Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2011) 1–10

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Component Approach to Computational Applications on Clouds

Maciej Malawski[a,b,*], Jan Meizner[a], Marian Bubak[a,c], Paweł Gepner[d]

[a] *AGH University of Science and Technology, Krakow, Poland*
[b] *Center for Research Computing, University of Notre Dame, USA*
[c] *Informatics Institute, Universiteit van Amsterdam, The Netherlands*
[d] *Intel Corporation, Pipers Way, Swindon Wiltshire SN3 1RJ, United Kingdom*

Abstract

Running computational science applications on the emerging cloud infrastructures requires appropriate programming models and tools. In this paper we investigate the applicability of the component model to developing such applications. The component model we propose takes advantages of the features of the IaaS infrastructure and offers a high-level application composition API. We describe experiments on a scientific application from the bioinformatics domain, using a hybrid cloud infrastructure which consists of a private cloud running Eucalyptus and the Amazon EC2 public cloud. The measured performance of virtual machine startup time and virtualization overhead indicate promising prospects for exploiting such infrastructures along with the proposed component-based approach.

Keywords: cloud computing, component model, virtualization, performance, IaaS, FASTA

1. Introduction

Recently, cloud computing has become an interesting alternative method of providing computing and storage resources, which can be of potential interest for the computational science community. Cloud providers such as Amazon (EC2) or Microsoft (Azure) are starting to offer their resources to researchers free of charge while large-scale projects consider using cloud resources instead of buying and maintaining their own infrastructures. The Virtual Physiological Human research community [1] can serve as an example of this trend. At the same time some computer centers (e. g. SARA) have started to operate their own clouds [2]. Although cloud computing can be considered an evolution of grid systems as it focuses on distributed shared resource provisioning, there are some differences resulting e.g. from technological (virtualization and different middleware suites) and organizational (e.g. cost models) perspectives. These similarities and differences give rise to interesting research issues regarding methods, tools and environments for programming and execution of scientific applications on such infrastructures.

We have investigated the applicability of the component model to providing an environment for programming and running computational science applications on the cloud. It is based on our experience with the development of component models and tools for grid systems, such as the MOCCA framework [3] based on CCA [4] standard and the Grid Component Model (GCM) [5] based on Fractal. We observe an analogy between the distributed components

*Corresponding author
Email address: `malawski@agh.edu.pl` (Maciej Malawski)

which are dynamically deployed on a set of component containers and an Infrastructure-as-a-Service (IaaS) cloud model where virtual machine instances are dynamically deployed on the cloud. This analogy drives us to define the principles of a component model extension in such a way that cloud virtual machines may be considered as components which can be deployed, configured and composed as in the traditional component-based application. Besides its advantages such as portability and better isolation of components thanks to virtualization, this approach also introduces potential overhead which may be associated with the deployment and startup times of virtual machines as well as with virtualization, e.g. Xen. The objective of our investigation was to elaborate the cloud-component approach and to estimate the performance degradation.

To answer these research questions, we have organized the paper as follows. Section 2 discusses the related work on cloud computing for scientific applications. In section 3 we describe the rationale for the cloud component model and outline the principles of our approach. Section 4 describes a high-level scripting API which can be used for development of components and client applications. In section 5 we describe the experiments which we performed on a hybrid cloud installation consisting of an Intel cluster and Amazon services running a bioinformatics application on the example of the *ssearch36* program from the FASTA suite [6]. We measured the startup time costs, the virtualization overhead and the scalability of the application. In section 6 we draw conclusions and outline future work.

2. State of the art

Cloud computing is of interest for the computational science community as a promising alternative and extension of the grid computing model [7]. The advantages of virtualization include on-demand deployment and access to computing resources and also its applicability for provenance studies and result reproduction [8].

Several programming models and abstractions have been proposed to support scientific applications on cloud infrastructures. Among them one of the most promising is the workflow model. Experiments with e.g. Pegasus indicate [9] the advantages and costs of using these infrastructures. Traditional job processing techniques known from e.g. Condor can be applied to cloud infrastructures [10]. The Map-Reduce model known from Google [11] has been effectively used for computationally intensive bioinformatics applications [12]. There are also examples of clusters and grids being provisioned on demand [13], where traditional processing techniques can be applied. The Aneka framework proposes a programming model for economic-based scheduling and execution of cloud-based applications [14]. Other programming approaches (e.g. RightScale [15]) rely directly on cloud computing services.

