How to Use Google App Engine for Free Computing

Maciej Malawski[1,3], Maciej Kuźniar[1], Piotr Wójcik[1], Marian Bubak[1,2]
(1) AGH University of Science and Technology,
Department of Computer Science,
Mickiewicza 30, 30-059 Krakow, Poland
(2) University of Amsterdam, Institute for Informatics,
Science Park 904, 1098 XH Amsterdam, The Netherlands
(3) Center for Research Computing, University of Notre Dame,
Notre Dame, IN 46556, USA
Email: {malawski,bubak}@agh.edu.pl
{maciek.kuzniar,piotr.iwo.wojcik}@gmail.com

Abstract—The objective of research presented in this paper was to investigate if the Google App Engine cloud service may be used for free of charge execution of parameter study problems. Based on the provided Task Queue API, a simple and extensible framework implementing the master-worker model has been developed, which enables usage of the App Engine application servers as computational nodes as well as monitoring the task execution. The results of the feasibility study are presented and discussed, followed by the comparison with free cloud offering from Amazon EC2.

Index Terms—Distributed programming, Cloud computing, Economic and other policies

1 Introduction

The Internet has always been seen as a potentially unlimited source of computing power which will push the borders of problem sizes to be solved. The increasing demand for computing power may be satisfied with individual machines (from PC to HPC) connected to the Internet, by using clusters of workstations [1], grid e-infrastructures [2], volunteer computing facilities or by cloud computing providers [3], [4]. There are also unconventional ways, such as "parasitic computing" [5] which can force computers on the Internet to perform required calculations. Using such "free" resources obviously involves a trade-off between cost, reliability and ease of use. The parasitic computing proposed by Barabasi was based on usage of the TCP checksum function: specially constructed messages were sent to a number of web servers, and the packet with a code to be computed was inserted at IP level. The solution we propose avoids such a very low level programming and is based on usage of a functionality of clouds.

We focus on such computing problems which may be solved by partitioning a large computing job into multiple independent tasks which could be executed in parallel (so called parameter study problem). Many computing problems belong to this class, including Monte Carlo simulations in high energy physics or computational finance, virtual screening in computational chemistry or analysis of multiple combinations of genomical or proteomical data in bioinformatics, to name just a few [6]. Although parallelization of such applications is relatively easy, an appropriate computing system that would take advantage of their simple structure is still needed.

The objective of research presented in this paper was to investigate if the Google App Engine [7], which is a service for hosting Web applications, may be used for free of charge execution of compute intensive applications. With App Engine used in a conventional way it is difficult to use Google infrastructure to solve parameter study type problems. However, recently Google introduced a new experimental service: Task Queue, which allows web application developers to create sets of small tasks which can be executed asynchronously in the background. We show that it is possible to exploit the App Engine features and to use it to construct a free computing environment. Based on Task Queue API, provided recently by Google, we have developed a simple and extensible framework which enables usage of the Google App Engine servers as computational nodes to solve computationally intensive problems.

First, we give a short overview of the most important ways of obtaining access to the computing resources offered on the Internet. Then, we introduce the Google App Engine platform, its new Task Queue service, the free usage quotas and performance constraints. Subsequently, we describe the details of our computing frame-

work built on top of App Engine and present how to prepare the application to be executed using our framework. We present results of performance measurements and assess applicability of the framework and performance gain. Finally, we compare the performance of the free version of AppEngine with the paid one as well as with free cloud offering from Amazon EC2.

2 Related work

There are many ways of using the Internet to access computing resources for solving computationally demanding tasks. Individual machines (from PC to HPC) can be connected together to form simple clusters or even more complex and globally coordinated grid e-infrastructures [8]. These systems can benefit from the resource managers such as Condor which rely on queueing techniques for batch processing of large number of tasks. Another approach is the volunteer computing and such such platforms as BOINC known from SETI@Home and similar projects [9] where people donate free cycles of their computers or game consoles to solve some challenging problems.

Nowadays, cloud computing is foreseen as the mainstream means of obtaining simple and on-demand access to virtualized computing resources [10]. Although there are voices that cloud computing will never be free [11] there are examples which seem to contradict this opinion. We can quote here offers coming from Amazon EC2 which gives limited access to free resources to their users during the first year after sign-up, or Microsoft Azure and Google App Engine platforms which provide their resources free of charge if keeping the usage within the assigned quotas. More and more often scientists are granted free access to cloud computing resources, which can result in a wider access to computing power and drive innovation, as foreseen in [12].

