Algorithm Benchmarking Services in the Cloud

No Author Given

No Institute Given

Abstract. We describe the design, prototype implementation and use of a cloud service for benchmarking algorithms. The need for such a service and the application domain for the service is defined. We illustrate the architectural design of the service. Data gathered by using the prototype on a public cloud platform is compared with data gathered using a local host. Because the ranked results are strongly correlated, they support the validity of relying on the cloud for benchmarking in terms of comparative, though not absolute outcomes.

Key words: PaaS, Algorithm benchmark

1 Introduction

There are many cases, both in the industry and in research, where benchmarking of a software product is important. Hence, an investigation into developing a benchmarking service which can accept any source code of any modern programming language seems like a worthwhile endeavour.

Of course, in certain contexts cloud-based benchmarking would not be appropriate. Issues such as deciding which cloud vendor to choose becomes of particular importance. Other critical issues include internet latency [6] and data security [8]. Yet, a benchmarking cloud service can potentially lower the total amount of time spent benchmarking. Since the cloud is virtually limitless, a customer can run as many benchmarks concurrently on the cloud as needed.

Another significant advantage is that the customer does not need to be concerned about hardware failures or about the service being down for extended periods of time, depending on the contents of the Service Level Agreement (SLA) between the customer and the cloud service provider.

This paper focuses on providing a provisionally secure benchmarking cloud service that can run any program with an associated data set. The research described below set out to explore the implications and opportunities of running cloud-based benchmarks. In the next section, the work most relevant this research is briefly described. Then, in Section 3 the REST-based service-oriented architecture of the benchmarking system that was implemented is described. Section 4 is devoted to a description of how data produced by the benchmark was gathered and Section 6 indicates how this data was subsequently analysed. The final section then offers the conclusions to be drawn from the study and potential avenues for future research.

2 Related work

Services for benchmarking of algorithms are rare. Faro and Lecroq [4] provide software for benchmarking of string matching algorithms. To use the system the user has to intall a local copy of it. Chen et al [2] implemented an algorithm benchmark system as a web service. It focuses on comparing the performance among algorithmic solutions rather than measuring the exact information of machine computing power. Pampara et al [10] describe a system that provides simulation services for algorithms with artificial intelligence applications while Kerschbaum et al [8] presented an example cloud application for collaborative benchmarking of encrypted data. There are attempts at benchmarking the performance of different clouds. Recently Iosup et al [6] analysed the performance of cloud computing services for scientific computing workloads and conclude that current cloud services need significant enhancements before they will be useful for scientific applications. Contrary to this verdict, we are of the opinion that some scientific applications such as comparative benchmarking may already be possible. To our knowledge a formal benchmarking cloud service, where one can benchmark a specific user-supplied program, has not been provided yet.

3 Architectural design

The design of the service is according to Service Orientated Architecture (SOA) [3], which complies to the HTTP/1.1 [5] protocol and transforms the service into a component. This means that as long as one can access the service, one can request valid services.

The client does not require further knowledge of the contents of the component, such as language in which it was written. This is shown in Figure 1. The client uses the service by accessing its provided interfaces. These interfaces are defined in terms of public ports that are accessible via the Uniform Resource Locator (URL) of the service.

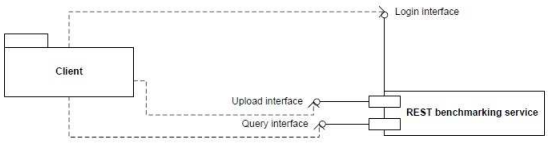


Fig. 1: Component diagram of the benchmarking service.

To use each of the provided interfaces, the client has to submit valid data in the format that is defined by the interface. For example, to use the “Upload service”, the client has to specify the benchmarking options as well as the location of the archive that has to be uploaded to the benchmarking service in the format required by this interface.

Other design approaches were considered, for example using Java Remote Method Invocation (RMI) [12]. This would, however, increase the maintenance requirements on both the web-service and the client program.

The architectural style we chose for the web service is Representational State Transfer (REST) [7]. It is compatible with SOA and it means that no state is kept in memory or storage by the web-service. Each of our instances have such a service running, which also is the entry-point. Another standard would be to use a normal Web-Service Description Language(WSDL). This would, however, pose the need to update the WSDL when the service changes. The REST service automates this procedure, where the programmer or maintainer does not need to update the WSDL file any-more.

The service itself makes use of a façade into the benchmarking program, which is the backbone of the whole benchmarking operation. The benchmarking program makes use of other software that is pre-set-up with the benchmarking service. This includes software to extract the source code and datasets that are received and scripts to compile and run these programs. The compiler that is used to compile the source code is specified by the user.

