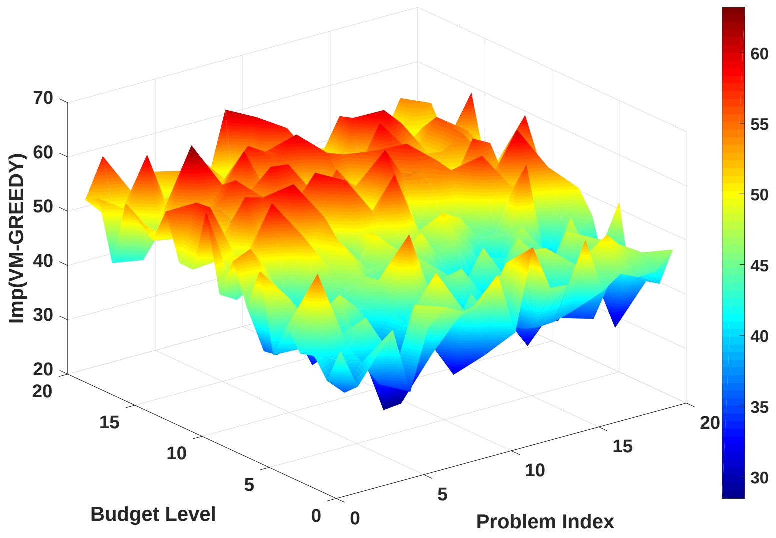


Figure 4.4 The overall performance improvement percentages (%) of MCWM over VM-GREEDY with varying budget levels and problem sizes.

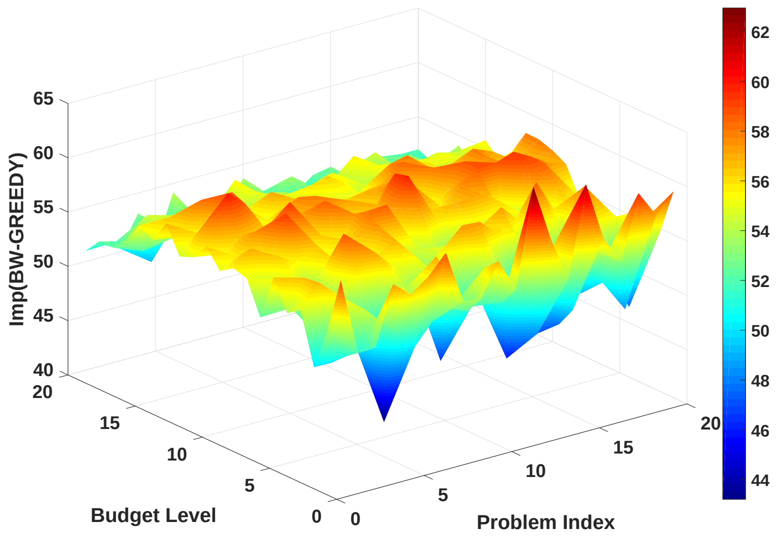


Figure 4.5 The overall performance improvement percentages (%) of MCWM over BW-GREEDY with varying budget levels and problem sizes.

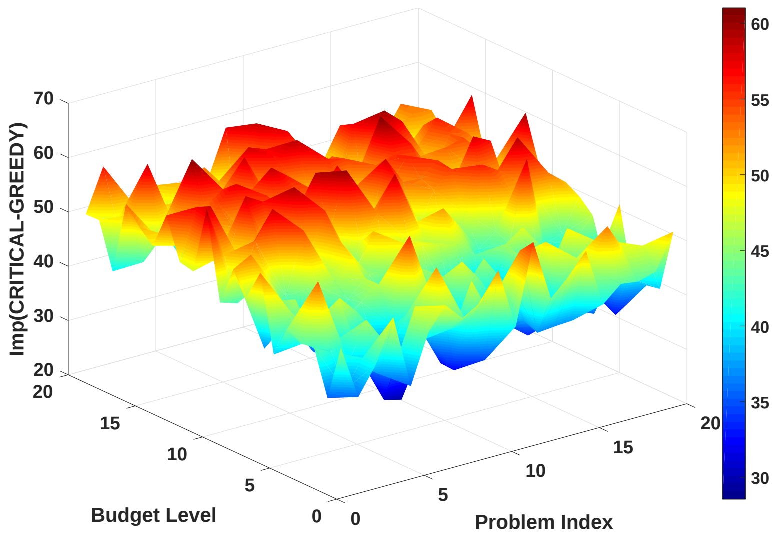


Figure 4.6 The overall performance improvement percentages (%) of MCWM over CRITICAL-GREEDY with varying budget levels and problem sizes.

CHAPTER 5

SERIAL COMPUTING WORKFLOWS REUSING VM

5.1 Problem Formulation

In addition to the common cost model provided in Chapter 3, we construct a set of additional cost models specific to the execution of the serial computing workflows reusing VM instances.

5.1.1 Time and Financial Cost of Module Execution:

A computing module may contain multiple code segments that are either CPU bound or I/O bound. In a CPU-bound segment, the program running time T_{CPU} is mainly determined by the workload WL of the module w and the CPU speed f'_{CPU} of the mapped VM type vt, i.e.,

$$T_{CPU} = \frac{WL(w)}{f'_{CPU}(vt)}. \tag{5.1}$$

In an I/O-bound segment of module w, the time $T_{I/O}$ spent on I/O operations depends on the input data size DS_i, the output data size DS_o, and the I/O speed $r'_{I/O}$ of the mapped VM type vt, i.e.,

$$T_{I/O} = \frac{DS_i(w) + DS_o(w)}{r'_{I/O}(vt)}. \tag{5.2}$$

The total execution time T_{me} of a module on a mapped VM is:

$$T_{me} = T_s + T_{CPU} + T_{I/O}, \tag{5.3}$$

where the startup time $T_s = 0$ if the module is executed on a reused VM instance.

End-to-end Delay of a Workflow: The end-to-end delay ED of a workflow G_w is determined by the critical path (CP), which contains critical modules along the longest execution path, and is calculated as

$$ED = \sum_{\text{for all } w \in CP} (T_{me}(w) + T_{wait}(w)) + \sum_{\text{for all } e_w \in CP} (T_{dt}(e_w)), \tag{5.4}$$

where $T_{wait}(w)$ is the time a critical module w spends in waiting for its input data from all of its preceding modules.

Financial Cost Models The total financial cost C_{Total} of an entire workflow execution process is calculated as

$$C_{Total} = C_{Total}(dt) + C_{Total}(VM), \tag{5.5}$$

where

$$C_{Total}(dt) = \sum_{\text{for all } e_w \in E_w} DS(e_w, l_{ic}) \cdot p(dt), \tag{5.6}$$

$$C_{Total}(VM) = \sum_{\text{for all } w \in V_w} ((T_{me}(w, vt) + T_{wait}(w, vt)) \cdot p(vt)), \tag{5.7}$$

where the $C_{Total}(VM)$ calculates the financial cost of all module executions, and the $C_{Total}(dt)$ calculates the financial cost of all inter-cloud data transfers. $DS(\cdot)$ is the data size of edge e_w over an inter-cloud network links l_{ic}. Again, $p(dt)$ denotes the volume-based pricing model for inter-cloud data transfer.

5.1.2 Problem Formulation

We first define a workflow mapping scheme $\mathcal{M}$ as

$$\mathcal{M} : v_w \to VM(PM, vt), \text{ for all } w \in V_w, \tag{5.8}$$

where $VM(PM, vt)$ represents a VM instance of type vt provisioned on a PM.

Based on the above cost models, we formulate a Budget-Constrained workflow mapping problem for Minimum End-to-end Delay in Multi-Cloud environments, referred to as BCMED-MC, as follows.

Definition 2. Given a DAG-structured workflow $G_w(V_w, E_w)$, a complete weighted multi-cloud network $G_{mc}(V_{mc}, E_{mc})$, a set BT of available inter-cloud bandwidth types, a set VT of available VM types, and a fixed financial budget B, we wish to find a workflow mapping scheme $\mathcal{M}$ to achieve the minimum end-to-end delay:

$$MED = \min_{all\,possible\,\mathcal{M}} ED, \tag{5.9}$$

while satisfying the following financial constraint:

$$C_{Total} \leq B. \tag{5.10}$$

BCMED-MC considers inter-cloud data transfers and I/O operations for big data workflows across multiple clouds. This is a generalized version of the MED-CC problem in [40], which only considers the execution time of CPU-bound modules in a single cloud. Since MED-CC, which is a special case of BCMED-MC, has been proved to be NP-complete and non-approximable, so is BCMED-MC.

5.2 Algorithm Design

Our work targets data-, network-, and compute-intensive workflows deployed in multiple cloud-based data centers, where the inter-cloud data transfer time and financial cost could be significant and must be explicitly accounted for.

5.2.1 Multi-Cloud Workflow Mapping Algorithm

We design an iterative critical path (CP)-based multi-cloud workflow mapping algorithm, referred to as ICPWM, to solve BCMED-MC. This algorithm uses a key

Algorithm 4: **ICPWM**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a complete weighted multi-cloud network graph $G_{mc}(V_{mc}, E_{mc})$, a set of available VM types $VT[]$, the attributes $PM[][]$ of the PMs in each cloud, the workload $WL[\,]$ of each module, the inter-cloud bandwidth type $BT[]$, a fixed financial budget B, the number NI of iterations.

Output: the workflow's minimum end-to-end delay T_{Total}.

```
 1: sort the VM types $VT[]$ in a decreasing order according to the disk space, $CPU$
    frequency, and $I/O$ speed;
 2: $LeastT_{Total} = INF$;
 3: for $i = 0; i < NI; i++$ do
 4:    $AssignedVT\_Group[] =$
       $AssignVT(VT[], G_w, WL[], B, AssignedVT\_Group[])$;
 5:    $C_{Total} =$
       $SelectPM(G_w, AssignedVT\_Group[], BT[], PM[][], VT[], B)$;
 6:    if $(C_{Total} > B)$ then
 7:       continue;
 8:    use Equation. 6.15 to calculate $T_{Total}$;
 9:    if $(T_{Total} < LeastT_{Total})$ then
10:       $LeastT_{Total} = T_{Total}$;
11: if $LeastT_{Total} == INF$ then
12:    return $-1$;
13: return $LeastT_{Total}$;
```

concept of module-group, which is defined as a subset of two or more contiguous modules in the workflow running on the same VM instance (of the same VM type). The pseudocode of ICPWM is provided in Algorithm. 4, which consists of the following key steps.

Step 1) We sort the available VM types VT by three hardware resources in the order of disk space, CPU frequency, and I/O speed, and then call Function $AssignVT()$ in Algorithm. 5, which in turn calls Function $FindCP()$[1] to compute an initial critical path (CP) based on the data size and computing workload of each module running on the most powerful VM type vt, and decides the worst possible VM type vt that meets the minimum storage requirement defined in Equation. 3.1 for each module.

Step 2) On the CP calculated in Step 1), if there exists only one module or one module-group, we upgrade its vt by one level; if there exist two or more modules or module-groups, we take the following four steps, each of which calls Function *Grouping* in Algorithm. 6 as follows:

- Step a) i) Group all adjacent modules on the CP using the same VM type vt to a module-group. ii) Check every pair of adjacent modules and upgrade the module using the worse VM type by one level. iii) Group all pairs of adjacent modules on the CP using the same VM type to a module-group.
- Step b) i) Group any module into its adjacent module-group using the same VM type. ii) For every module-group and its adjacent module, if the module's VM type is worse, upgrade it by one level. iii) Group any

[1]There exist efficient algorithms for finding a CP in a weighted DAG-structured network.

module into an adjacent module-group on the CP using the same VM type.

  - Step c) i) For every module-group and its adjacent module, if the module-group's VM type is worse, upgrade it by one level. ii) Group any module into an adjacent module-group on the CP using the same VM type.
  - Step d) i) Group all adjacent module-groups using he same VM type to a module-group. ii) For every pair of adjacent module-groups, upgrade the module-group with a worse VM type by one level. iii) Group all pairs of adjacent module-groups on the CP using the same VM type to a module-group. iv) Remove this current CP, create a new temporary graph G'_w, compute a new CP, and go back to Step a) until no more CP is found.

Step 3) In Function $SelectPM()$ in Algorithm. 7, for each VM type vt, we select a PM in a cloud site to provision a corresponding VM instance. One basic guideline is to use the same PM in the same cloud whenever possible to avoid intra- and inter-cloud data transfer. If an inter-cloud data transfer is inevitable, we assign the smallest BW type bt to the corresponding inter-cloud connection l_{ic}. Once a mapping scheme $\mathcal{M}$ is obtained, we calculate the total financial cost C_{Total} and check it against the budget B: i) if it is over the budget, we repeatedly downgrade the VM type vt of the VM instance that runs the smallest number of modules until the mapping scheme $\mathcal{M}$ becomes feasible; ii) otherwise, we upgrade the BW type bt for each inter-cloud data transfer by one level.

Step 4) Go back to Step 1) until the termination condition is met and output the minimum end-to-end delay $LeastT_{Total}$.

In $SelectPM()$, we choose the PM and cloud site to provision a VM instance of a selected VM type vt for every module in the workflow. In every cloud, we check if it can provision the required VM instances one at a time. If some VM instances cannot be provisioned from this cloud, we move to another cloud. Since we consider homogeneous PMs in the same cloud, we try to provision the VM instances on the same PM whenever possible to avoid intra-cloud data transfer. However, the number and type of VM instances that can be provisioned on one PM is limited by the available physical resources of that PM. After finding all the cloud sites, we select the lowest bt between every two of those selected cloud sites for inter-cloud data transfer. We upgrade the inter-cloud connection bandwidth by one level. If there is no budget to support even the worst bandwidth, we repeatedly downgrade the VM type of the VM instance that runs the smallest number of modules until the mapping scheme $\mathcal{M}$ becomes feasible.

The time complexity of each iteration in ICPWM is $O(\max(m^2 \cdot |E_w|, m \cdot n) + \max(|E_w|^2, |E_w| \cdot m \cdot |V_{mc}|^2, |E_w| \cdot m^2 \cdot n)$, where m is the number of modules in the workflow, n is the number of available VM types, and $|V_{mc}|$ is the number of clouds. is intra- or inter-cloud data transfer so that we can calculate the shared bandwidth.

5.3 Algorithm Implementation and Performance Evaluation

5.3.1 Simulation Settings

We implement the proposed ICPWM algorithm in C++ and evaluate its performance in comparison with three algorithms, i.e., VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY [54]. VM-GREEDY is greedy on VM optimization. It assigns as many high-performance computers as possible within the budget and then randomly assigns the modules to the clouds. If the cost is over the budget, it

Algorithm 5: **AssignVT**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a set of available VM types $VT[]$, the workload $WL[]$ of each module, a fixed financial budget B, and the VM type and module-group index assigned to each module $AssignedVT\_Group[]$.

Output: the VM type and module-group index assigned to each module $AssignedVT\_Group[]$

```
1:  $G'_w = G_w$;
2:  **while** $|V_w| > 0$ **do**
3:      $CP[] = FindCP(G'_w, VT[], WL[], AssignedVT\_Group[])$;
4:      **if** ($CP[]$ contains only one module or one module-group) **then**
5:          upgrade its $vt$ by one level;
6:      **else**
7:          $AssignedVT\_Group[] =$
            $Grouping(CP[], AssignedVT\_Group[], VT[], B,$
            $module, module)$;
8:          $AssignedVT\_Group[] =$
            $Grouping(CP[], AssignedVT\_Group[], VT[], B,$
            $module, module\text{-}group)$;
9:          $AssignedVT\_Group[] =$
            $Grouping(CP[], AssignedVT\_Group[], VT[], B,$
            $module\text{-}group, module)$;
10:         $AssignedVT\_Group[] =$
            $Grouping(CP[], AssignedVT\_Group[], VT[], B,$
            $module\text{-}group, module\text{-}group)$;
11:     $G'_w = G'_w - CP[]$;
12: **return** $AssignedVT\_Group[]$;
```

Algorithm 6: **Grouping**

Input: a CP, a set of available VM types $VT[]$, a fixed financial budget B, a group unit GU_1, a group unit GU_2, the VM type and module-group index assigned to each module $AssignedVT\_Group[]$.

Output: the VM type and module-group index assigned to each module $AssignedVT\_Group[]$.

```
 1: for (adjacent $GU_1$ and $GU_2$ on the $CP$) do
 2:    if ($AssignedVT\_Group[GU_1].vt ==$
       $AssignedVT\_Group[GU_2].vt$
       and $AssignedVT\_Group[GU_1].Group \neq$
       $AssignedVT\_Group[GU_2].Group$) then
 3:       group $GU_1$ and $GU_2$;
 4: if (both $GU_1$ and $GU_2$ are modules or module-groups) then
 5:    for (adjacent $GU_1$ and $GU_2$ on the $CP$) do
 6:       upgrade the worse $vt$ of $GU_1$ and $GU_2$ by one level;
 7: else
 8:    for (adjacent $GU_1$ and $GU_2$ on $Path$) do
 9:       if ($GU_1$'s $vt$ is worse than $GU_2$'s $vt$) then
10:          upgrade $GU_1$'s $vt$ by one level;
11: for (adjacent $GU_1$ and $GU_2$ on the $CP$) do
12:    if ($AssignedVT\_Group[GU_1].vt =$
       $AssignedVT\_Group[GU_2].vt$
       and $AssignedVT\_Group[GU_1].Group \neq$
       $AssignedVT\_Group[GU_2].Group$) then
13:       group $GU_1$ and $GU_2$ together;
14: return $AssignedVT\_Group[]$;
```

Algorithm 7: **SelectPM**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, the VM type and module-group index assigned to each module $AssignedVT\_Group[]$, the inter-cloud bandwidth type $BT[]$, the attributes $PM[][]$ of the PMs in each cloud, a set of available VM types $VT[]$, a fixed financial budget B.

Output: the total cost C_{Total}.

```
1:  for all (cloud $i$) do
2:      for all (module $w_j$) do
3:          if ($w_j$ has not been assigned to any cloud) then
4:              assign $w_j$ to cloud $i$;
5:  update the BW and unit price on each edge;
6:  calculate the total workflow execution cost $C_{Total}$;
7:  if ($C_{Total} < B$) then
8:      if (there are inter-cloud data transfers) then
9:          for all (inter-cloud link $l_{ic}$) do
10:             while ($l_{ic}$'s BW type $bt$ is not the best) do
11:                 upgrade $l_{ic}$'s $bt$;
12:                 calculate $C_{Total}$;
13:                 if ($C_{Total} > B$) then
14:                     if ($l_{ic}$'s $bt$ is not the worst) then
15:                         downgrade $l_{ic}$'s $bt$;
16:                         break;
17: else
18:     for all modules and module-groups sorted in an increasing order of module count do
19:         while (its $vt$ is not the worst) do
20:             downgrade its $vt$;
21:             calculate $C_{Total}$;
22:             if ($C_{Total} < B$) then
23:                 break;
24: return $C_{Total}$.
```

downgrades the VM type of the modules until within the budget. On the contrary, BW-GREEDY is greedy on data transfer optimization. It assigns each module to the VM type with the minimum possible resources to satisfy its storage requirement, randomly deploys the modules in the clouds, and assigns the largest bandwidth for inter-cloud data transfer. CRITICAL-GREEDY was proposed in [54] to solve the MED-CC problem. We first use this algorithm to determine the VM type for each module, and then randomly deploy the modules in clouds using the smallest bandwidth for any inter-cloud data transfer. For the algorithms in comparison, we also reuse a VM instance when any contiguous modules are assigned to the same VM type.

To evaluate the algorithm performance, we consider different problem sizes from small to large scales. The problem size is defined as a 3-tuple $(m, |E_w|, n)$, where m is the number of workflow modules, $|E_w|$ is the number of workflow links, and n is the number of available VM types. We generate workflow instances with different scales in a random manner as follows: i) lay out all m modules sequentially along a pipeline, each of which is assigned a workload randomly generated within range $[5, 500]$, which represents the total number of instructions; ii) for each module, add an input edge from a randomly selected preceding module and add an output edge to a randomly selected succeeding module (note that the first module only needs output and the last module only needs input); iii) randomly pick up two modules from the pipeline and add a directed edge between them (from left to right) until we reach the given number of edges.

We compare our ICPWM algorithm with VM-GREEDY, BW-GREEDY and CRITICAL-GREEDY [40], in terms of the total workflow execution time under the same budget constraint. The MED performance improvement of ICPWM over the

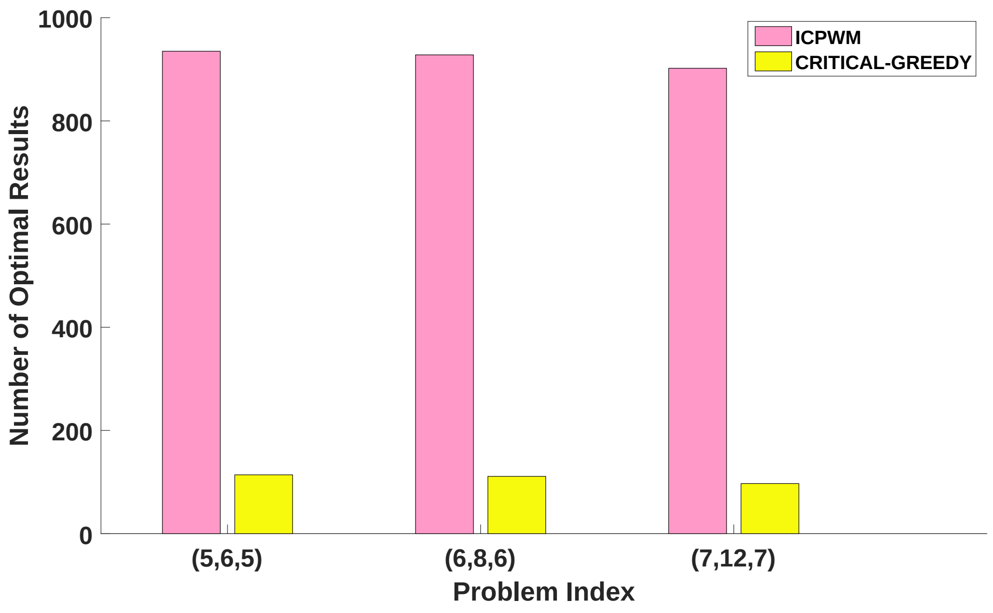


Figure 5.1 The number of optimal results among 1000 instances (50 workflow instances × 20 budget levels) produced by ICPWM, VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY under different problem sizes.

other algorithms in comparison is defined as:

$$Imp(Other) = \frac{MED_{Other} - MED_{ICPWM}}{MED_{Other}} \times 100\%,$$

where MED_{Other} is the MED achieved by the other three algorithms, i.e., VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY, and MED_{ICPWM} is the MED achieved by ICPWM.

5.3.2 Simulation Results

Comparison with Optimal Solutions in Small-Scale Problems We first compare the performance of ICPWM with optimal solutions in small-scale problems with 5, 6 and 7 modules and VM types. For each problem size, we randomly generate

50 problem instances with different module workloads and DAG topologies. For each problem instance, we specify 20 different budget levels. We run all these four algorithms on these instances and compare the MED results with the optimal ones computed by an exhaustive search-based approach. Figure 5.1 shows the number of optimal results among 1000 instances (50 workflow instances $\times$ 20 budget levels) achieved by ICPWM, VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY under different problem sizes. We observe that ICPWM is more likely to achieve the optimality than the other algorithms in a statistical sense, which indicates the efficacy of ICPWM. We also observe that neither VM-GREEDY nor BW-GREEDY achieves any optimal results in these cases (no bars are produced for them in Figure 5.1).

Comparison with VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY We further consider 20 problem sizes from small to large scales, indexed from 1 to 20. In each problem size, we randomly generate 20 problem instances, for each of which, we choose 20 budget levels with an equal interval of $\Delta B = (B_{max} - B_{min})/20$ within a certain budget range $[B_{min}, B_{max}]$. We provide in Table 5.1 the average MED improvement percentages together with standard deviations achieved by ICPWM over VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY across all the 20 budget levels, which are further plotted in Figure 5.2 for a visual comparison.

Also, for each of the 20 budget levels from low to high values, indexed from 1 to 20, we run the scheduling simulation by iterating through 20 problem sizes from small to large scales. We provide in Table 5.2 the average MED improvement percentages together with standard deviations achieved by ICPWM over VM-GREEDY, BW-GREEDY, and CRITICAL-GREEDY across all the 20 problem sizes, which are further plotted in Figure 6.7 for a visual comparison.