The method presented in this paper is related to an idea of "parasitic computing" [5] which takes advantage of computing the checksums in network packets to force the computers on the Internet to perform interesting calculations. However, this approach was extremely inefficient, and moreover, it raised many ethical issues so it has been generally considered as illegal abuse of the Internet devices. In contrast to such parasitic methods invented in the early days of the Internet, utilization of currently available free computing quotas offered by cloud providers is legal and, as our results show, their peak efficiency goes beyond that of several modern PCs.

3 Google App Engine, task queue and free usage quotas

Google App Engine [7], [13] is a service for hosting Web applications. It is often presented as an example of platform-as-a-service cloud computing solution, since it can be used by Web application developers to create their applications using a set of tools provided by Google. In contrast to other cloud services, such as Amazon EC2, Google App Engine requires application developers to use only a limited set of supported programming languages, together with a small set of APIs and frameworks which are mostly dedicated to Web applications. The applications are executed in a secure hosting environment running on the computing infrastructure provided and managed by Google. The infrastructure offers load balancing and auto-scaling capabilities, i.e. it automatically adjusts number of application instances (virtual machines) to the request rate.

We focus on the recently introduced Task Queue which is particularly interesting for computational applications. It allows execution of small jobs asynchronously in the background of HTTP request processing. By dispatching tasks to multiple App Servers according to the current load, Task Queue becomes a perfect tool to provide scalability. However, there are several difficulties to be dealt with when using Task Queue and the execution environment provided by Google App Engine. Task execution must be idempotent, since Google guarantees that each task will be executed at least one time, which may lead in exceptional circumstances to re-execution of some of the tasks. The execution model does not ensure any kind of task concurrency control, and Google does not provide APIs for inter-task communication. It is impossible to specify when and on which Application Server a given task should be executed. The only way to provide data flow between tasks is to store and read data from the Bigtable database [14].

The important feature, which we have used in our research, is that Google App Engine may be used for free of charge execution of the applications, while respecting the resource usage quotas introduced by the framework. First of all, the execution time limit for each web request is 30 seconds and for tasks processed by the Task Queue the limit has been recently increased to 10 minutes. Consequently, all computations we intend to execute have to be divided into subtasks small enough to complete their execution in this time. Quotas that turned out to have the greatest impact on task execution speed are associated with available CPU cycles. The application may use up to 15 CPU minutes per minute and 6.5 CPU hours per day. In this case 1 CPU minute means the number of cycles that can be executed on a 1.2 GHz processor in 1 minute time. Other quotas are not so critical such as number of task invocations per second, number of database API calls or number of HTTP requests. For compute intensive applications they are high enough not to affect the performance. It is noteworthy that some of these quotas have been increased during last year which is a promising sign for the future.

Although Google App Engine is mainly dedicated to hosting web applications which handle many short (interactive) requests, there are use cases where longer background processing is required. In addition to Task Queue API, which we have used in our work, there are other experimental extensions recently announced:

Pipeline API[1] and MapReduce[2]. Pipeline API allows development of workflow-based applications, where larger computation is split into tasks (pipelines) which are connected using data dependencies. They can be used to represent not only processing tasks but also human interaction using Web requests or e.g. e-mail. Pipeline API is used to implement MapReduce [15] framework on top of App Engine. Users need to define map() function which will be iterated by the framework over entries in input dataset, which may include datastore objects, lines in blobstore or provided by a custom reader. The tasks in both Pipeline and MapReduce APIs follow the same constraints and quotas as the ones of Task Queue, which means that these APIs can also be used for free computing as our framework for parameter study applications.

4 A FRAMEWORK FOR PARAMETER STUDY PROBLEMS

4.1 Framework structure

The framework implements the master-worker model and its structure is presented in Fig. 1. Master and worker tasks are implemented as Java classes extending HttpServlet. Their execution is triggered upon HTTP request. Parameters are passed in the HTTP GET message. Information about tasks is stored in the distributed, object-oriented Bigtable database (App Engine DB), as the queue interface does not provide methods for task monitoring. Data concerning a single task consist of: input data, creation time, execution start time, execution finish time, task execution status and task execution result. Additionally, for master task a list of its worker tasks' IDs is stored. Every data entity stored in Bigtable database is identified by an unique ID.

