Pervasive and Mobile Computing 60 (2019) 101082

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Maximizing MapReduce job speed and reliability in the mobile cloud by optimizing task allocation

Jin-woo Lee [a], Gwangseon Jang [b], Hohyun Jung [c], Jae-Gil Lee [a,*], Uichin Lee [a]

[a] Graduate School of Knowledge Service Engineering, KAIST, Republic of Korea
[b] Digital Solution Laboratory, KEPCO Research Institute, Republic of Korea
[c] Department of Mathematical Sciences, KAIST, Republic of Korea

ARTICLE INFO

Article history:
Received 16 February 2019
Received in revised form 6 September 2019
Accepted 10 September 2019
Available online 3 October 2019

Keywords:
Mobile cloud computing
Task allocation
MapReduce
Mobile ad-hoc cloud
Fault tolerant computing

ABSTRACT

Mobile cloud computing has become a widespread phenomenon owing to the rapid development and proliferation of mobile devices all over the globe. Furthermore, revolutionary mobile hardware technologies, such as 5G and IoT, have led to increased competition for mobile intelligence among tech-giants, like Google, Apple, and Facebook, further leading to developments in the field of mobile cloud computing. However, several challenges still remain; above all, resolving the task allocation problem that determines the nodes on which tasks will be executed is of paramount importance, and therefore, is the focus of many previous studies on mobile cloud. To this end, we propose a novel *Mobile MapReduce Task Allocation (MTA)* strategy that simultaneously maximizes both job speed and reliability by modeling communication delay and task reliability. Based on extensive evaluations using various task allocation strategies, representative workloads, and real mobility traces on a mobile MapReduce simulator validated against a platform running on an actual smartphone cluster, we show that MTA significantly outperformed the state-of-the-art task allocation algorithms by up to 3.7 times and 41%, respectively, in terms of job speed and reliability, confirming its resource efficiency and scalability as well.

1. Introduction

1.1. Background and motivation

1.1.1. Mobile computing

A single Apple Watch today surpasses the computing ability of a supercomputer from 30 years ago, owing to the rapid advancement in mobile computing technology in recent times [1]. By 2021, the total number of mobile devices that will be connected to the Internet is estimated to approximately reach 12 billion [2], which is one and a half times the projected world population in that year. The success of Pokemon Go, a virtual reality mobile game, can be attributed to proliferation of advanced sensors on modern smartphones [3]. Furthermore, companies like Google, Apple, Facebook, and Baidu have already released their AI libraries for mobile environments to the general public [4–6]. Deep learning, which recently gained significant attention from stakeholders in the field of AI, has already been introduced to several

* Corresponding author.
E-mail addresses: jinwoo.lee@kaist.ac.kr (J.-w. Lee), gwangseon.jang@kepco.co.kr (G. Jang), hhjung@kaist.ac.kr (H. Jung), jaegil@kaist.ac.kr (J.-G. Lee), uclee@kaist.ac.kr (U. Lee).

https://doi.org/10.1016/j.pmcj.2019.101082
1574-1192/

mobile applications. Identifying medicines using pictures captured on smartphones to avoid drug administration errors [7] or sensing obstacles for seniors with poor vision using smart glasses [8] are examples of the benefits of these deep-learning-based applications. However, even with these developments in mobile computing, further improvements are still necessary. For example, current smartphones can only process four out of 15 frames per second, when real-time face recognition is performed on videos [9]. Moreover, feature extraction in mobile image searches or object recognition based on mobile deep learning can take more than 100 s [8].

1.1.2. Mobile cloud computing

To overcome the abovementioned limitations of mobile computing, cloud-based mobile computing, which involves offloading of data and operations on to a cloud, was introduced [10]; in this approach, the mobile client device only functions as an interface, while, in reality, the operations are performed on the cloud, which includes abundant computational and energy resources. This mobile computing approach can be classified into four categories based on two dimensions referred to as locality (remote/local) and mobility (fixed/mobile). In particular, *remote-fixed* cloud is a conventional cloud with multiple servers on a remote site; in contrast, *local-mobile* cloud is a movable cloud based on a mobile ad-hoc network (MANET) that is dynamically established upon the agreement of nearby device users. *Remote-mobile* cloud is not considered further herein because it is rarely used. *Local-fixed* cloud, also called fog or edge computing, provides resources using local servers; however, is out of the scope of this paper.

Our study is primarily related to the *local-mobile* cloud, which has several advantages over the other types of clouds. The *local-mobile* cloud is based on device-to-device communication technology (e.g., Wi-Fi Direct or Bluetooth), which has features of free communication, short latency, low energy consumption, and low probability of privacy violations [11]. In the remainder of this paper, for convenience, the *remote-fixed* and *local-mobile* clouds are referred to as *conventional cloud* and *mobile cloud*, respectively.

Furthermore, mobile cloud computing can be classified into two types based on the time-intensiveness of a workload [10]. In particular, when the workload is not time-intensive, a job for the workload can be processed on a single device; for example, a certain speaker in a room can be quickly recognized by a smartphone with high accuracy based on shared models that are individually trained on each device [12]. On the contrary, when the workload is time-intensive, mobile cloud computing can involve collaboratively processing a job by splitting it into several sub-tasks that are performed by multiple mobile devices for fast job processing; for example, Skyship [13] flies a blimp as an infrastructure coordinator to create an ad-hoc 5G network in disastrous situations, and dispatches drones to the situation areas, allowing it to process a large collection of images quickly. We categorize real-world workloads by the time-intensiveness in Section 6.1.4 and discuss the applications and use cases for the workloads in Appendix A.

1.1.3. Distributed computation model in the mobile cloud

The distributed computation models primarily used in mobile cloud computing can be classified into Master–Slave, MapReduce, and Dataflow [9,10,14]. First, the Master–Slave is the simplest one in which a master distributes tasks to multiple slaves and collects the processed results [15]; this model is inappropriate for use in a mobile cloud because the master device could become a bottleneck. Next, MapReduce is the most universally used model that represents a job using map tasks for filter, transformation, or expansion operations and reduce tasks for aggregation operations [16]. The abstraction simplicity of map and reduce enables high scalability and fault tolerance [17]. Lastly, Dataflow is the most expressive model that represents any job as a directed acyclic task graph [18]. MapReduce and Dataflow show similar processing performance; nevertheless, MapReduce is more fault-tolerant than the Dataflow model because of the associated overhead cost of managing tasks as a complicated task graph in the latter [19], when these models are compared using Hadoop and Spark, which are representative execution platforms for MapReduce and Dataflow, respectively, and without considering special platform features like memory caching. Therefore, in our study, first, we solely consider MapReduce, and will extend our approach to Dataflow in a later study. Hereafter, MapReduce in the mobile cloud is referred to as *Mobile MapReduce*.

1.1.4. Task allocation challenges of mobile MapReduce

The primary topics in mobile cloud computing include task allocation, fault tolerance, and energy awareness. Among these, resolving the task allocation problem, which involves determining the nodes on which tasks are executed, is of paramount importance; this is clear considering that the highest portion of mobile cloud computing studies were on this topic or at least covered it as an issue, as summarized in Appendix B. In a similar vein, we identified two task allocation challenges that arose when the MapReduce task allocation strategy of the conventional cloud was used in the mobile cloud, owing to differences in cloud characteristics; Fig. 1 illustrated these differences from the perspective of network contention and mobility.

- **Task Allocation Challenge 1 (Long Communication Delay)**: In general, a conventional cloud connects hundreds of geo-distributed racks through routers arranged in a tree hierarchy and each rack, in turn, connects tens of servers. In contrast, a mobile cloud uses mesh topology with multi-hop connectivity between mobile devices in which each device itself concurrently acts as a server and router. It should be noted that, due to contention between communication channels and relatively low bandwidth, communication quality is relatively poor in mobile clouds [20]. Furthermore, the conventional MapReduce task allocation strategy prioritizes hop-distance based data locality

(a) Conventional cloud **(b)** Mobile cloud

Fig. 1. Comparison of conventional and mobile clouds in terms of network contention and mobility. Contention (red) appears as a shaded circle over device E. Mobility (blue) is represented by the cloud shape and node departure.

[21], whereas in the mobile cloud, performance degradation can occur when the conventional strategy is used [22]. For example, if the data in storage server A of a conventional cloud needs to be processed in one of the processing servers B, C, or D, as shown in Fig. 1(a), the closest server B processes the data. In contrast, when the data in devices A and B of a mobile cloud needs to be processed in devices C and D, as shown in Fig. 1(b), even though hop distances of A–D and B–C pairs are shorter than those of A–C and B–D pairs, communication delay between the former pairs can be longer than that of the latter pairs because device E, which is an intermediate device for communication, generates contention. Therefore, when the MapReduce task allocation strategy in a mobile cloud is based on the conventional hop distance, communication between tasks might be delayed due to contention, which can consequently result in low job speed.

- **Task Allocation Challenge 2 (Low Job Reliability)**: Considering network mobility, because network topology in a mobile cloud changes as nodes join and leave the cloud, a task fails if the task-allocated node leaves the cluster. Thus, because the conventional MapReduce task allocation strategy assumes a fixed topology, job reliability can be low because of task failures in the mobile cloud [14].

To the best of our knowledge, there is currently no specific task allocation strategies for *Mobile MapReduce*, aside from those based on the conventional MapReduce. For example, well-known mobile MapReduce platforms, such as Hyrax and Misco, are based on Hadoop [14,23,24], which is the most widely used conventional MapReduce platform. Furthermore, other than the importance of addressing this issue, finding a solution for it is not straightforward, because the complex characteristics of both conventional cloud and mobile cloud need to be considered [10].

1.2. Key contributions, performance summary, and outline

In this study, we propose a novel *Mobile MapReduce Task Allocation* (**MTA**) strategy to solve the two *Task Allocation Challenges* specified earlier. Our primary objective is to improve job speed and reliability by considering network contention and mobility: *Contention Awareness* & *Mobility Awareness*. Based on the *Contention Awareness* principle, we aim to model contention-based communication delays in a mobile cloud and allocate tasks to nodes that can reduce the delay, consequently increasing job speed. In addition, based on the *Mobility Awareness* principle, we identify unreliable nodes with high mobility and then allocate tasks by giving priority to reliable nodes instead. The increased reliability of task processing or communication between tasks can, in turn, lead to increased job reliability. The contributions of our study are summarized as follows:

- To the best of our knowledge, MTA is the first task allocation strategy to simultaneously consider both job speed and reliability issues in the mobile cloud. The application of MTA can lead to improved synergy, and consequently, better performance in the mobile cloud.
- We extensively evaluate the efficiency of MTA using state-of-the-art strategies (Hadoop, Purlieus, Mantri, and HPSO) on representative mobile cloud computing workloads (Naive Bayes, SIFT, and Join) and hundreds of real mobility traces (Conference, Village, and Race). In addition, we developed a new task allocation simulator for mobile MapReduce and validated it against a mobile MapReduce platform running on an actual smartphone cluster. We have released our software including the algorithms and simulator to help other researchers studying task allocation in a mobile cloud. The software is available at https://github.com/kaist-dmlab/mta.
- Our results show that the proposed MTA significantly outperformed the state-of-the-art task allocation algorithms by up to 3.7 times and 41% in terms of job speed and reliability, respectively. Furthermore, MTA is resource-efficient as it can improve performance with fewer cluster resources than other algorithms and is scalable to input data size.

The remainder of this paper is organized as follows. In Section 2, previous task allocation studies related to both conventional and mobile clouds are discussed. The task allocation problem for *Mobile MapReduce* is defined in Section 3 and its theoretical analysis is presented in Section 4. The design and evaluation details of MTA are described in Sections 5 and 6, respectively. Finally, Section 7 provides our conclusion.

Table 1
Summary of notations.

Notation	Description
$\mathbb{N}$, $\mathbb{L}$	Set of all nodes and set of all links in a mobile cloud
$\mathbb{T}$, $\mathbb{E}$	Set of all tasks and set of all edges between tasks in a MapReduce job
$\mathbb{N}_P$	Set of nodes with allocated tasks in phase P
$\mathbb{T}_P$	Set of tasks in P
$\mathbb{TA}_P$	Set of task allocations in P. $\mathbb{TA}_P = \{(T_1, N_1), (T_2, N_2), \ldots\}$
K	Number of task allocations. $K = \lvert\mathbb{TA}_P\rvert = \lvert\mathbb{N}_P\rvert = \lvert\mathbb{T}_P\rvert = \lvert\mathbb{N}\rvert \times ClusterUtilization$
$d_{S,i}$	Success duration, or delay, of the ith task
$d_{F,i}$	Failure duration, or delay, of the ith task
r_i	Reliability of the ith task
λ_i	Failure rate of the node to which the ith task is allocated

2. Related work

2.1. Task allocation in the conventional cloud

Task allocation in the conventional cloud is primarily aimed at increasing job speed based on a certain data locality. In particular, Hadoop [21] allocates map tasks to nodes that minimize hop-distance based data locality and reduce tasks to random nodes. LARTS [25] improves the random reduce task allocation by identifying nodes with the largest number of map outputs in order to increase reduce task locality. Purlieus [26] allocates map tasks based not only on data locality, but also on connected components, according to three representative workload types; in addition, it allocates reduce tasks to the same nodes as map tasks. Mantri [27] follows the same approach as Hadoop in terms of map task allocation, but maximizes node communication speed for reduce task allocation. Spark [28] implements data locality on a resilient distributed dataset (RDD), which is a collection of data partitioned across nodes; for example, ShuffledRDD, an RDD for representing shuffle communication, allocates reduce tasks by following the same objective as LARTS. Furthermore, in a heterogeneous conventional cloud, task allocation strategies consider not only job speed, but also job reliability and energy conservation. For example, HPSO [29] allocates tasks to maximize job reliability by modeling both processor and communication reliability. In contrast, the approach proposed by Datla et al. [30] minimizes both job makespan and energy consumption.