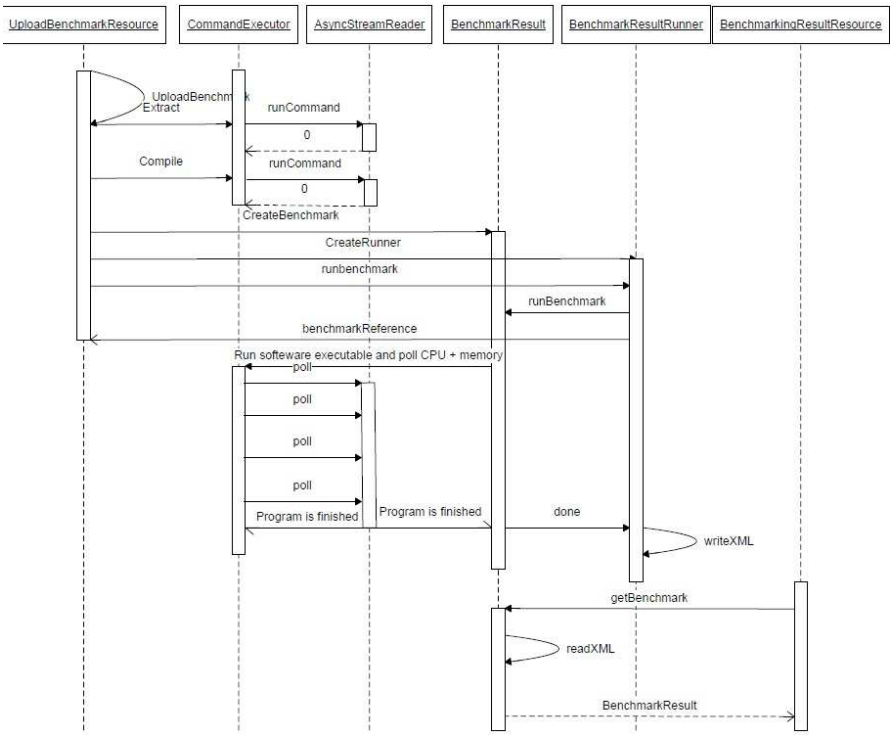


Fig. 2: Sequence diagram of the benchmarking service.

This benchmarking method is similar to the one described in [11] and is recommended in [14]. Just before the execution starts, a timer is started and an additional thread is started to monitor the memory resident set size (RSS).

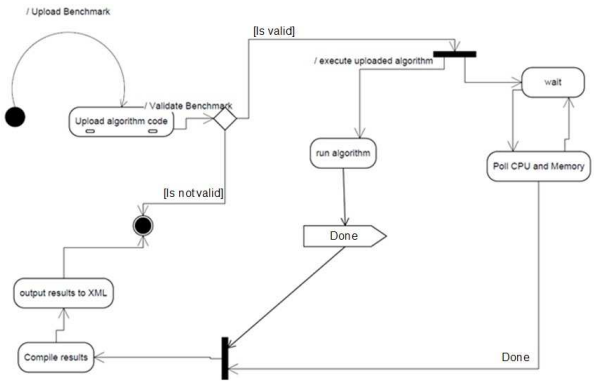


Fig. 3: State diagram of the benchmarking service.

This thread polls the RSS at regular user specified intervals. The polling stops when the program being benchmarked has stopped. These measurements are written to a XML file, which can later be queried by the user. These steps are depicted in Figures 2 and 3.

Figure 4 shows how the system is deployed. The cloud instances run on Amazon EC2 instances powered by Ubuntu, where each instance is set-up in the same way. The service runs on a glassfish server and the security of the instances has been set-up to only accept traffic on the ports operated by the service. From the clouds that were examined, the Amazon EC2 cloud was chosen, because of its free experimental tier as well as its stable resource usage policy. Another attraction was that its cloud specializes in computations.

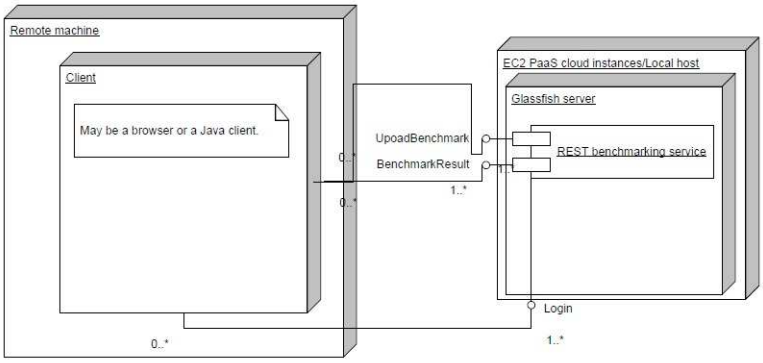


Fig. 4: Deployment diagram of the benchmarking service.