The job execution scenario consists of the following steps which are presented in Fig. 1. The client uses a web application as the main interface to submit a job (1). The corresponding data is added to the database (2) and an unique job ID is returned to the user (3). This ID can be used to monitor the execution status of the job and to get results. Simultaneously, a master task is created by the web interface and executed (4). When the master task is executed, it splits the job data into given number of chunks and spawns corresponding number of worker tasks, which are added to the Task Queue (5). The execution of worker tasks is controlled by the queue engine (7) and the worker task results are stored in the database (8). Finally, when all partial results are ready, master task aggregates them and returns the job result to the user (11). Aggregation process involves fetching partial job results computed by worker tasks (9) from the Bigtable data storage system. Then, accordingly to problem nature, these results are merged to produce final job result (10). The task management, such as new task submission, monitoring and getting the results, is controlled by the web interface.

1. http://code.google.com/p/appengine-pipeline/
2. http://code.google.com/p/appengine-mapreduce/

4.2 Preparing the program to run on App Engine infrastructure

Adding a new type of task takes just a couple of simple steps. First, classes implementing the IMasterTask and IWorkerTask interfaces have to be programmed. There are two required methods in the Master:

- `execute()` – responsible for dividing a job into worker subtasks and enqueuing them,
- `aggregate()` – executed when the user requests the results. It may perform additional computations such as summing or finding average of results returned by the workers.

The worker has to provide only one method:

- `execute()` – performs the computations and returns the results.

Finally, the application is deployed on the server using the standard tools of App Engine.

The pseudo-codes of master and worker running a simplified Monte Carlo integration are shown in Figures 2 and 3. The helper methods are used to create worker tasks, add them to the queue, obtain a list of workers, retrieve their results and check the termination condition.

```
1  public List<Long> execute(String taskData, int workersNumber)
2  {
3     List<Long> workersList = new ArrayList<Long>();
4
5     for (int i=0; i<workersNumber; i++)
6     {
7        Long workerTaskId = helper.createWorkerTask();
8        workersList.add(workerTaskId);
9
10       helper.enqueueWorker(workerTaskId,
11              "MonteCarloSimulationWorker",
12              dataForThisWorker);
13    }
14    return workersList;
15 }
16
17
18 public String aggregate(Long masterTaskId)
19 {
20    String result;
21    List<Long> workersIds = helper.getWorkers(masterTaskId);
22
23    for (Long id: workersIds)
24       processWorkerResult(result,
25                           helper.getWorkerResults(id));
26
27    return result;
28 }
```

Figure 2. Simplified code of Monte Carlo master task

The master-worker pattern enables an easy implementation of useful solutions to a wide range of compute intensive problems. Although in many cases it is possible to run parameter study applications using MapReduce model and vice versa, our framework has a slightly different design philosophy (task-driven vs. data driven). Master task explicitly creates tasks for workers and enters them into the queue, while in MapReduce

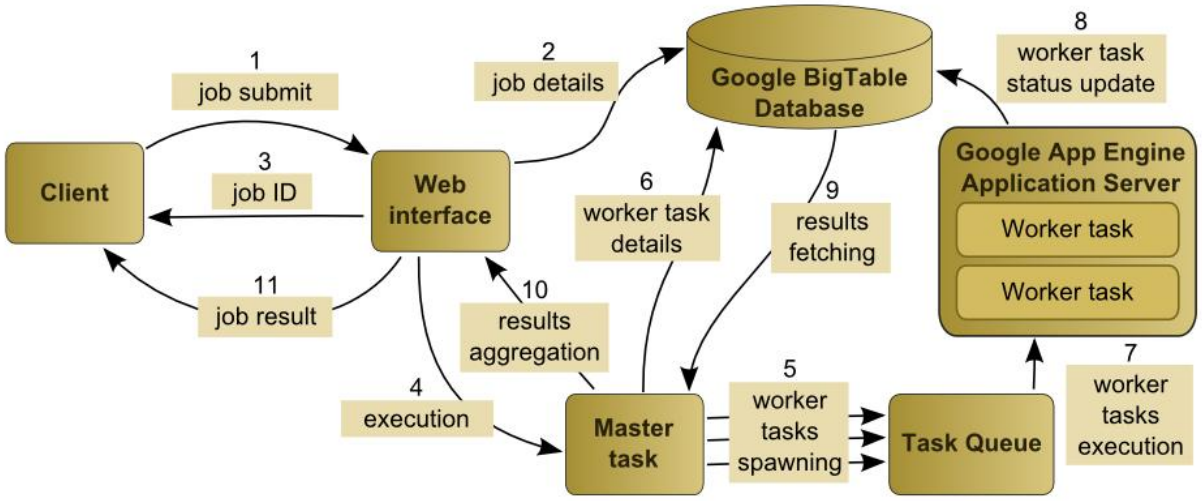

Figure 1. The computing environment based on App Engine Task Queue service and its functionality; details are explained in Section 4.1.

```
1  public String execute(String data)
2  {
3      helper.startTimer();
4      for (long i = 0; i<numberOfSteps; i++)
5      {
6          doWork();
7
8          if (helper.shouldIFinish())
9              return WorkerTaskHelper.TIME_LIMIT_EXCEEDED;
10     }
11     return result;
12 }
```

