View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AIS Electronic Library (AISeL)

Bonus Computing: An Evolution from and a Supplement to Volunteer Computing

Zheng Li *imlizheng@gmail.com*
Department of Computer Science, University of Concepción
Concepción, Chile

Yiqun Chen *yiqun.c@unimelb.edu.au*
Centre for Spatial Data Infrastructures & Land Administration, University of Melbourne
Melbourne, Australia

María Andrea Rodríguez *andrea@udec.cl*
Department of Computer Science, University of Concepción
Concepción, Chile

Lu Deng *mf1632014@smail.nju.edu.cn*
Software Institute, Nanjing University
Nanjing, China

Abstract

Despite the huge success in various worldwide projects, volunteer computing also suffers from the possible lack of computing resources (one volunteered device can join one project at a time) and from the uncertain job interruptions (the volunteered device can crash or disconnect from the Internet at any time). To relieve the challenges faced by volunteer computing, we have proposed bonus computing that exploits the free quotas of public Cloud resources particularly to deal with problems composed of fine-grained, short-running, and compute-intensive tasks. In addition to explaining the loosely-coupled functional architecture and six architectural patterns of bonus computing in this paper, we also employ the Monte-Carlo approximation of Pi (π) as a use case demonstration both to facilitate understanding and to help validate its functioning mechanism. The results exhibit not only effectiveness but also multiple advantages of bonus computing, which makes it a valuable evolution from and supplement to volunteer computing.
Keywords: Bonus Computing, Master-worker Model, Parasitic Computing, Recipient-broker-contributor Model, Volunteer Computing

1. Introduction

Among the variations of metacomputing that integrates distributed computing resources to fulfill parallel processing jobs, volunteer computing has been successful in various large projects and scientific collaborations ranging from discovering new stars [3] to searching Mersenne prime [12]. However, it is also known that volunteer computing may suffer from the shortage of computing resources and even from uncertain interruptions. Firstly, it has been identified that “the number of people participating in volunteer computing compared to the number of users on the Internet is insignificantly small” [2]. Secondly, the volunteered device can be dedicated only to a single project at a time. Considering that multiple donation instances will result in conflicting judgements about idle resources, it is impractical to install multiple clients on the same computer for different volunteer computing projects. Thirdly, the volunteered devices can be out of usage at any time due to unexpected crashes, network disconnections, or intentional leaving. Thus, it is valuable and beneficial to explore and exploit new resources for volunteer computing.

Given the booming of Cloud computing that is supposed to provide always-on or at least extremely-high-availability computational utility [15], we have observed an increasing amount of free-tier opportunities offered in the public Cloud market. Unfortunately, it is hard to directly utilize those free-of-charge resources in volunteer computing projects because of their infrastructural difference, limited capacity and strict conditions regulated by Cloud providers. Further inspired by parasitic computing [1] that is based on surprisingly trivial computing power across the Internet, we proposed a new computing paradigm, namely bonus computing, to efficiently take advantage of the free quotas of Cloud resources and to deal with metacomputing problems differing from the volunteer-computing scenario. In specific, bonus computing is equipped with a loose-coupling functional architecture including three roles, i.e. recipient, broker, and contributor. The recipient follows the Divide-and-Conquer (D&C) strategy and uses asynchronous mechanisms to break a whole job into tasks and receive task results. The contributor employs lightweight service technologies (e.g., Function-as-a-Service, microservice, and RESTful Web service) to donate his/her free quota of Cloud resources. Note that, since the service calls can be made independently, it is possible to deploy different project services on the same contributor's Cloud resource. The broker maintains a resource & service registry to facilitate scheduling and coordinating recipient tasks with contributor services. As such, bonus computing eventually becomes a standalone evolution from, and a valuable supplement to, volunteer computing.

This paper clarifies bonus computing and particularly explains its functional architecture together with six architectural patterns. To help understand and validate the functioning mechanism of bonus computing, we also implemented the Monte-Carlo approximation of Pi (π) as a use case demonstration, and conducted performance evaluations. The implementation and evaluation confirm not only the effectiveness (e.g., functioning and shorter job latency) but also other benefits (e.g., flexible and lightweight technology stack) of bonus computing. Accordingly, this work makes two main contributions. Firstly, this new paradigm enriches computing power for scientific collaborations in the traditional metacomputing community. Secondly, the constraints of contributor capacity in bonus computing in turn creates research opportunities on characterizing problems to match distributed solutions with fine-grained, short-running, and compute-intensive tasks.

2. Bonus Computing

2.1. Definition and Justification

As mentioned previously, we define bonus computing as a new metacomputing form that satisfies computation demands by taking advantage of the free computing power from the public Cloud market. The inspiration behind our bonus computing comes both from parasitic computing that exploits the trivial while pervasive Internet resources [1] and from volunteer computing that coordinates the donated idle computing power [17]. Firstly, although the free Cloud resources can be mediocre and even trivial, it is still possible to accumulate them to make many a little into a mickle. Secondly, from the perspective of potential customers and even non-customers, the free Cloud resources are always idle and cost nothing to be volunteered.

