Genome Analysis in a Dynamically Scaled Hybrid Cloud

Chris Smowton*, Georgiana Copil†, Hong-Linh Truong†, Crispin Miller* and Wei Xing*
*CRUK Manchester Institute
University of Manchester, Manchester, UK
Email: {Chris.Smowton, Crispin.Miller, Wei.Xing}@cruk.manchester.ac.uk
†Institute of Information Systems
Vienna University of Technology, Wien, Austria
Email: {e.copil,truong}@dsg.tuwien.ac.at

***Abstract*—In this paper, we explore the benefits of automatically determining the degree of parallelism used to perform genetic mutation calling in a hybrid cloud environment. We propose algorithms to automatically control both the hiring of hybrid cloud resources and the selection of the degree of parallelism employed in analysis tasks executed against that cloud. Using the Broad Institute's Genome Analysis Toolkit as a case study, we then conduct profile-driven simulation studies to characterise the circumstances in which our algorithms are beneficial or deleterious compared to simple, conventional baseline algorithms. We find that there are a wide range of cloud workload scenarios where our algorithms outperform the baselines, and thereby argue that automatic control of cloud scaling and task parallelism, using techniques like those proposed, are likely to be beneficially applicable to real-world biocomputing.**

***Keywords*-cloud computing; auto-scaling; dynamic scalability; thread-level parallelism; genome analysis; biocomputing**

I. Introduction

Modern biology increasingly relies upon computer processing to relate experimental data to known information stored in electronic form, such as mapping short DNA reads against the human genome [1]–[3]. As a result, biological laboratories often purchase large high-performance computing (HPC) facilities, or spend significant proportions of their budget hiring compute resources from an academic or public cloud provider. However, in neither case is their purchasing as efficient as possible: it is common for HPC facilities to spend large proportions of their time idle, or for external cloud purchasing to be sized in an ad-hoc manner, either manually or with minimal automation. Applying intelligent algorithms to determine what cloud resources to hire, and how to distribute resources between different analysis jobs, has the potential to improve resource usage efficiency and thus represent a significant saving to biological laboratories and institutions.

Existing systems such as Jockey [4] and Quasar [5] provide the kind of intelligent control required, regulating the resources assigned to cloud workloads in order to achieve quality-of-service goals specified by their users. However, they have typically been applied to services designed with this sort of dynamic rescaling in mind, such as scalable persistent services (e.g. a distributed filesystem) or distributed execution frameworks (e.g. Hadoop or Dryad).

In this paper, we apply similar automatic resource allocation methods to classical HPC workloads that are not designed for dynamic scalability, and present candidate algorithms to control cloud resource hiring (*horizontal scaling*) and assignment of resources to jobs (*resource allocation*) with an HPC-style workload. We describe a case study applying our candidate algorithms to the Broad Institute's Genome Analysis Toolkit (GATK) [6], and show using discrete event simulation that its performance in a hybrid cloud environment can be significantly improved compared to more straightforward resource allocation methods. We outline a model of a hybrid cloud environment and its users, describe our discrete event simulator, and explore the benefits achieved by applying dynamic scaling in diverse circumstances.

We contend that these promising initial results suggest that automatic dynamic scaling could be profitably applied to a broader range of classical HPC applications, and outline potential directions for this future work.

II. Background

The hybrid cloud resource allocation problem has been richly explored. To give some representative examples, Van den Bossche et al. described an algorithm for scheduling tasks in order to meet their deadlines, including predicting the cost and likely performance of execution in multiple possible public cloud environments [7]. Malawski et al. analysed scheduling DAGs of jobs, taking into account the knock-on delays to future tasks caused by delaying those that are currently runnable [8]. Kim et al. investigated scheduling against a hybrid cloud, and additionally introspected on running and past tasks in order to predict the likely performance characteristics of future tasks [9].

All three of these studies investigated kinds of workloads typically encountered in an HPC environment: the structure of the job, or graph of jobs is known in advance, and the degree of parallelism exploited is configured by the user submitting the job for execution – for example, they might submit a graph

Correspondence: Wei.Xing@cruk.manchester.ac.uk

of jobs annotated with the number of concurrent cores required by each node for optimal execution. Thus they all considered automating horizontal scaling of the compute cluster (how much public compute time to hire, and in some cases where to hire it from), but did not consider automating the allocation of resources to jobs.

Several other systems *do* automatically allocate and reallocate ongoing tasks' resources in order to help them achieve their performance goals. Ferguson et al. designed Jockey [4] to monitor on-going cluster workloads, such as MapReduce and Dryad jobs, and dynamically adjust the number of concurrent cores (and other resources) they are provided with in order to keep them on schedule. Delimitrou et al. wrote the Quasar system with a similar approach and similar goals: Quasar could adjust the resources dedicated to ongoing jobs based on online monitoring and automated offline profiling data, including considering inter-job interference effects [5]. Han et al. designed a lightweight platform to adjust the resources available to on-going jobs or services by adjusting the specification of the virtual machine they inhabit [10], while Tian et al., Schwarzkopf et al. and Agarwal et al. all considered giving a cloud or cluster scheduler control over the execution plan of a distributed execution framework such as MapReduce or Dryad [11]–[13], effectively placing the scheduler in charge of parallelism.

These two groups of studies respectively show that automatically controlling the hybrid cloud placement of jobs with fixed resource requirements can be beneficial, and that automatically *determining* the degree of parallelism and thus resource requirements used in distributed execution environments can likewise improve job performance and user quality of service. In this paper we extend the prior work on HPC-style workloads to consider the automatic scaling techniques that have been applied to persistent services and distributed execution frameworks designed for the cloud. Where prior work with HPC workloads has only considered scheduling jobs with externally specified resource requirements, we give our scheduler control of the degree of parallelism employed by each job, analogous to the automatic MapReduce or Dryad planning described above.

Whilst the work is described in terms of a hybrid cloud, it also applies to any situation where multiple resource tiers are available for hire; for example, different tiers of HPC resources (in-house, local, national) or different cloud providers (a federated cloud).

III. Model

In this section we describe our model of the GATK application, a hybrid cloud and its resource manager, and the users who submit analysis tasks (in this paper, specifically GATK analysis tasks) for execution in the hybrid cloud. We also describe the known shortcomings of our model compared to the real world.

