Journal of Big Data

RESEARCH Open Access

Data-aware optimization of bioinformatics workflows in hybrid clouds

Athanassios M. Kintsakis*, Fotis E. Psomopoulos and Pericles A. Mitkas

*Correspondence: akintsakis@issel.ee.auth.gr
Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

Abstract

Life Sciences have been established and widely accepted as a foremost Big Data discipline; as such they are a constant source of the most computationally challenging problems. In order to provide efficient solutions, the community is turning towards scalable approaches such as the utilization of cloud resources in addition to any existing local computational infrastructures. Although bioinformatics workflows are generally amenable to parallelization, the challenges involved are however not only computationally, but also data intensive. In this paper we propose a data management methodology for achieving parallelism in bioinformatics workflows, while simultaneously minimizing data-interdependent file transfers. We combine our methodology with a novel two-stage scheduling approach capable of performing load estimation and balancing across and within heterogeneous distributed computational resources. Beyond an exhaustive experimentation regime to validate the scalability and speed-up of our approach, we compare it against a state-of-the-art high performance computing framework and showcase its time and cost advantages.

Keywords: Cloud computing, Component-based workflows, Bioinformatics, Big data management, Hybrid cloud, Comparative genomics

Introduction

There is no doubt that Life Sciences have been firmly established as a Big Data science discipline, largely due to the high-throughput sequencers that are widely available and extensively utilized in research. However, when it comes to tools for analyzing and interpreting big bio-data, the research community has always been one step behind the actual acquisition and production methods. Although the amount of data currently available is considered vast, the existing methods and extensively used techniques can only hint at the knowledge that can be potentially extracted and consequently applied for addressing a plethora of key issues, ranging from personalized healthcare and drug design to sustainable agriculture, food production and nutrition, and environmental protection. Researchers in genomics, medicine and other life sciences are using big data to tackle fundamental issues, but actual data management and processing requires more networking and computing power [14]. Big data is indeed one of today's hottest concepts, but it can be misleading. The name itself suggests mountains of data, but that's just the start. Overall, big data consists of three v's: volume of data, velocity of processing the data, and

Springer Open

© The Author(s) 2016. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

variability of data sources. These are the key features of information that require particular tools and methodologies to efficiently address them.

The main issue with dealing with big data is the constantly increasing demands for both computational resources as well as storage facilities. This in turn, has led to the rise of large-scale high performance computing (HPC) models, such as cluster, grid and cloud computing. Cloud computing can be defined as a potentially high performance computing environment consisting of a number of virtual machines (VMs) with the ability to dynamically scale resources up and down according to the computational requirements. This computational paradigm has become a popular choice for researchers that require a flexible, pay-as-you-go approach to acquiring computational resources that can accompany their local computational infrastructure. The combination of public and privately owned clouds defines a hybrid cloud, i.e. an emerging form of a distributed computing environment.

From this perspective, optimizing the execution of data-intensive bioinformatics workflows in hybrid clouds is an interesting problem. Generally speaking, a workflow can be described as the execution of a sequence of concurrent processing steps, or else computational processes, the order of which is determined by data interdependencies as well as the target outcome. In a data-intensive workflow, data and metadata, either temporary or persistent, are created and read at a high rate. Of course, a workflow can be both data and computationally intensive and the two are often found together in bioinformatics workflows. In such workflows, when scheduling tasks to distributed resources, the data transfers between tasks are not a negligible factor and may comprise a significant portion of the total execution time and cost. A high level of data transfers can quickly overwhelm the storage and network throughput of cloud environments, which is usually on the order of 10–20 MiB/s [6], while also saturating the bandwidth of local computational infrastructures and leading to starvation of resources to other users and processes.

It is well known that a high level of parallelization can be achieved in a plethora of bioinformatics workflows by fragmenting the input of individual processes into chunks and processing them independently, thus achieving parallelism in an embarrassingly parallel way. This is the case in most evolutionary investigation, comparative genomics and NGS data analysis workflows. This fact can be largely taken advantage of in order to achieve parallelism by existing workflow management approaches emphasizing parallelization. The disadvantage of this approach however is that it creates significant data interdependencies, which in turn lead to data transfers that can severely degrade performance and increase overall costs.

In this work, we investigate the problem of optimizing the parallel execution of data-intensive bioinformatics workflows in hybrid cloud environments. Our motivation is to achieve better time and cost efficiency than existing approaches by minimizing file transfers in highly parallelizable data-intensive bioinformatics workflows. The main contributions of this paper are twofold; (a) We propose a novel data management paradigm for achieving parallelism in bioinformatics workflows while simultaneously minimizing data-interdependency file transfers, and (b) based on our data management paradigm, we introduce a 2-stage scheduling approach balancing the trade-off between parallelization opportunities and minimizing file transfers when mapping the execution of bioinformatics workflows into a set of heterogeneous distributed computational resources

comprising a hybrid cloud. Finally, in order to validate and showcase the time and cost efficiency of our approach, we compare our performance with Swift, one of the most widely used and state-of-the-art high performance workflow execution frameworks.

The rest of the paper is organized as follows: a review of the state-of-the-art on workflow management systems and frameworks in general and in the field of bioinformatics in particular is presented in "Related work" section. "Methods" section outlines the general characteristics and operating principles of our approach. "Use case study" section briefly presents the driving use case that involves the construction of phylogenetic profiles from protein homology data. "Results and discussion" section provides the results obtained through rigorous experimentation, in order to evaluate the scalability and efficiency as well as the performance of our approach when compared against a high performance framework. Finally, concluding remarks and directions for future work are given in "Conclusions and future work" section.

Related work

The aforementioned advantages of cloud computing have led to its widespread adoption in the field of bioinformatics. Initial works were mostly addressed on tackling specific, highly computationally intensive problems that outstretched the capabilities of local infrastructures. As the analyses became more complex and incorporated an increasing number of modules, several tools and frameworks appeared that aimed to streamline computations and automate workflows. The field of bioinformatics has also sparked the interest of many domain agnostic workflow management systems, some of the most prolific applications of which were bioinformatics workflows, thus leading to the development of pre-configured customized versions specifically for bioinformatics workflows [34].

Notable works addressing well-known bottlenecks in computationally expensive pipelines, the most characteristic of which are Next Generation Sequencing (NGS) data analysis and whole genome assembling (WGA) include [18], Rainbow [9], CloudMap [29], CloudBurst [40], SURPI [31] and RSD-Cloud [45]. These works, although highly successful, lack a general approach as they are problem specific and are often difficult to setup, configure, maintain and most importantly integrate within a pipeline, when considering the experience of a non-expert life sciences researcher.

Tools and frameworks aiming to streamline computations and automate standard analysis bioinformatics workflows include Galaxy [17], Bioconductor [16], EMBOSS [39] and Bioperl [43]. Notable examples of bioinformatics workflow execution in the cloud include [11, 33] and an interesting review on bioinformatics workflow optimization in the cloud can be found in [15]. In the past few years, there is a significant trend in integrating existing tools into unified platforms featuring an abundance of ready to use tools, with particular emphasis on ease of deployment and efficient use of resources of the cloud. A platform based approach is adopted by CloudMan [1], Mercury [38], CLoVR [3], Cloud BioLinux [22] and others [24, 32, 42, 44]. Most of these works are addressing the usability and user friendly aspect of executing bioinformatics workflows, while some of them also support the use of distributed computational resources. However, they largely ignore the underlying data characteristics of the workflow and do not perform any data-aware optimizations.

Existing domain agnostic workflow management systems including Taverna [48], Swift [49], Condor DAGMan [23], Pegasus [13], Kepler [26] and KNIME [5] are capable of also addressing bioinformatics workflows. A comprehensive review of the aspects of parallel workflow execution along with parallelization in scientific workflow managements systems can be found in [8]. Taverna, KNIME and Kepler mainly focus on usability by providing a graphical workflow building interface while offering limited to non-existent support, in their basic distribution, for use of distributed computational resources. On the other side, Swift, Condor DAGMan and Pegasus are mainly inclined over accomplishing parallelization on both local and distributed resources. Although largely successful in achieving parallelization, their scheduling policies are non data-aware and do not address minimizing file transfers between sites.

Workflow management systems like Pegasus, Swift and Spark can utilize shared file systems like Hadoop and Google Cloud Storage. The existence of a high performance shared file system can be beneficial in a data intensive worfklow as data can be transferred directly between sites and not staged back and forth from the main site. However, the advantages of a shared file system can be outmatched by a data-aware scheduling policy which aims to minimize the necessity of file transfers to begin with. Furthermore, the existence of a shared file system is often prohibitive in hybrid clouds comprising of persistent local computational infrastructures and temporarily provisioned resources in the cloud. Beyond the significant user effort and expertise required in setting up a shared file system, one of the main technical reasons for this situation is that elevated user operating system privileges are required for this operation, which are not usually granted in local infrastructures.

A Hadoop MapReduce [12] approach is capable of using data locality for efficient task scheduling. However, its advantages become apparent in a persistent environment where the file system is used for long term storage purposes. In the case of temporarily cloud provisioned virtual machines, the file system is not expected to exist either prior or following the execution of the workflow and consequently all input data are loaded at the beginning of the workflow. There is no guarantee that all the required data for a specific task will be placed in the same computational site and even if that were the case, no prior load balancing mechanism exists for assigning all the data required for each task to computational sites while taking into account the computational resources of the site and the computational burden of the task. Additionally, a MapReduce approach requires re-implementation of many existing bioinformatics tools which is not only impractical but also unable to keep up to date with the vanilla and standardized versions.

Finally, it is important to note that none of the aforementioned related work clearly addresses the problem of applying a data-aware optimization methodology when executing data-intensive bioinformatics workflows in hybrid cloud environments. It is exactly this problem that we address in this work, by applying a data organization methodology coupled with a novel scheduling approach.

Methods

In this section we introduce the operating principles and the underlying characteristics of the data management and scheduling policy comprising our methodology.

