The Optimization Potential of Volunteer Computing for Compute or Data Intensive Applications

Wei Li and William W. Guo
School of Engineering & Technology, Central Queensland University, Australia
Email: w.li@cqu.edu.au; w.guo@cqu.edu.au

Abstract —The poor scalability of Volunteer Computing (VC) hinders the application of it because a tremendous number of volunteers are needed in order to achieve the same performance as that of a traditional HPC. This paper explores optimization potential to improve the scalability of VC from three points of view. First, the heterogeneity of volunteers' compute-capacity has been chosen from the whole spectrum of impact factors to study optimization potential. Second, a DTH (Distributed Hash Table) based supporting platform and MapReduce are fused together as the discussion context. Third, transformed versions of work stealing have been proposed to optimize VC for both compute- and data-intensive applications. On this basis, the proposed optimization strategies are evaluated by three steps. First, a proof-of-concept prototype is implemented to support the representation and testing of the proposed optimization strategies. Second, virtual tasks are composed to apply certain compute- or data-intensity on the running MapReduce. Third, the competence of VC, running the original equity strategy and the optimization strategies, is tested against the virtual tasks. The evaluation of results has confirmed that the impaired performance has been improved about *24.5%* for compute-intensive applications and about *19.5%* for data-intensive applications.

Index Terms—Optimization, volunteer computing, work stealing, big data, MapReduce, scalability

I. Introduction

Large scale scientific projects or big data applications are the two main areas that are in need of HPC. Recently the need of HPC goes an ascending tendency but on the contrary, there is no indication of a cheaper way to access HPC. To clarify this claim, the historical business transactions and social events have become the big data of an enterprise. Nowadays, big data is important to an enterprise because it supports smart decision, prediction of business trends, deepening customer engagement or optimizing business operations [1]. Big data is so coined because it cannot be processed by a single commodity computer in a reasonable amount of time. Consequently a HPC is necessary for big data processing. However, not every enterprise, particularly the small or medium enterprises, can afford a HPC center. The situation is similar to the scientific areas. A scientific project is either compute-intensive or data-intensive, and it needs a HPC for parallel or distributed processing. Unfortunately not every scientific project is able to access a HPC in a cheap way.

To explore a cheaper alternative to traditional HPC, Volunteer Computing (VC) [2] has been successfully applied to a number of scientific projects, such as SETI@Home [3], ATLAS@Home [4] and DrugDiscovery@Home [5], but not yet extensively applied to data-intensive applications. There are mainly two reasons that restrict extensive application of VC. First, the master/worker structure hinders scalability in terms of single point failure and bottleneck performance of the central servers [6]. Second, whether VC scales for big data applications has not been well studied by the existing literature. In our previous work [7-9], the scalability of VC has been confirmed for big data applications, which have a large amount of data exchanges in the course of computing. Our previous study results are promising because VC is effective for both compute- and data-intensive applications. That implies it is practical to use cheaper VC as an alternative to the expensive HPC. Our previous results also propose a challenge that VC evidently does not scale as efficiently as traditional HPC. For example, for the same speedup of a big data processing, a HPC needs *8.5K* computing nodes but VC needs *33K* (with *30%* churn) volunteers. The heterogeneous and opportunistic environments of VC contribute to such an inefficiency. To list the whole spectrum of impact factors, the heterogeneous compute-capacity, the communication cost, the data exchange during the course of computing, the round trip time of lookup of a task and the churn pattern comprise such a dynamic environment. The scope of this paper is to explore the optimization potential of VC from two aspects. The first is to improve the impaired performance caused only by the heterogeneity of compute-capacity of volunteers among the whole spectrum of impact factors. The other is to apply the optimization to both compute- and data-intensive applications.

To understand the impact of heterogeneity, volunteers are the internet computers having big difference in compute-capacity. When the computing work is evenly divided into a number of tasks, it must happen that some volunteers are overloaded and some volunteers are underutilized. That situation, fast volunteers are idles and at the same time slow volunteers hog some tasks, is the reason to slow down the overall progress of computing. The ideal situation is no idle time, that is, every volunteer

Manuscript received March 2, 2019; revised September 5, 2019.
doi:10.12720/jcm.14.10.971-979

keeps busy in the course of computing till the end of entire work. It is evident that the way, grabbing a task and hogging it to the end, is an inefficient scheduling in the face of heterogeneous compute-capacity of volunteers. On the contrary, when there is competing for tasks, a slower volunteer yielding its task to a faster volunteer optimizes the situation.