A. Application

The GATK application consists of a set of tools that are usually built into a pipeline consisting of multiple *stages*. The first stage consumes the user's input data; every other stage depends on the full output of its predecessor. Each stage runs as a separate process, whose degrees of thread-level parallelism can be independently configured. We consider a particular 7-stage mutation detection pipeline: the user submits aligned DNA or RNA reads in Binary Aligned Map (BAM) format [14], and receives a list of suspected mutations. Many of the GATK tools used in this pipeline require a great deal of I/O bandwidth, but they are ultimately compute- or memory-bound rather than I/O-bound. Most (but not all) GATK tools can be accelerated by local multi-threading, which is normally set manually, but in this paper will be controlled by our resource allocation algorithm. The degree of multi-threading is fixed on startup, but can differ between pipeline stages. The stages communicate via intermediate files. We model the typical use case where the stages are run sequentially: whilst it is possible to chain some stages together using FIFOs, this can lead to wasted CPU time as stages become bound by their predecessor's output rate.

We model GATK pipeline stages with single-threaded execution time that is a linear function of the size of the first stage's input data. Thus for each pipeline stage i we can specify coefficients a_i and b_i such that execution time E_i can be given in terms of input data size d:

$$E_i(d) = a_i d + b_i$$

We compute multi-threaded execution time assuming that the single-threaded execution time for a particular stage may be split by a constant factor c_i into a sequential part and a part which scales perfectly. Thus threaded execution time using t threads $T_i(t, d)$ relates to $E_i(d)$ by:

$$T_i(t, d) = c_i \frac{E_i(d)}{t} + (1 - c_i) E_i(d)$$

Thus adding more and more threads to a particular pipeline stage yields diminishing returns in accordance with Amdahl's law. The values of a_i, b_i and c_i were determined for each pipeline stage by linear regression of offline profiling data. The profile measured the time taken to analyse a variety of input sizes and thread counts. We found these simple models represented the profiling data very accurately [15].

B. Hybrid Cloud

We model a hybrid cloud with two tiers,*private* and*public*. Both have a constant cost per core per unit time, with private cores being cheaper. The private tier represents the owned compute resources which are common in scientific institutions, whose cost may represent an internal incentive for fair sharing. The public tier represents machines hired from a public provider. Apart from differences in cost, machines from the two tiers are used and managed in the same way. The model could trivially be extended to consider more tiers (for example, many different cloud computing providers or

products); however we restrict ourselves to two tiers here for simplicity of exposition.

We model both tiers as a fungible pool of cores that can be assigned to stages in any combination. In practice this is effectively true of the public tier, as providers sell virtual machines of sufficiently variable size. In the private tier, in practice cores do go to waste because of problems optimally bin-packing stages onto physical nodes; however we assume that this phenomenon has negligible impact in practice and do not model it. We assume that private and public cores perform the same: although this may not be true in practice, any slowdown caused by hardware differences could be approximately modelled by adjusting the cost of public vs. private cores. We also assume both tiers have a constant price per core per unit time (i.e. we do not model variable cost products such as Amazon EC2's spot pricing).

To begin with, for simplicity we assume that the link between the private and public tier machines is sufficiently high-bandwidth that we can use a shared-store model – that is, a stage can run on either tier without moving data in advance, and the GATK pipeline stages do not become I/O bound due to executing in one location or the other. The cost of moving data between sites is not explicitly modelled; however if compute cost is roughly proportional to the cost of importing data to compute upon, then it can be factored into the public tier's price per core. Later in the paper we will consider the impact of an explicit data transfer phase.

C. Resource Manager

The resource manager maintains a single FIFO queue of runnable pipeline stages, and assigns private and public cores to each in turn. It implements a *horizontal scaling algorithm* that decides, for each stage, whether to execute it using the private tier (which may entail waiting for free resources) or the public tier (which is always available, having effectively unlimited size, but is more expensive than the private tier). It also implements a *resource allocation algorithm* that decides, in cooperation with the horizontal scaling algorithm, how many cores to assign each stage as it starts execution, and so how much thread-level parallelism each stage can employ. The scaling algorithm and the allocation algorithm both seek to maximise profit: the difference between the core hiring cost and the reward offered by users.

Whilst this resource manager model is unusually advanced in taking charge of both scaling and resource allocation, it does not incorporate some features that are common in practically deployed managers, such as dispatching stages in priority rather than FIFO order, accounting for task interference effects, or introspecting on running jobs in order to construct a performance model or determine whether they are performing as the model predicts.

D. Users

We assume that many independent users are submitting GATK analysis pipeline requests to our resource manager, and that they all offer reward for completing the entire pipeline (i.e. they do not care how long individual pipeline stages take). They also all offer reward on the same terms (i.e. there are no users with more urgent work than others willing to offer a premium for fast execution).

Since the users are assumed to be acting independently, we model first pipeline stage arrival as a Poisson process, with each arrival carrying a batch of analysis pipeline requests. The batch size, and the input sizes of each request, are normally distributed. The parameters of these distributions can be varied to adjust the stress on the resource manager in terms of arrival rate and its variability. We experimented briefly with an alternative user model in which the mean arrival rate exhibited long-term variation, but found that this did not significantly effect our experimental results. We assume throughout that the GATK workload is the only demand on our pool of cores; in reality there would be other workloads vying for the same resources.

We consider two possible schemes under which the users offer reward for completing their requested analysis pipeline runs: the *time-oriented* scheme and the *throughput-oriented* scheme.

1) Time-oriented Reward: Under the time-oriented reward scheme, users offer a reward proportional to input size for completion of the whole GATK pipeline, with a constant penalty per unit time the work is delayed. Thus the reward R given for finishing an analysis with input data size d in total time t is related by constants R_{max} and $R_{penalty}$:

$$R(d,t) = d(R_{max} - tR_{penalty})$$

This scheme represents the case where the primary undesirable factor is the time wasted by users waiting for analysis pipelines to complete: they are paid proportional to time, and so the penalty for delaying work is similarly a linear function of time.

2) Throughput-oriented Reward: Under the throughput-oriented reward scheme, users offer a reward proportional to the *rate* at which analysis requests are completed, and thus inversely proportional to the time to run the complete pipeline. It can be given in terms of input data size d and analysis time t in terms of a scaling factor R_{scale}:

$$R(d,t) = \frac{dR_{scale}}{t}$$

This scheme represents the case where the user is more concerned with relative speedup than with the total time consumed for a particular run: whilst the time-oriented scheme values any minute it can save equally, the throughput-oriented scheme rewards according to the *proportion* of runtime that was eliminated.