Component programming model has been of interest for business and scientific applications for a long time and standards such as Common Component Architecture (CCA) [4] and Corba Component Model (CCM) [16] were created to support composition and deployment of applications on distributed resources. Of special interest is the Grid Component Model (GCM) [5], designed to support large-scale distributed applications on grid computing infrastructures. Component models have been also applied to large-scale master-worker applications [17]. Our previous work on the MOCCA component environment yielded a component framework for grid applications [3]. Component model is similar in many aspects to service-oriented architecture (SOA), but here we prefer to use the term “component” to emphasize that it is a unit of deployment.

Cloud computing relies, to a large extent, on virtualization techniques which offer flexibility and platform independence; however they also impose certain overhead upon systems and applications. Experiments and benchmarks indicate that virtualization overhead, whether through paravirtualization or Intel VT extensions [18, 19], is not particularly significant. The overhead for scientific applications tends to be more visible, especially in the case of parallel tightly-coupled benchmarks [20]. Recent studies of bioinformatics applications, including similarity searches, report overhead on the order of 10% [21] when running these applications on clouds.

The above described effort in the area of computational science applications on clouds indicates that the topic still requires research on new programming abstractions, improving efficiency and minimizing overhead. We believe that the work described in this paper constitutes an interesting approach in this regard.

3. Component model for cloud applications

3.1. Rationale for the cloud component model

The motivating observation for our work is the analogy between distributed component models and the capabilities offered by cloud infrastructures (Table 1). In the case of traditional distributed component models, such as CCM [16],

Table 1: Analogy between traditional distributed components and components on the cloud.

Distributed Components	Cloud components
Component	Virtual machine instance
Container	Cloud execution environment (EC2, Eucalyptus)
Create component instance	Run instance on the cloud
Component package	Virtual Machine Instance (virtual appliance, AMI, EMI)
Composition (connecting components)	Passing appropriate references (WSDLs, queue names, etc.)

an application is composed of independent units of code and execution called components. Components conform to a specific component standard which defines the interfaces (ports), dependencies, and rules governing interactions between the components, their lifecycle and other aspects. Components are deployed in containers which provide the execution environment and a virtualization (or abstraction) layer which isolates the components from the operating system and actual computing resources. Component models often define packaging standards which facilitate reuse and deployment of component code in a distributed environment.

IaaS cloud platforms are in many aspects similar to the traditional distributed component models, although there are obvious differences on the level of virtualization and the scale of the environment. The term "component" may denote a virtual machine instance which can be executed on the cloud. Such a "heavyweight" component includes a complete operating system with libraries required to execute the application code. The size of such a virtual appliance is definitely greater than that of a traditional component developed e.g. in Java, but the advantage is its full portability and OS independence. The role of container is played by the IaaS cloud platform which provides a hosting environment using virtualization technologies for creating components on demand. Thus, the whole Amazon EC2 [22], as well as the private Eucalyptus [23] cloud, can be considered large-scale containers for components. Actually, the API which is provided by Amazon and other cloud platforms enabling such operations as "create instance", "describe instances" or "destroy instances", resembles the API provided by component standards such as CCA, CCM or Fractal. Component packaging can be compared to standards for bundling and storing virtual machine images which are provided by cloud providers.

Based on this analogy we propose a component model which combines the advantages of both approaches. The advantages resulting from such a combination are as follows:

- On-demand deployment of custom application code on the infrastructure: computational science applications in the form of virtual appliance components can be easily deployed on the resources available on the cloud;
- Platform independence thanks to virtualization: ability to compose applications with e.g. Linux- and Windows-based components which is often the case in modern complex application scenarios requiring integration of multiple components produced by multidisciplinary teams;
- Better support for heterogeneous and legacy systems which impose specific software dependencies;
- Access to cloud-based services such as storage, queues and databases: Cloud platforms, in addition to basic IaaS capabilities, provide many other useful services which can be likened to container-provided component services;
- Better provenance and reproducibility thanks to permanent storage of software snapshots used for performing computations;
- On-demand pricing model well suited for multiphased computational science applications; It may become more economical for research teams and funding agencies to lease these resources on-demand from cloud providers instead of directly purchasing equipment such as servers and storage.

Clearly, the cloud computing model and the proposed component-based approach have drawbacks as well as advantages. Some of them are of an organizational or economical nature, such as pricing models, vendor lock-in problems, security issues and lack of standards. Here we focus on technical limitations, namely performance degradation and overhead introduced by the cloud platforms and virtualization technologies as well as by the proposed approach of creating "heavyweight" components in the form of virtual appliances. We believe that these limitations, although noticeable and measureable, can be estimated and that their cost can be minimized.