Data management policy

The fact that data parallelism can be achieved in bioinformatics workflows has largely been taken advantage of in order to accelerate workflow execution. Data parallelism involves fragmenting the input into chunks which are then processed independently. For certain tasks of bioinformatics workflows, such as sequence alignment and mapping of short reads which are also incidentally some of the most computationally expensive processes, this approach can allow for a very high degree of parallelism in multiprocessor architectures and distributed computing environments. However, prior to proceeding to the next step, data consistency requires that the output of the independently processed chunks be recombined. In a distributed computing environment, where the data is located on multiple sites, this approach creates significant data interdependency issues as data needs to be transferred from multiple sites in order to be recombined, allowing the analysis to proceed to the next step. The same problem is not evident in a multiprocessor architecture, as the data exists within the same physical machine.

A sensible approach to satisfying data interdependencies with the purpose of minimizing, or even eliminating unnecessary file transfers would be to stage all fragments whose output must be recombined on the same site. Following that, the next step, responsible for processing the recombined output, can also be completed on the same site, and then the next step, that will operate on the output of the previous, also on the same site, further advancing this course until it is no longer viable. It is becoming apparent that this is a recursive process that takes into account the anticipated data dependencies of the analysis. In this way, segments of the original workflow are partitioned into workflow ensembles (workflows of similar structure but differing in their input data) that have no data interdependencies and can then be executed independently in an approach reminiscent of a bag-of-tasks. Undoubtedly, not all steps included in a workflow can be managed this way, but a certain number can, often also being the most computationally and data intensive.

Instead of fragmenting the input of data parallelizable tasks into chunks arbitrarily, we propose fragmenting into chunks that can also sustain the data dependencies of a number of subsequent steps in the analysis. Future tasks operating on the same data can be grouped back-to-back into forming a pipeline. To accomplish the aforementioned, we model the data input space as comprising of Instances. An Instance (*Inst*) is a single data entry, the simplest form data can exist independently. An example of an *Inst* would be a single protein sequence in a .fasta file. Instances are then organized into organization units (*OU*), which are sets of instances that satisfy the data dependencies of one or more tasks. The definition of an *OU* is a set of *Insts* that can satisfy the data dependencies of a number of consecutive tasks, thus allowing the formation of an *OU* pipeline.

However, before attempting to directly analyze the data involved, a key step is to preprocess the data instances in order to allow for a structured optimization of the downstream analysis process. A common occurrence in managing big data is the fact that their internal organization is dependent on its specific source. Our data organization model is applied through a preprocessing step that restructures the initial data organization into sets of *Insts* and *OUs* in a way reminiscent of a base transformation.

The process involves identifying *Inst*s in the input data, and grouping them together into *OUs* according to workflow data interdependencies. An identifier is constructed for

each *Inst* that also includes the *OU* it belongs to. The identifier is permanently attached to the respective data and therefore is preserved indefinitely. The initial integrity of the input data is guaranteed to be preserved during workflow execution, thus ensuring the accessibility to this information in later stages of the analysis and allowing for the recombination process. The identifier construction process is defined as follows.

Definition 1 Each *OU* is a set that initially contains a variable number (denoted by $n, k, l, ...$) of instances $Inst_j$ where $j = [1, n]$. The internal order of instances within an *OU* is preserved as the index assigned to each unique identifier $Inst_j$ (i.e. the order $1 < i < n$ of the instances) is reflected directly upon the constructed identifier. The total number of *m OUs* themselves are grouped into a set of *OUs* and are each assigned unique identifiers OU_i constructed in a semi-automated manner to better capture the semantic context of the defined *OUs*. Finally, the instance identifier, *InstID* consists of the concatenated OU_i and $Inst_j$ parameters, as shown below:

$$OUs = \{OU_0, OU_1, OU_2, \ldots, OU_m\} \tag{1}$$

$$\begin{aligned} OU_0 &= \{Inst_0, ..., Inst_n\},\ OU_1 = \{Inst_0, ..., Inst_k\}\ ... \\ OU_n &= \{Inst_0, ..., Inst_l\} \end{aligned} \tag{2}$$

$$InstID = F(OU_i, Inst_j) = OU_i_Inst_j \tag{3}$$

At some point, some or all the pipelines may converge in what usually is a non parallelizable merging procedure. This usually happens at the end of the workflow, or in intermediate stages, before a new set of *OU* pipelines is formed and the analysis continues onward.

Scheduling policy

It is obvious that this data organization approach although highly capable of minimizing data transfers, it severely limits the opportunities for parallelization, as each *OU* pipeline is processed in its entirety in a single site. In very small analyses where the number of *OUs* is less than the number of sites, obviously some sites will not be utilized, though this is a boundary case, unlikely to occur in real world analyses.

In a distributed computing environment, comprised of multiprocessor architecture computational sites, ideally each *OU* pipeline will be assigned to a single processor. Given that today's multiprocessor systems include a significant number of CPU cores, the number of *OU* pipelines must significantly exceed, by a factor of at least 10, the number of sites in order to achieve adequate utilization. Unfortunately, even that would prove inadequate, as the computational load of *OU* pipelines may vary significantly, thus requiring an even higher number of them in order to perform proper load balancing. It is apparent that this strategy would be fruitful only in analyses where the computational load significantly exceeds the processing capabilities of the available sites, spanning execution times into days or weeks. In solely data-intensive workflows, with no computationally intensive component, under-utilization of multiprocessor systems may not

become apparent as storage and network throughput are the limiting factors. Otherwise, it will most likely severely impact performance.

Evidently, a mechanism for achieving parallelism in the execution of an *OU* pipeline in a single site is required. Furthermore, in a heterogeneous environment of computational sites of varying processing power and *OU* pipelines of largely unequal computational loads, load balancing must be performed in order to map the *OU* pipelines into sites. To address these issues we propose a novel 2-stage scheduling approach which combines an external scheduler at stage 1 mapping the *OU* pipelines into sites and an internal to each site scheduler at stage 2 capable of achieving data and task parallelism when processing an *OU* pipeline.

External scheduler

The external scheduler is mainly concerned with performing load balancing of the *OU* pipelines across the set of computational resources. As both the *OU* pipelines and the computational sites are largely heterogeneous, the first step is performing an estimation regarding both the *OU* pipeline loads and the processing power of the sites. The second step, involves the utilization of the aforementioned estimations by the scheduling algorithm tasked with assigning the *OU* pipelines to the set of computational resources.

In order to perform an estimation of the load of an *OU* pipeline, a rough estimation could be made based on the size of the *OU* input. A simple approach would be to use the disk file size in MB but that would most likely be misleading. A more accurate estimation could be derived by counting the number of instances, this approach too however is also inadequate as the complexity cannot be directly assessed in this way. In fact, the computational load can only be estimated by taking into account the type of information presented by the file, which is specific to its file type. For example, given a .fasta file containing protein sequences, the most accurate approach for estimating the complexity of a sequence alignment procedure would be to count the number of bases, rather than count the number of instances. Fortunately, the number of distinct file types found in the most common bioinformatics workflows is small, and therefore we have created functions for each file type that can perform an estimation of the computational load that corresponds to them. We already support formats of .fasta, .fastq and plain ASCII (such as tab-delimited sequence similarity files) among others.

In order to better match the requirements of the data processing tasks to the available computational resources, the computational processing power of each site must also be assessed. This is accomplished by running a generic benchmark on each site which is actually a mini sample workflow that aims to estimate the performance of the site for similar workflows. The benchmarks we currently use are applicable on comparative genomics and pangenome analysis approaches, and measure the multithreaded performance of the site, taking into account its number of CPU cores. We also use the generic tool UnixBench [41] to benchmark the sites when no similar sample workflow is available.

The problem can now be modeled as one of scheduling independent tasks of unequal load to processors of unequal computational power. As these tasks are independent, they can be approached as a bag of tasks. Scheduling bag of tasks has been extensively studied and many algorithms exist, derived from heuristic [46], list scheduling [20] or metaheuristic optimization approaches [30]. In this work we utilize one of the highest performing algorithms, the FPLT (fastest processor largest task) algorithm. According to

FPLT, tasks are placed in descending order based on their computational load and each task, starting from the largest task, is assigned to the fastest available processor. Whenever a processor completes a task, it is then added to the list of available processors, the fastest of which is assigned the largest remaining task.

FPLT is a straightforward and lightweight algorithm, capable of outperforming other solutions most of the time [20] when all tasks are available from the start, as is the case here, without adding any computational burden. The disadvantage of FPLT is that when the computational power of processors is largely unequal, a processor might be assigned a task that severely exceeds its capabilities, thus delaying the makespan of the workflow. This usually happens when some processors are significantly slower than the average participating in the workflow.

The external scheduler initially performs an assessment of the type and load of the *OU* pipelines. It then determines the capabilities of the available sites in processing the pipelines by retrieving older targeted benchmarks or completing new on the fly. The *OU* pipelines are then submitted to the sites according to FPLT and job failures are handled by resubmission. The pseudocode of the external scheduler is presented in Algorithmic Box 1.

Algorithm 1 External Scheduler

```
1: ouPipelines[] = assessOuPipelines()
2: pipelineTypes[] = retrievePipelineTypes(ouPipelines)
3: sites[] = assessSites(pipelineTypes[])
4: sites[] = sortDescending(sites[])
5: ouPipelines[] = sortDescending(ouPipelines[])
6: while (size(ouPipelines) > 0) do
7:     if (size(sites) > 0) then
8:         assign(ouPipelines[0], sites[0])
9:         ouPipelines.remove(0)
10:        sites.remove(0)
11:    else
12:        waitForWorkers()
13:        worker, task, success = notifiedByWorker()
14:        if success then
15:            sites.add(worker)
16:        else
17:            sites.add(worker)
18:            ouPipelines.add(task)
19:            ouPipelines[] = sortDescending(ouPipelines[])
20:        end if
21:        sites[] = sortDescending(sites[])
22:    end if
23: end while
```

Internal scheduler

The internal scheduler is local to each site and is responsible for achieving data and task parallelism when processing an *OU* pipeline. Task parallelism involves executing independent tasks directly in parallel while data parallelism requires the identification of tasks whose input can be fragmented in chunks and processed in parallel. The second requires that such tasks are marked as suitable for fragmentation at the workflow description stage or maintaining a list of such tasks for automatic identification. Our approach supports both.