3) Alternative Schemes: In practice users may value attributes of their analysis pipelines other than their simple execution time, such as the system's ability to reliably predict when they will complete, or the variability of their duration. We leave these alternative possibilities as future work.

IV. Scaling Algorithms

In this section, we describe our proposed horizontal scaling and resource allocation algorithms and the baseline algorithms we compare them against. The baselines we consider here are kept simple to facilitate easy analysis; we aim to establish that our dynamic algorithms have potential, rather than to establish that they are best in class. In future we will investigate how the opportunity for dynamic resource allocation can be exploited as part of more practical existing algorithms.

A. Horizontal Scaling

The resource manager's horizontal scaling algorithm selects, for each pipeline stage that reaches the front of the queue, whether to execute it using private or public tier resources.

We consider two baseline horizontal scaling algorithms for comparison. The first is the *always-scale* algorithm, which always hires public tier resources if all private-tier resources are currently occupied. Under this scheme, tasks never queue. The second is the *never-scale* algorithm, which never hires public resources at all, modelling a conventional local HPC cluster.

Competing with these two baselines, we propose a scheme called *predictive horizontal scaling*. If all private cores are occupied, this algorithm weighs the cost of public cores against the loss of reward that will result from waiting for private cores to become free.

Our predictive algorithm keeps track of analysis stages that are currently running in private tier resources, and an *expected completion time* for each one, which is computed using the application model given in section III-A. Thus it can compute the *expected queueing time* for each task in the queue, which is simply the time until sufficient expected completion times have passed that enough private tier cores have been released to run the task. The expected execution time for the whole pipeline run is also computed, with the precise method depending on the resource allocation algorithm in use, since it depends on how many cores will be dedicated to future pipeline stages.

Writing q for the current queue of runnable pipeline stages, writing Q_i for the time a stage i in that queue is expected to wait to reach the front, and writing E_i for the expected *total execution time for i's whole pipeline, including this stage*, the predictive algorithm computes the reward loss R_{loss} due to waiting for the stage h at the head of the queue to acquire private resources:

$$R_{loss}(q,h) = \sum_{i \in q} (R(d_i, E_i + Q_i) - R(d_i, E_i + Q_i - Q_h))$$

Thus, writing C_{public} and $C_{private}$ for the two tiers' prices and S_h for the expected execution time of the *single* pipeline stage currently at the head of the queue, it calculates the profit due to running stage h using public resources:

$$P(q,h) = R_{loss}(q,h) - S_h(C_{public} - C_{private})$$

This simply weighs the extra cost due to running in a more expensive resource tier compared to the reward loss resulting from delaying every task in the queue. The expected reward is recomputed each time the queue's state changes, either because of an analysis stage being dequeued and dispatched, or a new stage added to the end of the queue. The next task is dispatched against private cores whenever they become available, or against public cores whenever this formula indicates doing so would be profitable.

When public resources are expensive, this has the practical result of roughly establishing a threshold queue size beyond which hiring public resources is considered profitable. This is because the expected reward from reducing the delay experienced by every stage in the queue, on average, scales linearly with the length of the queue.

One problem that this algorithm experiences is how to deal with inaccurate predictions regarding a task's completion time. The simplest version assumes that a running task that should already have finished, but has not, will complete immediately. Later in this paper we will investigate alternative approaches to predicting when late stages will complete.

B. Resource Allocation

As described previously, each GATK analysis stage has distinct scaling behaviour with respect to the degree of thread-level parallelism employed. Some stages scale near-perfectly, whilst others do not scale at all. The resource allocation algorithm's task is to consider their scaling behaviour and the current level of cloud resource utilisation, and thus choose what degree of thread-level parallelism should be used for each pipeline stage when it starts execution. The resource allocation algorithm can also be called upon to predict how many cores it will assign a future pipeline stage, in order to help the horizontal scaling algorithm estimate whole pipeline execution times.

The baseline resource allocation algorithm allocates a constant degree of parallelism to each pipeline stage; in our figures we call the algorithm *best constant plan*. The allocation may differ from one pipeline stage to another, but does not depend on current resource utilisation. In effect we take the maximum profit achievable over all possible constant execution plans, although in practice we use a hill-climbing algorithm to lessen the cost of finding the best plan.

We propose three variants of our dynamic resource allocation algorithm, all of which attempt to account for current levels of resource utilisation when picking how many cores to assign the next runnable task, but which vary slightly in their priorities. Our goal in proposing three variants is to explore the relative merits of their contrasting choices of optimism and pessimism concerning the future, and of fixed vs. adaptive short-term planning.

The first, called *greedy* resource allocation, calculates the anticipated reward if the next pipeline stage were executed with a variety of possible levels of parallelism, assuming that future stages of the same pipeline run single-threaded. Thus, if available resources permit, it will achieve maximum reward *now* by running the current pipeline stage with a high degree of parallelism. It will be discouraged from doing so either

because the additional resources do not reduce execution time enough to generate sufficient extra reward, or because public-tier resources would be needed and they are too expensive. When predicting *future* stage parallelism, it pessimistically assumes it will be forced to run them single-threaded.

Our second proposed algorithm, called *long-term* resource allocation, takes the opposite approach: rather than try to maximise reward from each pipeline stage as it becomes runnable, it selects an execution plan for *all* pipeline stages when the *first* one becomes runnable, basing the reward it anticipates from doing so on the *current* degree of resource utilisation. The execution plans it considers represent a spectrum ranging from optimal resource usage (a purely single-threaded execution) to optimal profit when running on private infrastructure (assigning extra cores to the most scalable pipeline stages). The array of possible plans are pre-computed by calculating the expected profit from running every possible permutation of thread counts assigned to each pipeline stage. These plans are sorted by profit and filtered such that expected profit increases monotonically with CPU time invested (i.e. plans are eliminated whenever they would invest more resources and achieve a worse outcome than some other plan). Ideally we would consider every such plan whenever a new pipeline run is about to begin; however doing so is too computationally costly, so we sample the list 10 times at regular intervals to produce a shortlist, which is consulted per arrival to yield an approximation to the best possible execution plan. Since the execution plan is decided once and for all when the first stage starts to run, when asked to predict future pipeline stage parallelism it can do so accurately.

