Scalable and Efficient Whole-exome Data Processing Using Workflows on the Cloud

J. Cała[a,*], E. Marei[a], Y. Xu[b], K. Takeda[c], P. Missier[a,**]

[a] *School of Computing Science, Newcastle University, Newcastle upon Tyne, UK*
[b] *Institute of Genetic Medicine, Newcastle University, Newcastle upon Tyne, UK*
[c] *Microsoft Research, Cambridge, UK*

Abstract

Dataflow-style workflows offer a simple, high-level programming model for flexible prototyping of scientific applications as an attractive alternative to low-level scripting. At the same time, workflow management systems (WFMS) may support data parallelism over big datasets by providing scalable, distributed deployment and execution of the workflow over a cloud infrastructure. In theory, the combination of these properties makes workflows a natural choice for implementing Big Data processing pipelines, common for instance in bioinformatics. In practice, however, correct workflow design for parallel Big Data problems can be complex and very time-consuming.

In this paper we present our experience in porting a genomics data processing pipeline from an existing scripted implementation deployed on a closed HPC cluster, to a workflow-based design deployed on the Microsoft Azure public cloud. We draw two contrasting and general conclusions from this project. On the positive side, we show that our solution based on the e-Science Central WFMS and deployed in the cloud clearly outperforms the original HPC-based implementation achieving up to 2.3x speed-up. However, in order to deliver such performance we describe the importance of optimising the workflow deployment model to best suit the characteristics of the cloud computing infrastructure. The main reason for the performance gains was the availability of fast, node-local SSD disks delivered by D-series Azure VMs combined with the implicit use of local disk resources by e-Science Central workflow engines. These conclusions suggest that, on parallel Big Data problems, it is important to couple understanding of the cloud computing architecture and its software stack with simplicity of design, and that further efforts in automating parallelisation of complex pipelines are required.

Keywords: Workflow-based application, Whole-exome sequencing, Performance analysis, Cloud computing, HPC

1. Introduction

In this paper we report on our experience in porting a complex genome processing pipeline, from a home-made scripted implementation deployed on a closed department HPC cluster, to a workflow-based parallel implementation deployed on a public cloud. Genomics is only one of several areas of science where these porting exercises are becoming commonplace. The growing demand for resource capacity for Big Data processing combined with its simultaneous decrease in cost make moving to the cloud increasingly appealing. Therefore, we believe that the experience gained from such an exercise has value beyond the particular case study. Indeed, while the idea of "sequencing as a service" is gaining ground, pushing the NGS data processing pipelines closer to the sequencing facilities, it is important

*Corresponding author
**Principal corresponding author
Email addresses: Jacek.Cala@ncl.ac.uk (J. Cała), Marei.Eyad@gmail.com (E. Marei), Yaobo.Xu@ncl.ac.uk (Y. Xu), Kenji.Takeda@microsoft.com (K. Takeda), Paolo.Missier@ncl.ac.uk (P. Missier)

for many labs and "analysis-as-a-service" outfits (e.g. the EBI in the UK) to retain control over the structure and composition of their pipelines. Engineers and practitioners who have responsibility for maintenance and evolution of such home-grown implementations will specifically benefit from the outcomes of this research.

Our results show that a cloud-based deployment of a complex Big Data processing pipeline, when properly tuned for a specific workflow middleware and an underlying cloud infrastructure, provides better scalability properties than an equivalent HPC-based deployment, at lower cost and with improved performance. Here we report in detail on performance results and scalability. Furthermore, we also reflect realistically on the complexities of undertaking such a project, which we balance against the expected benefits of the configuration, namely scalability, understandability, evolvability, and cost efficiency.

Regarding the cost estimation, in particular, we note that the ability to monetise the resource utilisation associated with Big Data processing is becoming very important in settings such as health care. The cost of processing a single patient's sample is one of the dominating factors in the future large-scale deployments of genetic testing based on the entire genome (Whole-Genome Sequencing; WGS) and potentially at population scale. In particular, the creation of *cloud commons* has recently and authoritatively been advocated as a way to address the increasing requirements for computation resources that follow from the widespread adoption of genomics techniques for diagnostic purposes [1].

1.1. Background

The cost of sequencing human genomes continues to decrease [2]. With the number of DNA base pairs sequenced per $ unit reportedly doubling every five months [3], genetic testing is poised to become a routine diagnostic technique that can be deployed on a large scale [4]. At the same time, allocating the computation resources needed to process the data is also becoming increasingly affordable. In the UK, the cost of sequencing a single patient sample is currently below $1.5K and decreasing. Large initiatives like the 100,000 Genome Project in the UK[1] promise to deliver genetic testing at population scale within the next few years.

Genetic testing based on Next-Generation Sequencing (NGS) aims at enumerating the mutations that are present in a human patient's genome[2] and identifying those mutations that are known, from research literature, to be deleterious. This process involves three distinct phases: DNA sequencing which produces raw genome data; variant calling, i.e. the identification of the variants within the genome (mutations); and analysis of these variants. The processing pipeline described in this paper is concerned with the second phase, namely variant calling.

Ideally, DNA sequencing includes the entire genome (WGS). While WGS technology is rapidly coming on the market at affordable prices, in the last few years most research labs have adopted Whole-Exome Sequencing (WES) as interim technology. WES is limited to the exome information, that is to the regions of DNA that are responsible for protein expression by way of RNA translation. These are the priority areas of the genome, where mutations can be more easily identified as deleterious, as they have a higher chance of directly compromising protein synthesis. WES-based diagnosis provides a good trade-off between diagnostic power and the cost of data processing, as exomes only account for about 1.5% of the entire genome and can, therefore, be processed using in-house computational resources.

1.2. Requirements for NGS data processing

Genetic research facilities around the world have adopted in-house solutions to implementing WES data processing

[1] http://www.genomicsengland.co.uk/

[2] And, increasingly, in mitochondrial DNA as well as non-human genomes.

pipelines for variant calling, e.g. [5, 6, 7, 8]. A pipeline generally consists of a composition of configurable library packages and tools that implement genome analysis algorithms, and which are freely available to the community. Many of these, including GATK[3], Picard[4], those found in the Bioconductor repository[5] and others, have been recently surveyed [9]. These pipelines are deployed either on HPC clusters, or on a cloud infrastructure. Yet, both Next-Generation Sequencing and analysis of NGS data are still challenging [10].

A first requirement in the data processing architecture for NGS is **scalability**. As NGS technology progresses and NGS-based genetic diagnostics moves into population-wide deployment, e.g. with publicly-funded initiatives such as the 100K Genome Project in the UK, genome data processing must be able to scale simultaneously in the size of the input datasets (from about 15 GB of compressed WES data for one sample to 1 TB for WGS), and in the number of genomes processed over time.

Flexibility in pipeline design and evolution is a second requirement. Ongoing progress in the third party, community tools that compose the pipelines promises increases in variant detection coverage and accuracy, which may translate into improved diagnostic power. It ought to be relatively simple for non-expert programmers to track this evolution, by making incremental changes to an existing pipeline solution.

Finally, with broad clinical deployment of these techniques in mind, it is important to provide **traceability** and, therefore, **accountability** for the outcomes of a diagnostic process, from the raw data to variant calling, to selection of the variants implicated in a disease.

1.3. The Cloud-e-Genome project

The *Cloud-e-Genome* project, started in late 2013, aims to address the three requirements above. For the NGS data processing pipeline design it employs a high-level, workflow programming model based on the *e-Science Central* scientific workflow manager [11]. e-Science Central workflows can be deployed on a cluster of cloud nodes, and can be designed to exploit the parallelism that is implicit in the data processing logic, leading to efficient usage of cloud resources. The workflow engine, which orchestrates the execution of the pipeline steps, also captures the provenance of the execution, making it possible to trace its details *post hoc* and thus providing both accountability of the variant selection process, and its reproducibility.

To demonstrate our solution, we have used the pipeline implementation currently in use at the Institute of Genetic Medicine (IGM) at Newcastle University as a starting point. This version of the pipeline is implemented as a complex collection of shell scripts, which invoke third party tools as described in more detail below, and coordinates their execution on the departmental HPC cluster. Such a solution does not meet any of the three requirements above: it cannot scale beyond the limits of the local cluster, it requires expert knowledge of the scripts for maintenance and evolution, and does not provide provenance collection for *post hoc* accountability.

One notable example of an integrated solution that attempts to fulfill the same goals as ours is the Globus Genomics[6] system [12] that integrates the well-known Galaxy[7] workflow model for genetics applications with the Globus toolkit. Globus Genomics aims at improving Galaxy's native data management capabilities, and allowing workflows to scale across cloud resources. We discuss differences between this system and our architecture in Sec. 5 (Related Work).

[3] https://www.broadinstitute.org/gatk/
[4] http://broadinstitute.github.io/picard/
[5] http://www.bioconductor.org/

[6] https://www.globus.org/genomics
[7] galaxyproject.org

1.4. Contributions and relevance to this Journal special issue

This paper extends our preliminary workshop publication [13] which reported on initial progress on the Cloud-e-Genome project, a collaboration between the School of Computing Science and Institute of Genetic Medicine at Newcastle University. This extended version offers the following new contributions:

- A detailed description of the porting of the original genomics pipeline implementation to the e-Science Central system. We discuss migration challenges (Sec. 2) and then describe three designs (Sec. 3) which we call *synchronous*, *asynchronous* and *chained*. They provide different options for exploiting data parallelism. Our main result here, somewhat surprisingly, is that the simplest amongst the approaches to parallel processing we have experimented with showed better performance than the more sophisticated ones;
- A full evaluation of the implementation on the Microsoft Azure cloud infrastructure, where we present performance results that demonstrate pipeline scalability as we increase the number of input samples and processing cores (Sec. 4). Our results achieve up to 2.3x speed-up over the scripted pipeline running on our HPC cluster, when the latter is allocated exclusively to our workflow (i.e. with no other jobs contending for resources);
- A thorough cost analysis that illustrates the trade-off between response time and cost (£/exome) with changing number of input samples and processing cores (Sec. 4.4). At current commercial rates for our Azure-based configuration, the cost per 150 Gbases whole-exome sample is around £5 (about $8).

These contributions directly address some of the specific topics that informed this special issue, as our work is set in the context of scientific workflows for Big Data in the Cloud, and in this setting, we propose innovative methods of processing Big Data in the Cloud and demonstrate performance and low costs.