In fact, driven by business competitions, many Cloud vendors have provided free resources for various reasons ranging from attracting more consumers to increasing market reputations. Take Google App Engine (GAE) as an example, users can always have free quotas of each type of resource without necessarily upgrading to billing accounts [10]. For instance, the default standard environment of GAE offers 9 basic instance-hours per day as well as 1 GB of data storage in total for free. Meanwhile, these free resources are still powerful enough to deal with real-world problems, e.g., supporting Geographic Information System (GIS) services [4]. In addition, new customers are further eligible to get a one-time free trial worth $300 credit for trying different Google Cloud products over 12 months [11]. Considering that the Cloud market is booming with an increasing amount of providers [15], it is feasible to explore and utilize Cloud free quotas as bonus computing resources in proper computation scenarios.

Since public Cloud resources are out of consumers' control and different free quotas may further have various specifications and regulations, resource characterization and performance evaluation are crucial prerequisites for bonus computing. We have conducted a set of empirical investigations into the fundamental characteristics of different Cloud resource (e.g., [14], [13]). For example, in GAE's standard environment, the maximum request timeout is limited to 60 seconds [5], while one continuous execution session can last around 10 minutes only [13]. Given those strict regulations and limitations, we claim that bonus computing is particularly suitable for fine-grained, short-running, and compute-intensive distributed tasks, which is one of the essential differences between bonus computing and the other metacomputing forms.

After all, without dedicated resources, bonus computing is not expected to store, transmit and manipulate large amounts of data at individual free-resource sites remotely. When it comes to the compute capacity only, the computing power of even idle resources in volunteer computing are still unlimited to afford heavyweight and long-running workloads (with fault-tolerance mechanisms), while the elementary function of free Cloud quotas are mainly aimed at temporary trials. On the other hand, the small granularity of bonus computing tasks eventually relieves the need of sophisticated fault-tolerance mechanisms, i.e. the cost of a task failure is negligible and can quickly be compensated by launching new tasks. More detailed comparisons between similar computing paradigms are specified in Section 4.1.

2.2. A Proof-of-Concept Architecture

The functional architecture of bonus computing is compatible with, while different to, the well-known master-worker model. As the name suggests, the master-worker model refers to a parallelism mechanism in which a Master node orchestrates a bunch of Worker nodes to address particular workloads under the D&C strategy. This parallelism mechanism has been proven successful in a wide range of distributed computing applications. Nevertheless, the master and the workers are always tightly-coupled and a worker's resources are generally exclusive to one master job at a time, although the worker is flexible to quit in some cases like in volunteering computing.

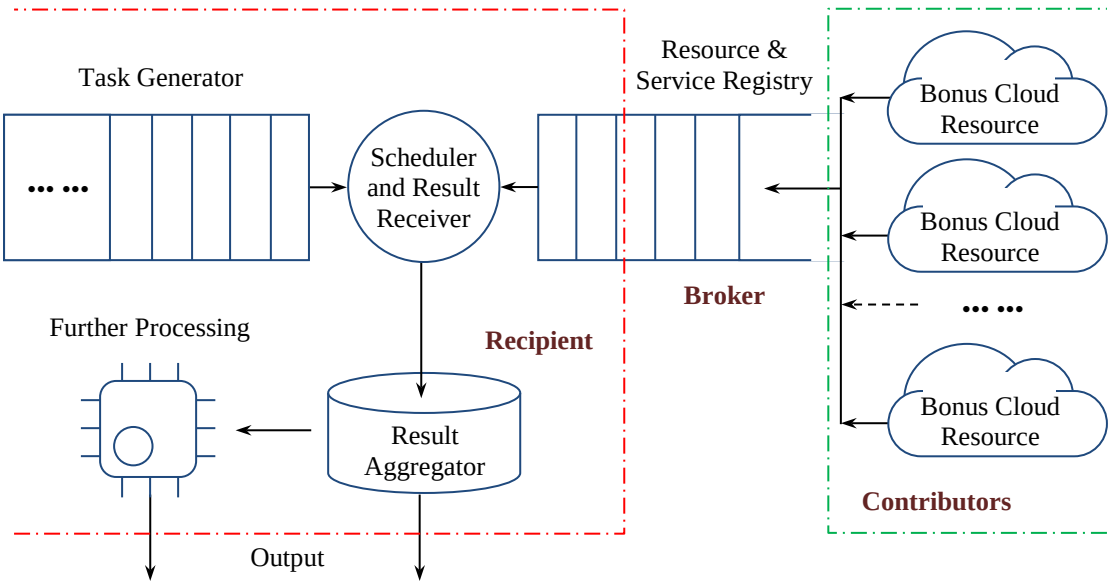


Fig. 1. Proof-of-concept architecture of bonus computing.