The third algorithm, called *long-term adaptive* resource allocation, represents a compromise between the first two: it precomputes a mapping from a task's *history* (how many cores were employed in each stage that has already run) to the *ideal* plan for the rest of the job assuming private tier resources. It picks how many cores to actually assign using the same algorithm as greedy scaling, but limited not to exceed the ideal case. For example, if the ideal plan for a new job indicated the first pipeline stage should use 2 threads, this algorithm would weigh the anticipated profit from using either one or two threads, but not more. If it deviates from the ideal plan due to current circumstances, then upon running the second stage it will look up the *ideal plan given that stage one ran single-threaded*, which will most likely assign more cores to later stages to make up for lost time, or on the other hand might note that given the slowdown experienced so far, it is no longer worth investing extra resources in later stages. The long-term adaptive algorithm predicts future stage parallelism by optimistically assuming the ideal plan given the pipeline's history will in fact be used.

The simplest versions of these algorithms tend to snap between two different execution plans: the optimal scaling for private and public tier resources. However, all can be modified by a *sharing factor*, under which the algorithms assume that the present overall workload will shortly increase by that factor and compute the best option for *all* of these hypothetical stages rather than just the stage that is presently under consideration. This represents the consideration that if another stage arrived right now with similar needs it would be better overall if they shared them, rather than run in series or hire public resources for the other task. Setting the sharing factor may be beneficial because it encourages fair sharing, or deleterious when resources are left available for future tasks that never in fact materialise.

V. Implementation

We implemented a discrete event simulator realising the model and scaling algorithms described above in roughly 1,000 lines of Python. It models the queue of runnable pipeline stages, the pool of private and public cores available to run them, our baseline and candidate scaling and allocation algorithms, and the GATK application itself, the latter being accomplished by predicting GATK execution times using the equations given in III-A.

Our model gives formulae for *predicting* pipeline stages' execution time given their characteristics; however, in reality these predictions will not always be accurate. Our simulator accounts for this by multiplying the predicted time by a factor drawn from a normal distribution centred at one, constrained to the range $(0.1, 1.9)$, and with configurable standard deviation. Thus all algorithms experience a world where their predictions are *on average* correct, but with variable error.

In future work we may extend the simulator to model interference effects which may reduce the performance of multiple jobs running concurrently below that observed for each in isolation. However, for the purposes of this paper we continue to assume that no systematic error is introduced to the tasks' predicted runtime.

The simulator source is available at `https://github.com/smowton/scan/tree/escience/sim`.

VI. Simulation Results

We explored the parameter space of our model and its simulated universe in order to determine how our scaling and resource allocation algorithms behave compared to the baselines given, and compared to each other. Whilst the arbitrary nature of the simulated cost and time units means we cannot draw conclusions that are concretely applicable to running the GATK application in the real world, we can suggest how and why certain algorithms produce good behaviour in simulation, and therefore present interesting directions for further study. Our goal is not to establish that our proposed algorithms are *optimal*, but rather to establish when and why they are beneficial or harmful compared to their respective baselines, which we assert are representative of typical simple policies seen in practice.

The arbitrary time and cost units employed throughout these simulations are denoted TUs and CUs respectively.

A. Experimental Configuration

We ran a very large number of simulation sessions, varying many different attributes of the simulated situation. Table

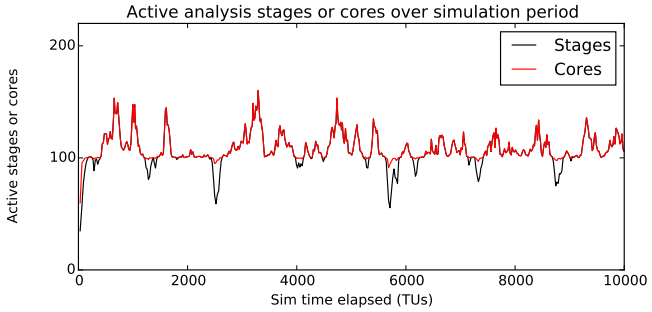

Fig. 1. Trace of active pipeline stages and allocated cores for a system with mean inter-arrival interval 2.0 TUs.

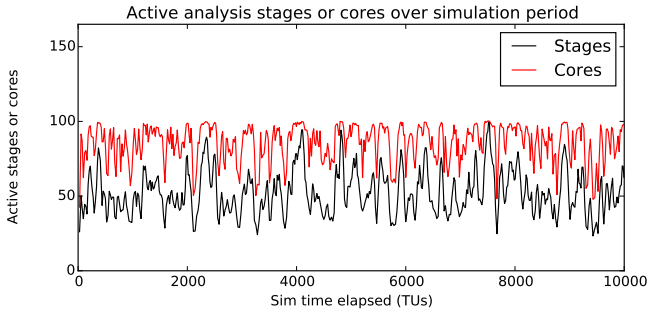

Fig. 2. Trace of active pipeline stages and allocated cores for a system with mean inter-arrival interval 3.0 TUs.

I gives the variable and fixed parameters we used (recall that the various R parameters influence the reward given for completing a pipeline run in a given time, as described in III-D). It also gives the fixed a_i, b_i and c_i parameters (described in section III-A) that determine the relationship between input data size, thread count and execution time for each pipeline stage. The scalability factors were derived from profiling of the real GATK with a variety of thread counts and input data sizes; the other parameters were selected by trial and error in order to exhibit a cross-section of the system's behaviour. The arrival batch size and job size parameters were chosen to produce significant short-term workload variation, such that the scaling and resource allocation algorithms would experience a wide range of cloud utilisation during a given simulation run.