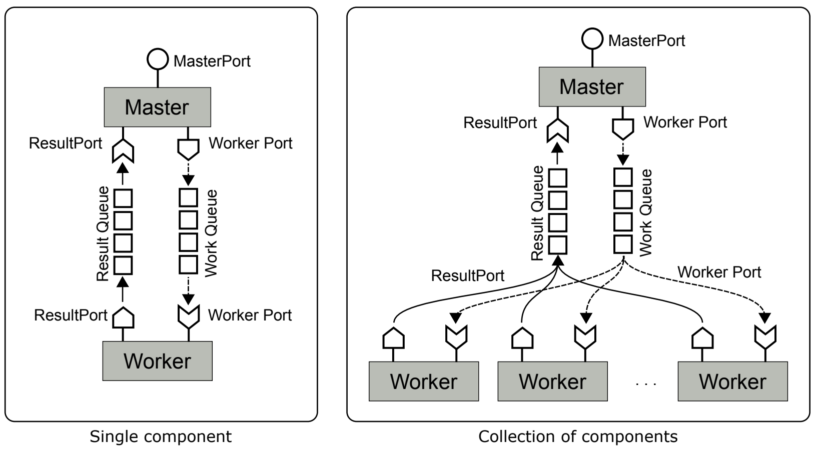


Figure 1: Simple Master-worker application decomposed into components. The left-hand side shows a single Worker instance, while on the right-hand we can see a collection of worker components connected to the same queue. The Master component is the same in both cases.

3.2. Principles of the proposed cloud component model

As indicated above, the main principle of the proposed component model is that a component is a virtual machine deployed on demand on the cloud. The component is packaged in the form of a virtual appliance, i.e. an operating system image with all the libraries and code required to run the component. As the component has to conform to some standard in order to be composable with other components, we define several types of interfaces which a component can have. The interface (port) types which we have identified include: (1) web service interfaces using SOAP and WSDL; (2) REST interfaces available using HTTP; (3) queue interfaces for accessing e.g. Amazon SQS or AMQP services; and (4) data ports for accessing cloud storage such as S3 object store or MongoDB store.

The list of port types is intended to be open, since more port types (e.g. streaming ports or different data source types) may become necessary depending on the application requirements. The common feature of these interfaces is that they can be dynamically configured at runtime in the process of application composition (for instance, Web service endpoints can be provided and queue identifiers assigned).

The composition process is based on the dependency injection design pattern so that the component itself only has to provide the basic functionality allowing it to be configured. This functionality has to be provided by a simple REST API which includes operations for: listing component ports (introspection), setting component ports (composition, dependency injection), initiating the processing of a component (lifecycle management). This minimal set of API operations in the form of a REST interface allows the component to be developed using virtually any programming language and platform, from lightweight scripting languages (such as Python or Ruby) to more sophisticated solutions (such as Java-based ESB or .NET platforms).

Such a simple component model facilitates building complex applications which take advantage of the capabilities offered by cloud services. Fig. 1 shows a simplified processing application representing a master-slave computation. The master and the worker are two component instances which communicate using queue interfaces to exchange computing tasks and their results. Additionally, the master provides a Web service interface (MasterPort) which can be used to initiate computation by providing input data. The master-worker application demonstrates several features of the proposed approach. First of all, the components provide standard interfaces so they can be dynamically connected and interacted using standard protocols. Moreover, by using queue ports, the components are not tightly coupled to each other: actually, they are not aware of the existence of their partners. It is thus possible to e.g. add more worker components and distribute the load among them. Moreover, it might be possible to dynamically adjust the number of workers based on the load or queue length.

We are convinced that such a minimal but powerful component model provides a convenient abstraction for programming and execution of cloud-based applications. Although heavyweight in terms of packaging (as it requires components to consist of full virtual machine images), it remains lightweight and flexible in terms of programming interfaces and APIs.

```
1  require 'rubygems'
2  require 'cloud_obj'
3
4  count = 16
5  input_data = '........................'
6
7  master = CloudObj.new('test.sqs.Master')
8  master.set_port('worker','queue_worker')
9  master.set_port('result','queue_result')
10
11 master.compute(input_data)
12 master.start
13
14 workers = CloudObj.new('test.sqs.Worker', count)
15 workers.set_port('worker','queue_worker')
16 workers.set_port('result','queue_result')
17
18 workers.start
19
20 ...# wait for results from the master
21 puts master.status
22 puts master.result
23
24 workers.destroy
25 master.destroy
```

Figure 2: Script code for composition of sample master-worker components.

4. High-level API and prototype based on Ruby

The proposed component model, in addition to defining component interfaces (based on REST, see Section 3.2), should also provide an interface enabling clients to interact with the framework. The client interface in the form of API should provide the basic functionality to create components, connect their ports and invoke operations on component interfaces. The interface should be positioned on a high level of abstraction, allowing users to specify component classes and interface names and not to deal with low-level cloud infrastructure details.