In contrast, driven by the service-oriented nature of Cloud resources, we emphasize the loose-coupling relationship between "master and workers" in bonus computing. The duty of a "worker" should be represented as a lightweight online service, such as a Function-as-a-Service, a microservice, or a RESTful Web service. The "master's orchestration of the workers" can then be realized by service calls. As such, the same Cloud resource can serve different

applications and/or multiple application instances simultaneously. To distinguish the loose coupling in bonus computing against the traditional master-worker model, we name our mechanism to be a recipient-broker-contributor model by naturally considering the free Cloud resource donators as contributors and treating the resource end-users as recipients. Based on our prototyping work, we illustrate the proof-of-concept architecture of bonus computing, as shown in Fig. 1.

As can be seen from the figure, the functional architecture of bonus computing essentially includes three roles, namely recipient, broker, and contributor. We list their main activities in bonus computing respectively as follows.

Contributor:

- Potential contributors can look up the resource registry to understand what free Cloud resources they actually have or are able to apply for.
- A contributor donates his/her free quotas of Cloud resources by deploying or migrating task-specific services initially published by recipients.
- By using standard APIs provided in the service registry, a contributor can deploy or migrate composite services that replicate the recipient-broker-contributor model. In other words, the composite services act as intermediate recipients and eventually construct a contribution tree of free Cloud resources. Note that this feature is another distinguishing characteristic of bonus computing.

Broker:

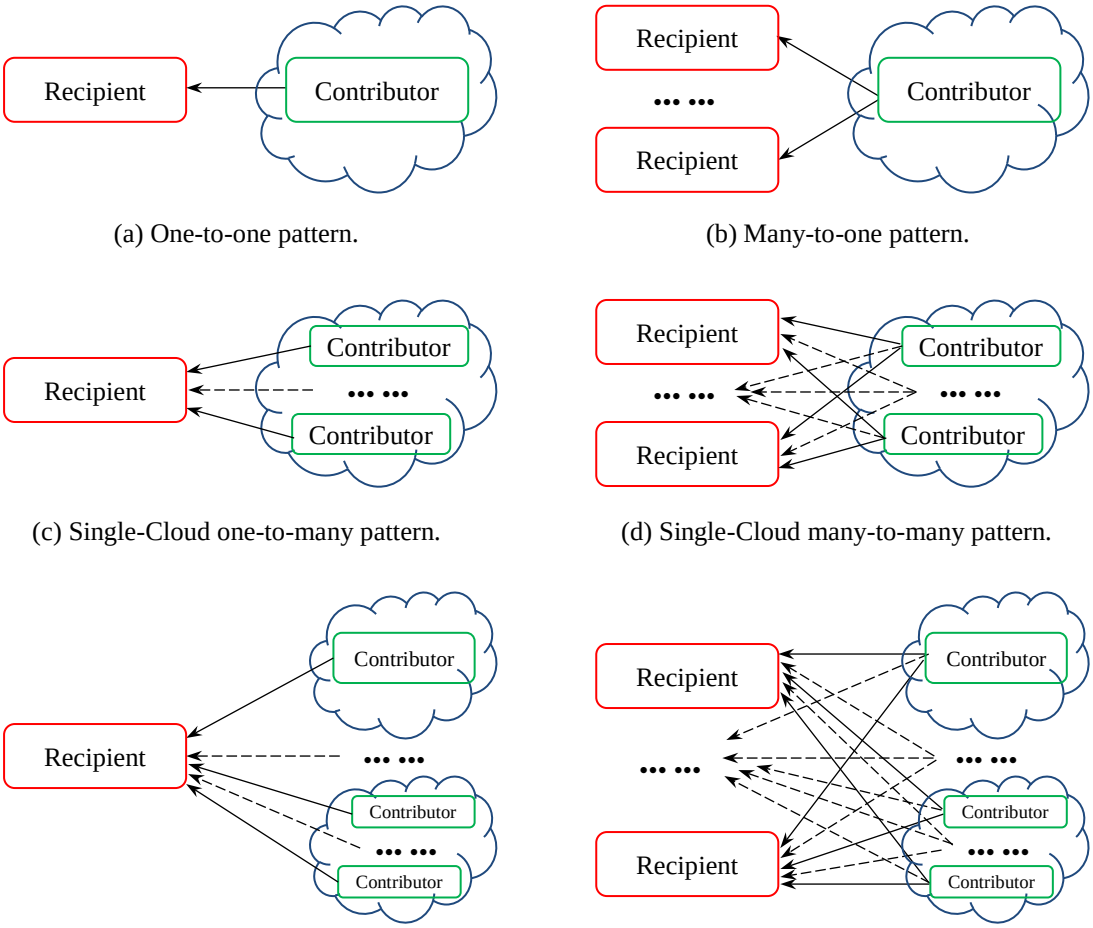
- The resource registry keeps exploring available resource opportunities in the Cloud market, stores and makes the information easy to find, and publishes call for contributors.
- The service registry facilitates contributors to register their task-specific services, and automatically discovers miss-registration services that also intend to donate computing resources.
- Ideally, the broker is acted by a third party in order to make the recipient-broker-contributor model further loosely-coupled. Considering the large motive of benefiting recipients themselves, the resource & service registry can be maintained by a third party together with recipients (or a recipient committee).