All our experiments measure how performance varies with changing system workload by varying the mean job inter-arrival interval. For context, a mean interval of 2.0 time units represents a very busy system where much public resource hiring is necessary to keep the task queue from growing out of control, whilst a mean interval of 3.0 time units corresponds to a quiet system where the private resource tier is rarely if ever fully occupied. Figures 1 and 2 show smoothed traces for each of these two extremes. Both examples run with our predictive horizontal scaling and greedy resource allocation algorithms enabled; note their clear preference for running with 100 cores allocated (the size of the private tier). To give an idea of scale, a total of roughly 10,000 and 15,000 pipeline runs completed for the lighter and heavier workloads exhibited respectively. Whenever a best constant plan result is depicted, the best plan has been determined *per value of the independent variable*; profit would be reduced if a single constant plan were selected

Variable Parameter		**Values**	
Resource allocation algorithm		Greedy, long-term, long-term adaptive, best constant	
Horizontal scaling algorithm		Always-scale, never-scale, predictive scaling	
Mean job inter-arrival interval		2.0, 2.1 ... 3.0 TUs	
Task completion reward function		Time-based, throughput-based	
Public tier core cost (CUs / TU)		20, 50, 80, 110	
Fixed Parameter		**Value**	
Simulation time (TUs)		10,000	
Private tier core cost (CUs / TU)		5	
R_{max} (CUs)		400	
$R_{penalty}$ (CUs)		15	
R_{scale} (CUs / TU)		15,000	
Possible instance sizes (cores)		1, 2, 4, 8, 16	
Mean jobs per arrival event		3	
Jobs per arrival variance		2	
Mean job size (arbitrary units)		5	
Job size variance		1	
Pipeline stage	$\mathbf{a_i}$	$\mathbf{b_i}$	$\mathbf{c_i}$
1	0.35	5.38	0.89
2	2.70	-0.53	0.02
3	1.74	3.93	0.69
4	3.35	0.53	0.79
5	1.03	17.86	0.91
6	0.02	0.39	0.25
7	0.01	5.10	0.02

TABLE I
VARIABLE AND FIXED SIMULATION ATTRIBUTES, AND PIPELINE STAGE PARAMETERS

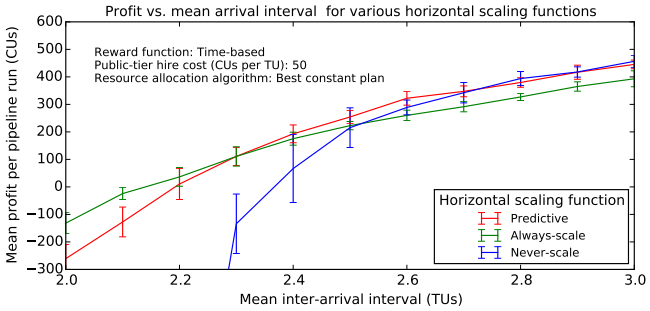

Fig. 3. Horizontal scaling algorithm vs. profit

to use across all workload levels.

Due to space limitations we will not present all of our experimental data, but instead will call out and illustrate specific interesting subsets. When figures show the two different reward schemes under consideration, note that the absolute values of profit achieved are not directly comparable because the rewards given by the two schemes are arrived at using different formulae, and so are themselves incomparable.

All measurements in this section were repeated 10 times, and all error bars represent a single standard deviation either side of the mean over these 10 repetitions.

B. Horizontal Scaling

We found that our predictive horizontal scaling algorithm achieves a sound compromise between the behaviour of the always-scale and never-scale baselines when running with a constant resource allocation policy (i.e. when every run for a particular data point uses the same execution plan). Figure 3 shows an example of this behaviour, where the

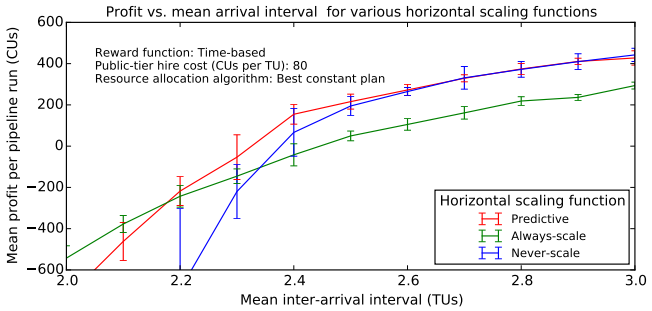

Fig. 4. Horizontal scaling algorithm vs. profit

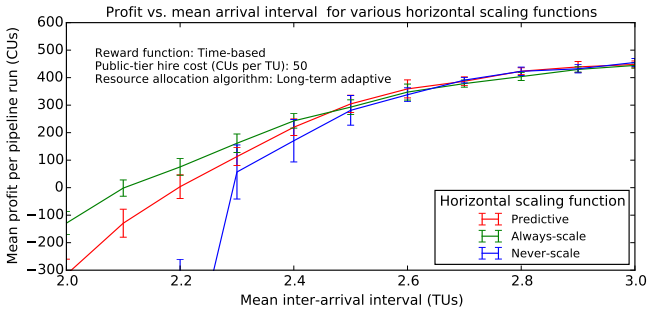

Fig. 5. Horizontal scaling algorithm vs. profit

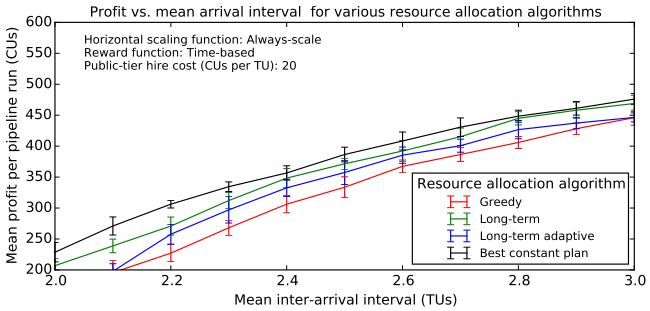

Fig. 6. Resource allocation algorithm vs. profit

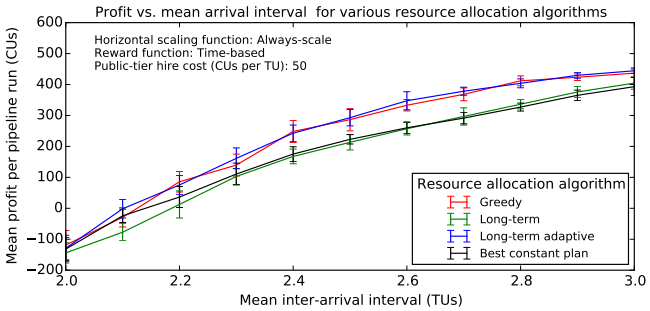

Fig. 7. Resource allocation algorithm vs. profit

predictive algorithm mimics the never-scale baseline with a light workload and the always-scale baseline with a heavy load. At intermediate loads it performs marginally better than either baseline, although it remains within a standard deviation of either.

