Technical Report

TR-2012-016

Experiences with a computational fluid dynamics code on clouds, grids, and on-premise resources

by

Tiziano Passerini, Jaroslaw Slawinski, Umberto Villa, Alessandro Veneziani, Vaidy Sunderam

Mathematics and Computer Science

EMORY UNIVERSITY

Experiences with a computational fluid dynamics code on clouds, grids, and on-premise resources

Tiziano Passerini, Jaroslaw Slawinski, Umberto Villa, Alessandro Veneziani, Vaidy Sunderam

Mathematics & Computer Science, Emory University, Atlanta, GA 30322, USA

Abstract

Heterogeneity is often manifested in different forms even in environments that seem to be similar. In particular, clusters, grids, and infrastructure as a service (IaaS) clouds may appear straightforward to configure to be interchangeable. However, our experiences with mainstream parallel codes for solving complex computational fluid dynamics (CFD) problems demonstrate that secondary attributes – support software, interconnect type, availability, access, and cost – expose heterogeneous aspects that impact overall effectiveness of application execution as well as overall productivity. Our experiences provide preliminary insights into characterizing three different types of platforms to which users typically have access – local clusters, grids, and clouds – and show where the tradeoffs can be, in terms of deployment effort, actual and nominal costs, application performance, and availability (both in terms of resource size and time to gain access). The considered benchmark is the numerical solution of partial differential equations for a blood flow problem. For this application, we compare five platforms with respect to two metrics: time to completion and cost per simulation. We further introduce a third metric, the *utility function* of the simulation, and discuss different user profiles leading to different rankings of the tested platforms. Additionally, we propose a systematic approach to overcome heterogeneity introduced by software dependencies required for application execution. Our results suggest that IaaS clouds are a viable approach for intensive CFD simulations.

Email address: {tpasser, jslawin, uvilla, avenez2, vss}@emory.edu (Tiziano Passerini, Jaroslaw Slawinski, Umberto Villa, Alessandro Veneziani, Vaidy Sunderam)

Keywords: Platform heterogeneity, Cloud computing, Computational fluid dynamics, Cost characterization, Adaptation

1. Introduction

Computing as a utility has become a reality in many domains; clouds deliver storage and processing resources on demand via methods analogous to more traditional utilities. This is the case, for example, of Amazon's Elastic Compute Cloud (Amazon EC2), designed to provide access to fully configurable computing resources via virtualization. Such a paradigm is steadily evolving and many researchers would benefit if it were applicable to high performance computing (HPC) applications in science and engineering. Typically, applications in the HPC domain are characterized by computing and/or data intensive codes that are parallelized explicitly, commonly based on the message passing programming model. These applications are most commonly run on local, on-premise clusters or on platforms referred to as computational grids – although in practice, grid-computing predominantly manifests simply as remote access to clusters, just in a different administrative domain. In both settings, it has been traditional to measure the performance of HPC applications by a single metric, viz. time to completion for the particular application in question, parameterized along two dimensions: *problem size* and *number of processing elements* used. With the advent of cloud computing, two interesting perspectives have become relevant: (1) the viability of executing parallel applications on the cloud (either through self-assembly or renting a pre-built cluster); and (2) the actual dollar cost effectiveness of executing HPC applications on different target platforms. In this paper we report on experiences with executing a Finite Element Method (FEM) code on five different platforms that are heterogeneous in secondary respects [1] (e.g., interconnect, access method, software stack, use cost) and attempt to characterize the overall "expense factor" of each. In particular, the present work includes benchmarks of Amazon's EC2 instances, a computational offering that is a candidate to compete with traditional computing clusters. In the following sections, we provide some background information on normal modes of scientific application execution and subsequently outline the FEM

[1] Primary characteristics such as machine instruction set and programming model are assumed to be equivalent across platforms.

code used in our exercise. We then describe the process used and issues involved in preparing and deploying the application on five different platforms. Possible automatic strategies are considered for the deployment of the stack of software dependencies for the target application. Measurements of execution times and their pertinent usage costs [2] are presented and discussed; we also present a model for the subject-specific *utility function* of the computation. We discuss user profiles leading to different rankings of the tested platforms. The paper concludes with a summary of factors that characterize the effectiveness of using different kinds of platforms.

2. Background

HPC is intrinsic and integral to most fields of scientific endeavor. Message passing parallel programs are a staple modality of numerical simulations and computational analyses. In addition to the parallel framework, e.g. MPI, codes depend on various other auxiliary components: scientific and mathematical libraries, header files, particular compiler options and flags. These parameters (or subsets thereof) are quite specific to a particular *target platform*; executing the application on a different target platform may require a non-trivial amount of re-building effort (even if the actual application source code is untouched). Hence, applications often continue to be executed only on the default "home" platform, even if other viable and better options are present.

Grids and especially clouds present real opportunities for applications to move away from their home environments. If an application run can be obtained in minutes on the cloud instead of waiting for overnight turnaround times on a local cluster, clouds may be an attractive proposition – the perceived cost related to money and manpower effort is acceptable [31]. In the ADAPT project at Emory, we are investigating the feasibility and ease of deploying classes of applications on target platforms other than those on which they normally execute. As a benchmark test, we have experimented with a Finite Element Method (FEM) CFD code based on the C++ library LifeV [10] whose home environment is a 128-core cluster, and ported it on other computational platforms: clusters, grids and Amazon's EC2 cloud. Our

[2] By "usage cost" we refer to actual payments in the case of cloud environments and estimated cost in case of other environments. Estimates are based on normal parameters including capital cost, operation cost etc. amortized over normal life cycles.

preliminary experiences were reported in an extended abstract presented at the 21st International Heterogeneity in Computing Workshop (HCW 2012) [47]. Here we expand the study to include benchmarks of a CFD code for the solution of a blood fluid dynamics problem. Moreover, we introduce a novel discussion on the cost effectiveness of the different platform, based on a model of the *utility* of the computational task to the user.

3. Related work

The role of cloud computing as an extension of current HPC capabilities has been evaluated by many researchers. In various scientific fields, the rate of increase of available computing power is closely matched or outpaced by the increase in model complexity and therefore of the requirements for fast, large scale computations – prompting serious consideration of "unlimited, on-demand resources" that clouds promise. This however is still controversial [31, 41, 28, 33]. Cloud vendors have been reshaping their services, experimenting with new technologies, and exploring new price policies while users are assessing viability. Several cloud-effectiveness benchmarks have appeared in the literature ([41, 19, 35]). We believe, however, that an assessment of cloud computing as a viable choice in real-life applications requires evaluation of its support for more complex scientific software, as we detail in the next section. The present work also includes benchmarks of Amazon's `cc2.8xlarge` instances, a recent computational offering that is a candidate to match the performance of traditional computing clusters. Furthermore, most studies focus on time-to-completion; our study takes a broader perspective, including a preliminary assessment of cost aspects [25], and the set of activities required to prepare the execution environment for scientific codes on diverse platforms.