Recipient:

- To enable assigning tasks to contributors, the recipient needs to split a whole job into workload pieces in advance, or to generate tasks just in time.
- Given the candidate services from the registry, the recipient schedules the task assigning and correspondingly handles result receiving.
- The received results are aggregated, stored, and can be directly output if they are immediately ready to use.
- In most application scenarios, the aggregated results will further be processed before being delivered as the final output.
- It is noteworthy that the task generation and the result aggregation are with asynchronous mechanisms that enable the aforementioned activities to be conducted independently and simultaneously.

2.3. Six Architectural Patterns

Recall that, benefiting from the lightweight service technologies and the loose coupling among the three roles, one contributor can donate computing power to multiple recipients simultaneously in bonus computing. By focusing on the relationship between recipients and contributors only, we distinguish between six architectural patterns, as portrayed in Fig. 2.



(a) One-to-one pattern.

(b) Many-to-one pattern.

(c) Single-Cloud one-to-many pattern.

(d) Single-Cloud many-to-many pattern.

(e) Multi-Cloud one-to-many pattern.

(f) Multi-Cloud many-to-many pattern.

Fig. 2. Six architectural patterns of bonus computing.

Here we briefly explain these architectural patterns of bonus computing:

- **One-to-One** (cf. Fig. 2a) indicates a single pair of recipient and contributor, which is the elementary architectural pattern in bonus computing. Although one-to-one by itself is too simple to reflect the nature of bonus computing, it can still be a practical pattern in student projects or assignments.
- **Many-to-One** (cf. Fig. 2b) means that multiple recipients share the same computing resource from a single contributor. This pattern reveals one of the aforementioned features of bonus computing, i.e. being exposed as a service, the donated computing power does not have to be exclusive to a single recipient and a single application.
- **Single-Cloud One-to-Many** (cf. Fig. 2c) is a typical counterpart pattern against many-to-one. Since a Cloud provider offers independent free quotas to its individual users, one recipient can benefit from multiple contributors even in the same Cloud. Note that the different contributors generally belong to different Cloud user accounts.
- **Single-Cloud Many-to-Many** (cf. Fig. 2d) can be viewed as a combination of the many-to-one and the single-Cloud one-to-many patterns. It is clear that the services deployed by multiple contributors in the same Cloud can be freely consumed by multiple recipients.
- **Multi-Cloud One-to-Many** (cf. Fig. 2e) is a generalization of the single-Cloud one-to-many pattern. Given the increasing number of providers in the de facto Cloud market [15], it will be common for a recipient to take advantage of free resource quotas from different Cloud providers, as specified in Section 2.2.

- **Multi-Cloud Many-to-Many** (cf. Fig. 2f) is the most generic architectural pattern in bonus computing. Although our proof-of-concept architecture illustrated in Fig. 1 shows a single recipient only, it can be conveniently extended by including various bonus computing applications with different owners.

In practice, as mentioned previously, these pattern are further replicable by contributors to construct a resource contribution tree as the functional architecture of a bonus computing application.

3. Use Case Demonstration and Validation

To demonstrate and initially validate the effectiveness and efficiency of bonus computing, we decided to employ a random sampling problem to represent a broad class of easy-to-parallel applications that can normally be solved by Monte-Carlo approximation. By implementing the bonus-computing solution to this problem, we have seen both economic and performance advantages compared to other solutions based on our local resources, as explained in the following subsections.

3.1. Random Sampling Problem: Monte-Carlo Approximation of Pi (π)

A random sampling problem is to statistically select a subset of situations from a tremendous range of, or even infinite, possibilities to approximate the answer under a particular condition, which can be addressed by the Monte-Carlo method. Following the Monte-Carlo method, suppose the exact probability of a conditional situation is *P*, we can randomly generate situations, and the percentage of situations that satisfy the condition will approximately be equal to *P* if the amount of generated situations is large enough. Correspondingly, the workload of situation generation can conveniently be divided into any size of pieces to match our bonus computing scenario.

For the purpose of conciseness, we select the approximation of Pi (π) to be the demonstration in this paper, as visualized in Fig. 3. Given a square and its inscribed circle with the radius *r*, their area ratio can be calculated through Equation (1).

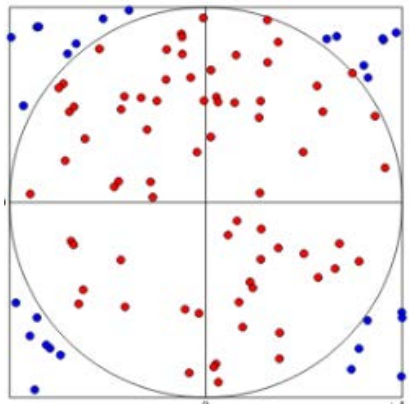


Fig. 3. Visualization of the Monte-Carlo approximation of Pi (π).

$$t = \frac{A_{circle}}{A_{square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \tag{1}$$

where A_{square} and A_{circle} represent the areas of the square and its inscribed circle respectively, while *t* indicates the area ratio. Consequently, Pi (π) can be obtained by $4 \times t$, i.e. $\pi = 4t$.