For a better illustration, we plot the overall performance improvement percentage of ICPWM over VM-GREEDY in Figure 5.4, where x axis denotes

Table 5.1 The Average MED Improvement Percentages $\alpha = \text{Imp(VM-GREEDY)}$, $\beta = \text{Imp(BW-GREEDY)}$, and $\gamma = \text{Imp(CRITICAL-GREEDY)}$ of ICPWM and the Corresponding Standard Deviations Across 400 Instances (20 Budget Levels $\times$ 20 Random Workflow Instances) of Different Problem Sizes

Idx	$(m, \|E_w\|, n)$	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	$(5, 7, 5)$	47.02	16.63	52.20	16.84	42.76	46.38
2	$(8, 14, 6)$	52.17	23.38	57.22	19.37	42.34	31.86
3	$(10, 18, 6)$	57.35	12.65	61.49	13.72	54.99	17.38
4	$(13, 30, 7)$	50.64	23.83	54.76	23.41	40.74	37.80
5	$(15, 60, 7)$	52.58	25.54	58.56	21.53	51.54	29.80
6	$(18, 80, 8)$	56.66	20.64	60.75	20.24	61.23	23.33
7	$(20, 100, 8)$	58.72	18.32	63.36	16.53	62.15	19.65
8	$(23, 150, 9)$	37.92	27.46	40.55	26.35	55.26	23.30
9	$(25, 200, 9)$	45.61	25.95	48.37	25.32	65.51	16.78
10	$(28, 250, 10)$	30.90	26.42	34.51	25.38	52.26	20.57
11	$(30, 300, 10)$	38.91	28.34	40.33	26.56	54.10	27.17
12	$(33, 380, 11)$	24.38	21.67	29.69	21.05	51.87	14.60
13	$(35, 400, 11)$	40.45	26.72	44.13	24.17	65.01	16.18
14	$(38, 450, 12)$	45.67	27.60	49.29	25.34	62.91	19.68
15	$(40, 500, 12)$	45.59	32.22	48.07	30.00	41.36	22.88
16	$(43, 550, 13)$	59.17	27.77	60.56	26.93	65.71	31.86
17	$(45, 580, 13)$	65.44	18.68	67.96	16.75	70.07	22.89
18	$(48, 600, 14)$	56.70	26.04	58.37	25.52	55.71	24.64
19	$(50, 650, 14)$	64.63	15.15	66.22	15.00	62.37	26.66
20	$(53, 700, 15)$	64.12	19.91	65.56	19.18	49.48	32.51

Table 5.2 The Average MED Improvement Percentages α = Imp(VM-GREEDY), β = Imp(BW-GREEDY), and γ = Imp(CRITICAL-GREEDY) of ICPWM and the Corresponding Standard Deviations Across 400 Instances (20 problem sizes $\times$ 20 Random Workflow Instances) at Difference Budget Levels

Budget Level	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	57.78	24.00	61.15	21.64	50.45	37.73
2	57.60	23.92	60.53	22.51	51.12	37.63
3	58.97	21.95	62.36	19.57	55.63	29.48
4	57.70	23.91	60.98	21.75	51.73	32.83
5	58.13	22.18	61.29	20.55	52.93	40.95
6	58.65	22.71	61.90	21.04	50.99	40.11
7	59.71	22.90	62.79	20.80	53.85	36.87
8	59.58	25.65	59.75	23.54	50.46	43.33
9	55.14	27.68	58.76	24.35	47.53	46.70
10	56.88	26.38	60.75	21.90	53.29	37.03
11	58.91	22.47	62.16	20.93	53.17	38.17
12	57.20	24.22	60.37	22.26	52.72	36.94
13	59.25	21.96	62.38	20.60	53.49	31.84
14	60.37	20.99	63.37	19.56	55.69	40.20
15	58.42	23.68	61.78	20.60	53.61	36.56
16	57.27	24.88	60.35	22.79	51.61	37.97
17	57.29	23.82	60.40	22.56	50.36	44.12
18	58.05	23.37	61.08	21.37	54.14	32.49
19	57.73	23.81	60.90	21.94	50.26	45.36
20	57.31	24.51	60.51	22.63	51.32	42.72

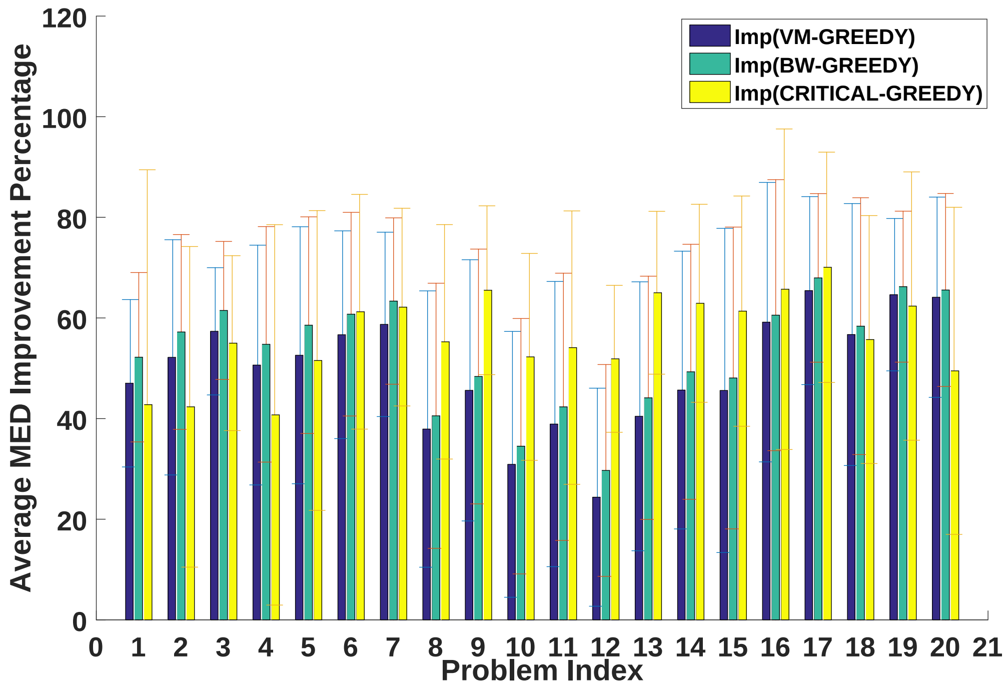


Figure 5.2 The average MED performance improvements (%) with standard deviations across 400 instances (20 budget levels × 20 random workflow instances) for each problem size.

the budget increment across 20 levels and y axis denotes the index of 20 problem sizes from small to large scales. Each point (x, y, imp) in the $3D$ plot represents the average performance improvement across all 20 problem instances of the same problem size under the same budget level (the actual budget values may be different in different instances). problem indices for a microscopic examination. These performance results show that ICPWM achieves 58% performance improvement over VM-GREEDY on average. Similarly, we plot the overall performance improvement percentage of ICPWM over BW-GREEDY in Figure 5.5. These performance results show that ICPWM achieves 62% performance improvement over BW-GREEDY on average. Also, we plot the overall performance improvement percentage of ICPWM over CRITICAL-GREEDY in Figure 5.6. These performance results show that

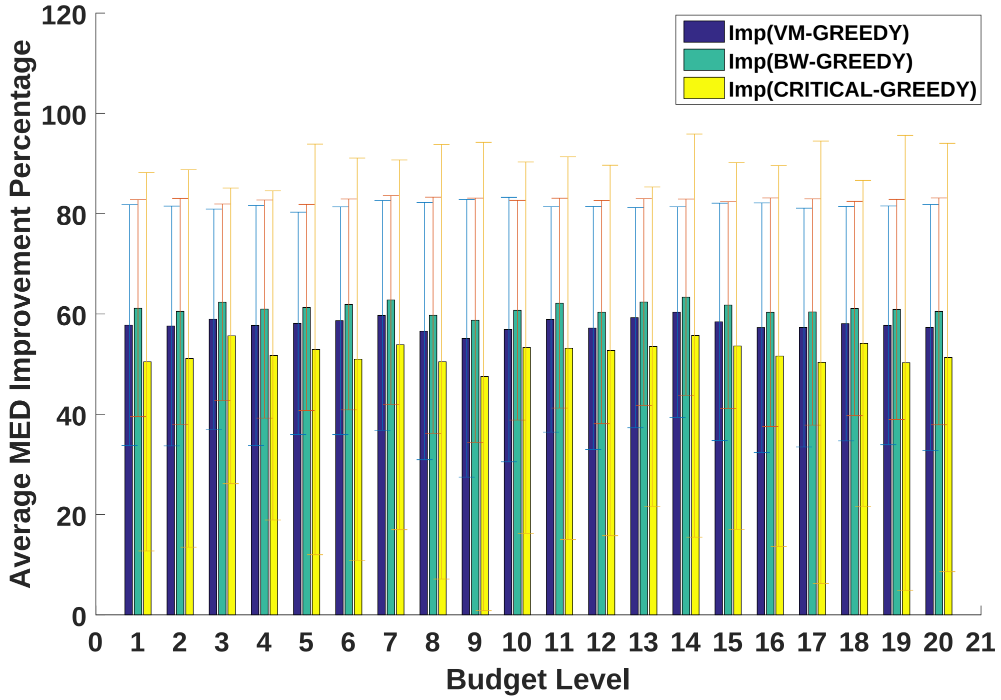


Figure 5.3 The average performance improvements (%) with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) for each budget level.

ICPWM achieves 55% performance improvement over CRITICAL-GREEDY on average. Such performance improvements are considered significant for workflow execution in large-scale scientific applications.

Simulation Using Pegasus Workflows We further evaluate the performance of our algorithm using five Pegasus workflow types of different topologies, i.e., Montage, CyberShake, Epigenomics, LIGO's Inspiral Analysis and Sipht: i) the Montage workflow is used to combine multiple input images to create custom mosaics of the sky; ii) the CyberShake workflow is used to characterize earthquake hazards in a region; iii) the epigenomics workflow is used to automate various operations in genome sequence processing; iv) the LIGO's Inspiral Analysis workflow is used

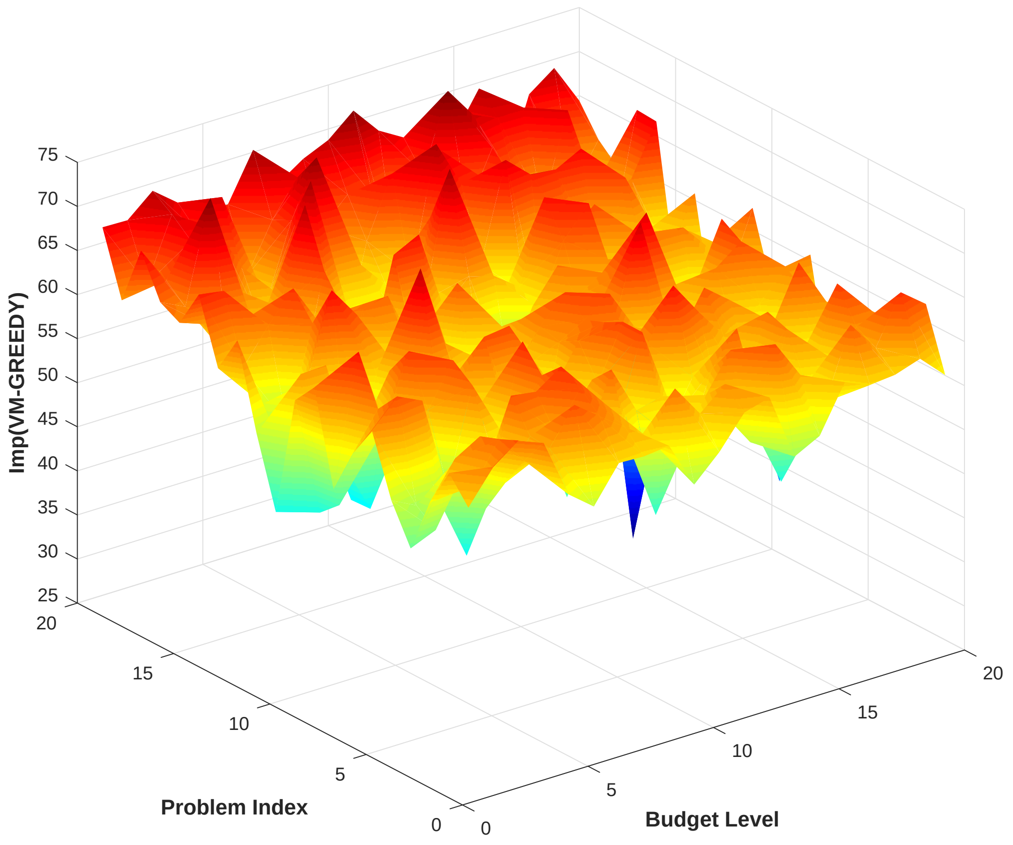


Figure 5.4 The overall performance improvements (%) of ICPWM over VM-GREEDY with varying budget levels and problem sizes.

to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems; and v) the Sipht workflow is used to automate the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database. Their workflow structures are illustrated in Figures. 5.7, 5.8, 5.9, 5.10, and 5.11.

We use WorkflowSim to generate simulated workflows, where the number of nodes (i.e., modules) in each workflow type is set to be 50. The topologies vary from

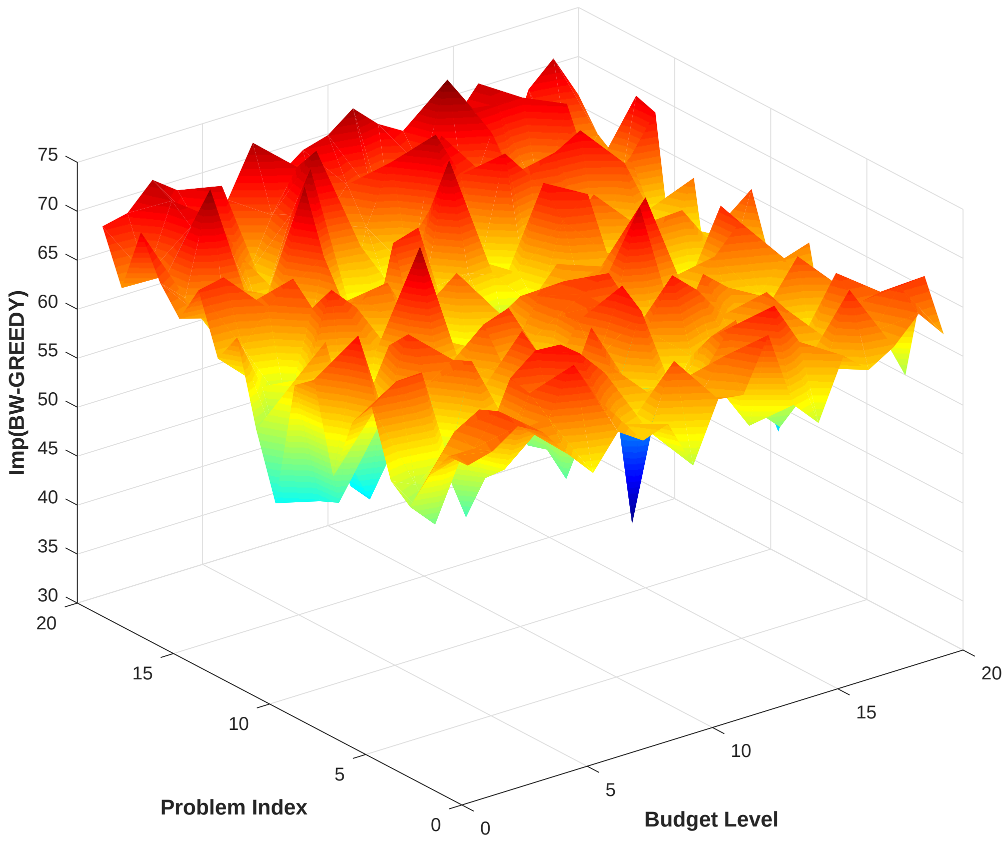


Figure 5.5 The overall performance improvements (%) of ICPWM over BW-GREEDY with varying budget levels and problem sizes.

type to type and WorkflowSim automatically searches for the closest number of nodes for each workflow type of a different directed graph density, as shown in Table 5.3. We set 20 different budget levels and generate 20 workflow instances at each budget level for each workflow type. After obtaining the mapping schemes from ICPWM, VM-GREEDY, BW-GREEDY and CRITICAL-GREEDY, we plot the corresponding average performance improvements of ICPWM over the other three algorithms with standard deviations across all 20 instances in Figures. 5.12, 5.13, 5.14, 5.15, and 5.16 for five different workflow types, respectively. These results show that ICPWM

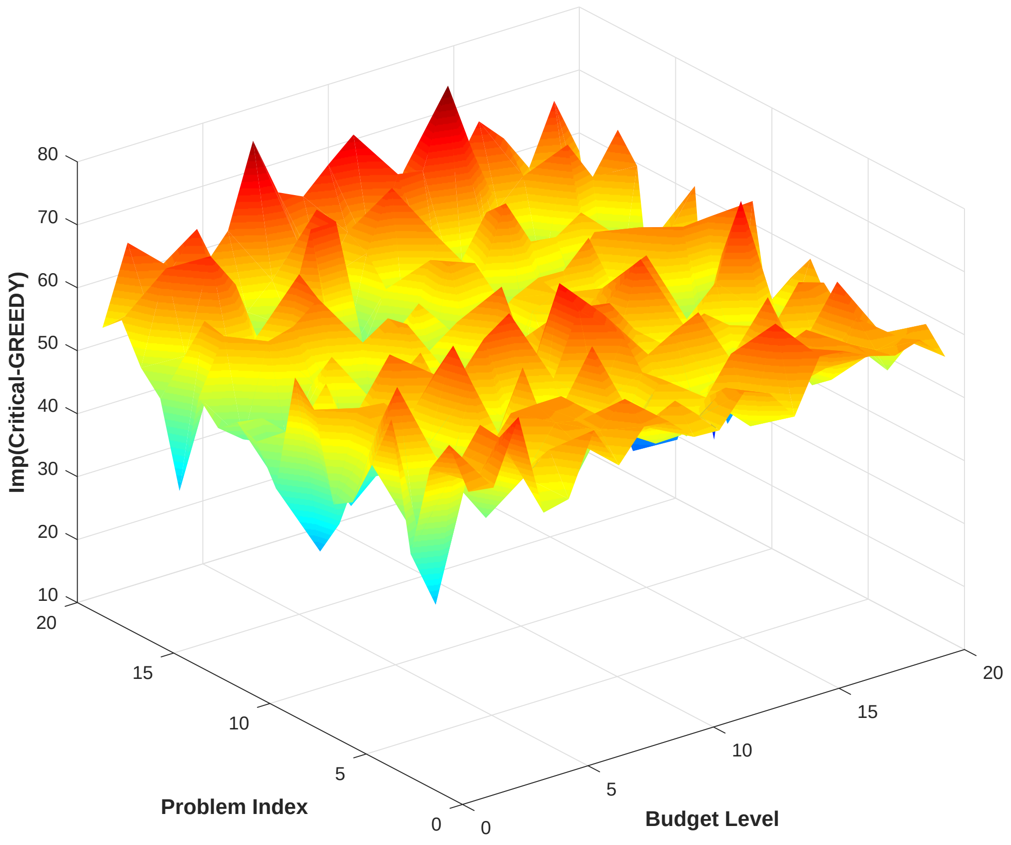


Figure 5.6 The overall performance improvements (%) of ICPWM over CRITICAL-GREEDY with varying budget levels and problem sizes.

achieves 70-80% performance improvement over VM-GREEDY and BW-GREEDY, and 25-40% performance improvement over CRITICAL-GREEDY.

Convergence of ICPWM To investigate the convergence property of ICPWM, we run this algorithm on the problem instance of Index 15 under three different budget levels, i.e., low, medium, and high. Note that the medium budget level is used in the above simulations, the low budget level is one third and the high budget level is three times the medium budget level. We plot the optimization process of

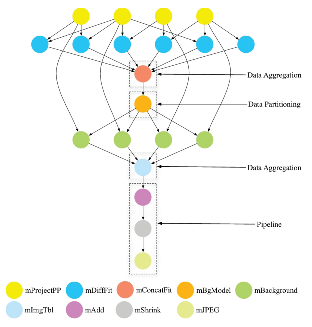


Figure 5.7 Montage

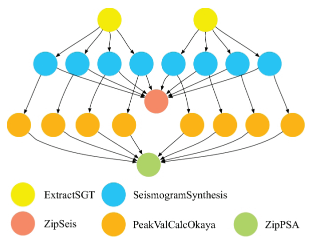


Figure 5.8 CyberShake

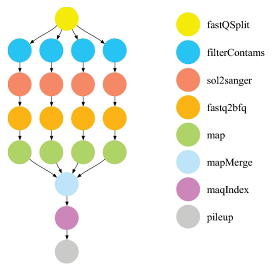


Figure 5.9 Epigenomics

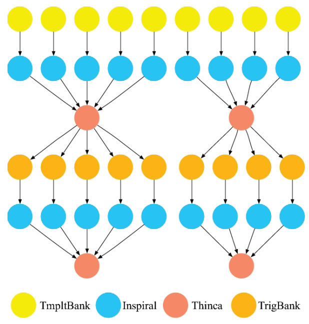


Figure 5.10 LIGO's Inspiral Analysis

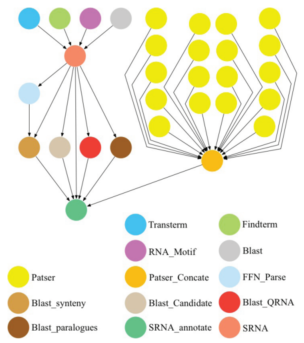


Figure 5.11 Sipht

ICPWM in these three scenarios in Figure 5.17, which shows that ICPWM converges to the minimum end-to-end delay in about 20 iterations.

5.3.3 Experimental Results

WRF Workflow To evaluate the performance of our algorithm in real computing environments, we conduct workflow experiments based on the Weather Research and Forecasting (WRF) model [50], which has been widely adopted for regional to continental scale weather forecast. The WRF model [9] generates two large classes of simulations either with an ideal initialization or utilizing real data. In our experiments, the simulations are generated from real data, which usually requires preprocessing from the WPS package [10] to provide each atmospheric and static field with fidelity appropriate to the chosen grid resolution for the model.

Table 5.3 The Number of Nodes/Edges and Directed Graph Density in Five Different Workflow Types

Attribute	Montage	CyberShake	Epigenomics	LIGO Inspiral Analysis	Sipht
Node count	51	51	47	51	49
Edge count	109	94	54	61	58
Graph density	0.04275	0.03686	0.02498	0.02392	0.02466

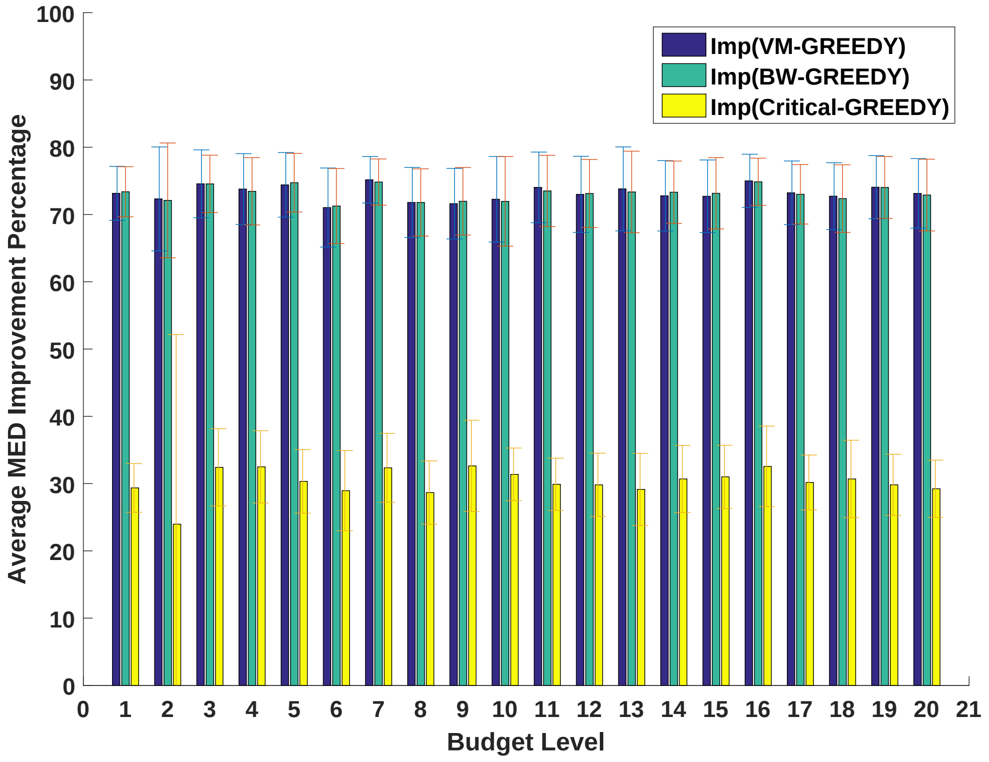


Figure 5.12 The average performance improvements (%) with standard deviations across 20 instances at each budget level for the Montage workflow.