As an optimization strategy for multithreading, work stealing was originally proposed in the area of parallel computing by Blumofe and Leiserson [10] to allow the underutilized processors steal threads from overloaded processors. In the environment of multicore and shared or distributed memory systems [11]-[13], parallel computing uses processors in a time-varying manner; the effectiveness of dynamic workload balancing by work stealing was confirmed in such an environment. The goal of this paper is to transform work stealing into a version that fits for a VC environment, maximizing the overall speedup when it scales. Such a transformation faces two challenges, one is the dynamic and opportunistic VC environment, and the other is the variety of demanding applications. The approach to this study consists of the following steps:

- Propose a supporting model to represent VC environments and place the two main types of compute- and data-intensive applications on it.
- Scrutinize the properties of compute- and data-intensive applications in order to explore the optimization potential in the face of heterogeneous compute-capacities of volunteer.
- Propose two transformed versions of work stealing in regards to compute-intensive and data-intensive applications to cope with the impaired performance caused by the heterogeneity of volunteers' compute-capacities.
- Implement a proof-of-concept prototype of the model and optimization strategies.
- Compose virtual tasks with certain compute- and data-intensity and conduct stress tests to compare the original equity strategy with the proposed work stealing strategies.

The preliminary evaluation result of this paper has demonstrated that work stealing has improved the overall performance about *24.5%* for the compute-intensive applications and about *19.5%* for data-intensive applications.

The rest of this paper has been structured as follows: related work is reviewed in Section 2. The VC supporting model and MapReduce paradigm are fused in Section 3. On the basis of the supporting models, Section 4 is to identify the key conditions and optimization potential for both compute-intensive and data-intensive applications. Section 5 transforms work stealing into two versions to optimize the two types of application in the face of heterogeneous and unreliable volunteers. Section 6 evaluates the effectiveness of the optimization. Section 7 concludes the study of this paper that certain improvement has been achieved for both types of application and proposes future work.

II. Related Work

There has been a very limited number of direct applications of work stealing to VC for either compute- or data-intensive computation. The following literature studies work stealing for some particular concerns in the area of parallel or distributed computing and is worth to have a review of them. Perarnau and Sato [13] studied the impact on the overall performance by the different selection strategies of computing nodes (termed victims) to steal a piece of work from. Their study results from a HPC environment showed that the best strategy could scale up to *8,192* nodes. A similar work was Dinan et al. [11]. They studied some work stealing strategies on a distributed memory system and evaluated the scalability of those strategies on a few of benchmarks. They found different strategies behaved differently for the benchmarks and scaled differently in a HPC cluster of *2,310* nodes with *8,192* processors. The work of Tallent and Mellor-Crummey [14] and the work of Kumar et al. [15] were similar because both studied how the stealing granularity and the concerned overhead had impact on the overall performance of work stealing. Their work were done on different HPC hardware and software environments, but they both concluded that those concerned factors affected the performance. They both proposed some optimization strategies to reduce the cost of stealing in order to improve the performance to a certain extend. Vu and Derbel's [16] work is closer our work of this paper because both aimed at designing a work stealing algorithm to cope with the performance issues that were caused by the heterogeneity of distributed environments. To achieve this goal, Vu and Derbel constructed an algorithm by fine tuning an existing work stealing algorithm through introducing some adaptive operations to increase work locality and decrease stealing cost. The algorithm was able to save about *30%* of computing time on a HPC cluster with *128* computing nodes. Their work was different from our work of this paper because their study environment was not an opportunistic VC environment. That is, their computing nodes did not commit churn.

In the area of data-intensive computing, Fadika et al. [17] studied extensively about the impacts on the performance of computing by certain data operations, the number of tasks, the replication of data and the network bandwidth. On a HPC environment, their results confirmed the impact of these factors on a few of big data applications. Dede *et al.* [18] did a similar extensive study on the impact from the heterogeneity, unreliability or unstable computing power of computing nodes. They contrasted their own implementation of MapReduce [19] with the other two existing implementations in the three testing environments: a heterogeneous cluster, a homogeneous but load-imbalanced cluster and a cluster

with unreliable nodes. Their results demonstrated the difference of these implementations in processing data-intensive, CPU-intensive and memory-intensive applications. Su *et al*. [20] focused on the impact of heterogeneity of compute-capacity and the dedication of computing nodes. Their optimization included the identification of *stragglers* to adjust task allocation dynamically to the higher dedicated nodes. Their optimization also included the locality of data, moving the reduce tasks to the computing node, where the largest region of an intermediate key is stored, rather than exchanging a large amount of data. The motivation of Zhang *et al*. [21] was similar to our work of this paper because both regard that the main reason of the poor performance is the equity task allocation, without considering the heterogeneity of computing nodes. The straggler nodes slow down the overall progress of map or reduce steps, particularly for the reduce step, which cannot start until the slowest node completes the map task and exchanges the data to the reduce tasks. Zhang *et al*. [21] optimized the situation by allowing faster nodes to steal some work from the stragglers. The difference between their work and our work of this paper is: theirs was a reliable environment but ours is the unreliable opportunistic VC environment. Jothi and Indumathy [22] took the effort in the same direction of ours. They split a big data into data sets with different sizes and assigned a larger data set to a faster computing node to balance the workload between them. Their work showed optimized results, which were able to alleviate the impact of the heterogeneous computing nodes on the overall performance of MapReduce in a distributed environment. The limitation of their work was the small scale of the application and the assumption of node reliability. Yildiz *et al*. [23] coped with the impact of unreliability of computing nodes by proposing a failure recovery model. They prioritized tasks and recovered the tasks with higher priority by pre-empting the tasks with lower priority with the goal to recover tasks in certain time. When re-scheduling recovered tasks, their model was assisted by the data locality. Their results showed the reduction in overall completion time.