Then, imagine we blindly draw points inside the square, there must be some points drawn inside the circle and others not (cf. the red dots and the blue dots respectively in Fig. 3). When there are numerous points randomly spreading over the square, its area is able to be replaced with the amount of points, so is the circle's area. Thus, we will be able to use the ratio of point amounts to fulfill the Monte-Carlo approximation of Pi (π). The approximation effect in our

local experiments is exemplified in Fig. 4. It is clear that, the more points we draw (essentially generated by computer), the more accurate result we can obtain.

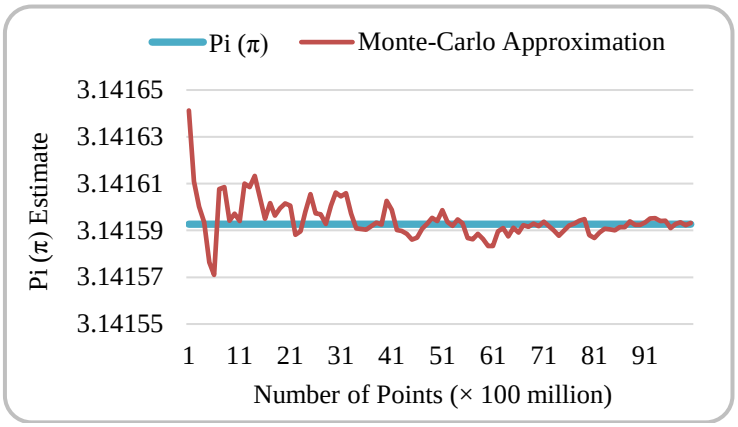


Fig. 4. Monte-Carlo approximation of Pi (π) from 100 million to 10 trillion points.

3.2. Implementation of the Bonus Computing Solution

Since this Monte-Carlo approximation of Pi (π) is a single application, we employ the multi-Cloud one-to-many architectural pattern (cf. Fig. 2e) to implement our bonus computing solution, i.e. we setup a single recipient in our local experimental environment and assign individual workload pieces to different contributors across multiple Clouds. When it comes to the multiple Clouds, in addition to GAE, we also employ the NeCTAR Cloud that is a research-oriented federated computational infrastructure for the Australian scholar community [14]. As for the multiple candidate contributors, we have asked our colleagues to help deploy a number of RESTful services on free quotas of GAE Java Runtime and of NeCTAR virtual machines respectively.[1] In particular, the Java function for contributors to share Cloud resources is shown below. Note that the contributors only "draw" random points and return the total number of inside-circle points, based on which the calculation of Pi (π) is eventually done by the recipient.

```
Static long MonteCarloPI(long N){
      double x, y;
      long sum = 0;
      for (long i = 0; i < N; i++){
            x = Math.random();
            y = Math.random();
            if ((x*x+y*y) <= 1)
                  sum++;
      }
      Return sum;
}
```

As for the processing function of recipient, we decided to run a whole job of Monte-Carlo approximation of Pi (π) and compare the job latency between bonus computing, multi-thread local computing, and single-thread local computing. According to our pre-experimental trials, our local computer (Intel^{R} Core^{TM} i5-3230M CPU @ 2.60GHz) can barely afford more than ten-thread 100-million-point approximation. Therefore, we set the workload size for each contributor to be "drawing and counting" 100 million points inside a square with the side length 1, and setting the number of contributors to be 10. Moreover, to facilitate "apple-to-apple" performance evaluation in this study, we use multiple threads to issue contribution requests and

[1] For example, one of the contributor prototypes on GAE for this experiment is https://vocal-plateau-201001.appspot.com/hello, which directly returns the number of points that fall inside the circle from the 100 million random-point generations.

collect results instead of asynchronously generating tasks and storing results in a database. Due to the space limit, we only show partial code segments below. Note that we particularly code in Python to further demonstrate the loose coupling between different roles in bonus computing.

```
PiList = []
global i=0, time_start, time_end
time_start=time.time()
t1 = threading.Thread(target=service1)
t1.start()
... ...
t11 = threading.Thread(target=pitimer)
t11.start()

def service1():
    global i
    PiList.append(int(request.get(RESTful API 1)))
    i = i + 1
... ...
def pitimer():
    global i, time_end
    while True:
        if i==10:
            break;
    mcPi = 4.0 * sum(PiList)/(i*N))
    time_end = time.time()
    mcElapse =  time_start – time_end
```

3.3. Performance Evaluation

To better understand the performance of bonus computing, we conducted evaluation experiments both from the individual task's perspective and from the whole job's perspective. By calling a single contributor service deployed on GAE, NeCTAR and local computer respectively and repeatedly, we measured the latency of the aforementioned task of "drawing and counting" 100 million points, as shown in Fig. 5.