The structure of a general WRF workflow is illustrated in Figure 7.13, where the WPS consists of three independent programs: geogrid.exe, ungrib.exe,

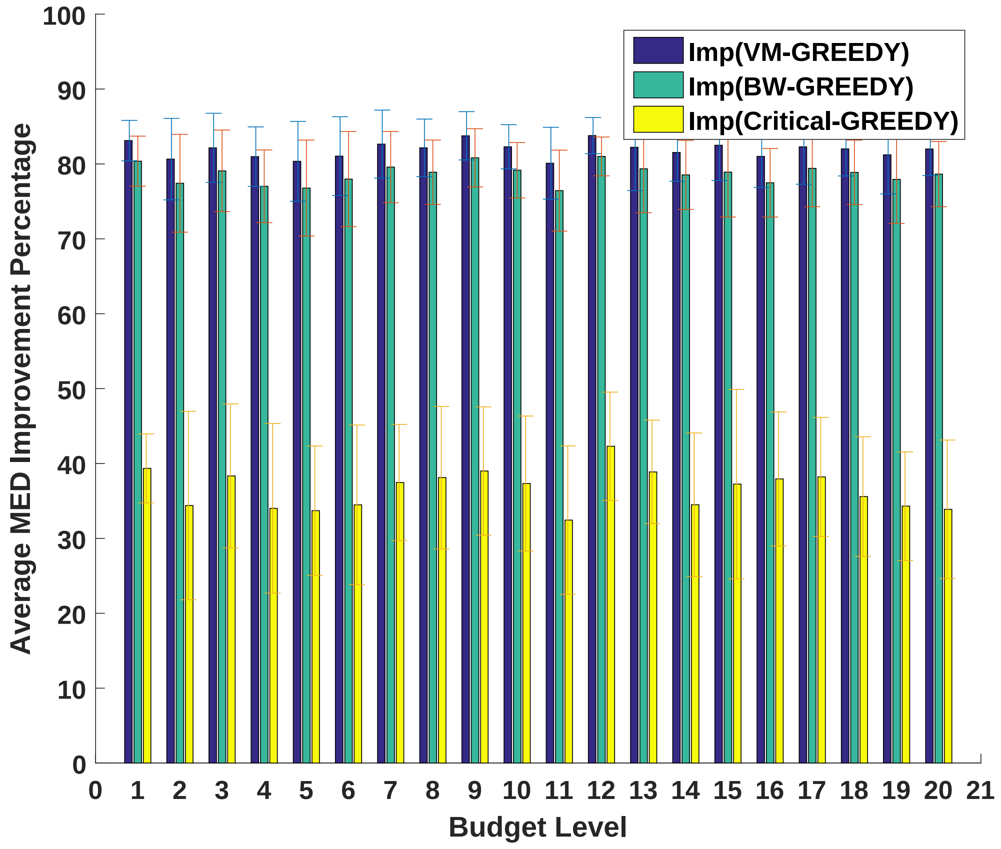


Figure 5.13 The average performance improvements (%) with standard deviations across 20 instances at each budget level for the CyberShake workflow.

and metgrid.exe [54]. The geogrid program defines the simulation domains and interpolates various terrestrial datasets to the model grids. The user can specify information in the namelist file of WPS to define simulation domains. The ungrib program "degrib" the data and stores the results in a simple intermediate format. The metgrid program horizontally interpolates the intermediate-format meteorological data that are extracted by the ungrib program into the simulation domains defined by the geogrid program. The interpolated metgrid output can then be ingested by the WRF package, which contains an initialization program real.exe for real data and a numerical integration program wrf.exe. The postprocessing model

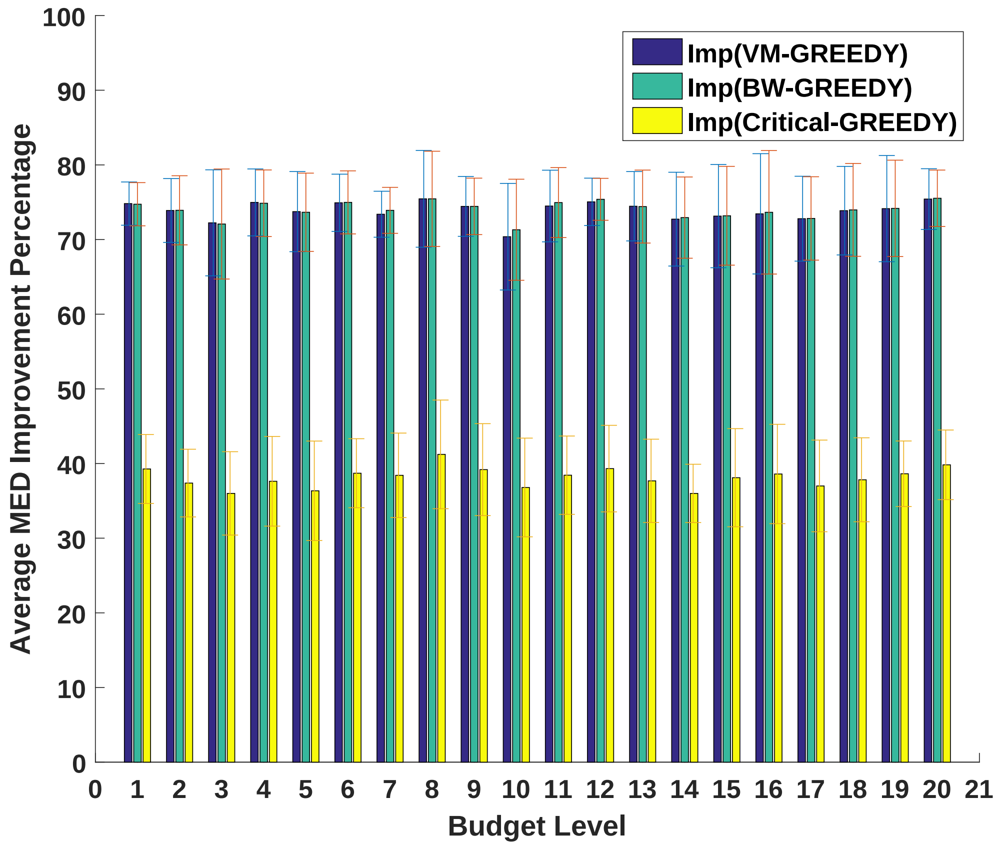


Figure 5.14 The average performance improvements (%) with standard deviations across 20 instances at each budget level for the Epigenomics workflow.

consists of ARWpost and GrADs. ARWpost reads-in WRF-ARW model data and creates output files for display by GrADS.

Experimental Settings We set up a Hadoop 1.2.1 cluster in a multi-cloud environment involving both Google Cloud Platform (GCP) [11] and Amazon Web Services (AWS) [12]. As shown in Table 7.1, the cluster consists of four VM instances of different CPU speeds, two from AWS and two from GCP, each of which has sufficient RAM size and disk space for WRF execution. We launch the VM instances in advance before actually running workflow modules. When data transfer is inevitable, different bandwidths may be used over different links, as shown in

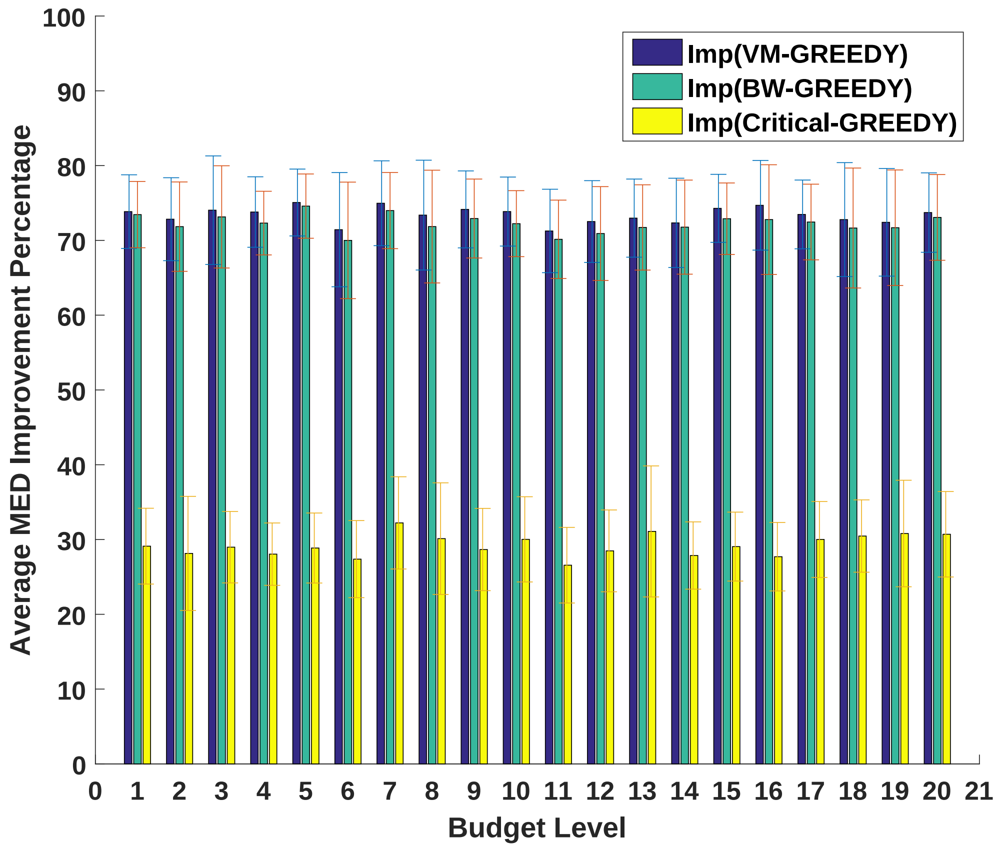


Figure 5.15 The average performance improvements (%) with standard deviations across 20 instances at each budget level for the LIGO's Inspiral Analysis workflow.

Table 6.5, and the volume-based pricing policy for inter-cloud data transfer is shown in Table 5.6.

We duplicate three WRF pipelines each from ungrib.exe to ARWpost.exe, and group these programs into different aggregate modules to simulate real-life workflow clustering and provide various module parallelism, as shown in Figures. 5.19 and 7.15. Figure 7.15 is a high-level view of grouped workflow in Figure 5.19, where w_0 and w_7 are the start and end modules [54].

Performance Comparison We store the entire WRF workflow as well as the input data on each VM instance before execution. To estimate the module execution

Table 5.4 Hadoop Cluster and VM Specifications of Different VM Types in AWS and GCP

VM Type	Cloud	Core count	CPU (GHz)	RAM (GB)	Disk (GB)	I/O read (MB/s)	I/O write (MB/s)	Price (USD/hour)
$VT1$	AWS	1	2.5×1	2	80	10702	80	0.026
$VT2$	GCP	2	2.3×2	8	80	9581	124	0.074
$VT3$	GCP	8	2.3×8	30	80	8856	125	0.284
$VT4$	AWS	16	2.3×16	64	80	10288	132	0.958

Table 5.5 Bandwidth Matrix BW

BW (Mb/s)	$VT1$	$VT2$	$VT3$	$VT4$
$VT1$	—	32.6	32.6	97.7
$VT2$	16.3	—	97.7	32.6
$VT3$	14.0	97.7	—	48.8
$VT4$	97.7	32.6	32.6	—

Table 5.6 Data Transfer Price Matrix *price*

price (USD/GB)	AWS	GCP
AWS	0.000	0.155
GCP	0.120	0.000

Table 5.7 Execution Time Matrix T_E

$T_{i,j}$ (ms)	w1	w2	w3	w4	w5	w6
VT_1	1743	881	2960	6220	1534260	777022
VT_2	1595	746	2953	5912	714140	349283
VT_3	1470	736	2887	5644	57909	34606
VT_4	1323	661	2735	5487	43831	21251

Table 5.8 Execution Cost Matrix C_E

$C_{i,j}$ (cents)	w1	w2	w3	w4	w5	w6
VT_1	45.4	22.9	79.96	161.7	39890.8	20202.6
VT_2	118.0	55.2	218.5	437.5	52846.4	25846.9
VT_3	417.5	209.0	819.9	1602.9	16446.2	9829.1
VT_4	1267.4	633.2	2620.1	5256.5	41990.1	20358.5

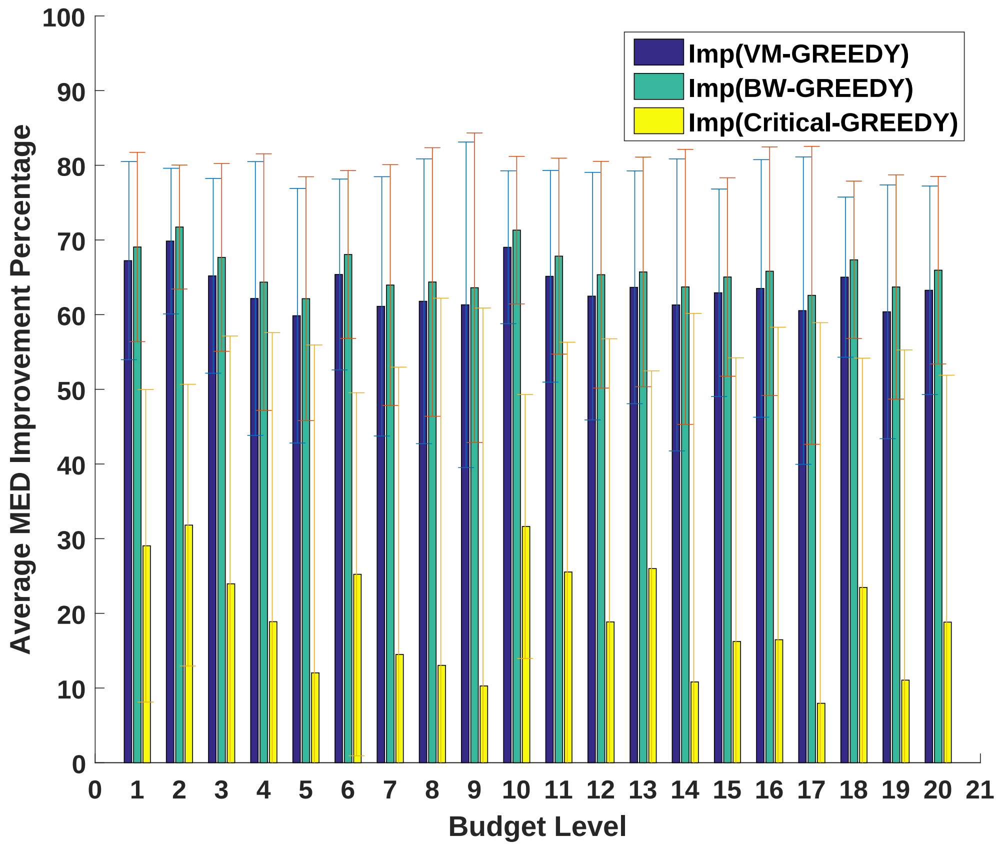


Figure 5.16 The average performance improvements (%) with standard deviations across 20 instances at each budget level for the Sipht workflow.

time, we run each module on each VM instance for multiple times and measure the corresponding execution time. We observe that the module execution time remains relatively stable on the same type of VM. Based on these performance measurements, we construct the execution time matrix T_E in unit of milliseconds in Table 5.7 and execution cost matrix C_E in US cents in Table 5.8.

Since the adjacent modules along an execution path in the workflow always have execution precedence constraints (i.e., execution dependencies), their execution times do not overlap. Therefore, when such adjacent modules are mapped to the same VM instance, we execute them one at a time in the workflow experiments. The

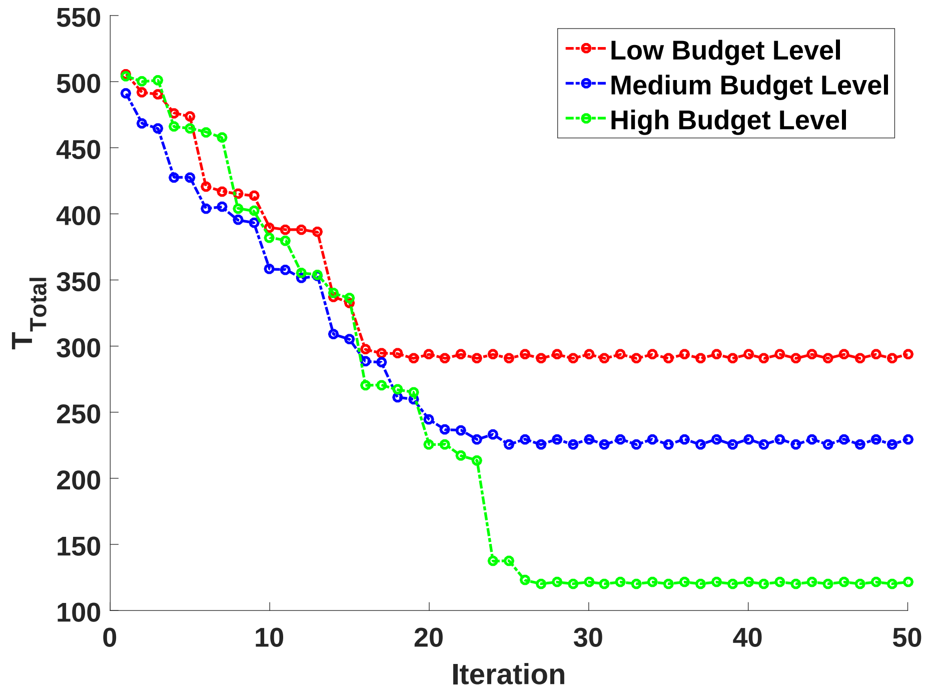


Figure 5.17 The optimization process of ICPWM running the problem instance of Index 15 under three different budget levels.

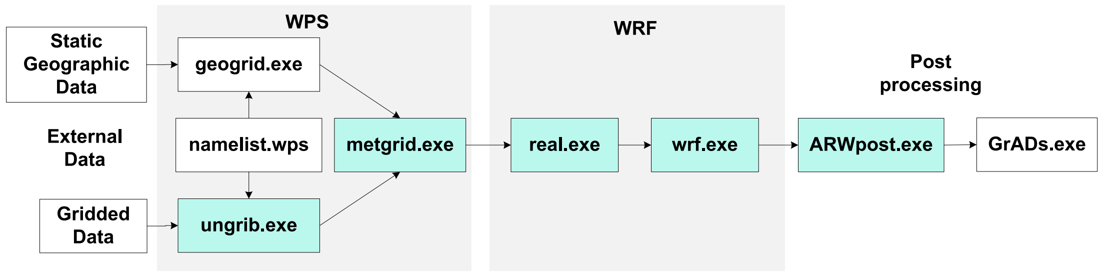


Figure 5.18 A general structure of the executable WRF workflow.

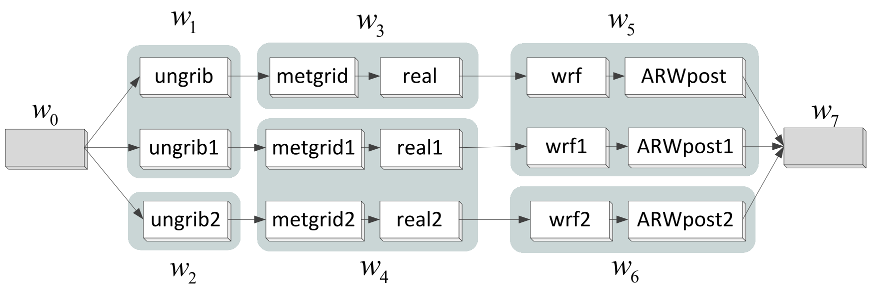
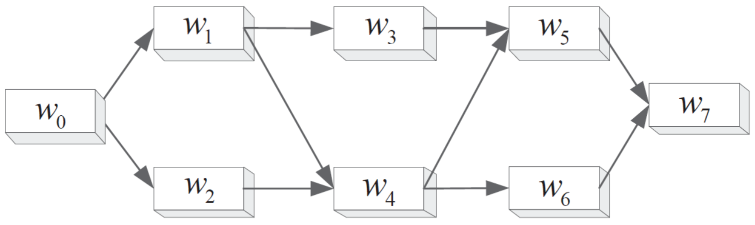


Figure 5.19 The WRF workflow of three pipelines in the experiments.



$$\begin{tikzcd}
 & w_1 \arrow[r] \arrow[dr] & w_3 \arrow[r] & w_5 \arrow[dr] & \\
w_0 \arrow[ur] \arrow[dr] & & & & w_7 \\
 & w_2 \arrow[r] & w_4 \arrow[r] \arrow[ur] & w_6 \arrow[ur] &
\end{tikzcd}$$

Figure 5.20 The WRF workflow after grouping.

schedules generated by these algorithms and their corresponding MED measurements at different budget levels are tabulated in Table 5.9 and further plotted in Figure 5.21 for comparison. These performance measurements show that the proposed ICPWM algorithm outperforms the other three algorithms in comparison. In addition, we implement an optimal solution based on exhaustive search for these small-scale workflow instances and observe that ICPWM achieves the optimal results in most of the test cases.

Note that BW-GREEDY only considers bandwidth in its mapping process and assigns a workflow module to a VM instance with the minimum required resources. In these experiments, since every VM instance has sufficient resources to execute every module, the mapping scheme of BW-GREEDY does not change as the budget level increases. Also, the execution cost of the last two modules ($w5$ and $w6$) is much higher than that of the rest modules, the mapping scheme may not change until the budget has been increased significantly over multiple budget levels, which explains the stair-like performance shape, especially at lower budget levels.

We would like to point out that ICPWM considers different pricing models for inter-cloud data transfer. However, since the testbed in these experiments has a limited scope based on GCP and AWS, there is only one pricing model available between these two clouds. We believe that ICPWM would yield more performance gains over other algorithms in larger-scale experiments, as demonstrated by the simulations results.

Table 5.9 The Average MED Measurements (in Milliseconds) by ICPWM, VM-GREEDY, BW-GREEDY, CRITICAL-GREEDY and Optimal Solution at Difference Budget Levels Using the WRF Workflow in GCP-AWS Environments

Budget Level	ICPWM	VM-GREEDY	BW-GREEDY	CRITICAL-GREEDY	OPT
1	1623871	1623871	2329125	1623871	1623871
2	812881	1623871	2329125	1623871	812881
3	812881	1623871	2329125	1623871	812881
4	812881	1623871	2329125	1623871	812881
5	812881	1623871	2329125	1623871	812881
6	812881	1623871	2329125	812881	812881
7	812881	812881	2329125	812881	812881
8	100031	812881	2329125	100031	100031
9	99750	100031	2329125	100031	99750
10	97431	100031	2329125	99750	97431
11	97431	100031	2329125	99750	97431
12	97431	100031	2329125	99750	97431
13	58165	97431	2329125	97431	58156
14	58156	97431	2329125	58156	58156
15	53412	58165	2329125	58156	53412
16	53412	53412	2329125	53412	53412
17	53412	53412	2329125	53412	53412
18	53412	53412	2329125	53412	53412
19	53412	53412	2329125	53412	53412
20	53412	53412	2329125	53412	53412

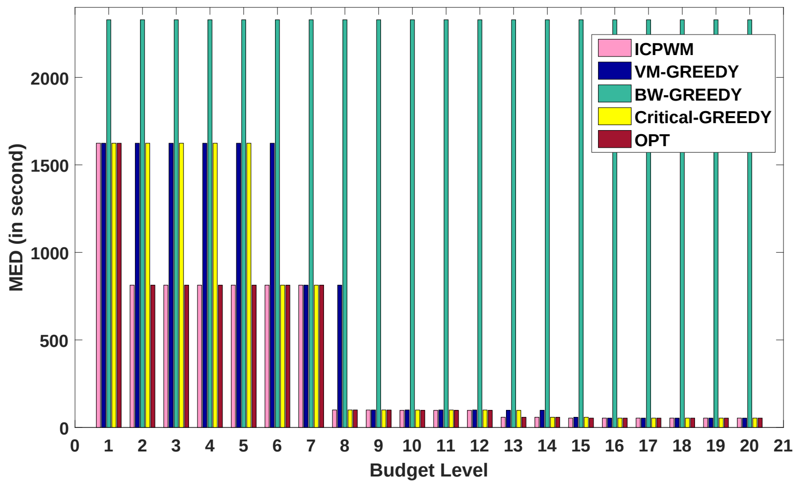


Figure 5.21 The MED comparison (in seconds) between ICPWM, VM-GREEDY, BW-GREEDY, CRITICAL-GREEDY and optimal solution at different budget levels using the WRF experiment workflow in GCP-AWS environments.

CHAPTER 6

MAPREDUCE-BASED WORKFLOWS

6.1 Problem Formulation

6.1.1 Time and Financial Cost Models

In addition to the common cost model provided in Chapter 3, we construct a set of additional cost models specific to the execution of MapReduce workflows.

