The Scalability of Volunteer Computing for MapReduce Big Data Applications

Wei Li(✉) and William Guo

School of Engineering and Technology, Central Queensland University, Rockhampton, QLD 4702, Australia
{w.li,w.guo}@cqu.edu.au

Abstract. Volunteer Computing (VC) has been successfully applied to many compute-intensive scientific projects to solve embarrassingly parallel computing problems. There exist some efforts in the current literature to apply VC to data-intensive (i.e. big data) applications, but none of them has confirmed the scalability of VC for the applications in the opportunistic volunteer environments. This paper chooses MapReduce as a typical computing paradigm in coping with big data processing in distributed environments and models it on DHT (Distributed Hash Table) P2P overlay to bring this computing paradigm into VC environments. The modelling results in a distributed prototype implementation and a simulator. The experimental evaluation of this paper has confirmed that the scalability of VC for the MapReduce big data (up to 10 TB) applications in the cases, where the number of volunteers is fairly large (up to 10K), they commit high churn rates (up to 90%), and they have heterogeneous compute capacities (the fastest is 6 times of the slowest) and bandwidths (the fastest is up to 75 times of the slowest).

1 Introduction

When the data sets of business transactions or social media become massive as termed as Big Data, the computational analysis of the big data, in order to predict business trends, deepen customer engagement and optimize operations, challenges the traditional data processing and demands newer parallel and distributed approaches and tools [16]. The issues, such as distributing the data, parallelizing the computation and synthesizing results, must be handled in a reasonable amount of time, in order to support timely smart decision. In this area, MapReduce [7] has been a successful programming paradigm developed by Google to process the large data sets, such as *crawled documents*, *inverted indices* and *web request logs*. Nowadays, MapReduce has been extensively used by the enterprises, such as Yahoo, Facebook and Microsoft, to process their enterprises big data.

MapReduce consists of 3 steps to process a big data set. In the *map step*, the original, big data set is divided into a number of small data sets, which are distributed onto a cluster of computers as *map tasks*. The data sets will be computed in parallel by the same *map function* by the entire cluster so that each computer will emit a number of <*key, value*> pairs at the end of this step. In the *shuffle step*, all the <*key, value*> pairs with the same key from the last step will be merged together in the form of <*key, a list of values*>. These pairs will be further sorted by the keys into a number of *reduce tasks*,

© Springer Nature Singapore Pte Ltd. 2017
B. Zou et al. (Eds.): ICPCSEE 2017, Part I, CCIS 727, pp. 153–165, 2017.
DOI: 10.1007/978-981-10-6385-5_14

which are redistributed onto the cluster. In the *reduce step*, the reduce tasks are computed in parallel by the same *reduce function* by the entire cluster so that each computer will emit a number of <*key, value*> pairs as the final results. The difference between the map step and the reduce step is that any 2 computers in the map step may emit the <*key, value*> pairs with the same key, but neither in the reduce step will emit a <*key, value*> pair with the same key. This is because the keys emitted from the reduce step are the same as the keys of the shuffle step that already merged by the same keys.

When MapReduce has succeeded for a variety of big data applications such as *inverted indices*, *k-means*, *classification* and more in cluster environments [1], it involves moving a large amount of data, particularly in the shuffle step, and therefore puts stress on network bandwidth and introduces runtime overhead. This concern was reflected in the research of [1], which proposed the MaRCO model to achieve a full overlap between the map computation and shuffle communication to speed up the overall performance, and [2], which proposed Meta-MapReduce to avoid big data set migration across remote sites and only transmit the very essential data for obtaining the result, in the standard venue of *grid (cluster) computing environments*. Coming to the scope of research of this paper, it remains open in the current literature whether MapReduce can be effective in the extended environment of Volunteer Computing (VC) as defined by [17]. From one aspect, VC makes use of the potential computing power from millions of volunteer computers from the Internet and has been successfully applied to large-scale scientific projects such as SETI@home [11], FiND@Home [8] and Climateprediction.net [4]. Thus instead of using expensive computing clusters (grids), it is inspiring to utilize the free volunteered computing power for MapReduce big data applications. From another aspect, MapReduce challenges VC in terms of whether the system is able to scale (1) when a cluster mainly needs data communication for the shuffle step only, VC needs data communication for all 3 steps of MapReduce; (2) when a cluster fails rarely, volunteers commit churn; (3) when a cluster consists of homogeneous machines, working in a high speed network, volunteers are heterogeneous in compute capacity, bandwidth and storage.