We illustrate the usage of the proposed high-level API on the example shown in in Fig. 2, which corresponds to the right-hand diagram in Fig. 1. The API is an extension of our previous work on grid operations [24] and is provided in the form of a Ruby gem (lines 1 and 2). Upon setting configuration and input parameters we create an instance of the component on the cloud (line 7) by providing a component class name ('test.sqs.Master'). The class name is mapped to the actual virtual machine image by the registry so that the script does not deal with such details. In lines 8–9 ports of the components are connected to two queues. Line 11 invokes an operation on the component using its REST web service; subsequently (lines 14–18) a similar set of operations is run on a collection of worker components (the number of worker instances is parameterized). The components can be accessed using REST methods to check their status or results (lines 20-22). Finally, the components can be destroyed which results in termination of virtual machine instances.

The current prototype of the framework for development of components and client tools is written in Ruby. The prototype supports Amazon EC2 public cloud and Eucalyptus private cloud framework through the use of RightScale Ruby libraries. The framework contains a simple registry with descriptions of available component classes and maps them to actual images stored on Amazon or local Eucalyptus installations. It supports delegation of security credentials (access and secret keys) to component instances which are dynamically passed to the created component instances. These, in turn, can use other cloud-based services. There is support for hybrid cloud installations, e.g. computing instances created on the private cloud can still access public S3 storage or SQS queues from Amazon.

The prototype includes a framework for development of components as Ruby classes which provide a lightweight method for wrapping e.g. legacy applications as components. We support the following interface types: (1) web service interfaces using SOAP and WSDL – the prototype is based on the action web service Ruby library; (2) REST operations using HTTP based on the Sinatra framework. The same framework is used to provide component operations such as dependency injection via set_port operations. Other component methods are invoked based on naming conventions using Ruby internal dynamic dispatch capabilities; (3) queue ports supporting interaction with Amazon SQS queues; and (4) data ports supporting access to the Amazon S3 object store.

The Ruby framework has been prepared in a way which facilitates the process of component development and testing. It is possible to instantiate components locally as plain Ruby objects (e.g. on the developer's laptop) without access to the cloud, virtualization and Web service stack. Component classes can then be tested using scripts very similar to the one shown in Fig. 2 and the methods are then invoked locally. This experimental feature is very useful as otherwise the cycle of development, deployment and testing on the cloud would be very time-consuming. The prototype also provides serialization of method parameters based on JSON and access to application logs via HTTP. Since it is wrapped as a Ruby gem, client tools can be easily used from any scripting framework, notably including our GridSpace virtual laboratory which provides interactive Web-based programming and execution environment supporting exploratory programming [25].

5. Experimental evaluation

The objectives of experiments were: (1) measurement of performance of a private cloud installation acting as a component container, with focus on component instantiation time; (2) measurement of overhead incurred by paravirtualization for a specific application; and (3) overall performance and scalability measurements of the whole application to identify bottlenecks and possible optimization strategies.

We used the hybrid cloud environment which consisted of services provided by the public Amazon cloud (S3 and SQS), with computing nodes residing on the Intel cluster running Eucalyptus software. For our test application we selected the similarity search program from the FASTA [6] package.

5.1. Similarity search application

Searching for similarities between sequences of nucleotides or aminoacids is a very important task in bioinformatics. It may be used to locate similar genes or pseudogenes among or within species. The sequence libraries used for typical experiments can be quite large: for example, the human mRNA dataset comprises 46572 sequences totalling 128 megabytes. The most accurate (and computationally demanding) similarity search application is the Smith-Waterman algorithm. It is able to align two sequences of length m and n in $O(mn)$ time. It is available in the FASTA package and the implementation uses Intel SSE extensions to considerably speed up the computing. Nevertheless, similarity searches against a large sequence library may consume up to hours of CPU time. These searches have to be often repeatedly iterated over multiple query sequences which produces heavy workloads of a parameter-study type, but with a high variance of computing time per task, due to large differences in sizes between sequences.

In our experiments we ran the *ssearch36* program from the FASTA package and compared human mRNA sequences by aligning each of the query sequences with the entire library. In addition to the whole mRNA dataset we prepared a smaller subset comprising 3855 sequences and totalling 6.5 megabytes in size.

The FASTA package was wrapped as a virtual appliance component with our Ruby-based framework. The component was configured to read input identifiers from the queue port, then to fetch input data from the object store (using the assigned data port), run the computation and store similarity scores in the object store.