The use of the cloud as a platform for computational fluid dynamics (CFD) analysis has been explored by several software projects. Among the open source projects we cite CAELinux [4], a Linux distribution including a large set of open source packages for computer-aided engineering (Salome (Open CASCADE) [16], Code_Aster (EDF) [5], Code_Saturne (EDF) [6], OpenFOAM (SGI Corp) [12] and Elmer (CSC) [8]). CAELinux currently supports cloud execution on Amazon EC2 by providing a set of pre-defined virtual machines to be run on the EC2 service. OpenFOAM, an open source package for CFD analysis, can be also executed as a standalone package on the Amazon EC2 [18] computing service and on the SGI Cyclone Technical

Computing Cloud. We note here that our work is concerned with *comparing* effort, cost, and issues in executing applications on *multiple target platforms exhibiting secondary heterogeneity* rather than the aspect of porting applications to the cloud.

User-centric performance analysis has recently been applied to research on HPC and Grid scheduling strategies. The value that users associate with a completed job is modeled as a *utility function*, with a generally non-trivial dependence on time [39]. In other words, the importance of a job to a user can be seen as a function of time, combining an index for the importance of the results and the user sensitivity to delay. The design of proper utility functions has been object of study, and it has been shown that a proper job scheduling strategy can significantly increase the performance of HPC systems, measured as the aggregate utility of their users [23, 24]. We refer to these previous works to define a ranking of the tested architecture based on user-centric metrics.

4. The mathematical problem and its numerical solver

Partial differential equations (PDE) are a formidable tool for modeling problems in different fields, ranging from aerospace and automotive, mechanical and structural engineering to biology and biomedicine, ecology and finance [45]. Explicit and analytical solutions to PDE's of real interest are seldom available and numerical approximations are the only viable approach [30]. FEM is a well established approach to the numerical solution of PDE's [27, 22]. The FEM solution is a piecewise polynomial approximation of the exact one and the differential problem is replaced by a system (usually large) of algebraic linear equations. In matrix form, the system is *sparse*, most part of the entries of the matrix being zero; this feature dramatically reduces the memory footprint of the solver. The accuracy of the approximation is in general related to the size of each portion (*element*) of the computational domain where the solution is assumed to be polynomial. The finer the reticulation (*mesh*) defining the elements, the larger the algebraic problem to be solved after discretization – and consequently, the computational cost – but the more precise the solution.

Incompressible fluid dynamics represents one of the most challenging, attractive and impactive problems in modern scientific computing. Fast and reliable numerical solutions of the Navier-Stokes equations (NSE) – the basic mathematical model for incompressible fluid dynamics – are required in

several engineering fields, ranging from automotive/aerospace to geophysical and biomedical engineering [43, 26]. If $[u_1, u_2, u_3]$ denotes the velocity vector and p the pressure of a liquid in the 3-D space with coordinates x_1, x_2, x_3, the incompressible NSE read

$$\begin{cases} \rho \dfrac{\partial u_i}{\partial t} - \displaystyle\sum_{j=1}^{3} \left(\frac{\partial}{\partial x_j} \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \rho u_j \frac{\partial u_i}{\partial x_j} \right) + \frac{\partial p}{\partial x_i} = f_i \quad i = 1, 2, 3 \\ \displaystyle\sum_{j=1}^{3} \frac{\partial u_j}{\partial x_j} = 0. \end{cases} \tag{1}$$

Here ρ is the fluid density and μ is the viscosity (that we assume to be constant for simplicity). Vector $[f_1, f_2, f_3]$ is an external forcing term. From the numerical viewpoint, this problem is very challenging, not only due to size (this is a vector problem involving four scalar fields), but to intrinsic mathematical features and the non-linear term (see, e.g. [26]).

4.1. Test case: blood flow in a cerebral aneurysm

Computational models based on the NSE have become a valuable tool for the study of blood flow problems, due to their cost-effectiveness and flexibility with respect to *in vitro* experimental studies. They allow the analysis of blood flow dynamics in subject-specific vascular geometries, and the controlled and reproducible assessment of the effect of experimental parameters such as heart rate, average flow rate, and flow conditions in the neighboring parts of the circulatory system.

The *image-based CFD pipeline* for blood flow problems starts with the image of a part of the circulatory system (typically including one or more arteries), acquired with techniques such as magnetic resonance (MR) or 3-D rotational angiography. A geometric model of the vascular structures is extracted from the image by means of segmentation techniques (see e. g. [42]). Such methods aim at characterizing the shape of the vessels, by identifying the contour separating blood from other biological structures in the image. This yields the definition of a surface that represents the domain of interest in which blood flow is simulated. To this end, a three-dimensional mesh is built. The Navier-Stokes equations are solved at some selected instants within the time interval of interest, assuming that blood velocity or blood pressure are known on the boundary of the volume of interest. In particular, subject-specific measurements may be used to quantify the unknowns on the

inlet and *outlet* sections of the vascular geometry, while blood velocity is assumed to be zero in contact with the vascular wall (*no-slip* condition). This information is used to prescribe boundary conditions, required for the solution of the differential problem. The time discretization is performed by a suitable approximation of time derivatives, using the value of the solution in the collocation instants. Among others possibilities, we use here *Backward Difference Formulas (BDF)* with an error proportional to the square of the distance between two consecutive instants (second order methods) [29, 44].

In this paper we use for our experiments a problem proposed in the Inaugural CFD Challenge Workshop at the ASME 2012 Summer Bioengineering Conference, to benchmark the sensitivity of CFD pressure predictions and flow patterns to some particular aspects of the image-based CFD pipeline. The benchmark involves the simulation of blood flow in a giant aneurysm grown in the internal carotid artery. All the physical features are assigned ($\mu = 0.04$ Poise and $\rho = 1$ g/cm^3). We consider a single scenario among the ones proposed in the original benchmark, that is the simulation of a pulsatile flow with a mean flow rate of 5.13 mL/sec.

The physical phenomenon that we simulate is the movement of blood in the internal carotid artery and in the aneurysm, during a single heart beat. We assume that at the beginning of the simulation ($t = 0s$) the system is at rest, with the blood velocity and the pressure both being zero. In the first part of the simulation (from $t = 0s$ to $t = 0.05s$) we linearly increase the prescribed blood velocity at the inlet section until we reach the physiologic value corresponding to the beginning of the heart beat. We then prescribe a time-varying flow waveform with a period of $1s$, end we stop the simulation at $t = 1.05s$. The prescribed blood velocity at the inlet section in the last simulated frame coincides with the velocity prescribed at $t = 0.05s$.

We solve the equations at 100 instants within the cardiac cycle (i. e. the simulation time step is $0.01s$). A snapshot of the computed solution is shown in Figure 1.

4.2. The organization of the program

The numerical solution of problems like the proposed test case involves operations that are conceptually split into two categories. Some operations are independent of the time advancing and are performed out of the temporal loop. Other operations need to be performed at each time step. These typically constitute the computationally-intensive kernel of the software. Schematically, we represent the stages of the application in Figure 2.