3.1 Algorithms

The algorithms that need to be benchmarked are coded in a supported programming language. The currently supported programming language is limited to C++, but this can be easily extended to virtually any programming language.

Currently the user has to provide information of the algorithm that needs to be benchmarked in a compressed tar archive or zip file. The archive has to contain the the data, the source files and all other files that are required to successfully compile the software. It should also contain a Makefile to compile the source code.

3.2 Data

Currently we require the implemented algorithm to read its data using standard input. This input should be provided in a text file. This text file has to be included in the same archive mentioned in Section 3.1 that is used to submit the algorithm to be benchmarked. Our system runs the algorithm with this supplied data by substituting standard input stream with the data in the given text file.

3.3 Options

Before the customer submits the benchmark, one can change options that affect the benchmarking service. One such option is the verboseness of the collection of the data, which means that an indication is given of how often the benchmarking variables are to be polled. A second option is the sampling size of the benchmark, which means that an indication is given of many times the execution of the benchmark algorithm should be repeated.

3.4 Execution

The current service only supports the benchmarking of algorithms using the C++ programming language. It uses the g++ compiler, version 4.4.6, to compile the code on the cloud platform. Since the customer of the service has to provide their own makefiles to compile the source code, they are free to use any compiler flags of their choice, for example to specify the level of optimising. The makefile should also be in the compressed file that is specified in Section 3.1.

3.5 Measurements

At regular user-specified intervals in time during the benchmarking, the `ps` command is issued. This Linux command lists the processes that are running and associated information, including memory usage. An instance is executed in a separate thread, which polls the memory usage of that process.

If the benchmarking service is running on a single-core instance, as was the case here, the effects are noticeable in when considering the execution time. To account for this, the customer is advised to keep the rate of polling on the RSS the same throughout the benchmarking. On a multi-core instance, the threads are managed by a thread-pool and there is virtually no effect on the system.

For each algorithm run, the average memory usage and execution time was computed as follows, respectively:

$$Avg_{mem} = \frac{\sum_{i=1}^{N}(\frac{\sum_{n=1}^{J_i} m_{n,i}}{J_i})}{N}$$
$$Avg_{time} = \frac{\sum_{i=1}^{N} t_i}{N}$$

where N is the total number of iterations of the benchmark; J_i is the number of times the memory is polled during the i^{th} iteration; $m_{n,i}$ is the memory reading taken at the n^{th} poll during iteration i; and t_i is the time measured for the i^{th} iteration to execute.

4 Data gathering

4.1 Platforms

To be able to validate the results of benchmarks that were performed using our service, we performed the same benchmarks with two configurations. The one configuration entailed installing the service on a cloud service while the other configuration entailed simulating the service on a local machine.

The benchmarking service on the cloud was running on Ubuntu 11.10 on a Glassfish server in a Java VM. The Virtual Machine, or the cloud instance, was powered by a 2 GHz Xeon single-core processor with multi-threading support. The instance had 670 MB of RAM which it could use. The cloud platform chosen was the Amazon EC2 PaaS cloud and the cloud vendor abstracts the above-mentioned resources with 1 elastic compute unit, or EPU.

The simulated benchmarking service ran on Ubuntu 12.04 on a Glassfish server in a Java VM. This local host was powered by a second generation Intel i7 sandybridge 8 core processor, which operated normally at 2.4 GHz and has a turbo mode of 3.1 GHz. The local host has 6 GB of memory at its disposal.

4.2 Algorithms

In this study we used different algorithms that solve the same problem namely to calculate the transitive closure of a finite binary relation. Ideally implementations of these algorithms should also be provided in toolkits that in turn can be used to support industrial software construction.

The problem that is solved by the algorithms that we used to illustrate our benchmarking service is not relevant to the scope of this article. Table 1 shows their theoretical time and space complexities.