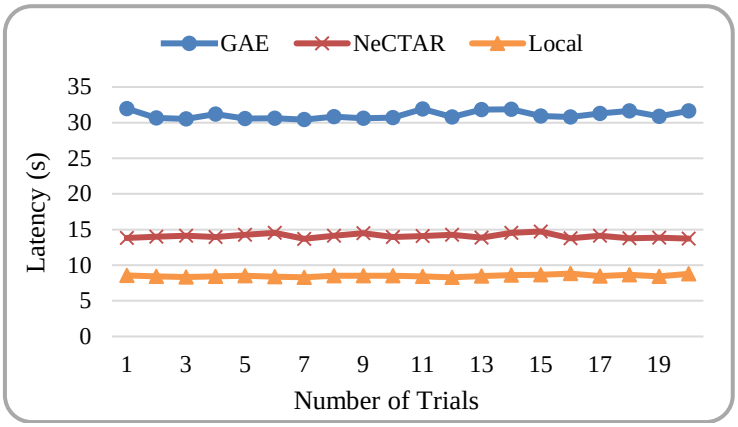


Fig. 5. Performance comparison among three resource types w.r.t a single contributor task.

It can be seen that out local computer has the most powerful computing resource, and finishing the task takes around 9 seconds only. The free tier of NeCTAR VM delivers close computing power that finishes the task within 15 seconds. In contrast, the task running on GAE Java Runtime suffers not only the longest latency but also the largest variability between 30 and 32 seconds.

When focusing on the whole job, we measured the latency of Monte-Carlo approximation of Pi (π) with 10×100 million points by bonus computing, single-thread local computing, and

multi-thread local computing respectively. According to the result shown in Fig. 6, it is clear that the single-thread solution on our local computer has the worst performance; the multi-thread solution benefits from the four logical CPU cores while still encounters resource competition among the ten threads; and although the individual Cloud contributor tasks run relatively slowly, bonus computing can finish the whole job as quickly as 37.75 seconds on average.

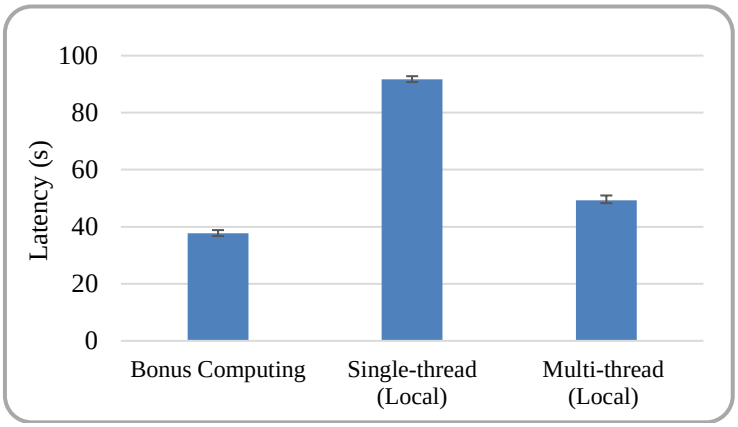


Fig. 6. Performance comparison between bonus and local computing w.r.t the whole job.

Recall that we did not implement the asynchronous task generation and result aggregation mechanisms in this study, in order to make sure that the recipient has exactly ten contributors (cf. Section 3.2). Consequently, the performance of bonus computing shown in Fig. 6 has been bottlenecked by the slowest contributor task. In practice, benefiting from the asynchronous mechanism (cf. Section 2.2), bonus computing can respond even faster and flexibly by retrieving and processing contributors' results from the aggregator at any time.

4. Related Work

Using Cloud resources to enhance volunteer computing has been investigated in the project Cloud@Home [6]. The authors observed the booming of Cloud computing and claimed that the barriers of volunteer computing can be knocked down by involving more flexible and general Cloud services. Compared to functional services in bonus computing, the Cloud services in the Cloud@Home paradigm are still considered to be dedicated computing resources and are supposed to share long-lasting computational tasks. Nevertheless, Cloud@Home can eventually turn out to be the paradigm of Cloud computing if its architecture is in a pure Cloud environment. More importantly, the pay-as-you-go nature of commercial Cloud services has violated the spirit of volunteer computing.

Since free-of-charge service samples emerged in the Cloud market, exploiting free Cloud resources has been studied in two types of work. The first one is to outsource particular functional components of a local application to the Cloud. For example, a map service for constructing GIS systems was successfully deployed on GAE [4]. However, this type of work does not intend to employ free Cloud resources to realize or apply any metacomputing scenario. The second type, in contrast, is trying to enjoy free computing in the Cloud to deal with metacomputing-alike problems. For example, a completely inside-Cloud implementation of the master-worker model was proposed in [16]. In fact, our bonus computing also belongs to this type of work. However, as explained previously, the master-worker model has generally been implemented as a tight-coupling architecture with respect to a single job. Moreover, the study [16] tried to exhaust the free quotas of different resource types including the GAE-specific ones like Task Queue, which makes their work hard to migrate between Clouds and correspondingly their mechanism becomes even more tightly coupled. On the contrary, our bonus computing tries to take advantage of free computational infrastructure in a generic sense and avoids sticking to provider-specific Cloud functionalities. In addition, we advocate lightweight service

forms (e.g., Function-as-a-Service, microservice, and RESTful Web service) for contributing free resources, which essentially enables our recipient-broker-contributor model to be loosely coupled, better scalable and high migratable.