Time Cost Models

- Time Cost of Data Transfer

A MapReduce job has three phases of data processing: Map, data copying from Map tasks to Reduce tasks, and Reduce [59]. Generally, the MapReduce process for scientific computing is both data- and compute-intensive, and hence both I/O and CPU time must be considered. In a real Hadoop system, the Map tasks from a MapReduce job may not fit at once in the system memory and hence may be processed on a round basis. Each MapReduce job in the workflow has an input data size DS_i and an output data size DS_o. Generally, DS_i is evenly split and processed by a number n_M of Map tasks, followed by a number n_R of Reduce tasks. By default, each split is a data block in Hadoop Distributed File System (HDFS). The input data size of a Reduce task is subject to the (key, value) distribution. We only need to consider the Reduce task with the largest input since all Reduce tasks start running at the same time. We use T_M and T_R to denote the total time in the Map and Reduce phase, respectively, and use T_{CP} to denote the total data copying time over network from Map tasks to Reduce tasks. The makespan T_E of a MapReduce job is upper limited by:

$$T_E \leqslant T_M + T_{CP} + T_R. \tag{6.1}$$

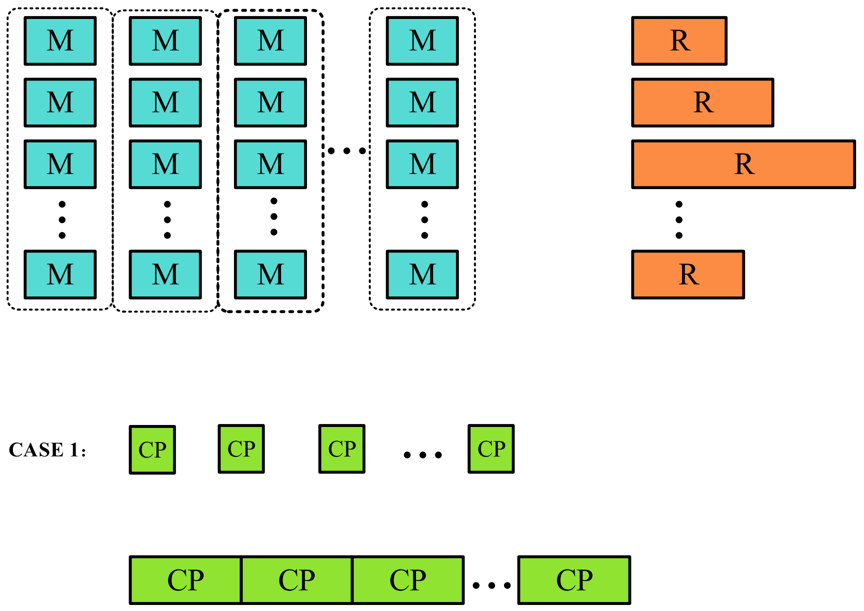


Figure 6.1 A single MapReduce job execution.

Each Map task contains two time cost components: CPU time and I/O time. There are two phases of I/O: it first performs sequential reading to load data from disk to memory, and then spills data from memory to disk after each Map task is completed. The time cost $T_{singleM}$ of a single Map task can be calculated as:

$$T_{singleM} = \frac{DS_i}{n_M \cdot r'_{Mslot}} + \frac{(1+\gamma) \times \frac{DS_i}{n_M}}{r'_{I/O}}, \tag{6.2}$$

where $r'_{Mslot} = \frac{N'_c \cdot f'_{CPU}}{S_m(vt)}$ is the CPU processing speed of each Map slot, and γ denotes its output/input ratio. As illustrated in Figure 6.1, the total time cost of the Map phase is calculated as:

$$T_M = T_{singleM} \times \frac{n_M}{S_m(vt) \cdot n(vt)}, \tag{6.3}$$

where $n(vt)$ is the number of VM instances of the same VM type vt allocated to the workflow user in the cloud.

The time cost $T_{singleCP}$ for copying the output of a single Map task over network to n_R Reduce tasks in the same cloud is calculated as:

$$T_{singleCP} = \frac{\frac{\gamma \cdot DS_i}{n_M}}{BW_{up}}. \tag{6.4}$$

Note that during the shuffling phase, we monitor the data transfer from mappers to reducers upon the completion of mappers in order to account for the sharing of BW_{up}. However, since this is a dynamic process measured in real time, there is no explicit analytical form for such measurement-based bandwidth sharing during this phase. If the corresponding reducer is determined for each mapper, we may be able to estimate bandwidth sharing in advance. during shuffling phase.

The total time cost T_{CP} of the data copying phase is calculated as:

$$T_{CP} = \frac{n_M}{S_m(vt) \cdot n(vt)} \cdot T_{singleCP}. \tag{6.5}$$

In the Reduce phase, we consider a random distribution of the (key, value) pairs in the input file (i.e., the output file of a Map task). Let C_r^i denote the input data size of Reduce task i. Thus, according to the Central Limit Theorem [35], C_r^i follows a normal distribution $N \sim (\mu, \sigma)$, $i = 1, 2, ..., b$, where μ is determined by $DS_i \cdot \gamma$. As per the "Three-Sigma" rule [35], the largest input data size of a Reduce task is:

$$C_r^* = \frac{DS_i \cdot \gamma}{b} + 3\sigma. \tag{6.6}$$

Therefore, the time cost of the Reduce phase is :

$$T_R = \frac{C_r^*}{r'_{Rslot}} + \frac{C_r^*}{r'_{I/O}}, \tag{6.7}$$

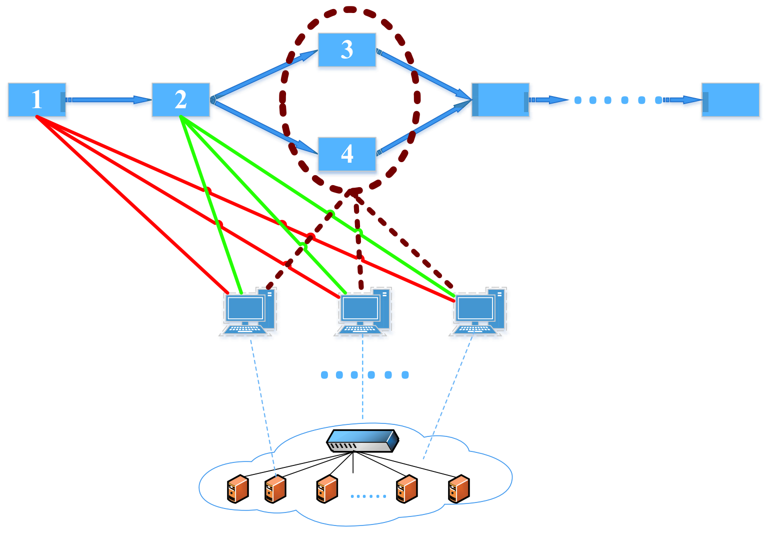


Figure 6.2 Execution of MapReduce jobs in the same cloud.

where $r'_{Rslot} = \frac{N'_c \cdot f'_{CPU}}{S_r(vt)}$ is the CPU speed of a Reduce slot.

The makespan T_E of a single MapReduce job running in a mapped cloud is:

$$T_E = \begin{cases} T_M + T_{singleCP} + T_R, \text{ if } T_{singleM} \geqslant T_{singleCP}, \\ T_{singleM} + T_{CP} + T_R, \text{ if } T_{singleM} < T_{singleCP}. \end{cases} \tag{6.8}$$

- Concurrent Execution of MapReduce Jobs

As shown in Figure 6.2, modules with execution dependencies are executed sequentially, such as Module 1 and Module 2. However, modules that are independent of each other are placed in a First-In-First-Out $(FIFO)$ ready queue Q, such as Module 3 and Module 4. If Module 3 is ahead of Module 4, during the Map phase of Module 3, whenever there is an empty Map slot available, Module 4 can use that slot to run its Map task, which means that the Map phases of independent Modules 3 and 4 may partially overlap.

- Start and Finish Time of a MapReduce Module

We calculate the start time T_s and the finish time T_f of the i-th MapReduce module in the workflow as

$$T_f^i = T_s^i + T_{MC}^i + T_R^i + T_{WR}^i, \tag{6.9}$$

$$T_s^i = \begin{cases} 0, & i = 0. \\ \max(\max\limits_{(j,i)\in E_w}(T_f^j + T_{dt}(j,i)), t_{MA}^{PreQ(i)}), & \text{otherwise.} \end{cases} \tag{6.10}$$

$$t_{MA}^i = \begin{cases} \frac{n_M^i}{S_m(vt)\cdot n(vt)} \cdot T_{singleM}^i + T_s^i, & i = Q[0], \\ t_{MA}^{PreQ(i)} + \frac{n_M^i}{S_m(vt)\cdot n(vt)} \cdot T_{singleM}^i, & i \neq Q[0], S_{Avail}^{PreQ(i)} = 0, \\ t_{MA}^{PreQ(i)} + \frac{n_M^i - S_{Avail}^{PreQ(i)}}{S_m(vt)\cdot n(vt)} + 1 \cdot T_{singleM}^i, & \text{otherwise.} \end{cases} \tag{6.11}$$

$$S_{Avail}^i = S_m(vt)\cdot n(vt) - \begin{cases} n_M^i \;\%\; (S_m(vt)\cdot n(vt)), & i = Q[0], \\ (n_M^i - S_{Avail}^{PreQ(i)}) \;\%\; (S_m(vt)\cdot n(vt)), & \text{otherwise.} \end{cases} \tag{6.12}$$

$$T_{MC}^i = \begin{cases} T_{singleM}^i + T_{CP}^i, & T_{singleM}^i > T_{singleCP}^i \\ T_M^i + T_{singleCP}^i, & \text{otherwise.} \end{cases} \tag{6.13}$$

$$T_{WR}^i = \begin{cases} 0, & T_f^{PreQ(i)} < T_s^i + T_{MC}^i \\ T_f^{PreQ(i)} - T_s^i - T_{MC}^i, & \text{otherwise} \end{cases} \tag{6.14}$$

Here, we use $PreQ(i)$ to denote the index of the module preceding the i-th module in the ready queue Q. As illustrated in Figure 6.3, T_{MC}^i denotes the time cost of the Map and data copying phases of the i-th MapReduce module; T_{WR}^i denotes the time the i-th MapReduce module spends on waiting for $PreQ(i)$ to finish its Reduce phase; and $t_{MA}^{PreQ(i)}$ denotes the time point when Map slots are made available, either

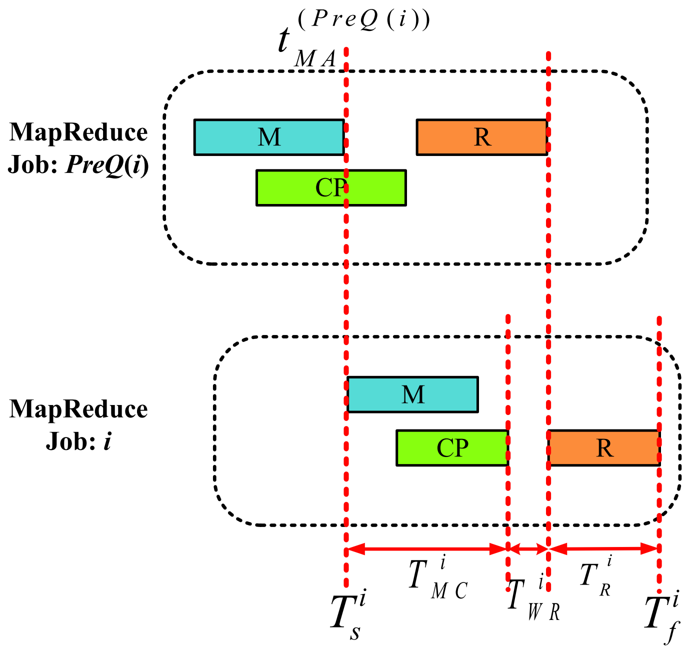


Figure 6.3 Illustration of the start time and finish time of a MapReduce module in a ready queue.

at the completion time of module $PreQ(i)$, or at the beginning of executing the last round of Map tasks of module $PreQ(i)$, as illustrated in Figure 6.4.

- Workflow Makespan

The makespan MS of workflow G_w is determined by the finish time of the last module and the start time of the first module in the workflow, and is the sum of module execution time T_E defined in Equation. 6.8, module waiting time T_{wait}, and data transfer time T_{dt} defined in Equation. 3.2, on the critical path (CP):

$$MS = \sum_{\text{for all } w \in CP} (T_E(w) + T_{wait}(w)) + \sum_{\text{for all } e_w \in CP} (T_{dt}(e_w)). \tag{6.15}$$

Note that the waiting time T_{wait} of a critical module is the amount of time it spends in waiting for the inputs from all of its non-critical preceding modules.

Financial Cost Models The total financial cost C_{Total} of an entire workflow execution process is calculated as

$$C_{Total} = C_{Total}(dt) + C_{Total}(VM), \tag{6.16}$$

$$C_{Total}(dt) = \sum_{\text{for all } e_w \in E_w} DS(e_w, l_{ic}) \cdot p(dt). \tag{6.17}$$

$$C_{Total}(VM) = \sum_{\text{for all } w \in V_{mc}} p(vt) \cdot n(vt) \cdot (\max_{i \in V_w} T_f^i - \min_{i \in V_w} T_s^i + T_{SU} + T_{rd}), \tag{6.18}$$

Here, $DS(\cdot)$ is the data size of edge e_w over an inter-cloud link l_{ic}, and $C_{Total}(dt)$ and $C_{Total}(VM)$ are the financial cost spent on all inter-cloud data transfers and on the use of all VMs, respectively. Again, $p(dt)$ is the volume-based pricing model for inter-cloud transfer. In $C_{Total}(VM)$, T_{SU} denotes the VM's startup time and T_{rd} denotes the time the VM spends in receiving all input data from its preceding modules.

6.1.2 Problem Formulation

We define a MapReduce workflow mapping scheme $\mathcal{M}$ as

$$\mathcal{M} : v_w \to VM(PM, vt), \text{ for all } w \in V_w, \tag{6.19}$$

where $VM(PM, vt)$ represents a VM instance of type vt provisioned on a PM.

Based on the above mathematical models, we formulate a budget-constrained MapReduce workflow mapping problem for minimum makespan in multi-cloud environments, referred to as MinMRW-MC, as follows.

Definition 3. Given a DAG-structured workflow graph $G_w(V_w, E_w)$, a complete weighted multi-cloud network graph $G_{mc}(V_{mc}, E_{mc})$, a set VT of available VM types,

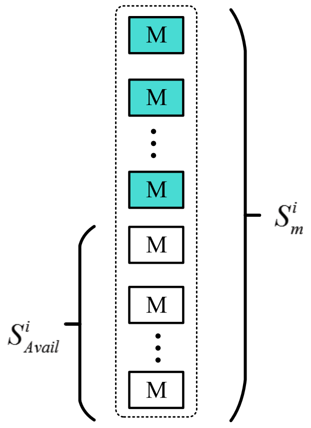


Figure 6.4 The last round of Map tasks of the i-th Module with S^i_{Avail} Map slots ($S^i_m = S_m(vt) \cdot n(vt)$).

and a fixed financial budget B, we wish to find a MapReduce workflow mapping scheme $\mathcal{M}$ to achieve the minimum makespan:

$$MinMS = \min_{all\,possible\,\mathcal{M}} MS, \tag{6.20}$$

while satisfying the following financial constraint:

$$C_{Total} \leq B. \tag{6.21}$$

The formulated MinMRW-MC problem considers inter-cloud data transfers and I/O operations for big data workflows in multi-cloud environments. This is a generalized version of the MED-CC problem in [40], which only considers the execution time of CPU-bound serial modules in a single cloud. Since MED-CC, which is a special case of MinMRW-MC, has been proved to be NP-complete and non-approximable, so is MinMRW-MC.

6.2 Algorithm Design

Our work targets network- and compute-intensive big data workflows deployed in multiple cloud-based data centers, where the inter-cloud data transfer time and financial cost could be significant and must be explicitly accounted for.

6.2.1 MapReduce Workflow Mapping Algorithm in Multi-Clouds

We design a MapReduce Workflow Mapping (MRWM) algorithm in multi-clouds to solve MinMRW-MC. The pseudocode of MRWM is provided in Algorithm. 8, which consists of the following key steps.

Step 1) We first sort the VM types in an increasing order according to the number of virtual cores. If there is a tie, we further consider memory size, CPU speed, I/O speed, and so on. Initially, the entire workflow is assigned to the worst VT in the worst cloud.

Step 2) Call Function $AssignVT()$ in Algorithm. 9, which in turn calls Function $CalculateTime$ in Algorithm. 10, whenever there exists a module that needs to be processed. We put this module in the ready queue $Q[]$ in its corresponding cloud when all of its preceding modules are finished. If there is only one module in the ready queue $Q[]$, we process this module right away and mark it as $finished$ as soon as it is done. We calculate the inter- or intra-cloud data transfer time from this module to its succeeding modules and update the $ReadyTime$ upon the receival of input data from one of its preceding modules. Once all of its preceding modules are marked as $finished$, we put this module in the $FIFO$ ready queue $Q[]$ in its corresponding cloud based on the last updated $ReadyTime$. Note that $ReadyTime$ is not necessarily the actual $StartTime$, which depends on the scheduling order and the execution status of those modules ahead of it in the ready queue. After processing the modules in $Q[]$, we are able to determine the actual $StartTime$ and $FinishTime$ of each module. The makespan is then calculated as the finish time of the last module in the workflow.

Step 3) After calculating the makespan, call Function $FindCP()$ in Algorithm. 11 to compute the critical path CP.

Step 4) If all the critical modules on CP are assigned to the same VT, we upgrade the VT of the critical modules along with other modules using the same VT by one level. If the upgraded VT is provided in a better cloud, it means that all the modules are processed in that particular cloud; otherwise, we identify the best VT among the critical modules and upgrade those critical modules using a worse VT and those non-critical modules using the same VT to the best VT within the budget B. If there is budget left, we upgrade the VT of the rest modules by one level until the budget is exhausted. We provision as many VMs as possible on the same PM.

Step 5) Go back to Step 2), and repeat the above process for a certain number of iterations and record the minimum makespan $MinMS$.

The time complexity of each iteration in MRWM is $O(\max(m \cdot |E_w|, m \cdot n) + \max(|E_w|^2, |V_{mc}|^2 \cdot |E_w| \cdot m))$, where m is the number of modules in the workflow, n is the number of VM types, and $|V_{mc}|$ is the number of clouds.

Algorithm 8: **MRWM**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a set of available VM types $VT[]$, the input data size of each module DS_i, a fixed financial budget B, and the number ITR of iterations.

Output: the minimum workflow's makespan $MinMS$.

```
1: sort the VM types $VT[]$ in an increasing order according to the number of
   virtual cores, memory size, $CPU$ speed, and $I/O$ speed;
2: $MinMS = INF$;
3: for all (module $w_i \in V_w$) do
4:     $AssignedVT[i]$ = the worst $VT[]$;
5: while ($ITR-- \geq 0$) do
6:     $[AssignedVT[], MS] = AssignVT(G_w, VT[], DS_i, B)$;
7:     if ($MS < MinMS$) then
8:         $MinMS = MS$;
9: return $MinMS$;
```

Algorithm 9: **AssignVT**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, a set of available VM types $VT[]$, the input data size of each module DS_i, and a fixed financial budget B.

Output: the VM types $AssignedVT[]$ that each MapReduce module is assigned to and the makespan MS.

```
1:  $MS = CalculateTime(G_w, AssignedVT[], DS_i)$;
2:  $CP[] = FindCP(G_w, AssignedVT[], DS_i)$;
3:  calculate the total financial cost $C_{Total}$;
4:  for all (module $w_j \in CP[]$) do
5:      if (all modules on $CP$ have the same $VT[j]$ in the same cloud) then
6:          upgrade $AssignedVT[j]$ and non-critical modules
            with the same $VT[j]$ by one level;
7:          if ($C_{Total} > B$) then
8:              downgrade $AssignedVT[j]$ and non-critical
                modules with the same $VT[j]$ by one level;
9:              break;
10:     else
11:         while ($AssignedVT[j]$ is worse than the best $VT[]$
            on $CP$) do
12:             upgrade $AssignedVT[j]$ and non-critical modules with the same $VT[j]$ to the best $VT[]$;
13:             if ($C(Total) > B$) then
14:                 downgrade $AssignedVT[j]$ and non-critical modules with the same $VT[j]$ by one level;
15:                 break;
16: for all (module $w_k \notin CP[]$ not yet upgraded) do
17:     upgrade $AssignedVT[k]$ and modules with the same
        $VT[]$ by one level;
18:     if ($C_{Total} > B$) then
19:         downgrade $AssignedVT[k]$ and modules with the
            same $VT[]$ by one level;
20: return [$AssignedVT[], MS$];
```

Algorithm 10: **CalculateTime**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, the VM types $AssignedVT[]$ of each module, and the input data size of each module DS_i.

Output: makespan MS.

```
1: while (there exists a module that needs to be executed) do
2:    for all (module $w_k$ that need to be executed) do
3:       if (all of $w_k$'s preceding modules are marked as $finished$) then
4:          add $w_k$ to the ready queues $Q[]$ in its corresponding cloud;
5:    for ($Q[]$ of each cloud) do
6:       if (there is only one module in $Q[]$) then
7:          execute it right away and mark it as $finished$ once done;
8:          update the $ReadyTime$ of its succeeding modules;
9:       else
10:         sort $Q[]$ by $FIFO$ according to their last updated $ReadyTime$; calculate
            the actual $StartTime$ and $FinishTime$ of each module in $Q[]$ using
            Eqs. 6.9 and 6.10, and mark it as $finished$ once it is done;
11:         update the $ReadyTime$ of its succeeding modules;
12: return makespan $MS$.
```

Algorithm 11: **FindCP**

Input: a DAG-structured workflow graph $G_w(V_w, E_w)$, the VM type $AssignedVT[]$ of each module, and input data size DS_i of each module.

Output: the critical path $CP[]$.

```
1: $w_j$ = the last executed module in $G_w$;
2: while ($w_j \neq$ the first executed module in $G_w$) do
3:     add $w_j$ into $CP[]$;
4:     $w_j$ = the last preceding module that updates $w_j$'s ready time;
5: add $w_j$ into $CP[]$;
6: return $CP[]$;
```

6.3 Algorithm Implementation and Performance Evaluation

6.3.1 Simulation Results

Simulation Settings We implement MRWM on the CloudSim platform and evaluate its performance in comparison with three algorithms, i.e., VM-GREEDY, MCMA [53], and CRITICAL-GREEDY [40]. VM-GREEDY, which is greedy on VM optimization, assigns as many high-end computers as possible within the budget. If the cost (including data transfer cost) is over the budget, it downgrades the VM type of the modules until the budget is satisfied. MCMA, as proposed in [53] to solve BCMED-CC, determines the VM type for each MapReduce module. CRITICAL-GREEDY, as proposed in [40] to solve MED-CC, determines the VM type for each module. Since it does not consider data transfer cost, we also need to downgrade the VM type of each module until the budget is satisfied.

For performance evaluation, we consider different problem sizes from small to large scales in a multi-cloud environment consisting of five clouds, where the inter-cloud data transfer takes time and incurs financial cost based on our cost models.

The problem size is defined as a 2-tuple $(m, |E_w|)$, where m is the number of workflow modules, and $|E_w|$ is the number of workflow links. We generate workflow instances of different scales in a random manner as follows: i) lay out all m modules sequentially along a pipeline, each of which is assigned a workload randomly generated within range $[5, 500]$, which represents the total number of instructions; ii) for each module, add an input edge from a randomly selected preceding module and add an output edge to a randomly selected succeeding module (note that the first module only needs output and the last module only needs input); iii) randomly select two modules from the pipeline and add a directed edge between them (from left to right) until we reach the given number of edges.

We compare MRWM with VM-GREEDY, MCMA [53] and CRITICAL-GREEDY [40], in terms of the workflow makespan under the same budget constraint. The MinMS performance improvement of MRWM over the other algorithms in comparison is defined as:

$$Imp(Other) = \frac{MinMS_{Other} - MinMS_{MRWM}}{MinMS_{Other}} \times 100\%,$$

where $MinMS_{Other}$ is the MinMS achieved by the others, i.e., VM-GREEDY, MCMA, and CRITICAL-GREEDY, and $MinMS_{MRWM}$ is the MinMS achieved by MRWM.

Comparison with Optimal Solutions We compare MRWM with optimal solutions in three small-scale problems of size $(5, 6), (5, 7)$, and $(6, 8)$. For each problem size, we randomly generate 50 problem instances with different module workloads and DAG topologies. In each problem instance, we specify 20 different budget levels. We run all these four algorithms on these instances and compare the MinMS results with the optimal ones computed by an exhaustive search-based

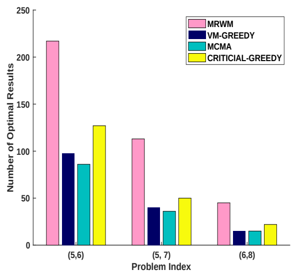


Figure 6.5 The number of optimal results among 1000 instances (50 workflow instances × 20 budget levels) produced by MRWM, VM-GREEDY, MCMA, and CRITICAL-GREEDY under different problem sizes.