At the heaviest workloads, when using predictive scaling, tasks often waited briefly in the queue for private resources, before the queue grew and dispatching a task to the public cloud became profitable. This leads to worse performance than the always-scale baseline because tasks are being dispatched to the public cloud almost as often, but sometimes queued briefly beforehand. A more refined algorithm might learn that its queueing decisions were often wrong and adjust its threshold for taking them, thus approaching the (in this case) optimal always-scale policy more closely. For light workloads the predictive algorithm's hesitation is almost always correct, and it performs like the never-scale baseline. Prediction briefly outperforms either baseline when a queued stage has a roughly 50/50 chance of ending up pushed into the public cloud or successfully waiting for private resources.

Altering the cost of public-tier resources simply changes where the workload becomes heavy enough that the predictive scaling scheme briefly becomes superior, followed by the always-scale scheme. Figure 4 shows the case when the public tier cost has been changed from 50 to 80 CUs per TU; other data points follow similar patterns. Altering the reward scheme similarly alters the break-even point between the different schemes, but not the fundamental pattern of behaviour.

Using a more flexible resource allocation policy leads to the three scaling algorithms behaving very similarly at light workloads. For example, Figure 5 shows the same situation as Figure 3 only using long-term adaptive resource allocation instead of the best constant baseline. The adaptive algorithm reduces the number of cores used per pipeline stage when running in the public cloud, reducing the impact of the always-scale baseline's eagerness to use expensive public resources.

C. Resource Allocation

When running with the always-scale baseline horizontal scaling algorithm, we found that our dynamic resource allocation policies do best when public tier cost is moderate. When public resources are cheap (Figure 6) the baseline constant resource allocation policy performs best because its chosen execution plan fits both tiers; when they are expensive the public tier is rarely used, and so a single plan that suits the private tier is acceptable. At intermediate levels our long-term adaptive and greedy resource allocation algorithms are able to beat the best constant plan at most workload levels, as shown in Figure 7. They perform better because the optimal plan involves a roughly even mix of highly-parallel and single-threaded executions, and so the penalty for trying to constantly adopt either one is significant.

Figure 8 shows the same situation as 7, only using the throughput-based reward scheme instead of the time-based scheme used so far. The long-term adaptive scheme outperforms the greedy scheme because the latter pessimistically assumes that the current stage is the *only* opportunity to optimise, whilst the former optimistically plans the rest of the pipeline; when using a reward scheme that favours *proportional* reductions to the whole pipeline runtime this makes the optimistic scheme more aggressive. The non-adaptive long-term allocation plan performs worst for both reward schemes because it plans the whole pipeline run at the beginning based on *current* circumstances, rigidly following its plan even if it switches from one cloud tier to the other between pipeline stages.

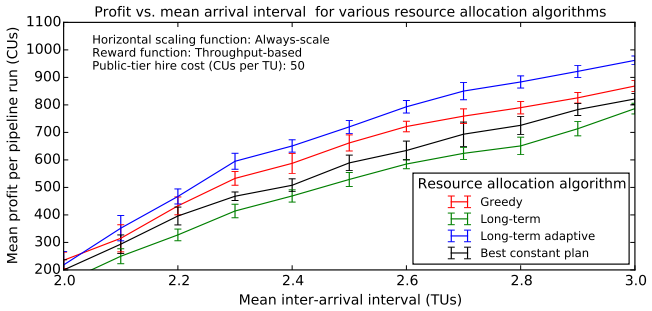

Fig. 8. Resource allocation algorithm vs. profit

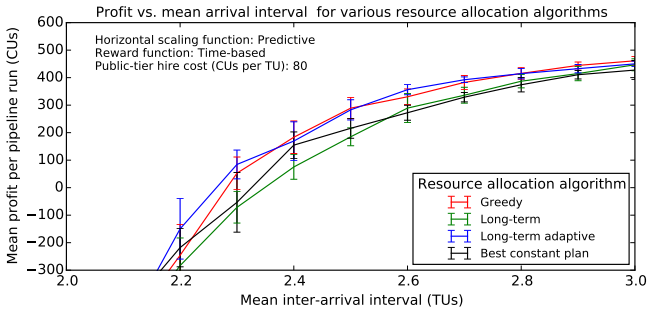

Fig. 9. Resource allocation algorithm vs. profit

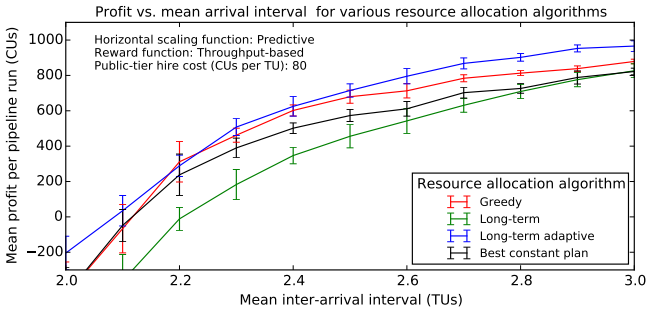

Fig. 10. Resource allocation algorithm vs. profit

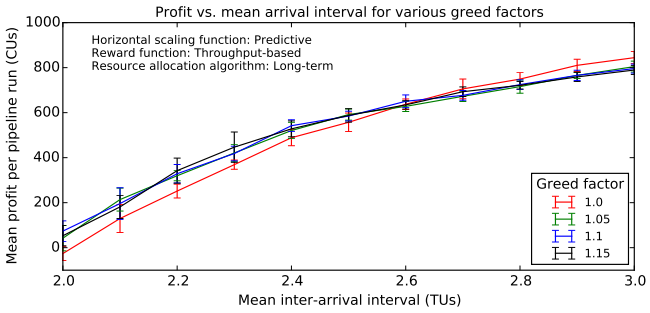

Fig. 11. Sharing factor vs. profit

Running our resource allocators with predictive horizontal scaling also enabled results in the greedy and long-term adaptive allocators consistently improving over the best constant plan. Figure 9 shows a case where both function usefully, with the time-based reward scheme and moderately high public resource costs. The benefits are clearest at moderate workloads where the optimal workload balance mixes single- and multi-threaded pipeline stage runs; at very low and high workloads the schemes converge as the range of viable choices shrinks. Switching to the throughput-oriented reward scheme but keeping all other attributes the same (Figure 10) differentiates the greedy and long-term adaptive algorithms, as the long-term adaptive algorithm's optimism about future pipeline stages again leads to more aggressive exploitation of opportunities to increase profit. Varying the cost of public cores yields similar behaviour when public tier cores cost 50, 80 or 110 CUs / TU; when the public cores are at their cheapest (20 CUs / TU) the best constant plan wins out because there is no longer an incentive to behave differently when running on private vs. public cores.