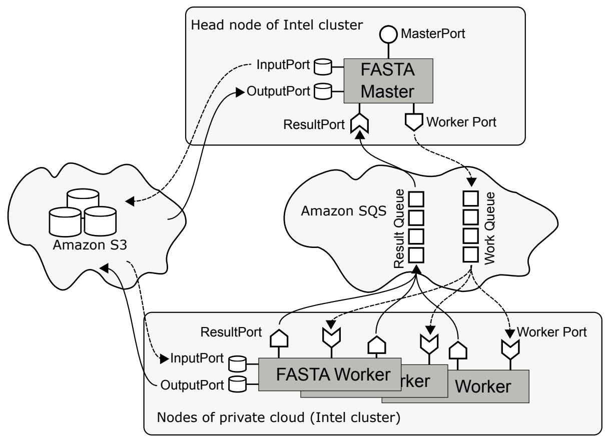
5.2. Experimental setup

Our performance tests were run on a hybrid cloud environment consisting of the following resources:

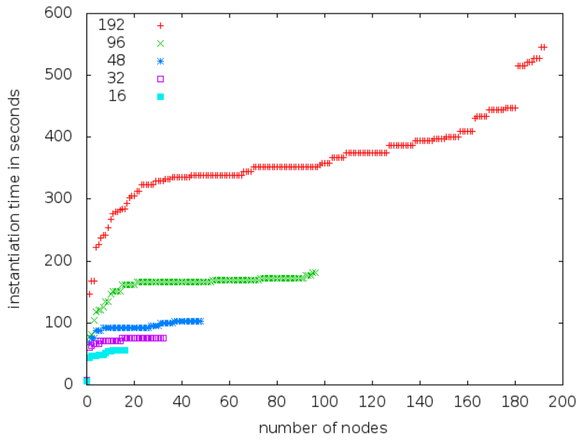
- Private cloud installed on an Intel cluster of 16 nodes, 12 cores each. The computing nodes were running the CentOS Linux 5.5 operating system with Eucalyptus 2.0 cloud software, configured with Xen 3.1.2 hypervisor using paravirtualization. Worker nodes were used to deploy worker components which were virtual machines running Ubuntu Linux with FASTA software wrapped as components using our Ruby-based framework (see Section 4). The virtual machine disk image size was 1GB.

- The head node of the cluster running Linux was used to create the Master component. It was instantiated locally as a plain Ruby object (since it is not computationally intensive, it does not require a dedicated (virtual) machine).

- Amazon SQS was used to provide queues which connect component queue ports. These ports were used to send identifiers of sequences to be computed.



(a) The hybrid cloud environment consisting of a private cloud installed on an Intel cluster running Eucalyptus along with publicly-available Amazon S3 and SQS cloud services.



(b) Relation between the number of virtual machine nodes (components) and time required for their instantiation in five cycles with different numbers of requested nodes - 16,32,48,96 and 192 respectively.

Figure 3: Hybrid cloud setup and performance

- The Amazon S3 storage service was used to store input sequences and output files. The sizes of these files ranged from a few kilobytes to several megabytes, so they could not be passed directly within the SQS message payload.

Tests were performed on the Urbana-4 Intel cluster. This system is based on 16 SR1625UR nodes. Each node is a 1U server based on Intel S5520UR motherboard and utilizing 2 Intel Xeon X5670 CPUs. The Intel Xeon X5600 family is the first generation of six-core Intel CPUs dedicated to dual-socket servers it is also the first Intel six-core processor with an integrated memory controller. Intel Xeon X5670 is a 32 nm six-core monolithic die with 12MB of L3 cache, 3-channel integrated memory controller and integrated Quick Path Interconnect interface. The Intel Xeon X5670 processor is a 6-core 95W processor clocked at 2.93GHz. It boasts many enhancements and new mechanisms which improve both overall performance and per-watt performance. Every node was equipped with 48GB 1333MHz DDR3 RAM and utilized a 450GB SAS 15K drive HDD. The Urbana-4 cluster is compliant with Intel Cluster Ready version 1.5 (EM64T) and the system has 2250 GFLOPS of theoretical peak performance with an actual LINPACK performance of 2047 GFLOPS (91% efficiency).

5.3. Private cloud performance

The goal of these tests was to measure the component instantiation time, which, in our case, is the startup time of the virtual machine instance on the cloud. For the purpose of running a realistic scenario, we deployed (on our private cloud) a virtual machine image component with minimal Ubuntu Linux 9.10, Eucalyptus tools, Ruby gems required to access the cloud, and the FASTA computing program wrapped as a worker component. The size of the image was 1 gigabyte and the disk space assigned to the virtual machine was 2 gigabytes. The tests were run on the Eucalyptus 2.0 installation, which caches the virtual machine images on the computing nodes instead of fetching them each time from central storage. Thus, the measured startup times included the creation of an empty virtual disk using the loopback device, copying the image from the cache and booting Ubuntu Linux.