The internal scheduler automatically identifies the number of CPUs on the computational site and sets the number of simultaneous processing slots accordingly. It receives commands from the master and assigns them to threads in order to execute them in parallel. In case it receives a task where data parallelism is possible, it will fragment the input into individual chunks, or else subsets, and then launch threads in order to process them in parallel. A decision must be made on the number of fragments a task must be split to, which involves a trade off between process initialization overhead and load balancing between threads.

Given the widely accepted assumption that the CPU cores of a given site have the same computational capabilities, a simple solution would be to launch a number of threads equal to the machine's CPU count and divide the total number of input data, or else the instances, across them. This solution is in turn predicated on the assumption that the load assigned to a thread should directly correspond to the amount of data it has to process and as such is prone to variations. In our case however, as all required data exists within the same site, it is no longer desirable to distribute the data processing load among the threads in advance, as the data can be accessed by any thread at any time without any additional cost thus providing greater flexibility.

Therefore, when considering the situation within a single $site_l$, our approach can be defined by the process of splitting the superset of all m *Inst*s of the *OU* pipeline into k subsets of fixed size n. The number of subsets is given when dividing m by n.

$$Superset\{Inst_0, ..., Inst_m\} = Subset_1\{Inst_0, ..., Inst_n\} \cup ... \cup Subset_k\{Inst_0, ..., Inst_n\} \quad (4)$$

$$k = \frac{m}{n} \quad (5)$$

Each given $Subset_i$, is assigned to a thread responsible for completing the respective task. Initially the subsets are placed into a list in random order. Each thread attempts to process the next available subset and this continues recursively until all available subsets are exhausted. In order to synchronize this process and to ensure that no two threads process the same subset, a lock is established that monitors the list of subsets. Every time a thread attempts to obtain the next available subset it must first acquire the lock. If the lock is unavailable the thread is set to sleep in a waiting queue. If the lock is available, the thread acquires the requested subset and increases an internal counter that points to the next available subset. It then immediately releases the lock, an action that also wakes the first thread that may be present in the queue. The pseudocode describing the operation of the internal scheduler is presented in Algorithmic boxes 2 and 3.

Algorithm 2 Internal Scheduler

```
1: availableSlots = machineCpuCoreCount()
2: while (true) do
3:     if (availableSlots > 0) then
4:         command = waitForExternalScheduler()
5:         if (command == terminate) then
6:             break()
7:         else if (isMarkedForDataParallelism(command)) then
8:             subsets[] = fragmentInput()
9:             launchProcesses(command, subsets[], availableSlots)
10:            availableSlots = 0
11:        else
12:            launchProcesses(command, 1)
13:            availableSlots = availableSlots − 1
14:        end if
15:    else
16:        slotsReleased = waitForProcessToFinish()
17:        availableSlots = slotsReleased
18:    end if
19: end while
```

Algorithm 3 Launch data parallel processes

```
1: count = 0
2: while count < size(subsets[]) do
3:     lock()
4:     currentFile = subsets[count]
5:     count + +
6:     release()
7:     executeProcess(currentFile, command)
8: end while
```

As the probability of two threads completing the execution of a subset at exactly the same time is extremely low, the synchronization process has been proven experimentally to be very efficient, where most of the time there are no threads waiting on the queue. The average waiting time along with the time of acquiring and releasing the lock is usually minuscule. However, there is an important overhead that is associated with the initialization of the process that will complete the task. An accurate estimation of this overhead time is difficult to obtain as it is dependent on the actual processes being launched and the overall status of the operating system at any given time. We estimate this overhead to be around 300–1000 ms. A *totalDelay* parameter that indicates the estimated initialization delay involved in processing a given subset can be evaluated. This parameter can be constructed by multiplying the number k of subsets with the overhead parameter that reflects the average time wasted on synchronization and launching the respective processes, and dividing the result by the number of threads, as follows:

$$totalDelay = k * \frac{overhead}{threadCount} \tag{6}$$

It becomes apparent that minimizing the *totalDelay* time is equal to minimizing the number of subsets k. The minimum value of k is equal to the number of threads in which case the overhead penalty is suffered only once by each thread. However it is unwise

to set k equal to the number of threads as the risk of unequally distributing the data between the threads far outweighs the delay penalty.

We make the reasonable hypothesis that the execution times of chunks of fixed size $n = 1$ resemble a Log Normal distribution, which is typically encountered in processing times [4]. Our hypothesis was verified on an individual basis experimentally by running a BLAST procedure as presented in Fig. 1. BLAST is the most computationally intensive task of our use case study workflow presented in . Evidently, this does not apply to all tasks but is a reasonable hypothesis and a common observation in processing times.

A Log Normal distribution appears approximately like a skewed to the right, positive values only, normal distribution. This particular distribution presented in Fig. 1 allows us to estimate that only 8.2 % of the processing times were twice as large as the average processing time. Moreover, less than 0.5 % of the processing times were larger than five times the average processing time. It can easily be asserted that from a given set size and below, it is highly unlikely for many of the slower processing times to appear within it. However, it must be noted that this already low probability is further reduced by the fact that this is a boundary situation, to be encountered by the end of the workflow where other threads have terminated. After experimentation we have established that an empirical rule to practically eliminate the chance is to set n equal to 0.01 % of the number m of instances.

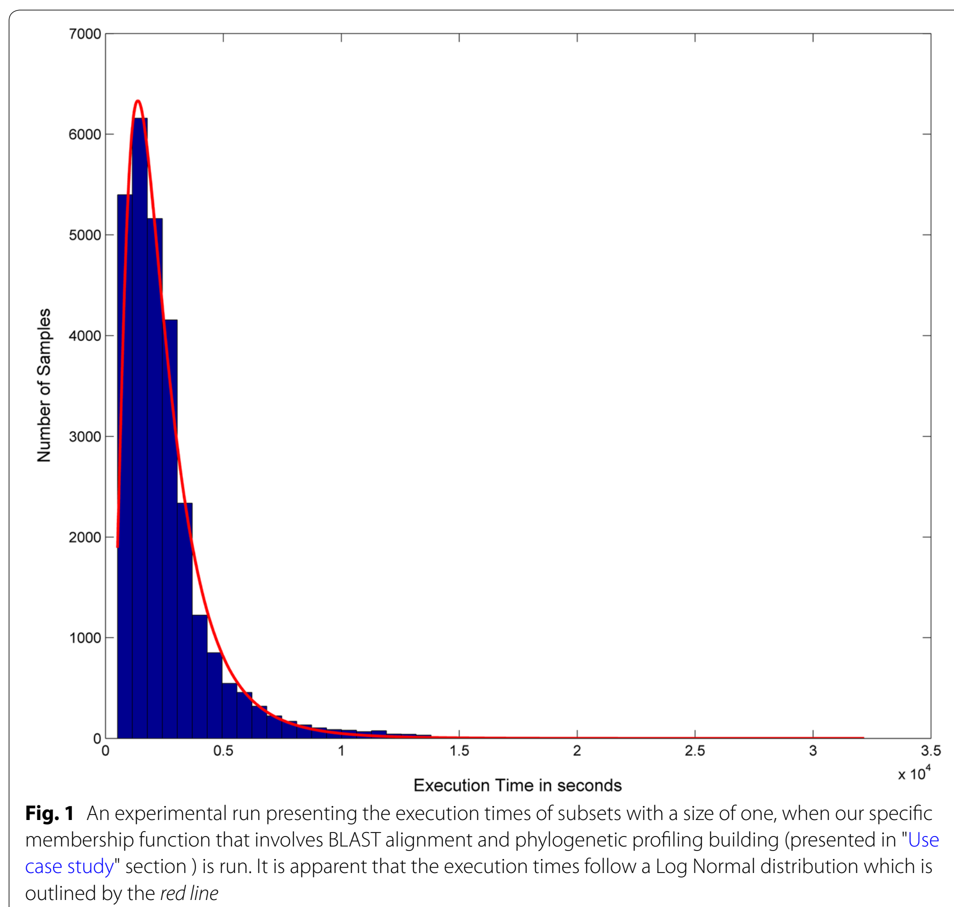

Fig. 1 An experimental run presenting the execution times of subsets with a size of one, when our specific membership function that involves BLAST alignment and phylogenetic profiling building (presented in "Use case study" section) is run. It is apparent that the execution times follow a Log Normal distribution which is outlined by the *red line*

The *delayTime* % defined by Eq. 7 is the total time wasted as a percentage of the actual processing time.

$$delaytime\,\% = \frac{totalDelay}{\frac{m}{n} * avgProcessingTime * threads} * 100 \quad (7)$$

Assuming that the average processing time, *avgProcessingTime*, of a single instance is at least two and a half times greater than the overhead time and the number of threads is at least eight, then by setting n at 0.01 % of m will lead to a *delayTime* % value equal to 0.05 % which is considered insignificant.

We conclude that a value of n approximating 0.01 % of m is a reasonable compromise. In practice, other limitations to the size of the subset n may exist, that are related to the nature of the memberships functions involved and must be taken into account. For example, in processes using hash tables extensively or having significant memory requirements, a relatively high subset size would not be beneficial as there is risk for the hash tables to be overloaded resulting in poor performance and high RAM usage.

It is evident that an accurate size n of the subsets cannot be easily calculated from a general formula as it may have specific constraints due to the actual processes involved. However, a general rule of thumb can be established of setting n around 0.01 % of m and is expected to work reasonably well for the majority of cases. It is however, classified as a parameter that can be optimized and thus its manipulation is encouraged on a use case basis.

Execution engine

A number of requirements motivated us to implement a basic workflow execution engine that was used in our experiments for validating our approach. These requirements are the deployment of containers on sites that include all the necessary software and tools, graphic workflow description, secure connections over SSH tunneling and HTTPS and not requiring elevated user privileges for accessing sites. The execution environment is comprised of a number of computational sites having a UNIX based operating system and a global, universally accessible cloud storage similar to Amazon S3, referred to as object storage. The object storage is used to download input data, upload final data and to share data between sites. It is not used for storing intermediate data that temporarily exist within each site. We have implemented the proposed framework using Java 8 and Shell scripting in Ubuntu Linux 14.04.