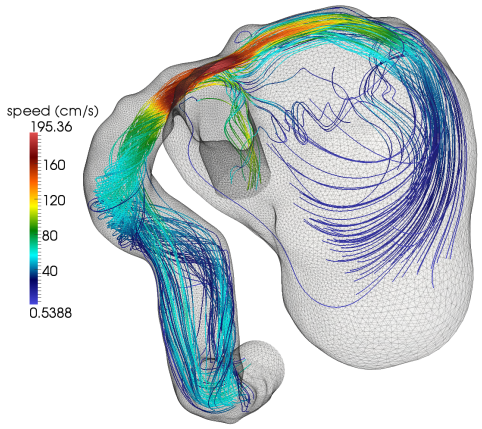

Figure 1: Solution of the problem, based on equation (1), when $t = 0.28$s. Streamlines of the velocity field, when the flow rate is maximum over the cardiac cycle.

Here we detail each phase. Step (i) consists of the definition of the *computational domain* – given by the *mesh* – where the equations have to be solved numerically. Mesh generation is typically accomplished with in-house software (for structured meshes) or third-party software such as NetGen [37] and GMSH [32]. In the parallel application, the domain is *partitioned* so that each process takes care of only a subset of the global mesh. This splitting is achieved through the use of graph partitioning algorithms, such as those implemented in the library ParMETIS [38], guaranteeing a proper load balancing among processes. The load is measured as the number of mesh elements assigned to each process. At the same time, high quality partitionings should minimize the edge-cut, or the number of connections between disjoint partitions. This property is valuable because in the considered application the communication between processes involves only the exchange of data related to neighboring elements. Other operations of this step refer to all the computations that are time independent and can be performed once.

Step (ii) concerns the computation (or more specifically the *assembly*) of the matrices and vectors required for the construction of the discretized algebraic problem. Each process has local access to a subset of the matrices and vectors corresponding to its own portion of the mesh and requires limited access to adjacent submeshes. Assembly is carried out with algorithms provided by LifeV, while the data structures for the management of the distributed matrices and vectors are provided by Trilinos [46].

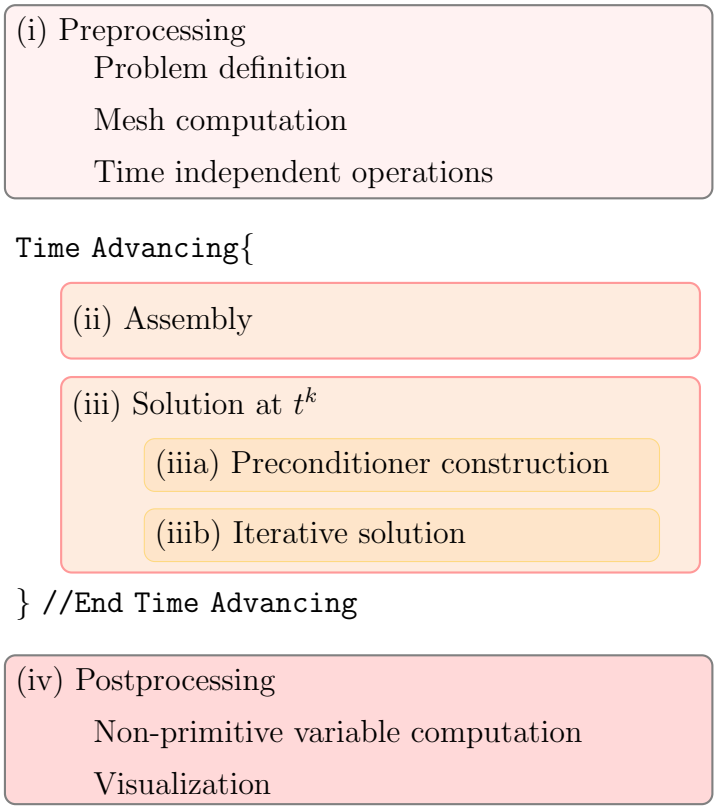

Figure 2: Steps for the numerical solution of a time-dependent PDE problem.

Trilinos also provides algorithms for the solution of the algebraic problem (Step (iii)). In particular, we use *iterative preconditioned methods*, where the solution of the large linear systems assembled at each time step is replaced by the iterative solution of simpler systems (called *preconditioners*). For this reason, we distinguish Step (iiia) for the computation of the preconditioner, and Step (iiib) for the actual solution of the preconditioned system.

Step (iv) concerns the visualization of the solution to the differential problem, and can be delegated to third-party software such as Paraview [13]. This step may also include the computation of quantities of interest related to the solution u.

For the purpose of the present paper, we are mostly concerned with Steps (ii) and (iii), which have a major impact on the entire computational cost of the application.

4.3. Summary of the packages used in LifeV

The complete list of required packages to build our CFD solver follows:

- LifeV library [10], for the formulation of the algebraic counterparts to differential problems; this library is the direct dependency for our solver application;
- Third-party scientific libraries: (1) Trilinos [46] for the solution of linear systems (data structures and algorithms); (2) ParMETIS [38], used for mesh partitioning; (3) SuiteSparse [17], as a support library extending the capabilities of Trilinos; (4) BLAS/LAPACK libraries (generic or vendor-specific implementations);
- General-purpose and communication libraries: (1) Boost C++ libraries [2] 1.44 or above, mainly used for memory management (smart pointers); (2) HDF5 [51], for the storage of large data on file; (3) MPI libraries (e.g., Open MPI);
- Compilers: C++ compiler (e.g., GCC version 4 or above); [optional] Fortran compiler, compatible with C++;
- Deployment tools: (1) GNU make; (2) Autotools; (3) CMake (version 2.8 or above).

5. Heterogeneous Target Platforms

In our study, we compared five heterogeneous computational platforms supporting the parallel hemodynamics simulation. As the starting point for our analyses, we selected the in-house computing cluster `puma`[3] constituting a computational test bed for the LifeV developer team. As the second platform, we used a larger compute cluster `ellipse` provided on a fee-for-use basis within our university. The third platform was the HPC supercomputer `lonestar` made available to the U. S. research community by Texas Advanced Computing Center. Next, we evaluated the usability of on-demand resources. The first such platform was `rockhopper` [14] offered as a part of the Penguin's On-Demand HPC Cloud Service [9] and the second platform was the IaaS cloud provided by Amazon's Elastic Compute Cloud (EC2)

[3]This is the "home" environment where the application is run by default

	puma	ellipse	lonestar	rockhopper	ec2
cpu	Opteron 2214	Opteron 280	Xeon 5680	Opteron 6174	Xeon E5
cores	2x2	2x2	2x6	4x12	4x4
RAM	8GB	4GB	24GB	2.5GB/slot	66GB
network	SDR IB	1GbE	QDR IB	QDR IB	10GbE
storage	NFS	NFS	Lustre	Lustre	local fs
support	full	very limited	limited	online	none
OS	Rocks 5.1	CentOS 4.8	CentOS 5.5	CentOS 5.6	AMI 12.03
access	user space	user space	user space	user space	privileged
bld.env.	yes	yes	yes	yes	none
comp.*	GCC 4.3.4	GCC 4.1.2	ICC 11.1	GCC 4.4.4	none
sci.libs	all	none	MKL	MPI	none
MPI	Open MPI	none	MVAPICH2	Open MPI	none
exec.*	PBS	SGE	SGE	SGE	shell

Table 1: Specification of a single node of the test architectures.
* If a system offers more options, only the used one is reported.

service. From the rich EC2 resource offerings, we picked the most powerful instances `cc2.8xlarge` from *Cluster Compute* (referred to as `ec2` in the following).