The results are shown in Fig. 3(b). In this figure we present a combination of results for five series of runs. For each series we used a different number of requested nodes, i.e. virtual machines. This was achieved by issuing a request to deploy a given number of component instances: 16, 32, 48, 96 and 192 respectively. Results clearly demonstrate a strong positive correlation between the number of requested nodes and instantiation time of all nodes. Such behavior was expected as the larger number of requested VMs causes Eucalyptus to start more of them at the same time, resulting in higher load (especially I/O). We can also see that deployment of 192 machines takes approximately 9 minutes, which gives an average more then 20 machines per minute, while the minimum observed times were on the order of 5 minutes. Naturally, the machines do not appear linearly, as the process of copying virtual machine images runs concurrently but is limited by I/O (hard disk). This results in much slower instantiation at the beginning (when images are being created) with speed increasing later on (when some VMs are booting but I/O is more or less idle). During tests we initially observed some problems with Eucalyptus performance in the "system" networking mode, as not all the machines properly obtained IP addresses and were registered by Eucalyptus as running. This is reflected in Fig. 3(b) as a slowdown on the trailing end of longer series (involving 96 and 192 VMs). The problem only appears for a large number of VMs as they use many IPs (1 IP per VM) and generate heavy network traffic which can apparently confuse the Eucalyptus VM IP detection system.

The results obtained on our private cloud testbed can be compared e.g. to typical scheduling delays of batch processing systems on clusters. In the case of the Zeus cluster at ACC Cyfronet AGH, the Maui PBS scheduler is executed at intervals of ca. 200 seconds, which, on average, incurs a 100-second wait time for access to available free resources. As we can see, the wait times for the component deployment using our private cloud are on the same order of magnitude, resulting in similar user experience.

5.4. Virtualization overhead

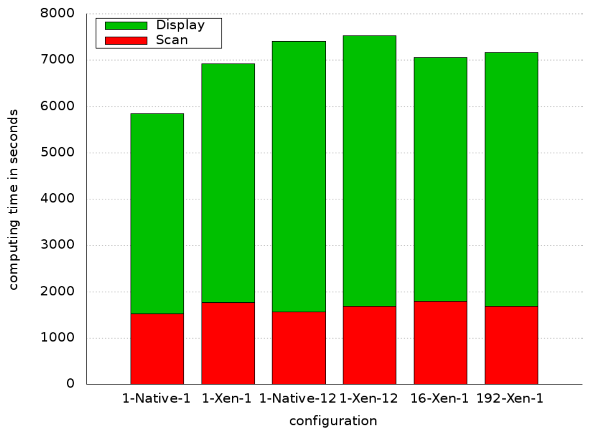
The experiments were prepared in order to determine the relation between overhead and granularity of components, or, more specifically, decide (for single-threaded program) whether we should wrap it as a single component and run each component (virtual machine) per each core of the cluster, or create multicore virtual machines and run multiple processes per each core within a component. For this purpose we prepared 6 configurations:

- 1-Native-1: a single 12-core native Linux node, running 1 *ssearch36* process;
- 1-Xen-1: a single node with a single 12-core Xen virtual machine (Linux guest), running 1 *ssearch36* process;
- 1-Native-12: a single 12-core native Linux node, running 12 processes of *ssearch36* program;
- 1-Xen-12: a single node with a single 12-core Xen virtual machine, running 12 *ssearch36* processes;
- 16-Xen-1: 16 nodes, each hosting a single 1-core Xen virtual machine, running 1 *ssearch36* process;
- 192-Xen-1: 16 nodes, each hosting 12 of 1-core Xen virtual machines, each running 1 *ssearch36* process;

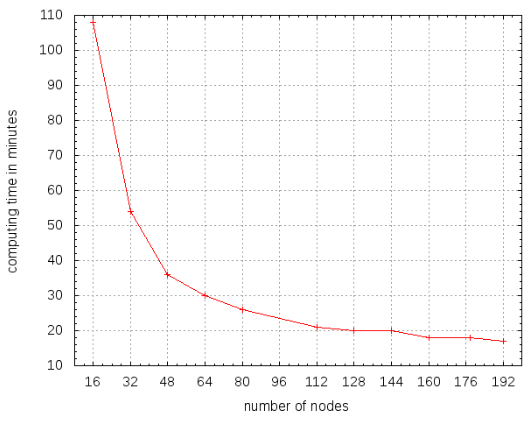
The results of these tests are shown in Fig. 4(a) and present the duration of computing phases (scan and display time) as returned by the *ssearch36* program, summed over a set of sample sequences. The first four configurations were tested on an isolated system while the remaining ones used our component framework.