The overall architecture is loosely based on a master/slave model, where a master node responsible for executing the external scheduler serves as the coordinator of actions from the beginning to the completion of a given workflow. The master node is supplied with basic information like the description of the workflow and input data, the object storage and the computational sites. The workflow can be described as a directed acyclic graph (DAG) in the GraphML [7] language by specifying graph nodes corresponding to data and compute procedures and connecting them with edges as desired. To describe the workflow in a GUI environment, the user can use any of the available and freely distributed graph design software tools that supports exporting to GraphML.

The only requirement for using a computational site is the existence of a standard user account and accessibility over the SSH protocol. Each site is initialized by establishing a

secure SSH connection through which a Docker [28] container equipped with the software dependencies required to execute the workflow is fetched and deployed. Workflow execution on each site takes place within the container. The object storage access credentials are transferred to the containers and a local daemon is launched for receiving subsequent commands from the master. The daemon is responsible for initiating the internal scheduler and passing all received commands to it. Communication between the master and the daemons running within the Docker container on each site is encrypted and takes place over SSH tunneling. File transfers between sites and the object storage are also encrypted and take place over the HTTPS protocol.

Use case study

The selected case study utilized in validating our approach is from the field of comparative genomics, and specifically the construction of the phylogenetic profiles of a set of genomes. Phylogenetic profiling is a bioinformatics technique in which the joint presence or joint absence of two traits across large numbers of genomes is used to infer a meaningful biological connection, such as involvement of two different proteins in the same biological pathway [35, 37]. By definition, a phylogenetic profile of a genome is an array where each line corresponds to a single sequence of a protein belonging to the genome and contains the presence or absence of the particular entity across a number of known genomes that participate in the study.

The first step in building phylogenetic profiles involves the sequence alignment of the participating protein sequences of all genomes against themselves. It is performed by the widely used NCBI BLAST tool [25] and the process is known as a BLAST all vs all procedure. Each protein is compared to all target sequences and two values are derived, the identity and the e-value. *Identity* refers to the extent to which two (nucleotide or amino acid) sequences have the same residues at the same positions in an alignment, and is often expressed as a percentage. *E-value* (or expectation value or expect value) represents the number of different alignments with scores equivalent to or better than a given threshold *S*, that are expected to occur in a database search by chance. The lower the E-value, the more significant the score and the alignment.

Running this process is extremely computationally demanding, the complexity of which is not straightforward to estimate [2], but can approach $O(n^2)$. For example, a simple sequence alignment between 0.5 million protein sequences, can take up to a week on a single high-end personal computer. Even when employing high-performance infrastructures, such as a cluster, significant time as well as the expertise to both run and maintain a cluster-enabled BLAST variant are required. Furthermore the output files consume considerable disk space which for large analyses can easily exceed hundreds of GBs.

Based on the sequence alignment data, each phylogenetic profile requires the comparison and identification of all homologues across the different number of genomes in the study. The phylogenetic profiling procedure for each genome requires the sequence alignment data of all its proteins against the proteins of all other genomes. Its complexity is linear to the number of sequence alignment matches generated by blast. Different types of phylogenetic profiles exist, including binary, extended and best bi-directional all 3 of which are constructed in our workflow procedure.

According to our data organization methodology, in this case proteins correspond to *Insts* and are grouped into *OUs*, which in this case are their respective genomes. Independent pipelines are formed for each *OU* consisting firstly of the BLAST process involving the sequence alignment of the proteins of the *OU* against all other proteins of all *OUs* and secondly of the three phylogenetic profile creation processes which utilizes the output of the first in order to create the binary, extended and best bi-directional phylogenetic profile of the genome corresponding to the *OU*. These pipelines are then scheduled according to the scheduling policy described in .

Results and discussion

A number of experiments have been performed in order to validate and evaluate our framework. Therefore, this section is divided into (a) the validation experiments further discussed in "Validation" subsection, where the methods outlined in "Methods" section are validated and (b) the comparison against Swift, a high performance framework, further discussed in "Comparison against a high performance framework" subsection where the advantages of our approach become apparent.

The computational resources used are presented in Table 1. Apart from the privately owned resources of our institution, the cloud resources consist of a number of virtual machines belonging to the European Grid Infrastructure (EGI) federated cloud and operated by project Okeanos [21] of GRNET (Greek Research and Technology Network). Okeanos is based on the Synnefo (the meaning of the word is "cloud" in Greek) open source cloud software which uses Google Ganeti and other third party open source software. Okeanos, is the largest academic cloud in Greece, spanning more than 5400 active VMs and more than 500,000 spawned VMs.

As the resources utilize different processors of unequal performance, their performance was compared to the processors of the cloud resources which served as a baseline reference. As such, the number of CPUs of each site was translated to a number of baseline CPUs, so a direct comparison can be performed. In this way, non integer numbers appear in the number of baseline CPUs of each site.

This combination of local, privately owned computational resources with cloud-based resources represents the typical use case we are addressing, individuals or research labs that wish to extend their computational infrastructure by adopting resources of one or multiple cloud vendors.

Table 1 The pool of available computational resources along with their hardware type, number of threads and number of baseline processors are presented

# Count	CPU type	# CPUs	# Baseline CPUs
1×	2 × Intel Xeon E5 2660 @ 2.2 GHZ	24	21.7
1×	2 × Intel Xeon E5 2660 @ 2.2 GHZ	12	16.7
1×	Intel i7 6700 @ 4.0 GHZ	8	15
1×	Intel i7 4790S @ 3.5 GHZ	8	11.3
10×	AMD Opteron 6172 @ 2.1 GHZ	8	8
Total			
14	–	132	144.7

All machines were equipped with more than 6 GB of RAM and were connected to the internet through a 100 MBps connection

The input data used in our experiments consists of an extended plant pangenome of 64 plant genomes including 39 cyanobacteria for which the complete proteome was available. The total size was 268 MB and includes 619,465 protein sequences nd 2.3×10^8 base pairs. In order to accommodate our range of experiments, the data was divided into sub-datasets.

It must be noted that, although the input data used may appear relatively small in file size, it can be very demanding to process, requiring weeks on a single personal computer. The particular challenge in this workflow is not the input size but the computational requirements in conjunction with the size of the output as will become apparent in the following sections. The dataset consist of files downloaded from the online and publicly accessible databases of UniProt [10] and PLAZA[36] and can also be provided by our repositories upon request. The source code of the proposed framework along with the datasets utilized in this work can be found in our repository https://www.github.com/akintsakis/odysseus.

Validation

In order to experimentally validate the optimal subset size value as outlined in "Internal scheduler" section and the overall scalability performance of our approach, a number of experiments were conducted utilizing the phylogenetic profiling use case workflow. All execution times reported below involve only the workflow runtime and do not include site initialization and code and initial data downloads as they require a nearly constant time, irrespective of both problem size and number of sites and as such they would distort the results and not allow for accurately measuring scaling performance. For reporting purposes, the total time for site initialization is approximately 3–5 min.

Optimal subset value n

The phylogenetic profiling workflow was executed with an internal scheduler subset size value n of 0.0010, 0.0025, 0.0050, 0.0100, 0.0250, 0.0500 and 0.2500 % as a percentage of the total number of protein sequences in three distinct datasets comprising 189,378, 264,088 and 368,949 protein sequences. All sites presented in Table 1 except for the first one, participated in this experiment. The site execution times for each subset size for all three datasets are presented in boxplot form in Fig. 2. They verify the hypothesis presented in "Internal scheduler" subsection, we observe that the fastest execution time is achieved when the subset size n is set close to our empirical estimation of 0.01 % of the total dataset size. It is apparent that smaller or larger values of n lead to increased execution times. Generally, in both three datasets analyzed we observe the same behavior and pattern of performance degradation when diverging from the optimal subset size.

Smaller values of n lead to substantially longer processing times mainly due to the delay effect presented in Eq. 7. As n increases, the effect gradually attenuates and is diminished for values larger than 0.0050 % of the dataset. Larger subset sizes impact performance negatively, with the largest size of 0.2500 % tested, yielding the slowest execution time overall. This can be attributed to the fact that for larger subset sizes, the load may not be optimally balanced and some threads that were assigned disproportionately higher load might prolong the overall total execution time while other threads are idle. Additionally, large subset sizes can lead to reduced opportunities for parallelization,