All the aforementioned review comes to the issue that the optimization potential has not been well explored to improve the impaired performance caused by any impact factors in the whole spectrum in dynamic opportunistic environments. This paper takes the effort in this direction, aiming at explore the optimization potential of VC against one of the impact factors: the heterogeneity of volunteers' compute-capacities and for both compute-intensive and data-intensive applications.

III. The Supporting Model

To study optimization potential, we first need a VC supporting model to ensure volunteer communication and the reliability and lookup performance of overlay. There are many off-the-shelf peer-to-peer overlays in current literature, either unstructured or structured. They are different in reliability, scalability and lookup performance. Among them, the DHT protocol overlay Chord [24] is chosen as the VC supporting model of this study for the following reasons.

- Chord tolerates churn and has built-in reliability. The uploaded key-value pairs are transparently replicated.
- Chord ring topology is always maintained by Chord stabilization protocol, which periodically updates the routing information to reflect churn.
- Chord consistent hashing ensures the storage load balance among volunteers on the overlay.
- Chord guarantees the logarithmic lookup performance of a key-value pair $O(_{log}n)$, where $n \in N_1=\{1, 2, ...\}$ is the number of peers currently on the Chord overlay.

Built on Chord, the basic communication for task scheduling will involve the standard DHT operations only through the extended form of key-value pairs, which are always stored on the overlay. A task is an extend key-value pair in the form of $<K_{i0}, t_i>$, where $i \in \{1, 2, ..., m\}$ and m is the number of tasks. The scheduling of a compute-intensive task consists of the following steps. If a task is acquired by a volunteer, it will be changed into the form of $<K_{i1}, ts_i, t_i>$, where ts_i is the timestamp when the task is updated. If the volunteer leaves before finishing the task, the progress will be check-pointed and the task will be changed back to the form of $<K_{i0}, t_i>$. If the volunteer crashes before finishing the task, the progress must be wasted. Because of the agreement on the time interval of updating timestamp ts_i, the task will be identified as crashed by an out-of-date timestamp. The compute-intensive applications is simpler than data-intensive applications in task scheduling because the former is one step computing for each task without data exchange in the course of computing. On the contrary, the latter could have a large amount of data exchange in the middle of computing. The study of optimization potential for compute-intensive applications will be directly constructed on the above extended Chord model.

The data-intensive application is more complex, consequently, we need the second model for the application itself. Although there are a number of application models [25] to process big data, MapReduce [19] is chosen for this study because it has firm theoretical model, off-the-shelf software products and successful application cases. A big data application is performed by MapReduce by three steps as shown in Fig 1. A big data problem is divided into a number of map and reduce tasks and the results of map tasks are the input of reduce tasks. That involves a large amount of data exchange in the shuffle step, which is the serialization part of the programming paradigm that applies to a variety of HPCs but challenges VC for bandwidth and reliability requirements. Another reason of choosing Chord is the natural match between MapReduce and Chord DHT, where the latter provides key-value pairs operations and hash functions for the former to support

the sorting operations in the map step, classification operations in the shuffle step and merger operations in the reduce step.

The basic scheduling of MapReduce tasks in the extended Chord is the same as that for compute-intensive tasks, but the scheduling model needs to go a further extension for the shuffle step. For such an extension, a hash function is used to find the identity of a reduce task, to which a part of the result set of a map task redistributes. For performance modelling, the *redistribution factor* is proposed to quantify the number of reduce tasks into which the result set of a map task needs to be injected by the hash function. For example, if redistribution factor is *50*, the result set of a map task needs to be divided and redistributed into *50* reduce tasks.

IV. The Conditions and Optimization Potential

The optimization potential that this paper explores is to improve the impaired performance caused by the heterogeneity of compute-capacity of volunteers. Naturally when the entire work is evenly divided into multiple tasks, the overall performance achieves the highest if all the volunteers have the same compute-capacity. However, volunteers are from the Internet, having heterogeneous compute-capacity. Furthermore it is unable to predict the compute-capacity of the next coming volunteer. Even though a volunteer can honestly report its compute-capacity when joining, a re-calculation of the current compute-capacity of the overlay for a redistribution of the current workload is impractical. The main reason is: although the redistribution is achieved at cost of disturbing the whole overlay, the redistribution could soon become invalid due to the churn of volunteers, that is, the join of new volunteers or the leave or crash of the existing volunteers. This section is firstly to list the conditions of compute-intensive or data-intensive application. On this basis, this section identifies two optimization potentials in regards to the two types of application. From this section, the original compute- or data-intensive problem is called *work*, once the work is divided, each part of the division is called a *task*.

A compute-intensive application has the following properties.