This paper is to extend VC to big data applications via MapReduce by proposing a DHT (Distributed Hash Table) based strategy for task scheduling and data migration, a prototype implementation to verify the functional correctness of the model, and a simulator to evaluate the scalability of the model. The evaluation aims at answering whether VC is an appropriate distributed model for big data applications like MapReduce.

The rest of this paper has been structured as: related work is reviewed in Sect. 2. Modelling MapReduce for VC environments is presented in Sect. 3. Section 4 describes the experimental settings for the evaluation of a virtual MapReduce job. Section 5 details the simulation results and analysis. Section 6 concludes the initial evaluation that VC scales for 10K peers in the opportunistic volunteer environments.

2 Related Work

Some works in the current literatures are worth to review in terms of promoting VC for MapReduce style big data applications. [6] presented a model BOINC-MR, which is based on BOINC but exploited a pull method to allow inter-peer data transfer for

moving data between mappers and reducers to speed up the shuffle step data communication and reduce the burden on the central servers. There are 2 open issues with BOINC-MR. First, the MapReduce progress, depending on direct peer communication with each other rather than on a higher level overlay, would cause a halt when peers commit churn. Second, a hybrid structure, combing super nodes and P2P rather than pure P2P, has to be used because direction communication cannot go through firewalls. When the model tried to extend VC for MapReduce applications, their experimental results could not confirm whether VC is effective for MapReduce in that the model had no better performance (over the original BOINC), even on 1 GB data set and small number (40) of peers in a grid environment without churn.

VMR [5] is an extension to their previous work [6], aiming at the execution of MapReduce tasks on large scale VC resources from the Internet by direct peer communication, tolerating transient server failure, peer failure or byzantine behaviors. Their experimental results showed that VMR performed better (in terms of the number of map and reduce tasks and the replication factors) for the MapReduce applications like word count, inverted indices, N-Gram ans NAS EP for 50, 100, 200 peers. However, how peers committed churn and how the performance could be affected by the churn were not mentioned. The results still could not confirm VC for the Internet scope, where peers are in a large number (much more than 200) and commit churn frequently.

MOON [14] extended the MapReduce middleware Hadoop [9] for adaptive task and data scheduling to offer reliable MapReduce services to VC systems that were supported by a small set of dedicated nodes. MOON tried to confirm how well the existing MapReduce frameworks performed on VC environments. To cope with churn, MOON exploited data replication and task replication and proposed corresponding management and scheduling models. When the evaluation results of MOON were provided to compare its performance with Hadoop, they also showed the scalability of the model against churn rates. Their results, in a couple of real world applications and in a small cluster environment (60 volunteers plus 6 dedicated nodes), were somehow compliant with our simulation results in this paper.

P2P-MapReduce was proposed by [15] to provide a reliable middleware for MapReduce in dynamic cloud environments. The main idea was to use backup of data and tasks to cope with peer churn. Peers were treated differently as master nodes, slave nodes and user nodes, who performed different roles for the computation and backup, and might change their roles upon peer failure. The simulation results showed that P2P-MapReduce outperformed MapReduce in the centralized master/worker model, in terms of reliability and scalability with a large scale of network of 40,000 nodes and a small failure rate of up to 0.4%, measured by the number of failed jobs and the amount of data exchanged for the maintenance of the peer network.

The idea of [3] was similar to MOON in that it exploited peers to store and transfer data files so as to reduce the overhead on the central servers. In structure, it was a hybrid of SCOLARS (a BOINC based master/worker model) and BitTorrent (a P2P data share overlay) and used replication of input, intermediate and output data files to improve reliability. Comparing with BOINC and SCOLARS, the evaluation results showed that the proposed model was more efficient for data distribution and more scalable with varying file sizes and available bandwidths. The shortcoming of the evaluation was a small grid of 92 nodes without churn.

Based on the above review in this section, whether VC is scalable for big data applications remains open and makes the research of this paper necessary.