D. Sharing Factor

We expected that setting the sharing factor (defined in IV-B) would improve the performance of our resource allocation algorithms by encouraging collective optimisation when several jobs are competing for the last available private tier cores. However, in fact we found that only the long-term resource allocation policy benefited from setting the sharing factor above 1. This is most likely because the long-term allocation policy's inability to modify its execution plan when a particular pipeline run moves from one tier to the other between stages can lead to excessive cost as private-tier-optimised plans are run against the more expensive public tier; the sharing factor ameliorated that behaviour slightly by encouraging it to plan conservatively when the private tier is nearly fully occupied. Even then the benefit is only realised when the workload is heavy enough that resource usage frequently exceeds the capacity of the private tier. Figure 11 gives an example where the benefit resulting from setting the sharing factor is most clear, while Figure 12 shows a similar situation that is typical of the other resource allocation policies, where the default sharing factor of 1 produces the best performance at all workload levels.

All of these results were obtained with the public tier price per core set to 50 CUs per TU (i.e. 10 times the cost of a private tier core). It is possible that more costly public-tier cores, and so a greater incentive to fit tasks into the private tier whenever possible, might make a sharing factor above one profitable for other resource allocation policies; time did not permit us to investigate this possibility, but we intend to do so in future.

E. Prediction Accuracy

Our predictive horizontal scaling algorithm forecasts when running jobs will complete in order to choose between waiting for private tier cores and hiring public tier cores now. As described in section V, our simulator derives the actual time for a pipeline stage to complete from the prediction, multiplied by an *error factor* drawn from a distribution centred on one, but with a configurable standard deviation. In Figure 13 we plot a typical case when the error factor is varied from perfect (standard deviation = 0) to quite poor (standard deviation = 0.3). Profit falls significantly when the system is busy and the error factor is high across all reward schemes and resource allocation algorithms.

This occurs because the basic predictive horizontal scaling

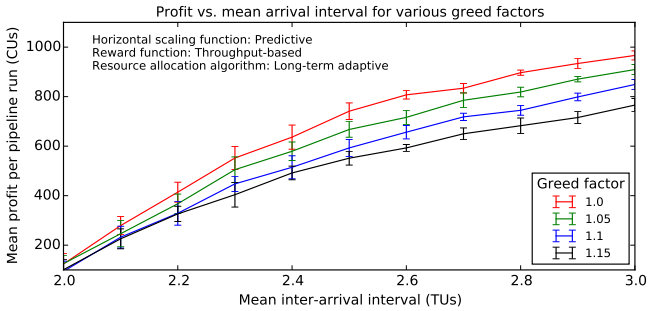

Fig. 12. Sharing factor vs. profit

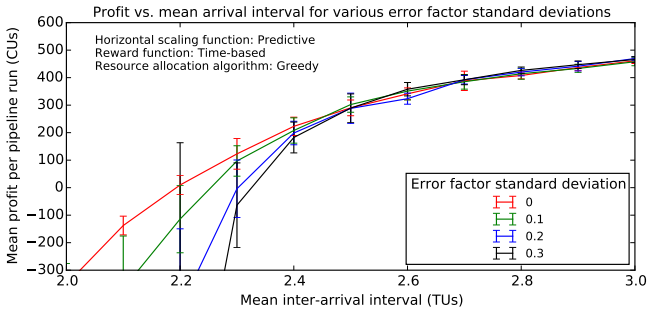

Fig. 13. Error factor vs. profit

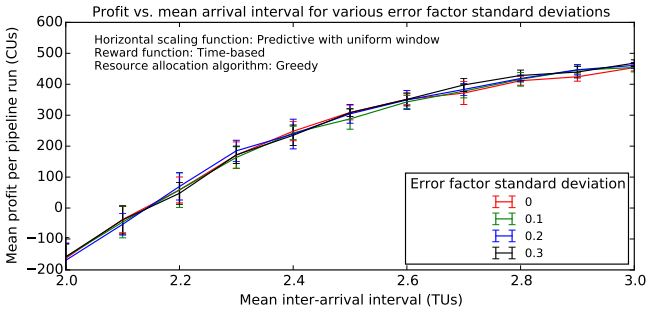

Fig. 14. Error factor vs. profit

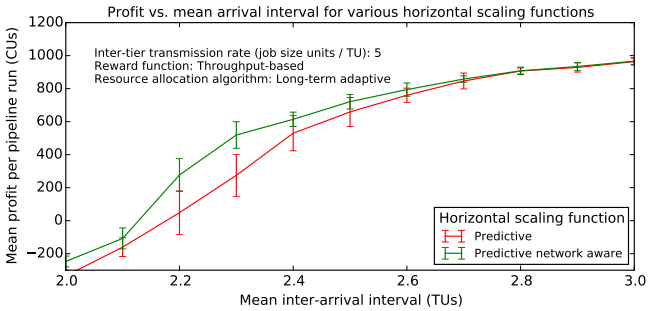

Fig. 15. Network bandwidth vs. profit

algorithm forecasts that late pipeline stages will complete *immediately*, which is not born out in practice. Therefore we made a simple augmentation to the basic algorithm to predict job completion according to a uniform probability distribution centred around the basic prediction, with a configurable width relative to their total runtime, called the *maximum error*. For example, setting the maximum error to 0.5 says that we should expect job completion in a window from 0.5 to 1.5 times the basic predicted completion time. Once some of the window has passed, this augmented algorithm predicts the expected completion time *given the knowledge that it has not yet completed*. Since the distribution is uniform, this is simply the midpoint between the current simulation time and the end of the window. The upshot is that when tasks are running late, we continue to predict their completion at least some time into the future. This encourages public tier core hiring if it is profitable to avoid the expected wait.

Figure 13 shows how profit varies with workload using an otherwise identical setup to Figure 14 and a maximum error of 0.9 – chosen because it is the maximum error that our simulation can in fact produce, and therefore the ideal case. The performance improvement is significant when the system is busy. The specifics of this improvement to the predictive scaling algorithm are clearly imperfect – for one thing, the distribution of finishing times for late tasks is not uniform, either in our simulator or in real life, and for another setting the maximum error parameter ought properly to be learned rather than manually supplied – but the point to be made is that some mechanism for sensibly predicting when late tasks will complete is necessary for the predictive horizontal scaling algorithm to function in the presence of inaccurate predictions.