The five platforms are heterogeneous in many respects: they differ in hardware configuration, availability (measured as wait-time before execution), access modality (privileged vs. unprivileged user), storage (e.g., size of user disk space and presence of a shared file system), build (e.g., the compilers and system tools availability), computational aggregation (e.g., presence of configured MPI environment), and execution (e.g., interactive shell). In this section we compare the considered architectures, pointing out differences and similarities. Table 1 collects all features of the chosen targets: hardware, system software, user access privileges, build environment, presence of the required scientific libraries, and simulation launch method; below we note a few relevant details.

`puma`. This in-house computing cluster comprises 32 four-core nodes. Users have unprivileged access to the machine, so they need to install any missing required software (libraries, etc.) in user space. However, as the machine is maintained by local department personnel, users are often able to request system-wide installation of generic dependencies (e.g., BLAS, LAPACK or

MPI). As the "home" environment for LifeV developers, this cluster was pre-provisioned with the entire set of packages required to run LifeV-based CFD simulations. Being an internal resource with restricted user access, `puma` does not implement a monetary accounting system for computing resource usage. However, for comparison with other architectures, we estimated the cost of our department cluster, based on its initial price and operating expenses, at 2.3¢ per core-hour, which is consistent with other published estimates [49].

`ellipse`. This university cluster consists of 256 four-core nodes and is well-configured for sequential execution of a specific, preinstalled class of applications such as scripts in Matlab or R [11, 15]. As a result, we conditioned in user space the full stack of software necessary for executing the hemodynamics simulation, including Open MPI. The cluster configuration did not allow execution of parallel Open MPI jobs exceeding 128 processes, as the MPI frontend node designated by the SGE system was unable to start a higher number of remote MPI processes. All tenants of `ellipse` pay a flat rate of 3¢ per CPU core per hour.

`lonestar`. This Linux cluster consists of 1,888 12-core nodes. Users have unprivileged access to a small backed up home filesystem, a larger non-backed up disk subsystem and additional scratch space. Software packages, compilers, and libraries are provided to users by *environment modules*, that allow conditioning the user space environment for provisioning the needed software. We used the default set of compilers and MPI libraries, and selected the Intel vendor-specific implementation of BLAS and LAPACK (available through the `mkl/10.3` module). The available queues accept jobs with a maximum wall time of 24 hours on at most 4104 cores, a constraint that did not impose restrictions on our benchmark. Access to this facility is granted to off-site users upon request of allocation to the National Science Foundation XSEDE project[4]. User accounts receive a certain amount of Service Units (SU), corresponding to core-hours. The equivalent value in dollars of an SU on `lonestar`, based on an estimate of the acquisition cost and the project cost (personnel, power, cooling, etc.) is 7¢ per CPU core per hour.

`rockhopper`. Despite this eleven node cluster being offered as an on-demand HPC resource, it does not differ from typical HPC machines and requires

[4]Extreme Science and Engineering Discovery Environment, `https://www.xsede.org`

classic procedures in order to build and execute the user applications. The only recognized limitation regarded the size of the parallel job which limited our tests to a maximum of 256 cores/processes. The system offers several build environments, preexisting scientific applications as well as Open MPI. To use the present dependency (the compiler and MPI environment), we loaded the `openmpi/gcc44/1.5.5` environment module. The setting of the system was described in detail on the companion, well-maintained web page which fully supplied answers for our issues regarding conditioning. The vendor charges a fee for using its cluster – we paid 10¢ per core-hour including an academic discount, but excluding marginal expenses for storage use.

`ec2`. Our final target architecture was a typical infrastructure as a service (IaaS) cloud offered by Amazon. IaaS resources provide on-demand computing in the form of computing chunks virtualized from the vendor's multi-tenant machines. These chunks are delivered for users as standard ssh-able, root-accessible computational hosts. Users requesting these chunks specify the number of hosts, a resource class (characterized by computational power, number of CPU cores, memory capacity, and network interconnect) and the Operating System (OS) controlling the hosts (*public* or users' *private* OS images). The vendor offers several sizes of virtualized hosts, ranging from small instances to HPC-class cluster nodes equipped with GPGPU processors and with network-aware host allocation strategy – *placement group*. All setup conditions, as well as management and monitoring measures can be controlled by users in various ways, including direct interactions with the AWS (Amazon Web Services) Management Console web toolkit or Amazon EC2 API command-line tools [1], programming third-party libraries [3], or frameworks providing higher level services over IaaS clouds [40, 48].

For our experiment we exclusively used `cc2.8xlarge` instances from the `us-east-1d` Amazon datacenter zone. In order to provide the execution platform for our application, we launched a certain number of these instances in a single placement group and configured them to achieve a parallel execution environment. In contrast to conventional computational resources, EC2 users obtain full (root) access to hosts instantiated on Amazon's service. As a result, we could use a package management system (`yum`, in our case) and modify the system configuration. Moreover, to facilitate the entire soft-conditioning operation and eliminate duplications in this process, we used the public `Amazon AMI 2012.03` image to establish a private image that was fully configured and had the complete simulation data. Next, we used this

image to instantiate all requested nodes.

Amazon does not levy any upfront costs and charges users merely for the actual use of resources (time and computational power), external data transfers, and scratch space (size); however, some OS images and additional services (e.g., static IPs) incur additional costs. Furthermore, Amazon has two price polices for its EC2 resources: (1) the official, flat-rate price for on-demand instances that are delivered to the user immediately after the request; and (2) the *spot request* scheme whose pricing is based on the nodes of multitenant machines that are not assigned to users on a regular basis, but rather offered for bid prices that can be much lower. The disadvantages of using a spot request policy are (1) the unpredictable nature of the actual price of the resource at execution time (Amazon updates the spot price every hour); (2) temporal spikes in the actual price (fluctuations can be of the order of multiples of the regular price); and (3) Amazon reserves the right to reclaim the instances at runtime, in case the actual spot price becomes higher than the maximum bid price offered by the user. During the benchmarks, the regular price of a single 16-core `cc2.8xlarge` instance was $2.4 per hour while the average spot request price calculated for all simulation runs we launched was 36.35¢ per hour (or about 2.27¢ per core-hour).

6. Metrics

The chosen test case is representative of a routine task in computational studies of the human cardiovascular system, that is a simulated experiment for investigating the features of blood flow in a diseased artery. We aim to compare different hardware platforms with respect to the execution of the same software application, evaluating several different metrics.