3 Modelling Volunteer Computing for MapReduce

The underlying P2P VC overlay to model MapReduce is the work of [12, 13], where a pure P2P is built on the Chord DHT protocol [18]. In the model, a central point is maintained for only advertising available VC projects and providing a bootstrap pool for a volunteer to join a project. When a project is of the interest of a newer coming volunteer, the volunteer will join the Chord overlay from one of the bootstrap nodes. Once joined the overlay, the peer keeps searching for and then doing a task. A peer can leave the overlay when there is no more tasks; a peer can leave or crash at any time even when doing a task. When a peer leaves, the unfinished task is check-pointed and will be picked up by another peer. However the unfinished task of a crashed peer must be totally redone by another peer. Built on Chord DHT protocol, the VC model takes the advantage of Chord in terms of the proved reliability, scalability and performance. In this paper, the remodeling of VC for MapReduce is based on the above fundamental overlay by adding newer components and coordination as illustrated in Fig. 1 and detailed in the following sections.

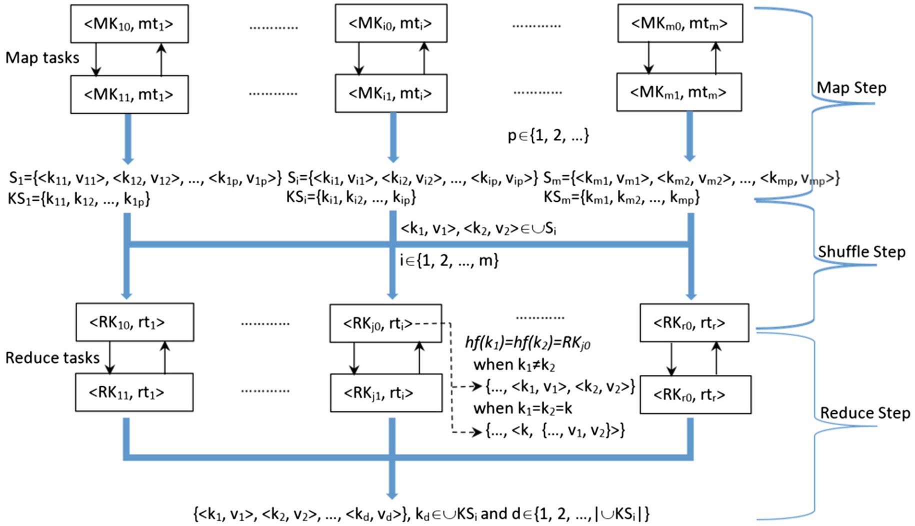

Fig. 1. The coordination of peers for MapReduce

3.1 The Map Step

The global object $<MK, m>$ represents the number of map tasks *m* that can be accessed by the key *MK*, which is known to every peer. Initially the *m* map tasks are: $<MK_{10}, mt_1>$, $<MK_{20}, mt_2>$, …, $<MK_{m0}, mt_m>$, where *0* manes the task is available to download

for execution by a peer. Once a map task $<MK_{i0}, mt_i>$ is put into execution by the peer, the task will change to $<MK_{i1}, mt_i>$, where *1* means it is now in execution and where $i \in \{1, 2, ..., m\}$.

Each peer looks up $<MK_{i0}, mt_i>$, where $i \in \{1, 2, ..., m\}$, for an available map task. If $<MK_{i0}, mt_i>$ is available, the peer changes it to $<MK_{i1}, mt_i>$ and then put it into execution. If the peer leaves before finishing the task, it will change it back to $<MK_{i0}, mt_i>$. When a map task is in execution, the peer will need to re-timestamp the task in a regular time interval *ui*. The *ui* is a predefined parameter, which can be accessed by the key *UI* (via $<UI, ui>$ on the overlay) that is known to every peer. It is worth to note that in a real implementation, it is unnecessary to re-timestamp the real task object mt_i. Instead, a simple timestamp mts_i is accessed by the key MK_{i11}. To check or update the state of a map task, a peer retrieves $<MK_{i11}, mts_i>$ more efficiently than retrieving the real task object mt_i. To access the real task object mt_i, a peer retrieves $<MK_{i1}, mt_i>$.

If failed with looking up $<MK_{i0}, mt_i>$, each peer looks up $<MK_{i1}, mt_i>$, where $i \in \{1, 2, ..., m\}$, for an available map task that satisfies the condition: (the current time - the timestamp mts_i of $mt_i) > ui$. Such a map task was in execution by a peer that is treated as crashed already.

A map task mt_i, where $i \in \{1, 2, ..., m\}$, is a self-satisfied object, which includes the executable code and data that are encapsulated in the data structures that are appropriate to a particular MapReduce application. A method call on mt_i such as $mt_i.execute()$ will perform the map task and emit *p* key-value pairs $S_i = \{<k_{i1}, v_{i1}>, <k_{i2}, v_{i2}>, ..., <k_{ip}, v_{ip}>\}$, where *p* is at least *1* in the general sense of MapReduce as defined by [7].