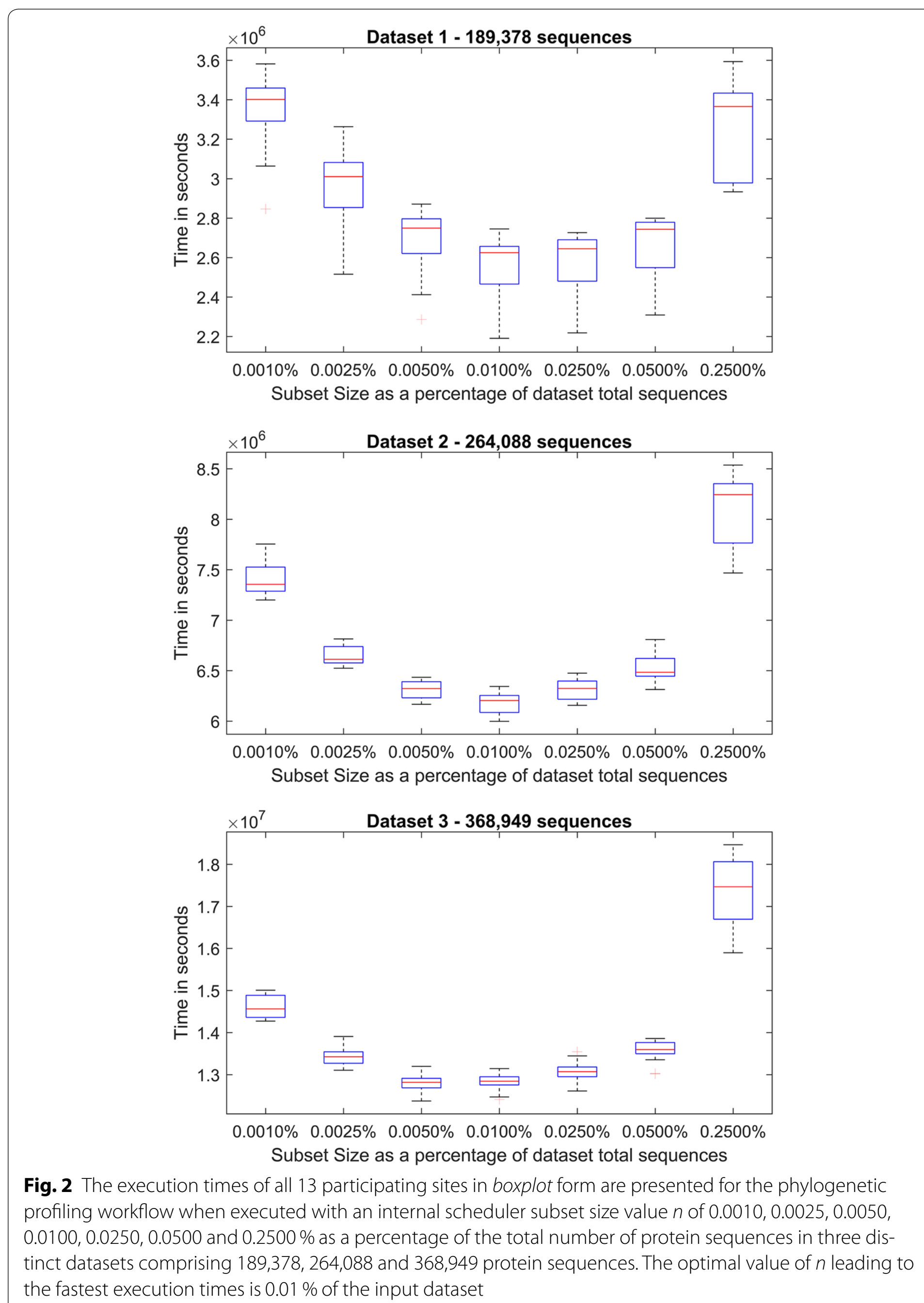

Fig. 2 The execution times of all 13 participating sites in *boxplot* form are presented for the phylogenetic profiling workflow when executed with an internal scheduler subset size value *n* of 0.0010, 0.0025, 0.0050, 0.0100, 0.0250, 0.0500 and 0.2500 % as a percentage of the total number of protein sequences in three distinct datasets comprising 189,378, 264,088 and 368,949 protein sequences. The optimal value of *n* leading to the fastest execution times is 0.01 % of the input dataset

especially on smaller OUs that are broken into fewer chunks than the available threads on site, thus leaving some threads idle.

The average memory usage of all execution sites for each subset size for all three datasets is presented in Fig. 3. It is apparent that both the subset size and the size of the dataset increase memory consumption. Between smaller subset sizes, differences in memory usage are insignificant and inconsistent, thus difficult to measure. As we reach the larger subsets, the differences become more apparent. Due to the current workflow not being memory intensive, increases in memory usage are only minor. However, in a

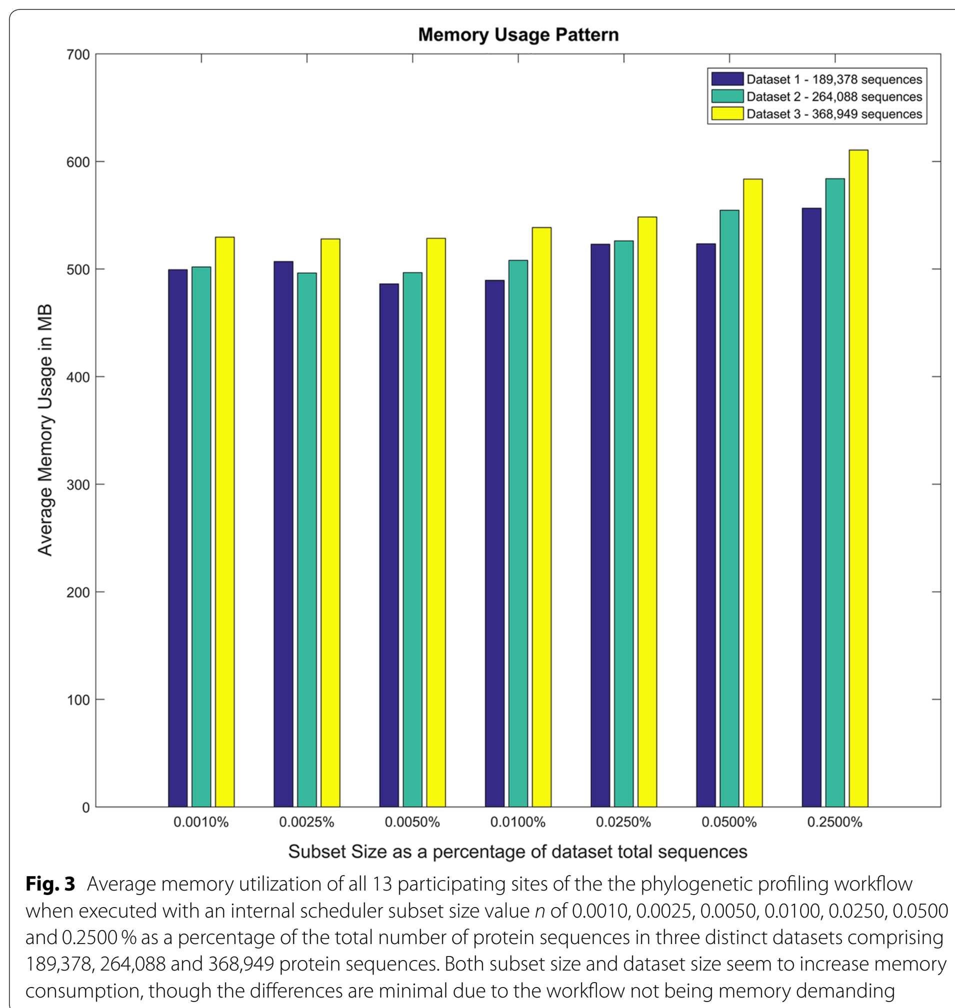

Fig. 3 Average memory utilization of all 13 participating sites of the the phylogenetic profiling workflow when executed with an internal scheduler subset size value *n* of 0.0010, 0.0025, 0.0050, 0.0100, 0.0250, 0.0500 and 0.2500 % as a percentage of the total number of protein sequences in three distinct datasets comprising 189,378, 264,088 and 368,949 protein sequences. Both subset size and dataset size seem to increase memory consumption, though the differences are minimal due to the workflow not being memory demanding

memory demanding workflow these differences could be substantial. Although the size of the dataset to be analyzed cannot be tuned, the subset size can and it should be taken into account in order to remain within the set memory limits. A subset size n value of 0.0100 % is again a satisfactory choice when it comes to keeping memory requirements on the low end.

Although we have validated that an adequate and cost-effective approach is to set the value of n at 0.0100 % of the total size of the dataset, we must state that optimal selection of n is also largely influenced by the type of workflow and thus its manipulation is encouraged on a use case basis.

Execution time reduction scaling

In this experiment the performance scalability of our approach was evaluated. For the needs of this experiment, a subset of the original dataset was formed, consisting of only the 39 cyanobacteria. The purpose was to evaluate the speed-up and efficiency and compare it to the ideal case, in which a linear increase in the total available processing power would lead to an equal reduction in processing time. Speed-up $S(P)$, and efficiency $E(P)$,

are fundamental metrics for measuring the performance of parallel applications and are defined in literature as follows:

$$S(p) = \frac{T(1)}{T(p)} \tag{8}$$

where $T(1)$ is the execution time with one processor and $T(p)$ is the execution time with p processors.

$$E(p) = \frac{T(1)}{p * T(p)} \tag{9}$$

The above equations found in literature assume that p processors of equal computational power are used. As in our case we use resources of uneven computational performance, we translate their processing power into baseline processor units. Consequently, p can take continuous and not discrete values, corresponding to the increase in computational power as measured in baseline processors.

All sites presented in Table 1 participated in this experiment. The sites were sorted in ascending order according to their multithreaded performance and the workflow was executed a number of times equal to the number of sites, by increasing the number of participating sites one at a time.

The execution times of all sites are presented in boxplot form for all workflow runs in Fig. 4. The X axis represents the total computational power score of the sites participating in the workflow and the Y axis, in logarithmic scale, represents the site execution time in seconds. The dashed magenta line is the ideal workflow execution time (corresponding to linear speed-up) and it intersects the mean values of all boxplots. As we can