Computational experiments are particularly valuable as compared to more traditional *in vitro* studies when considering their flexibility and cost-effectiveness in simulating several different scenarios. In the study of hemodynamics, computational experiments can be used to perform parametric studies for assessing the sensitivity of a measure of interest to changes of a chosen parameter (e.g., the heart rate, or the shape of an artery). For this reason, computational simulations are being increasingly employed as tools of choice in the design of medical devices [50] and surgical planning [34, 52]. Crucial to their penetration in the medical routine as tools for the diagnosis, prognosis, and treatment planning is their usability, in terms of how much time is required for a single computational experiment and how much it costs. It is worth

noting that minimizing these two aspects may lead to conflicting strategies, as it is often the case that the most expensive hardware resource provides the result in the shortest time, as we will see later on.

A user-specific combination of the two metrics "time to completion" and "cost per simulation", that we are going to introduce in detail, yields a third metric, corresponding to the "perceived cost" of the computational experiment.

Time to completion. This is the time necessary to complete the execution of the application on the particular platform. We expect the following factors to have the most significant effect on this metric:

- performance of the computing unit (CPU clock rate, load balancing among CPU cores, memory-CPU affinity, and cache size);
- quality of the network interconnect aggregating the computing nodes (bandwidth, latency, network topology, and congestion);
- software optimization to exploit the specific hardware architecture (e.g. vendor-specific implementations of the BLAS/LAPACK libraries).

This time metric does not include the queue waiting time, that we considered negligible. In fact, in the experiments presented in this paper, the queue systems were responsible for short delays, compared to the execution time of the application. However, this assumption may be inaccurate depending on the turnaround times on the considered batch systems. Some systems might offer several queues, including queues with higher priorities (i. e. less wait time) for a higher price. This would couple the time and cost metrics, and was not included in our analysis for the sake of simplicity. IaaS systems, instead, are characterized by the predictable, zero waiting time which could be an advantage when defining a ranking based on the time metric alone.

Cost per simulation. The overall cost for the execution of the job depends mainly on the unit cost of the hardware resource (cost per core-hour), its pricing policy (by core or by node, by hour or prorated), and on the execution wall-clock time. Other factors, that we consider negligible relative to the former (for our application), are the size of occupied storage and/or volume of data staged in and out. As detailed in section 5, we report the cost of the different resources used in the study at

- `puma`: 2.3¢ per core-hour;

- `ellipse`: 3¢ per core-hour;

- `lonestar`: 7¢ per core-hour.

- `rockhopper`: 3¢ per core-hour;

- `amazon`: 36.35¢ per 16-core EC2 instance per hour. Since Amazon charges the users for the entire machine, cost effectiveness decreases if not all cores are utilized;

Utility function. The *utility function* expresses the job's value to a user, as a function of time. This has a user-specific, complex dependency on several parameters, including expenditures, time to completion, and significance of the task. Following [24, 36], we consider a simple linear utility function with customizable maximum (starting) value and slope, as shown in figure 3. Its equation is

$$U(t) = \begin{cases} U_{max} & \text{if } t \leq T^* \\ U_{max}\left(\dfrac{T_0 - t}{T_0 - T^*}\right) & \text{if } T^* < t \leq T_0 \\ 0 & \text{if } t > T_0 \end{cases}$$

U_{max} is a measure of the *importance* of the job to the user, and we assume that we can give it a monetary value, as the price that the user would be willing to pay for the simulation. T^* is the expected completion time, which can be estimated in several ways; we use a simple averaging method defined in section 7.5, based on the performance of the available platforms. T_0 is the user-defined time at which the utility is zero, while the distance $(T_0 - T^*)$ is a measure of the user's *delay tolerance* and can be measured as a multiple of the expected completion time T^*.

With this formulation, we assume that there is no loss of value during the expected duration of the job (when $t \leq T^*$). An extension of the model could take into account the decrease in the utility function during runtime, reflecting the fact that faster runtime is valuable to users [39].

7. Experimental Results

Our experiments on different architectures are compared first in terms of the steps required for porting the software; then by evaluating the proposed

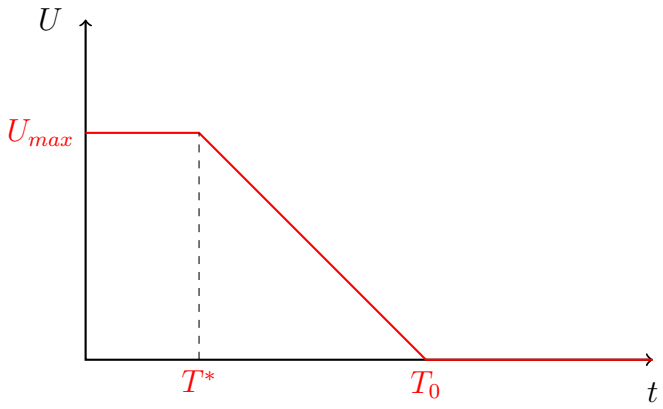

Figure 3: The considered utility function. U_{max} is a measure of the *importance* of the job to the user, T^* is the expected completion time, T_0 is the time at which the utility is zero [24].

metrics. Overall, we assess the usability of each architecture for our CFD simulations.

7.1. Porting experience

Execution of our simulation application on the target architectures requires (1) providing all software dependencies, (2) running the actual build program (make) that links against the appropriate libraries and produces the final executable file, and (3) providing the parallel execution environment. We minimized the effort related to adapting the software stack to the target architecture. Consequently, we did not make any change to the application source codes, utilized all compatible software that was already available on the target (even if not the latest version), and resorted to installation (preferably from package repositories) only if the dependency was missing or incompatible. In the `ec2` case, we had to commit an extra, minimal configuration allowing aggregation of computational hosts for a single parallel execution.

7.2. Soft-provisioning

Methods that we used to elevate the resource capabilities to the LifeV build and execution environment range from no-action to operations reserved for privileged users, and from building from source codes to providing a required package through a binary installation. We give here a short account of our experiences; more details can be found in [47].

`puma`. This system fully supports the build and execution of LifeV-based applications. As a result, we only needed to use a generic Makefile to create the executable. To launch the simulations, we used the Portable Batch System (PBS) job submission command.

`ellipse`. This cluster already provided the GNU compiler collection as well as all standard deployment toolkits; the remaining requirements were fulfilled using source code builds. For the BLAS/LAPACK package we resorted to a CPU vendor-specific implementation, available as ACML [20].

The Sun Grid Engine (SGE) on `ellipse` was not configured to support parallel tasks; however, Open MPI could detect and liaise with SGE to start and monitor processes on assigned job nodes. Thus we were able to use SGE commands to reserve computing nodes and submit `mpiexec` jobs.

`lonestar`. All of the low-level required software packages were available as modules on `lonestar` (Intel compilers and MVAPICH2, Boost libraries, CMake, CPU vendor-specific implementation of BLAS and LAPACK). The remaining parts of the software stack were provisioned via source installation. Parallel execution was managed via the SGE system. `lonestar` reserves entire nodes regardless of the number of processes requested for a job (i.e., the system allocates nodes in 12-core groups).

`rockhopper`. This target platform offered several variants of build and MPI environments loadable by prepared modules (we selected GCC 4.4.4 and Open MPI). We provisioned the remaining dependencies through user space installation as in the case of `ellipse`. To execute the parallel simulation we used SGE, although this system also provides PBS. The execution configuration limited jobs to a maximum of 128 processes.