With the exception of the Prosser algorithm, two implementations of each of these algorithms were benchmarked. Algorithms A to D used a vector of character arrays to represent the adjacency matrix of the relation being manipulated while all the other algorithms used a vector of dynamic_bitsets as defined in

Table 1: Algorithms that were benchmarked

Algorithm	Author	Time Complexity	Space Complexity
A and E	Warren [15]	$O(n^3)$	$O(n^2)$
B and F	Warshall [16]	$O(n^3)$	$O(n^2)$
C and G	Baker [1]	$O(n^3 log\ n)$	$O(n^2)$
D and H	Martynyuk [9]	$O(n^3 log\ n)$	$O(n^2)$
I	Prosser [13]	$O(n^4)$	$O(n^2)$

the boost library to represent this matrix. Owing to its marked inefficiency in relation to the other algorithms, we decided not to the benchmark of the Prosser algorithm that used character arrays. Thus a total of nine algorithms were included in this illustration of our benchmarking service.

4.3 Input data

Ten synthetic data sets were generated, each representing a two-dimensional array of bits. These data sets were stored and used in our benchmarks. One array of each of ten different sizes were generated. The smallest data set was 100×100 and the largest 1000×1000. When generating a data set, we used the random function provided in the Java platform edition 1.6 to generate a 0 or a 1 for each entry in the matrix. The function was used with lower value 0 and the upper value 1. The resulting data sets were thus arrays with an approximate density of 50%.

5 Results

We performed 50 executions of each of the algorithms with each of the data sets on each of the platforms described in Section 4.

Figure 5a shows the average time taken for the associated algorithms to finish execution for each of the data sets on the local machine, while Figure 5b shows the average time taken for the associated algorithms to finish execution for each of the data sets on the cloud. Note that the scale for these results are significantly different. The average performance time on the local machine is 4.22% of the average performance time on the cloud.

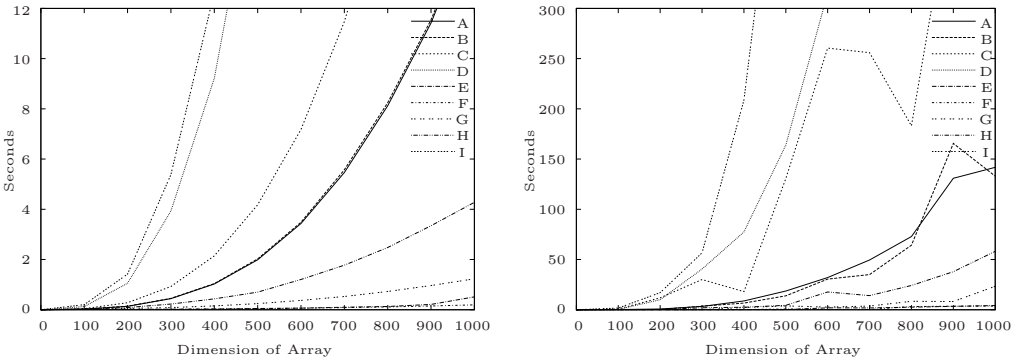
Figure 5c shows the average memory required for the associated algorithms when executing on each of the data sets on the local machine, while Figure 5d shows the average memory required for the associated algorithms when executing on each of the data sets on the cloud. Note that the scale for these results are the same.

6 Data analysis

An overall measure of algorithm performance was defined and computed for each algorithm on each platform as follows: for a given array size, algorithms were ranked from 1 (fastest) to 8 (slowest). These ranks were obtained for each array size. The sum of an algorithm's ranks over all array sizes was used as that algorithm's overall performance measure. Overall algorithm performance was then ranked from 1 (overall fastest algorithm) to 8 (overall slowest algorithm) on each platform. The well-known non-parametric Spearman's ranked correlation coefficient was then computed to indicate how the rankings on the two platforms

(a) Execution times on the local host.

(b) Execution times on the cloud host.



(c) Memory usage on the local host.

(d) Memory usage on the cloud host.