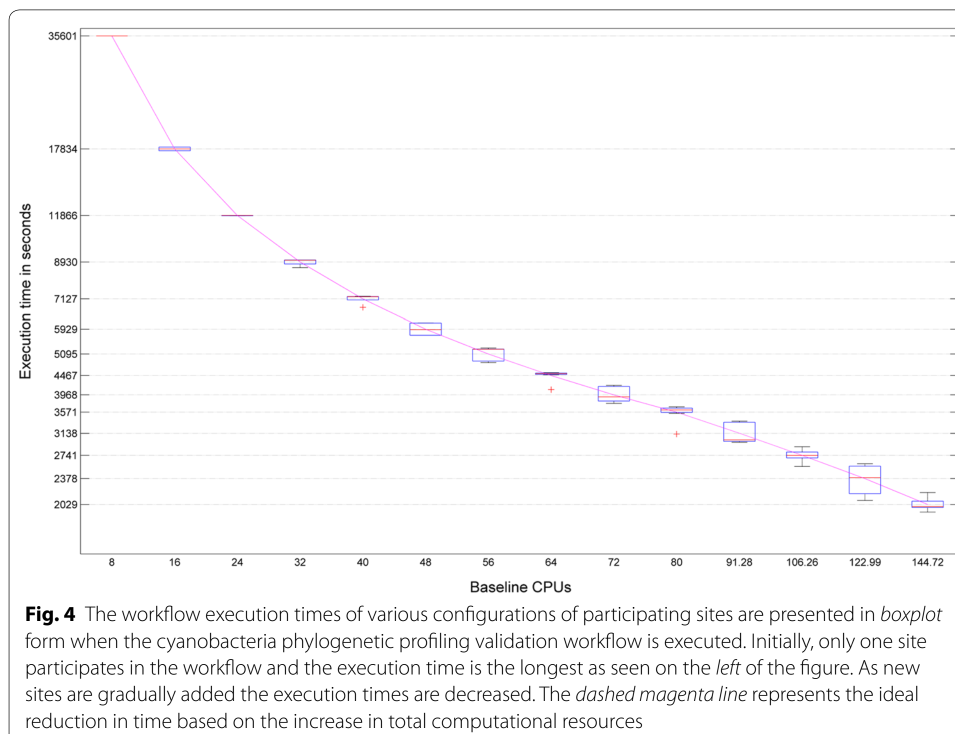

Fig. 4 The workflow execution times of various configurations of participating sites are presented in *boxplot* form when the cyanobacteria phylogenetic profiling validation workflow is executed. Initially, only one site participates in the workflow and the execution time is the longest as seen on the *left* of the figure. As new sites are gradually added the execution times are decreased. The *dashed magenta line* represents the ideal reduction in time based on the increase in total computational resources

see variations in site execution time for each workflow run are consistent with no large deviations present. There are outliers in some workflow runs towards the lower side, where one site would terminate before others as there are no more *OU* pipelines to process. Despite being an outlier value however, they are not laid too far away in absolute quantities. Execution times fall consistently as new sites are added and computational resources are increased.

A closer inspection of the results can be found in Table 2 where the execution times and the average and makespan speed-up and efficiency are analyzed. As expected, the average speed-up is almost identical to the ideal case, where the speed-up is equal to p and efficiency approaches the optimal value of 1. This was to be expected, as our approach does not introduce any overhead and keeps file transfers to a minimum, almost like as if all the processing took place on a single site. The minuscule variations observed can be attributed to random variations in the processing power of our sites and/or our benchmarking of the sites and to random events controlled by the OS. It can be observed that when using a high number of CPUs the efficiency tends to marginally drop to 0.97. This is attributed to the fact the data intensive part of the workflow is limited by disk throughput and cannot be accelerated by increasing the CPU count. Although the data intensive part is approximately 3–5 % of the total workflow execution time when excluding potential file transfers, using such a high number of CPUs for this workflow begins to approach the boundaries of Amdahl's law [19].

On average, the makespan efficiency is 0.954 % for all runs. It can be presumed that the makespan speed-up and efficiency tend to reach lower values when a higher number of sites are involved. This is to be expected as some sites terminate faster than others when the pool of *OU* pipelines is exhausted and as such their resources are no longer utilized. This effect becomes apparent mostly when using a very high number of CPUs for the given workflow that results in a workflow completion time of less than 30 min. Although it is apparent in this experiment, we are confident it will not be an issue in

Table 2 Speed-up and efficiency as scalability metrics of our approach

#CPUs	Execution times in seconds				Average		Makespan	
	Average	Min	Max	Std	S(p)	E(p)	S(p)	E(p)
8.0	35,601	35,601	35,601	0.0	8.00	1.00	8.00	1.00
16.0	17,834	17,629	18,037	288.3	15.9	0.99	15.8	0.98
24.0	11,866	11,845	11,880	18.1	24.0	1.00	24.0	0.99
32.0	8930	8,628	9,039	201.4	31.9	0.99	31.5	0.98
40.0	7127	6,768	7,240	201.5	39.9	0.99	39.3	0.98
48.0	5929	5,713	6,159	234.7	48.0	1.00	46.2	0.96
56.0	5095	4,834	5,285	206.8	55.9	0.99	53.9	0.96
64.0	4467	4,097	4,546	150.5	63.8	0.99	62.6	0.97
72.0	3968	3,767	4,207	178.0	71.7	0.99	67.7	0.94
80.0	3571	3,122	3,688	164.5	79.7	0.99	77.1	0.96
91.3	3138	2,970	3,378	182.7	90.7	0.99	84.3	0.92
106.2	2741	2,561	2,888	85.0	103.9	0.98	98.5	0.93
123.0	2378	2,080	2,604	191.8	119.8	0.97	109.3	0.89
144.7	2029	1,937	2,183	69.4	140.3	0.97	130.4	0.90

real world cases as using 14 sites for this workflow, can be considered as an overkill and therefore slightly inefficient.

In general, the average speed-up and efficiency is the metric of interest when evaluating the system's cost efficiency and energy savings as our approach automatically shuts down and releases the resources of sites that have completed their work. The makespan speedup corresponds to the actual completion time of the workflow when all sites have terminated and the resulting data is available. Our approach attempts to optimize the makespan speed-up but with no compromise in the average speed-up, i.e. the system's cost efficiency. We can conclude from this experiment that the average speed-up is close to ideal and the makespan speed-up is inferior to the ideal case by about 5 % on average and can approach 10 % when using a high number of resources when compared to the computational burden of the workflow.

Comparison against a high performance framework

To establish the advantages of our approach against existing approaches, we chose to execute our use case phylogenetic profiling workflow in Swift and perform a comparison. Swift [49] is an implicitly parallel programming language that allows the writing of scripts that distribute program execution across distributed computing resources [47], including clusters, clouds, grids, and supercomputers. Swift is one of the highest performing frameworks for executing bioinformatics workflows in a distributed computing environment. The reason we chose Swift is that it is a well established framework that emphasizes parallelization performance and in use in a wide range of applications also including bioinformatics.

Swift has also been integrated [27] into the popular bioinformatics platform Galaxy, in order to allow for utilization of distributed resources. Although perfectly capable of achieving parallelization, Swift is unable to capture the underlying data characteristics of the bioinformatics workflows addressed in this work, thus leading to unnecessary file transfers that increase execution times and costs and may sometimes even become overwhelming to the point of causing job failures.

The testing environment included all sites presented in Table 1 except for the first one, as we were unable to set the system environment variables required by Swift, due to not having elevated privileges access to it. In the absence of a pre-installed shared file system, the Swift filesystem was specified as local, where all data were staged from the site where Swift was executing from. This is the default Swift option that is compatible with all execution environments and does not require a preset shared file system. The maximum number of jobs on each site was set equal to the site's number of CPUs.

Three datasets were chosen as input to the phylogenetic profiling workflow, these are the total of 64 plant genomes and its subsets of 58 and 52 genomes. The datasets were chosen with the purpose of approximately doubling the execution time of each workflow run when compared to the previous one. Uptime, system load and network traffic among others were monitored on each site. In order to perform a cost analysis, we utilized parameters from the Google Cloud Compute Engine pricing model, according to which, the cost per hour to operate the computational resources is 0.232$ per hour per 8 baseline CPUs and the cost of network traffic is 0.12$ per GB as per Google's internet egress worldwide cheapest zone policy.

The makespan execution time, total network traffic and costs of our approach against Swift when executing the phylogenetic profiling workflow for the three distinct datasets are presented in Table 3. The values presented are average values of 3 execution runs. As can be seen, for workflow runs 1 and 2, Swift is approximately 20 % slower in makespan and 16 % slower in the case of workflow run 3. This is attributed mostly to the time lost waiting for the file transfers to take place in the case of Swift. It must be noted that we were unable to successfully execute workflow 3 until termination with Swift, due to network errors near the end of the workflow that we attribute to the very large number of required file transfers. Had the workflow reached termination, we expect Swift to be about 17–18 % slower. As the particular use case workflow is primarily computationally intensive, an increase in the input size of the workflow increases the computational burden faster than the data intensive part, thus the performance gap is slightly smaller in the case of workflow 3.

The total network traffic includes all inbound and outbound network traffic of all sites. It is apparent that it is significantly higher in Swift thus justifying the increased total execution time accounting to file transfers. Regarding the cost of provisioning the VMs, it was calculated by multiplying the uptime of each site with the per processors baseline cost of operation. The external scheduler of our approach will release available resources when the pool of *OU* pipelines is exhausted, thus leading to cost savings that can range from 10 to 25 % when compared to keeping all resources active until the makespan time. Oppositely, this feature is not supported by Swift and as such in this case all sites are active until makespan time, leading to increased costs. The cost savings of our approach regarding provisioning of VMs were higher than 40 % in all three workflow.

The cost of network transfers is difficult to interpret as it is dependent on the locations and the providers of the computational resources. The cost presented here is a worst case estimate that would take place when all network traffic between sites were

Table 3 Execution time, network traffic and cost comparison of our approach against Swift

	Workflow 1 $1.16 * 10^8$ Bases	Workflow 2 $1.67 * 10^8$ Bases	Workflow 3 $2.3 * 10^8$ Bases
Makespan			
Ours	9302 s	18278 s	33752 s
Swift	11194 s	21933 s	39229 s
Diff	+20.3 %	+19.9 %	+16.2 %
Network total traffic			
Ours	0.183 GB	0.338 MB	0.403 GB
Swift	57.525 GB	88.944 GB	140.982 GB
Cost of Provisioning VMs			
Ours	8.86 $	17.6 $	32.63 $
Swift	13.05 $	25.56 $	45.73 $
Diff	+47.2 %	+45.2 %	+40.0 %
Cost of network transfers			
Ours	0.02 $	0.04 $	0.05 $
Swift	6.90 $	10.67 $	16.91 $
Total cost			
Ours	8.88 $	17.64 $	32.68 $
Swift	19.95 $	36.23 $	62.64 $
Diff	+124.6 %	+105.3 %	+91.6 %

charged at the nominal rate. That is not always true, for example if all sites were located within the same cloud facility of one vendor there would be no cost at all for file transfers. However, they would still slow down the workflow leading to increased uptime costs, unless the sites were connected via a high speed link like InfiniBand often found in supercomputer configuration environments. In a hybrid cloud environment, which this work addresses, as computational sites will belong to different cloud vendors and private infrastructures, the file transfer cost can be significant and may even approach the worst case scenario. In total, our approach is significantly more cost effective than Swift, which can be anywhere from 40 to 47 % to more than 120 % more expensive, depending on the pricing of network file transfers.