It is important to note that many conventional task allocation strategies based on well-known heuristic algorithms such as A* [31,32] or dynamic programming [33] are inadequate because of the weakly optimal substructure of our mobile task allocation problem. Please refer to Section 5.2 for details.

2.2. Task allocation in the mobile cloud

In general, a majority of mobile cloud computing studies (e.g., Hyrax or Mobile Storm) [9,14,23,24] are based on conventional task allocation strategies from conventional cloud computing platforms (e.g., Hadoop or Storm). In contrast, thus far, few studies on task allocation research have considered the characteristics of mobile cloud. MobiDic [34] allocates tasks to nearby mobile devices in a round-robin fashion. Scavenger [35] allocates tasks to minimize job completion time based on the assumption of heterogeneous communication bandwidth and processing power of mobile devices. Serendipity [36] minimizes either job completion time or energy consumption. However, to the best of our knowledge, no effort has been made in designing a task allocation strategy focusing on the *Task Allocation Challenges* faced in a mobile cloud.

3. Problem statement

In this section, we define the task allocation problem for *Mobile MapReduce*. Table 1 lists the notations used in this paper.

In a mobile cloud, as time elapses, nodes might join and leave, and links might connect and break, resulting in changes to the topology; in particular, the topology of a mobile cloud can be represented as an undirected graph consisting of nodes $N \in \mathbb{N}$ and links $L \in \mathbb{L}$. Because task allocation must be completed in an instant to meet the responsiveness requirement of mobile cloud computing [37], all the variables for an allocation are assumed to be for the same time, and thus, time notations are not denoted on the variables.

A MapReduce job can be represented as a directed acyclic graph consisting of tasks $T \in \mathbb{T}$ and edges connecting the tasks $(T_j, T_k) \in \mathbb{E}$. Tasks in a MapReduce job can be either *Map* or *Reduce* tasks, and the same type of tasks are processed together in a set called phase $P \in \{Map, Reduce\}$ [21]. In particular, map tasks primarily involve filter, transformation, or expansion operations, while reduce tasks involve aggregation operations [16]. In general, a task $T_P \in \mathbb{T}_P$ belonging to phase P initially receives the previous outcomes from tasks in the previous phase, performs the current operation for the received data, and sends the current outcome to tasks in the next phase.

Task allocation involves determining a set of task and node pairs $\mathbb{TA}_P = \{(T_1, N_1), (T_2, N_2), \ldots\}$ that describes the nodes on which tasks are executed. The allocation of tasks to certain nodes depends on the objective of *Mobile MapReduce*. As indicated in the *Task Allocation Challenges*, job speed, job reliability, or energy conservation can be regarded as the objective of *Mobile MapReduce*. Thus, a task allocation problem refers to the optimization process of finding a certain task allocation arrangement that best achieves the set objective, formally defined by Definition 1.

Definition 1 (*Task Allocation Problem*)**.** When a phase P of a MapReduce job begins, for all candidate task allocation sets $\mathbb{TA}_P \in \mathcal{TA}_P$ generated from the given tasks $\mathbb{T}_P$ and all nodes $\mathbb{N}$ in a mobile cloud, a certain task allocation arrangement $\mathbb{TA}_P{}^*$ is selected such that it minimizes the associated *Cost*, which can be expressed as Eq. (1).

$$\begin{aligned} \mathbb{TA}_P{}^* = \underset{\mathbb{TA}_P \in \mathcal{TA}_P}{\operatorname{argmin}} \; Cost\left(\mathbb{TA}_P\right) \quad \text{s.t.} \quad & 0 < ClusterUtilization \leq 1 \\ & |\mathbb{TA}_P| = |\mathbb{N}| \times ClusterUtilization \end{aligned} \tag{1}$$

In a mobile cloud, because all nodes are not always available due to network unreliability or resource poverty in a cluster ($0 < ClusterUtilization \leq 1$) [10], not all nodes in the cluster participate in job processing, instead only a subset thereof does. Hereafter, K refers to the number of task allocations $|\mathbb{TA}_P|$ determined by $|\mathbb{N}| \times ClusterUtilization$. In addition, K equals $|\mathbb{N}_P|$ or $|\mathbb{T}_P|$.

Task allocations for all phases are conducted independently.

4. Theoretical analysis of task allocation problem

In this section, we describe our proposed novel optimization cost for *Mobile MapReduce* and derive the associated properties by analyzing the task allocation problem from the perspective of *Mobile MapReduce*. First, in order to simplify mathematical formulation, we prescribe three assumptions for relatively minor issues compared with and aside from the *Task Allocation Challenges* in a mobile cloud even though the issues are frequently addressed in a conventional cloud [18,21,28]. These assumptions are listed as follows:

- **Assumption 1 (Homogeneity)**: Each node has equal processing power and equal maximum communication bandwidth. However, this assumption on homogeneity can be relaxed in Eq. (5).
- **Assumption 2 (Balanced Load)**: Each task is assigned an equal amount of workload. This assumption on balanced load can also be relaxed in Eq. (5).
- **Assumption 3 (Task Synchronization)**: The beginning of tasks in a phase are synchronized to the same time. This is because, in the case of map tasks, all of them begin as a job starts. In contrast, reduce tasks cannot start until all the map tasks have finished processing all key-value pairs, because, otherwise, there is no guarantee that all values associated with the key of the reduce tasks would have been processed to be sorted and grouped by the key [21]. Furthermore, because modeling asynchronous tasks additionally requires expected states of tasks in the next phase, such modeling is relatively harder and will be studied in a future work.

Next, more importantly, for the problem analysis, we establish two principles to resolve the *Task Allocation Challenges*, which are specified as follows:

- **Principle 1 (Contention Awareness)**: Tasks should be allocated to nodes that decrease contention-based communication delay.
- **Principle 2 (Mobility Awareness)**: Tasks should be allocated to reliable nodes by determining network mobility.

Considering both abovementioned principles, we introduce *Time to Phase Success* (*TPS*) cost, which can account for communication delays between tasks as well as task reliability simultaneously. Based on a universally adopted failure recovery convention in which a failed task is restarted on another node until it is successfully completed [21,28], the probability of task success at the jth ($j = 1, 2, \ldots$) trial is

$$(1 - r_i)^{j-1} r_i, \tag{2}$$

where r_i is the task reliability. Then, we define the time to task success averaged over all K tasks in the same phase as *TPS*, which can be expressed as

$$TPS = \frac{1}{K} \sum_{i=1}^{K} \sum_{j=1}^{\infty} \left(d_{F,i}\,(j-1) + d_{S,i}\right) (1 - r_i)^{j-1} r_i, \tag{3}$$

where $d_{S,i}$ and $d_{F,i}$ denote the duration, or delay, of the task success and failure, respectively. By replacing $\sum_{j=1}^{\infty} j\,(1 - r_i)^{j-1} r_i$ with $1/r_i$ and $\sum_{j=1}^{\infty} (1 - r_i)^{j-1} r_i$ with 1, *TPS* becomes

$$TPS = \frac{1}{K} \sum_{i=1}^{K} \left\{ d_{S,i} + \left(\frac{1}{r_i} - 1\right) d_{F,i} \right\}. \tag{4}$$

Thus, *TPS* can account for effects from both task delay and task reliability factors.

We can express the three variables $d_{S,i}$, r_i, and $d_{F,i}$ in further detail. First, task success delay $d_{S,i}$ is the sum of communication and processing delays. Based on the *Homogeneity* and *Balanced Load* assumptions, the processing delay d_{Proc} is same for all tasks. However, the communication delay is highly likely to be different for all communication paths because of differences in contention. Thus, $d_{S,i}$ can be expressed as

$$d_{S,i} = d_{Comm,i} + d_{Proc}. \tag{5}$$

In Section 5.1, we introduce a method to determine communication delay $d_{Comm,i}$. To relax the assumptions on *homogeneity* and *balanced load*, d_{Proc} in Eq. (5) needs to be extended to $d_{Proc,i} = InputSize_i/ProcessingThroughput_i$ by considering the input size and throughput of each task.

Second, assuming that the cluster departure time of the task-allocated node follows an exponential distribution, which is most commonly used for modeling reliability [38], r_i can be expressed as

$$r_i = \exp\left(-\lambda_i d_{S,i}\right), \tag{6}$$

where λ_i is the task failure rate; we introduce a method to determine this in Section 5.1.

Lastly, $d_{F,i}$ can be deduced from the relationship of $d_{S,i}$ and r_i. According to the reliability theory [38], the probability density function of task failure is

$$\text{FailurePdf}(t) = \frac{d}{dt}\left[1 - \exp\left(-\lambda_i t\right)\right] = \lambda_i \exp\left(-\lambda_i t\right). \tag{7}$$

Then, $d_{F,i}$, which is the average of t during $d_{S,i}$, can be represented as

$$d_{F,i} = \int_0^{d_{S,i}} t \times \text{FailurePdf}(t)\, dt = \int_0^{d_{S,i}} \lambda_i t \exp\left(-\lambda_i t\right) dt = \left(-d_{S,i} - \frac{1}{\lambda_i}\right)\exp\left(-\lambda_i d_{S,i}\right) + \frac{1}{\lambda_i}. \tag{8}$$

Consequently, by replacing $d_{S,i}$, r_i, and $d_{F,i}$ in Eq. (4) with Eqs. (5), (6), and (8), respectively, *TPS* becomes

$$TPS = \frac{1}{K}\sum_{i=1}^{K}\left\{\frac{1}{\lambda_i}\exp\left(\lambda_i d_{S,i}\right) + \left(d_{S,i} + \frac{1}{\lambda_i}\right)\exp\left(-\lambda_i d_{S,i}\right) - \frac{2}{\lambda_i}\right\}. \tag{9}$$

Eq. (9) is utilized as the optimization cost, which can be understood through Lemma 1.

Lemma 1. *Minimizing TPS is the same as maximizing task speed and task reliability for all tasks.*

Proof. By partially differentiating Eq. (9) on each $d_{S,i}$, we obtain

$$\frac{\partial TPS}{\partial d_{S,i}} = \exp\left(\lambda_i d_{S,i}\right) - \lambda_i d_{S,i}\exp\left(-\lambda_i d_{S,i}\right). \tag{10}$$

Because the range of the equation is positive in the given domain, $\lambda_i > 0$ and $d_{S,i} > 0$, thus *TPS* is increasing for $d_{S,i}$. By partially differentiating Eq. (9) on each λ_i, we obtain

$$\frac{\partial TPS}{\partial \lambda_i} = \frac{1}{{\lambda_i}^2}\left\{\left(\lambda_i d_{S,i} - 1\right)\exp\left(\lambda_i d_{S,i}\right) - \left({\lambda_i}^2 {d_{S,i}}^2 + \lambda_i d_{S,i} + 1\right)\exp\left(-\lambda_i d_{S,i}\right) + 2\right\}. \tag{11}$$

Because the range of the equation is positive in the given domain, $\lambda_i > 0$ and $d_{S,i} > 0$, thus *TPS* is increasing for λ_i. Therefore, minimizing *TPS* is the same as minimizing $d_{S,i}$ and λ_i for all tasks, i.e., based on Eqs. (5) and (6), minimizing *TPS* is the same as maximizing task speed and task reliability for all tasks. ■

The effect of minimizing *TPS* can be explained using Theorem 1.

Theorem 1. *Minimizing TPS implies maximizing job speed and reliability.*

Proof. First, minimizing *TPS* of each independent phase in a job implies minimizing job duration, and, in turn, maximizing job speed. Next, because job reliability is the product of reliability of all tasks in a job [38], maximizing task reliability by minimizing *TPS* as in Lemma 1 implies maximizing job reliability. The converse does not hold. ■

Furthermore, a unique property of the task allocation problem can be derived as presented in Theorem 2. This property should be considered for choosing a task allocation strategy in a mobile cloud.

Theorem 2. *The task allocation problem in a mobile cloud exhibits weakly optimal substructure.*

Proof. A problem is said to exhibit weakly optimal substructure if at least one optimal solution can be constructed from the optimal solutions of its subproblems [39]. When we determine a new task allocation (T_2, N_2) after a predetermined one

(T_1, N_1), the predetermined communication delay $d_{Comm,1}$ can change if contention arises on the shared communication paths of the two task allocations for reasons stated in *Task Allocation Challenge 1*. However, in the case that the communication paths are not shared; contention does not arise at all; a communication delay of a sub-task allocation does not influence another; and therefore, at least one set of task allocations can be deterministically constructed from the sub-task allocations. ■

5. Proposed task allocation strategy

In this section, we present our proposed novel *Mobile MapReduce Task Allocation* (**MTA**) strategy to solve the task allocation problem in mobile cloud computing, which is known to be strongly NP-Hard [32]. If N denotes the number of all nodes in a mobile cloud and K denotes the number of tasks to be allocated, the complexity of the entire search $({}_NP_K)$ is the product of the complexity of a 0–1 knapsack problem $({}_NC_K)$ and that of a traveling salesman problem $(K!)$, which involve selecting certain nodes for the tasks and ordering the tasks for each node selection, respectively, and both these problems are known to be NP-Hard [39].

In addition, we derive two heuristics of the MTA based on whether cluster utilization is known. This is because the cluster utilization is typically specified in the job requirements in a conventional cloud, whereas it should not be specified in the mobile cloud for a dynamically changing resource caused by mobility [10]. Thus, we refer to MTA with static and dynamic cluster utilization as MTA-S and MTA-D, respectively.

In Section 5.1, we discuss the estimation of the aforementioned undecided variables. Then, we present MTA-S that involves maximizing job speed and reliability based on genetic node selection and greedy task ordering in Section 5.2 and MTA-D that involves efficiently exploring the entire cluster utilization range to further maximize job speed and reliability in Section 5.3.

5.1. Variable estimation

Before designing the heuristic, we set the failure rate λ_i and communication delay $d_{Comm,i}$, which were introduced in Section 4. Because predicting these variables is beyond the scope of this study, we estimate these variables from history information instead, which is a common convention in the field [22,27].