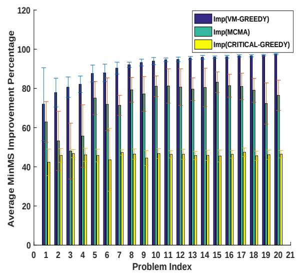


Figure 6.6 The average MinMS performance improvement percentages (%) with standard deviations across 400 instances (20 budget levels × 20 random workflow instances) under 20 problem sizes.

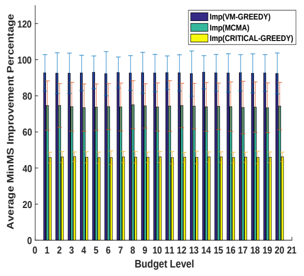


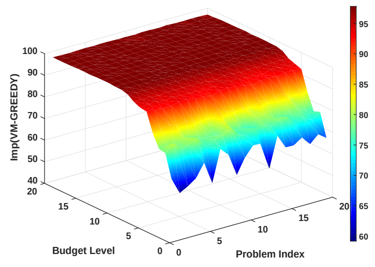
Figure 6.7 The average MinMS performance improvement percentages (%) with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at 20 budget levels.

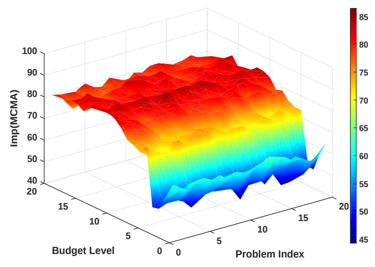
approach. Figure 6.5 shows the number of optimal results among 1000 instances (50 workflow instances × 20 budget levels) achieved by MRWM, VM-GREEDY, MCMA, and CRITICAL-GREEDY under different problem sizes. We observe that MRWM is more likely to achieve the optimality than the others in a statistical sense, which indicates the efficacy of MRWM. However, since these are small-scale problem instances, the absolute values of the differences from the optimal results are not significant.

Performance Comparison with Other Heuristics We further consider 20 problem sizes from small to large scales, indexed from 1 to 20. For each problem size, we randomly generate 20 problem instances, in each of which, we choose 20 budget levels with an equal interval of $\Delta B = (B_{max} - B_{min})/20$ within a certain budget range $[B_{min}, B_{max}]$. We provide in Table 6.1 the average percentage of MinMS improvement together with standard deviations achieved by MRWM over VM-GREEDY, MCMA, and CRITICAL-GREEDY across all the 20 budget levels, which are further plotted in Figure 6.6 for a visual comparison.

Also, for each of the 20 budget levels from low to high values, indexed from 1 to 20, we run the scheduling simulation by iterating through 20 problem sizes from small to large scales. We provide in Table 6.2 the average MinMS improvement percentages together with standard deviations achieved by MRWM over VM-GREEDY, MCMA, and CRITICAL-GREEDY across all the 20 problem sizes, which are further plotted in Figure 6.7 for a visual comparison. We set the budget levels to ensure that the largest problem instance has a feasible solution. Hence, the budget levels are sufficient especially for smaller problem instances, which explains the stable performance across different budget levels.

For a better illustration, we plot the overall performance improvement percentage of MRWM over VM-GREEDY, MCMA, and CRITICAL-GREEDY in Figures. 6.8, 6.9, and 6.10, respectively, where x axis denotes the budget increment across 20 levels and y axis denotes the index of 20 problem sizes from small to large scales. Each point (x, y, imp) in the $3D$ plot represents the average performance improvement across all 20 problem instances of the same problem size under the same budget level (the actual budget values may be different in different instances). These results show that MRWM achieves about 90%, 70%, and 45% performance improvement over VM-GREEDY, MCMA, and CRITICAL-GREEDY on average, respectively. Such performance improvements are considered significant for large-scale workflow execution. In these simulations, MRWM takes less than 2





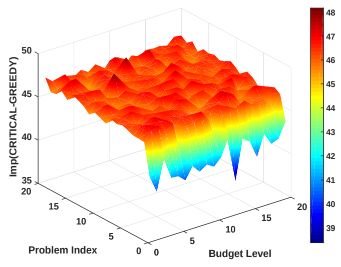


Figure 6.8 The overall performance improvement percentages (%) of MRWM over VM-GREEDY with different budget levels and problem sizes.

Figure 6.9 The overall performance improvement percentages (%) of MRWM over MCMA with different budget levels and problem sizes.

Figure 6.10 The overall performance improvement percentages (%) of MRWM over CRITICAL-GREEDY with different budget levels and problem sizes.

seconds to run on large-scale problem instances, which makes it possible to perform real-time scheduling of big data workflows.

Convergence of MRWM To investigate the convergence property of MRWM, we run this algorithm on the problem instance of Index 11 under three different budget levels, i.e., low, medium, and high. Note that the high budget level is used in the above simulations, the low budget level is 2/5 and the medium budget level is half of the high budget level. We plot the optimization process of MRWM in these three scenarios in Figure 7.9, which shows that MRWM converges to the minimum makespan in about 30 iterations. Hence, We use 40 iterations in the simulations and also later in the experiments to ensure a satisfactory mapping scheme.

6.3.2 Experimental Results

MapReduce Workflow For evaluation in real systems, we conduct workflow experiments to compute statistics on 22 years of global flight datasets of about 12GB

Table 6.1 The Average MinMS Improvement Percentages $\alpha = \text{Imp(VM-GREEDY)}$, $\beta = \text{Imp(MCMA)}$, and $\gamma = \text{Imp(CRITICAL-GREEDY)}$ and the Corresponding Standard Deviations Across 400 Instances (20 Budget Levels $\times$ 20 Random Workflow Instances) with Different Problem Sizes

Idx	$(m, \|E_w\|)$	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	$(5, 7)$	71.98	18.57	62.90	10.36	42.28	6.79
2	$(8, 14)$	77.86	7.36	53.23	15.10	45.80	3.54
3	$(10, 18)$	80.64	5.19	48.03	14.25	46.75	2.07
4	$(13, 30)$	82.10	4.22	55.69	15.97	46.09	3.36
5	$(15, 60)$	87.61	4.30	75.08	8.44	45.78	3.24
6	$(18, 80)$	87.94	4.45	71.90	13.48	43.58	16.06
7	$(20, 100)$	90.34	3.05	71.37	5.10	47.30	1.37
8	$(23, 150)$	92.10	1.31	79.27	6.32	46.49	2.63
9	$(25, 200)$	93.20	1.74	77.18	8.88	44.48	3.82
10	$(28, 250)$	93.95	1.80	81.13	5.23	46.77	2.51
11	$(30, 300)$	94.58	0.94	81.24	8.75	46.35	1.87
12	$(33, 380)$	94.80	1.15	80.67	9.34	46.40	2.55
13	$(35, 400)$	95.55	0.83	79.63	5.75	45.76	2.10
14	$(38, 450)$	95.92	0.94	80.49	9.82	45.84	2.61
15	$(40, 500)$	95.98	0.57	83.18	5.23	45.50	3.01
16	$(43, 550)$	96.17	0.73	81.41	5.83	46.37	1.93
17	$(45, 580)$	96.64	0.58	81.02	6.74	47.44	2.14
18	$(48, 600)$	96.66	0.57	79.06	6.12	45.62	2.41
19	$(50, 650)$	96.91	0.37	72.30	10.55	46.16	2.36
20	$(53, 700)$	97.46	0.31	76.48	7.74	46.44	1.89

Table 6.2 The Average MinMS Improvement Percentages $\alpha = \text{Imp(VM-GREEDY)}$, $\beta = \text{Imp(MCMA)}$, and $\gamma = \text{Imp(CRITICAL-GREEDY)}$ and the Corresponding Standard Deviations Across 400 Instances (20 problem sizes $\times$ 20 Random Workflow Instances) at Different Budget Levels

Budget Level	α (%)	StdDv	β (%)	StdDv	γ (%)	StdDv
1	92.70	10.11	74.62	13.63	45.81	2.95
2	92.53	11.31	74.63	12.18	46.12	2.96
3	92.51	11.14	73.96	13.54	46.27	2.65
4	92.65	9.82	73.43	13.13	45.96	3.08
5	93.00	9.08	73.69	12.84	45.75	3.10
6	92.24	12.27	74.00	12.76	45.77	3.76
7	92.90	8.59	73.80	13.28	46.11	3.08
8	92.61	9.62	75.04	13.31	46.00	3.11
9	92.77	11.35	74.40	12.56	45.97	2.89
10	92.63	10.39	73.80	13.33	46.14	3.02
11	92.81	9.28	74.32	14.07	45.73	3.38
12	92.72	10.01	74.61	12.33	45.98	2.73
13	92.31	12.51	74.29	12.63	45.87	3.45
14	93.05	9.17	73.86	13.52	46.15	2.77
15	92.68	10.23	74.16	12.89	46.11	2.87
16	92.63	10.63	73.97	13.75	45.79	2.99
17	92.79	10.01	73.50	14.58	46.04	2.76
18	92.46	10.79	73.71	13.99	45.89	3.43
19	92.64	10.17	73.34	13.80	45.94	2.96
20	92.37	11.35	74.30	13.18	46.20	2.61

Table 6.3 VM Instances Provisioned by Different Cloud Computing Platforms in the Experiment

VM Type	Platform	Instance Type	Availability Zone	Number of Instances
$VT1$	AWS	t2.small	us-west-2c	20
$VT2$	AWS	t2.medium	us-east-2a	20
$VT3$	GCP	n1-highmem-2	us-central1-a	12
$VT4$	GCP	n1-standard-4	us-east1-c	6
$VT5$	GCP	n1-standard-8	us-west1-a	3

Table 6.4 System Specifications of Different VM Types in the Experiment

VM Type	CPU (GHz)	RAM (GB)	Disk (GB)	I/O read (MB/s)	I/O write (MB/s)	Price (USD/min)
$VT1$	2.40×1	2	50	290	95	0.000433
$VT2$	2.39×2	4	50	918	253	0.000867
$VT3$	2.60×2	13	50	820	470	0.001167
$VT4$	2.30×4	20	50	851	579	0.002333
$VT5$	2.20×8	40	50	1024	585	0.004666

from 1987 to 2008 at Statistical Computing [14]. The workflow structure is shown in Figure 7.14, where every module is a MapReduce job: w_0 filters out the headline in each data file; w_1 calculates the frequency of each "Flight Cancellation Reason";

Experimental Settings We set up 3 homogeneous clusters of Hadoop 2.7.3 [20] in Google Cloud Platform (GCP) and 2 homogeneous clusters of Hadoop 2.7.3 in Amazon Web Services (AWS) [18]. Table 7.3 tabulates the VM instances provisioned in the availability zone of each platform, and Table 7.1 tabulates the system

Table 6.5 Bandwidth Matrix *BW*

BW (Mb/s)	$VT1$	$VT2$	$VT3$	$VT4$	$VT5$
$VT1$	—	16.7	33.3	20.0	33.3
$VT2$	3.2	—	16.7	11.1	7.7
$VT3$	8.5	33.3	—	50.0	50.0
$VT4$	4.2	14.3	50	—	25.0
$VT5$	10.0	7.7	50	25.0	—

Table 6.6 Data Transfer Price matrix *price*

price (USD/GB)	AWS	GCP
AWS	0.000	0.155
GCP	0.120	0.000

specification and pricing model (in US dollar/minute) of each VM type. In GCP, since each user is limited to 24 virtual cores in each availability zone, we set up 3 clusters as follows: 12 VM instances with 2 virtual cores, 6 VM instances with 4 virtual cores, and 3 VM instances with 8 virtual cores. In AWS [18], since there is a limit of 20 on the number of VM instances of each type we select in the experiments, we set up 2 clusters with 20 VM instances each. The data transfer throughput between each pair of VM instances of different VM types is estimated through active bandwidth measurements as shown in Table 6.5, and the volume-based data transfer pricing model under 1 TB is shown in Table 6.6.

Performance Comparison We run the statistical analysis workflow on three different scales of flight data: i) small scale from 1987 – 1994 (8 years of 3433MB), ii) medium scale from 1987 – 2002 (16 years of 7627MB), and iii) large scale from 1987 – 2008 (22 years of 11473MB). Under each scale, we choose 6 budget levels within the range $[B_{min}, B_{max}]$ at an equal interval of $\Delta B = (B_{max} - B_{min})/6$, where B_{min} is 20% more than the minimum budget to run the entire workflow on the worst cluster (i.e., VT1 on AWS), and B_{max} is 20% more than the maximum budget to run the entire workflow on the best cluster (i.e., VT5 on GCP). We plot the minimum makespan (MinMS) measurement produced by the proposed MRWM algorithm and the three algorithms in comparison across 6 budget levels on the three scales of datasets in Figures. 6.13, 6.14, and 6.15. These experimental results clearly illustrate the performance superiority of MRWM over the other three algorithms in real multi-cloud environments. At level 6, since there is more than sufficient budget (20% more than the maximum budget to run the entire workflow on the best cluster), all the algorithms achieve the optimal results. Since the problem is NP-complete and non-approximable, these algorithms are all heuristic in nature without performance guarantee. In very few cases, MRWM does not perform as well as others. Also, we observe about 15% discrepancy between our theoretical estimates and real-world experimental measurements, which validates the correctness of our models and also ensures accurate workflow mapping in real systems.

Performance Analysis The problem we tackle is a multi-cloud workflow mapping problem, where the inter- and intra-cloud data transfer plays a very important role in both financial cost and time cost. The performance superiority of MRWM comes from a careful design that follows two critical guidelines: i) reuse a cluster whenever it is possible to avoid inter- and intra-cloud data transfer, and ii) prioritize the upgrading of the VM selection for the modules on the critical path. VM-GREEDY neither considers inter-cloud data transfer nor handles critical modules; MCMA updates the VMs of critical modules to the best one and resets the VM selection at the beginning of each iteration, while MRWM updates those VMs gradually and records the VM selection from the previous iteration; CRITICAL-GREEDY only focuses on VM selection without considering inter-cloud data transfer.

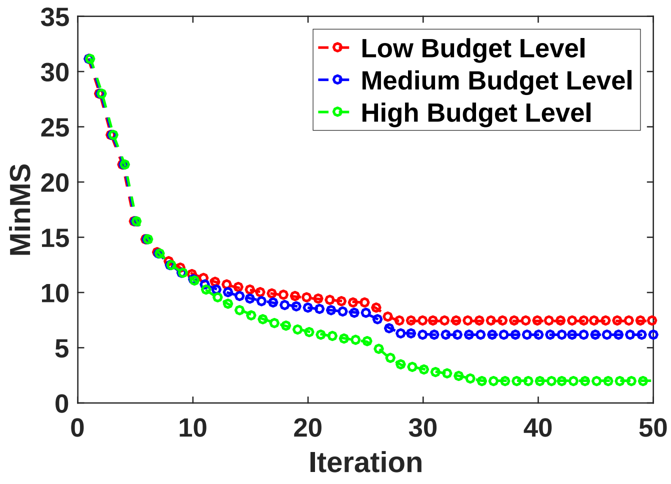


Figure 6.11 The optimization process of MRWM running the problem instance of Index 11 under three different budget levels.

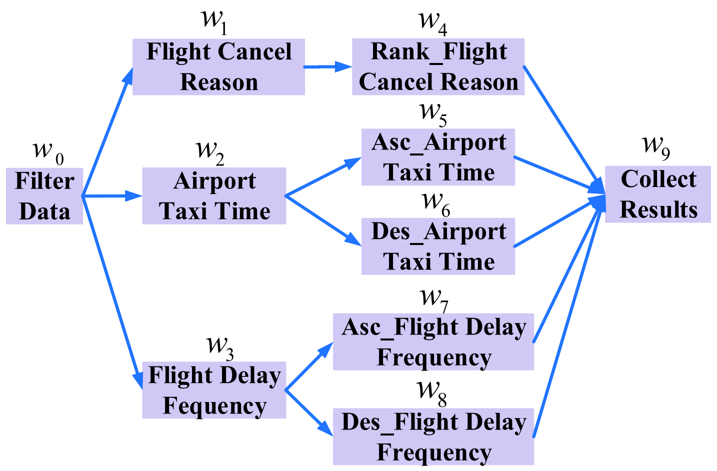

w_0 Filter Data
w_1 Flight Cancel Reason
w_2 Airport Taxi Time
w_3 Flight Delay Fequency
w_4 Rank_Flight Cancel Reason
w_5 Asc_Airport Taxi Time
w_6 Des_Airport Taxi Time
w_7 Asc_Flight Delay Frequency
w_8 Des_Flight Delay Frequency
w_9 Collect Results

Figure 6.12 The structure of the MapReduce workflow.

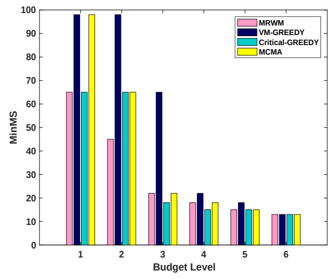


Figure 6.13 The MinMS performance in minutes across 6 budget levels on small-scale datasets.

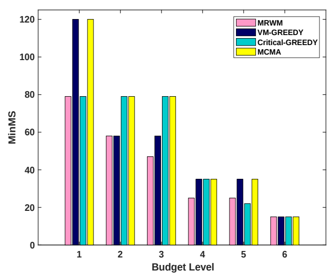


Figure 6.14 The MinMS performance in minutes across 6 budget levels on medium-scale datasets.

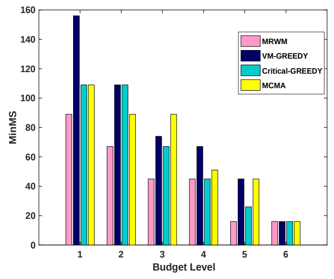


Figure 6.15 The MinMS performance in minutes across 6 budget levels on large-scale datasets.

CHAPTER 7

STORM-BASED STREAM DATA PROCESSING WORKFLOWS

7.1 Problem Formulation

In addition to the common cost model provided in Chapter 3, we construct a set of additional cost models specific to the execution of Storm-based workflows (topology) for streaming data processing.

We begin with the construction of cost models used for problem definition. We consider a Storm topology as a directed acyclic graph (DAG) $G_{tp}(V_{tp}, E_{tp})$ with $|V_{tp}|$ modules[1] and $|E_{tp}|$ edges, each of which represents the execution dependency and data movement between two neighbor modules. A Storm-based streaming application is executed in a heterogeneous cluster deployed in a cloud with n virtual machine (VM) types $VT = \{vt_1, vt_2, \cdots, vt_n\}$, for each, there may exist multiple VM instances. Each VM type vt has a set of performance attributes including CPU frequency $f_{CPU}(vt)$, number of virtual cores $nc(vt)$, and memory capacity $m(vt)$, as well as a commonly used "pay-as-you-go" VT pricing model $p(vt) = f(f_{CPU}(vt), nc(vt), m(vt))$, which determines the financial cost per time unit for using a VM instance of that type. Note that the actual computing or processing power of a given core is typically measured in unit of MIPS (million instructions per second). In this work, we consider a single cloud environment and the cost for data transfer is not accounted as in most real-life public clouds.

[1]We refer to the smallest computing entity in a general workflow (or more specifically, spout/bolt in a Storm topology) as a computing module, which represents either a serial computing task or a parallel processing job such as a typical MapReduce program in Hadoop.

We define a Storm topology mapping scheme $\mathscr{M}$ as

$$\mathscr{M} : v_{tp}(DoP) \rightarrow VM(vt), \text{for all } v_{tp} \in V_{tp}, \tag{7.1}$$

where $VM(vt)$ represents the VT selection and DoP represents the degree of parallelism for module v_{tp}, which denotes a spout or a bolt $Bolt$ in the Storm topology.

In Storm [24], a data stream is comprised of tuples. As a source of data streams in a topology, a spout reads tuples from an external source and emits them into the topology. A bolt represents a data processing unit in the topology, such as filtering, aggregation, join, communicating with databases, etc. Based on a pre-specified DoP, each spout or bolt executes multiple tasks concurrently across the cluster, each of which corresponds to one thread of execution, and stream groupings define how to send tuples from one set of tasks to another. Note that DoP determines the number of VM instances of vt selected for executing v_{tp}. In other words, for each v_{tp}, we create a number DoP of worker instances that are launched on different VMs. In this work, a single worker is created on each VM instance with a single executor to process one tuple of input data.

We define the gap time of module v_{tp} as the time interval between the finish time of two adjacent tuples processed by two copies of v_{tp}. Note that the gap time of each module may or may not be uniform during the entire streaming data processing. We have the following theorem on the pattern of the module gap time.

Theorem 1. *The gap time of any module v_{tp} in the Storm topology occurs periodically.*

Proof. We prove Theorem 1 by mathematical induction. In the base case, we analyze the first bolt $Bolt_1$ with $N_1 = m$ copies running in parallel. As shown in Figure 7.1, the gap time (i.e., t_1, t_2, etc.) between m workers of the first bolt in the topology

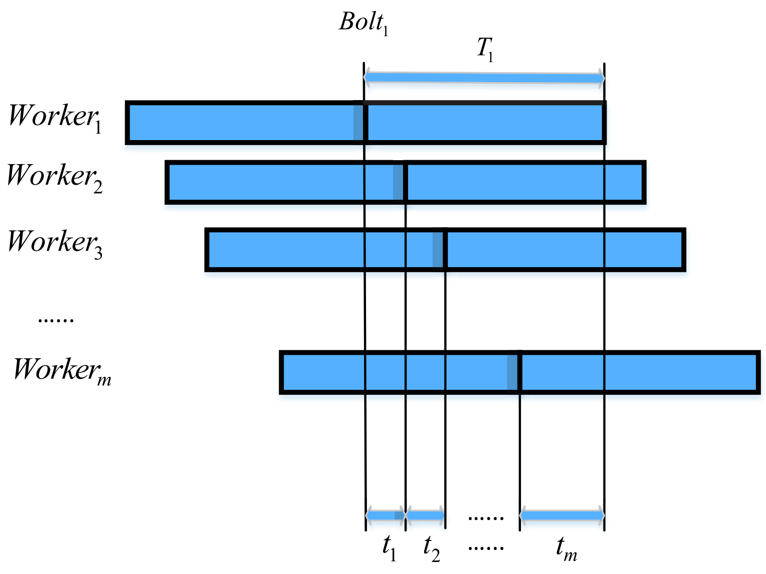


Figure 7.1 Execution dynamics of the first bolt of the topology with DoP $= m$, i.e., there are m concurrent workers executing the first bolt.

is the same as the time interval of two adjacent tuples emitted from the spout. T_1 denotes the execution time of one copy of $Bolt_1$. Obviously, the base case is established.

Suppose that $Bolt_i$'s gap time occurs periodically. Basically, there are three cases of $Bolt_{i+1}$:

- Case 1: when $N_{i+1} = N_i$, i.e., $Bolt_i$ and $Bolt_{i+1}$ have the same number of workers. Let T_i denote the execution time of a worker of $Bolt_i$ processing one tuple. Figure 7.2 shows the case when $T_i = T_{i+1}$, so for $Bolt_i$, worker j of $Bolt_{i+1}$ always has a delay of T_i after worker j of $Bolt_i$, where $j = 1, 2, \cdots, N_i$. Hence, $Bolt_{i+1}$ has the same cyclic pattern as $Bolt_i$. Figure 7.3 shows the case when $T_i > T_{i+1}$. Similar to Figure 7.2, there is still a delay of T_i between $Bolt_i$ and $Bolt_{i+1}$ on each corresponding worker, which means that $Bolt_{i+1}$ has the same cyclic pattern as $Bolt_i$. Figure 7.4 shows the case when $T_i < T_{i+1}$. Each

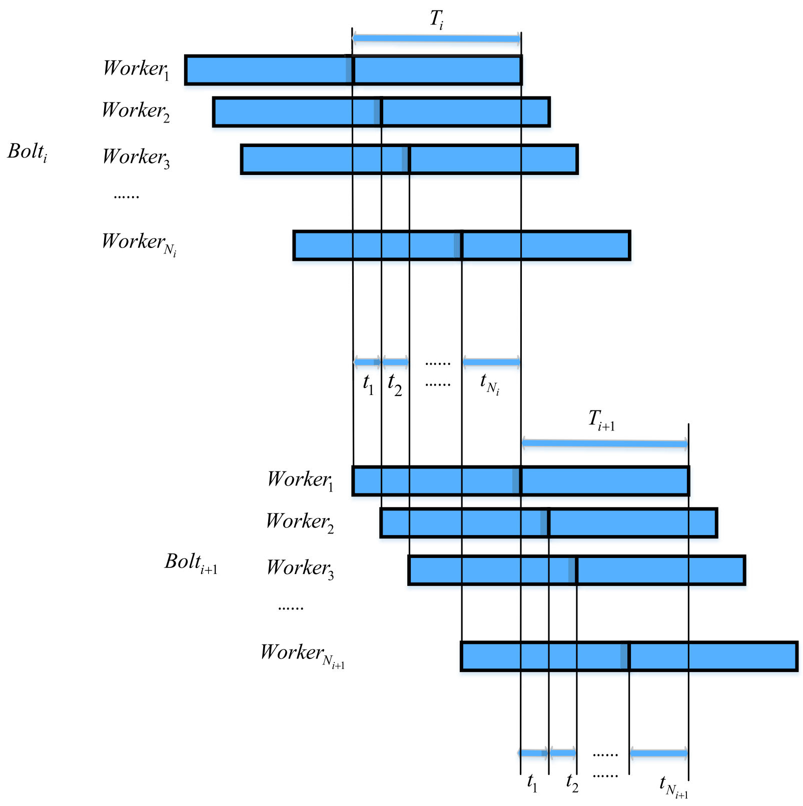


Figure 7.2 Execution dynamics in Case 1: the gap time when $T_i = T_{i+1}$.