- The computing load of the entire work is very high.
- The entire work can be divided into a number of tasks, and each task may be divided into a number of tasks and so on. A task cannot be divided if it had already reduced to the minimum division granularity.
- The size of the entire work is just sound, thus, the download/upload time of the work is trivial. Consequently, the download/upload time of an individual task is trivial as well.
- The entire work is one step computing of all the divided individual tasks. That is, there is no data exchange between the tasks in the course of computing.

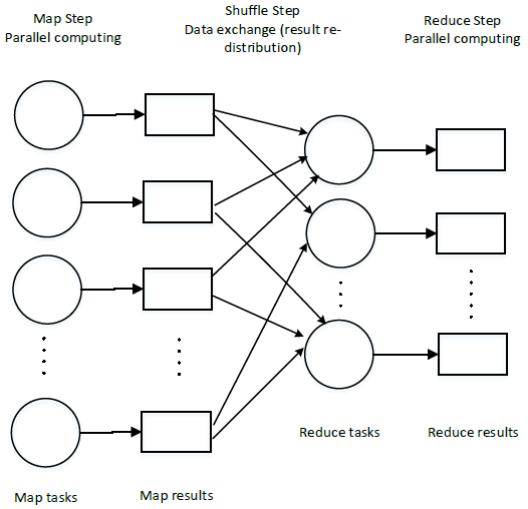

Fig. 1. The MapReduce abstraction of big data processing

- The final result of the entire work is the simple combination of the results of each task. Thus the result of each task just needs to be uploaded to the owner, which is assumed reliable on the overlay.
- The size of the result set of the entire work is just sound, thus, the download/upload time of the entire result is trivial. Consequently, the download/upload time of the result set of an individual task is trivial as well.
- Lookup of an available task is the Chord provable lookup performance $O(_{log}n)$, where $n \in N_1$ is the number of volunteers currently on the overlay.
- The owner of the entire work is the only peer on the overlay at the beginning of computing; it is also the result collector as well; it is reliable in the whole course of computing.

At the beginning of computing, the entire work is with the work owner. To make use of the above conditions, the optimization should happen each time when a new volunteer joins or when an existing volunteer finishes its current task. At that moment, the volunteer is allowed to steal a task from another volunteer. The first stealing happens when the first joining volunteer steals a task from the work owner. Thereafter, stealing can happen between any volunteers. To minimize the cost of stealing, the volunteer is limited to get the first available task only. Although each stealing does not consider the compute-capacity of the involved two parties and is not optimized, the performance of the overlay as a whole is optimized. That reason is that over time any volunteers can steal tasks with each other, the fast volunteer can always get tasks to do or the slow volunteers always yield tasks to the fast volunteers. That is, the tasks are always prioritized for the fast volunteers to dynamically balance the computing load on the overlay.

A data-intensive application has the following properties.

- The computing load of the entire work is very high and the entire work can be accomplished by MapReduce paradigm in a distributed environment.

- The entire work can be divided into multiple map tasks along with their own data sets and reduce tasks with empty data sets.
- The data set of the entire work is very big. Consequently, the download/upload time of the data set of each individual task is non-trivial.
- There is a large amount of data exchange in the shuffle step in the course of computing.
- Based on point 4, the intermediate result set of a map task needs to be redistributed into the empty data sets of a number of reduce tasks.
- The final result of the entire work is the simple combination of the result sets of each reduce task. Thus the result set of each reduce task just needs to be uploaded to the overlay.
- The size of each intermediate result set or a final result set matters. Thus the download/upload time of a result set is non-trivial.
- Lookup of an available task is the Chord provable lookup performance $O(_{log}n)$, where $n \in N_1$ is the number of volunteers currently on the overlay.
- None of the volunteers is reliable, but the overlay as a whole is reliable and there are always some volunteers on the overlay at any time during the course of computing.
- The compute-capacities of volunteer are heterogeneous, but the heterogeneity is evenly distributed when the number of volunteers is fairly large.

In the aforementioned conditions, the cost of data exchange matters and makes work stealing directly from an in-computing volunteer impractical. To make use of other conditions, the optimization is based on the classification of compute-capacity of volunteers. The classification quantifies the compute-capacity of similar volunteers as a tier. The number of tasks can be pre-determined for each tier of volunteers. If a volunteer is allowed to consume the tasks of its own tier, the situation that a slow volunteer hogs tasks to make a fast volunteer idle cannot happen. Furthermore, if a fast volunteer is allowed to consume (steal) tasks from lower tiers when the tasks in its own tier run out, the optimization is achieved. The rationale is that the tasks are always made available for faster volunteers.

V. The Optimization Model

This section transforms the original work stealing into two versions: version 1 (V_1) for compute-intensive applications and version 2 (V_2) for data-intensive applications. The transformed work stealing versions are to make use of the optimization potentials as described in last section, aiming at reducing the impact of heterogeneous compute-capacity of volunteers and improving the impaired performance. Although the optimization strategies of V_1 and V_2 are different, the goal that faster volunteers are always prioritized for tasks is the same.