These experiments were conducted at a public tier cost of 50 CUs per core-TU. Since the basic predictive scaling algorithm inappropriately discourages public tier hiring when running tasks are late to finish, if the experiment was repeated with varying public-tier cost we would expect the harm resulting from doing so to be greatest when public cores are cheapest.

F. Network Bandwidth

Previous experiments assumed a high-bandwidth inter-tier connection such that transfer time and costs are irrelevant. We extended our model to limit the inter-tier connection bandwidth, and introduced a *network aware* variant of our predictive horizontal scaling system that accounts for transmission delay when projecting a task's runtime. Pipeline stages queue to use the inter-tier link whenever a pipeline stage is to execute at a different tier than its predecessor, with the time taken to move their data determined by the size of the particular pipeline run's input data and the capacity of the link. The network-aware predictive algorithm accounts for the link by adding the expected time for the transmit queue to clear to the estimated execution time for a public tier stage.

We found that when the link's bandwidth is low (Figure 15 shows a representative example) then adding network awareness improves its performance by avoiding costly trips to the public tier; however when its bandwidth is high (Figure 16 shows a case that is otherwise identical to Figure 15) adding network awareness actually hurts performance when the workload is heavy. This is likely because, as seen in previous experiments, at this level of workload an always-scale policy is optimal, and network awareness is providing a slight extra incentive *not* to hire public resources.

We did not model a particular *cost* associated with network transmission; however, since data transmission time is proportional to data size which is itself nearly proportional to compute time, adding this would be very similar in effect to

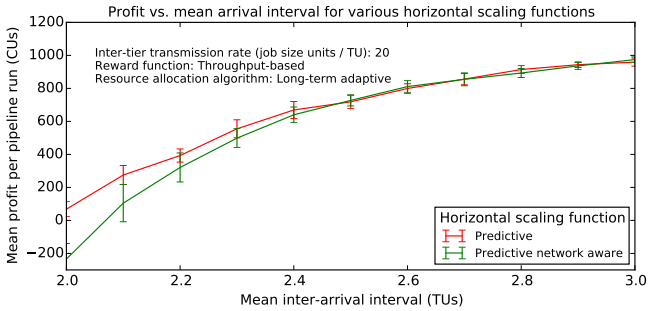

Fig. 16. Network bandwidth vs. profit

increasing the public tier's compute price. We also assumed a network link that can be shared optimally, so concurrent flows do not impact one another's throughput. While this is not true in practice when large numbers of independent machines fight for bandwidth, as the resource coordinator we are well positioned to use an efficient cooperative sharing solution if necessary.

G. Summary

We explored all permutations of resource allocation algorithm, horizontal scaling algorithm, reward scheme and workload, and found that our proposed algorithms are often able to improve performance above their respective baselines, demonstrating that real-world workloads using the GATK and similar analysis tools may benefit from intelligent cloud scaling and pipeline stage sizing.

We found that our long-term adaptive resource allocation algorithm outperforms the best-constant baseline algorithm in many circumstances, and that our predictive horizontal scaling represents a useful compromise between the two baseline schemes that always or never scale when private resources are fully occupied. We also found that altruistic or generous behaviour is not globally profitable in the situation we analysed, and that a scheme to forecast the likely completion time of late jobs is vital to the correct functioning of scaling algorithms that rely on predicting future job completion times.

VII. Conclusion

We have explored the design space of automatic algorithms to control cloud resource hiring and task resource allocation in a hybrid cloud situation, taking the Genome Analysis Toolkit as a particular case study for this exploration. Our simulation study found that our candidate algorithms were able to outperform simple baseline schemes in a wide variety of circumstances, and therefore argue that (a) classical HPC-style workloads like the GATK can benefit from this kind of automatic scaling and control, and (b) that our candidate algorithms represent a useful starting point to improve their performance in the real world.

In future work we intend to adapt existing scaling and scheduling algorithms that have been used to manage persistent services or distributed execution frameworks to the HPC use case and see how their behaviour compares to our candidate algorithms. We also plan to conduct large-scale tests in real cloud environments to validate the simplifying assumptions of our simulated cloud environment, or to determine what barriers to real world applicability exist.

Acknowledgement

This work was supported by the European Commission's CELAR (317790) FP7 project (FP7- ICT-2011-8). We thank the Scientific Computing team and RNA Biology Group at CRUK MI for their helpful comments.

References

[1] R. Chen, G. I. Mias, J. Li-Pook-Than, L. Jiang, H. Y. Lam, R. Chen, E. Miriami, K. J. Karczewski, M. Hariharan, F. E. Dewey *et al.*, "Personal omics profiling reveals dynamic molecular and medical phenotypes," *Cell*, vol. 148, no. 6, pp. 1293–1307, 2012.

[2] M. S. Lawrence, P. Stojanov, C. H. Mermel, J. T. Robinson, L. A. Garraway, T. R. Golub, M. Meyerson, S. B. Gabriel, E. S. Lander, and G. Getz, "Discovery and saturation analysis of cancer genes across 21 tumour types," *Nature*, vol. 505, pp. 495–501, 2014.

[3] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The cancer genome atlas pan-cancer analysis project," *Nature Genetics*, vol. 45, pp. 1113–1120, 2013.

[4] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 99–112.

[5] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 127–144.

[6] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data." *Genome research*, vol. 20, no. 9, pp. 1297–1303, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1101/gr.107524.110

[7] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 973–985, 2013.

[8] M. Malawski, K. Figiela, M. Bubak, E. Deelman, and J. Nabrzyski, "Scheduling multi-level deadline-constrained scientific workflows on clouds based on cost optimization," *Scientific Programming*, 2014.

[9] H. Kim, Y. El-Khamra, I. Rodero, S. Jha, and M. Parashar, "Autonomic management of application workflows on hybrid computing infrastructure," *Scientific Programming*, vol. 19, no. 2-3, pp. 75–89, 2011.

[10] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 644–651.

[11] F. Tian and K. Chen, "Towards optimal resource provisioning for running mapreduce programs in public clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 155–162.

[12] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[13] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 21–21.

[14] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[15] C. Smowton, A. Balla, D. Antoniades, C. Miller, G. Pallis, M. Dikaiakos, and W. Xing, "Analysing cancer genomics in the elastic cloud," in *Proceedings of the CCGrid Workshop on Clusters, Clouds and Grids for Life Sciences (to appear)*. IEEE/ACM, 2015.