To further analyze the behavior of our framework against Swift, in Fig. 5 we present the system load and network activity of all sites when executing the phylogenetic profiling workflow with the 64 genome input dataset for both our approach and Swift. The Swift system load and network activity are denoted by the blue line and red line respectively, while the system load and network activity of our approach are denoted by the green and magenta lines respectively. Figure 6 plots each line separately for site 0, allowing for increased clarity. A system load value of 1 means that the site is fully utilized, while values higher than 1 means that the site is overloaded. A network activity value of

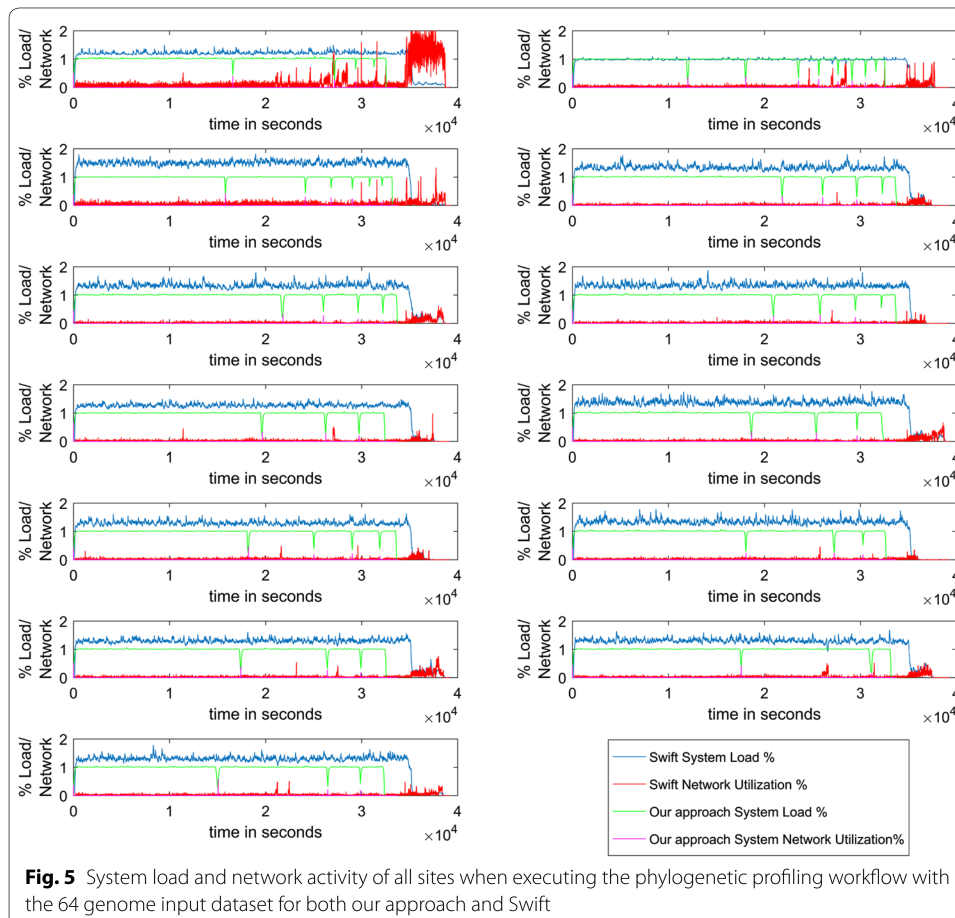

Fig. 5 System load and network activity of all sites when executing the phylogenetic profiling workflow with the 64 genome input dataset for both our approach and Swift

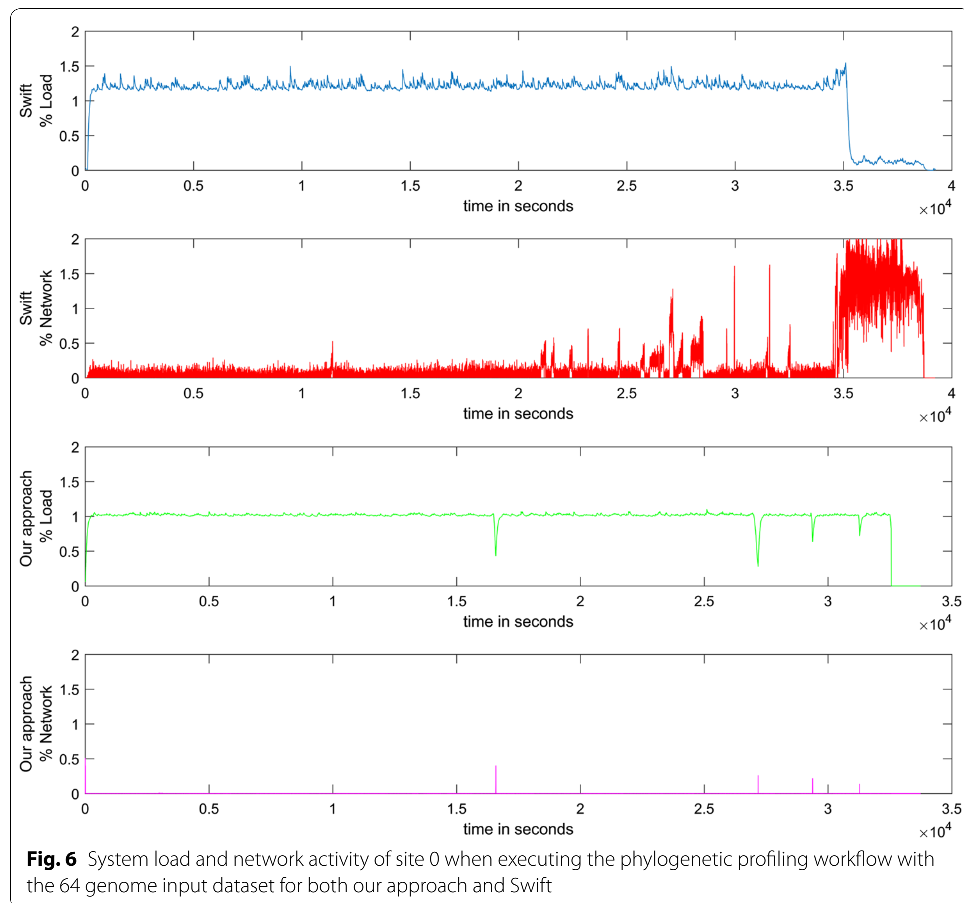

Fig. 6 System load and network activity of site 0 when executing the phylogenetic profiling workflow with the 64 genome input dataset for both our approach and Swift

1 corresponds to utilization of 100 MBps. The network activity reported is both incoming and outgoing, so the maximum value it can reach is 2, which means 100 MBps of incoming and outgoing traffic simultaneously, though this is difficult to achieve due to network switch limitations.

Regarding our approach, the network traffic magenta line is barely visible, marking only a few peaks, that coincide with drops in system load as denoted by the green line. This is to be expected as network traffic takes place while downloading the input data of the next *OU* pipeline and simultaneously uploading the output of the just processed *OU* pipeline, during which the cpu is mostly inactive. It is apparent that the number of sections between the load drops are equal to the number of *OU* pipelines, 64 in this case. Other than that, the system load is consistently at a value of 1.

In the Swift execution case, load values are slightly higher than 1 in all sites except site 1 which has 12 instead of 8 CPUs. This can be attributed to the slightly increased computational burden of submitting the jobs remotely and transferring inputs and outputs to the main site. The internal scheduler of our approach operating on each site can be more efficient. Network traffic is constant and on the low end for the duration of the workflow, as data is transferred to and from the main site. However, near the end of the workflow, system load drops and network traffic increases dramatically, especially on site 0 which is the main site from which Swift operates and stages all file transfers to and from the

other sites. As the computationally intensive part of most *OU* pipelines comes to an end, the data intensive part then requires a high number of file transfers that overloads the network and creates a bottleneck. This effect significantly slows down the makespan and is mostly responsible for the increased execution times of Swift and the costly file transfers. In large workflows where the data to be moved is hundreds of GBs, it can even lead to instability due to network errors.

Conclusions and future work

In this work, we presented a versatile framework for optimizing the parallel execution of data-intensive bioinformatics workflows in hybrid cloud environments. The advantage of our approach is that it achieves surpassing time and cost efficiency than existing solutions through minimization of file transfers between sites. It accomplishes that through the combination of a data management methodology that organizes the workflow into pipelines with minimal data interdependencies along with a scheduling policy for mapping their execution into a set of heterogeneous distributed resources comprising a hybrid cloud.

Furthermore, we compared our methodology with Swift, a state of the art high performance framework and achieved superior cost and time efficiency in our use case workflow. By minimizing file transfers, the total workflow execution time is reduced thus leading to directly decreasing costs based on uptime of computational resources. Costs can also decrease indirectly, as file transfers can be costly especially in hybrid clouds where resources are not located within the facility of a single cloud vendor.

We are confident that our methodology can be applied to a wide range of bioinformatics workflows sharing similar characteristics with our use case study. We are currently working on expanding our use case basis by implementing workflows in the fields of metagenomics, comparative genomics, and haplotype analysis according to our methodology. Additionally, we are improving our load estimation functions so as to more accurately capture the computational load of a given pipeline through an evaluation of the initial input.

In the era of Big data, cost-efficient high performance computing is proving to be the only viable option for most scientific disciplines [14]. Bioinformatics is one of the most representative fields in this area, as the data explosion has overwhelmed current hardware capabilities. The rate at which new data is produced is expected to increase significantly faster compared to the advances, and the cost, in hardware computational capabilities. Data-aware optimization can be the a powerful weapon in our arsenal when it comes to utilizing the flood of data to advance science and to provide new insights.