In particular, we trace mobility history, including node departure or link breakage, for all nodes and estimate λ_i by taking the reciprocal of the expected time to failure of the node N_i based on the traced mobility history. Thus, the estimated failure rate $\hat{\lambda}_i$ can be expressed as

$$\hat{\lambda}_i = \frac{f_i}{\sum_j TimeToFailure_{ij}}, \tag{12}$$

where f_i is the number of failures of N_i and $TimeToFailure_{ij}$ is the jth time to failure of N_i.

Furthermore, we trace bandwidth history for all links and estimate $d_{Comm,i}$ in terms of link bandwidth, which, in turn, can be estimated by averaging the traced link bandwidth history. Based on the *Task Synchronization* assumption, $d_{Comm,i}$ for a task allocation (T_i, N_i) is determined as the maximum edge delay from all edges $(T_i, T_j) \in \mathbb{E}_{T_i}$ of the task T_i communicating with other tasks T_j [21]. In addition, the bandwidth of the edge (T_i, T_j) is determined as the minimum link bandwidth from all links $L_k \in (N_i, N_j)$ in the shortest path composed of the edge. Thus, the estimated communication delay $\hat{d}_{Comm,i}$ can be expressed as

$$\hat{d}_{Comm,i} = \max_{(T_i,T_j)\in\mathbb{E}_{T_i}} \frac{o_j}{\min_{L_k\in(N_i,N_j)} \hat{b}_k}, \tag{13}$$

where o_j is the communication traffic between task T_i and T_j, which is same for all tasks based on the *Balanced Load* assumption, and $\hat{b}_k$ is the estimated bandwidth of L_k. Consequently, by replacing $d_{Comm,i}$ in Eq. (5) with Eq. (13), the estimated task success delay $\hat{d}_{S,i}$ can be expressed as

$$\hat{d}_{S,i} = \hat{d}_{Comm,i} + d_{Proc}. \tag{14}$$

5.2. MTA-S

The first heuristic MTA-S significantly reduces the search space of the task allocation problem by assuming static cluster utilization, i.e., a fixed number of task allocations K. However, the product of ${}_NC_K$ for node selection and $K!$ for task ordering is still high in the case of a mobile cloud; therefore, efficient search algorithms need to be employed.

MTA-S involves genetic node selection for the combinatorial outer search and greedy task ordering for the factorial inner search of each outer node selection. Thus, the genetic and greedy algorithms can be appropriately used for designing MTA-S regardless of the weakly optimal substructure property defined by Theorem 2. On the contrary, the conventional heuristic algorithms, such as A* [31,32] or dynamic programming [33], that require a problem to exhibit strongly optimal substructure are inadequate in the mobile cloud. In particular, for the node selection as the outer search, we adopt the

Algorithm 1 MTA-S (Overall Procedure)

INPUT: $\mathbb{T}_P$, $\mathbb{N}$
OUTPUT: $\left(\mathbb{TA}_P{}^*\ ,\ Cost_P{}^*\right)$
1: /* OUTER SEARCH: GENETIC NODE SELECTION */
2: $\left(\mathcal{N}_P = \left\{\mathbb{N}_{P,1}, \cdots, \mathbb{N}_{P,PopulationSize}\right\}\right) \leftarrow$ Sample K-sized $(|\mathbb{T}_P|)$ node subsets *PopulationSize* times
3: $\left(\mathbb{TA}_P{}^*\ ,\ Cost_P{}^*\right) \leftarrow \text{argmin}_{\mathbb{N}_P \in \mathcal{N}_P}$ EVALUATE_FITNESS $(\mathbb{T}_P\ ,\ \mathbb{N}_P)$
4: **while** NOT_FINISHED $\left(\mathcal{N}_P\ ,\ Cost_P{}^*\ ,\ MaxGenerations\right)$
5: $\left(\mathcal{N}_{P,S}, \mathcal{N}_{P,O}\right) \leftarrow$ SELECT $(\mathcal{N}_P)$
6: $\mathcal{N}_P \leftarrow \mathcal{N}_{P,S} +$ ALTER $\left(\mathcal{N}_{P,O}\right)$
7: $\left(\mathbb{TA}_P{}^*\ ,\ Cost_P{}^*\right) \leftarrow \text{argmin}_{\mathbb{N}_P \in \mathcal{N}_P}$ EVALUATE_FITNESS $(\mathbb{T}_P\ ,\ \mathbb{N}_P)$
8: **return** $\left(\mathbb{TA}_P{}^*\ ,\ Cost_P{}^*\right)$
9:
10: **function** EVALUATE_FITNESS($(\mathbf{T} = (T_1, \cdots, T_K)) \leftarrow \mathbb{T}_P\ ,\ (\mathbf{N} = (N_1, \cdots, N_K)) \leftarrow \mathbb{N}_P$)
11: /* INNER SEARCH: GREEDY TASK ORDERING */
12: $\left(\mathbf{T}^* = \left(T_1{}^*, \cdots, T_K{}^*\right)\right) \leftarrow$ GREEDY_TASK_ORDERING $(\mathbf{T}\ ,\ \mathbf{N})$ // Algorithm 2
13: $\mathbb{TA}_P{}^* \leftarrow \left\{\left(T_1{}^*, N_1\right), \cdots, \left(T_K{}^*, N_K\right)\right\}$
14: **return** $\left(\mathbb{TA}_P{}^*\ ,\ \text{TPS}\left(\mathbf{T}^*\ ,\ \mathbf{N}\right)\right)$
15: **function** TPS($\mathbf{T}\ ,\ \mathbf{N}$)
16: $\left(\hat{\lambda}_1, \cdots, \hat{\lambda}_K\right) \leftarrow$ Estimate failure rates of $\mathbf{T}$ on $\mathbf{N}$ based on Eq. (12)
17: $\left(\hat{d}_{S,1}, \cdots, \hat{d}_{S,K}\right) \leftarrow$ Estimate success delays of $\mathbf{T}$ on $\mathbf{N}$ based on Eq. (14)
18: **return** $\frac{1}{K}\sum_{i=1}^{K}\left\{\frac{1}{\hat{\lambda}_i}\exp\left(\hat{\lambda}_i\hat{d}_{S,i}\right) + \left(\hat{d}_{S,i} + \frac{1}{\hat{\lambda}_i}\right)\exp\left(-\hat{\lambda}_i\hat{d}_{S,i}\right) - \frac{2}{\hat{\lambda}_i}\right\}$ // Eq. (9)

genetic algorithm, one of the most widely used meta heuristics for task allocation research [29,33,40]. On the other hand, for the task ordering as the inner search, we adopt a sorting-based greedy algorithm instead of the genetic algorithm. This is because the complexity of the greedy algorithm $O(K^2)$ is significantly less than that of the genetic algorithm $O(MaxGenerations \cdot PopulationSize)$ in the case of a mobile cloud, where *MaxGenerations* is the maximum number of generations and *PopulationSize* is the population size per generation, and thus it can be considered that the greedy algorithm is suitable for real-time task allocation, as shown in Section 6.2.2. Task ordering in the mobile cloud is a greedy process, because it solely focuses on changing the order of two tasks at each comparison operation, ignoring the implicit influence of these changes on other tasks owing to corresponding changes in the shared communication paths. Finally, the overall complexity can be represented as $O(MaxGenerations \cdot PopulationSize \cdot K^2)$, which is in polynomial time. Algorithm 1 describes the overall procedure of MTA-S.

In more detail, MTA-S takes the task set $\mathbb{T}_P$ to be allocated for P and entire node set $\mathbb{N}$ in the mobile cloud as input and returns the best task allocation set $\mathbb{TA}_P{}^*$ that minimizes *TPS* as output. As specified earlier, MTA-S begins with genetic node selection for the outer search. First, node population $\mathcal{N}_P$ is initialized by randomly sampling K-sized node subsets *PopulationSize* times from $\mathbb{N}$; then, the fitness of the initial population is evaluated to determine the initial best task allocation set $\mathbb{TA}_P{}^*$ (Lines 2–3). Next, the genetic algorithm is used to select the survivor $\mathcal{N}_{P,S}$ and offspring $\mathcal{N}_{P,O}$ populations for recombination. Then, these populations are combined together; however, they are altered using crossover and mutation operations to create genetic diversity (Lines 5–6). A crossover operation combines the genetic information of two node sets to generate new sets, and a mutation operation alters one or more nodes in a node set while leaving others unchanged. Then, the fitness of the evolved node population $\mathcal{N}_P$ is evaluated (Line 7). Lines 4–7 of the algorithm are repeated until the population has evolved *MaxGenerations* times or the cost becomes constant. Finally, the outer search is terminated by returning the best task allocation set with its *TPS* cost (Line 8). The determination of the parameters for the genetic algorithm, including SELECT, ALTER, *PopulationSize*, and *MaxGenerations*, is discussed in Section 6.1.5.

The EVALUATE_FITNESS function is used for greedy task ordering for the inner search and calculates the *TPS* cost for the ordered tasks. To maintain task order during function execution, input parameters are passed as sets and then transformed into vectors (Line 10). Furthermore, the GREEDY_TASK_ORDERING function in Algorithm 2 performs the inner search, in essence, and returns the local best task order $\mathbf{T}^*$ (Line 12). This function then terminates by returning the local best task allocation $\mathbb{TA}^*$, which is obtained by pairing $\mathbf{T}^*$ with $\mathbf{N}$ element-wise (Lines 13–14).

For each outer node selection, greedy task ordering is performed in the MTA-S. In particular, we adopt the indexing approach of the bubble sort for the aforementioned reason. The task ordering procedure is described by Algorithm 2; it determines the best task order $\mathbf{T}^*$ with the minimum *TPS* based on a task order $\mathbf{T}$ and node order $\mathbf{N}$ as inputs from the outer search. In an iteration of the bubble sort (Lines 2–3), $T_{j-1}{}^*$ is swapped with $T_j{}^*$ from the current best order ($\mathbf{T}^*$) to define the expected next order ($\mathbf{T}_{Next}$) (Line 4), and $\mathbf{T}^*$ is compared with $\mathbf{T}_{Next}$ to update the current best order if *TPS* of the next one is lower than the current one (Lines 5–6). After the search is completed, the best task order is returned (Line 7).

Algorithm 2 Greedy Task Ordering (Inner Search)

INPUT: $(\mathbf{T} = (T_1, \cdots, T_K)\ ,\ \mathbf{N} = (N_1, \cdots, N_K))$
OUTPUT: $\mathbf{T}^*$
1: $\left(\mathbf{T}^* = \left({T_1}^*, \cdots, {T_K}^*\right)\right) \leftarrow (T_1, \cdots, T_K)$
2: **foreach** $i \in (1, \cdots, K)$
3: **foreach** $j \in (2, \cdots, K - i)$
4: $\left(\mathbf{T}_{Next} = \left(T_{Next,1}, \cdots, T_{Next,K}\right)\right) \leftarrow ({T_1}^*, \cdots, {T_j}^*, {T_{j-1}}^*, \cdots, {T_K}^*)$
5: **if** TPS $\left(\mathbf{T}^*\ ,\ \mathbf{N}\right) >$ TPS $\left(\mathbf{T}_{Next}\ ,\ \mathbf{N}\right)$ **then** // Algorithm 1
6: $\mathbf{T}^* \leftarrow \mathbf{T}_{Next}$
7: **return** $\mathbf{T}^*$

Algorithm 3 MTA-D (Overall Procedure)

INPUT: $\mathbb{N}$
OUTPUT: ${\mathbb{TA}_P}^*$
1: $\left(K_{Low}\ ,\ K_{High}\right) \leftarrow (1\ ,\ |\mathbb{N}|)$
2: **while** $K_{Low} <= K_{High}$
3: $\left(K_{Left}\ ,\ K_{Mid}\ ,\ K_{Right}\right) \leftarrow \left(\left(3 \times K_{Low} + K_{High}\right)/4\ ,\ \left(K_{Low} + K_{High}\right)/2\ ,\ \left(K_{Low} + 3 \times K_{High}\right)/4\right)$
4: $\left(\mathbb{T}_{Left} = \left\{T_1, \cdots, T_{K_{Left}}\right\}\ ,\ \mathbb{T}_{Right}\right) \leftarrow$ Generate K_{Left} and K_{Right} tasks
5: $\left(\mathbb{TA}_{Left}\ ,\ Cost_{Left}\ ,\ \mathbb{TA}_{Right}\ ,\ Cost_{Right}\right) \leftarrow \left(\text{MTA-S}\left(\mathbb{T}_{Left}\ ,\ \mathbb{N}\right)\ ,\ \text{MTA-S}\left(\mathbb{T}_{Right}\ ,\ \mathbb{N}\right)\right)$
6: **if** $Cost_{Left} < Cost_{Right}$ **then**
7: $\left(K_{High}\ ,\ {\mathbb{TA}_P}^*\right) \leftarrow \left(K_{Mid} - 1\ ,\ \mathbb{TA}_{Left}\right)$
8: **else**
9: $(K_{Low}\ ,\ {\mathbb{TA}_P}^*) \leftarrow \left(K_{Mid} + 1\ ,\ \mathbb{TA}_{Right}\right)$
10: **return** ${\mathbb{TA}_P}^*$

5.3. MTA-D

In contrast to MTA-S, which is a special case of MTA-D, MTA-D involves exploring the entire cluster utilization up to N to decide a certain K that is most suitable with the dynamically changing resource in consideration. Because linear search over N is inefficient, MTA-D utilizes binary search and performs MTA-S for each K selection. Therefore, the complexity of MTA-D is $O(\log_2 N \cdot MaxGenerations \cdot PopulationSize \cdot K^2)$, which is still in polynomial time. Algorithm 3 describes the overall procedure of MTA-D.