1.5. The e-Science Central workflow manager

e-Science Central (e-SC) is a workflow manager designed for scientific data management, analysis and collaboration. It has been used in a number of scientific projects such as spectral data visualisation, medical data capture and analysis, and chemical property prediction. Yet its prior use in bioinformatics was to run only simple NGS analyses in the Cloud4Science project [14].

e-SC realises a classic dataflow programming model [15]. A dataflow consists of *workflow blocks*, connected through data dependency links. A block may either implement a function locally, or it may invoke remote service operations; Fig 3 shows an example of this simple model. The dataflow model has no control primitives (conditionals, loops) and besides passing data along the links, blocks can only share data through explicit file system operations. Importantly, a block may also represent a sub-workflow. This adds hierarchical structure to a workflow design, but it also provides a simple mechanism for parallel execution because sub-workflows are scheduled independently from each other and can be executed concurrently on a cluster of compute nodes. An example of such hierarchical arrangement is shown in Fig. 6, where the red boxes indicate sub-workflow blocks.

The simple dataflow model translates into the ease of programming and flexibility, one of our requirements stated above. Given a palette of pre-defined workflow blocks, geneticists may create their data processing pipelines simply by assembling pre-defined components visually, using a web-based workflow editor provided by the system. The interface allows scientists to upload data, edit and run workflows, experiment with parameter changes, and share results in the cloud. More advanced users with software

development skills may build and upload their own analysis services into the system and share them with others as ready-to-use workflow blocks. A REST-based API is also provided so that external applications can leverage the platform's functionality, making it easier to build scalable and secure cloud-based applications.

e-SC can be deployed on private and public clouds, and has been tested on Eucalyptus, Amazon AWS and Microsoft Azure. The system architecture follows the master-worker pattern, with the e-SC server connected via a messaging queue to one or more workflow engines each allocated on a different processing node [11]. The workflow execution follows the principle that a single workflow invocation is indivisible and always executed on a single workflow engine. Although it is unlike other similar systems such as Pegasus [16] and Taverna [17], it provides predictable system performance for short-running blocks, as well as faster communication between blocks; more details can be found in [18].

The exception to the indivisibility principle in e-SC is when a composite workflow contains sub-workflows (possibly nested at multiple levels). During runtime, each of them will be treated as a new workflow invocation by the system, and executed independently from other sub-workflows. Sub-workflows are enqueued on the server and any available engine can pull one or more of them from the central queue and execute them. These executions may be either synchronous or asynchronous. In particular, for an asynchronous execution an explicit `Wait` block with a list of sub-workflows can be used to create a barrier and force suspension of the current invocation until all the listed invocations have completed. We exploit this execution model in one of our pipeline designs, as described later in Sec. 3.

2. Pipeline migration challenges

Due to the amount of input data generated during sequencing, NGS pipelines require substantial amount of resources to run. Departments which have local access to sequencers can greatly benefit from running them together with a local HPC cluster. This can save a lot of time and cost on data transfer.[8] However, majority of research labs outsource sample sequencing, using "sequencing-as-a-service", and so need to transfer the raw sequence reads before they can start the analysis. In these cases, deployment of the NGS pipeline on the cloud becomes an attractive alternative approach to purchasing and maintaining a local HPC cluster.

2.1. Starting point: the legacy pipeline

The current pipeline, developed for research purposes at the Institute of Genetic Medicine, is illustrated in Fig. 1. In our setting, sample sequencing is outsourced. For each patient's sample submitted to an external service, scientists receive compressed, 2-lane, pair-end raw sequence reads in the FASTQ format. It means that each sample consists of four compressed fastq.gz files and on average is nearly 15 GiB in size (36 GiB uncompressed).

The pipeline starts with sequence alignment of the reads (BWA [19]). This is followed by cleaning (Picard), sequence recalibration, filtering, variant calling and recalibration (GATK [20]), coverage analysis (bedTools), and annotation (Annovar [21] as well as a in-house annotation tool). This processing sequence closely resembles the best practices defined by the Broad Institute[9] and adds only extra annotation and coverage steps to it.

Overall, the pipeline involves three key stages: (1) preparation of the raw sequences for variant discovery and coverage calculation, (2) variant calling and recalibration, (3) variant filtering and annotation. Stages 1 and 3 are executed in a loop so that all tools involved are invoked on each sample separately. As there is no dependency between samples in these two stages, paralellisation at

[8] https://www.emc.com/collateral/brochure/h10628-br-challenges-in-ngs.pdf

[9] http://www.broadinstitute.org/gatk/guide/best-practices

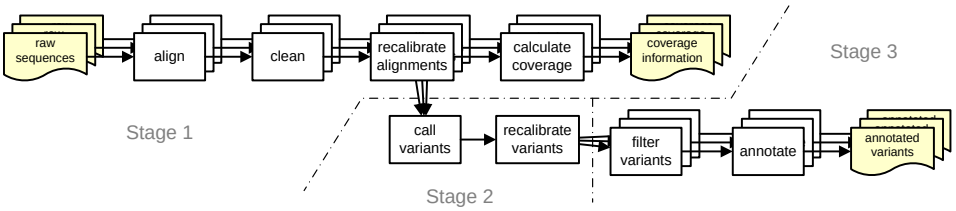

Figure 1: Our existing NGS processing pipeline.

this stage is straightforward. Conversely, Stage 2 runs only once for all input samples, thus parallel processing across samples is no longer possible. However, since the tools used in Stage 2 can operate independently on individual chromosomes (or even on smaller sub-chromosomal regions), we can still exploit parallelism at this stage, by splitting each exome within each sample along chromosome boundaries. We refer to this as *chromosome-split*.

The current pipeline is implemented as a number of shell scripts that coordinate the sequential execution of the tools and is deployed on a shared HPC cluster with the Open Grid Engine (OGE) submission environment. Stages 1 and 3 are submitted using the standard OGE `qsub` command. Stage 2 uses the GATK Queue framework to split a large set of input data among all compute nodes available in the cluster.

A fragment of the implementation script, namely to invoke the Picard tools, is shown in Fig. 2. From this example, it should be clear that the lack of abstraction in the programming model complicates even the simplest pipeline evolution task. In addition, pipeline developers must also be knowledgeable about the available deployment options. For instance, job submission to the local HPC cluster requires explicit allocation of the desired number of nodes and cores within the nodes. This configuration is specific to the tools invoked by the scripts and to the cluster itself, making the entire implementation hardly portable. Moreover, inter-task dependencies, and thus effectively the structure of the pipeline, are hidden in the code. This includes knowledge of physical file locations and how files are shared across the steps of the pipeline. Finally, the cluster provides no isolation; interference from other cluster users, in the form of apparently minor issues such as saturated disk space of compute nodes (scratch space), causes long-running executions to fail arbitrarily, often wasting hours or even days of computing time.

2.2. Migration tasks

Porting an existing scientific pipeline involves the following sequence of tasks: (i) developing new *tool blocks* and libraries to wrap the tools used in the pipeline, (ii) developing adapter blocks (*shims*) for data format conversion in between blocks [22], and (iii) designing workflows that replicate the pipeline's original functionality, possibly using a nested workflow structure. The combination of these tasks has been challenging, requiring about six months of a workflow design expert's time.

2.2.1. The tool blocks

These are for the most part wrappers that can drive underlying tools using their native, command-line interface. They are complemented by e-SC shared libraries, which are installed only once and cached by the workflow engine for any future use. The shared libraries provide not only better efficiency in running the tools but they also promote reproducibility because they eliminate dependencies on external data and services. For instance, to access the human reference genome (HG19 from UCSC), we created a shared library that included the specific version and flavour of the genome. By following this design principle for all data dependencies and tools, our pipeline is fully reproducible in e-SC and can also be "rolled back"

to previous versions of any library. By the end of the migration, we had created 30 new tool block types and 14 libraries specific to the NGS data processing.

2.2.2. The adapter blocks

These *shims* are often necessary for mapping across data formats, or to re-organise intermediate data. As a design principle, their use should be minimal because they are custom-made, non-reusable components. Our implementation only required two of them.

2.2.3. Designing workflows

As an example of partial migration, Fig. 2 shows the main part of the script that implements the cleaning task showed earlier in Fig. 1. The same functionality "recoded" using a workflow is presented in Fig. 3. Wrapper blocks, such as `Picard-CleanSAM` and `Picard-MarkDuplicates`, communicate via files in the local filesystem of the workflow engine, which is explicitly denoted as a connection between blocks. The workflow includes also utility blocks to import and export files, i.e. to transfer data from/to the shared data space (in this case, the Azure blob store). Data in the shared space can be used by other workflows, which are potentially running on different execution nodes. This resembles the HPC configuration, where data between jobs are shared via the parallel file system and jobs can also use the local, compute node disk space (scratch space) to store their intermediate data.

2.2.4. Implementing limited loop functionality

In all stages the pipeline iterates either over a set of samples (Stages 1 and 3) or over chromosomes within a sample (Stage 2). For instance, in Stage 1 a sequence of jobs including alignment, cleaning and recalibration are run separately for each sample. This functionality cannot be directly reproduced using e-SC, which is missing the loop control primitive. However, it can be replicated using e-SC's **map** functionality, whereby a lambda function is applied independently to each element of a list. In this

```
echo Preparing directories $PICARD_OUTDIR and
     $PICARD_TEMP
mkdir -p $PICARD_OUTDIR
mkdir -p $PICARD_TEMP

echo Starting PICARD to clean BAM files...
$Picard_CleanSam INPUT=$SORTED_BAM_FILE
     OUTPUT=$SORTED_BAM_FILE_CLEANED

echo Starting PICARD to remove duplicates...
$Picard_NoDups INPUT=$SORTED_BAM_FILE_CLEANED \
    OUTPUT=$SORTED_BAM_FILE_NODUPS_NO_RG \
    METRICS_FILE=$PICARD_LOG REMOVE_DUPLICATES=true \
    ASSUME_SORTED=true TMP_DIR=$PICARD_TEMP

echo Adding read group information to bam file...
$Picard_AddRG INPUT=$SORTED_BAM_FILE_NODUPS_NO_RG
    OUTPUT=$SORTED_BAM_FILE_NODUPS RGID=$READ_GROUP_ID \
    RGPL=illumina RGSM=$SAMPLE_ID \
    RGLB="${SAMPLE_ID}_${READ_GROUP_ID}" \
    RGPU="platform_Unit_${SAMPLE_ID}_${READ_GROUP_ID}"

echo Cleaning intermediate files
rm $SORTED_BAM_FILE_CLEANED
rm $SORTED_BAM_FILE_NODUPS_NO_RG
rm -r $PICARD_TEMP

echo Indexing bam files...
samtools index $SORTED_BAM_FILE_NODUPS
```

Figure 2: The main part of the *clean* script from Stage 1.