`ec2`. To exercise the port of our software to EC2, we selected the `cc2.8xlarge` instance and the *Cluster Compute Amazon Linux AMI 2012.03* (`ami-e965ba80`) image. To facilitate software preconditioning steps we exploited `root` access. The chosen image contains only the essential packages, with neither development software nor scientific library support. In order to provide the source code build environments, we installed, using `yum`, GCC libtool (with autoconf, automake), and Open MPI (with extra configuration of the environment). To install CMake, we resorted to a source code installation as the required version was not available from the repositories. After this phase, we downloaded all required scientific dependencies in source form, built, and

installed them. After these preconditioning steps, building the simulation application was straightforward.

We also encountered cloud-specific issues not seen on traditional grids. One concerned ssh host mutual authentication to enable password-less launch of remote processes by `mpiexec`, requiring pre-generation and storage of keys. The second issue was related to configuration of the EC2 service. We modified the *security group* by enabling all intranet TCP ports to allow MPI process intercommunication. Another issue was related to the lack of a shared file system commonly provided by clusters. However, we decided not to configure the NFS service and, consequently, we supplied the input files from the local volume. All the changes committed on the running instance can be preserved by creating a private image stored on the Amazon service. This image, in turn, may be used to launch several identical copies of the instance. Such on-demand hosts behave like cluster nodes. Further conditioning may provide a high-availability computing cluster with services such as monitoring or automatic checkpointing. Nonetheless, we prepared an image that contains only the essential software packages and services that allow this on-demand resource to support our CFD simulations.

In order to execute a simulation, we instantiated an appropriate number of copies of the prepared image. The service assigned intranet IP addresses for the on-demand hosts and we used these IPs to create the run-specific hosts list for the `mpiexec` command. Finally, this command was invoked directly from the command line on a selected host from the list.

7.2.1. Adaptation

A complementary aim of this experiment was to gain experience required in our companion project ADAPT. This project seeks to develop methods to enhance the portability of science and engineering software applications on multiple target platforms. Portability is achieved through dynamic and adaptive environment conditioning prior to application deployment. The key goal for such conditioning is to satisfy the application *requirements* using the actual *capabilities* of a particular target platform.

The key premise of ADAPT is that several classes of unmodified applications can be executed on multiple types of computational back-ends with the assistance of a flexible and adaptive middleware environment. We realize this model using *adapters* which elevate the software capabilities on the platform to those required by the application in a step-by-step manner. The simplest action of an adapter is soft-conditioning of missing dependencies, as needed

in our test cases. In this sense, our study allows gathering knowledge needed to implement soft-conditioning adapters.

In table 2 we present a set of adapter features that would automate porting for our five settings. First, the adapter performs a (autotools-like) *probe* to check availability of the software package needed. This check may test the existence of include files, libraries, and executables; in addition, the adapter may validate the version of the software package in question. Next, the adapter needs to locate software *prerequisities*. This step may also be performed using probe-like mechanisms. Moreover, it may initiate other soft-conditioning to be performed by other adapters. After these preliminary steps, a conditioning *action* is invoked – this may require user space operations such as staging, compilation from source codes, and configuration or, in some situations, installation from repositories. This action may also include modification of `Makefile` templates and other build-related files. Such changes may relate to specific filesystem paths and compilation options. Another set of adapters might check for available environment modules to detect and load the required software. Finally when the adapters complete, the target in question manifests the newly conditioned *capability*.

Note that adapters do not have to be implemented as monolithic solutions. In our approach, we envision them to be flexible and present in different variants appropriate to the circumstance (e.g., architecture, current capabilities, privilege level of the user). Additionally, to support new architectures, extension schemes are needed to create adapters that soft-condition targets by building upon existing adapters. This composability of adapters is intended to enable the full set of required capabilities to be deployed on a given target [21] [48]. In the exercise described in this paper, the adaptation process was performed largely manually, but several scripts were developed in the process that we expect will form the foundation for partially automated adapter deployment.

7.3. Performance analysis

In our study we tested the selected platforms using the strong scaling benchmark principle – we ran the parallel simulation solving a fixed-size problem using an increasing number of processes. The number of unknowns in the linear system to be solved at each time step of the simulation was 3,162,146. We performed a series of simulations on each platform, each time doubling the number of processors and starting from the minimal number of processes which could support the computation on a given target. This lower

adapter	probe	prerequisities	action	capability
The application to execute and its direct dependency				
SimApp	–	LifeV, HDF5, Boost, ParMETIS, Trilinos	git simApp edit Makefile; make	simApp.o
LifeV	–	Trilinos, ParMETIS, HDF5, Boost	git LifeV cmake; make install	lifev*.h lifev*.a
The scientific libraries				
Trilinos	Lib Inc Ver	MPI, SuiteSparse, ParMETIS, Boost, HDF5, BLAS	wget cmake make install	trilinos*.h trilinos*.a
ParMETIS	Lib Inc Ver	MPI	wget; make	[par]metis*.h [par]metis*.a
Suite-Sparse	Lib Inc Ver	MPI, BLAS, ATLAS	wget; edit Makefile make install	sparse*.h sparse*.a
BLAS & LAPACK	Lib	–	wget ACML bin ./install.sh --- wget ATLAS ./configure make install	blas.a lapack.a
General purpose and communication libraries				
Boost	Lib Inc Ver	–	wget; ./bootstrap ./b2 install	boost*.a boost*.h
HDF5	Lib Inc Ver	MPI	wget; ./configure make install	hdf5*.a hdf5*.h
MPI	module avail --- Lib Inc	–	module load --- wget; ./configure make install --- rep install openmpi	mpiexec mpi.so mpi.h
Build tools				
compiler	Bin Ver	–	rep install c++	c++
Auto- & systools	Bin Ver	–	rep install tools	tools
CMake $\geq$ 2.8	Bin Ver	–	wget; ./configure make install	cmake ccmake

Table 2: The soft-conditioning adapters

limit is governed by memory availability – the whole problem to be solved has to fit in memory in order to perform the computation.

All tested clusters allowed the reservation of computing resources by specifying through the queue system the number of processes (or slots) used by the parallel job. However, in the case of the Amazon EC2 cloud, we needed to set the execution policy: we assumed that each `ec2` instance can host a maximum of 16 processes (as they have 16 physical cores) and we decided to map the MPI processes onto the physical nodes in round-robin fashion. As Amazon charges users on the basis of running instances, we decided to optimize the cost of the benchmark by testing small assemblies of `ec2` first, and then to increase the number of nodes in the assembly by powers of 2. For this reason, we present several configurations of cloud instances; we label such separate assemblies as `ec2-`i, where i is the number of `ec2` nodes.

As shown in Figure 2, the application repeats the same set of operations in each simulated time frame (in our case corresponding to 0.01s intervals). For each considered hardware platform, the time required to compute a single frame was practically constant during the course of the simulation. We, therefore, use the average computing time for a single frame as a proxy for the performance of the hardware resource. This facilitates a side-by-side comparison of all platforms, including cases in which the simulation could not be completed due to cluster usage policies (e.g., `ellipse` limits the job execution time to 12 hours so for jobs that spanned small numbers of nodes only a fraction of the entire simulation could be done).