Figure 3. Simplified code of Monte Carlo worker task

the tasks (mappers) are created implicitly based on entries in the input dataset. MapReduce is better suited for large-scale data processing, while our framework is more convenient for compute-intensive parameter-study applications, such as Monte Carlo simulations or various optimization problems. Our approach is also better suited for the cases when tasks are dynamically added to the queue or when the aggregate phase needs to be run periodically or interactively e.g. to assess the current work progress.

One of the problems which has to be solved manually by application programmer is the task partitioning granularity. When submitting tasks, the master class has to ensure that run times of all the tasks fit into the 10-minute limits imposed by the framework. Moreover, as observed in our experiments (see Section 5) higher granularity yields better performance due to the underlying auto-scaling and load-balancing algorithms of App Engine. The task granularity can be determined using simple benchmarking by running tasks locally and on the infrastructure. The situation is very similar to computing clusters with batch processing systems such as Condor [1], where users have to adjust their tasks sizes to the limits imposed on queues for scheduling purposes.

5 Performance Evaluation

The goals of the tests were to verify the framework operation and to measure its performance when utilizing the freely available resources provided by Google App Engine.

A special attention was devoted to the case where one large job, consisting of multiple worker tasks is submitted to the system. The tests were run in two series of different granularity, but of equal total number of computing steps. In the first series each worker task performed 130×10^6 simple Monte Carlo computation steps which took approximately 20-28 seconds. Number of worker tasks in one job varied from 1 to 50. In the second series tasks were 5 times smaller but the total number of computational steps was kept the same.

As the load balancing is provided dynamically by the App Engine platform, it is impossible to predetermine how many computational nodes will be actually used. Consequently, the traditional way of measuring parallel efficiency is not applicable in this case. As a result an average number of computational steps executed per minute as a function of number of computational steps in the job has been chosen as a benchmark. The results are presented in Fig 4.

The results show that it is possible to achieve up to almost 3×10^9 computation steps per minute when the tasks are larger and up to more than 3.5×10^9 steps per minute in the second scenario. Considering the fact that it takes almost 30 seconds to finish 30×10^6 steps it means that the throughput is mostly limited by the 15 CPU minutes/minute quota (15 CPU minutes/minute is approximately equal to $130 \times 10^6 \times 2$ steps per minute $\times 15$ cores = 3.975×10^9 steps per minute). As a result of these

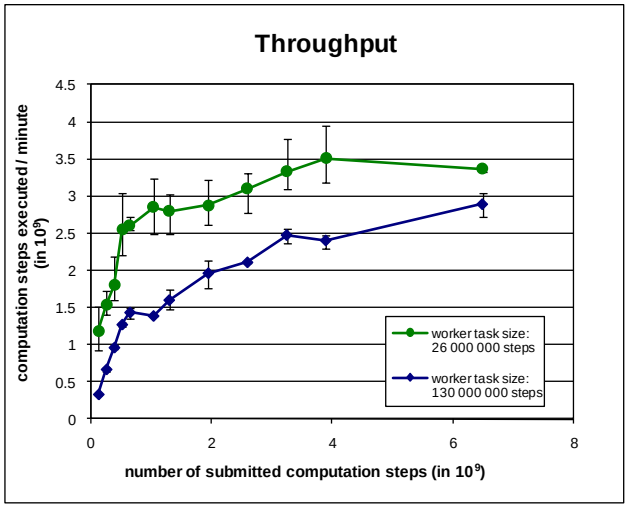

Figure 4. Worker tasks throughput when submitting one big computational job. Task size differs in the number of Monte Carlo computation steps.

quotas, during longer tests we have observed periods of inactivity (no tasks executed) up to 90 seconds long.

We have observed that using smaller tasks (finer granularity) yields significantly better performance. It is easy to explain when there are a few tasks and the execution may be parallelized. In larger tests, however, the difference should disappear. The explanation comes from analysis of the way new computational nodes are instantiated by the load balancer of App Engine. When there are numerous smaller tasks, more servers are prepared as the queue is longer. In the first test up to 13 application instances have been created, while in the second one up to 24 have been used (as measured by the monitoring console of App Engine). The observation that smaller tasks (finer granularity) yield better performance contradicts a known principle of parallel programming which recommends agglomerating smaller tasks into larger sets to minimize communication to computation ratio. However, in the case of App Engine infrastructure which uses automatic scaling based on the task (request) submission rate, the opposite strategy of increasing granularity turns out to be more effective.