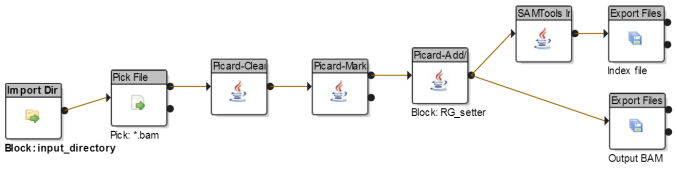

Figure 3: Pipeline fragment shown in Fig. 2 ported to e-Science Central.

instance, the list contains the input samples and is generated by an initial block, and the lambda function is a sub-workflow that encodes the sequence of operations to be applied to each sample; Fig. 4 shows this pattern.

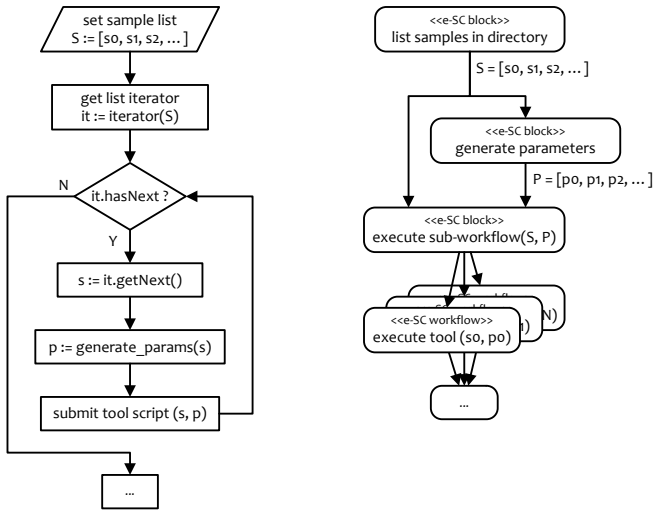

Figure 4: Conversion of loops in a script (left) into an e-SC map dataflow (right).

2.2.5. Automating the end-to-end pipeline

One additional advantage of the workflow solution is the complete automation of the entire pipeline, from the raw sequence alignment to the last step, variant annotation. In contrast, the legacy pipeline makes use of the batch-queuing system to submit the scripts and automates only Stage 1, whilst the following stages are be submitted manually.

3. Exploring alternative parallel workflow designs

One of our workflow design goals has been to fully exploit the distributed deployment model of e-SC workflows over multiple engines. We wanted to take advantage of the data parallelism in the genome processing logic and make sure that a bioinformatician could still understand the overall design.

As mentioned earlier, the pipeline consists of three stages (Fig. 1). Stage 1 can process N input samples in parallel. Stage 2, on the other hand, can process *chromosomes* independently from each other but requires input from as many samples as possible. This means that at the end of Stage 1 the intermediate results from each sample are collected, each sample (an exome) is split by chromosome, and each of these M chromosomes is allocated to a Stage 2 thread. Additionally, because of large variation in the length of chromosomes on a human genome,[10] longer chromosomes can usually be split and processed in parts, whilst shorter ones are processed as a whole. Once Stage 2 is completed, all resulting data fragments, one for each chromosome, are merged again and split into the original input samples to be processed in Stage 3 by N independent threads. Fig. 5 depicts this multiple split structure along two different axes (samples and chromosomes).

This data-parallel pattern can be exploited in different ways using e-SC. To illustrate the range of options available to the workflow designer, we now describe three different yet functionally equivalent solutions.

3.1. Synchronous pipeline

The most intuitive and easy to understand approach is the *synchronous* design. It consist of a top-level, coordinating workflow that invokes eight sub-workflows, each of which implements one step of the pipeline (Fig. 6). The sub-workflows of each step are executed in parallel but synchronously over a number of samples. This means that the top-level workflow submits N sub-workflow invocations for a particular step, waits until all of them complete, and then moves on to the following step.

The complexity of the data structure imposed two variations to this basic behaviour. Firstly, each input file consists of multi-lane sequence reads, so initially the alignment step runs independently for each lane within a sample. Then the aligned lanes are merged together to per-

[10] The longest human chromosome is `chr1` with 249 Mbases, whereas the shortest one is a mitochondrial chromosome `chrM`, only 16 kbases long.

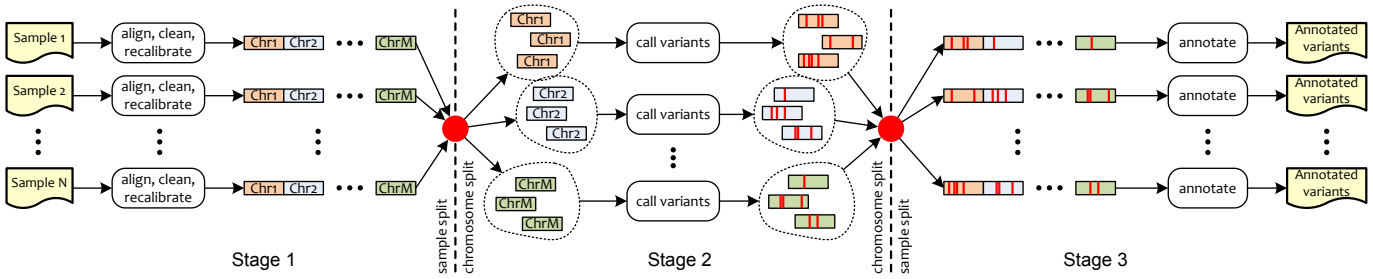

Figure 5: Data parallelism pattern in the genomics pipeline; Stages 1 and 3 exploit per-sample parallelism, Stage 2 exploits per-chromosome parallelism.

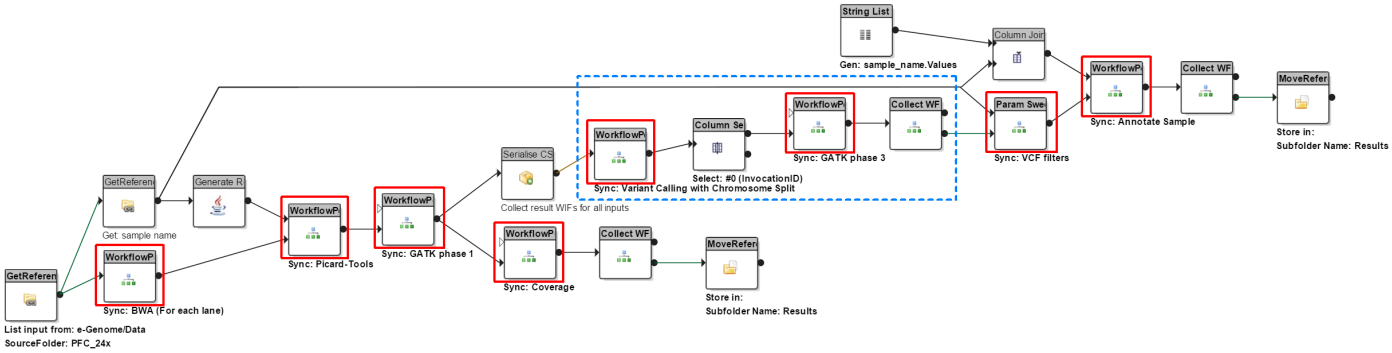

Figure 6: The top-level workflow implementing the WES pipeline; blocks highlighted in red submit subworkflows that implement the pipeline steps; highlighted in dashed blue is Stage 2.

form per-sample refinement. This refinement is shown in Fig. 7, where the separation into two workflows: parent *align sample* and child *align lane*, is designed to improve resource utilisation. Secondly, in Stage 2 we use two workflows: *variant calling with chromosome-split* (parent) and *haplotype caller* (child), to improve the utilisation in the *variant calling* step.

The latter optimisation concerns collecting all intermediate results from Stage 1 (BAM files) for subsequent processing by the variant caller. As it is common for a single sample cohort to include 30 or more input samples (about 450 GB of compressed data), this step splits the data by chromosome region and processes each region in parallel. Thus, the parent workflow implements the split-merge pattern, whereas the child workflow does actual variant discovery on a selected chromosome region. Afterwards, all parts are merged together and we obtain a multi-sample variants file (a VCF file), which is then recalibrated and split into single-sample VCF files.

The synchronous design is easy to understand. The structure of the pipeline is modular and clearly represented by the top-level orchestrating workflow that mainly includes control blocks to run sub-workflows. The control blocks take care of the interaction with the system to submit the sub-workflows, and also suspend the parent invocation until all sub-workflows complete. In this case the parallelisation and synchronisation is managed by e-SC automatically and does not affect the design of the pipeline.

The main drawback of this simple approach is underutilisation of the computing resources allocated to the engines. This is because each step introduces a synchronisation point, where the parent workflow waits for the slowest sub-workflow invocation to complete (note red dots in Fig. 7). The result is a saw-like utilisation graph, where each step consists of the initial period of high resource utilisation, followed by a "tail" during which the use of re-

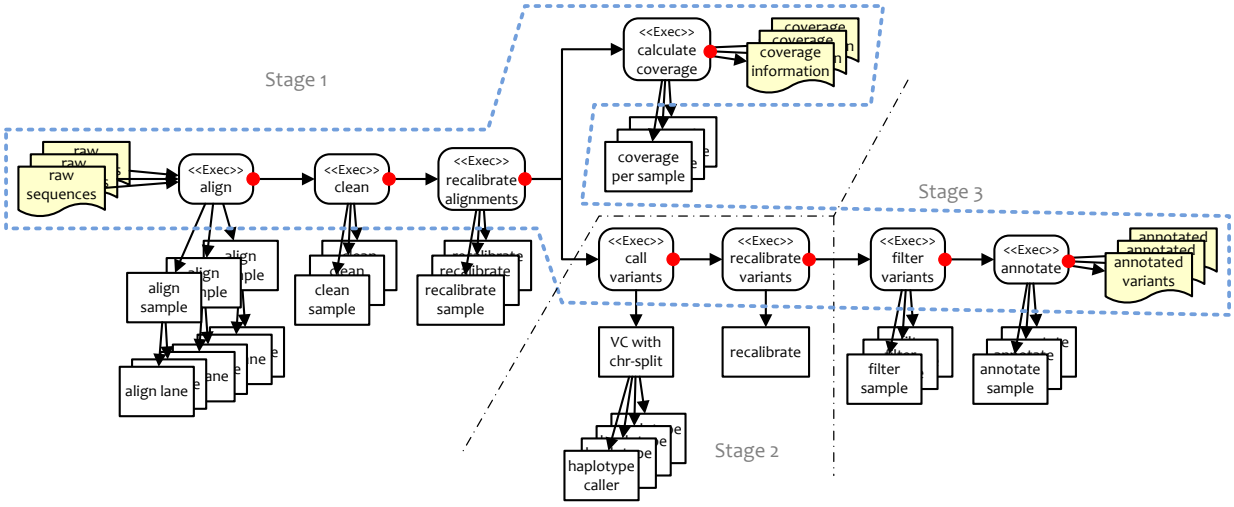

Figure 7: The structure of the invocation graph of the synchronous pipeline design; highlighted in dashed blue is the top-level workflow; red dots indicate the synchronisation points when the top-level invocation waits for all child invocations to complete.

sources degrades (Fig. 8). Importantly, the more variance there is in the size of the input data files (sample cohort), the longer the tail becomes.