The graph in Figure 4 shows a comparison of the performances of the different platforms, as a function of the number of computing cores. On-premise resources (`puma`, `ellipse`), the HPC cluster (`lonestar`) and `rockhopper` achieve good strong scaling up to 128 computing cores, while they show a significant decrease in performance for larger numbers of cores. In particular, Point **A** in the figure corresponds to the fastest execution case in our experiment, that is running the simulation with 128 computing cores on `lonestar`. On this platform speed up increases by a remarkable factor of 11.2 when passing from 8 computing cores to 128 computing cores (corresponding in the figure to points **B** and **A** respectively)

`ec2` resources scale less well. `ec2-1` achieves good scaling only in the range 4-8 cores, `ec2-2` up to 16 cores, `ec2-4` up to 32, `ec2-8` up to 16 cores, while `ec2-16` does not achieve strong scaling in any range. Point **C** in Figure 4 corresponds to the case in which the simulation was sustained by 16 computing cores on a single `ec2` instance. It is worth noting that the time

to completion in this case matches the time to completion obtained using 16 computing cores on `lonestar`. Most significantly, the time to completion required by `ec2-1` when using 8 computing cores is lower than the time required by `lonestar` with the same number of computing cores. This result suggests that one of the advantages of IaaS clouds is the availability of powerful hardware configurations (both in terms of memory and CPU clock speed), that can match and outperform the computing nodes provided by standard grid resources. On the other hand, the performance of `ec2` platforms seems to be sensitive to overload of the instances, as shown by the poor strong scaling achieved by `ec2-1` when all of the 16 available computing cores are used for the execution of the simulation. Point **D** corresponds to the fastest execution on `ec2` resources, that is running the simulation with 32 computing cores using `ec2-16`. In this case, we launched 16 EC2 instances and allocated 2 computing cores on each instance in a round robin fashion. The loss of performance of `ec2-16` as the number of cores per instance increases suggests that when requiring a relatively large number of instances, the physical connectivity of the nodes may become an issue, and the timings seem to be dominated by communication overheads.

Based on the metric *time to completion* we can rank the different resources – Table 3 shows the wall clock times for the fastest run on each platform. `lonestar` is by far the fastest resource, while `ec2` is generally the slowest. However, one of the solutions provided by `ec2` (namely `ec2-16`) matches the performance of on-premise resources (`ellipse` and `puma`), using a significantly smaller amount of computing cores. This result further demonstrates one of the strengths of on-demand resources as compared to on-premise resources, i.e., `ec2` can count on a more efficient hardware configuration. `rockhopper` performs best among the tested on-demand resources and better than the tested on-premise resources. However, it is still significantly slower than the tested HPC cluster.

7.4. Cost analysis

The cost of a complete simulation of the test problem varies significantly across the hardware platforms. The cost of the simulation on a given hardware resource is related to its performance, as mentioned above, and to the actual price paid for usage.

The graph in Figure 5 shows the cost of simulation for the different architectures as a function of the number of computing cores. For all the

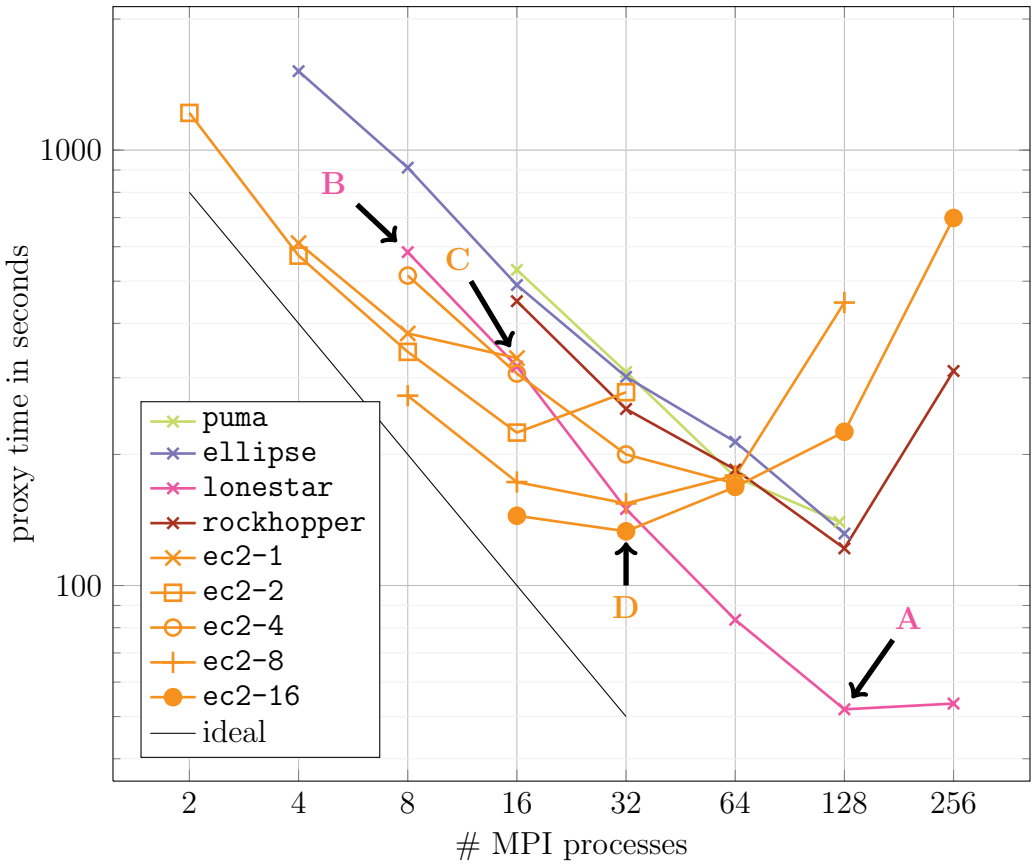

Figure 4: The average computation time per simulated time step (proxy), for the benchmarked architectures

considered HPC platforms, the cost increases with the number of computing cores involved in the computation. If the platform were able to achieve an ideal scaling, this expense would be perfectly balanced by the shorter time to completion. In reality, as the scaling factor is not perfect, the most cost-efficient scenario is generally achieved the minimal number of cores involved in the computation. Nonetheless, consistent with what was observed in the previous section, it can be convenient – from the user stand point – to increase the number of computing cores when using HPC machines. The increase in cost due to the use of a larger share of computing resources remains bounded thanks to the good scaling achieved by the platform, and on the other hand the overall computing time decreases. The cost of running

rank	time to completion [s]	target	# of MPI proc.
1	1h 31m	`lonestar`	128
2	3h 33m	`rockhopper`	128
3	3h 50m	`ellipse`	128
4	3h 53m	`ec2-16`	32
5	4h 05m	`puma`	124*
6	4h 30m	`ec2-8`	32
7	5h 00m	`ec2-4`	64
8	6h 33m	`ec2-2`	16
9	9h 43m	`ec2-1`	16

Table 3: The performance ranking of the hardware resources based on the metric *time to completion*.
* One node is permanently down.

a complete simulation on `lonestar`, in particular, is fairly independent of the number of cores used, up to 128 cores (consistent with the performance analysis). Points **A** and **B** in Figure 5 correspond to the two use cases of `lonestar` that have been discussed in the previous section. While the cost of the run on 128 cores (point **A**: \$13.6) is 43% larger than the cost of the run on 8 cores (point **B**: \$9.5), the simulation time is notably lower by a factor 11.2 (cf. Figure 4). `puma`, `ellipse`, and `rockhopper` perform less well in this respect, with the cost steadily increasing with the number of cores.