Some time is required for Google App Engine to distribute computational load to additional application servers and for them to prepare the environment. This explains why measured execution times have highly fluctuated (up to 20% difference in some cases).

Finally, we compared the performance of the application on free App Engine to the same computation run sequentially on a single core of 2.16 GHz Intel Core 2 Duo T7400 processor. The top performance (3.5×10^9 steps per minute, see Fig. 4) of free App Engine is over 10 times higher than the one we achieved on this single core. We find this result very promising, taking into account the obvious difference in cost of these computing facilities. Certainly, the App Engine free quotas do not offer such sustained performance, but for short peaks (also known as cloud bursts) the actual performance may become useful for some practical applications.

6 Comparison with paid version of App Engine

Another interesting feature of Google App Engine is that once the paid version is activated, the resource quotas are increased, but the same free usage limits remain available. From the performance perspective, the most relevant quota of 15 CPU-minutes / minute is increased to 72 CPU-minutes/minute. This enables to increase the peak performance and still the free computing is possible for up to 6.5 CPU-hours daily.

To compare the performance of paid version of App Engine to the free one, we run the tests similar to the ones of section 5. Fig. 5 shows the results of the tests in which worker task size was 130×10^6 Monte Carlo computation steps. Despite random fluctuations observed, the performance of free version is similar to the results from Fig 4, but the paid version achieves nearly double average performance. Similarly, the peak throughput achieved was 7×10^9 computation steps per minute.

To better understand how the performance depends on task granularity, we run another series of experiments in which we measured the number of instances versus task submission rate. Every 30 seconds, we submitted a set of N subtasks, each of 3.3×10^6 computing steps. During the test, we were increasing N gradually: for paid version we used step values of $N = 200, 300, 400, 800$ and for free version $N = 200, 300, 400, 600$. The test on a free version had to be shorter to fit into the daily quota.

The results of the tests are shown in Fig. 6 and 7. It can be seen that in both cases App Engine platform scales the number of instances with the tasks submission rate, but the paid version achieves nearly double number of

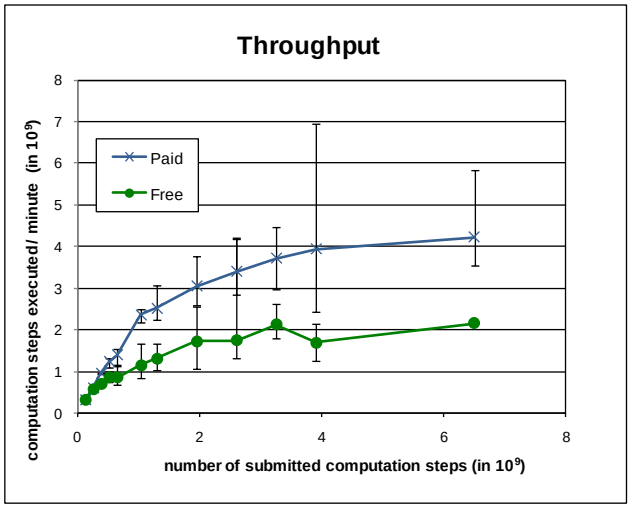

Figure 5. Comparison of performance of paid and free version of App Engine

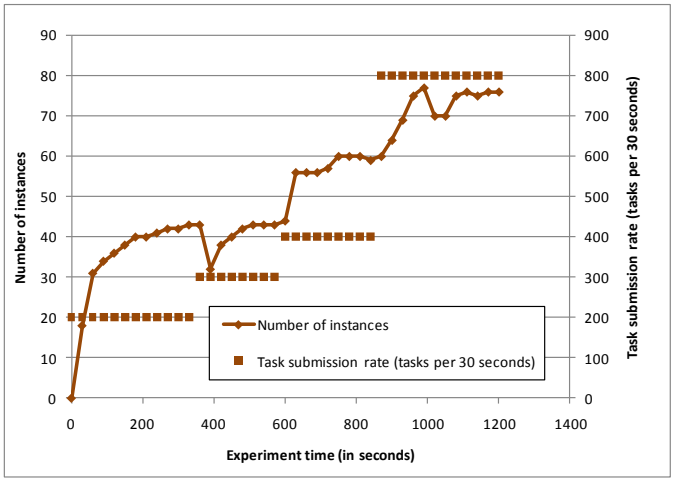

Figure 6. Scaling of instances with task submission rate for paid version of App Engine.