3.2 The Shuffle Step

The global object $<RK, r>$ represents the number of reduce tasks *r* that can be accessed by the key *RK*, which is known to every peer. Initially the *r* reduce tasks are: $<RK_{10}, rt_1>, <RK_{20}, rt_2>, ..., <RK_{r0}, rt_r>$, where *0* manes that the task is available to download for execution by a peer. A reduce task rt_j, where $j \in \{1, 2, ..., r\}$, is a self-satisfied object, which includes the executable code but its data set is initially empty and will be filled up by this shuffle step. The data structure of rt_j is in principle similar to a hash table, which will chain the values when keys clash. As a consequence, the data structure of rt_j will be treated as a hash table in the following description without losing generality.

For the *p* key-value pairs $S_i = \{<k_{i1}, v_{i1}>, <k_{i2}, v_{i2}>, ..., <k_{ip}, v_{ip}>\}$ that are emitted from the execution of a map task mt_i, where $i \in \{1, 2, ..., m\}$, we assume $KS_i = \{k_{i1}, k_{i2}, ..., k_{ip}\}$ and there is a hash function *hf*. For any $k \in \cup KS_i$, $hf(k) = RK_{j0}$, where $j = \{1, 2, ..., r\}$. For $<k_1, v_1>$ and $<k_2, v_2> \in \cup S_i$, we assume $hf(k_1) = hf(k_2) = RK_{j0}$ and thus $<RK_{j0}, rt_j>$ is retrieved. Under such a situation, if $k_1 \neq k_2$, $rt_j.put(<k_1, v_1>)$ and $rt_j.put(<k_2, v_2>)$ will store data v_1 and v_2 for the 2 different keys k_1 and k_2. If $k_1 = k_2$, $rt_j.put(<k_1, v_1>)$ and $rt_j.put(<k_2, v_2>)$ will store data v_1 and v_2 by chaining them for key k_1 or k_2. Then the reduce task $<RK_{j0}, rt_j>$ is stored back to the DHT ring. In this design, when the mapping step finishes, the shuffle step finishes in principle.

3.3 The Reduce Step

Each peer looks up $<RK_{j0}, rt_j>$, where $j = \{1, 2, ..., r\}$, for an available reduce task. If $<RK_{j0}, rt_j>$ is available, the peer changes it to $<RK_{j1}, rt_j>$ and then put it into execution. If the peer leaves before finishing the task, it will change it back to $<RK_{j0}, rt_j>$. When a reduce task is in execution, the peer will need to re-timestamp the task in a regular time interval *ui*, which was defined in Sect. 3.1.

If failed with looking up $<RK_{j0}, rt_j>$, each peer looks up $<RK_{j1}, rt_j>$, where $j \in \{1, 2, ..., r\}$, for an available reduce task that satisfies the condition: (the current time - the timestamp rts_j of rt_j) $> ui$. Such a reduce task was in execution by a peer that is treated as crashed already.

A method call on rt_j such as $rt_j.execute()$ will perform the reduce task and emit at least *1* key-value pair. When a peer emits a $<k, v>$, where $k \in \cup KS_i$, where $i \in \{1, 2, ..., m\}$, it will simply store it back to the DHT ring.

4 The Experimental Environment

We have implemented 2 versions of the proposed VC model for MapReduce. One is distributed version that has been implemented on the Open Chord platform [10]. This version of prototype implementation is to verify the functional correctness of the proposed model. In our experiments, the prototype was functionally correct for a 5-machines overlay, which was connected by a high speed Ethernet. This part is omitted from this paper as it is not the focus of scalability. The other version is a simulator, aiming at the evaluation of scalability of the model. When a large number of volunteer machines is unavailable to access, this version of simulator is able to evaluate the model on any numbers of virtual volunteer and map or reduce task. The compute load of tasks, the compute capacity and network speed of volunteers are all allowed to set for evaluation as detailed in this section.

Instead of using any real MapReduce applications, a virtual MapReduce job is proposed to generalize any MapReduce jobs as long as they comply with the programming paradigm as defined by [7]. In other words, this virtual job is able to demonstrate the generality of the simulation for any MapReduce jobs in the volunteer computing environments. Thus the runtime behaviors of the job execution could be applied to a wide range of MapReduce applications.