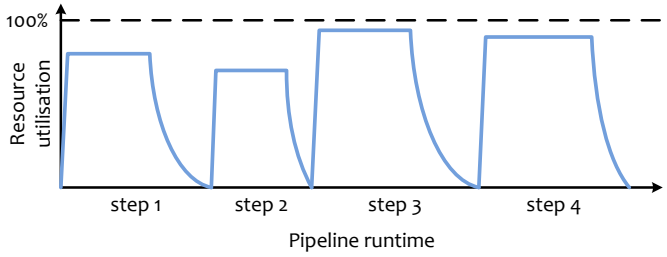

Figure 8: The saw-like utilisation observed with the synchronous design.

3.2. Asynchronous pipeline

An alternative design, aimed at removing waiting times, is the *asynchronous* mode workflow, where the parent workflow does not wait for all children to complete (easily configurable in the workflow submission blocks). In this design, synchronisation points are defined as *barriers* specified by adding the `Wait` blocks before each stage only. The effect of these new wait points is that the top-level workflow is not suspended at each step, but rather all the steps are submitted one after another, and synchronisation occurs only at the end of each stage.

Using the asynchronous approach, most of the synchronisation points move from the top-level workflow to each individual sub-workflow invocation. For example, step *clean* cannot start until the previous step *align* completes. However, an invocation of the *clean* subworkflow for sample A needs to be blocked only by the invocation of the *align* subworkflow related to the same sample A. The `Wait` block, added at the beginning of the *clean* subworkflow, achieves the desired effect by suspending it until results from the predecessor, *align* step, are available. Note that this mode resembles the way job dependencies are expressed in the OGE cluster, where the successor jobs need to know the invocation names of the predecessor jobs and are dispatched only *after* the predecessor completes.

Unfortunately, this design has its own drawbacks. It exhibited very uneven resource allocation and poor overall performance due to certain subtleties in the invocation dispatch policy in e-SC. In more detail, a barrier block can be located anywhere in a workflow and does not suspend the invocation until it is executed. This means that even if a workflow includes a barrier block, it may be dispatched and executed *before* the predecessor invocations finish. However, once locked on the barrier waiting for predecessors, the invocation does not consume an execu-

tion thread and the workflow engine is free to pull another invocation from the queue. As a result, it may happen that a single workflow engine accepts many workflows with barriers in them, while other engines are busy processing predecessor invocations. Later, when the predecessors are finished and barriers are released the workflows accumulated on that single engine resume while the other engines remain idle.

The analysis of the problem suggested that we experiment with yet another implementation model, which we called the *chained pipeline* mode.

3.3. Chained pipeline

The main reason why the asynchronous design failed in reducing the "tails" in the resource utilisation was that the successor invocations (e.g. *clean sample* and *recalibrate sample* in Stage 1) were dispatched and started *before* the predecessor invocations completed (*align lane* and *clean sample*, respectively). As a consequence, the decision on which engine to allocate to a sub-workflow invocation occurred too early, i.e. before the actual resources for the workflow were needed; these are only known at a later stage, namely when locked invocations are resumed. To correct this issue, we redesigned the pipeline so that a successor workflow (e.g. *clean sample*) was submitted only *after* a predecessor invocation has completed (*align lane*, respectively). In this way we were able to create independent invocation chains, one per sample such as: *Align* $\rightarrow$ *Clean* $\rightarrow$ *Recalibrate Sample* in Stage 1. This time, however, the structure of the top-level workflow had to change, too. It reflected stages rather than steps of the pipeline, while the steps in Stages 1 and 3 were linked together in a chain of invocations (Fig. 9).

From the performance perspective, the chained version is very efficient as it does not use any synchronisation at the top level (except between stages and to calculate coverage) nor does it use barriers to wait on predecessor invocations. Instead, it relies on subworkflows calling one another and thus making the chain of invocations.

There are two drawbacks to this design too, however. Firstly, the top-level workflow no longer shows all the steps of the pipeline, but rather only the first step of each stage. This makes the workflow much harder to understand, requiring careful analysis not only of the top-level workflow, but also all subworkflows. Secondly, and more importantly, the pipeline is susceptible to *resource depletion*. This effect occurs because storage resources are not released when a sub-workflow is suspended while waiting for the chain of its successors to complete. For example, *Align* waits until *Clean* and then *Recalibrate Sample* complete. Although the execution thread of an upstream workflow is evicted from the engine, the data it requires is retained on the engine's local storage; the thread will eventually resume and may use that data. However, if too many of these upstream workflows are allocated to an engine, ultimately their storage requirements exhaust the available space on the cloud node, resulting in failure of the workflow and of the whole pipeline.

One could experiment with further redesign of the workflow, for instance using a combination of the chained and asynchronous modes. Yet this would make the design even more complicated and, as evaluation shows, the benefit of using the chained version over the synchronous one is not significant. For this reason, in our performance analysis we used the simpler, synchronous design and compared it with the HPC-based solution.

4. Evaluation

Our evaluation aimed at testing the scalability, reliability, and cost properties of our resulting pipeline. We measure scalability in terms of the *number of exome samples in the workflow input* (but we are also going to report scalability results relative to the raw input file size). The need to increase the number of samples in a single input batch comes mainly from the underlying bioinformatics tools used in the pipeline. In particular, the accuracy of

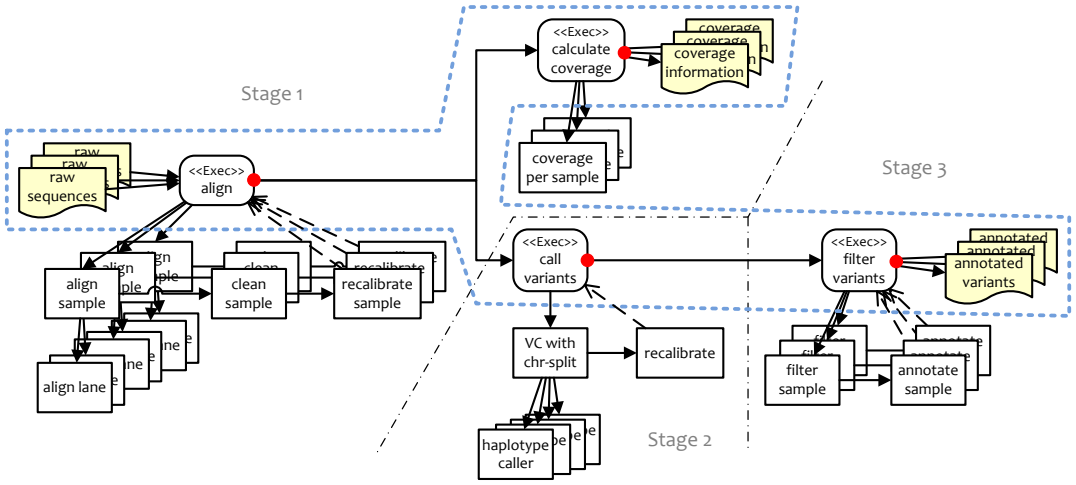

Figure 9: The structure of the invocation graph of the "chained" pipeline design; highlighted in dashed blue is the top-level workflow; red dots indicate the synchronisation points when the top-level invocation waits for all child invocations to complete.

variant calling provided by the GATK tool (Stage 2) is known to increase with the number of samples in the cohort.

As discussed earlier, there are opportunities for exploiting the available data parallelism in the pipeline, namely by launching independent sub-workflows, which are then automatically allocated to engines deployed on different physical nodes. Thus, our strategy for controlling response time with increasing input size is to increase the number of engines available to the system. At the same time, we want to ensure that the allocated resources are used effectively; we may easily over provision the system by allocating too many engines relative to the available sub-workflow workload.

The complexity of the hierarchical workflow designs shown in the previous sections, compounded with the fact that different stages in the pipeline exhibit different degrees of data parallelism, suggest that it would be difficult to determine analytically the optimal number of engines. Therefore, our goal is to show that we can achieve better response time than on an equivalent configuration of nodes allocated to our HPC cluster, while providing better scalability through the elastic properties of the underlying cloud infrastructure. With these considerations, we have designed our experimental evaluation of scalability properties in three phases.

Phase I. Firstly, we explored a space of VM configurations in the cloud that can reliably handle our NGS scenario, while providing response times that compete with those of our HPC cluster. As a full-size input requires over 30 hours of wall clock time to compute, in this phase we tested the pipeline on a small scale (three compute nodes) and with small input sizes (six samples, or about 83 GiB of compressend raw sequence reads), in order to collect enough data points within a reasonable time.

Phase II. Secondly, having found a configuration of cloud resources that works well on a small input, we tested its reliability against increasing input size, at the same time comparing its performance across the different workflow designs discussed earlier, namely synchronous, asynchronous and chained. Reliability, defined as the probability that a workflow runs to completion, is important because failed workflows are expensive, as each workflow execution on a realistic input size can take up to 40 hours of wall clock time, which translates into cloud node allocation costs. This second phase resulted in a final cloud configuration which used a synchronous design that was able to reliably support long running workflows with a 24-

sample input.

Phase III. Finally, we performed scalability testing on the configuration found in the previous phase, measuring response time as a function of input samples over increasing engine numbers. In this phase we also assessed the cost per sample through direct observation of billing information from the Azure cloud provider.

Note that, once a suitable configuration has been found as shown in the rest of this section, it can be simply cloned on the Azure cloud to achieve *virtually* indefinite scalability over multiple batches of input samples.