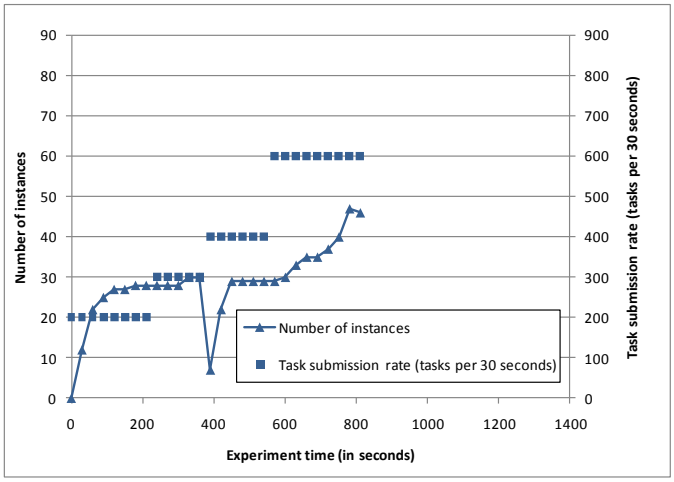

Figure 7. Scaling of instances with task submission rate for free version of App Engine.

instances vs. free version. Similarly, we observed on the App Engine console that the number of active instances for free version was also smaller, which reflects the fact that the quota of 15 CPU-minutes / minute is active. In the paid version of App Engine, the cost of CPU hour above the free daily quota of 6.5 CPU hours is $0.10. Thus, e.g. an experiment similar to the one shown in Fig. 6 consuming 20 CPU hours, out of which 6.5 are free, would cost $1.35.

The main conclusion from the experiments is that although the paid version of App Engine allows increasing daily CPU consumption (up to the budget available), it is not easy to increase the peak performance. To achieve that, the proper task granularity and submission rate need to be maintained, and adjusted to the autoscaling mechanisms of App Engine Platform.

7 Comparison with other free computing options

To compare the App Engine to other freely available cloud computing platforms, we run our tests on Amazon EC2, which offers free access for the first year to a single virtual machine instance of t1.micro type. According to EC2 specification, this instance type "provides a small amount of consistent CPU resources and allows to burst CPU capacity when additional cycles are available". In contrast to App Engine, EC2 is an IaaS cloud infrastructure, which gives full (root) access to the running virtual machine: it thus allows logging in into the running instance, installing required programs, copying data and running the application.

To estimate the actual performance, in our tests we run ca. 2500 tasks of 20 million steps each during the period of 3 days. The allocated virtual machine was running on Intel Xeon E5430 CPU at 2.66GHz, but our measurements show that the full processor capacity is indeed available only for short periods of time. The shortest processing time of a single task was 1.8 seconds, which gives 0.66×10^9 steps per minute and corresponds to the period when the VM had full access to the CPU core. However, the longest processing time was 99 seconds, which gives only 0.01×10^9 steps per second. Since the bursts of high CPU allocation are relatively short (approximately 10 seconds for each 3-minute interval) the average processing time is 35 seconds, which gives 0.03×10^9 steps per minute. This suggests that our t1.micro instance was assigned a fraction of ca 5% of a single CPU core. On the other hand, it is possible to manually instantiate up to 20 instances on EC2: this will consume monthly free quota in 37 hours, but will allow to increase peak performance 20 times, which is equivalent to having access to 100% of single CPU core for that period of time.

We can note that although free App Engine can provide higher peak performance, EC2 can be used to provide consistent performance over long periods of time. However, the App Engine quota of 6.50 CPU hours per day (27% of CPU time) is much larger than the observed EC2 quota of 5% monthly.

The estimated parameters of free cloud offerings are summarized in Table 1. We can conclude that both cloud providers offer a fraction of CPU time for free. Currently Google App Engine is not limited in time and it offers a reasonable performance of 27% of daily CPU usage, but due to its platform-as-a-service constraints, it requires specific frameworks like the one we developed to exploit the offered processor cycles.

Our approach to free computing is not a replacement for the mainstream sources of CPU power available nowadays to scientists. TeraGrid or DEISA offer access to supercomputing facilities in the USA and Europe, while grid computing infrastructures such as European Grid Infrastructure (EGI) or Open Science Grid (OSG) provide tens of millions of CPU hours per month. There

Table 1
Comparison of free computing capabilities of major cloud providers. Peak speedup computed vs 2.16 GHz PC (laptop) processor.