There is no restriction for the number of map tasks, reduce tasks or the number of peers involved in a job. There is no restriction for the number of peers that commit churn and when they commit churn. There is no restriction on the compute load of a map or reduce task.

A peer needs a *search time* to look for an available map or reduce task. The *standard step* is used as a time unit to simulate a real world time unit such as a second, minute, hour or day etc. For example, if the search time is set as 100, it means that a peer needs 100 standard steps to find an available task or make sure that no any available tasks.

A peer's *compute capacity* is the relative compute speed in standard steps. If a peer's capacity is 1, it can perform a standard step in one step. However, if a peer's

capacity is ½, it needs 2 steps to perform a standard step. That is, the latter is 2 times slower that the former.

Each map or reduce task has a *compute load*, which is defined by standard steps as well. For example, if a map or reduce task's computing load is 1,000 steps, a peer of capacity of 1 will need 1,000 steps to finish the task, but a peer of capacity of ½ will need 2,000 steps to finish the task.

Each map or reduce task has a *download time* and an *upload time*, which are defined by standard steps as well. The download time simulates the network capacity that a peer accesses the network to download a map or reduce task, and similarly the upload time is the time to upload the result of a map or reduce task.

As illustrated in Fig. 2, the normal workflow for a peer to do a map or reduce task consists of 4 time slots: *search*, *download*, *compute* and *upload*. A peer can leave or crash at any time slot.

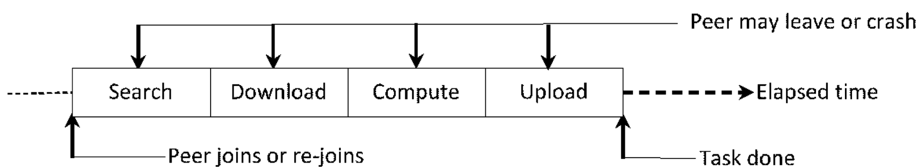

Fig. 2. The workflow of a peer in doing a map or reduce task

As our model is built on Chord, a safe assumption for the search time is naturally based on the proved efficiency of Chord lookup services, looking up a map or reduce task in *O(logN)* messages [18], where *N* is the number of peers. If we assume that each message needs *3* standard steps (e.g. *3* s) for safety, the search time for a *10,000* peers overlay will be $3x\log(10{,}000) \approx 40$ when these messages are serially processed. The real situation could be better when these messages are somehow processed in parallel.

A safe assumption for the download time and upload time is based the use of ADSL2$^+$ as the volunteer internet connection, which is neither a high end nor a low end internet plan that can be possessed by most of the home internet users. The download speed of ADSL2$^+$ is about 24 Mbps (3 MB/s) and the upload speed is 1.4 Mbps (0.18 MB/s).

A safe assumption for the size of a map or reduce task or its result is 50 MB, which needs 50 MB ÷ 3 MB/s = 16.7 s to download a task and 50 MB ÷ 0.18 MB/s = 278 s to upload a result. Thus a safe assumption of download time is 17 steps and of upload time is 300 steps, provided the result of a map or reduce task is not expended or shrunk. If we assume that the entire job consists of 200,000 such size of map or reduce tasks, there will be 200,000 × 50 MB = 10 TB data set to be processed.

5 The Scalability Evaluation

In this section, the scalability of the proposed VC model in performing MapReduce tasks will be evaluated from 3 aspects when peers commit churn or have heterogeneous communication cost or compute capacity.

5.1 The Scalability Against Churn

The evaluation scenario is set as in Table 1, where the search time, download or upload time and the compute load are all set as standard steps. Particularly, the download and upload time are set as 17 and 300, corresponding to ADSL2$^+$ bandwidth minimum standards of 24 Mbps and 1.4 Mbps for downloading and uploading 50 MB data. The peers join the overlay in 20 (randomly chosen) standard steps interval, while peers leave or crash in 40 (randomly chosen) standard steps of interval. The numbers of peer are set as from 2,000 to 10,000, of which half of them have the compute capacity of 1, the other half have the capacity of ½, and the average capacity is 0.75. To evaluate the scalability against churn, the churn rate is set as from 10% to 90% of the total number of peers. For example, for 10,000 peers, the evaluation will be performed when there are 1,000 (10%); 3,000 (30%); 5,000 (50%); 7,000 (70%) and 9,000 (90%) peers to leave or crash respectively. For the churn peers, half leaves and the other half crashes. The leave or crash peers are distributed from the middle backward and forward. For example, if the number of peers is 10,000 and the churn rate is 50%, the middle position is P_{5000}, the first leave or crash peer will be P_{2500}, and the last leave or crash peer will be P_{7499}. Peers start to leave or crash when half (randomly chosen) of the total peers have joined. The evaluation will be measured by speedup, which is:

$$\frac{\text{the total standard steps to complete the entire job by a single peer of the average of capacity}}{\text{the total standard steps to complete the entire job by the overlay of peers}}.$$