To facilitate clarifying bonus computing, we further give a brief comparison between similar computing paradigms in the following subsection.

4.1. A Brief Comparison between Similar Computing Paradigms

We compare our bonus computing particularly with grid computing, parasitic computing and volunteer computing. Their typical advantages and disadvantages are highlighted in **Table 1.** Advantages and disadvantages of different comparable computing paradigms. To avoid duplication, we do not repeat the definition and explanations of bonus computing here.

Table 1. Advantages and disadvantages of different comparable computing paradigms.

	Advantages	Disadvantages
Grid Computing	• Affordable and convenient computing power. • Innovative ways of solving complex scientific problems.	• Complicated architecture. • Dedicated technology stack. • Inherent complexity without any centralized control.
Parasitic Computing	• Zero-effort from the unwitting contributors. • Pervasive compute resource over the Internet in theory.	• Ethical and legal concerns. • Computationally inefficient in practice. • Similar to denial-of-service attacks in the worst case.
Volunteer Computing	• Lightweight and flexible mechanism for anticipation. • Cost-wise. • Self-scalable with respect to the whole volunteer computing system.	• Frequent job interruptions. • Suitable fault-tolerance mechanisms are required. • Improper volunteering activities sometimes.
Bonus Computing	• Completely free from the contributor's perspective. • Lightweight architecture and flexible technology stack. • Sophisticated fault-tolerance mechanism is not needed.	• Double authorization by both contributors and Cloud providers. • Free quotas of Cloud resources need to be well characterized in advance.

Grid Computing

Grid computing makes world-widely distributed resources shareable and co-operable for high-performance scientific collaborations on an unprecedented scale across the Internet [8]. Since it used to be expensive and difficult to access high-performance computing resources, grid computing emerged to deliver affordable and convenient computing power at least as strong as what supercomputers and dedicated clusters could supply. Moreover, the cooperation among shared storage and compute resources leads to innovative ways of solving complex scientific problems, which has been demonstrated in various application scenarios ranging from distributed computing to large-scale data analysis [7].

Unlike supercomputers and individual clusters, however, the grid environment is not only geographically distributed but also infrastructural diverse. As a result, the implementation of grid computing requires a uniform middleware infrastructure to cater the uncertain resource heterogeneity. Furthermore, since the fundamental Internet and Web technologies are far from satisfying grid computing, dedicated mechanisms, techniques and tools are needed to address the various challenges like resource discovery, provisioning, scheduling and management [9]. For example, standard protocols and standard application programming interfaces (APIs) are particularly crucial in the grid architecture. Meanwhile, since grid computing does not rely on

any centralized control, such a decentralization nature makes it inherently more complex than cluster computing.

Parasitic Computing

Parasitic computing utilizes the standard protocols (e.g., TCP checksum function) to exploit the trivial while pervasive computing power from the Internet communication infrastructure [1]. Following the same D&C strategy, parasitic computing divides a complex computational problem into small pieces to be addressed merely by communicating with target computers over the Internet, while the target computers are unwittingly forced to perform computation without consents and permissions. As such, unlike the other similar computing paradigms, parasitic computing does not require the target computers to download and execute particular software to join a large-scale cluster for dealing with a distributed computation job.

Unfortunately, since the Internet protocols have a low computation-to-communication ratio due to their nature of message transferring, the implementation of parasitic computing is computationally inefficient in practice, at least at this current stage. More importantly, there are ethical and legal concerns about parasitic computing in terms of using remote hosts without any authorization. Although it is claimed to be harmless to the security of the target computers, parasitic computing can incur similar effects of denial-of-service attacks and will consequently cause delays and disruptions to Internet services.

Volunteer Computing

Volunteer computing exploits the idle processing power of computers contributed by general public and even casual owners across the Internet, to deal with scientific problems in a distributed and parallel way [17]. Compared to the other metacomputing forms, volunteer computing emphasizes the ease and flexibility of contributors' anticipation. The contributors do not have to be equipped with any knowledge of cooperation technologies in scientific computing. Meanwhile, volunteer computing has been considered to be cost-wise and self-scalable [2]. Firstly, by employing only idle resources from volunteers' computers that are already on and connected to the Internet for their own reasons, most costs for maintaining the computing power (e.g., cooling, electricity, and hardware) are essentially eliminated. Secondly, when more volunteers join the computing and/or upgrade their computers, the whole volunteer computing system correspondingly scales horizontally and/or vertically without extra effort from the coordinator's perspective.