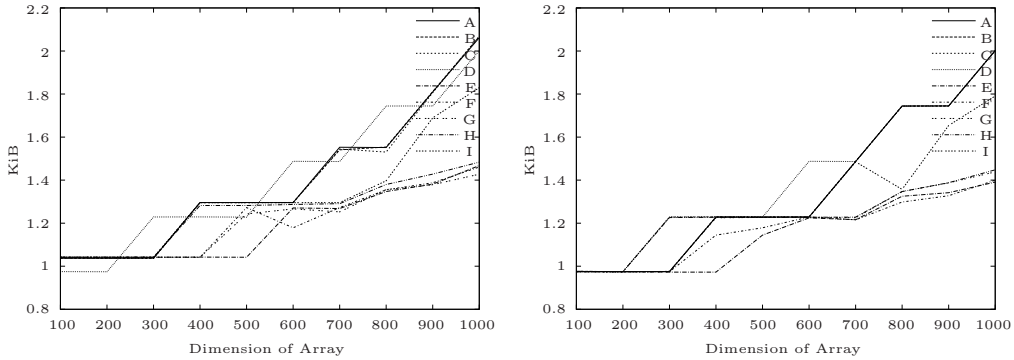


Fig. 5: Performance Measurements

compared. The correlation coefficient of $\rho = 0.983$ which was obtained is well above the 99% one-tailed significance level value (0.833), affirming that the correlation of performance on the different platforms is highly significant. A similar procedure was followed in respect of *memory usage* between the platforms. The correlation coefficient of $\rho = 0.679$ is just lightly above the 95% one-tailed significance level value (0.643) suggesting a somewhat weaker correlation of memory performance on the two platforms. This is probably a result of different memory management policies that are applied on the different platforms. Furthermore, the memory management of the virtual machine on the cloud may be effected by the memory management imposed on it by the physical machine running the virtual machine. If the memory usage of algorithms need to be compared, the memory consumption of the process itself needs to be isolated from the memory that is consumed to provide the overhead to run the process. In our case this was not done.

7 Conclusion and future work

This paper compared the results of benchmarking of algorithms that were executed on a prototype cloud benchmarking service with the results when performing the same benchmarks on a local host. Our data analysis show that the results delivered by the benchmarking service on the cloud has a strong correlation with the results obtained using a local machine in terms of the execution time of the algorithms relative to one another. This is an indication that the idea of using a cloud service for benchmarking of algorithms is viable to measure relative time performance between algorithms. The correlation may be improved when the service is deployed on a more stable cloud platform as a service.

Our measurements to compare memory consumption of algorithms were not quite as convincingly correlated as in time measurements. To ensure that memory usage benchmarking is more convincingly performed by a cloud service, methods will have to be devised to isolate the memory usage of a process from the memory used by the system which created the process. Further research is to be conducted to identify techniques to achieve this.

We plan to improve the stability and usability of our cloud benchmarking service. Our goal is to make our service publicly available. A longer term vision will entail a service in which the user specifies a range of hardware parameters and desired performance metrics. The results of the benchmarks can be saved in a public archive. Once the service becomes popular this archive may evolve to provide a standard against which new algorithms can be measured.

References

[1] Baker, J.J.: A note on multiplying Boolean matrices. Communications of the ACM 5(2), 102 (1962)

[2] Chen, M.Y., Wei, J.D., Huang, J.H., Lee, D.T.: Design and applications of an algorithm benchmark system in a computational problem solving environment. In: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education. pp. 123–127. ITICSE '06, ACM, New York, NY, USA (2006)

[3] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)

[4] Faro, S., Lecroq, T.: Smart string matching research tool. `http://www.dmi.unict.it/~faro/smart/index.php` (2007), [Online:] Accessed 2012-06-08

[5] Group, N.W.: Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt` (June 1999), [Online:] accessed 2012-05-30

[6] Iosup, A., Ostermann, S., Yigitbasi, M., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. Parallel and Distributed Systems, IEEE Transactions on 22(6), 931 – 945 (june 2011)

[7] Jim Webber, Savas Parastatidis, I.R.: REST in Practice. O'Reilly Media, Inc. (2010)
[8] Kerschbaum, F.: Secure and sustainable benchmarking in clouds - a multi-party cloud application with an untrusted service provider. Business and Information Systems Engineering 3(3), 135–143 (2011), http://dblp.uni-trier.de/db/journals/bise/bise3.html#Kerschbaum11
[9] Martynyuk, V.V.: Transitively equivalent directed graphs. Cybernetics and Systems Analysis 9(1), 45 – 49 (January 1973)
[10] Pampara, G., Engelbrecht, A., Cloete, T.: Cilib: A collaborative framework for computational intelligence algorithms - part i. In: Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on. pp. 1750 –1757 (june 2008)
[11] Pieterse, V.: Performance of C++ Bit-vector Implementations. In: SAICSIT '10 Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists. pp. 242–250 (2010)
[12] Pitt, E., McNiff, K.: Java.rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
[13] Prosser, R.T.: Applications of Boolean matrices to the analysis of flow diagrams. In: Proceedings of the Eastern Joint Computer Confonference No. 16. pp. 133–138 (1959)
[14] Stapenhurst, T.: Benchmarking Book - A How-to-Guide to Best Practice for Managers and Practitioners. Elsevier (2009)
[15] Warren, Jr, H.S.: A modification of Warshall's algorithm for the transitive closure of binary relations. Communications of the ACM 18(4), 218 – 220 (1975)
[16] Warshall, S.: A Theorem on Boolean Matrices. Journal of the ACM 9(1), 11–12 (1962)