The evaluation results in terms of speedup are showed in Fig. 3. The following 2 observations support the system scalability against churn in the dynamic, opportunistic VC environments.

1. At the same churn rate, the more peers the system has, the faster the overall computation is.
2. At the same peer number, the smaller the churn rate is, the faster the overall computation is.

Table 1. The experimental setting to evaluate scalability against churn

Scenario variable	Value
The number of map tasks	200,000
The number of reduce tasks	200,000
The compute load of a map or reduce task	8,000
The search time for a map or reduce task	40
The download time for a map or reduce task	17
The upload time for a map or reduce result	300
The number of peers	2,000; 4,000; 6,000; 8,000; 10,000
The compute capacity	50% of peers: 1; 50% of peers: ½
The churn rate	10%; 30%; 50%; 70%; 90%
The peer join interval	20
The peer leave or crash interval	40

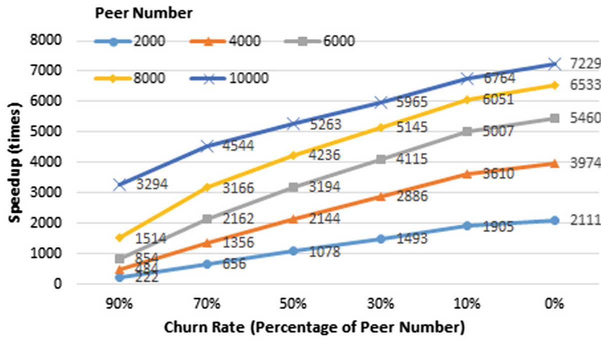

Fig. 3. The speedup against churn

There is 1 more observation about the speedup difference between neighbor churn rates as showed in Fig. 4, where a bigger churn rate affects the speedup more significantly than a smaller churn rate does.

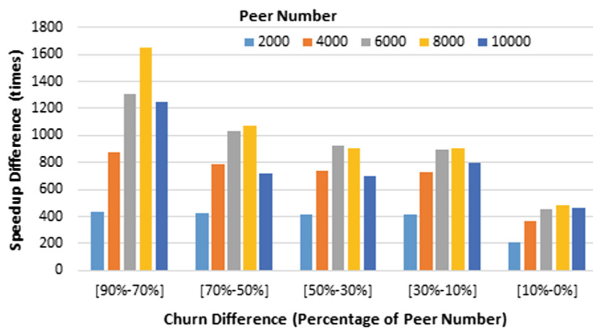

Fig. 4. The speedup difference against churn difference

5.2 The Scalability Against Communication Cost

The scenario setting of this evaluation is the same as those in Table 1 except that the churn rate is fixed as 30% and the download and upload time are set as in Table 2 to reflect the communication cost for the commonly available network bandwidth. The download and upload speed are chosen for the minimum standard of the service in each bandwidth. As showed in Table 2, the fastest connection is 75 (300/4) times (download) and 160 (800/5) times (upload) of the slowest connection.

The evaluation results in terms of speedup are showed in Fig. 5. The following 2 observations support the system scalability against the heterogeneous bandwidth of the commonly available internet connections for volunteers in the dynamic, opportunistic VC environments.