A. Work Stealing V_1

This version of work stealing is to cope with the heterogeneous compute-capacity of volunteers for compute-intensive applications. At the beginning, we assume that there is only one volunteer, the owner of the work, on the overlay. The first joined volunteer will contact the work owner to get *half* of the unfinished part of the work, that is, a task to compute. The second joined volunteer may contact the owner of the work or the first joined volunteer for a task. Over time, there are a number of volunteers on the overlay; the existing volunteers may leave or crash on the overlay or the new volunteers may join on it.

- When a volunteer leaves, its progress is check-pointed. When a volunteer crashes, its progress is not able to be check-pointed and will be fully wasted.
- A newly joined volunteer will firstly try to pick up the tasks that are discarded by the left volunteers or crashed volunteers.
- If the above discarded tasks are not found, the volunteer will steal a task from an in-computing volunteer.
- To steal a task from an in-computing volunteer, the in-computing volunteer will pause its current computing and enter a servicing state, separating half of the unfinished task to the stealing-volunteer and then resuming its computing progress for the reserved half of the task.
- When a volunteer finishes its current task, it will upload the result to the owner of the work.
- Every volunteer needs to update the timestamp of its task on an agreed time interval. An unfished task will be check-pointed and marked by a leaving volunteer; an unfinished task of a crashed peer will be identified by an out-of-date timestamp.

The aforementioned strategy is an optimization because the stealing may not disturb the current computing (if a discarded task is found from the left or crashed peers) or may just disturb a single volunteer's computing. However at the same time, a fast volunteer is always allowed to have a task to do but none of the slow peers can hog tasks. The workload is dynamically balanced. The cost of this optimization is the communication between volunteers.

B. Work Stealing V_2

This version of work stealing is to cope with the heterogeneous compute-capacity of volunteers for data-intensive applications. This version of work stealing is based on tiering of volunteers and tasks.

- The compute-capacities of volunteer are grouped into a number of tiers C_1, C_2,..., C_p, where $p \in N_1$ is the total number of tiers of the compute-capacity of volunteers on the overlay. The total capacity of the overlay is $C_1 + C_2 + \ldots + C_p$. For example, if there are *8* tiers of the compute-capacity of *1, 2, ..., 8*, where *capacity 2* means *2* times faster than *capacity 1* and so on, the total compute-capacity of the overlay is

1+2+…+8=36. If there are *8* tiers of the compute-capacity of $1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{8}$ where *capacity ½* means *2* times slower than *capacity 1* and son on, the total compute-capacity of the overlay is $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{8}=\frac{761}{280}$.

- The number of tasks that will be allocated to the *ith* tier is determined by the ratio between the compute-capacity of the *ith* tier C_i and the total capacity of the overlay by formula: $\frac{C_i}{1+C_1+C_2\ldots+C_p} \times total\ number\ of\ tasks$, where $i \in \{1, 2, \ldots, p\}$. For example, if there are *8* tiers of compute-capacity of *1, 2, …, 8*, the number of tasks of the *4th* tier is *4/36=1/9* of the total tasks. If there are *8* tiers of the compute-capacity of $1, \frac{1}{2}, \frac{1}{3}, \ldots \frac{1}{8}$, the number of tasks of the *4th* tier is $\frac{1}{4}/\frac{761}{280}$ *=70/761* of the total tasks.
- A volunteer is restricted to lookup tasks in its own tier and the tiers of a lower compute-capacity.
- When a volunteer leaves, its progress is check-pointed. When a volunteer crashes, its progress is not able to be check-pointed and will be fully wasted.
- If a volunteer cannot find any tasks from the tier task pools, the volunteer will pick up the tasks that are discarded by the left volunteers or crashed volunteers.

The above strategy is an optimization because the volunteers of a slower tier will never able to access the amount of the tasks that are beyond its tier capacity. The volunteers of a faster tier can consume (steal) all the slower tiers' tasks if they finish their own tiers' tasks.

VI. The Evaluation

The VC supporting model and the MapReduce model that are proposed in Section 3 have been fully implemented on the Open Chord platform [26]. The optimization strategies of work stealing V_1 and V_2 have been fully implemented on top of the supporting models. This section evaluates the effectiveness of the optimization strategies by providing the evaluation scenarios, experimental settings and analysis of results.