	App Engine	Amazon EC2
Free period	Unlimited	1 year
CPU share	27% daily	5% monthly
Peak speedup	10	2

are also community cloud projects such as FutureGrid[3] or Open Science Data Cloud[4] offering resources for researchers. Moreover, cloud providers such as Amazon offer access to their resources to researchers in the form of grants[5], and there are also offers from Google[6] to fund 10 research projects which will consume total of 1 billion CPU hours per year. However, these projects are not available to general public and even for researchers they are highly competitive. On the other hand, our method should be positioned on the opposite end of the spectrum, more similar to parasitic computing of the early days of the Web. We have to note that the Internet has often proved that such bottom-up approaches can evolve into mainstream solutions. One example is the Condor project which started more than 25 years ago as an attempt to harness the power of idle workstations and now is one of the crucial components of largest production grid infrastructures (OSG and EGI). Also the BOINC project which evolved from early SETI@Home experience was the first to achieve the PetaFLOP performance[7].

8 Conclusions and Future Work

In this paper we have presented a new method of providing free computing facility to solve computationally intensive problem, using the framework based on recently introduced Google App Engine Tasks Queue service. The conclusion is that the proposed new approach may support solving parameter study parallel problems using a free, auto-scalable Google infrastructure. This is, at least partially, in contradiction to the conclusion of [11] that the cloud computing will never be free. The Task Queue allows running jobs in parallel and asynchronously when the servers are available. The infrastructure and the Queue provide load balancing efficient enough to almost totally utilize available resources. We have also shown that Google App Engine can provide a computational environment which is more effective than a personal computer.

Moreover, when keeping within the constraints, the infrastructure can be used at no cost. Unfortunately, the usability of presented solution is limited by quotas available for free users. However, we expect that in the future, when the number of resources available from Google and other providers will increase, it will be possible to run more advanced and complex applications on such freely accessible infrastructures.

Future work includes integration of our framework with AppScale [16] which is an open source implementation of App Engine framework. One of its advantages is that it can be deployed on Amazon EC2 and community clouds running e.g. Eucalyptus cloud management software stack. By combining our framework with AppScale capabilities it would be possible to use the same API as the one offered by Google App Engine to exploit the free computing capabilities of other cloud providers. Finally, it would be also interesting to evaluate the performannce of free resources available from Microsoft Azure.

We envision that free computing will become more prolific, as cloud offerings will become more popular. An interesting futuristic scenario would be possible by combining our App Engine-based free computing method with the volunteer computing systems, such as Folding@Home etc. Currently, participants of such systems donate free CPU cycles of their home PCs or gaming consoles such as Sony Playstation, which in fact requires some contribution from the volunteers in the form of their invested personal time and also increased electricity consumption. The volunteers may donate their free computing quotas from e.g. App Engine and free storage from e.g. Dropbox.com, Microsoft SkyDrive or Apple iCloud. Although such contributions may seem cost-free, in fact they will also require some form of sacrifice: users who donate their free App Engine CPU cycles may not be able to host their personal website there for free and the users sharing their cloud storage will have less space for their personal photos or backups. Moreover, some users may also donate their spare cycles or gigabytes of paid accounts. This may lead to interesting new forms of volunteering in the Internet enabled by cloud computing.

3. https://portal.futuregrid.org/
4. http://www.opensciencedatacloud.org/
5. http://aws.amazon.com/education/
6. http://research.google.com/university/exacycle_program.html
7. http://boinc.berkeley.edu/dev/forum_thread.php?id=5214

Acknowledgments

The authors would like to thank Google for providing free access to App Engine infrastructure. Maciej Malawski acknowledges the support from the UDA–POKL.04.01.01-00-367/08-00 AGH grant. This work was partially supported by EU project VPH-Share: Virtual Physiological Human: Sharing for Healthcare - A Research Environment (contract 269978).