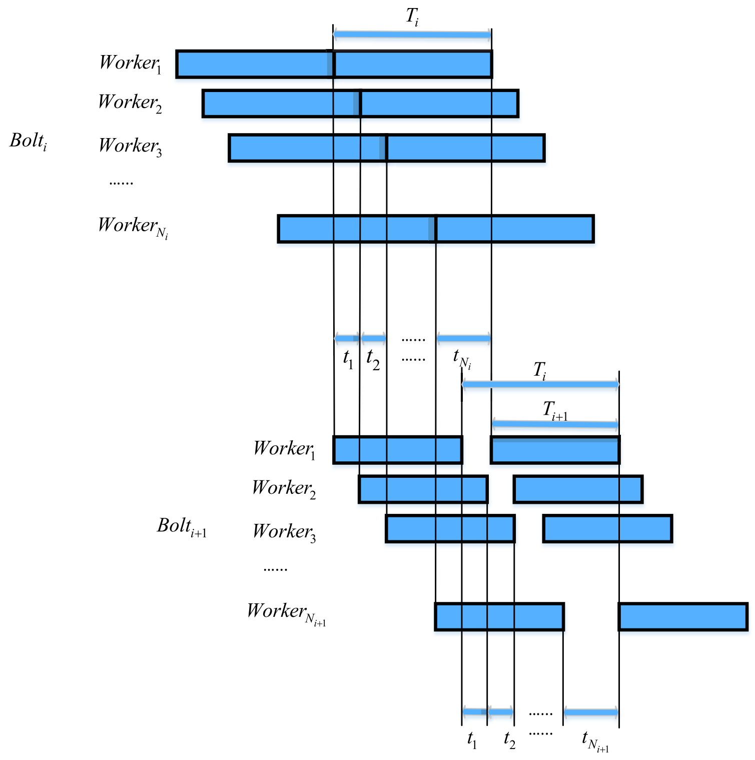


Figure 7.3 Execution dynamics in Case 1: the gap time when $T_i > T_{i+1}$.

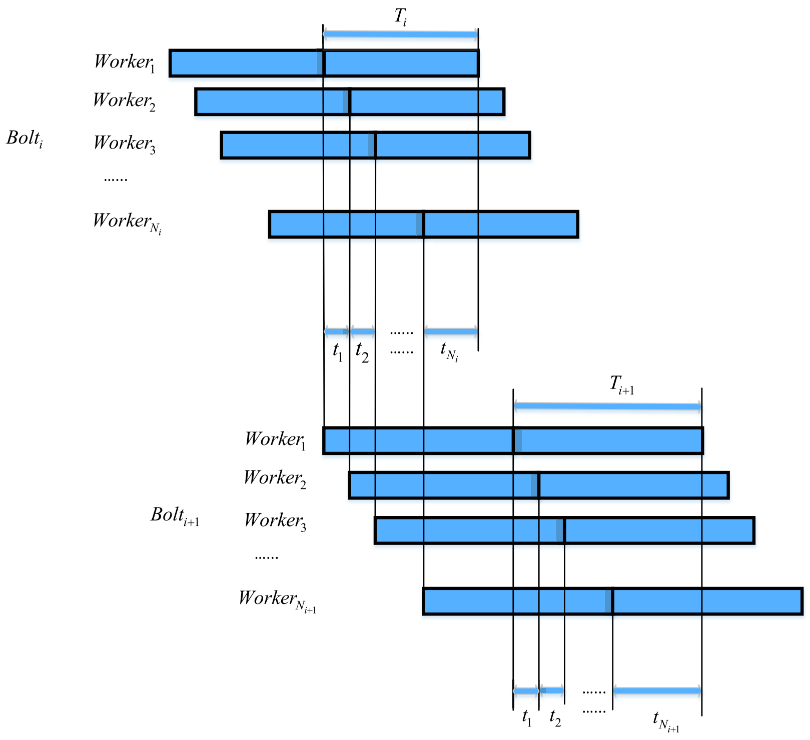
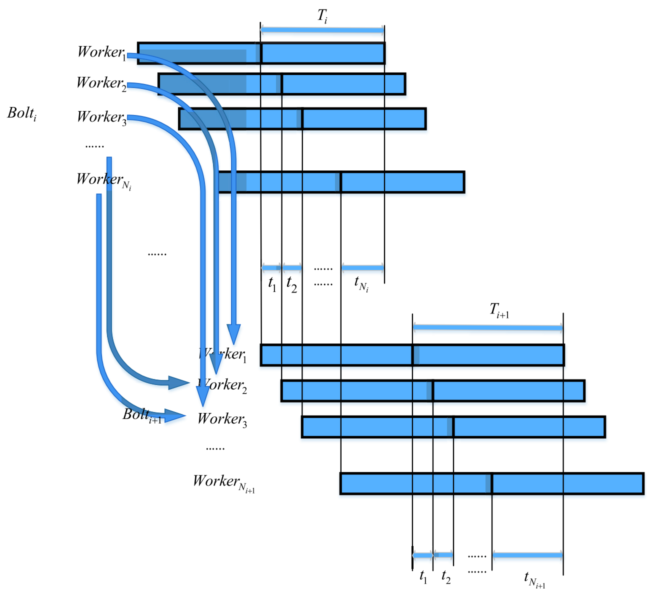
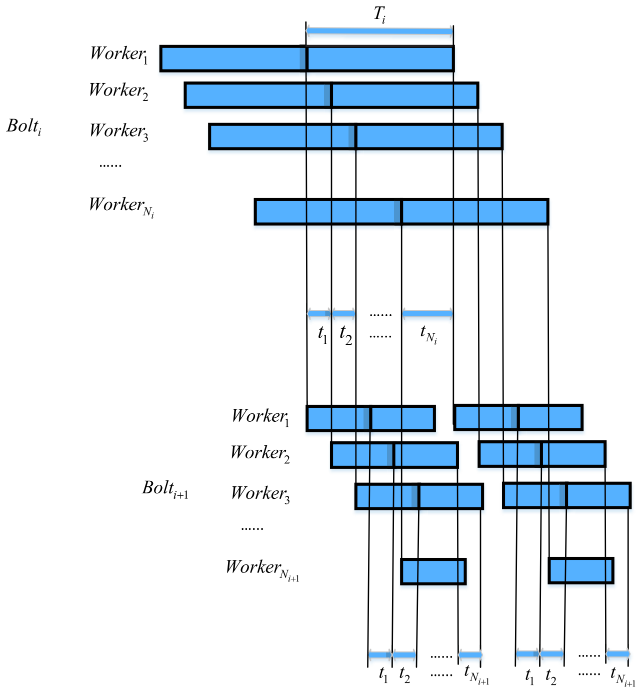


Figure 7.4 Execution dynamics in Case 1: the gap time when $T_i < T_{i+1}$.



(a) The gap time when $T_i < T_{i+1}$.



(b) The gap time when $T_i > T_{i+1}$.

Figure 7.5 Execution dynamics in Case 2.

corresponding worker has a delay of T_{i+1}. Since $T_i < T_{i+1}$, the gap time is different from that of $Bolt_i$. However, there is a one-to-one mapping between the finish time of $Bolt_i$ and $Bolt_{i+1}$, as well as the gap time of $Bolt_i$ and $Bolt_{i+1}$. Therefore, $Bolt_{i+1}$'s gap time sequence can be mapped to $Bolt_i$'s gap time sequence, and $Bolt_{i+1}$ should have the same cyclic pattern as $Bolt_i$.

- Case 2 when $N_i > N_{i+1}$. Figure 7.5(a) shows the case when $T_i < T_{i+1}$. We assign the k-th worker of $Bolt_{i+1}$ to process the next tuple emitted from the j-th worker of $Bolt_i$. There is a one-to-one mapping from the finish time of each tuple processed by $Bolt_i$ to the finish time of the same tuple processed by $Bolt_{i+1}$. Since the gap time of $Bolt_i$ has a cyclic pattern, so does the gap time of $Bolt_{i+1}$. Figure 7.5(b) shows the case when $T_i > T_{i+1}$, where the situation is similar to Figure 7.5(a). The only difference is that there may exist a certain waiting time between the first tuple's start time in the next cycle and the last tuple's finish time of each worker of $Bolt_{i+1}$. Since the tuple mapping and the corresponding delay time remain the same in each cycle, so the cyclic pattern carries on in $Bolt_{i+1}$. When $T_i = T_{i+1}$, it is obvious that $Bolt_{i+1}$ exhibits a cyclic pattern.
- Case 3 when $N_i < N_{i+1}$. The execution dynamics analysis is similar to Case 2 and hence is omitted.

Note that $Bolt_i$ may have multiple upstream bolts. Assume that there are n upstream bolts $Bolt_k$, where $k = i - n, ..., i - 2, i - 1$. Since the number of workers for each bolt may be different, we consider the lowest common multiple LCM_i of all N_k as the number of workers for each bolt. These n upstream bolts can be treated as a single virtual bolt with LCM_i workers. The j-th worker, $j = 1, 2, ..., LCM_i$, emits a tuple at the latest time when $Bolt_k$ emits the j-th tuple. After the transformation, based on the above case, we can prove that the gap time of any bolt has a cyclic

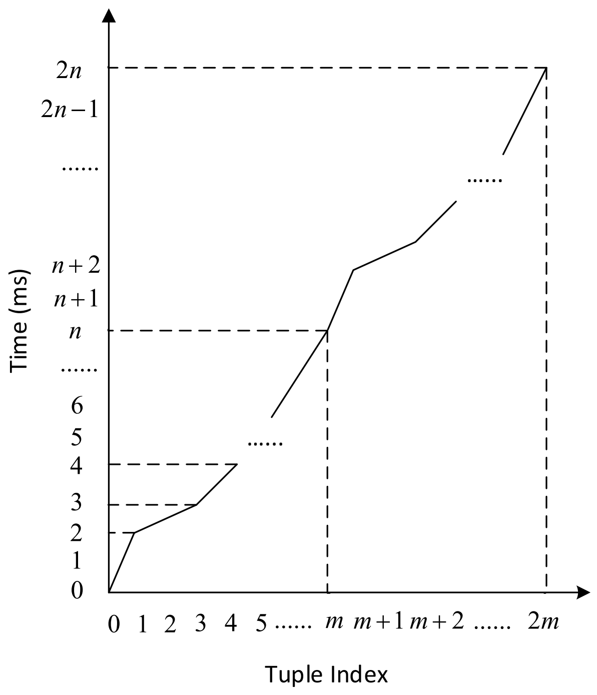


Figure 7.6 Illustration of gap time for throughput calculation.

pattern.

Proof ends. □

This cyclic pattern is critical to modeling the throughput of any module v_{tp}, which denotes either a spout or a bolt *Bolt* in the Storm topology. According to Theorem 1, we plot the relationship between tuple index and processing time for each tuple on module v_{tp} in Figure 7.6, which shows two cycles for illustration. To calculate throughput, we consider a period of time and the number of tuples processed during this period. Since the gap time of $Bolt_i$ has a cyclic pattern, we calculate the throughput by counting the number of tuples processed per cycle. The first cycle is from time 0 to n and the second one is from time $n+1$ to $2n$, where n is the end time of the first cycle in ms (time unit). Hence, the cycle time $CT_{v_{tp}} = n$.

In each cycle, v_{tp} processes m tuples, defined as tuple count per cycle $TCPC_{v_{tp}}$. We define the throughput $T(\mathscr{M}, v_{tp})$ of module v_{tp} under the mapping scheme of $\mathscr{M}$ as the inverse of the average processing time for each tuple during each cycle:

$$T(\mathscr{M}, v_{tp}) = \frac{1}{\frac{CT_{v_{tp}}}{TCPC_{v_{tp}}}} = \frac{TCPC_{v_{tp}}}{CT_{v_{tp}}}. \tag{7.2}$$

A bottleneck is a process in a chain of processes whose computing power limits the computing capacity of the whole execution chain, and may result in stalls in execution. A global bottleneck module is the one with the smallest $T(\mathscr{M}, v_{tp})$, and the throughput of the entire topology is determined by the bottleneck module's throughput, defined as:

$$GT(\mathscr{M}) = \min_{v_{tp} \in V_{tp}} T(\mathscr{M}, v_{tp}). \tag{7.3}$$

Based on the above mathematical models, we formulate a Storm Topology Mapping problem for maximum throughput in clouds under Budget Constraint, referred to as STM-BC, as follows.

Definition 4. Given a DAG-structured Storm topology $G_{tp}(V_{tp}, E_{tp})$, a set VT of available VM types, and a fixed financial budget b per time unit, we wish to find a topology mapping scheme $\mathscr{M}$ to achieve the Maximum Throughput (MT):

$$MT = \max_{\text{all possible } \mathscr{M}} MT(\mathscr{M}), \tag{7.4}$$

while satisfying the following budget constraint:

$$C \leq b, \tag{7.5}$$

where C is the total financial cost of VMs used for the Storm topology execution per time unit, calculated as

$$C = \sum_{VMs(vt) \text{ used in } \mathscr{M}} p(vt), \tag{7.6}$$

where v_{tp} is mapped to $VM(vt)$, for each $v_{tp} \in V_{tp}$.

The problem formulated above is a generalized version of the MFR-ANR problem in [28], which only considers a pipeline structured workflow without parallel computing for each module. Specifically, in MFR-ANR, the authors consider a linear computing pipeline consisting of a number of sequential modules and a computer network represented as a directed arbitrary graph. They aim to find a one-on-one mapping scheme between a module and a computing node to achieve maximum frame rate. Note that a pipeline is a special case of workflow, and one-on-one mapping does not allow parallel computing as one module must be processed exclusively by one computing node. In our work, we formulate STM-BC, which supports parallel computing since one module (either a spout or a bolt in Storm) can be processed by multiple workers. Since MFR-ANR, which is a special case of STM-BC, has been proved to be NP-complete and non-approximable, so is STM-BC. Hence, we focus on the design of heuristic solutions to our problem.

We would like to point out that our cost models can be adapted to other stream data processing platforms, such as Spark Streaming [3] workflows where each module in the workflow is a Spark Streaming job. Such adapted cost models can be used to find the mapping of Spark jobs in the workflow to a set of physical or virtual computing nodes.

7.2 Algorithm Design

We design a bottleneck-oriented topology mapping (BOTM) algorithm in Storm to solve STM-BC. BOTM determines not only the VT selection but also the degree of parallelism (DoP) for each module in the topology. The key idea is to iteratively identify the global bottleneck module and strategically compute an appropriate adjustment for this module's VT selection and degree of parallelism to achieve the maximum increase of the global workflow throughput. Note that the default scheduler in Storm assigns executors in a round-robin manner without considering the global bottleneck.

7.2.1 Bottleneck-Oriented Topology Mapping

The pseudocode of BOTM is provided in Algorithm.12, which consists of the following key steps.

Step 1) Sort the available VM types VT according to the total CPU frequency of all virtual cores, which determines the aggregate computing power in unit of MIPS (million instructions per second), memory space, and I/O speed. Initially, every module in the workflow is assigned to the worst vt in the cloud. If this mapping scheme exceeds the budget, there is no feasible solution; otherwise, continue.

Step 2) Calculate the throughput for each module in the workflow based on the initial mapping scheme from Step 1. The module with the smallest throughput determines the global bottleneck.

Step 3) Call Function $SelectVT()$ in Algorithm.13, check if it is possible to adjust the degree of parallelism and upgrade the type of VMs in order to achieve a higher global throughput within the budget. There are multiple options to determine the degree of parallelism and the vt for the global bottleneck module: add one more VM of the current vt within N_{vt}, which denotes the number of VM instances of the current vt; try to upgrade vt one level up at a time until reaching the best vt, and for each vt, gradually decrease the degree of parallelism from the degree of the current vt selection to 1. Every time we try to make an adjustment, we first eliminate the options that exceed the budget, and then compare the new global throughput after the adjustment. The option that results in the maximum increase in the global throughput is selected. Note that after each adjustment, the global bottleneck module may change.

Step 4) If any upgrade adjustment within the budget does not lead to a better global throughput, the algorithm terminates. Otherwise, update the degree of parallelism and the *vt* for the current bottleneck module, as well as the current global throughput.

Step 5) Go back to Step 2, and repeat the above process until no feasible upgrade adjustment option is available.

After identifying the global bottleneck module in Step 2, we try to increase the throughput of the current global bottleneck module by making an adjustment to the VT selection and the degree of parallelism of this module within the budget in Step 3. We consider several adjustment options and select one that leads to the maximum increase of the global workflow throughput. We would like to point out that the global bottleneck may shift to a different module after the adjustment, and therefore, it does not always yield the best performance if we only maximize the throughput increase of the current bottleneck module.

To increase the throughput of the current bottleneck module, there are two ways to make adjustments: i) increase the number of VM instances of the current *vt* by one; ii) choose a more powerful *vt* and vary the module's *DoP* from its current *DoP* to one. Any option that exceeds the budget constraint is ruled out. Among the feasible options within the budget constraint, we select the option that maximizes the throughput increase of the entire workflow. Note that in some cases adding

Algorithm 12: **BOTM**

Input: a DAG-structured topology $G_{tp}(V_{tp}, E_{tp})$, a set VT of VM types, the number N_{vt} of available VM instances of each vt, and a fixed financial budget b.

Output: the max throughput MT of the topology.

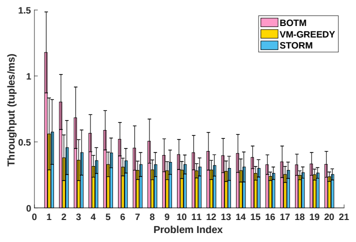
```
1: $curTH = 0$;
2: $MT = 0$;
3: sort the VM type $VT$ in an increasing order of system resources;
4: Assign every module $v_{tp} \in V_{tp}$ to a VM instance of the worst $vt$ for the topology;
5: if the cost $> b$ then
6:   throw ERROR("budget insufficient.");
7: Calculate $curTH$ and assign $MT$ with $curTH$
8: while true do
9:   $tIndex$ = the index of the bottleneck module with the smallest throughput $curTH$;
10:   $\{tType, tNum\} = selectVT(tIndex, VT, N_{vt}, b, curTH)$;
11:   if $(tType == -1)$ then
12:     break;
13:   update $tType$ and $tNum$ for this bottleneck module;
14:   $MT = curTH$;
15: return $MT$.
```

Algorithm 13: **SelectVT**

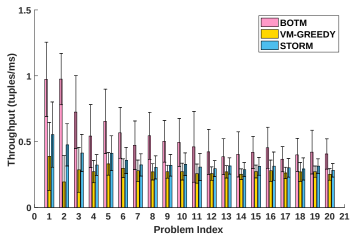
Input: the index of the bottleneck module $tIndex$, the VM type VT with the available number N_{vt} of VM instances of each vt, a fixed financial budget b, and the current topology throughput $curTH$.

Output: the VT type $tType$ and the degree $tNum$ of parallelism for the bottleneck module.

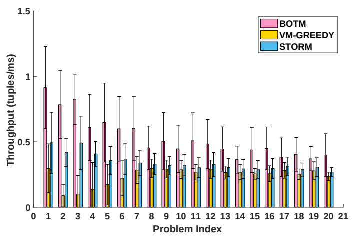
```
1: $tType = -1$;
2: $tNum = -1$;
3: $curType$ = VT type of $tIndex$;
4: $curNum$ = the degree of parallelism for the bottleneck module of $tIndex$;
5: for all $vt \in VT$ do
6:   if $vt$ is the same as $curType$ and one more VM instance of $vt$ is available then
7:     assign one more VM instance of $vt$ to module of $tIndex$;
8:     calculate the topology throughput $TH$ after the adjustment;
9:     if cost $\leq b$ and $TH > curTH$ then
10:      $tType = vt$;
11:      $curTH = TH$;
12:      $tNum = curNum + 1$;
13:  else if $vt$ is better than $curType$ then
14:    $tmpNum = curNum$;
15:    while $tmpNum > 0$ do
16:      assign $tmpNum$ VM instances of $vt$ to module of $tIndex$;
17:      calculate the topology throughput $TH$ after the adjustment;
18:      if cost $\leq b$ and $TH > curTH$ then
19:        $tType = vt$;
20:        $curTH = TH$;
21:        $tNum = tmpNum$;
22:      $tmpNum - -$;
23: return $\{tType, tNum\}$.
```



(a) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 1.



(b) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 3.



(c) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 5.

Figure 7.7 Performance measurements for simulations under different budget levels.

resources may result in reduced cost. Hence, we calculate the financial cost every time when we make a change to the selection of virtual machines.

In Storm, users are allowed to change the DoP for each module of the topology, but it is generally difficult for them to decide the most suitable DoP for each module. In many cases, users may specify an arbitrary DoP based on their empirical study. Our work not only selects the suitable vt but also determines the suitable degree of parallelism for each module of the workflow.

The time complexity of BOTM is $O(|V_{tp}| \cdot \max(N_{vt}) \cdot |VT|)$, where $|VT|$ is the number of VM types, $\max(N_{vt})$ denotes the largest number of VM instances for all $vt \in VT$. is intra- or inter-cloud

7.3 Simulation-based Performance Evaluation

7.3.1 Simulation Settings

We implement the proposed BOTM algorithm in C++ and evaluate its performance in comparison with the default Storm configuration, denoted as STORM_DEFAULT, and a heuristic algorithm VM-GREEDY. By comparing with Storm's default scheduler, which is used by many real-life applications, we are able to examine the benefits of BOTM to both service providers and end users when executing budget-constrained workflows. VM-GREEDY is a commonly used benchmark method that takes a greedy strategy for VM optimization to assign as many high-end VM instances as possible within the budget. The source code of BOTM implementation is available for download in GitHub Repository [13].

The problem size (reflected by the problem index) is defined as a 2-tuple $(|V_{tp}|, |E_{tp}|)$, where $|V_{tp}|$ is the number of Storm topology tasks, and $|E_{tp}|$ is the number of topology links. We generate topology instances of different scales in a random manner as follows [52]: i) lay out all $|V_{tp}|$ modules sequentially along a pipeline, each of which is assigned a workload randomly generated within the range

$[5, 500]$, which represents the total number of million instructions; ii) for each module, add an input edge from a randomly selected preceding module and add an output edge to a randomly selected succeeding module (the first spout module only needs output and the last bolt module only needs input); iii) randomly select two modules from the pipeline and add a directed edge between them (from left to right) until reaching the given number of edges.

We compare BOTM with the other two algorithms in comparison in terms of workflow throughput under the same budget constraint.

Comparison with Optimal Solutions We compare BOTM with optimal solutions in three small-scale problems of $(3, 3, 2), (4, 4, 3)$, and $(4, 5, 4)$, each in the form of (number of modules, number of edges, number of VTs). For each problem size, we randomly generate 10 problem instances with different module workloads and DAG topologies. In each problem instance, we specify five different budget levels. We run all three algorithms on these instances and compare the throughput measurements with the optimal ones computed by an exhaustive search-based approach. Figure 7.8 shows the number of optimal results among 50 instances (10 workflow instances $\times$ 5 budget levels) achieved by BOTM, VM-GREEDY, and STORM_DEFAULT, respectively, under different problem sizes. In $(3, 3, 2)$, STORM_DEFAULT does not produce any optimal solution and thus is not visible in the chart. We observe that BOTM is more likely to achieve the optimality than the others in a statistical sense, which indicates the efficacy of BOTM. However, since these are small-scale problem instances, the absolute values of the differences from the optimal results are not significant.