A. Work Stealing V_1

The evaluation of work stealing needs to contrast its performance with the performance of *equity* strategy, which evenly divides the entire work into a number of tasks and puts them in a single pool. By the equity strategy, every volunteer has equal chance to get a task from the pool and never yields the task unless the volunteer leaves or crashes on the overlay. To contrast performance, the following settings in Table I use two different task numbers for the equity strategy. When the entire workload is the same, a larger task number implies smaller tasks that slow volunteers can hog. The general setting is a *5K* to *40K* volunteer overlay, of which *50%* of volunteers commit churn in the course of computing. The compute-capacity of volunteers is classified into *8* tiers $(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{8})$. The volunteers join the overlay in a fixed interval of *20* (randomly chosen). The churn (leave or crash) volunteers are randomly chosen and a churn volunteer must stay on the overlay at least *20* (randomly chosen) before committing churn. The churn occurs in a fixed interval of *20* (randomly chosen) but the churn time is randomly chosen in the interval for every churn volunteer. The injected churn pattern is depicted in Fig. 2. The round trip time to locate a task on the overlay is *4*, so the performance to find a task is $4_{log}n$, where *n* is the number of volunteers currently on the overlay. Particularly for the equity strategy, there are two settings with different numbers of task and different compute-loads of an individual task. However, the workload of the entire work is the same (*1,400,000×16,000=2,800,000×8,000=22,400,000,000*), which is also the workload for the work stealing strategy. Particularly for work stealing strategy, when a volunteer steals a task from another volunteer, both parties need to pause for a certain period time to make the stealing happen. That pause time is termed as stealing cost, which is set as *30* (randomly chosen) in the evaluation. The minimum stealing granularity is *4,000*. That is, when the workload of a task is greater or equal to *4,000*, it can be divided and stolen.

TABLE I: THE GENERAL AND SPECIAL SETTINGS OF COMPUTE-INTENSIVE APPLICATIONS FOR THE WORK STEALING AND EQUITY STRATEGY

	Scenario Variable		Value
1. General Setting	The number of volunteers		5,000 to 40,000
	The percentage of leave and crash nodes		50%
	The round trip time to look up a task		4
	The volunteer join interval		20
	The churn starts position after join		20
	The churn occurring interval		20
	The tiers of compute-capacity		8 $(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{8})$
	Download of a task or upload of a result set		1/1
2. Setting for equity	The number of tasks	Setting 1	1,400,000
		Setting 2	2,800,000
	The compute-load of a task	Setting 1	16,000
		Setting 2	8,000
3. Setting for work stealing	The workload of entire work		22,400,000,000
	The stealing cost		30
	The minimum stealing granularity		4,000

The evaluation result has been contrasted for the original equity strategy for the two task granularities (*Setting 1* and *Setting 2*) and the work stealing strategy (*Setting 3*) in Fig. 3.

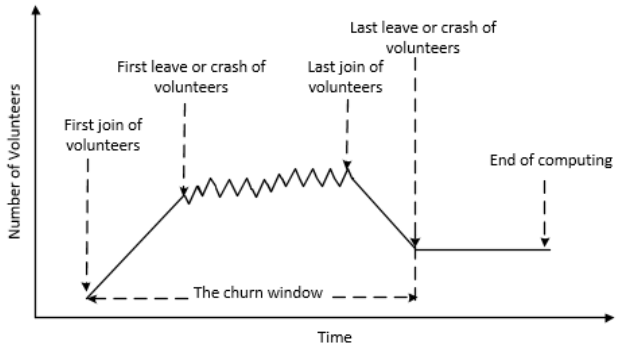

Fig. 2. The injected churn pattern for compute-intensive applications

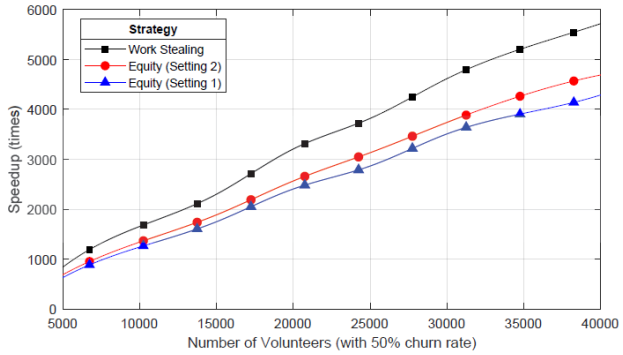

Fig. 3. The speedup comparison of the equity strategy and work stealing strategy for compute-intensive applications

For the overlays with different volunteer numbers from *5,000* to *40,000* and the same churn rate *50%* (*25%* leaves and *25%* crashes), the reduction of task granularity from *Setting 1* (*16,000*/task) to *Setting 2* (*8,000*/task) has improved the performance about *5%* to *7%* of the average of *6.3%*. That means that reducing task granularity optimizes the overall performance to a small extent for equity strategy. The performance has been improved significantly by the work stealing strategy between *18%* and *20%* of the average *18.5%* over the original equity strategy of *Setting 2* and between *24%* and *25%* of the average *24.5%* over *Setting 1*.

B. Work Stealing V_2

The evaluation of work stealing for data-intensive computing also needs to contrast its performance with the performance of *equity* strategy. To achieve the goal, the following settings in Table II considers the communication cost for downloading a data set of a map or reduce task or for uploading an intermediate result set in the shuffle step or the final individual result set at the end of computing. Both strategies use the same settings, but by the equity strategy, a volunteer is allowed to consume tasks equally from a single pool; work stealing allows a volunteer to consume tasks in its own tier's or a lower tier's pool.