As we can see, in the isolated scenario the Xen overhead when running the single program on a single core can reach 20% of display time and 15% of scan time. However, in more realistic configurations when all cores are used to run the program concurrently, the differences between Xen and native performance are smaller (below 2% of total time). This can be attributed to the fact that the application performance is mainly CPU-bound yet some time is needed for I/O to store the results on the disk. When all the cores of a node are used by processes (in both Xen and the native case) they have to compete for I/O and memory access. This results in longer computing times, making the overhead introduced by Xen less significant. In the case of the last two columns the computing times are shorter since the input data were obtained by worker components from the S3 object store, not directly from a file, which presumably resulted in more even load distribution and minimized I/O competition between processes.

The conclusions are promising. First, the overhead of Xen in scenarios when many cores of the machine are used is considerably lower than in the case of isolated single-core programs and in our specific case remains under 2%. Moreover, instantiating 12 virtual machine components on a 12-core node results in performance comparable to running 12 processes on a native system. This means that, for CPU-intensive applications, the component approach does not introduce significant overhead.



(a) Cumulative computing time for both stages of running the *ssearch36* program (scan and display stages)



(b) Computing times for application run on a set of 3855 query sequences and 3855 library sequences. Number of nodes means the number of virtual machine instances, each using a single processor core.

Figure 4: Application performance measurements

5.5. Application scalability

The application (Fig. 3(a)) consisted of the two phases: (1) input preparation and (2) computing. In phase (1) the master component retrieved input data from the file, placed the sequences in the object store and inserted their identifies as well as run parameters in the queue. In phase (2) the worker components began processing the sequences and stored results in the output object store. We observed that storing input data in S3 can be a time-consuming process, while retrieval is (not surprisingly) considerably faster. The same holds for the performance of the SQS queue. For these reasons, if the cost of a CPU-hour is relevant (as in the case of the Amazon cloud) it is reasonable to prepare input data and tasks in the queue first, and start the worker components no sooner than once input is ready.

The computing times obtained on our private cloud versus the number of nodes are shown in Fig. 4(b). The number of nodes is equal to the number of components, i.e. single-core virtual machines, each occupying a single core in the cluster. The plot only shows the duration of the computing phase, including the time needed by components to receive the task id from the SQS (using the queue port), fetch the input data from the S3 object store (using the data port), process the data using the *ssearch36* program and store output in S3. It can be observed that the speedup is regular but not linear and that parallel efficiency drops from ca. 0.9 (for 64 nodes) to ca. 0.5 (for 192 nodes). This behaviour is natural given significant wide-area network latencies and can be potentially further optimized by either moving data closer to computation (using local storage) or moving computation closer to the data (running tests on EC2 instances) – depending on the scenario requirements and cost analysis.

6. Conclusions and future work

In this paper we presented a component-based approach to computational science applications on cloud infrastructures. Based on our previous experience with grid component models and scientific applications, we discussed the advantages and costs associated with using a component-based approach along with cloud resources. We proposed the principles of a lightweight component model and prepared a prototype solution based on a high-level scripting approach. The main principle is to consider virtual machines running on the cloud as components and the cloud as a large-scale distributed container for hosting these components. Our prototype framework, which provides a high-level API, consists of a set of Ruby-based libraries and was tested with a real application in a hybrid cloud environment.

The experiments demonstrated that the component-based approach can be effective in building and deploying computational applications on the cloud. The measured overhead incurred by the component startup time as well as the performance penalty caused by virtualization are negligible or at least reasonable, in our opinion not outweighing the benefits of the possibility of creating computationally-intensive applications on demand.

We consider this work as a preliminary insight into the potential possibilities of using our component-based approach to solving computational science problems on the emerging cloud infrastructures. In the future we will extend the framework to support more complex application scenarios, as well as parallel components by e.g. exploiting the HPC cluster instances offered by Amazon and cluster-on-demand features of private cloud solutions such as those available in Nimbus [13]. Better support for cloud heterogeneity and security is also worth investigating. These new features should aim at providing a simple-to-use platform-as-a-service solution for computational science applications. Last, but not least, optimization methods based on performance and cost metrics will be of great importance, since the cloud introduces a business model which has to be taken into account by the computational science community.

Acknowledgments The authors are grateful to colleagues from PL-Grid, UrbanFlood and VPH-Share projects. We thank Krzysztof Sarapata from CMUJ for providing information about FASTA. We also thank Tomasz Gubała, Patryk Lasoń and Piotr Nowakowski from ACC Cyfronet AGH for valubale discussions. We gratefully acknowledge the help and support of Jamie Wilcox and Victor Gamayunov from Intel EMEA Technical Marketing HPC Lab. This work was supported by UDA – POKL.04.01.01-00-367/08-00 grant from AGH. Access to the Amazon EC2 cloud was supported by an AWS in Education grant.