In particular, MTA-D takes the entire node set $\mathbb{N}$ as input and returns the best task allocation set ${\mathbb{TA}_P}^*$ as output, which has the minimum *TPS*, and consequently, the best K value. It begins by initializing the search range with the lower limit K_{Low} and higher limit K_{High} (Line 1). Then, the range is repeatedly divided into half to select the half containing the better task allocations until the range of K is reduced to a single value (Lines 2–9). In every iteration, three new quartiles are defined (Line 3). K_{Left} and K_{Right} denote the representative positions for each half and K_{Mid} denotes the middle position. Then, the task set $\mathbb{T}_{Left}$ and $\mathbb{T}_{Right}$ for each half are generated such that the given workload for the phase is divided into equal K_{Left} and K_{Right} parts (Line 4). At the end of each loop, the better half is selected based on the costs obtained from performing the MTA-S procedure for each half (Lines 5–9). Finally, the best task allocation set is returned (Line 10).

6. Evaluation

In our study, the evaluation was systematically conducted to support the following:

- MTA is **faster and more reliable** than other state-of-the-art task allocation algorithms (Section 6.2.1).
- MTA is **resource-efficient** (Section 6.2.2).
- MTA is **scalable** to input data size (Section 6.2.3).
- MTA is compared with task offloading approach (Section 6.2.4).
- MTA is evaluated with different fault tolerance conventions (Section 6.2.5).

6.1. Experimental setting

6.1.1. Configuration

We developed a mobile MapReduce simulator to extensively evaluate the efficiency of the proposed MTA with various task allocation strategies, representative workloads, and real mobility traces. We used Hadoop 1.2.1 to execute the distributed MapReduce, J-Sim 1.3 to simulate the mobile ad-hoc network, Jenetics 4.3.0 to perform the genetic algorithm,

Table 2
Compared algorithms.

Algorithm	Task allocation objective		Variable estimation method		Optimization
	Map	Reduce	$d_{Comm,i}$	λ_i	Algorithm
Hadoop [21] (baseline)	Data locality	Random	Hop distance	None	Greedy
Purlieus [26]	Connected components	Same nodes as map tasks	Hop distance	None	Greedy
Mantri [27]	Data locality	Max Comm. Speed	Node history	None	Greedy
HPSO [29]	Max job reliability		Hop distance	Mobility history	PSO
MTA (proposed)	Max job speed & reliability		Link history	Mobility history	GA

Table 3
Statistics of faultload data sets.

Data set	Category	# Nodes	Duration	# Faultloads	Avg. Group size	Avg. NCD
haggle[a]	Conference	78	3 days	455	19.2	0.18
pmtr[b]	Village	44	19 days	188	11.0	0.39
rollernet[c]	Race	62	3 h	99	21.4	0.60

[a] https://crawdad.org/cambridge/haggle/20090529.
[b] https://crawdad.org/unimi/pmtr/20081201.
[c] https://crawdad.org/upmc/rollernet/20090202.

and JDK 1.8.0_191 as the Java compiler. The simulator was validated against a mobile MapReduce platform running on an actual smartphone cluster as described in Appendix E.

6.1.2. Algorithms

In Table 2, MTA is compared with *four* state-of-the-art task allocation algorithms in terms of task allocation objective, variable estimation method, and optimization algorithm. We selected more relevant algorithms among those introduced in Section 2; in particular, we excluded algorithms if their task allocation objective led to negligible performance improvements in task allocation [25,34] or if they were considered relatively unimportant in the field of mobile cloud computing [30,35,36], as is specified in Appendix B.

Baseline Hadoop [21] allocates map tasks to nodes that minimize hop-distance based data locality and reduce tasks to random nodes; in contrast, Purlieus [26] allocates map tasks by considering data locality as well as connected components and reduce tasks to the same nodes as map tasks. Mantri [27] allocates map tasks in the same manner as Hadoop, but allocates reduce tasks in order to maximize node communication speed. Unlike the aforementioned algorithms, HPSO [29] allocates tasks without phase distinction to maximize job reliability by modeling both processor and communication reliability; lastly, the proposed MTA allocates tasks by concurrently maximizing job speed and reliability based on link bandwidth and mobility history. MTA is the only algorithm that supports not only static but also dynamic cluster utilization.

The last column in Table 2 specifies the method using which each algorithm is optimized. Furthermore, while Hadoop, Purlieus, and Mantri allocate tasks at once in a greedy fashion, HPSO and MTA search for optimal solutions using particle swarm optimization and genetic algorithm, respectively. To assign more importance to the task allocation objective, rather than the optimization algorithm in our evaluation, we permit both to converge even though this favors HPSO, because MTA with complexity $O(MaxGenerations \cdot PopulationSize \cdot K^2)$ finishes earlier than HPSO with complexity $O(MaxGenerations \cdot PopulationSize^2 \cdot K)$ given that $PopulationSize$ is assumed to be greater than K in both cases.

6.1.3. Faultloads

In our study, we prepare the group mobility trace using node mobility traces, because raw mobility data usually represents some form of individual node mobility, such as node position or node contact, as time elapses. A group mobility trace is constructed by combining similar connected components over time based on edge Jaccard similarity, which is the number of common edges divided by the number of edges that belong to at least one of the two connected components being considered. For example, as shown in Fig. 2, two connected components at t_1 are derived from a previous connected component at t_0 by an edge break between nodes A and B and an arrival of node C; edge Jaccard similarity (in the middle of each triangle) is evaluated for each new connected component. The previous connected component at t_0 is combined with the left connected component at t_1 whose edge Jaccard similarity (0.6) is greater than the minimum Jaccard coefficient (0.4, see Section 6.1.5 for its calculation). Consequently, this process generates thousands of group mobility traces that are a few hundred seconds long.

A group mobility trace is also referred to as a faultload, because the trace describes nodes on which a fault (i.e., disconnection) occurs as well as the time when it occurs [41]. We evaluated *three* faultload data sets that encompass various real situations, such as a conference, village, and race, thus exhibiting different statistics, as summarized in Table 3.

Because there is no universal measure of mobility, we define a new measure *Normalized Centrality Distance* (NCD) that can be used to evaluate faultload mobility using *graph centrality distance* [42]. NCD values for all faultloads are listed in

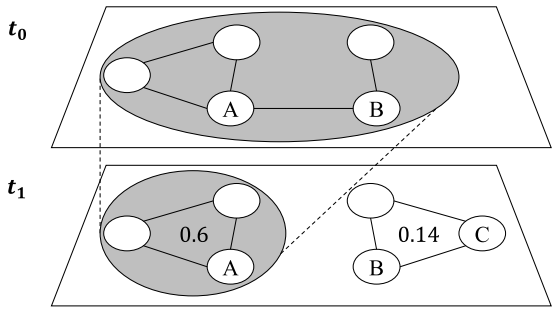


Fig. 2. Example of a group mobility trace construction from time t_0 to t_1. The group mobility trace is represented as a combination (dotted lines) of the similar connected components (gray circles).

Table 4
Statistics of workloads.

Workload		Naïve Bayes	SIFT	Join
Category		Decision making	Information retrieval	Sensor preprocessing
Map selectivity		0.04	1.5	0.6
Reduce selectivity		0.9	1	2
Map throughput (MB/s)		0.3	0.1	2
Reduce throughput (MB/s)		0.3	4.2	1
Input data size (MB)	haggle	35	10	60
	pmtr	40	13	70
	rollernet	55	20	100

the last column of Table 3. In particular, NCD value determination involves measuring the average of centrality distances between successive connected components for the entire faultload duration to account for the effects of node arrivals or departures to the group. To exclude the influence of group size in the NCD, we divide the average centrality distance by the average group size. Thus, NCD can be defined as Eq. (15).

$$\text{NCD}\,(\textit{Faultload}) = \frac{\textit{Average Centrality Distance over Time}}{\textit{Average Group Size over Time}} = \frac{\sum_t \sum_n |\text{C}\,(G_{t-1}, n) - \text{C}\,(G_t, n)|/T}{\sum_t w_t \times \text{GroupSize}\,(G_t)/\sum_t w_t}, \tag{15}$$

where t denotes time, T is the total duration, n represents node, G_t denotes the connected component graph at t, C is the centrality function, and w_t is the group weight at t. Closeness centrality is used for C, as suggested by Roy et al. [42], while group continuation time is used for w_t.

For a sanity check, we confirmed that a job of any workload with any task allocation strategy never fails on the faultload that exhibits no mobility.

6.1.4. Workloads

We evaluated *three* popular workloads in mobile cloud computing: Naïve Bayes Classifier, SIFT Feature Extraction, and Join, which belong to *three* representative application scenarios, namely Decision Making, Information Retrieval, and Sensor Preprocessing, respectively.

Table 4 lists workload statistics, such as data selectivity and processing throughput, which were obtained by running real map and reduce tasks on a Samsung Galaxy S8 (SM-G950N) device; this device is equipped with Samsung Exynos 8895, 4 GB RAM, and 64 GB storage and runs on Android 7.0 Nougat. Naïve Bayes is based on a Hadoop-based MapReduce implementation,[1] wherein a map task emits a count for each discretized attribute value, while a reduce task sums these counts. SIFT is implemented as a MapReduce algorithm based on the well-known OpenCV library,[2] wherein a map task extracts thousands of 144-dimensional features from an image, while a reduce task aggregates them. Furthermore, in the case of Join, a map task filters raw sensor values, while a reduce task calculates the Cartesian product of the sensor values on timestamp. As input data, Naïve Bayes and Join use real sensor data with 24 attributes collected from smartphone users [43], while SIFT uses 1440 $\times$ 1080 images.

As is clear from the data presented in Table 4, input data sizes differ based on workload and faultload, because the input size considered in our study is the one determined for the case when the baseline algorithm for a job takes 30 s on average to complete it. This 30 s time average is based on the responsiveness requirement of mobile cloud computing [37]. It should be noted that the smaller the input data size in Table 4, the more time-intensive the workload is. Then, the time-intensiveness of the three workloads is in the following order: Join $<$ Naïve Bayes $<$ SIFT. Please refer to Appendix C for the further analysis of the time-intensiveness in a mobile cloud.

1 https://github.com/theofilis/hadoop-Naïve-bayes.

2 https://github.com/opencv/opencv.

Table 5
Summary of parameters (the default value in bold).

Category	Parameter	Value
Genetic algorithm (Algorithm 1)	Survivor in SELECT	Tournament (Sample size 5)
	Offspring in SELECT	Roulette wheel
	Crossover in ALTER	Single point crossover (Rate 0.16)
	Mutation in ALTER	Stochastic mutator (Rate 0.115)
	MaxGenerations	1000 **2000** 3000 4000 5000
	SteadyGenerations	**5** 10 ··· 50
	PopulationSize	100 **200** 300 400 500
Faultload	Minimum Jaccard coefficient	0.4
Environment	Cluster utilization	0.5
	Maximum link bandwidth	5 MB/s

6.1.5. Parameters

Table 5 lists the three classes of parameters used in evaluation, namely Genetic Algorithm, Faultload, and Environment parameters.

- **Genetic Algorithm**: These experimentally determined parameters include the commonly used ones as well as those suggested in the previous studies [29,33]. Survivor and offspring selection in SELECT are tested with *Boltzmann*, *elite*, *roulette wheel*, *tournament*, and *truncation* algorithms; mutation and crossover operations in ALTER are tested with *stochastic*, *swap*, *uniform*, *single point*, and *multi point* algorithms. In particular, we randomly set input arguments for each candidate algorithm within their ranges for 100 trials. *MaxGenerations* was varied from 1000 to 5000 with increments of 1000; *SteadyGenerations* was varied from 5 to 50 with increments of 5; and *PopulationSize* was varied from 100 to 500 with increments of 100. We execute MTA on the three workloads and 100 sampled faultloads for every combination and determine the best parameters indicated in bold in Table 5.
- **Faultload**: The minimum Jaccard coefficient for generating faultload is decided as follows. As the minimum Jaccard coefficient increases, the similarity between the connected components in the faultload increases, and, in turn, the NCD, which is used to evaluate faultload mobility, starts decreasing. Based on this point in the NCD plot, the minimum Jaccard coefficient is roughly set to 0.4, which is the average of the NCD values.
- **Environment**: The cluster utilization is set to 0.5 for static algorithms; the influence of varying it is discussed in Section 6.2.2. Furthermore, based on a device-to-device communication study in an actual smartphone cloud [44], the maximum link bandwidth is set to 5 MB/s.

6.1.6. Fault tolerance

We evaluated the performance of the task allocation algorithms using the *two* fault tolerance conventions: *task migration* and *task dropping*. We follow the task migration convention by default except for the preliminary evaluation of the task dropping convention in Section 6.2.5.

- **Task Migration**: This is one of the universally-adopted fault tolerance conventions in both conventional and mobile clouds [10,21,28]. A failed task attempts to recover the input data from the previous tasks and even recursively traces back to the previous phases. However, if any data partition cannot be recovered even after the recursive traces, the task fails, and thus the job fails.
- **Task Dropping**: Since the inception of MapReduce, it has been a common practice to drop severely malfunctioning nodes or bad records on which tasks deterministically crash [17,45,46]. That is, this convention is more suitable for unreliable situations like mobile clouds than the *task migration* convention is. Recently, Pandey et al. [34] applied this convention to mobile cloud computing. Here, a user allows an approximate result, and a job can tolerate task failures up to the user-specified threshold at the expense of slight accuracy loss.

Furthermore, recent studies on mobile cloud consider data replication to increase data availability and job reliability [47,48]. However, we do not include data replication in our study, because data availability and job reliability introduced by data replication are not the primary focus of our task allocation approach.

6.1.7. Methodology

All our experiments are evaluated in the same manner as follows. We randomly sample 200 faultloads from each mobility data set and attempt to execute a job with the abovementioned fault tolerance convention until it is successfully completed in each faultload. If the job eventually fails, we mark the faultload as failed, and run the job on the next faultload. Because every algorithm involves randomness, we evaluate each algorithm 30 times and report the average result.