Table 2. The bandwidth setting to evaluate scalability against communication cost

	Connection speed in Mbps (MB/s)		Time for 50 MB data in seconds		Simulation setting in standard steps	
	Download	Upload	Download	Upload	Download	Upload
ADSL	1.5 (0.19)	0.5 (0.06)	266.7	800	300	800
ADSL2	12 (1.5)	1.3 (0.16)	33.3	307.7	35	300
ADSL2$^+$	24 (3)	1.4 (0.18)	16.7	285.7	17	300
NBN	50 (6.3)	20 (2.5)	8	20	8	20
Ethernet	95 (11.9)	82 (10.2)	4.2	4.9	4	5

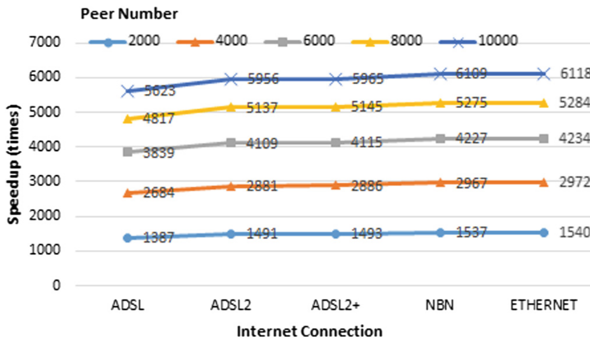

Fig. 5. The speedup against communication cost

1. At the same bandwidth, the more peers the system has, the faster the overall computation is.
2. At the same peer number, the larger the bandwidth is, the faster the overall computation is.

There is 1 more observation about the speedup difference between bandwidth differences as showed in Fig. 6, where the big speedup difference happened between the big bandwidth difference between ADSL and ADSL2 or between ADSL2$^+$ and NBN (National Broadband Network of Australia). That is, when bandwidth goes a big

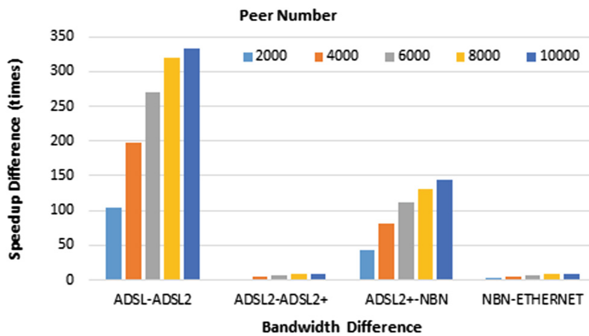

Fig. 6. The speedup difference against bandwidth difference

jump, the speedup does in the same way. There is no too much speedup difference happened between ADSL2 and ADSL2$^+$ or between NBN and Ethernet.

5.3 The Scalability Against the Heterogeneity of Compute Capacity

The scenario setting of this evaluation is the same as those in Table 1 except that the churn rate is fixed as 30% and the compute capacities are set as in Table 3 to reflect the heterogeneity of compute capacity of peers.

The evaluation results in terms of speedup are showed in Fig. 7. The following 2 observations support the system scalability against the heterogeneity of compute capacity of volunteers in the dynamic, opportunistic VC environments.

1. At the same capacity variation, the more peers the system has, the faster the overall computation is.
2. At the same peer number, the smaller the capacity varies, the faster the overall computation is.

There is 1 more observation about the speedup differences between the capacity differences as showed in Fig. 8, where, the more the capacity varies between peers, the more the speedup drops (or the slower the overall computation is).

Table 3. The setting to evaluate scalability against the heterogeneous compute capacity

Setting	Compute capacity variation
C1	All peers: 1
C2	1/2 peers: 1, 1/2 peers: 1/2
C3	1/3 peers: 1, 1/3 peers: 1/2, 1/3 peers: 1/3
C4	1/4 peers: 1, 1/4 peers: 1/2, 1/4 peers: 1/3, 1/4 peers: 1/4
C5	1/5 peers: 1, 1/5 peers: 1/2, 1/5 peers: 1/3, 1/5 peers: 1/4, 1/5 peers: 1/5
C6	1/6 peers: 1, 1/6 peers: 1/2, 1/6 peers: 1/3, 1/6 peers: 1/4, 1/6 peers: 1/5, 1/6 peers: 1/6