4.1. Phase I: Rapid configuration discovery on small scale input

In order to compare the pipeline performance in the HPC environment with that in the cloud, we set up comparable test clusters in both environments. Our HPC cluster consists of 20 8-core compute nodes with Intel Xeon E5640, 2.67GHz CPU, running Scientific Linux: 16 nodes with 48 GiB of memory and 160 GB of the local *scratch* space, and four with 96 GiB of memory and 900 GB of the scratch space. All nodes are connected with Gigabit Ethernet and have access to the shared parallel file system (Lustre) where all input, output and reference data were stored. Intermediate data produced during script execution are stored either in the compute nodes' scratch space based on regular HDD or on the Lustre file system.

To perform tests in the cluster in conditions as similar to the cloud as possible, we selected three "larger" compute nodes (with more memory and disk) with *exclusive access* for the duration of our test runs. The tests, therefore, produced unrealistically good results on the cluster, where NGS jobs are normally affected by workload from other users. Note also that in the experiments we did not consider login and head nodes of the cluster because they were used only to submit and manage jobs, with negligible overhead compared to the amount of processing needed from the compute nodes. We also did not consider time required to transfer raw sequence input data to the cluster.

The corresponding Azure tests were run using a small, 6-sample input set to find a configuration comparable with the HPC setup and reliable enough to handle the workload. This resulted in two candidate configurations. The first consisted of A7 VMs with 8-core CPU, 56 GiB of memory and 1.2 TB RAID level-0 disk array built of 600 GB local HDD and two attached 300 GB network disks. The second used D13 VMs with 8-core CPU, 56 GiB of memory and 400 GB SSD. Ubuntu 14.04 was used in both. To mirror the HPC cluster test configuration, we used three of these VMs to run the e-SC workflow engines.

For all tests in Azure the e-SC server ran on a D2 VM but, as for login and head nodes in the cluster, the amount of processing power it required was negligible. Data storage in e-SC was configured to use the Azure blob store, which meant that transfer of large data files between the e-SC workflow engines and the blob store was direct without the need to pass through the server.

Moreover, the Azure e-SC instance was deployed as a single cloud service, so network communication between the server and engines was direct and did not go via the Azure load balancer. Also, similarly to the HPC configuration, the instance was used exclusively to run the evaluation tests and the raw input sequences were stored in the cloud before the tests. Table 1 shows the summary of the configuration of the three selected environments.

Table 1: Basic information about the test infrastructure.

Node type	**CPU model**	**Cores**	**RAM**	**Local disk**
HPC compute node	Intel Xeon E5460 2.66GHz	8	96 GiB	900 GB HDD
Azure VM (A7)	Intel Xeon E5-2660 2.2GHz	8	56 GiB	1.2 TB L0-RAID
Azure VM (D13)	Intel Xeon E5-2660 2.2GHz	8	56 GiB	400 GB SSD

Lessons learnt. Small scale testing was crucial for exploring the large space of target cloud configurations, which is very time-consuming due to the long-running nature of the workflow. Configuration options focus mostly on the VM and its attached local storage. Variability and incremental evolution of the underlying cloud offering from a provider must be taken into account. For example, our tests were conducted at a time when Azure was phasing in SSD disks for attached storage at a competitive cost, prompting us to compare it with our prior RAID-based configuration. While the configuration space improves over time, this also complicates the exploration as it makes for a "mobile target".

4.2. Phase II: testing reliability over realistic workloads

In phase II we tested the candidate configurations on larger workloads, namely 10, 12 and 24 samples per input. Each sample included 2-lane, pair-end raw sequence reads (four files per sample). The average size of input was about 150 Gbases per sample, which was provided as compressed files of nearly 15 GiB size; file decompression is included in the pipeline as one of the initial tasks.

Fig. 10a shows the response time for different configurations relative to the number of samples. From the response time, we derived the throughput of the pipeline, shown in Fig. 10b (each figure is the average over two workflow runs).[11]

The most important finding from this set of tests is that not all configurations scale well with the number of input samples. The WES pipeline stresses file I/O of the underlying system, and we discovered that we were able to saturate the available I/O bandwidth for A-series VMs. Even systems running on the A7 VMs with the RAID L0 array built of three Azure attached disks did not have sufficient I/O throughput to sustain tests larger than six input samples. This is consistent with the limited write throughput of RAID L0 arrays in Amazon EBS as reported in [23]. In effect running our tests for 10 and 12 input samples, we experienced random errors such as blocks hanging infinitely or read/write errors to the file system.

Furthermore, due to the resource depletion problem discussed earlier, the *chained* version of the pipeline could not handle inputs larger than 12 samples. Despite e-SC offering a number of control blocks that can ease the designing of parallel pipelines, improving the workflow design was far more difficult than expected and revealed nuances in the e-SC invocation dispatch and resource eviction policies that were unsuitable to handle tasks large in terms of CPU and data size. Thus, the only two configurations which proved to be reliable enough were the one running in the HPC cluster and the synchronous version of the workflow-based pipeline running in Azure and hosted on the D13 VMs with SSD.

For these two solutions results show linear relation between the response time and number of input samples, and in the largest tested case, with 24 input samples, the cloud-based configuration was 2.3 times faster than the HPC variant running in the *exclusive access* mode. Note that in practice the response from the HPC system would be even longer due to workload of other users and the shared nature of these systems.

The primary reason for such good response time was the availability of the fast, local SSD storage and its extensive use by the e-SC workflow engines. In the HPC cluster the compute nodes were equipped with regular HDD but also the pipeline relied more heavily on the parallel, network filesystem. Presumably, this was also the reason why by adding more input data to the sample cohort, our cloud-based pipeline showed increasing performance, whereas in the cluster the throughput decreased (cf. Fig. 10b).

In the cloud the response time was very stable for VMs with SSD (for the worst case, 24 sample run, $\sigma/\mu \approx 2\%$)

[11] Note that due to the amount of time required to complete a single run, it was not feasible to repeat all tests for larger input sets in the HPC cluster. Thus, for 12- and 24-sample runs figures include only a single data point.

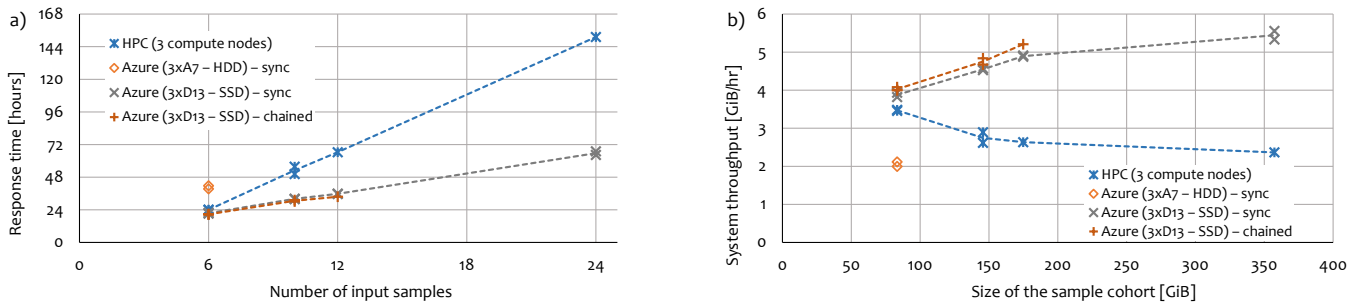

Figure 10: The response time (a) and throughput (b) for different variants of the pipeline and hardware configuration and the increasing number of input samples. Dots represent the actual observed time, lines connect average values (if available).

and slightly less stable for configuration with the RAID array ($\sigma/\mu \approx 4.1\%$ for 6 sample run). In the HPC cluster the response time was also slightly less stable (for the 10 sample run, $\sigma/\mu \approx 7.2\%$; note that we could not repeat tests for 12 and 24 input samples). Apparently, the dispersion of the response time was higher for configurations that relied more on shared network disks.

The outcome of this phase was a reliable cloud configuration coupled with a synchronous pipeline design.

Lessons learnt. Public cloud infrastructure is designed for scale-out computations, at large scale. In order to achieve this, applications must be designed to account for failures, as at the scale of hundreds to thousands of nodes, this is likely in any infrastructure. Configurations that appear to work well on a small scale may perform less optimally on higher workloads, even leading to failures where the software is not designed to account for the architecture. However, the flexibility afforded by a commercial cloud provider makes it possible to experiment with and engineer configurations that would be impossible on a closed in-house cluster. In the end, our target configuration outperformed the HPC cluster while giving us a specific price tag per input sample.

For very I/O bound Big Data problems the main area of improvement is in the data access. Therefore, the primary reason for performance gains was not the pipeline redesign but rather the combination of fast, local SSD storage on the VMs and its extensive use by the e-SC workflow engines. In the HPC cluster the compute nodes were equipped with regular HDD but also the pipeline relied more heavily on the parallel, network filesystem. With e-Science Central the use of local disk storage is implicit and, therefore, users would not make a mistake of using shared, network filesystem unless it is necessary to distribute work across nodes.

4.3. Phase III: testing scalability

In this phase we scaled out the deployment by allocating additional cloud nodes and adding workflow engines to the e-SC pool, one per node. The evaluation involves processing the same set of input samples over target configurations of 6 and 12 workflow engines (48 and 96 cores respectively), and comparing their response time against the 3-engine baseline from Phase II.

Fig. 11a presents the observed response time for the three configurations as the function of the input size. Fig. 11b shows system throughput (samples per day) in relation to the number of processing engines. It is compared with ideal linear speed-up and illustrates gains in the processing speed when adding workflow engines to the system.

Lastly, Fig. 11c shows how well the different configurations scale *when compared to baseline with 3 engines*, using a measure of *relative processing effectiveness* (RPE; the higher the better). Given a fixed input sample size s, if $T_s(n)$ is the response time for a configuration with n en-

gines, we define the relative processing effectiveness of the n-engine configuration relative to the b-engine configuration as:

$$RPE_s(b,n) = \frac{bT_s(b)}{nT_s(n)} \quad (1)$$

In our experimental space we have used baseline $b = 3$, $n = 6$ and $n = 12$, and s ranging from 6 to 24. 100% effectiveness is achieved when $T_s(n) = \frac{b}{n}T_s(b)$. For example, resources are perfectly utilised when doubling the number of engines ($n = 2b$) results in the halving of the response time relative to the baseline ($T_s(2b) = \frac{1}{2}T_s(b)$) on the same input size. In contrast, one of the actual data points in our chart, $RPE_{12}(3,6) = 75\%$, indicates that doubling the number of engines on the 12-sample input is only 1.5× faster than the baseline; ideally, it would be 2× faster.