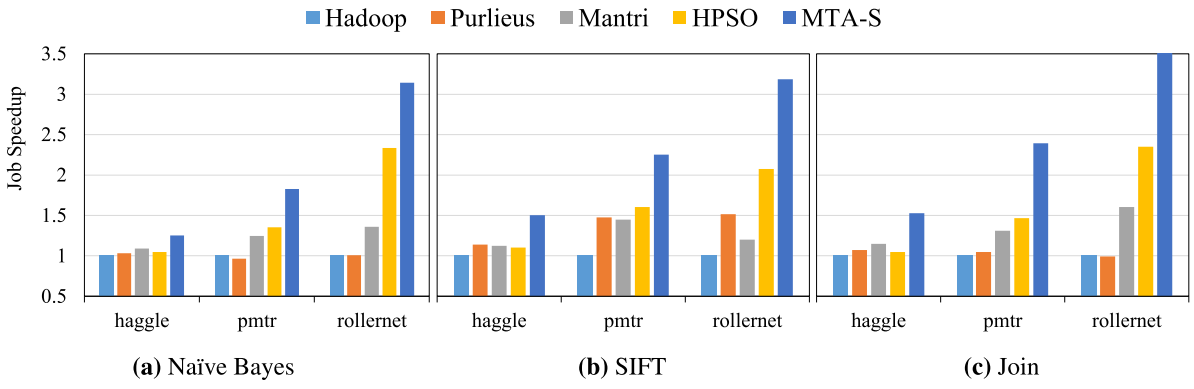


Fig. 3. Job speedup relative to Hadoop of three workloads on three faultloads.

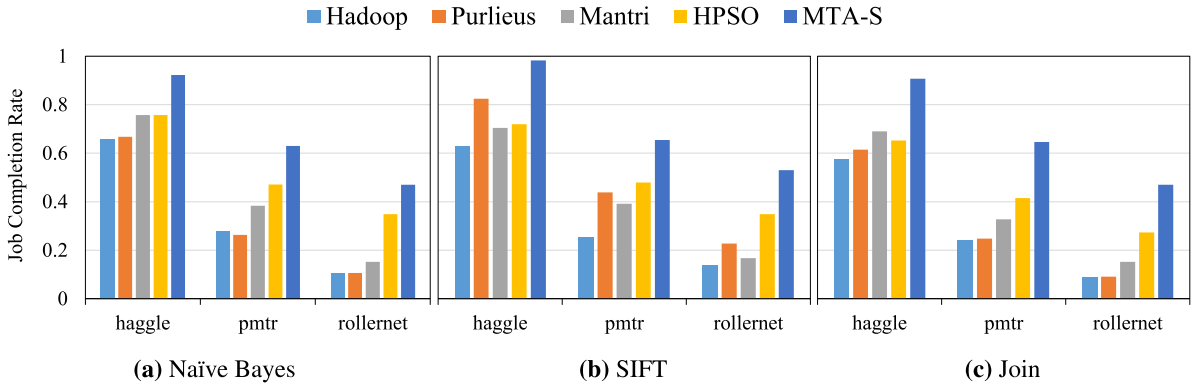


Fig. 4. Job completion rate of three workloads on three faultloads.

6.1.8. Metrics

We evaluated the performance of the task allocation algorithms using the following *four* metrics.

- **Job Speedup** is used to evaluate job speed over the time taken from the first task start to the last task finish. Because job speed could differ based on faultload even when the same algorithm and workload are considered, we measure job speedup of each algorithm relative to the baseline algorithm (Hadoop) when both algorithms have succeeded.
- **Job Completion Rate** is used to evaluate job reliability. In particular, we measure the job completion rate by calculating the ratio of the number of successful job trials to the total number of trials.
- **Task Speedup** is used to evaluate task speed over the task execution time averaged over all succeeded tasks. That is, the time for failed tasks is not included. Owing to the same reason as described above, we measure task speedup of each algorithm relative to the baseline algorithm.
- **Task Completion Rate** is used to evaluate task reliability. In particular, we measure the task completion rate by calculating the ratio of the number of successful task trials to the total number of trials.

6.2. Results

6.2.1. Job speed & reliability

Figs. 3 and 4 show the job speedup and job completion rate plots for the three workloads on the three faultloads, respectively. In terms of job speedup, MTA-S outperformed Hadoop by 1.2–3.7 times, Purlieus by 1.2–3.5 times, Mantri by 1.1–2.4 times, and HPSO by 1.2–1.7 times. Furthermore, in terms of job completion rate, MTA-S outperformed Hadoop by 26%–41%, Purlieus by 16%–40%, Mantri by 17%–36%, and HPSO by 12%–26%. Both performances consistently followed the order: **Hadoop < Purlieus < Mantri < HPSO < MTA-S**.

This performance trend can be explained based on the task allocation objectives of the different algorithms, as listed in Table 2. In particular, because Hadoop incorrectly models communication delay using hop-distance based data locality and does not model task reliability at all, it yielded the worst job speedup and worst job completion rate. Purlieus exhibited a similar job speedup and job completion rate to Hadoop for Naïve Bayes (map-input-heavy) and Join (map-and-reduce-input-heavy) workloads (Figs. 3(a), 3(c), 4(a), and 4(c)), because it is based on the same strategy as Hadoop in the

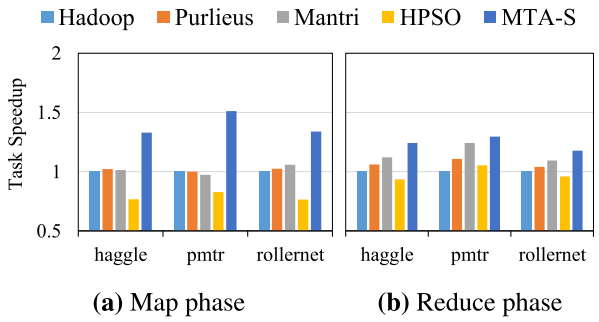


Fig. 5. Task speedup on three faultloads.

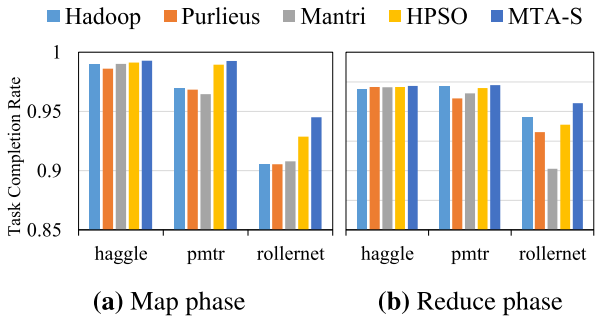


Fig. 6. Task completion rate on three faultloads.

specified cases. However, in the case of the SIFT (reduce-input-heavy) workload, Purlieus exhibited a slightly increased job speedup (Fig. 3(b)) because it allocated tasks to closely connected nodes to reduce off-rack traffic and slightly increased job completion rate (Fig. 4(b)) owing to the unintentional increase in reliability because of the close allocations that were established. This approach is not effective because, unlike in a conventional cloud, there is no clear distinction of a rack in a mobile cloud due to the relatively small cluster size.

Figs. 5 and 6 show the task speedup and task completion rate plots, respectively, with phase distinction, thus enabling a deeper analysis of Mantri, HPSO, and MTA-S, in that they consider task speed and task reliability unlike Hadoop and Purlieus. The results shown are those that are averaged over all workloads. Mantri behaves in the same manner as Hadoop for map allocation, which explains their similar task speedup (Fig. 5(a)) and task completion rate (Fig. 6(a)). In contrast, for reduce allocation, Mantri, which aims at maximizing task speed, exhibited an increased task speedup (Fig. 5(b)). Nevertheless, MTA-S was still faster than Mantri, because the link-based delay estimation of MTA-S has finer modeling granularity for communication bandwidth than the node-based delay estimation used in Mantri, and thus MTA-S exhibits better efficiency. Furthermore, in the case of Mantri, task completion rate slightly decreased (Fig. 6(b)) in favor of the biased improvement in task speed. Consequently, Mantri showed a slight performance improvement in both job speed and job completion rate (Figs. 3 and 4). Next, HPSO, which aims to maximize task reliability, exhibited a significantly increased task completion rate (Fig. 6) and thus ranked next to best in both job speed and job completion rate (Figs. 3 and 4). Nevertheless, MTA-S still exhibited better task reliability than HPSO (Fig. 6), because, unlike MTA-S, HPSO does not optimize task speed, a factor of task reliability; as a result, HPSO achieved the worst task speedup (Fig. 5). Lastly, MTA-S outperformed other algorithms in both job speed and job completion rate (Figs. 3 and 4) by maximizing both task speed and task reliability (Figs. 5 and 6). It is important to note that the obtained result is consistent with Theorem 1.

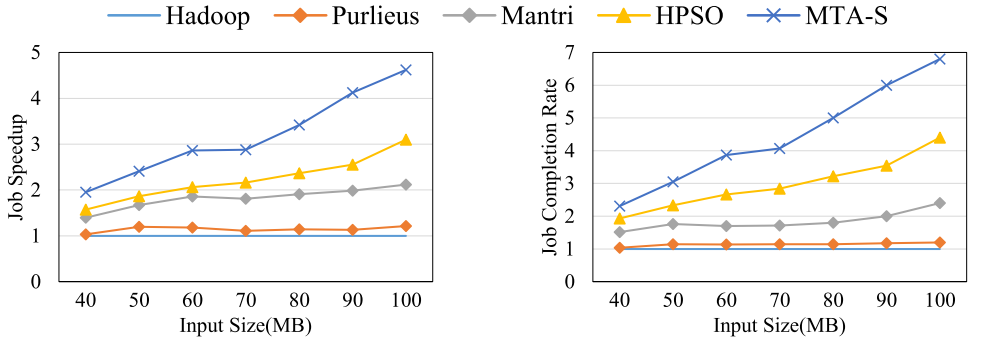
6.2.2. Resource efficiency

The resource efficiency of MTA can be identified from Table 6 by comparing MTA-D with MTA-S in terms of job speedup, job completion rate, and search cost on the three faultloads. Here, we only present the result for the Join workload owing to similar performance trends in the case of other workloads. Similar to the approach described in Section 6.1.4, the input data size is determined to be the one when MTA-S takes 30 s for the given cluster utilization of either 0.5 or 0.9.

In all cases, the cluster utilization of MTA-D consistently converged to around 0.67 achieving considerably better job speedup and job completion rate than MTA-S. In particular, when the cluster utilization for MTA-S was 0.5, the job speedup showed a greater increase than the job completion rate for MTA-D, because of relatively lesser resources leading to slow processing and communication. In contrast, when the cluster utilization for MTA-S was 0.9, the job completion rate

Table 6
Comparison of MTA-D with MTA-S.

Faultload	Cluster utilization		Job speedup	Job completion rate	Allocation time (s)	
	MTA-S	MTA-D	relative to MTA-S	relative to MTA-S	MTA-S	MTA-D
haggle	0.5	0.66	**1.18**	1.01	2.4	9.8
	0.9	0.68	1.06	**1.16**	2.1	9.7
pmtr	0.5	0.67	**1.15**	1.08	2.0	7.4
	0.9	0.67	1.06	**1.18**	1.9	6.7
rollernet	0.5	0.68	**1.42**	1.34	1.2	4.5
	0.9	0.69	1.21	**1.35**	1.0	3.7



(a) Job speedup relative to Hadoop **(b)** Job completion rate relative to Hadoop

Fig. 7. Scalability to input data size.

improved more significantly than the job speedup for MTA-D, because node selection could not exclude low reliability nodes, which resulted in low job reliability. It is interesting to note that the latter implies both job speedup and job completion rate can be improved even with few resources, which is counterintuitive to the conventional notion of cluster utilization.

The allocation time taken by both algorithms, MTA-S and MTA-D, are listed in the last two columns of Table 6; as previously mentioned, the algorithms were executed on a Samsung Galaxy S8 (SM-G950N) device having eight multi-threads. Based on these results, it can be considered that MTA-S is suitable for real-time operation, whereas MTA-D is not, because it requires up to about 10 s for task allocation in certain cases as it takes $log_2 N$ time more than MTA-S to evaluate costs given that it uses the binary search. To address this issue, MTA-D could be performed asynchronously; this requires further research, and till such time, the converged value of 0.67 for cluster utilization can be used as a rule of thumb.

6.2.3. Scalability

Fig. 7 shows the scalability of the algorithms in terms of job speedup and job completion rate relative to baseline Hadoop as the input data size varies from 40 MB to 100 MB. The results shown are those that are averaged over all workloads and faultloads. In all cases, MTA-S outperformed the other algorithms with near-linear scalability. This is because increased communication traffic caused by increased input data makes optimizations of communication delay and task reliability in MTA-S a lot more valuable. The overall performance consistently follows the following order: Hadoop $<$ Purlieus $<$ Mantri $<$ HPSO $<$ MTA-S, which is explained earlier in 6.2.1. Overall, in future, the superiority of MTA over current state-of-the-art algorithms will be more prominent, as the amount of mobile data is increasing at a rapid annual rate of roughly 1.5 times [2], owing to its high scalability.

6.2.4. Comparison of MTA with the task offloading approach

We compared MTA (the *mobile cloud* approach) with the *task offloading* approach that is based on a remote cloud.

Fig. 8 shows the job completion time of MTA and the task offloading approach averaged over all workloads. The results for the task offloading are *estimated* based on processing powers of different Amazon EC2 servers and cellular communication speeds at various cities around the world. The result of MTA is constant regardless of an EC2 server and a city because it uses neither remote servers nor cellular communication. MTA finished significantly faster than the task offloading approach on a c5.large server and achieved comparable performance on a c5.4xlarge server and a c5.18xlarge server which is the most compute-intensive instance of EC2. Therefore, in many situations, the mobile cloud is preferred to the task offloading approach in terms of job completion time. Please refer to Appendix D for the comparison details.