On the other hand, the flexible anticipation mechanism can in turn incur frequent job interruptions in volunteer computing. In fact, the volunteered machines can be unexpectedly offline at any time due to power off or Internet disconnection. Considering that it is common to assign relatively heavy or long-running workloads to volunteers because their idle resources can still be powerful, volunteer computing usually requires fault-tolerance mechanisms (e.g., checkpointing) to reduce the waste of rework. In addition, cheating has been found in the existing practices of volunteer computing, for example, the client software can improperly be installed on public computers (e.g., in libraries at schools) that are not supposed to be volunteered [2].

5. Conclusions and Future Work

Motivated by addressing the barriers of volunteer computing and inspired by parasitic computing, we proposed bonus computing as a new paradigm in the metacomputing community. In particular, the loose-coupling recipient-broker-contributor model acts as one of the most significant characteristics that distinguishes bonus computing from the other metacomputing forms. Our use case demonstration of the Monte-Carlo approximation of Pi (π) has initially validated the effectiveness and efficiency of bonus computing.

On the other hand, although sharing the same genes of scientific collaborations, bonus computing is expected only to be a supplement to (instead of a replacement for) volunteer

computing. Compared to the latter, bonus computing also has shortcomings. For example, in addition to the constraints of contributors' capacity, the deployment of task-conquering services requires extra effort to go through double authorizations by both contributors and Cloud providers. Driven by these shortcomings, our future work will be unfolded along two directions. First, we will snowball bonus computing projects to better characterize suitable problems in this new paradigm. Second, we will gradually develop automated tools to facilitate deploying and/or migrating contributor services to different Cloud platforms.

References

1. Barabási, A.-L., Freeh, V.W., Jeong, H., Brockman, J.B.: Parasitic Computing. Nature. 412, 894-897 (2001)
2. Beberg, A.L., Ensign, D.L., Jayachandran, G., Khaliq, S., Pande, V.S.: Folding@home: Lessons From Eight Years of Volunteer Distributed Computing. In: Proc. 23rd Int. Symp. Parallel & Distributed Processing (IPDPS '09), pp. 1-8. IEEE, Rome, Italy (2009)
3. Betz, E.: Donated Computer Time Discovers New Star, https://www.insidescience.org/news/donated-computer-time-discovers-new-star. Accessed April 2, 2018
4. Blower, J.D.: GIS in the Cloud: Implementing a Web Map Service on Google App Engine. In Proc. 1st Int. Conf. Computing for Geospatial Research & Application (COM.Geo 2010), article no. 34. ACM, Washington, D.C., USA (2010)
5. Choosing an App Engine Environment, https://cloud.google.com/appengine/docs/the-appengine-environments. Accessed April 15, 2018
6. Cunsolo, V.D., Distefano, S., Puliafito, A., Scarpa, M.: Volunteer Computing and Desktop Cloud: The Cloud@Home Paradigm. In: Proc. 8th IEEE Int. Symp. Network Computing and Applications (NCA 2009), pp. 134-139, IEEE, Cambridge, MA, USA (2009)
7. Foster, I.: The Grid: A New Infrastructure for 21st Century Science. In: Berman, F., Fox, G., Hey, T. (eds.) Grid Computing: Making the Global Infrastructure a Reality, pp. 51-63. Wiley (2003)
8. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. J. High Perform. Comput. Appl. 15 (3), 200-222 (2001)
9. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud Computing and Grid Computing 360-Degree Compared. In: Proc. 2008 Grid Computing Environments Workshop (GCE '08), pp. 1-10. IEEE, Austin, USA (2008)
10. Google App Engine Quotas, https://cloud.google.com/appengine/quotas. Accessed April 15, 2018
11. Google Cloud Platform Free Tier, https://cloud.google.com/free/docs/frequently-asked-questions#free-tier. Accessed April 15, 2018
12. Great Internet Mersenne Prime Search, https://www.mersenne.org/. Accessed April 3, 2018
13. Li, Z., Guo, X., Zhang, H.: Do Google App Engine's Runtime Environments Perform Homogeneously? An Empirical Investigation. Submitted to the 25th Asia-Pacific Software Engineering Conference (APSEC 2018). IEEE, Nara, Japan (2018)
14. Li, Z., Ranjan, R., O'Brien, L., Zhang, H., Ali Babar, M., Zomaya, A.Y., Wang, L.: On the Communication Variability Analysis of the NeCTAR Research Cloud System. IEEE Syst. J. Early access, 1-12 (2016)
15. Li, Z., Zhang, H., O'Brien, L., Cai, R., Flint, S.: On Evaluating Commercial Cloud Services: A Systematic Review. J. Syst. Softw. 86 (9), 2371-2393 (2013)
16. Malawski, M., Kuźniar, M., Wójcik, P., Bubak, M.: How to Use Google App Engine for Free Computing. IEEE Internet Comput. 17 (1), 50-59 (2013)
17. Sarmenta, L.F.G., Hirano, S.: Bayanihan: Building and Studying Web-based Volunteer Computing Systems using Java. Future Gener. Comput. Syst. 15 (5-6), 675-686 (1999)