Our results show that for larger configurations the response time grows slowly with the number of samples. For the smallest, 6-sample, input we observed very little gain when adding workflow engines to the system (cf. throughput). Only for the biggest, 24-sample, input the pipeline showed good effectiveness – about 86% when running on 6 engines (Fig. 11c). In our practice, however, it is common to run the pipeline with 30 input samples or more, which will allow us to scale the system to larger configurations and still reduce response time effectively.

The main reason for low effectiveness in processing small input datasets was the amount of parallelism hidden in the data that our pipeline could exploit. For N input samples during alignment we had $2 \cdot N$ pair-end reads, whilst the following steps of Stage 1 processed N aligned sequences. In Stage 2 we used a fixed (but configurable) value of 50 chunks to split the data across chromosomes. Later on, the pipeline again worked with N input samples. Given that for improved resource utilization we configured each engine to run 4 workflow invocations concurrently, for small N not all execution threads of all engines could be utilized. For example, if the system ran 12 engines, there were 48 execution threads waiting to accept invocations. Then, with only six samples of the input data the majority of these threads were idle causing less effective use of the resources.

Lessons learnt. The main insight gained from these results is that larger deployment configurations can easily be over-provisioned relative to the amount of parallelism available in the workflow. The computation/communication ratio has to be optimised, as with HPC applications. Therefore, care must be taken to best match the application workload with the appropriate services and deployment options. This situation is different to using on-premise HPC machines that have typically been pre-configured for maximum performance without reference to optimising for price/performance. In the cloud the application developer has some of the responsibilities for specifying the computing infrastructure, including system-level I/O performance. Thus, for complex applications such as WES, cloud computing can deliver enough scalability but work is required at the system-design level to ensure success and efficiency.

4.4. Cost estimation

As mentioned in the introduction, one of the advantages of adopting the cloud-based approach to NGS data processing over a closed HPC system is the ability to precisely quantify the monetary cost associated with processing one exome sample. These figures are very welcome to public health providers, who are planning to deploy WGS-based genetic testing at population scale. In the cost assessment described below, we have translated resource consumption into cost using the Azure commercial rates at the time of writing (June, 2015). Clearly, these figures only represent a point of reference, as continually decreasing prices for resource allocation on commercial clouds make them rapidly obsolete.

Our cost estimation model accounts for (i) compute time of the master (server) as well as of each compute

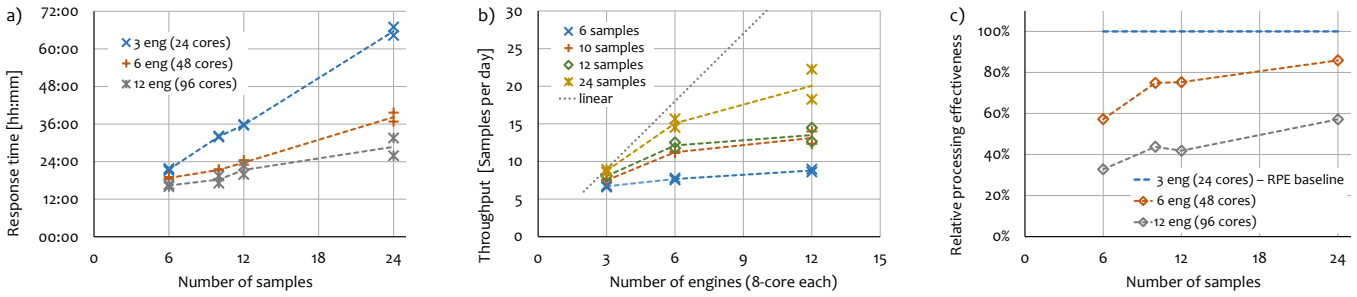

Figure 11: Response time, throughput and relative processing effectiveness of the pipeline with the increasing number of input samples and different number of workflow engines in the system.

node VMs (workflow engines), and (ii) storage usage during the run. We exclude one-off setup costs for the underlying infrastructure, as in a production system these would become insignificant. We also exclude the cost of data transfer to/from the cloud because they are negligible when compared to hiring VM. Transfer to the cloud is free in Microsoft Azure, whereas the output data is over 200x smaller than the input (about 70 MB per sample). Additionally, the first 5 GB/month of data transferred out of Azure is free.

While it is easy to measure precise uptime figures for each run, some storage costs, e.g. related to storing input and reference data, were shared between test runs and so were not easy to account precisely to a specific run. However, as Fig. 12 shows, the number of wall clock compute hours (VM uptime) dominates the cost (up to 83% of total charges over a month). Therefore, without losing too much accuracy we estimate storage cost per hour, denoted HSC, by aggregating the storage cost over the entire billing period (one month) and assuming its uniform distribution.

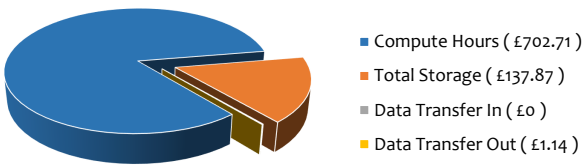

Figure 12: Cost by resource type for one billing period.

To estimate the cost C_R of one run, we use the following parameters: R^E_{CH} and R^S_{CH} are the uptime cost rates per Compute Hour for the engine and server VM, respectively; E_R is the number of engines involved in the run, and HSC is the estimated Hourly Storage Cost. Then, the cost of a single run is calculated as:

$$C_R = T \times (R^E_{CH} \times E_R + R^S_{CH} + HSC) \quad (2)$$

where T is the wall clock duration of the run expressed in hours.

Figure 13 reports on the cost per size of input data and per sample, C_R/N, using Eq. (2) for different input sample sizes N and number of engines $E_R \in \{3, 6, 12\}$; note that to run the server we used a single VM of the same size in all runs. The figure shows two data points for each configuration, with a line through their average. The exact figures used for this chart are presented in Table A.3 in Appendix A. The table shows remarkable consistency, as the values for each pair of runs are very close to each other. Actual cost figures used in Eq. (2) are R^E_{CH}=£0.47, R^S_{CH}=£0.10, and HSC=£0.21 and are up to date as of the time of writing.

This cost assessment is consistent with the results from the effectiveness chart in Fig. 11(c), which indicates that running the tests on the 3-engine configuration is the most cost-effective across all data points. Nonetheless, the significant amount of data processed and stored in the system means that for larger input data sets the storage costs can balance scalability inefficiencies. Thus, when the input

reaches 24 samples, using six engines provides a faster response time without increasing the overall cost per sample.

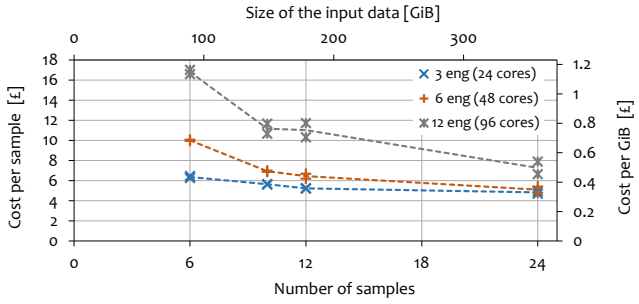

Figure 13: Cost per sample and GiB of compressed input data for the 3-, 6- and 12-engine configurations.

5. Related work

The areas of workflow, Cloud, HPC and NGS technologies have been covered by extensive literature and it is out of scope of this paper to address all of them in depth. Instead, we focus this section on the problems related to Big Data and, in particular, Next Generation Sequencing. From our and others previous experience (e.g. [18, 24, 25, 26]), achieving very good scalability properties for CPU-bound problems is possible. It is, however, very different from managing large amounts of data. Moreover, the current set of NGS tools, usually, do not require sophisticated MPI-based algorithms which make proper use of the HPC systems. Rather they need effective data splitting strategies that can turn Big Data into an embarrassingly parallel problem [27].

A number of projects have undertaken the task of automating NGS data processing such as [5, 6, 7, 8] amongst others. These pipelines run on HPC clusters or, more recently, on a cloud infrastructure. Some, like SIMPLEX, can do both.

SIMPLEX [6] is delivered as a preconfigured VirtualBox or cloud image for Amazon EC2. It can delegate compute intensive tasks to the cluster or cloud using in-house developed JClusterService. The pipeline combines a set of pre-defined tools in a fixed topology. As the only customisation is in the configuration of the tool parameters, SIMPLEX is easy to use but not very flexible. In contrast, our workflows provide a more flexible approach, where users can not only tune tool parameters, but they can also change tools and modify the design. Furthermore, e-SC keeps track of all versions of the processing tools as well as of the libraries. This makes it easier to track technology advances in the tools and to enforce reproducibility of older versions of the pipeline.

The evaluation results shown for SIMPLEX give only a limited view over its performance. They only include absolute response time for a single small-scale experiment consisting of a 10-sample input data set, with average sample size of 3.1 GB, obtained on a small HPC cluster with 128 cores and 1 TB of memory. In contrast, our experimental results shown in Section 4 include multi-lane, pair-end raw sequences reads of about 15GB compressed data each (or 37 GB of uncompressed files). Also, despite over 10-fold increase in size, our 10-sample experiment runs in less than 24 hours and using only 48 cores (6 nodes).

Bhuvaneshwar et al. [12] present the application of the Globus Genomics system to NGS analyses. Globus Genomics [28] integrates the Galaxy workflow system [29] with the Globus toolkit to improve data management capabilities and allow workflows to be scaled across cloud resources. Although there are similarities with our pipeline, there are also important differences. Firstly, the system includes data transfer to and from the Globus Genomics, whereas we focus only on data processing. Secondly, it includes automated quality control while our pipeline does not. However, we use a more recent version of the GATK variant caller, the HaplotypeCaller, as recommended by the Broad Institute in the GATK Best Practices document.[12] Lastly, our pipeline includes an automatic annotation step using ANNOVAR [21] in addition to our own

[12] https://www.broadinstitute.org/gatk/guide/best-practices

in-house annotation tool, which makes the output ready for a researcher to analyse. This also means that output from the pipeline can be directly fed into our variant interpretation tool [30].