7.3.2 Comparison with Other Methods

In the simulation, we consider 16 virtual machine types with their respective system specifications and in-cloud financial costs randomly selected from a range

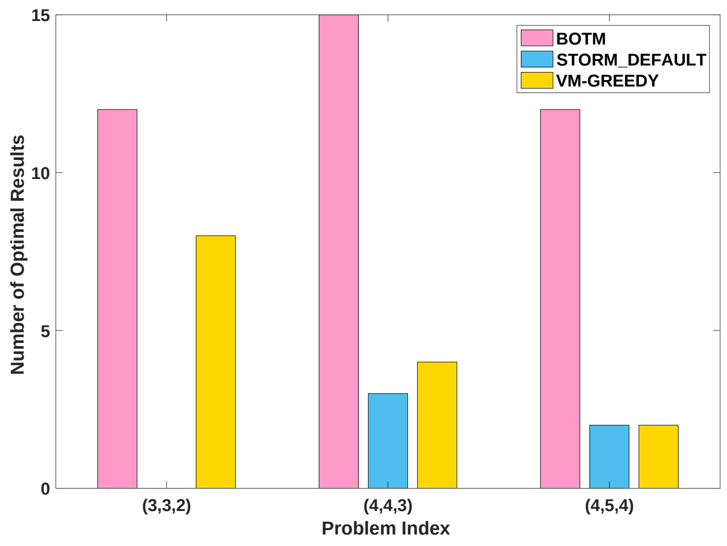


Figure 7.8 The number of optimal results among 50 instances (10 workflow instances $\times$ 5 budget levels) produced by BOTM, VM-GREEDY and STORM _DEFAULT, respectively, under different problem sizes.

corresponding to commonly used virtual machines provisioned by Amazon Web Services (AWS) [18]. We consider 20 problem sizes from small to large scales, indexed from 1 to 20. For each problem size, we randomly generate 20 problem instances, in each of which, we choose 6 budget levels with an equal interval of $\Delta b = (b_{max} - b_{min})/6$ within a certain budget range $[b_{min}, b_{max}]$, where B_{min} is 10% more than the minimum budget to run the entire workflow on the worst cluster, and B_{max} is 10% more than the maximum budget to run the entire workflow on the best cluster. For each of the 6 budget levels from low to high levels, indexed from 1 to 6, we run the scheduling simulation by iterating through 20 problem sizes from small to large scales. We measure the average throughput with a standard deviation achieved by BOTM, VM-GREEDY, and STORM_DEFAULT, respectively. These measurements show the performance superiority of BOTM at each of the six budget levels. The results at levels 1, 3 and 5 are plotted in Figure 7.7 for a visual comparison.

These performance results show that BOTM achieves performance improvement over VM-GREEDY and STORM_DEFAULT. Such performance improvements are considered significant for stream data processing in large-scale scientific applications. On average, the simulation results show that BOTM achieves a throughput that is 2.3 times of VM-GREEDY and 50% higher than STORM_DEFAULT. This is considered to be a significant improvement when dealing with large-scale stream data.

7.3.3 Convergence of BOTM

To investigate the convergence property of BOTM, we run this algorithm on the problem instance of Index 5 under three different budget levels, i.e., low, medium, and high. The low budget level is 10% more than the budget that is sufficient for the workflow to be executed using the worst virtual machines; the high budget level is 10% less than the budget that is sufficient for the workflow to be executed using the best virtual machines; the medium budget level is 50% of the budget that is sufficient for the workflow to be executed using the best virtual machines. We plot the optimization process of BOTM in these three scenarios in Figure 7.9, which shows that BOTM converges to the maximum throughput after 30 iterations within less than one second. For problem index 10 and above, we observe that BOTM converges after at most 50 iterations.

7.4 Experiment-based Performance Evaluation

In this section, we conduct two sets of experiments on two real-life datasets. Different data volumes (12GB and less than 1GB data) are tested for scalability evaluation.

7.4.1 Experiment 1 with Flight Data

Storm Topology We conduct Storm experiments for streaming data processing to compute various statistics on 22 years of global flight datasets of about 12GB from 1987 to 2008 at Statistical Computing [14]. The topology structure is shown

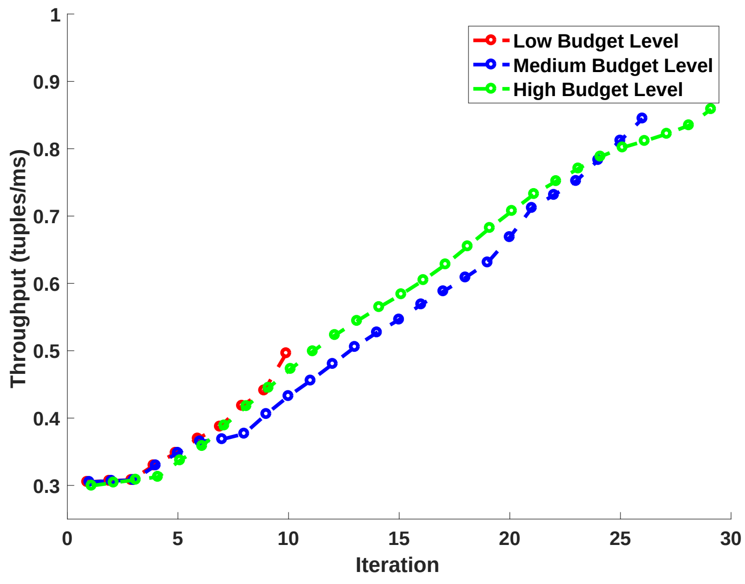
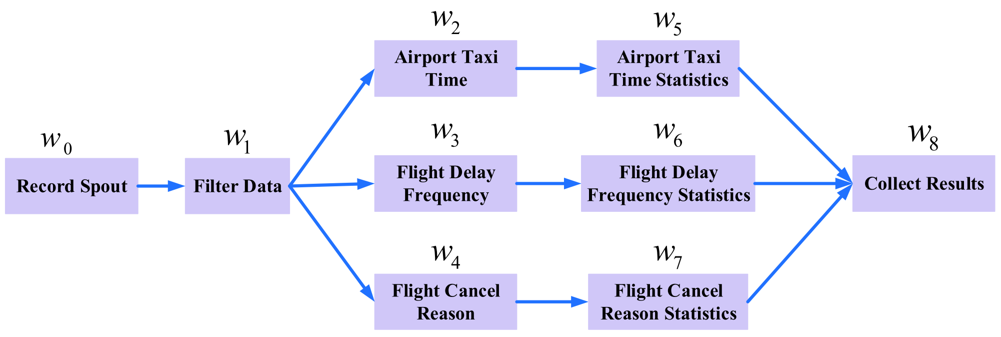


Figure 7.9 The optimization process of BOTM running the problem instance of Index 5 in the simulations under three different budget levels.



w_0 Record Spout
w_1 Filter Data
w_2 Airport Taxi Time
w_3 Flight Delay Frequency
w_4 Flight Cancel Reason
w_5 Airport Taxi Time Statistics
w_6 Flight Delay Frequency Statistics
w_7 Flight Cancel Reason Statistics
w_8 Collect Results

Figure 7.10 The structure of the Storm topology for flight data processing.

in Figure 7.10, where every module is a task (spout/bolt): w_0 emits streaming data instances every 1 ms; w_1 filters out the headline in each data file; w_2 and w_5 calculate the average taxi in/out time at each airport, where w_2's key is the airport name and value is the taxi time, w_5's key is the airport name and value is the average taxi time;

Table 7.1 System Specifications of Different VM Types in the Experiment

VM Type	Instance Name	Availability Zone	CPU (GHz)	RAM (GB)	Num of Instances	Price ($/min)
*vt*1	*t*2.*small*	US West (Oregon)	2.5×1	2	9	0.0230
*vt*2	*t*2.*medium*	US West (Oregon)	2.5×2	4	4	0.0464
*vt*3	*t*2.*xlarge*	US West (Oregon)	2.4×4	16	4	0.1856
*vt*4	*t*2.2*xlarge*	US West (Oregon)	2.4×8	32	4	0.3712

w_3 and w_6 calculate the average delay frequency of each flight, where w_3's key is the flight number and value is the delay frequency, w_6's key is the flight number and value is the average delay frequency; w_4 and w_7 calculate the frequency of each "Flight Cancellation Reason" over all of the years, where w_4's key is the cancellation code and w_4's value is 1, w_7's key is the cancellation code and w_7's value is the cancellation frequency. w_8 collects all the results in each category, where key is the ranking type (cancellation code, airport name, and flight number), and value is the result (the average taxi time, the average delay frequency, the cancellation frequency).

Experimental Settings We consider four VM types in Amazon Web Services (AWS) [18] and construct three different heterogeneous clusters. Table 7.1 tabulates the system specification and pricing model (in unit of US Dollar per minute) of each VM type, and the number of available VM instances of each VM type. In each

Table 7.2 Execution Time Matrix T_e in ms

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
$vt1$	1.85	46.21	53.44	105.83	97.19	12.56	17.39	55.70	11.23
$vt2$	1.47	26.79	6.57	12.04	39.98	1.09	4.18	3.52	2.05
$vt3$	1.38	14.58	5.63	11.06	26.18	1.08	4.09	3.47	1.54
$vt4$	1.21	13.33	3.13	6.42	24.77	1.07	3.37	2.62	1.41

Table 7.3 The VM Instances of the Storm Cluster Provisioned under Different Mapping schemes in AWS

Cluster	$vt1$	$vt2$	$vt3$	$vt4$	Total number of
	Inst.	Inst.	Inst.	Inst.	VM instances
$C1$: under BOTM	4	2	0	4	10
$C2$: under VM-GREEDY	6	1	1	4	12
$C3$: randomly generated	4	4	3	2	13

Table 7.4 Mapping Schemes Obtained by BOTM and VM-G (VM-GREEDY) in Experiment 1, Where Each Cell Stores (vt, DoP) for the Corresponding Module

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
BOTM	$(1,1)$	$(2,1)$	$(2,1)$	$(4,2)$	$(4,1)$	$(1,1)$	$(4,1)$	$(1,1)$	$(1,1)$
VM-G	$(1,1)$	$(2,1)$	$(4,4)$	$(1,1)$	$(1,1)$	$(3,1)$	$(1,1)$	$(1,1)$	$(1,1)$

Table 7.5 Throughput Measurements in Tuples/*min* of BOTM, VM-G (VM-GREEDY), and STORM (STORM_DEFAULT) in Experiment 1 on Flight Data, Where Each Run Lasts for 10 Hours

Algorithm.	Idx	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	Thruput	Average
		(average number of tuples processed by each module within a 10-min window)										Thruput
BOTM	1	25180	24800	24760	24940	24060	49040	24700	23980	97660	2398	
C1	2	24640	24420	24360	24580	23640	48280	24300	23560	95900	2356	2389
	3	25620	24880	24820	25240	24240	49200	25000	24140	97880	2414	
VM-G	1	19340	19340	19360	19300	18860	38360	19100	18860	76160	1886	
C2	2	22660	22660	22620	22600	21920	44720	22360	21920	89020	2192	1682
	3	11460	11460	11440	11440	9680	22620	11320	9680	43580	968	
STORM	1	15940	15920	15900	15880	15100	31460	15720	15120	62300	1510	
C1	2	16700	16700	16660	16680	15900	32980	16500	15900	65380	1590	1573
DoP=1	3	16680	16660	16660	16660	16180	32980	16480	16180	65520	1618	
STORM	1	20200	18640	18600	18600	17700	36780	18400	17700	72880	1770	
C1	2	18140	18140	18140	18120	16860	35860	17940	16860	70640	1686	1765
DoP=2	3	20300	20200	20160	20160	18380	39940	19960	18340	78260	1838	
STORM	1	2140	1440	1420	1440	720	2860	1440	720	5000	72	
C2	2	3480	1600	1540	1540	660	3020	1500	680	5220	66	68
DoP=1	3	2740	1580	1560	1580	660	3120	1560	680	5320	66	
STORM	1	7160	2180	1480	1500	820	2880	1500	840	5200	82	
C2	2	7380	1660	1260	1280	640	2440	1240	640	4300	64	74
DoP=2	3	7540	1480	1460	1460	760	2880	1440	740	5020	76	
STORM	1	9640	2080	1360	1480	760	2720	1440	800	4860	76	
C2	2	7980	2400	1380	1420	680	2660	1360	640	4700	64	74
DoP=3	3	9760	2180	1380	1320	820	2700	1300	840	4760	82	
STORM	1	10280	2120	1340	1360	780	2640	1280	780	4720	78	
C2	2	11700	1880	1360	1340	740	2620	1360	720	4660	72	82
DoP=4	3	10960	2200	1360	1300	960	2600	2600	2600	2600	96	
STORM	1	20440	19560	19500	19500	18920	38620	19320	18920	76860	1892	
C3	2	16700	16720	16660	16660	15860	33040	16520	15860	65420	1586	1673
DoP=1	3	16680	16120	16080	16080	15400	31840	15920	15420	63180	1540	
STORM	1	22260	21040	21000	21000	19900	41580	20800	19880	82280	1988	
C3	2	19220	17160	17140	17140	16220	33940	16960	16200	67140	1620	1320
DoP=2	3	7060	4440	4440	4440	3520	8800	4420	3540	16720	352	
STORM	1	5380	5140	5160	5160	2300	10260	5100	2300	17700	230	
C3	2	5540	4860	4860	4860	2380	9580	4800	2420	16760	238	236
DoP=3	3	5520	4880	4840	4840	2420	9600	4780	2400	16800	240	
STORM	1	6520	4160	4100	4120	2180	8200	4100	2160	14420	216	
C3	2	7120	4960	4960	4960	2320	9800	4920	2340	17040	232	228
DoP=4	3	6660	4360	4320	4320	2360	9560	4280	2380	15260	236	

Table 7.6 Execution Time Matrix T_e in ms for WRF

	$w0$	$w1$	$w2$	$w3$	$w4$	$w5$	$w6$	$w7$
$vt1$	100.17	6889.82	3434.38	8854.28	17591.02	55361.93	65107.75	1007.06
$vt2$	100.21	3592.12	1813.37	4069.33	7821.38	29563.50	28720.50	586.00
$vt3$	100.28	1868.93	1710.83	2145.80	4163.83	15309.33	11885.50	491.67
$vt4$	100.26	1119.35	554.91	1358.38	2595.86	3960.33	11646.00	141.50

cluster, we install STORM 1.0.0 on the VM instances, and install Zookeeper 3.4.8 on the VM instance where Nimbus is installed. As shown in Table. 7.9 and Table. 7.5), since the processing time of each tuple by any module of the workflow is on the order of seconds, their performance is not affected by the degradation of the virtual CPU performance, as experienced by some users running long-time jobs in AWS.

Performance Comparison We first execute the entire topology on one VM instance for each of four VT types in stand-alone mode to obtain the execution time matrix for one tuple on the module, as shown in Table 7.2. For w_1 to w_4, the time complexity of each task is $O(n)$, where n is the size of the record. For w_5 to w_8, the time complexity of each task is $O(1)$.

In the experiment, the time interval for emitting two contiguous tuples is set to be a random value within a range of $[0.5ms, 1.5ms]$, and the budget is set to be five times $p(vt_4)$. We run BOTM and VM-GREEDY to obtain two mapping schemes, as tabulated in Table 7.4, where each cell stores (vt, DoP) for the corresponding module. For example, $(4, 2)$ for module w_3 in the mapping scheme produced by BOTM means that 2 VM instances of VM type vt_4 are used to run 2 instances of w_3.

Based on the mapping schemes produced by BOTM and VM-GREEDY, we set up two corresponding clusters $C1$ and $C2$. The $C1$ cluster produced by BOTM contains 10 VM instances, while the $C2$ cluster produced by VM-GREEDY contains

12 VM instances. We also set up a randomly generated cluster $C3$ that contains 13 VM instances satisfying the budget constraint. The configurations of these three clusters are provided in Table 7.3.

We run the Storm topology for flight data processing in $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, for three times. Also, we run the topology in the default Storm system in clusters $C1$, and set the DoP for each module from 1 to the highest DoP in the mapping scheme achieved by BOTM, which is 2. Similarly, we run the topology in the default Storm system in clusters $C2$, and set the DoP for each module from 1 to the highest DoP in the mapping scheme achieved by VM-GREEDY, which is 4. In the randomly generated cluster $C3$, we set the DoP for each module from 1 to 4. Note that for each DoP, we run the experiment for three times. The performance measurements in all of these experiments are tabulated in Table 7.5, where the underlined throughput performance measured within a 10-minute window corresponds to the global bottleneck module. We provide such microscopic behaviors in every experiment to study the stability of each algorithm. These measurements show that the proposed BOTM algorithm achieves consistent performance in three runs while the other algorithms in comparison lack such stability.

We calculate the average throughput with standard deviation across different DoP based on these performance measurements, and plot them in Figures. 7.11 and 7.12 for a visual comparison. We observe that BOTM consistently outperforms the other algorithms in comparison.

Both BOTM and VM-GREEDY decide the VT selection and the DoP for each module of the workflow. In default Storm, we vary the DoP for every module from 1 to the highest DoP among all modules in the mapping scheme produced by BOTM and VM-GREEDY. These results show that a higher DoP does not always yield a

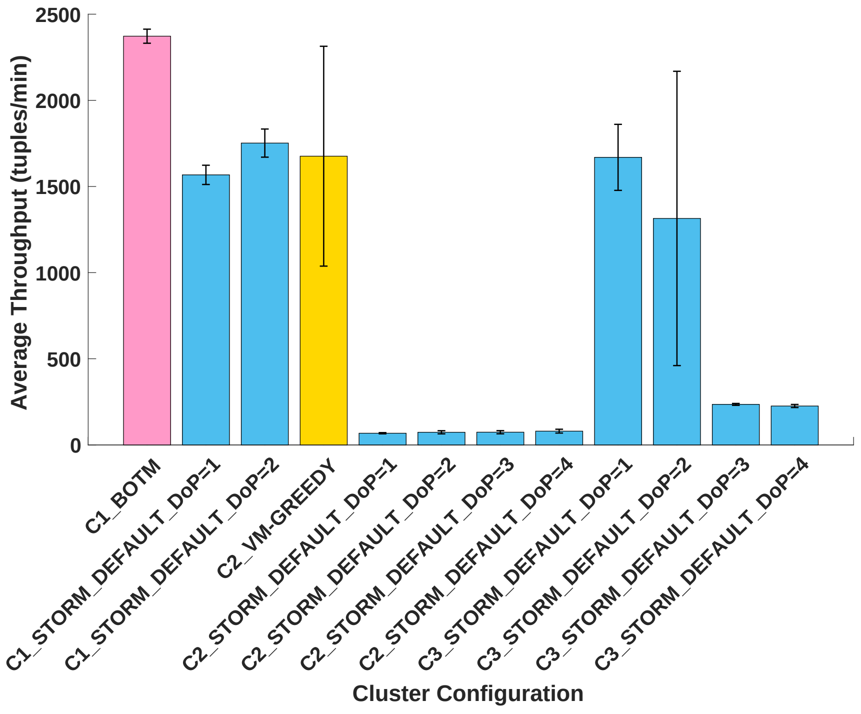


Figure 7.11 The average throughput with standard deviation of the Storm topology across different degrees of parallelism (DoP) in clusters $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, and a randomly generated $C3$ under a given budget.

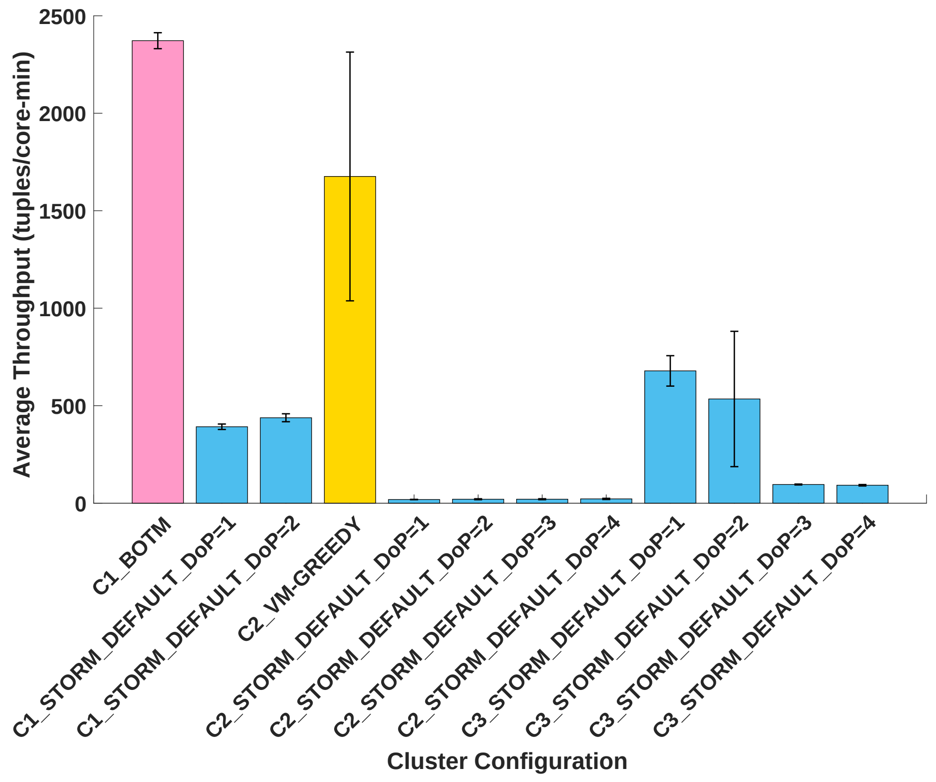


Figure 7.12 The average throughput (per core) with standard deviation of the Storm topology across different degrees of parallelism (DoP) in clusters $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, and a randomly generated $C3$ under a given budget.

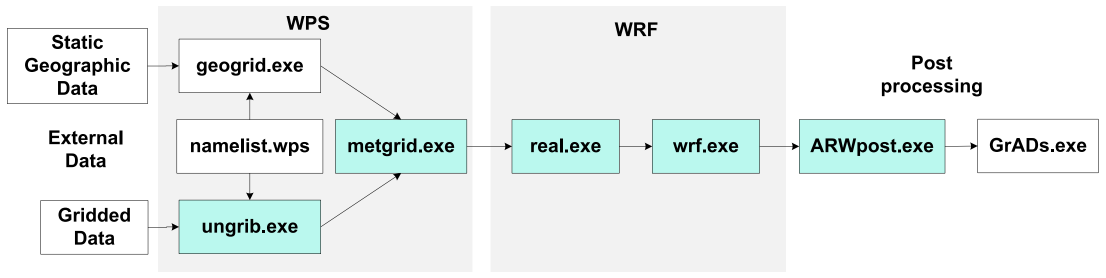


Figure 7.13 A general structure of the executable WRF workflow.

Table 7.7 Storm Cluster VM Instances Provisioned Under Different Mapping Schemes in AWS

Cluster	*vt*1 Inst.	*vt*2 Inst.	*vt*3 Inst.	*vt*4 Inst.	Total number of VM instances
$C4$: under BOTM	4	2	0	2	8
$C5$: under VM-GREEDY	4	2	1	3	10
$C6$: randomly generated	0	2	3	3	8

Table 7.8 Mapping Schemes Obtained by BOTM and VM-G (VM-GREEDY) in Experiment 2, Where Each Cell Stores (vt, DoP) for the Corresponding Module

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7
BOTM	$(1,1)$	$(1,1)$	$(1,1)$	$(1,1)$	$(2,1)$	$(4,1)$	$(4,1)$	$(2,1)$
VM-G	$(1,1)$	$(2,1)$	$(4,3)$	$(1,1)$	$(1,1)$	$(3,2)$	$(2,1)$	$(1,1)$

better workflow throughput performance as the default scheduler in Storm does not consider the global bottleneck.

7.4.2 Experiment 2 with Climate Data

WRF Workflow To evaluate the performance of our algorithm in real computing environments, we conduct Storm experiments based on the Weather Research and Forecasting (WRF) model [50], which has been widely adopted for regional to continental scale weather forecast. The WRF model [9] generates two large classes of simulations either with an ideal initialization or utilizing real data. In our experiments, the simulations are generated from real data, which usually requires preprocessing from the WPS package [10] to provide each atmospheric and static field with fidelity appropriate to the chosen grid resolution for the model.

The structure of a general WRF workflow is illustrated in Figure 7.13, where the WPS consists of three independent programs: geogrid.exe, ungrib.exe, and metgrid.exe [54]. The geogrid program defines the simulation domains and interpolates various terrestrial datasets to the model grids. The user can specify information in the namelist file of WPS to define simulation domains.that typically contain more fields than needed to initialize WRF. The ungrib program "degrib" the data and stores the results in a simple intermediate format. The metgrid program horizontally interpolates the intermediate-format meteorological data that are extracted by the ungrib program into the simulation domains defined by the geogrid program. The interpolated metgrid output can then be ingested by the WRF package, which the data by WPS, we will run the programs in WRF model.contains an initialization program real.exe for real data and a numerical integration program wrf.exe. The postprocessing model consists of ARWpost and GrADs. ARWpost reads-in WRF-ARW model data and creates output files for display by GrADS.

We duplicate three WRF pipelines each from ungrib.exe to ARWpost.exe, and group these programs into different aggregate modules to simulate real-life workflow clustering and provide various module parallelism, as shown in Figures. 7.14 and 7.15. Figure 7.15 is a high-level view of grouped workflow in Figure 7.14, where w_0 and w_7 are the start and end modules [54].

We execute the WRF topology in the same computing environment as the experiments for flight data processing.

Performance Comparison We first execute the entire topology on one VM instance of each of four VT types in the stand-alone mode to obtain the execution time matrix for one tuple on the module, as shown in Table 7.6.