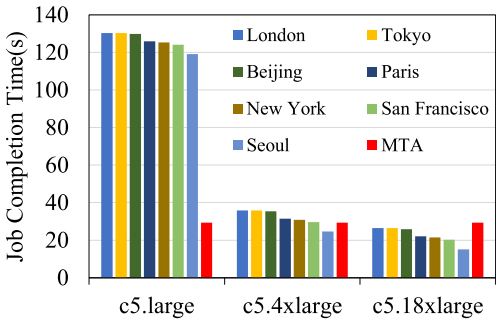


Fig. 8. Job completion time of MTA and the task offloading approach.

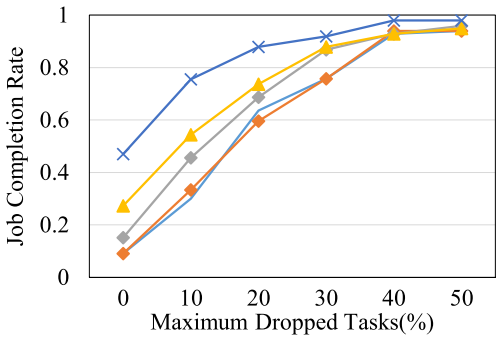


Fig. 9. Job completion rate of Join on rollernet for different % of maximum dropped tasks.

6.2.5. MTA with task dropping-based fault tolerance

We conducted a preliminary evaluation of the task allocation algorithms with *task dropping*-based fault tolerance convention that is configured to work with the default *task migration* convention. Fig. 9 shows the job completion rate as the percentage of maximum dropped tasks varies from 0% to 50%. Here, owing to similar performance trends, we only present the result for Join on rollernet, which achieves the lowest performance improvement in Fig. 4(c). MTA showed around 90% job completion rate by dropping up to 20% task failures and about 1.6 times improvement by dropping up to only 10% task failures. Such a low level of task failures is tolerable even in terms of accuracy because the accuracy loss caused by task dropping is less than only 10% for many similar workloads [46].

7. Conclusion

In this study, we proposed a novel task allocation strategy, referred to as *MTA*, to address the task allocation problem, which is one of the most urgent challenges in the field of mobile cloud computing. The proposed MTA concurrently maximizes job speed and reliability by modeling communication delay and task reliability for mobile MapReduce. We compared MTA with four state-of-the-art strategies on three representative workloads and hundreds of real mobility traces using a mobile MapReduce simulator validated against a platform running on an actual smartphone cluster. Based on our obtained results, it is evident that MTA significantly outperformed the state-of-the-art task allocation algorithms by up to 3.7 times and 41%, respectively, in terms of job speed and reliability; in addition, it was resource-efficient as well as scalable to different input data sizes. In conclusion, we believe that our study significantly enhances the usability of the task allocation strategy in mobile cloud computing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Jin-woo Lee: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Gwangseon Jang:** Conceptualization, Data curation, Investigation. **Hohyun Jung:** Formal analysis, Investigation. **Jae-Gil Lee:** Conceptualization, Formal

analysis, Funding acquisition, Methodology, Resources, Supervision, Writing - original draft, Writing - review & editing. **Uichin Lee:** Conceptualization, Writing - original draft.

Acknowledgments

This research was supported by the MOLIT (The Ministry of Land, Infrastructure and Transport), Korea, under the national spatial information research program supervised by the KAIA (Korea Agency for Infrastructure Technology Advancement) (19NSIP-B081011-06).

Appendix A. Mobile cloud computing applications

In this section, we introduce various real applications and use cases in mobile cloud computing which can benefit from MTA, the proposed task allocation strategy. These applications and use cases are classified into those for decision making, information retrieval, and sensor preprocessing.

A.1. Decision making

- **Collaborative Inference**: This scenario is primarily aimed at becoming better aware of surrounding context by facilitating collaboration of multiple inference models for multiple mobile devices in a mobile cloud [49]. For example, DarwinPhone [12] confirmed that a speaker can be recognized with higher accuracy and confidence when multiple devices share their models. In addition, it presents other use cases, such as discovering locations using RFID or tagging people in pictures.
- **Augmented Reality**: Augmented reality applications based on real-time object recognition can also benefit from our task allocation strategy by improving the response time. Locating injured persons in smoke environments, snipers in urban warfare settings, or obstacles in disaster situations are examples of this scenario [10,11].

A.2. Information retrieval

- **Multimedia Retrieval**: Hyrax [14] demonstrated the usability of searching multimedia files in a mobile cloud constructed in a real stadium. As another example, missing children can be found with the help of a mobile cloud as well [10], by transmitting the picture of the child to the nearby smartphones, comparing the picture with other kids in the vicinity of each participant, and reporting the results back to the requester.
- **Collaborative Photography**: LetsPic [50] proposed a new method for photography that enhances group activities by displaying group photography information overlaid on the camera screen. In particular, group members can easily see the pictures being captured by other members of the group by automatically comparing their viewfinder screen with all images captured and distributed over participating smartphones as a user moves the viewfinder in real-time.

A.3. Sensor preprocessing

- **Crowd Sensing**: Skyship [13] flies a blimp as an infrastructure coordinator to create an ad-hoc 5G network in disaster situations, dispatches drones as intermediate nodes, and processes a large collection of images quickly using these drones. In addition, we can use a mobile cloud for joining or sorting raw data that is distributed over multiple devices based on timestamp [14], or filtering unnecessary data while classifying trail conditions for group hikers [51].

Appendix B. More realistic task allocation problem in a mobile cloud

The mobile task allocation problem can be extended by considering additional issues in mobile cloud computing or different assumptions in contrast to the prescribed ones in Section 4. We suggest possible extensions of the task allocation problem in Appendix B.1 and show the effect of a different assumption in Appendix B.2.

B.1. Task allocation with additional objectives

Fig. B.10 shows the priority of primary topics in mobile cloud computing in terms of the percentage of research studies in which they were addressed out of 56 major researches. Above all, resolving the task allocation problem, which involves determining the nodes on which tasks are executed, is the most prevalent primary topic, which is directly or indirectly covered in 92% of the mobile cloud computing studies. To the best of our knowledge, this is the first topic-based priority statistic, which was obtained by clustering and ordering all the primary topics considered in four renowned surveys [10,52–54], five major platform researches [9,14,23,24,34], and 47 other papers related to mobile cloud that include citations for one of these nine, aforementioned paper.

Among these primary topics, energy-awareness and incentive issues can be incorporated into the task allocation problem as follows. (Fault tolerance and scalability are already discussed in Section 6; heterogeneity will be addressed in Appendix B.2.)

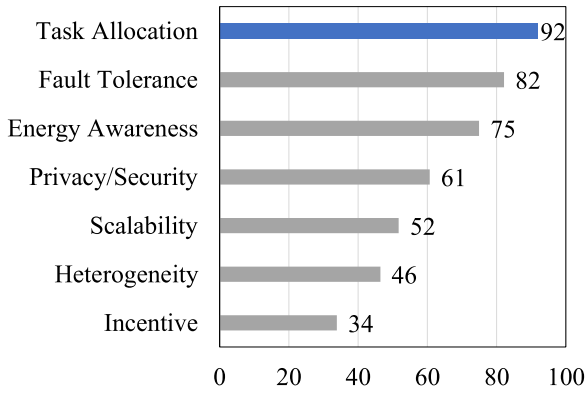


Fig. B.10. Priority of primary topics in mobile cloud computing.

Table B.7
Processing throughput (MB/s) statistics of three workloads on five mobile devices.

Type	Naïve Bayes		SIFT		Join	
	Map	Reduce	Map	Reduce	Map	Reduce
Google Nexus 7	0.07	0.07	0.02	1.02	0.49	0.24
LG Nexus 5	0.13	0.13	0.04	1.79	0.85	0.43
LG G3	0.20	0.20	0.07	2.77	1.32	0.66
Samsung Galaxy S6	0.23	0.23	0.08	3.23	1.54	0.77
Samsung Galaxy S8	0.30	0.30	0.10	4.20	2.00	1.00

- **Constraint 1 (Resource)**: There are M types of resources, such as energy, computation, storage, and even security level; each node to which the ith task is allocated restricts its type-m resource $\rho_{i,m}$ within the budget of type-m resource $R_{i,m}$ where $m \in \{1, \ldots, M\}$.
- **Constraint 2 (Incentive)**: According to the pay-per-use monetary incentive [10], each node to which the ith task is allocated participates in a job if the utility u_i, or incentive, is greater than the cost c_i, where the total cost is paid by the clients who benefit from the job result.

Then, these two additional constraints extends the task allocation problem from Eq. (1) to Eq. (B.1).

$$\begin{aligned} \mathbb{TA}_P{}^* = \underset{\mathbb{TA}_P \in \mathcal{TA}_P}{\operatorname{argmin}} \; Cost\,(\mathbb{TA}_P) \quad \text{s.t.} \quad & 0 < ClusterUtilization \leq 1 \\ & |\mathbb{TA}_P| = |\mathbb{N}| \times ClusterUtilization \\ & \rho_{i,m} \leq R_{i,m}, \quad i \in \{1, \ldots, |\mathbb{TA}_P|\}, \quad m \in \{1, \ldots, M\} \\ & c_i < u_i, \quad i \in \{1, \ldots, |\mathbb{TA}_P|\} \end{aligned} \tag{B.1}$$

B.2. Task allocation with a different assumption

By relaxing the homogeneity assumption, we evaluated MTA with a heterogeneous device setting as well. Table B.7 lists the processing throughput statistics of the three workloads, which were obtained by running real map and reduce tasks on *five* different mobile devices—Google Nexus 7, LG Nexus 5, LG G3, Samsung Galaxy S6, and Samsung Galaxy S8. For all evaluation instances, a node is assigned a device type uniformly at random among the five devices. All the other parameters hold the same.

Figs. C.11 and C.12 show the job speedup and the job completion rate for the three workloads on the three faultloads, respectively, with the heterogeneous device setting. It is worthwhile to note that the performances consistently follow the same order obtained with a homogeneous device setting: **Hadoop < Purlieus < Mantri < HPSO < MTA-S**.

Appendix C. The time-intensiveness and time-sensitiveness of workload in a mobile cloud

In this section, we discuss both the time-intensiveness and time-sensitiveness of a workload from the perspective of mobile cloud computing. First, a time-intensive workload can be more negatively impacted than a less time-intensive workload in a mobile cloud. As explained in Section 6.1.4, the time-intensiveness of the three workloads is in the following order: Join < Naïve Bayes < SIFT. Table C.8 tabulates the job speedup of Fig. 3 in detail and, as shown in the last column of Table C.8, a time-intensive workload tends to obtain lower speedup gain than a less time-intensive workload in a mobile

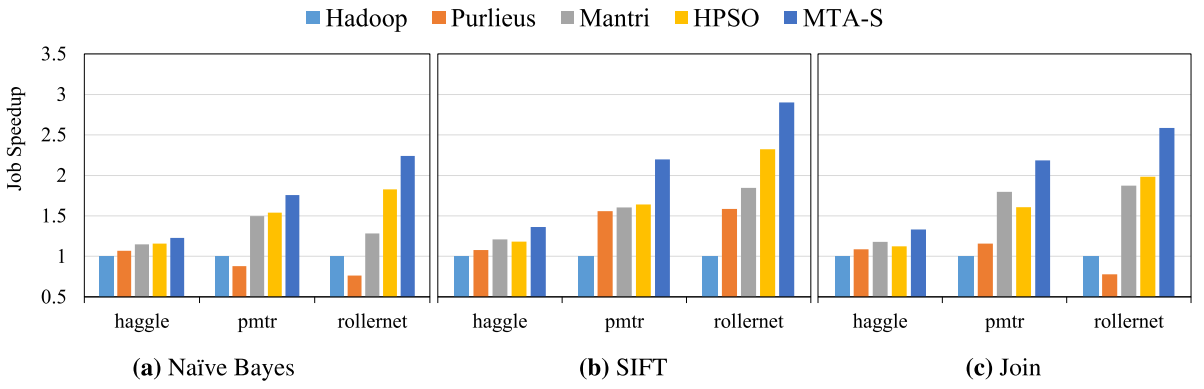


Fig. C.11. Job speedup relative to Hadoop of three workloads on three faultloads with heterogeneity.

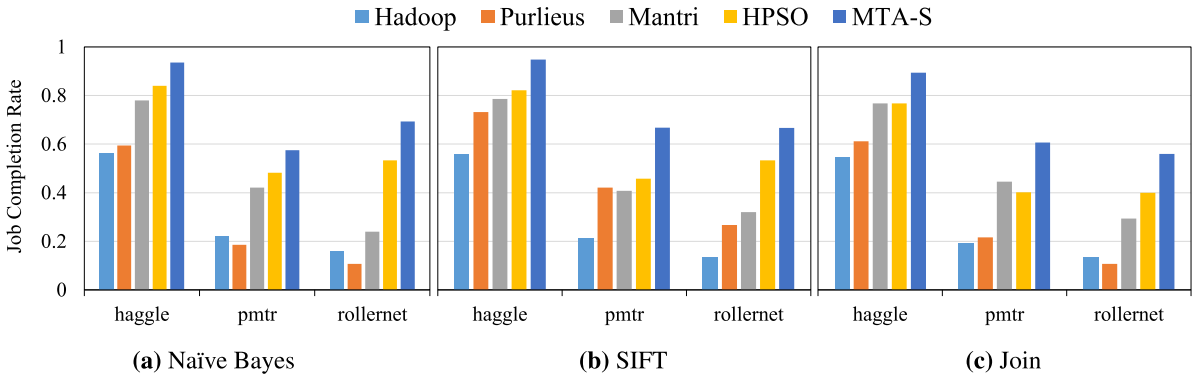


Fig. C.12. Job completion rate of three workloads on three faultloads with heterogeneity.