Comparing execution performance, our pipeline offers better throughput (in GB/s) even though we use a much slower variant caller. If we were to run our pipeline with the older UnifiedGenotyper used by Globus Genomics, instead of the current HaplotypeCaller, we could further increase the throughput and reduce the response time although with negative impact on the quality of results. Unfortunately, the authors present only scalability tests in relation to the increasing size of the input data, but no information is presented on how well the system scales when more processing nodes are added.

In [31] the authors discuss ways to accelerate processing of NGS pipelines in Azure using Hadoop over Azure. The authors show performance gains at various stages of their pipeline when using techniques such as CRAM compression [32] and careful storage mapping. However, their pipeline, although similar in the overall design, was not evaluated in a way which would allow for direct comparison with ours.

Finally, Gao et al. [33] have proposed recently to follow a direction opposite to ours. They provide the `Fastq2vcf` Perl program, which can generate command-line scripts to most commonly used NGS tools including BWA, Picard and GATK. Their goal is to simplify the construction of pipelines by delivering wrapper shell scripts, which can then be executed on a desktop computer or submitted to a HPC cluster. Nonetheless, we believe that an approach where the user has control over tool and data versions, can collect data provenance and run their pipeline effectively on a set of nodes, is better in the longer term.

Similarly, Kelly et al. [27] proposed recently a WGS resequencing solution called Churchill. It is an example of a careful redesign of the algorithms used in NGS analyses to achieve fast resequencing of the whole genome sequences. These efforts resulted in near-optimal CPU utilization on a single 8-core VM and very good scalability properties, so that Churchill can run effectively on 96 cores in HPC (up to 768 cores) and 128 cores (up to 512 cores) in the Amazon AWS cloud. At the core of Churchill is a novel data splitting algorithm that allows chromosomal subregions to be processed independently yet with high sensitivity and accuracy of the variants found. However, Churchill is a highly specialized, closed solution that is specifically tailored to do WGS analyses with limited flexibility for users. It may be seen as the opposite of the workflow-based approach which offers clear insight into the pipeline structure and easy customisation of the whole pipeline and each single step.

With regards to efficiency of the NGS analyses Carrier et al. show in [34] the impact of reimplementation of NGS tool called *Trinty* using the HPC best practices. They present scalability of the tool from 32 to 2048 CPU cores for the RNA sequencing of mouse and from 256 to 8192 CPU cores for the RNA sequencing of axolotl with response time speed-up of about 20x and 7x, respectively. This example indicates two important facts. On the one hand, access to a HPC platform can offer scalability capabilities which might be difficult to achieve in the cloud, for reason as simple as prohibitively high cost. On the other hand, ability to use a large pool of resources almost never results in equivalent gains in speed-up; cf. 8192 vs 256 CPU cores (32x) with speed-up of only 7x in this case, and Amdahl's law in general. In the cloud users need to pay much more attention to the amount of resources they hire, which usually pushes towards the more efficient use of the resources.

6. Discussion and conclusions

In this paper we have presented the results from a case study aimed at increasing the scalability, flexibility, and performance of a Big Data, WES processing pipeline. We have described requirements, design challenges (Sec. 2),

and alternative workflow designs for exploiting the latent parallelism in the input data and pipeline algorithms (Sec. 3). The main results, discussed in Sec. 4, indicate that our scientific workflows, once deployed on the Microsoft Azure cloud and fine-tuned (Sec. 4.1), achieve better performance than the original HPC configuration, while at the same time provide a higher level of abstraction for the design, and potentially indefinite scalability. Our pipeline redesign efforts showed that the main reason for achieving the performance gains was not due to improved pipeline structure but rather due to the availability of VMs with fast SSD disks combined with the extensive use of the local disk resources by e-SC workflow engines. For very I/O bound Big Data problems this combination is vital, which has also been observed by Zhao et al. [35], recently.

We have discussed some of the lessons learnt from this specific exercise throughout the paper. Genomics, however, is only one of several areas of science where these porting exercises are becoming commonplace as the demand for capacity increases while the cost of cloud resources continues to decrease. Some of our conclusions are, therefore, applicable to a whole class of projects where an existing implementation, deployed on a closed HPC architecture, is replaced with a new implementation of the same processing deployed over a commercial public cloud.

As a summary, we provide a balanced view of the key benefits and drawbacks we observed during the migration. These considerations are also summarised in Table 2.

Flexibility and Scalability. The combination of workflow technology with a cloud deployment provides flexibility, in terms of the challenges listed earlier, and scalability in the volume of computing resources that can be made elastically available to face peaks of demand in the amount of data to be processed. We have demonstrated this feature by showing three different designs to parallelise the WES pipeline. However, we also expose weaknesses of the software stack which was not always able to sustain very high CPU and I/O demands. Surprisingly, the best cloud solution was not one of the more sophisticated approaches but the simplest, synchronous pipeline.

Alternative paths to migrating legacy pipelines are available, however. For instance, one may allocate virtual clusters in the cloud, e.g. using StarCluster[13] or CloudMan [36], and then simply transfer data and scripts verbatim. While this would minimise the recoding effort, it would not meet our flexibility requirements. HPC performance in the cloud is becoming available, with Microsoft Azure Big Compute supporting low-latency, high-bandwidth Infiniband services. Presently this is a unique offering though, that is atypical of the usually lower performance of the cloud than HPC (cf. [37, 38]). Thus, the only benefit of running OGE in a cloud-based virtual cluster would be flexibility in resource allocation.

Effectiveness and cost control. To date there is little evidence on how well NGS pipelines scale with an increasing number of processing nodes, and scalability is typically only measured in terms of the number of input samples. Even when absolute response time and prices are reported, there is little concern about how effective it is to use multi-node and multi-core environments. On a cloud, this approach is no longer sufficient, simply because adding resources increases cost linearly but almost never linearly reduces time (the effectiveness is reduced). For instance, in our study the fastest response from the system was always provided by the largest configuration with 12 engines. Nonetheless, as shown in Fig. 13, the most *cost effective* configuration was the 3-engine, whilst for larger input data sets the 6-engine configuration offered comparable cost effectiveness and much shorter response time.

Reproducibility. Another advantage of our approach is the automated tracking and versioning of changes to data, workflows and workflow blocks. This gives a detailed insight into which tool, data and workflow version was used

[13] http://star.mit.edu

Table 2: Advantages and disadvantages of migration from script-based HPC to workflow-based Cloud NGS pipeline.

Migration direction	Advantages	Disadvantages
Hardware and OS: HPC → Cloud	+ more flexibility if resources are required only intermittently or there is a significant variation in workload, + clear cost control; encourages to design more efficient solutions, + transparent resource upgrades; e.g. introduction of VMs with local SSD disk, + easy access to monitoring tools which give insight into the performance of the system.	− continuous access to large resources may be costly, − access to very large resources (10k+ CPU) may be prohibitively costly, − HPC resources are carefully configured to reach the highest performance; in the Cloud some of the configuration aspects need to be addressed by the user.
Middleware: cluster manager → WfMS OGE/SGE → e-SC	+ transparent provenance tracking, + portability; easy migration between different Cloud providers, + transparent caching policy makes the most of the node-local disk resources; very beneficial when combined with fast SSD disks, + easy control and management of tool and reference data versions; increased reproducibility.	− OGE/SGE is a mature and widely adopted job management system, − some tools, e.g. GATK Queue, already support OGE/SGE and can dispatch work across a number of HPC compute resources.
Programming model and abstraction: scripting → scientific workflow	+ visual design with more prominent architecture of the pipeline, + low-level aspects such as file and directory management are less important and do not obfuscate the pipeline design, + implicit file management may save users from making certain mistakes, e.g. using networked file system for tasks other than sharing data between nodes.	− scripts offer a more expressive and, sometimes, more concise language; cf. shim blocks and support for loops, − designing an effective, parallel workflow-based pipeline requires substantial level of knowledge; equivalent to designing the pipeline using scripts and SGE/OGE over HPC system.

to produce particular result. This information is necessary to reproduce the workflow, using older versions of the tools, which may be required for comparison or validation purposes. Although similar levels of version control could be achieved in a HPC setting, e.g. by means of the Environment Modules[14] package and source version control systems, when using e-Science Central this happens automatically and almost transparently to the user. Importantly, together with version information, e-SC also keeps track of data provenance, a key element in documenting the geneticists' findings and making them reproducible.

Design complexity and performance. Finally, the workflow-based approach can be criticised on the grounds of limited expressiveness of the dataflow model. While this may translate into complexity of design to exploit the available data parallelism, we have shown that this effort pays off in terms of overall performance. Our workflow-based solution deployed in the cloud is over twice as fast as the original, script-based HPC pipeline running in the exclusive access mode. NGS pipelines, usually, do not require complex parallel algorithms but rather they combine simple tools in a sequence of tasks that need to process large data files one after another. This makes visual WfMS such as e-SC a very good fit for the problem, while the benefits of the HPC environment, e.g. low latency network, play less important role.

In summary, we have shown how migration from a local, script-based HPC pipeline to a workflow-based pipeline running in a public cloud infrastructure can provide benefits in terms of speed, flexibility, scalability and cost-effectiveness for NGS. Performing such a migration requires careful execution, but can ultimately lead to more scalable and manageable solutions for scientific applications.

[14] http://modules.sourceforge.net

Acknowledgments

This work was supported in part by a grant from the NIHR Newcastle Biomedical Research Centre, and by a grant from the Microsoft Azure Research programme.

References

References

[1] L. D. Stein, B. M. Knoppers, P. Campbell, G. Getz, J. O. Korbel, Data analysis: Create a cloud commons, Nature 523 (7559) (2015) 149–151. doi:10.1038/523149a.

[2] K. Wetterstrand, DNA sequencing costs: data from the NHGRI Genome Sequencing Program (GSP) (2015).

[3] L. D. Stein, The case for cloud computing in genome informatics, Genome Biology 11 (5) (2010) 207. doi:10.1186/gb-2010-11-5-207.

[4] J. Xuan, Y. Yu, T. Qing, L. Guo, L. Shi, Next-generation sequencing in the clinic: promises and challenges, Cancer letters 340 (2) (2013) 284–295. doi:10.1016/j.canlet.2012.11.025.

[5] T. Camerlengo, H. G. Ozer, R. Onti-Srinivasan, P. Yan, T. Huang, J. Parvin, K. Huang, From Sequencer to Supercomputer: An Automatic Pipeline for Managing and Processing Next Generation Sequencing Data, AMIA Summits on Translational Science proceedings AMIA Summit on Translational Science (2012) 1–10.