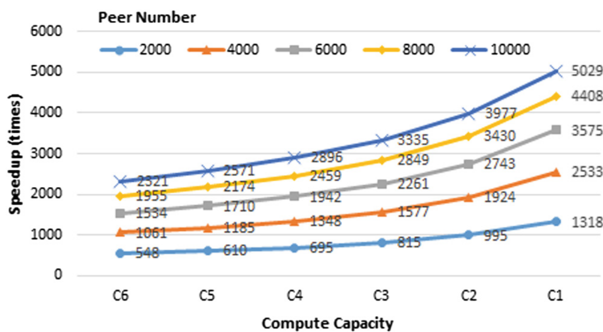

Fig. 7. The speedup against heterogeneous compute capacity

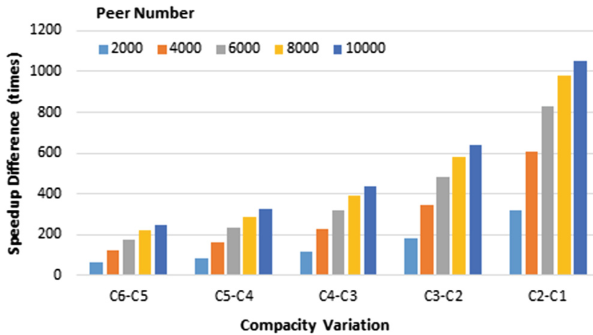

Fig. 8. The speedup difference against compute capacity variation

6 Conclusion

A DHT-based volunteer computing model is proposed for performing MapReduce big data applications and verified functionally correct by a prototype implementation on Chord protocol and Open Chord APIs. To evaluate the scalability of the model in the opportunistic volunteer computing environments, where a large number of peers have heterogeneous compute and network resources and commit churn, a simulator of the model has been implemented to test for an overlay of a large number of peers. The experimental evaluations showed that VC scales for a large number (up to 10,000) of peers, which commit high churn rate (up to 90%), have heterogeneous compute capacity (the fastest is 6 times of the slowest) and rely on different bandwidth (from the slow ADSL to high speed Ethernet). The results from this paper have confirmed that VC is suitable for big data applications like MapReduce when the model is built properly and the data splitting is appropriate. The future work will include the optimization of the simulator to verify the scalability of VC for even larger number of peers, who may commit more discrete churn.

References

1. Ahmad, F., Lee, S., Thottethodi, M., Vijaykumar, T.N.: MapReduce with communication overlap (MaRCO). J. Parallel Distrib. Comput. **73**(5), 608–620 (2013)
2. Afrati, F., Dolev, S., Sharma, S., Ullman, J.D.: Meta-MapReduce: a technique for reducing communication in MapReduce computations (2015). arXiv preprint arXiv:1508.01171
3. Bruno, R., Ferreira, P.: FreeCycles: efficient data distribution for volunteer computing. In: Proceedings of the Fourth International Workshop on Cloud Data and Platforms (2014)
4. Climateprediction.net (2016). http://www.climateprediction.net
5. Costa, F., Veiga, L., Ferreira, P.: Internet-scale support for map-reduce processing. J. Internet Serv. Appl. **4**, 18 (2013)
6. Costa, F., Silva, L., Dahlin, M.: Volunteer cloud computing: MapReduce over the Internet. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum (IPDPSW), pp. 1855–1862 (2011)

7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. FiND@Home (2016). http://findah.ucd.ie
9. Hadoop (2014). https://wiki.apache.org/hadoop/ProjectDescription
10. Kaffille, S., Loesing, K.: Open Chord Version 1.0. 4 User's Manual. The University of Bamberg, Germany (2007)
11. Korpela, E.J.: SETI@home, BOINC, and volunteer distributed computing. Annu. Rev. Earth Planet. Sci. **40**, 69–87 (2012)
12. Li, W., Franzinelli, E.: Decentralizing volunteer computing coordination. In: Che, W., et al. (eds.) ICYCSEE 2016. CCIS, vol. 623, pp. 299–313. Springer, Singapore (2016). doi:10.1007/978-981-10-2053-7_27
13. Li, W., Guo, W., Franzinelli, E.: Achieving dynamic workload balancing for P2P volunteer computing. In: Proceedings of the 44th International Conference on Parallel Processing Workshops (ICPPW), pp. 240–249 (2015)
14. Lin, H., Ma, X., Archuleta, J., Feng, W.C., Gardner, M., Zhang, Z.: Moon: MapReduce on opportunistic environments. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 95–106 (2010)
15. Marozzo, F., Talia, D., Trunfio, P.: P2P-MapReduce: parallel data processing in dynamic cloud environments. J. Comput. Syst. Sci. **78**(5), 1382–1402 (2012)
16. Oracle: An Enterprise Architect's Guide to Big Data - Reference Architecture Overview. Oracle Enterprise Architecture White Paper (2016)
17. Sarmenta, L.: Volunteer Computing. Ph.D., thesis, Massachusetts Institute of Technology (2001)
18. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Trans. Netw. (TON) **11**(1), 17–32 (2003)