Table C.8
Job speedup relative to Hadoop of three workloads on three faultloads (a tabulation of Fig. 3).

Workload	Faultload	Purlieus	Mantri	HPSO	MTA	Average
Naïve Bayes	haggle	1.03	1.07	1.03	1.20	1.08
	pmtr	0.97	1.21	1.28	1.69	1.29
	rollernet	1.01	1.33	2.09	2.76	1.80
SIFT	haggle	1.00	1.01	0.98	1.32	1.08
	pmtr	1.30	1.30	1.41	1.96	1.49
	rollernet	1.36	1.08	1.81	2.74	1.75
Join	haggle	1.07	1.12	1.03	1.45	1.17
	pmtr	1.05	1.28	1.39	2.18	1.47
	rollernet	0.99	1.57	2.16	3.66	2.10

cloud. For example, the job speedup ranges from 1.17 to 2.10 for Join, whereas it only ranges from 1.08 to 1.75 for SIFT. Furthermore, it should be noted that MTA always performs the best regardless of the time-intensiveness.

Next, a time-sensitive workload in a mobile cloud can benefit from effective task allocation [11,55]. Even though the performance improvement is only in units of milliseconds, this is significant from the perspective of a time-sensitive workload. However, based on the current MapReduce implementation, the initialization overhead itself far exceeds the potential improvement, because MapReduce is basically a *batch* query processing system and has difficulties in running *real-time* queries [21]. Therefore, we currently aim at supporting batch workloads in a mobile cloud, which take from 5 to 50 s based on the responsiveness requirement [37], and will extend our approach to time-sensitive workloads as well.

Appendix D. Methodology for the comparison of MTA with the task offloading approach

In this section, we describe the comparison of MTA (the mobile cloud approach) with the task offloading approach in detail, which is presented in Section 6.2.4. Since task offloading consists of cellular communication and remote processing, its job completion time is the sum of the time for cellular communication and remote processing, which is estimated as below.

Table D.9
Cellular download and upload speeds (MB/s) at various cities around the world.

Country	City	Download				Upload			
		Carrier 1	C2	C3	Average	C1	C2	C3	Average
U.K.	London	22.2	18.2	12.6	17.63	0.9	0.7	0.5	0.67
Japan	Tokyo	29.3	17.3	14.0	20.22	0.5	0.8	0.6	0.64
China	Beijing	14.7	13.5	16.4	14.88	0.4	0.9	0.9	0.75
France	Paris	29.9	23.7	21.3	24.94	1.0	0.7	0.8	0.84
U.S.	New York	33.4	19.9	17.8	23.68	1.5	0.3	0.9	0.89
U.S.	San Francisco	25.8	20.0	17.5	21.10	1.4	1.0	0.7	1.05
Korea	Seoul	49.0	34.4	33.3	38.88	1.6	1.9	1.2	1.55

Table D.10
Processing powers of mobile devices and remote servers in terms of multi-core floating point scores and performance gain ratios.

Type	Multi-core floating point Score						Performance
	Instance 1	I2	I3	I4	I5	Average	Gain ratio
Google Nexus 7	1719	783	1463	1446	1478		
LG Nexus 5	1167	1190	1240	1727	1767		
LG G3	1174	1145	1606	2474	1838	2788	1
Samsung Galaxy S6	3645	3614	3745	3474	3526		
Samsung Galaxy S8	5775	6462	6185	5883	5173		
Amazon c5.large	4349	5587	4761	5661	5651	5202	**1.9**
Amazon c5.4xlarge	33992	35134	39150	38778	34616	36334	**13.0**
Amazon c5.18xlarge	51747	106598	95782	95062	101803	90198	**32.4**

First, for cellular communication, Table D.9 shows the cellular download and upload speeds at various cities around the world, which are provided by OpenSignal[3] as of May 1, 2019. The carrier names are hidden to avoid unnecessary dispute, and the average of the three carriers is used for the estimation. Cellular communication, in turn, consists of offload and retrieve communications. The offload time can be expressed as (input size / (number of nodes $\times$ upload speed)) because input data partitions are distributed across multiple nodes, and the retrieve time can be expressed as (output size/download speed) because it is generally assumed that a single job client who initiated the job retrieves the result.

Next, for remote processing, Table D.10 shows the processing powers of Amazon c5.large, c5.4xlarge, and c5.18xlarge EC2 instances, in addition to the *five* mobile devices used in Appendix B.2. In Table D.10, a score indicates the floating point performance measured by Geekbench[4] as of May 1, 2019. The average of five instances or devices is used for the estimation. Then, we define the ratio of the average score of a remote server to the average score of all mobile devices as the *performance gain ratio*. Considering the performance gap between a remote server and a mobile device, the remote processing time is estimated by dividing the map and reduce time already measured on a mobile device by the performance gain ratio.

Appendix E. Simulator validation

For comprehensive evaluation of mobile MapReduce, we have considered various options to construct a *realistic* evaluation setting. First, we started this study with a prototype laptop testbed to conduct a preliminary evaluation on task allocation. We were able to quickly confirm the feasibility of our study because Hadoop did not need to be ported to Ubuntu, which was the operating system used for the laptop testbed; however, because the performance of a laptop is different from that of a mobile device, this setting was not perfectly realistic. Next, we adopted a smartphone testbed, which enabled us to conduct evaluations on a small mobile cloud; however, increasing the scale of the smartphone testbed was difficult in practice because it entails the cost of having dozens of smartphones or hiring dozens of people. As a result, we designed a simulator for evaluating a *large* mobile cloud as shown in Appendix E.1 and validated it using the smartphone testbed as shown in Appendix E.2.

E.1. Design of the simulator and testbed platform

Fig. E.13 shows an abstract architecture for mobile MapReduce with a class diagram that is used for designing the simulator and platform. Hereafter, the mobile MapReduce software built on the testbed is shortly referred to as the *platform*. It is important to note that we did not merely compile Hadoop on Android, but developed the simulator and

[3] https://www.opensignal.com.
[4] https://www.geekbench.com.

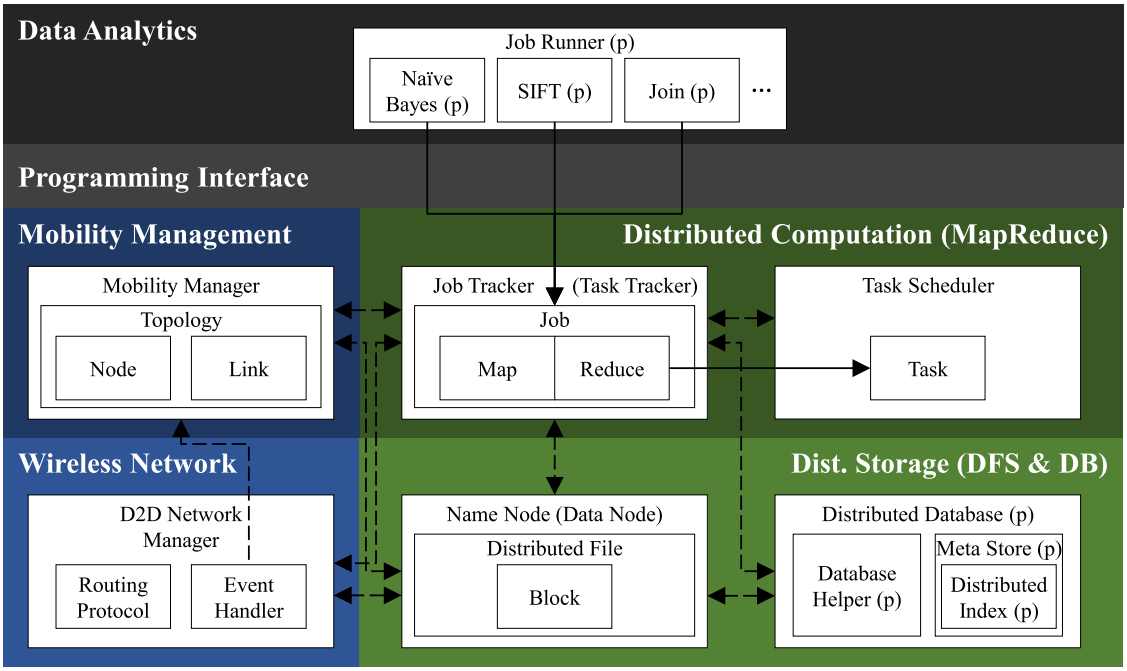


Fig. E.13. An abstract architecture for mobile MapReduce with a class diagram. A white box represents a class, a dotted line represents a dependency, and a solid line represents a generalization. The predicate letter 'p' in a class name is used for a class of which the implementation only exists for the platform.

platform by porting the design principles of core Hadoop components presented in the architecture to Android. Now, we explain the ported design principles and implementation details of each building block and each class in the architecture, which apply to both the simulator and platform unless otherwise stated.

- **Wireless Network**: The *D2D Network Manager* aims at forming a D2D group, sending or receiving data between nodes, and handling a node arrival or departure event. To achieve these features, the *Wireless Network* layer supports the IP address assignment and routing protocol. Above all, it is assumed that a unique IP address is assigned to each device, and there is no single DHCP server that generates IP addresses in a mobile cloud. To satisfy the assumptions, we assign a predetermined static IP address to each node, which is sufficient for validation. An approach like dynamic configuration of IP addresses[5] could be adopted as well. Next, for route maintenance and packet forwarding, we adopt AODV (Ad-hoc On-demand Distance Vector) for a routing protocol, which is a well-known reactive protocol that discovers routes only when it is needed. AODV implementations in a J-Sim network simulation library[6] and a well-known open source project on Android[7] are used for the simulator and platform, respectively.
- **Mobility Management**: The *D2D Network Manager* notifies the *Mobility Manager* of a node arrival or departure event, and the *Mobility Manager*, in turn, notifies other class instances of the event for subsequent operations. When a node joins the group, the node initializes itself by starting core class instances, such as a *Data Node* and a *Task Tracker*. The *Mobility Manager* of other nodes notifies the *Name Node* and the *Job Tracker*, which are hosted in the master node, of the node arrival for balancing resources. On the contrary, when a node leaves the group, the *Mobility Manager* of other nodes notifies the *Name Node* for block recovery and the *Job Tracker* for fault tolerance. In particular, if the master node leaves the cloud, the next node that participated in the group becomes a new master node; a new *Name Node* and *Job Tracker* are started on the new master node; the *Name Node* recovers a distributed file system by receiving block reports from all *Data Nodes*; the jobs from the old *Job Tracker* are restarted. To avoid an unnecessary master node departure during validation, a node that does not fail within a faultload is intentionally assigned the role of a master node.
- **Distributed Storage**: The distributed storage consists of a distributed file system (DFS) and a distributed database built on top of the DFS. First, the DFS follows the same design principles as Hadoop distributed file system (HDFS) in terms of block management between a *Name Node* and a *Data Node* except for block replication. The block replication

5 https://tools.ietf.org/html/rfc3927.
6 https://sites.google.com/site/jsimofficial.
7 https://code.google.com/archive/p/adhoc-on-android.

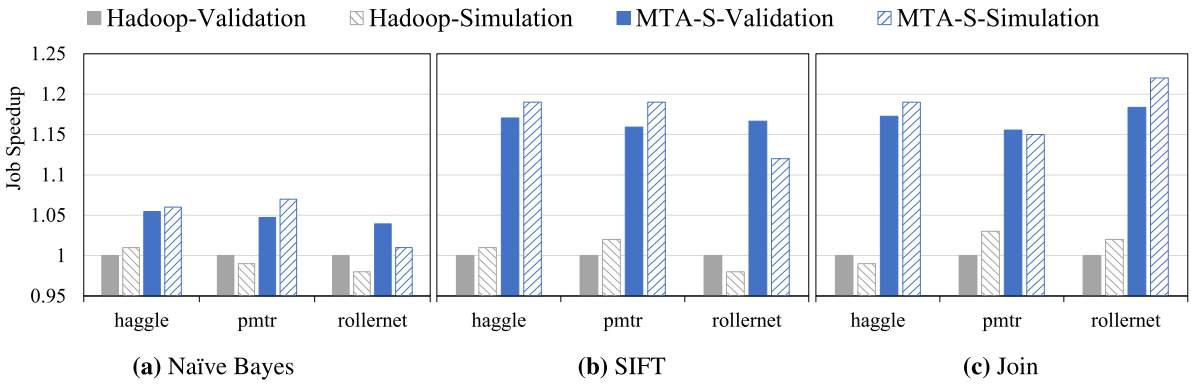


Fig. E.14. Job speedup relative to Hadoop of three workloads on three faultloads.

of the HDFS based on rack awareness is inappropriate in a mobile cloud where rack hierarchy does not even exist. Recent studies on mobile clouds consider data replication to increase data availability and job reliability [47,48], whereas we only provide a random replication because we do not consider data replication in our study.

Next, we additionally developed the *Distributed Database* within the platform to facilitate information retrieval in a mobile cloud, which is one of the representative application scenarios as presented in Appendix A.2. The *Database Helper* supports basic operations like select, insert, delete, and nearest neighbor search of high-dimensional vectors by using a KD-tree-based *Distributed Index*. This can be useful for a multimedia retrieval application based on a feature extraction algorithm like SIFT that extracts thousands of 144-dimensional features from an image.