Authors' contributions

AMK and FEP conceived and designed the study and drafted the manuscript. AMK implemented the platform as a software solution. PAM participated in the project design and revision of the manuscript. AMK and FEP analyzed and interpreted the results and coordinated the study. FEP edited the final version of the manuscript. All authors read and approved the final manuscript.

Acknowledgements

This work used the European Grid Infrastructure (EGI) through the National Grid Infrastructure NGI_GRNET - HellasGRID. We also thank Dr. Anagnostis Argiriou (INAB-CERTH) for access to their computational infrastructure.

Competing interests

The authors declare that they have no competing interests.

Received: 17 August 2016 Accepted: 11 October 2016
Published online: 21 October 2016

References

1. Afgan E, Baker D, Coraor N, Chapman B, Nekrutenko A, Taylor J. Galaxy cloudman: delivering cloud compute clusters. BMC Bioinform. 2010;11(Suppl 12):S4.
2. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215(3):403–10.
3. Angiuoli SV, Matalka M, Gussman A, Galens K, Vangala M, Riley DR, Arze C, White JR, White O, Fricke WF. Clovr: a virtual machine for automated and portable sequence analysis from the desktop using cloud computing. BMC Bioinform. 2011;12(1):356.
4. Baker KR, Trietsch D. Principles of sequencing and scheduling. Hoboken: Wiley; 2013.
5. Berthold MR, Cebron N, Dill F, Gabriel TR, Kötter T, Meinl T, Ohl P, Sieb C, Thiel K, Wiswedel B. Knime: the konstanz information miner. In: Data analysis, machine learning and applications. Berlin: Springer; 2008. p. 319–26
6. Bocchi E, Mellia M, Sarni S. Cloud storage service benchmarking: methodologies and experimentations. In: Cloud networking (CloudNet), 2014 IEEE 3rd international conference on, IEEE; 2014. p. 395–400
7. Brandes U, Eiglsperger M, Herman I, Himsolt M, Marshall MS. Graphml progress report structural layer proposal. In: Graph drawing. Berlin: Springer; 2001. p. 501–12
8. Bux M, Leser U. Parallelization in scientific workflow management systems. 2013. arXiv preprint arXiv:1303.7195
9. Chong Z, Ruan J, Wu CI. Rainbow: an integrated tool for efficient clustering and assembling rad-seq reads. Bioinformatics. 2012;28(21):2732–7.
10. Consortium U, et al. The universal protein resource (uniprot). Nucleic Acids Res. 2008;36(suppl 1):D190–5.
11. De Oliveira D, Ocaña KA, Ogasawara E, Dias J, Gonçalves J, Baião F, Mattoso M. Performance evaluation of parallel strategies in public clouds: a study with phylogenomic workflows. Future Gener Comput Syst. 2013;29(7):1816–25.
12. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
13. Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, et al. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci Progr. 2005;13(3):219–37.
14. Duarte AM, Psomopoulos FE, Blanchet C, Bonvin AM, Corpas M, Franc A, Jimenez RC, de Lucas JM, Nyrönen T, Sipos G, et al. Future opportunities and trends for e-infrastructures and life sciences: going beyond the grid to enable life science data analysis. Front Genet. 2015:6.
15. Emeakaroha VC, Maurer M, Stern P, Łabaj PP, Brandic I, Kreil DP. Managing and optimizing bioinformatics workflows for data analysis in clouds. J Grid Comput. 2013;11(3):407–28.
16. Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J, et al. Bioconductor: open software development for computational biology and bioinformatics. Genome Biol. 2004;5(10):R80.
17. Goecks J, Nekrutenko A, Taylor J, et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. 2010;11(8):R86.
18. Gurtowski J, Schatz MC, Langmead B. Genotyping in the cloud with crossbow. Curr Prot Bioinform. 2012:15–3.
19. Hill MD, Marty MR. Amdahl's law in the multicore era. Computer. 2008;7:33–8.
20. Iosup A, Sonmez O, Anoep S, Epema D. The performance of bags-of-tasks in large-scale distributed systems. In: Proceedings of the 17th international symposium on high performance distributed computing. New York: ACM; 2008. p. 97–108
21. Koukis V, Venetsanopoulos C, Koziris N. ~ Okeanos: Building a cloud, cluster by cluster. IEEE Internet Comput. 2013;3:67–71.
22. Krampis K, Booth T, Chapman B, Tiwari B, Bicak M, Field D, Nelson KE. Cloud biolinux: pre-configured and on-demand bioinformatics computing for the genomics community. BMC Bioinform. 2012;13(1):42.
23. Litzkow MJ, Livny M, Mutka MW. Condor-a hunter of idle workstations. In: Distributed computing systems, 8th international conference on, IEEE; 1988. p. 104–11.
24. Liu B, Madduri RK, Sotomayor B, Chard K, Lacinski L, Dave UJ, Li J, Liu C, Foster IT. Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses. J Biomed Inform. 2014;49:119–33.
25. Lobo I. Basic local alignment search tool (blast). Nature Educ. 2008;1(1):215.
26. Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones MB, Lee EA, Tao J, Zhao Y. Scientific workflow management and the kepler system. Concurr Comput Pract Exp. 2006;18(10):1039–65.
27. Maheshwari K, Rodriguez A, Kelly D, Madduri R, Wozniak J, Wilde M, Foster I. Enabling multi-task computation on galaxy-based gateways using swift. In: Cluster computing (CLUSTER), 2013 IEEE international conference on, IEEE; 2013. p. 1–3.
28. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux J. 2014;2014(239):2.
29. Minevich G, Park DS, Blankenberg D, Poole RJ, Hobert O. Cloudmap: a cloud-based pipeline for analysis of mutant genome sequences. Genetics. 2012;192(4):1249–69.
30. Moschakis IA, Karatza HD. Multi-criteria scheduling of bag-of-tasks applications on heterogeneous interlinked clouds with simulated annealing. J Syst Soft. 2015;101:1–14.
31. Naccache SN, Federman S, Veeraraghavan N, Zaharia M, Lee D, Samayoa E, Bouquet J, Greninger AL, Luk KC, Enge B, et al. A cloud-compatible bioinformatics pipeline for ultrarapid pathogen identification from next-generation sequencing of clinical samples. Genome Res. 2014;24(7):1180–92.
32. Nagasaki H, Mochizuki T, Kodama Y, Saruhashi S, Morizaki S, Sugawara H, Ohyanagi H, Kurata N, Okubo K, Takagi T, et al. Ddbj read annotation pipeline: a cloud computing-based pipeline for high-throughput analysis of next-generation sequencing data. DNA Res. 2013;dst017.
33. Ocaña KA, De Oliveira D, Dias J, Ogasawara E, Mattoso M. Designing a parallel cloud based comparative genomics workflow to improve phylogenetic analyses. Future Gener Comput Syst. 2013;29(8):2205–19.

34. Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock MR, Wipat A, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics. 2004;20(17):3045–54.
35. Pellegrini M, Marcotte EM, Thompson MJ, Eisenberg D, Yeates TO. Assigning protein functions by comparative genome analysis: protein phylogenetic profiles. Proc Natl Acad Sci. 1999;96(8):4285–8.
36. Proost S, Van Bel M, Sterck L, Billiau K, Van Parys T, Van de Peer Y, Vandepoele K. Plaza: a comparative genomics resource to study gene and genome evolution in plants. Plant Cell. 2009;21(12):3718–31.
37. Psomopoulos FE, Mitkas PA, Ouzounis CA, Promponas VJ, et al. Detection of genomic idiosyncrasies using fuzzy phylogenetic profiles. PLoS One. 2013;8(1):e52854.
38. Reid JG, Carroll A, Veeraraghavan N, Dahdouli M, Sundquist A, English A, Bainbridge M, White S, Salerno W, Buhay C, et al. Launching genomics into the cloud: deployment of mercury, a next generation sequence analysis pipeline. BMC Bioinform. 2014;15(1):30.
39. Rice P, Longden I, Bleasby A, et al. Emboss: the European molecular biology open software suite. Trends Genet. 2000;16(6):276–7.
40. Schatz MC. Cloudburst: highly sensitive read mapping with mapreduce. Bioinformatics. 2009;25(11):1363–9.
41. Smith B, Grehan R, Yager T, Niemi D. Byte-unixbench: a unix benchmark suite. 2011.
42. Sreedharan VT, Schultheiss SJ, Jean G, Kahles A, Bohnert R, Drewe P, Mudrakarta P, Görnitz N, Zeller G, Rätsch G. Oqtans: the rna-seq workbench in the cloud for complete and reproducible quantitative transcriptome analysis. Bioinformatics. 2014:btt731.
43. Stajich JE, Block D, Boulez K, Brenner SE, Chervitz SA, Dagdigian C, Fuellen G, Gilbert JG, Korf I, Lapp H, et al. The bioperl toolkit: Perl modules for the life sciences. Genome Res. 2002;12(10):1611–8.
44. Tang W, Wilkening J, Desai N, Gerlach W, Wilke A, Meyer F. A scalable data analysis platform for metagenomics. In: Big data, 2013 IEEE international conference on, IEEE; 2013. p. 21–6.
45. Wall DP, Kudtarkar P, Fusaro VA, Pivovarov R, Patil P, Tonellato PJ. Cloud computing for comparative genomics. BMC Bioinform. 2010;11(1):259.
46. Weng C, Lu X. Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. Future Gener Comput Syst. 2005;21(2):271–80.
47. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, Foster I. Swift: a language for distributed parallel scripting. Parallel Comput. 2011;37(9):633–52.
48. Wolstencroft K, Haines R, Fellows D, Williams A, Withers D, Owen S, Soiland-Reyes S, Dunlop I, Nenadic A, Fisher P, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Res. 2013:gkt328.
49. Zhao Y, Hategan M, Clifford B, Foster I, Von Laszewski G, Nefedova V, Raicu I, Stef-Praun T, Wilde M. Swift: fast, reliable, loosely coupled parallel computation. In: Services, 2007 IEEE Congress on, IEEE; 2007. p. 199–206.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com