References

[1] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," Concurrency and Computation: Practice and Experience, vol. 17, no. 2-4, pp. 323–356, 2005. [Online]. Available: http://dx.doi.org/10.1002/cpe.938

[2] J. Moscicki, M. Lamanna, M. Bubak, and P. Sloot, "Processing moldable tasks on the grid: Late job binding with lightweight user-level overlay," Future Generation Computer Systems, vol. 27, no. 6, pp. 725 – 736, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11000057

[3] G. Pallis, "Cloud Computing: The New Frontier of Internet Computing," IEEE Internet Computing, vol. 14, no. 5, pp. 70–73, September 2010. [Online]. Available: http://dx.doi.org/10.1109/MIC.2010.113

[4] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky Computing," Internet Computing, IEEE, vol. 13, no. 5, pp. 43–51, August 2009. [Online]. Available: http://dx.doi.org/10.1109/MIC.2009.94

[5] A. L. Barabási, V. W. Freeh, H. Jeong, and J. B. Brockman, "Parasitic computing." Nature, vol. 412, no. 6850, pp. 894–897, August 2001. [Online]. Available: http://dx.doi.org/10.1038/35091039

[6] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu, "Parallel scripting for applications at the petascale and beyond," Computer, vol. 42, no. 11, pp. 50–60, 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.365

[7] Google.com, "Appengine," 2011, http://code.google.com/appengine/.

[8] U. Schwiegelshohn, R. M. Badia, M. Bubak, M. Danelutto, S. Dustdar, F. Gagliardi, A. Geiger, L. Hluchy, D. Kranzlmueller, E. Laure, T. Priol, A. Reinefeld, M. Resch, A. Reuter, O. Rienhoff, T. Rueter, P. Sloot, D. Talia, K. Ullmann, R. Yahyapour, and G. von Voigt, "Perspectives on grid computing," Future Generation Computer Systems, vol. 26, no. 8, pp. 1104 – 1115, 2010.

[9] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[10] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud Computing: Distributed Internet Computing for IT and Scientific Research," IEEE Internet Computing, vol. 13, no. 5, pp. 10–13, September 2009. [Online]. Available: http://dx.doi.org/10.1109/MIC.2009.103

[11] D. Durkee, "Why cloud computing will never be free," Communications of the ACM, vol. 53, no. 5, pp. 62–69, May 2010. [Online]. Available: http://dx.doi.org/10.1145/1735223.1735242

[12] R. Barga, D. Gannon, and D. Reed, "The Client and the Cloud: Democratizing Research Computing," IEEE Internet Computing, vol. 15, no. 1, pp. 72–75, January 2011. [Online]. Available: http://dx.doi.org/10.1109/MIC.2011.20

[13] A. Bedra, "Getting Started with Google App Engine and Clojure," Internet Computing, IEEE, vol. 14, no. 4, pp. 85–88, 2010. [Online]. Available: http://dx.doi.org/10.1109/MIC.2010.92

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Trans. Comput. Syst., vol. 26, no. 2, pp. 1–26, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1145/1365815.1365816

[15] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communication of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[16] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "AppScale: Scalable and open AppEngine application development and deployment," in Cloud Computing, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, O. Akan, P. Bellavista, J. Cao, F. Dressler, D. Ferrari, M. Gerla, H. Kobayashi, S. Palazzo, S. Sahni, X. S. Shen, M. Stan, J. Xiaohua, A. Zomaya, G. Coulson, D. R. Avresky, M. Diaz, A. Bode, B. Ciciani, and E. Dekel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 34, ch. 4, pp. 57–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12636-9_4

Maciej Malawski Ph.D. in computer science and M.Sc. in computer science and in physics. Postdoc at Center for Research Computing, University of Notre Dame. Researcher and lecturer at the Institute of Computer Science AGH and at ACC Cyfronet AGH. Coauthor of over 50 international publications including journal and conference papers, and book chapters. Involved in the EU IST ViroLab project, where he was the leader responsible for the middleware task and for contacts with external users. Responsible for Virtual Laboratory developed in PL-Grid project. His scientific interests include parallel computing, grid and cloud systems, distributed service- and component-based systems, and scientific applications.

Maciej Kuźniar is a computer science student at the Institute of Computer Science AGH. His scientific interests include parallel algorithms, distributed systems, and scientific simulations.

Piotr Wójcik is a computer science student at the Institute of Computer Science AGH. His scientific interests include digital image analysis and scientific simulations.

Marian Bubak Ph.D., is an adjunct at the Institute of Computer Science AGH, a staff member at the ACC Cyfronet AGH, and the Professor of Distributed System Engineering at the Informatics Institute of the Universiteit van Amsterdam. His research interests include distributed and grid systems for scientific simulations. He co-authored about 230 papers. He lead the architecture team of the EU IST CrossGrid Project, he was the Scientific Coordinator of K-WfGrid Project and the member of the Integration Monitoring Committee of CoreGRID. He served as a program committee member, chairman and organiser of several international conferences (HPCN, Physics Computing, EuroPVM/MPI, SupEur, HiPer, ICCS, HPCC, e-Science'2006); he is co-editor of 17 proceedings of international conferences.