[6] M. Fischer, R. Snajder, S. Pabinger, A. Dander, A. Schossig, J. Zschocke, Z. Trajanoski, G. Stocker, SIMPLEX: Cloud-Enabled Pipeline for the Comprehensive Analysis of Exome Sequencing Data, PLoS ONE 7 (8) (2012) e41948. doi:10.1371/journal.pone.0041948.

[7] J. G. Reid, A. Carroll, N. Veeraraghavan, M. Dahdouli, A. Sundquist, A. English, M. Bainbridge, S. White, W. Salerno, C. Buhay, F. Yu, D. Muzny, R. Daly, G. Duyk, R. A. Gibbs, E. Boerwinkle, Launching genomics into the cloud: deployment of Mercury, a next generation sequence analysis pipeline, BMC Bioinformatics 15 (1) (2014) 30. doi:10.1186/1471-2105-15-30.

[8] A. Dander, S. Pabinger, M. Sperk, M. Fischer, G. Stocker, Z. Trajanoski, SeqBench: Integrated solution for the management and analysis of exome sequencing data, BMC Research Notes 7 (1) (2014) 43. doi:10.1186/1756-0500-7-43.

[9] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M. R. Speicher, J. Zschocke, Z. Trajanoski, A survey of tools for variant analysis of next-generation genome sequencing data, Briefings in Bioinformatics 15 (2) (2014) 256–278. doi:10.1093/bib/bbs086.

[10] D. C. Koboldt, L. Ding, E. R. Mardis, R. K. Wilson, Challenges of sequencing human genomes, Briefings in Bioinformatics 11 (5) (2010) 484–498. doi:10.1093/bib/bbq016.

[11] H. Hiden, S. Woodman, P. Watson, J. Cala, Developing cloud applications using the e-Science Central platform, Philosophical transactions. Series A, Mathematical, physical, and engineering sciences 371 (1983). doi:10.1098/rsta.2012.0085.

[12] K. Bhuvaneshwar, D. Sulakhe, R. Gauba, A. Rodriguez, R. Madduri, U. Dave, L. Lacinski, I. Foster, Y. Gusev, S. Madhavan, A case study for cloud based high throughput analysis of NGS data using the Globus Genomics system, Computational and Structural Biotechnology Journal (2014) 1–11doi:10.1016/j.csbj.2014.11.001.

[13] J. Cała, Y. Xu, E. A. Wijaya, P. Missier, From Scripted HPC-Based NGS Pipelines to Workflows on the Cloud, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2014, pp. 694–700. doi:10.1109/CCGrid.2014.128.

[14] I. B. Blanquer, G. Brasche, J. Cala, F. Gagliardi, D. Gannon, H. Hiden, H. Soncu, K. Takeda, A. Tomás, S. Woodman, Supporting NGS pipelines in the cloud, EMBnet.journal 19 (A) (2013) 14. doi:10.14806/ej.19.A.625.

[15] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-Science: An overview of workflow system features and capabilities, Future Generation Computer Systems 25 (5) (2009) 528–540. doi:10.1016/j.future.2008.06.012.

[16] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2014) 17–35. doi:10.1016/j.future.2014.10.008.

[17] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, C. Goble, The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud., Nucleic acids research 41 (Web Server issue). doi:10.1093/nar/gkt328.

[18] J. Cała, H. Hiden, S. Woodman, P. Watson, Cloud computing for fast prediction of chemical activity, Future Generation Computer Systems 29 (7) (2013) 1860–1869. doi:10.1016/j.future.2013.01.011.

[19] H. Li, R. Durbin, Fast and accurate long-read alignment with Burrows-Wheeler transform., Bioinformatics (Oxford, England) 26 (5) (2010) 589–95. doi:10.1093/bioinformatics/btp698.

[20] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, M. A. DePristo, The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data, Genome Research 20 (9) (2010) 1297–1303. doi:10.1101/gr.107524.110.

[21] K. Wang, M. Li, H. Hakonarson, ANNOVAR: functional annotation of genetic variants from high-throughput sequencing data, Nucleic Acids Research 38 (16) (2010) e164–e164. doi:10.1093/nar/gkq603.

[22] D. Hull, R. Stevens, P. Lord, C. Wroe, C. Goble, Treating shimantic web syndrome with ontologies, in: J. Domingue, L. Cabral, E. Motta (Eds.), AKT Workshop on Semantic Web Services, Milton Keynes, UK, 2004, pp. 1–8.

[23] I. Sadooghi, J. Hernandez Martin, T. Li, K. Brandstatter, Y. Zhao, K. Maheshwari, T. P. P. d. L. Ruivo, S. Timm, G. Garzoglio, I. Raicu, Understanding the Performance and Potential of Cloud Computing for Scientific Applications, IEEE Transactions on Cloud Computing PP (99) (2015) 1–1. doi:10.1109/TCC.2015.2404821.

[24] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, I. T. Foster, Turbine: A distributed-memory dataflow engine for high performance many-task applications, Fundamenta Informaticae (2012) 1001–1032doi:10.3233/FI-2012-0000.

[25] T. G. Armstrong, J. M. Wozniak, M. Wilde, K. Maheshwari, D. S. Katz, M. Ripeanu, E. L. Lusk, I. T. Foster, ExM: High level dataflow programming for extreme-scale systems, in: 4th USENIX Workshop on Hot Topics in Parallelism (HotPar), poster, Berkeley, CA, 2012.

[26] D. de Oliveira, K. A. Ocaña, E. Ogasawara, J. Dias, J. Gonçalves, F. Baião, M. Mattoso, Performance evaluation of parallel strategies in public clouds: A study with phylogenomic workflows, Future Generation Computer Systems 29 (7) (2013) 1816–1825. doi:10.1016/j.future.2012.12.019.
URL http://www.sciencedirect.com/science/article/pii/S0167739X13000034

[27] B. J. Kelly, J. R. Fitch, Y. Hu, D. J. Corsmeier, H. Zhong, A. N. Wetzel, R. D. Nordquist, D. L. Newsom, P. White, Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics, Genome Biology 16 (1) (2015) 6. doi:10.1186/s13059-014-0577-x.

[28] B. Liu, R. K. Madduri, B. Sotomayor, K. Chard, L. Lacinski, U. J. Dave, J. Li, C. Liu, I. T. Foster, Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses, Journal of Biomedical Informatics 49 (2014) 119–133. doi:10.1016/j.jbi.2014.01.005.

[29] J. Goecks, A. Nekrutenko, J. Taylor, T. G. Team, Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life

sciences, Genome Biology 11 (8) (2010) R86. doi:10.1186/gb-2010-11-8-r86.

[30] P. Missier, E. Wijaya, R. Kirby, SVI: A Simple Single-Nucleotide Human Variant Interpretation Tool for Clinical Use - Springer, in: N. Ashish, J.-L. Ambite (Eds.), Data Integration in the Life Sciences, Vol. 9162, Springer International Publishing, 2015, pp. 180–194. doi:10.1007/978-3-319-21843-4{_}14.

[31] N. M. Mohamed, H. Lin, W. Feng, Accelerating Data-Intensive Genome Analysis in the Cloud, in: Proceedings of the 5th International Conference on Bioinformatics and Computational Biology, Honolulu, 2013.

[32] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, E. Birney, Efficient storage of high throughput DNA sequencing data using reference-based compression, Genome Research 21 (5) (2011) 734–740. doi:10.1101/gr.114819.110.

[33] X. Gao, J. Xu, J. Starmer, Fastq2vcf: a concise and transparent pipeline for whole-exome sequencing data analyses, BMC Research Notes 8 (1) (2015) 72. doi:10.1186/s13104-015-1027-x.

[34] P. Carrier, B. Long, R. Walsh, J. Dawson, C. P. Sosa, B. Haas, T. Tickle, T. William, The Impact of High-Performance Computing Best Practice Applied to Next-Generation Sequencing Workflows, bioRxiv (2015) 017665doi:10.1101/017665.

[35] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, I. Raicu, High-Performance Storage Support for Scientific Applications on the Cloud, in: Proceedings of the 6th Workshop on Scientific Cloud Computing - ScienceCloud '15, ACM Press, New York, New York, USA, 2015, pp. 33–36. doi:10.1145/2755644.2755648.

[36] E. Afgan, B. Chapman, J. Taylor, CloudMan as a platform for tool, data, and analysis distribution, BMC Bioinformatics 13 (1) (2012) 315. doi:10.1186/1471-2105-13-315.

[37] E. Walker, Benchmarking Amazon EC2 for High-Performance Scientific Computing, ;login: 33 (5) (2008) 18–23.

[38] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, D. Epema, A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing, in: Dimiter R. Avresky, M. Diaz, A. Bode, B. Ciciani, E. Dekel (Eds.), Cloud Computing, Springer Berlin Heidelberg, 2010, Ch. 4, pp. 115–131. doi:10.1007/978-3-642-12636-9{_}9.

Appendix A. Detailed report on duration and costs of all experiments

Table A.3: Duration and cost* in £ of the tests ran on the 3-, 6- and 12-engine configuration.

Samples number	3-engine configuration				6-engine configuration				12-engine configuration			
	Duration [hh:mm]	Total test cost (£)	Cost per sample [£]	Cost per GiB [£]	Duration [hh:mm]	Total test cost [£]	Cost per sample [£]	Cost per GiB [£]	Duration [hh:mm]	Total test cost [£]	Cost per sample [£]	Cost per GiB [£]
6	21:52	**37.62**	6.27	.452	19:17	60.41	10.07	.725	17:09	102.11	17.02	1.226
6	22:32	**38.75**	6.46	.465	19:04	59.74	9.96	.717	16:44	99.63	16.60	1.196
10	32:38	56.15	5.62	.385	21:57	68.74	6.87	.472	19:35	116.62	11.66	.801
10	32:56	56.67	5.67	.389	22:20	69.94	6.99	.480	17:55	106.70	10.67	.732
12	36:38	63.04	5.25	.360	23:47	74.48	6.21	.426	20:47	123.76	10.31	.708
12	36:23	62.60	5.22	.358	25:37	80.25	6.69	.459	23:39	140.77	11.73	.805
24	65:44	113.10	**4.71**	**.316**	41:25	129.70	5.40	.363	31:50	189.52	7.90	.530
24	69:05	118.87	4.95	.333	36:47	115.21	**4.80**	**.322**	26:47	159.44	6.64	.446

*Unit costs used in the calculations: D2 VM £0.10/hour, D13 VM £0.47/hour, Geo Redundant (GR) Block Blob £0.029/GB, GR Page Blob £0.058/GB.