References

[1] VPH-Share Project, Virtual physiological human: Sharing for healthcare - a research environment (2011-2015).
[2] SARA, Hpc cloud, `https://grid.sara.nl/wiki/index.php/Using_the_HPC_Cloud` (2011).
[3] M. Malawski, D. Kurzyniec, V. Sunderam, MOCCA – towards a distributed CCA framework for metacomputing, in: IPDPS 2005, IEEE Computer Society, 2005.
[4] R. Armstrong, et al., The CCA component model for high-performance scientific computing, Concurrency and Computation : Practice and Experience 18 (2) (2006) 215–229. doi:http://dx.doi.org/10.1002/cpe.v18:2.
[5] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, Gcm: a grid extension to fractal for autonomous distributed components, Annales des Télécommunications 64 (1-2) (2009) 5–24.
[6] W. R. Pearson, D. J. Lipman, Improved tools for biological sequence comparison., Proc Natl Acad Sci U S A 85 (8) (1988) 2444–2448.
[7] U. Schwiegelshohn, R. M. Badia, M. Bubak, M. Danelutto, S. Dustdar, F. Gagliardi, A. Geiger, L. Hluchy, D. Kranzlmller, E. Laure, T. Priol, A. Reinefeld, M. Resch, A. Reuter, O. Rienhoff, T. Rter, P. Sloot, D. Talia, K. Ullmann, R. Yahyapour, G. von Voigt, Perspectives on grid computing, Future Generation Computer Systems 26 (8) (2010) 1104 – 1115.
[8] J. T. Dudley, A. J. Butte, In silico research in the era of cloud computing, Nature Biotechnology 28 (11) (2010) 1181–1185.
[9] E. Deelman, Grids and clouds: Making workflow applications work in heterogeneous distributed environments, International Journal of High Performance Computing Applications 24 (3) (2010) 284–298.
[10] D. Thain, C. Moretti, Abstractions for Cloud Computing with Condor, CRC Press, 2010, pp. 153–171.
[11] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[12] M. C. Schatz, CloudBurst: highly sensitive read mapping with MapReduce, Bioinformatics 25 (11) (2009) 1363–1369.
[13] K. Keahey, M. Tsugawa, A. Matsunaga, J. Fortes, Sky Computing, Internet Computing, IEEE 13 (5) (2009) 43–51.
[14] C. Vecchiola, S. Pandey, R. Buyya, High-performance cloud computing: A view of scientific applicationsarXiv:0910.1979.
URL `http://arxiv.org/abs/0910.1979`
[15] B. Adler, RightScale Grid: Grid Computing Applications in the Cloud, Tech. rep., RightScale Inc. (2010).
[16] Object Management Group, Inc., CORBA Component Model, v4.0, `http://www.omg.org/technology/documents/formal/components.htm` (2006).
[17] H.-L. Bouziane, C. Pérez, T. Priol, A software component model with spatial and temporal compositions for grid infrastructures, in: E. Luque, T. Margalef, D. Benitez (Eds.), Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings, Vol. 5168 of Lecture Notes in Computer Science, Springer, 2008, pp. 698–708.
[18] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, W. Yu, Extending Xen with Intel Virtualization Technology, Intel Technology Journal 10 (03) (2006) 193–204.
[19] J. P. Casazza, M. Greenfield, K. Shi, Redefining Server Performance Characterization for Virtualization Benchmarking, Intel Technology Journal 10 (03) (2006) 243–252.
[20] E. Walker, Benchmarking Amazon EC2 for high-performance scientific computing, LOGIN 33 (5) (2008) 18–23.
[21] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, D. Gannon, Cloud technologies for bioinformatics applications, in: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS '09, ACM, New York, NY, USA, 2009.
[22] Amazon.com, Elastic compute cloud (EC2), `aws.amazon.com/ec2` (2011).
[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The Eucalyptus open-source cloud-computing system, in: 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), Vol. 0 of CCGRID '09, IEEE, Washington, DC, USA, 2009, pp. 124–131.
[24] M. Malawski, T. Bartyński, M. Bubak, Invocation of operations from script-based grid applications, Future Gener. Comput. Syst. 26 (1) (2010) 138–146.
[25] E. Ciepiela, D. Harezlak, J. Kocot, T. Bartynski, M. Kasztelnik, P. Nowakowski, T. Gubała, M. Malawski, M. Bubak, Exploratory Programming in the Virtual Laboratory, in: Proceedings of the International Multiconference on Computer Science and Information Technology, Wisla, Poland, 2010, pp. 621–628.