- **Distributed Computation**: The *Job Tracker* and *Task Tracker* follow the same design principles as Hadoop MapReduce except for memory management on the platform. Hadoop reserves 100 MB memory by default for buffering and sorting map outputs, whereas the heap of an Android application cannot grow to 100 MB. We resolved this issue by adjusting `io.sort.mb` and `io.sort.record.percent` configurations of MapReduce according to the memory availability of each device and enabling the Android `largeHeap` configuration. Furthermore, the *Task Scheduler* also follows the same design principles as Hadoop.
- **Programming Interface & Data Analytics**: The *Programming Interface* layer is required to seamlessly integrate the MapReduce algorithms in the *Data Analytics* layer, such as Naïve Bayes, SIFT, and Join, with the aforementioned lower layer components. For the simulator, the integration is simple because, as described in Section 6.1.4, the simulator does not require an actual implementation of a MapReduce algorithm, but the workload statistics of an algorithm. In contrast, for the platform, a MapReduce job runs on an Android background `Service`, whereas the job is submitted to the `Service` and interacts with it on an Android foreground `Activity`. For the seamless integration between the background and foreground processes, we provide a number of callback functions, e.g., `onMapperFinished()`, `onIterationFinished()`, and `onJobFinished()`, which are invoked when the corresponding event in a MapReduce flow occurs.

For efficiency of development, we also followed the guidelines suggested by Hyrax [14], which involves optimizing MapReduce configurations, substituting processes for threads of a single process, and integrating job classes into the platform to resolve the classpath issue caused by dynamically deploying a job.

E.2. Validation methodology and results

To validate the simulator introduced in Section 6.1.1, we devised an actual smartphone testbed that supports a multi-hop network by placing intermediate wireless routers to bridge the smartphones in the mobile ad-hoc network. To recreate the same effect as that of using intermediate routers during simulation, we disable the processing ability of the router nodes. For network parameters, we use the AODV routing protocol and constrain the maximum link bandwidth to 5 MB/s. Finally, we set up a mobile cloud testbed with five Samsung Galaxy S8 (SM-G950N) devices running Android 7.0 Nougat and five ipTime A304 routers, which leads to a faultload group size of less than or equals to 10.

To address the difficulty of moving devices in reality for validation, we instead aim to reproduce pseudo faultload mobility; in particular, we place smartphones and routers as in the initial layout of the faultload and reproduce node departure by forcing the node out of the cluster using an Android application script instead of physically moving the node. Furthermore, for validation simplicity, we ignore node join events.

Figs. E.14 and E.15 show the job speedup and job completion rate of the three workloads on the three faultloads. In particular, we selected Hadoop and MTA-S as the representative heuristics of conventional and mobile clouds, respectively. In terms of job speed, the validation result seems to be highly correlated with the simulation result; the results are only slightly different because of the unavoidable incompleteness in a simulation. Furthermore, in terms of job completion

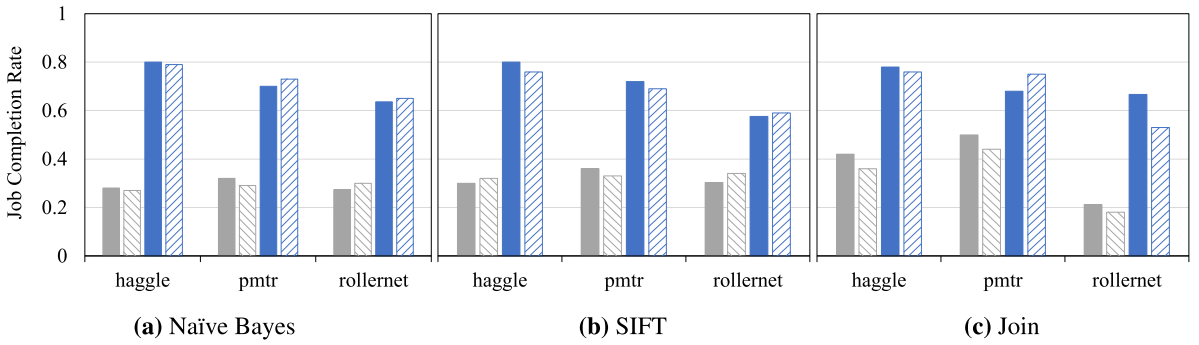


Fig. E.15. Job completion rate of three workloads on three faultloads.

rate, the validation result almost perfectly matched the simulation result because failures occur on the same nodes at any rate in both validation and simulation scenarios; however, a slight difference in these results can be attributed to the failure rate noise caused by the small contention difference.

References

[1] E. Zolfagharifard, How the Apple Watch is as powerful as TWO Cray supercomputers, 2015, http://dailym.ai/1PLwFWw.
[2] V. Cisco Mobile, Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021, 2017.
[3] Q. Han, S. Liang, H. Zhang, Mobile cloud sensing, big data, and 5G networks make an intelligent and smart world, IEEE Netw. 29 (2) (2015) 40–45.
[4] Google, Tensorflow lite, 2017, https://www.tensorflow.org/mobile/tflite. (Accessed 01 May 2019).
[5] Apple, Core ML, 2017, https://developer.apple.com/documentation/coreml. (Accessed 01 May 2019).
[6] Facebook, CaFfe2Go, 2017, https://code.facebook.com/posts/196146247499076. (Accessed 01 May 2019).
[7] X. Zeng, K. Cao, M. Zhang, MobileDeepPill: A small-footprint mobile deep learning system for recognizing unconstrained pill images, in: Proc. 15th ACM Int'L Conf. on Mobile Systems, Applications, & Services, 2017, pp. 56–67.
[8] L.N. Huynh, Y. Lee, R.K. Balan, DeepMon: mobile GPU-based deep learning framework for continuous vision applications, in: Proc. 15th ACM Int'L Conf. on Mobile Systems, Applications, &Services, 2017, pp. 82–95.
[9] Q. Ning, C.-A. Chen, R. Stoleru, C. Chen, Mobile storm: Distributed real-time stream processing for mobile clouds, in: Proc. IEEE 4th Int'L Conf. on Cloud Networking, 2015, pp. 139–145.
[10] N. Fernando, S.W. Loke, W. Rahayu, Mobile cloud computing: A survey, Future Gener. Comput. Syst. 29 (1) (2013) 84–106.
[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, MAUI: Making smartphones last longer with code offload, in: Proc. 8th ACM Int'L Conf. on Mobile Systems, Applications, & Services, 2010, pp. 49–62.
[12] E. Miluzzo, C.T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, A.T. Campbell, Darwin phones: The evolution of sensing and inference on mobile phones, in: Proc. ACM Int'L Conf. on Mobile Systems, Applications, & Services, 2010, pp. 5–20.
[13] KT, KT unveils 5G emergency rescue platform 'SKYSHIP', 2018, https://prn.to/2OTniKf. (Accessed 01 May 2019).
[14] E.E. Marinelli, Hyrax: Cloud Computing on Mobile Devices using MapReduce, Tech. Rep., Carnegie-Mellon Univ., 2009.
[15] L.M. Silva, R. Buyya, Parallel programming models and paradigms, High Perform. Cluster Comput. 2 (1999) 4–27.
[16] Y. Chen, S. Alspaugh, R. Katz, Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads, Proc. VLDB Endow. 5 (12) (2012) 1802–1813.
[17] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008).
[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, in: Proc. 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems, 2007, pp. 59–72.
[19] J. Shi, Y. Qiu, U.F. Minhas, L. Jiao, C. Wang, B. Reinwald, F. Özcan, Clash of the titans: MapReduce vs. Spark for large scale data analytics, Proc. VLDB Endow. 8 (13) (2015) 2110–2121.
[20] C.S.R. Murthy, B. Manoj, Ad hoc Wireless Networks: Architectures and Protocols, Pearson Education, 2004.
[21] T. White, Hadoop: The Definitive Guide, fourth ed., O'Reilly Media, 2015.
[22] S.-T. Sheu, J. Chen, A novel delay-oriented shortest path routing protocol for mobile ad hoc networks, in: Proc. IEEE Int'L Conf. on Communications, 2001, pp. 1930–1934.
[23] G. Huerta-Canepa, D. Lee, A virtual cloud computing provider for mobile devices, in: Proc. 1st ACM Workshop on Mobile Cloud Computing & Services, 2010, p. 6.
[24] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, V.H. Tuulos, Misco: A MapReduce framework for mobile systems, in: Proc. 3rd ACM Int'L Conf. on Pervasive Technologies Related to Assistive Environments, 2010, p. 32.
[25] M. Hammoud, M.F. Sakr, Locality-aware reduce task scheduling for MapReduce, in: Proc. IEEE 3rd Int'L Conf. on Cloud Computing Technology & Science, 2011, pp. 570–576.
[26] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: Locality-aware resource allocation for MapReduce in a cloud, in: Proc. ACM Int'L Conf. for High Performance Computing, Networking, Storage & Analysis, 2011, p. 58.
[27] G. Ananthanarayanan, S. Kandula, A.G. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in Map-Reduce clusters using Mantri, in: OSDI, 2010, p. 24.
[28] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, HotCloud 10 (10–10) (2010) 95.
[29] P.-Y. Yin, S.-S. Yu, P.-P. Wang, Y.-T. Wang, Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization, Syst. Softw. 80 (5) (2007) 724–735.
[30] D. Datla, H.I. Volos, S. Hasan, J.H. Reed, T. Bose, Task allocation and scheduling in wireless distributed computing networks, Analog Integr. Circuits Signal Process. 69 (2–3) (2011) 341.

[31] S.M. Shatz, J.-P. Wang, M. Goto, Task allocation for maximizing reliability of distributed computer systems, IEEE Trans. Comput. 41 (9) (1992) 1156–1168.
[32] S. Kartik, C.S.R. Murthy, Task allocation algorithms for maximizing reliability of distributed computing systems, IEEE Trans. Comput. 46 (6) (1997) 719–724.
[33] A. Doğan, F. Özgüner, Biobjective scheduling algorithms for execution time–reliability trade-off in heterogeneous computing systems, Comput. J. 48 (3) (2005) 300–314.
[34] P. Pandey, D. Pompili, MobiDiC: Exploiting the untapped potential of mobile distributed computing via approximation, in: Proc. IEEE Int'L Conf. on Pervasive Computing & Communications, 2016, pp. 1–9.
[35] M.D. Kristensen, N.O. Bouvin, Scheduling and development support in the scavenger cyber foraging system, Perv. Mobile Comput. 6 (6) (2010) 677–692.
[36] C. Shi, V. Lakafosis, M.H. Ammar, E.W. Zegura, Serendipity: Enabling remote computing among intermittently connected mobile devices, in: Proc. 13th ACM Int'L Symp. on Mobile Ad Hoc Networking & Computing, 2012, pp. 145–154.
[37] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, Scalable and low-latency data processing with stream mapreduce, in: Proc. IEEE 3rd Int'L Conf. on Cloud Computing Technology & Science, 2011, pp. 48–58.
[38] H. Arnljot, M. Rausand, System Reliability Theory: Models and Statistical Methods, Vol. 420, John Wiley & Sons, 2009.
[39] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, 2009.
[40] G. Attiya, Y. Hamam, Task allocation for maximizing reliability of distributed systems: A simulated annealing approach, Parallel Distrib. Comput. 66 (10) (2006) 1259–1266.
[41] A. Sangroya, S. Bouchenak, D. Serrano, Experience with benchmarking dependability and performance of MapReduce systems, Perform. Eval. 101 (C) (2016) 1–19.
[42] M. Roy, S. Schmid, G. Tredan, Modeling and measuring graph similarity: The case for centrality distance, in: Proc. 10th ACM Int'L Workshop on Foundations of Mobile Computing, 2014, pp. 47–52.
[43] M. Choy, D. Kim, J.-G. Lee, H. Kim, H. Motoda, Looking back on the current day: interruptibility prediction using daily behavioral features, in: Proc. ACM Int'L Joint Conf. on Pervasive & Ubiquitous Computing, 2016, pp. 1004–1015.
[44] Z. Mao, J. Ma, Y. Jiang, B. Yao, Performance evaluation of WiFi Direct for data dissemination in mobile social networks, in: Proc. IEEE Symp. on Computers & Communications, 2017, pp. 1213–1218.
[45] N. Laptev, K. Zeng, C. Zaniolo, Early accurate results for advanced analytics on mapreduce, Proc. VLDB Endow. 5 (10) (2012) 1028–1039.
[46] I. Goiri, R. Bianchini, S. Nagarakatte, T.D. Nguyen, ApproxHadoop: Bringing approximations to mapreduce frameworks, in: ACM SIGARCH Computer Architecture News, 2015, pp. 383–397.
[47] K. Chen, H. Shen, Maximizing P2P file access availability in mobile ad hoc networks though replication for efficient file sharing, IEEE Trans. Comput. 64 (4) (2015) 1029–1042.
[48] J. George, C.-A. Chen, R. Stoleru, G. Xie, Hadoop MapReduce for mobile clouds, IEEE Trans. Cloud Comput. (2016).
[49] M.M. Gaber, J.B. Gomes, F. Stahl, Pocket Data Mining: Big Data on Small Devices, Springer, 2014.
[50] A. Kim, S. Kang, U. Lee, LetsPic: Supporting in-situ collaborative photography over a large physical space, in: Proc. ACM Int'L Conf. on Human Factors in Computing Systems, 2017, pp. 4561–4573.
[51] K. Kim, H. Zabihi, H. Kim, U. Lee, TrailSense: A crowdsensing system for detecting risky mountain trail segments with walking pattern analysis, Proc. ACM Interact. Mobile Wearable Ubiquitous Technol. 1 (3) (2017) 65.
[52] M. Conti, S. Giordano, Mobile ad hoc networking: Milestones, challenges, and new research directions, IEEE Commun. Mag. 52 (1) (2014) 85–96.
[53] Y. Wang, R. Chen, D.-C. Wang, A survey of mobile cloud computing applications: Perspectives and challenges, Wirel. Pers. Commun. 80 (4) (2015) 1607–1623.
[54] I. Yaqoob, E. Ahmed, A. Gani, S. Mokhtar, M. Imran, S. Guizani, Mobile ad hoc cloud: A survey, Wirel. Commun. Mobile Comput. 16 (16) (2016) 2572–2589.
[55] D. Kovachev, T. Yu, R. Klamma, Adaptive computation offloading from mobile devices into the cloud, in: Proc. IEEE 10th Int'L Symp. on Parallel & Distributed Processing with Applications, 2012, pp. 784–791.