TABLE II: THE GENERAL AND SPECIAL SETTINGS OF DATA-INTENSIVE COMPUTING FOR THE WORK STEALING AND EQUITY STRATEGY

Scenario Variable	Value
The number of volunteers	5,000 to 40,000
The number of map tasks	1,400,000
The number of reduce tasks	1,400,000
The compute-load of a map or reduce task	8,000
The size of a map or reduce task or a result set	50MB
The total data size	140TB (1,400,000×2×50MB)
The tiers of compute-capacity	8 $(1,\frac{1}{2},\frac{1}{3},\ldots,\frac{1}{8})$
The cost of download/upload a task or a result set	16/80
The round trip time to look up a task	4
The redistribution factor of a result set	100
The percentage of leave and crash volunteers	50%
The volunteer join interval	20
The churn starts position after join	20
The churn occurring interval	20

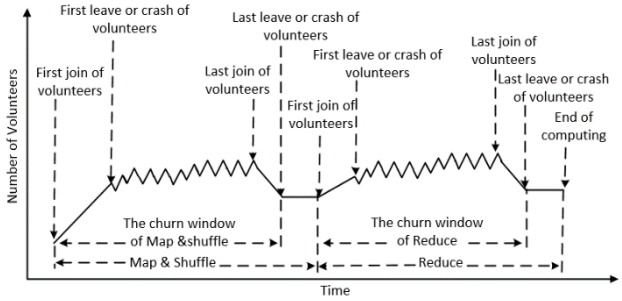

Fig. 4. The injected churn pattern for data-intensive applications

The general setting is for a *5,000* to *40,000* volunteer overlay, of which compute-capacity is classified into *8* tiers $(1,\frac{1}{2},\frac{1}{3},\ldots,\frac{1}{8})$. The entire work consists of *1,400,000* map tasks and also *1,400,000* reduce tasks. The compute load of each task is *8,000*. The size of a task or a result size is *50MB*, which requires *16/80* for download/upload on a very general home internet speed of *25/5Mbps*. The total size of big data is *2×1,400,000×50MB=140TB*. Each intermediate result set from a map task needs to be redistributed into *100* reduce tasks in the shuffle step. Thus the redistribution of an intermediate result set needs *100* lookups. The round trip time to locate a task on the overlay is *4*, so the performance to find a reduce task is $4_{log}n$, where n is the number of volunteers currently on the overlay. The dynamic feature of the overlay is of *50%* of volunteers committing churn in the course of computing. The volunteers join the overlay in a fixed interval of *20* (randomly chosen). The churn (leave or crash) volunteers are randomly chosen and a churn volunteer must stay on the overlay at least *20* (randomly chosen) before committing churn. The churn occurs in a fixed interval of *20* (randomly chosen) but the churn time is randomly chosen in the time period for every churn volunteer. The injected churn pattern happens twice, one for the map and the shuffle step and the other for the reduce step as depicted in Fig. 4.

The evaluation results have been contrasted for the original equity strategy and the work stealing strategy in Fig. 5. For the overlays with different volunteer numbers

from *5,000* to *40,000* and the same churn rate *50%* (*25%* leaves and *25%* crashes), the performance has been improved significantly by the work stealing strategy between *18%* and *21%* of the average *19.5%*.

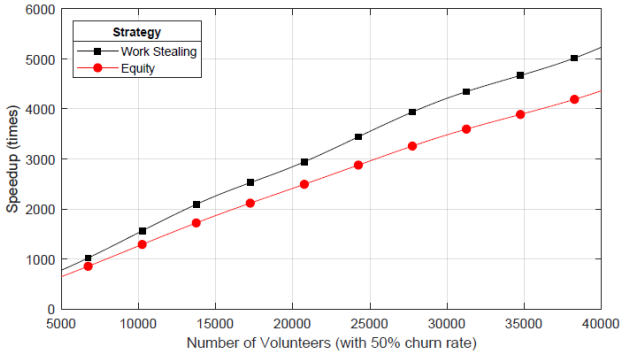

Fig. 5. The speedup comparison of the equity strategy and work stealing strategy for data-intensive applications

VII. Conclusions and Future Work

Among the whole spectrum of impact factors, the heterogeneity of compute-capacity has been chosen to explore the optimization potential of VC for both compute-intensive and data-intensive applications. Because the communication cost of downloading a task or uploading a result set is trivial for compute-intensive applications, the first version of work stealing is to balance workload of volunteers by allowing an idle volunteer to steal a task from another running volunteer directly. When the communication cost matters for the data-intensive applications, the second version of work stealing allocates tasks into tiers in regards to the tiers of compute-capacity of volunteers. This version allows a higher tier volunteer to steal tasks from a lower tier. The proposed work stealing strategies have been modelled on the extended Chord DHT protocol and MapReduce big data paradigm. The evaluation results are promising: about *24.5%* and *19.5%* have been improved for the impaired performance of compute-intensive and data-intensive applications respectively. The results have confirmed the optimization potential for constructing real world VC by using the donated compute-power and storage from the internet volunteers.