Similarly, in this set of experiments, the time interval for emitting two contiguous tuples is set to be a random value within a range of $[0.5ms, 1.5ms]$. The budget is set to be five times $p(vt_4)$. We run BOTM and VM-GREEDY to obtain two mapping schemes as tabulated in Table 7.8. Similar to Table 7.4, each cell stores

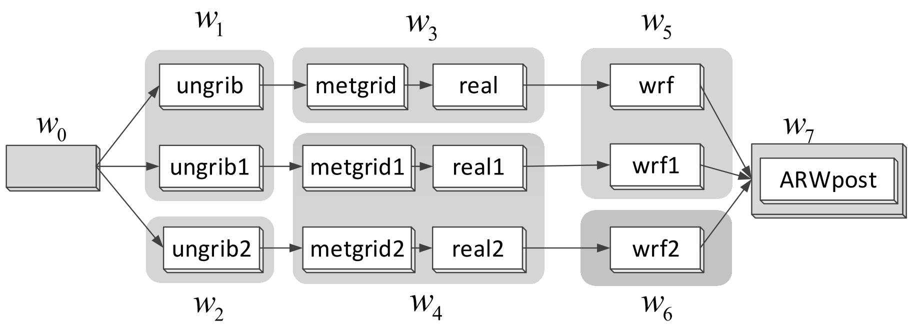


Figure 7.14 The WRF Storm workflow of three pipelines in the experiments.

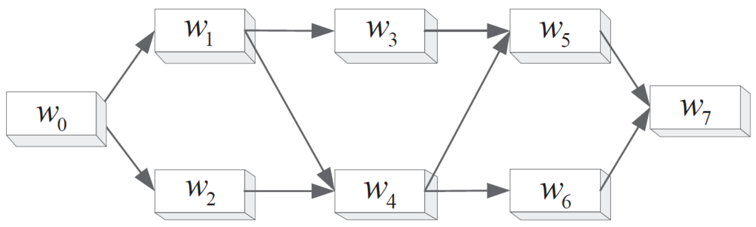


Figure 7.15 The WRF Storm workflow after grouping.

(vt, DoP) for each corresponding module. Based on the mapping schemes produced by BOTM and VM-GREEDY, we set up two corresponding clusters $C4$ and $C5$. The $C4$ cluster produced by BOTM contains 8 VM instances, while the $C5$ cluster produced by VM-GREEDY contains 10 VM instances. We also set up a randomly generated cluster $C6$ that contains 8 VM instances satisfying the budget constraint. The configurations of these three clusters are provided in Table 7.7.

We run the Storm topology for WRF workflow in $C4$ and $C5$ produced by BOTM and VM-GREEDY, respectively, for three times. Also, we run the topology in the default Storm system in clusters $C4$, and set the DoP for each module to be 1, which is the highest DoP in the mapping scheme achieved by BOTM. Similarly,

Table 7.9 Throughput Measurements in tuples/*hour* of BOTM, VM-G (VM-GREEDY), and STORM (STORM_DEFAULT) in Experiment 2 on WRF Workflow, Where Each Run Lasts for 10 Hours

Algorithm	Idx	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	Thruhput	Average Thruput
		(average number of tuples processed by each module within a 10-min window)									
BOTM	1	960	440	900	400	200	60	40	100	240	
C4	2	940	460	920	400	200	40	40	100	240	240
	3	940	440	900	380	220	40	60	100	240	
VM-G	1	875	482	71	393	196	54	36	120	216	
C5	2	893	429	321	339	339	36	36	89	216	210
	3	810	479	397	380	380	33	50	99	198	
STORM	1	480	21	42	10	10	10	10	10	60	
C4	2	920	60	140	60	60	20	20	40	120	66
DoP=1	3	137	9	24	9	23	3	3	3	18	
STORM	1	531	11	23	34	34	11	11	22	66	
C5	2	133	9	17	9	9	3	3	3	18	36
DoP=1	3	164	7	14	7	7	4	4	4	24	
STORM	1	261	20	38	14	6	3	6	6	18	
C5	2	209	5	9	5	9	2	5	5	12	22
DoP=2	3	564	58	122	58	38	13	6	13	36	
STORM	1	485	4	11	4	20	4	4	8	24	
C5	2	631	28	42	19	14	14	14	5	30	24
DoP=3	3	380	20	12	12	3	3	6	3	18	
STORM	1	375	17	25	8	17	8	8	8	48	
C6	2	143	3	3	3	6	3	3	6	18	28
DoP=1	3	120	5	8	3	10	3	3	3	18	
STORM	1	297	13	30	13	13	3	7	7	18	
C6	2	281	24	46	17	12	4	3	6	18	18
DoP=2	3	245	3	11	5	11	3	5	5	18	
STORM	1	812	18	36	18	61	18	24	48	108	
C6	2	137	2	4	1	5	3	3	7	6	48
DoP=3	3	345	8	16	10	23	8	5	13	30	

we run the topology in the default Storm system in clusters $C5$, and set the DoP for each module from 1 to the highest DoP in the mapping scheme achieved by VM-GREEDY, which is 3. In the randomly generated cluster $C6$, we set the DoP for each module from 1 to 3. For each DoP, we run the experiment for three times. All performance measurements are tabulated in Table 7.9, where the underlined throughput performance measured within a 10-min window corresponds to the global bottleneck module.

We calculate the average throughput with standard deviation across different degrees of parallelism based on these performance measurements, and plot them in Figures. 7.16 and 7.17 for a visual comparison. Again, we observe that BOTM consistently outperforms the other algorithms in comparison. In Figure 7.18, we also illustrate the resource consumption (number of cores $\times$ memory size $\times$ time unit) for WRF data processing across different degrees of parallelism (DoP) in clusters C4, C5, and randomly generated cluster C6 by BOTM, VM-GREEDY and STORM_DEFAULT, respectively, under a given budget.

In this experiment, we observe that the DoP for each module in the mapping scheme produced by BOTM is only 1. However, we still run Storm in its default setting in the cluster provided by BOTM and increase the DoP from 1 to the highest degree decided by VM-GREEDY. These results show that even without parallel processing, BOTM still outperforms the other algorithms with parallel processing.

7.4.3 Summary

The performance superiority of BOTM is brought by a careful design that follows two important guidelines: i) it is bottleneck-oriented as the global bottleneck module determines the overall throughput of the entire workflow, and ii) it is bottleneck-adaptive as the global bottleneck may shift to a different module after each adjustment and the most suitable adjustment is adopted to maximize the global

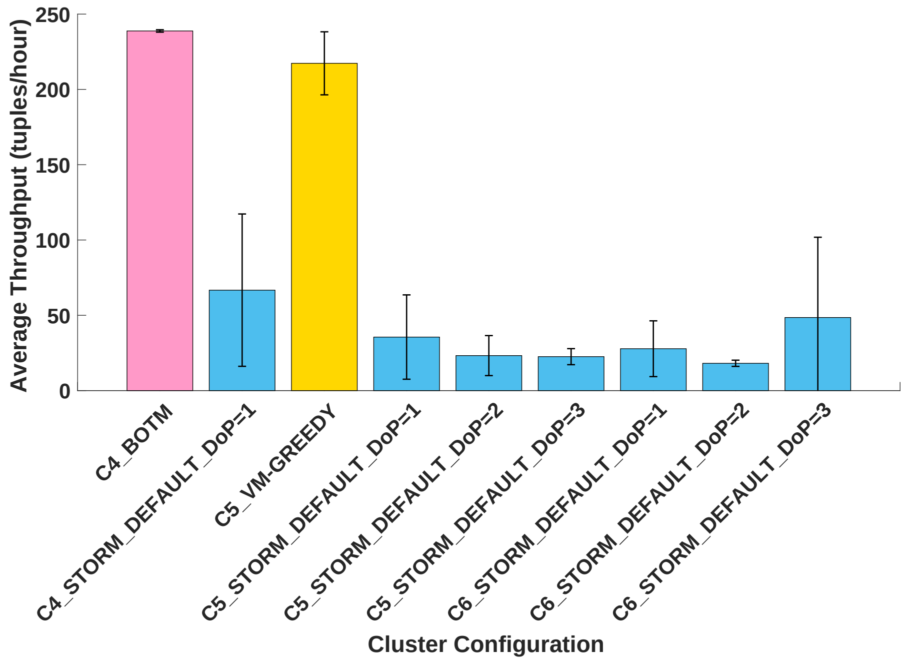


Figure 7.16 The average throughput with standard deviation of the Storm topology for WRF data processing across different degrees of parallelism (DoP) in clusters C4 and C5 produced by BOTM and VM-GREEDY, respectively, and randomly generated cluster C6 under a given budget.

throughput of the workflow instead of the local throughput of any component module. Storm's default scheduler (STORM_DEFAULT) neither considers the bottleneck module nor performs selective resource allocation; VM-GREEDY also neglects the bottleneck and only allocates resources to modules in a topologically sorted order.

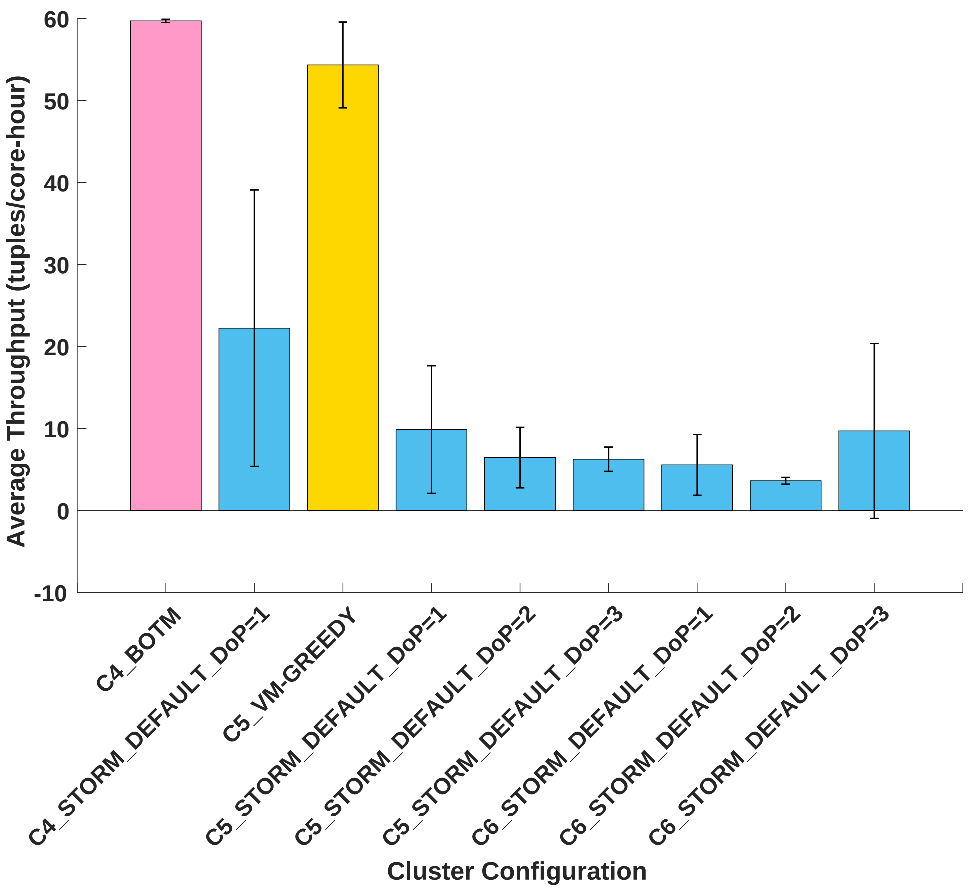


Figure 7.17 The average throughput (per core) with standard deviation of the Storm topology for WRF data processing across different degrees of parallelism (DoP) in clusters C4 and C5 produced by BOTM and VM-GREEDY, respectively, and randomly generated cluster C6 under a given budget.

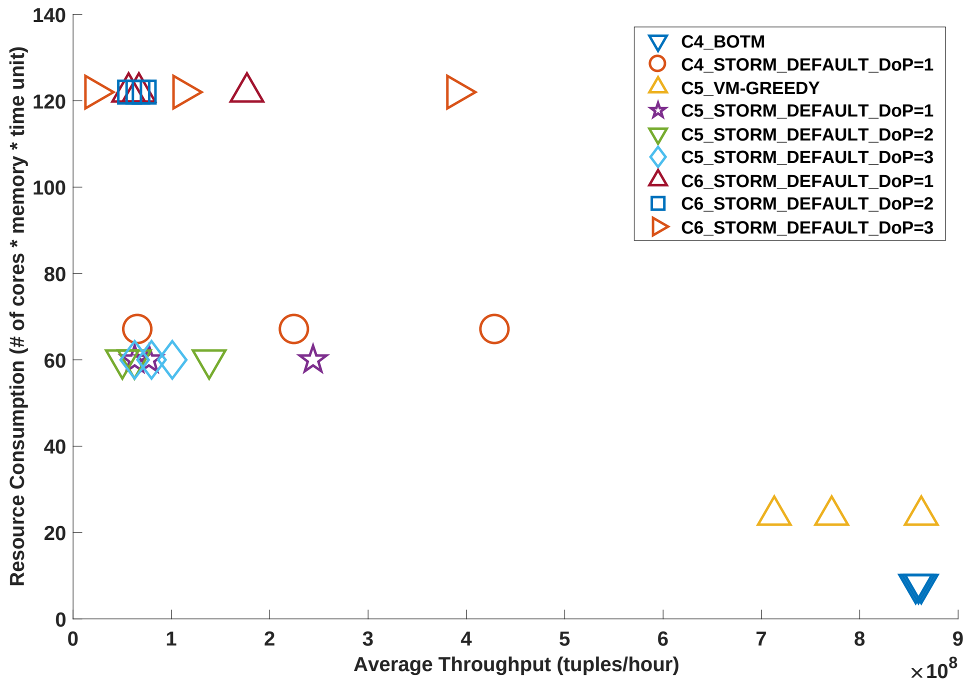


Figure 7.18 The resource consumption for WRF data processing across different degrees of parallelism (DoP) in clusters C4, C5, and randomly generated cluster C6 by BOTM, VM-GREEDY and STORM_DEFAULT, respectively, under a given budget.

CHAPTER 8

CONCLUSION

In this work, we focus on improving performance of big data computing workflows for batch and stream data processing in multi-clouds. We have worked on serial-computing workflows, MapReduce workflows and Storm-based streaming processing workflows.

For serial computing workflows, we constructed rigorous mathematical models to analyze the intra- and inter-cloud execution process of scientific workflows and we formulated a budget-constrained workflow mapping problem to minimize the makespan of scientific serial-computing workflows in multi-cloud environments, referred to as BCMED-MC, which was shown to be NP-complete. For each of the two senarios: mapping using dedicated VM, and mapping reusing VM, we designed a heuristic algorithm that incorporates the cost of inter-cloud data movement into workflow scheduling.

As we enter the big data era, several efforts have been made to develop workflow engines for Hadoop ecosystem in clouds with virtual resources. We expanded our work to Hadoop environments and formulated a budget-constrained workflow mapping problem to minimize the makespan of MapReduce workflows in multi-cloud environments, referred to as MinMRW-MC, which was shown to be NP-complete. We designed a heuristic algorithm for MinMRW-MC that adapts the execution dynamics of MapReduce programs..

Streaming data processing has become increasingly important due to its impacts on a wide range of use cases, such as real-time trading analytics, malfunction detection, campaign, social network, log processing and metrics analytics. To meet the demands of streaming data processing, many new computing engines have emerged, including Apache Storm and Apache Spark (Spark Streaming). We formulated a budget-constrained Storm topology mapping problem to maximize the throughput in cloud environments, referred to as STM-BC, which was shown to be NP-complete. We designed a heuristic algorithm BOTM for STM-BC.

The performance superiority of all solutions over other methods are demonstrated through extensive simulations and real-life experiments in public clouds.

It would be of our future interest to refine and generalize the mathematical models to achieve a higher level of accuracy for workflow execution time measurement in real-world cloud environments. For example, the actual execution time of different programs on different types of VMs is dependent on many factors such as program structures and machine configurations. Particularly, when provisioning multiple VMs on the same physical server, the performances of those VMs are correlated and constrained by the physical machine. Moreover, in real networks, physical servers may fail under a certain probability and the actual workload of workflow modules may be subject to dynamic changes.

REFERENCES

[1] Information Sciences Institute, University of Southern California. Pegasus in the cloud. http://pegasus.isi.edu/cloud. Access in 2016.

[2] Flink. https://flink.apache.org/. Access in 2017.

[3] Spark Streaming. https://spark.apache.org/streaming/. Access in 2017.

[4] Samza. http://samza.apache.org/. Access in 2017.

[5] Apex. https://apex.apache.org/. Access in 2017.

[6] Google Cloud Dataflow. https://cloud.google.com/dataflow/. Access in 2017.

[7] Jstorm. http://jstorm.io/. Access in 2017.

[8] Heron. https://twitter.github.io/heron/. Access in 2017.

[9] Weather Research and Forecasting (WRF) Model. http://wrf-model.org/index.php. Access in 2017.

[10] WRF Preprocessing System (WPS). http://www2.mmm.ucar.edu/wrf/users/wpsv2/wps.html. Access in 2017.

[11] Google Cloud. https://cloud.google.com/. Access in 2017.

[12] Amazon Web Services (AWS). https://aws.amazon.com/. Access in 2017.

[13] Bottleneck-Oriented Topology Mapping (BOTM). https://github.com/BigDataCenter/BOTM.

[14] *ASA Sections on:Statistical Computing Statistical Graphics*.

[15] S. Abrishami and M. Naghibzadeh. Deadline-constrained workflow scheduling in software as a service cloud. *Scientia Iranica*, (0), 2012.

[16] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: a scheduling heuristic for streaming application on the grid. In *Proc. of the 13th Multimedia Comp. and Net. Conf.*, San Jose, CA, 2006.

[17] H. Alshammari, J. Lee, and H. Bajwa. Improving current hadoop mapreduce workflow and performance. *Int. Journal of Computer Applications*, 116(15):0975–8887, 2015.

[18] Amazon, 2016. EC2. http://aws.amazon.com/ec2/. Access in 2016.

[19] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218, 2013.

[20] Apache, 2016. Hadoop. http://hadoop.apache.org. Access in 2017.

[21] Apache, 2016. Tez. https://tez.apache.org. Access in 2017.

[22] Apache, 2016. Oozie. https://oozie.apache.org. Access in 2017.

[23] Apache, 2016. Spark. http://spark.apache.org. Access in 2017.

[24] Apache, 2016. Storm. http://storm.apache.org. Access in 2017.

[25] Z. Hong R. Farivar B. Peng, M. Hosseini and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[26] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[27] Z. Chen, J. Xu, J. Tang, K. Kwiat, and C. Kamhoua. G-storm: Gpu-enabled high-throughput online data processing in storm. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 307–312. IEEE, 2015.

[28] C. Wu and G. Yi. Optimizing end-to-end performance of data-intensive computing pipelines in heterogeneous network environments. *Journal of Parallel and Distributed Computing*, 71(2):254–265, 2011.

[29] N. Rosa D. Oliveira, A. Brinkmann and P. Maciel. Performability evaluation and optimization of workflow applications in cloud environments. *Journal of Grid Computing*, 17:749 – 770, 2019.

[30] L. Eskandari, Z. Huang, and D. Eyers. P-scheduler: Adaptive hierarchical scheduling in apache storm. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 26. ACM, 2016.

[31] H. Takabi F. Abazari, M. Analoui and S. Fu. Mows: Multi-objective workflow scheduling in cloud computing based on heuristic algorithm. *Simulation Modelling Practice and Theory*, 93:119 – 132, 2019.

[32] T. Hacker and K. Mahadik. Flexible resource allocation for reliable virtual cluster computing systems. In *Proc. of the ACM/IEEE Supercomputing Conference*, pages 48:1–48:12, 2011.

[33] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the use of cloud computing for scientific workflows. In *Proc. of the 4th IEEE Int. Conf. on eScience*, pages 640–645, Washington, DC, USA, 2008.

[34] A. Gleave R.N.M. Watson I. Gog, M.Schwarzkopf and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of Operating Systems Design and Implementation*, pages 99–115, Savannah, GA, November 2016. USENIX Association.

[35] E. Thompson J. and G.L. Bretthorst. *Probability theory : the logic of science.* Cambridge University Press, Cambridge, UK, New York, 2003.

[36] Q. Jiang, Y.C. Lee, and A.Y. Zomaya. Executing large scale scientific workflow ensembles in public clouds. In *Proc. of the 44th Int. Conf. on Para. Proc.*, Beijing, China, September 1-4 2015.

[37] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. Berman, and P. Maechling. Data sharing options for scientific workflows on amazon ec2. In *Proc. of the ACM/IEEE Supercomputing Conf.*, pages 1–9, Washington, DC, USA, 2010.

[38] P. Cong P. Lu T. Wei J. Zhou, T. Wang and M. Chen. Cost and makespan-aware workflow scheduling in hybrid clouds. *Journal of Systems Architecture*, 100:101631, 2019.

[39] L. Bao, C. Wu, X. Bu, N. Ren and M.Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30:2114 – 2129, 2019.

[40] X. Lin and C.Q. Wu. On scientific workflow scheduling in clouds under budget constraint. In *Proc. of the 42nd Int. Conf. on Para. Proc.*, Lyon, France, Oct. 1-4 2013.

[41] Y. Xie Y. Tan J. Walda Z. Zhao L. Li, J. Tan and D. Gajewski. Waveform-based microseismic location using stochastic optimization algorithms: A parameter tuning workflow. *Computers Geosciences*, 124:115 – 127, 2019.

[42] Z.A. Mann. Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms. *ACM Computing Surveys*, 48(1):Article No. 11, Sept. 2015.

[43] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proc. of Supercomputing Conf.*, pages 49:1–49:12, 2011.

[44] M. Mao, J. Li, and H. Marty. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing*, pages 41–48, Oct. 2010.

[45] S. Moon, J. Lee, X. Sun, and Y. Kee. Optimizing the hadoop mapreduce framework with high-performance storage devices. *The Journal of Supercomputing*, 71(9):3525–3548, 2015.

[46] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. Yahoo s4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, 2010.

[47] M.A. Rodriguez and R. Buyya. Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds. *IEEE Trans. on Cloud Computing*, 2(2), 2014.

[48] A. Sekhar, B.S. Manoj, and C.S.R. Murthy. A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks. In *Proc. of Int. Workshop on Dist. Comp.*, pages 63–74, 2005.

[49] J. Shi, J. Luo, F. Dong, J. Zhang, and J. Zhang. Elastic resource provisioning for scientific workflow scheduling in cloud under budget and deadline constraints. *Cluster Computing*, 19(1):167–182, 2016.

[50] W.C. Skamarock, J.B. Klemp, J. Dudhia, D.O. Gill, D.M. Barker, M.G. Duda, X. Huang, W. Wang, and J.G. Powers. A description of the advanced research wrf version 3. Technical Report NCAR/TN-475+STR, National Center for Atmospheric Research, Boulder, Colorado, USA, June 2008.

[51] W. Tian, G. Li, W. Yang, and R. Buyya. Hscheduler: an optimal approach to minimize the makespan of multiple mapreduce jobs. *The Journal of Supercomputing*, 72(6):2376–2393, 2016.

[52] I. Gupta V. Singh and P.K. Jana. An energy efficient algorithm for workflow scheduling in iaas cloud. *Journal of Grid Computing*, 18:357 – 376, 2020.

[53] C.Q. Wu and H. Cao. Optimizing the performance of big data workflows in multi-cloud environments under budget constraint. In *Proc. of the 13th IEEE SCC*, San Francisco, USA, June 27 - July 2 2016.

[54] C.Q. Wu, X. Lin, D. Yu, W. Xu, and L. Li. End-to-end delay minimization for scientific workflows in clouds under budget constraint. *IEEE Trans. on Cloud Comp.*, 3(2):169–181, 2015.

[55] H. Xu X. Ma, H. Gao and M. Brian. An iot-based task scheduling optimization scheme considering the deadline and cost-aware scientific workflow for cloud computing. *Journal on Wireless Communications and Networking*, 249:1687 – 1499, 2019.

[56] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544, 2014.

[57] Y. Wang Y. Cheng R. Xu A.S. Sani D. Yuan Y. Xie, Y. Zhu and Y. Yang. A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloudedge environment. *Future Generation Computer Systems*, 97:361 – 378, 2019.

[58] L. Zeng, V. Bharadwaj, and X. Li. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Int. Conf. on Advanced Information Networking and Applications*, volume 0, pages 534–541, Los Alamitos, CA, USA, 2012.

[59] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. In *Proc. of VLDB Endow*, volume 5, pages 1184–1195, July 2012.

[60] W. Yu, Y. Lin J. Zhang, T. Gu, Z. Wang, Z. Zhan and J. Zhang. Dynamic group learning distributed particle swarm optimization for large-scale optimization and its application in cloud workflow scheduling. *IEEE Transactions on Cybernetics*, 50:2715–2729, 2020.