The future work of this study is to explore optimization potential of VC to cope with other impact factors in the spectrum regarding the opportunistic VC environments. One direction to optimization is locality of data processing. In that effort, volunteers are processing the data that is only stored on themselves. This version of optimization fuses together the processing capability and storage capacity of VC to minimize the communication cost in the course computing.

References

[1] Oracle 2016. An Enterprise Architect's Guide to Big Data - Reference Architecture Overview, Oracle Enterprise Architecture White Paper. [Online]. Available: http://www.oracle.com/technetwork/topics/entarch/articles/oea-big-data-guide-1522052.pdf

[2] L. Sarmenta, "Volunteer computing," PhD thesis, Massachusetts Institute of Technology, 2001.

[3] E. J. Korpela, SETI@home, BOINC, and Volunteer Distributed Computing, *Annual Review of Earth and Planetary Sciences*, vol. 40, pp. 69-87, 2012.

[4] ATLAS@home. (2018). [Online]. Available: http://lhcathome.web.cern.ch/projects/atlas

[5] DrugDiscovery@Home (2018). [Online]. Available: http://www.drugdiscoveryathome.com

[6] D. P. Anderson, E. Korpela, and R. Walton, "High-Performance task distribution for volunteer computing," in *Proc. 1st IEEE International Conference on e-Science and Grid Technologies*, 2005, pp. 196-203.

[7] W. Li and W. Guo, "The competence of volunteer computing for MapReduce big data applications," *Communications in Computer and Information Science*, Springer, vol. 901, pp. 8-23, 2018.

[8] W. Li and W. Guo, "The scalability of volunteer computing for MapReduce big data applications," *Communications in Computer and Information Science*, Springer, vol. 727, pp. 153-165, 2017.

[9] W. Li and W. Guo, "Work stealing based volunteer computing coordination in P2P environments," *Journal of Communications*, vol. 12, no. 10, pp. 557-564, 2017.

[10] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720-748, 1999.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proc. International Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 53-63.

[12] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal Tree: Low-overhead tracing of work stealing schedulers," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 507-518, 2013.

[13] S. Perarnau and M. Sato, "Victim selection and distributed work stealing performance: A case study," in *Proc. 28th IEEE International Symposium on Parallel and Distributed Processing*, 2014, pp. 659-668.

[14] N. R. Tallent and J. M. Mellor-Crummey, "Identifying performance bottlenecks in work-stealing computations," *Computer*, vol. 42, no. 12, pp. 44-50, 2009.

[15] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 297-314, 2012.

[16] T. T. Vu and B. Derbel, "Link-heterogeneous work stealing," in *Proc. 4th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 354-363.

[17] Z. Fadika, M. Govindaraju, R. Canon, and L. Ramakrishnan, "Evaluating hadoop for data-intensive scientific operations," in *Proc. IEEE 5th International Conference on Cloud Computing*, 2012. pp. 67-74.

[18] E. Dede, Z. Fadika, M. Govindaraju, and L. Ramakrishnan, "Benchmarking MapReduce implementations under different application scenarios,"

Future Generation Computer Systems, vol. 36, pp. 389-399, 2014.
[19] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.
[20] Y. L. Su, P. C. Chen, J. B. Chang, and C. K. Shieh, "Variable-sized map and locality-aware reduce on public-resource grids," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 843-849, 2011.
[21] X. Zhang, Y. Wu, and C. Zhao, "MrHeter: Improving MapReduce performance in heterogeneous environments," *Cluster Computing*, vol. 19, no. 4, pp. 1691-1701, 2016.
[22] A. Jothi and P. Indumathy, "Increasing performance of parallel and distributed systems in high performance computing using weight based approach," in *Proc. International Conference on Circuits, Power and Computing Technologies*, 2015.
[23] O. Yildiz, S. Ibrahim, and G. Antoniu, "Enabling fast failure recovery in shared hadoop clusters: Towards failure-aware scheduling," *Future Generation Computer Systems*, vol. 74, pp. 208-219, 2017.
[24] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17-32, 2003.
[25] R. Casado. (2013). The Three Generations of Big Data Processing. [Online]. Available: https://www.slideshare.net/Datadopter/the-three-generations-of-big-data-processing
[26] S. Kaffille and K. Loesing. (2007). Open Chord (1.0.4) User's Manual, The University of Bamberg, Germany. [Online]. Available: https://sourceforge.net/projects/open-chord/.

Dr. Wei Li holds a PhD degree in computer science from the Institute of Computing Technology of Chinese Academy of Sciences China. He currently works for the School of Engineering & Technology, Central Queensland University Australia. His research interests include dynamic software architecture, P2P volunteer computing and multi-agent systems. Dr Wei Li has been a peer reviewer of a number of international journals, including IEEE Transactions on Software Engineering, ELSEVIER Journal of Systems and Software and John Wiley & Sons Journal of Software Maintenance and Evolution: Research and Practice, and a program committee member of more than 30 international conferences.

Dr. William Guo is currently a professor in applied mathematics and computation at Central Queensland University Australia. His research interests include applied mathematics and computational intelligence, simulation and modelling, data mining, and STEM education.