In the case of IaaS clouds, the poorer scaling performance in terms of time to completion does not effectively balance the cost increase with increased number of computing cores. Moreover, since the policy of use requires that clients pay for the reserved nodes rather than per core, the cost of the resource is not proportional to the actual use. The cost of `ec2` instances has therefore a complex dependency on the number of computing cores; `ec2-1`, `ec2-2`, `ec2-4` are more cost-effective when using more cores, while `ec2-8` and `ec2-16` tend to exhibit an optimal cost-effectiveness in a limited range of configurations (16-64 (or 2-4 processes per node) and 16-32 (or 1-2 processes per node), respectively). Point **C** in Figure 5 corresponds to the cheapest execution in our experiment, in which `ec2-1` is fully utilized (all the computing cores on a single instance are used). The cost characteristic of this platform reflects Amazon's pricing policy. In fact, it was not convenient to run our test on `ec2-1` with a smaller number of computing cores, because the per-hour cost of the resource was the same and the simulation ran for a longer time (as

rank	price per simulation [$]	target	# of MPI proc.
1	$3.53	`ec2-1`	16
2	$4.76	`ec2-2`	16
3	$5.31	`ellipse`	4
4	$5.70	`puma`	16
5	$7.27	`ec2-4`	64
6	$9.52	`lonestar`	8
7	$13.09	`ec2-8`	32
8	$20.99	`rockhopper`	16
9	$22.59	`ec2-16`	32

Table 4: Cost of the benchmarked architectures

shown in Figure 4). Point **D** in Figure 5 corresponds to the fastest execution on `ec2` resources in our experiment, i.e. 32 computing cores on `ec2-16`. The combined effects of Amazon's pricing policy (pay per-instance rather than per-core) and the performance drawbacks of using a large number of `ec2` instances (as discussed in the previous section) cause this case to rank among the most expensive tested in our experiment.

A ranking of the different platforms based on the metric *cost per simulation* is shown in Table 4. For each platform, we reported in the table the cost of the cheapest use case.

7.5. Utility function

The two objectives of minimizing the simulation cost and the time to completion compete with each other. This is confirmed in our tests: the cheapest resource, namely `ec2-1`, was also the slowest one. In Figure 6 we present how the cost per simulation relates to the time to completion for the different architectures. The general trend of these characteristics shows an increase in the cost per simulation with the decreasing time to completion. The closest points of the graphs to the origin of the axes represent execution cases that minimize both metrics. Clearly, the decision on which architecture to prefer cannot be made based on a single attribute.

To define a ranking of the tested platforms based on a user-centric performance analysis we evaluate the utility function defined in section 6. We consider three user profiles,

Case 1. The job has high priority, and the user has little delay tolerance;

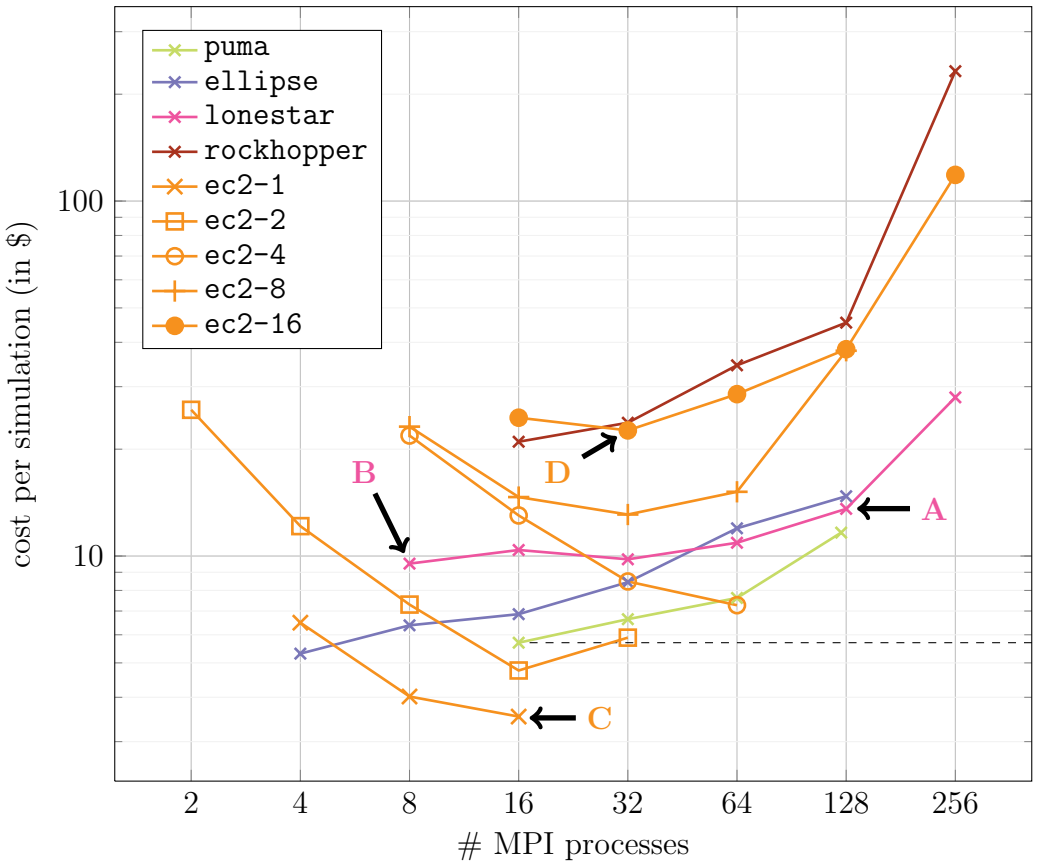

Figure 5: Cost per simulation for the different platforms. The dashed line shows the ideal cost characteristic for the in-house cluster `puma`.

Case 2. The job has average priority, and the user has average delay tolerance;

Case 3. The job has low priority, and the user has large delay tolerance.

Referring for the sake of example to the results of our benchmark, we assume that the value of a simulation to the user (i.e., the cost the user would be willing to pay) is in the range between \$3.53 (low) and \$22.59 (high, cf. table 4). More precisely, we assume that a job with low priority has a value to the user equal to the average cost of the simulation over the tested architectures, i.e., \$10.31. We assign double this value to a high priority job (\$20.62) while an average priority job will have an intermediate value between the previous two (\$15.465). We further assume that for all

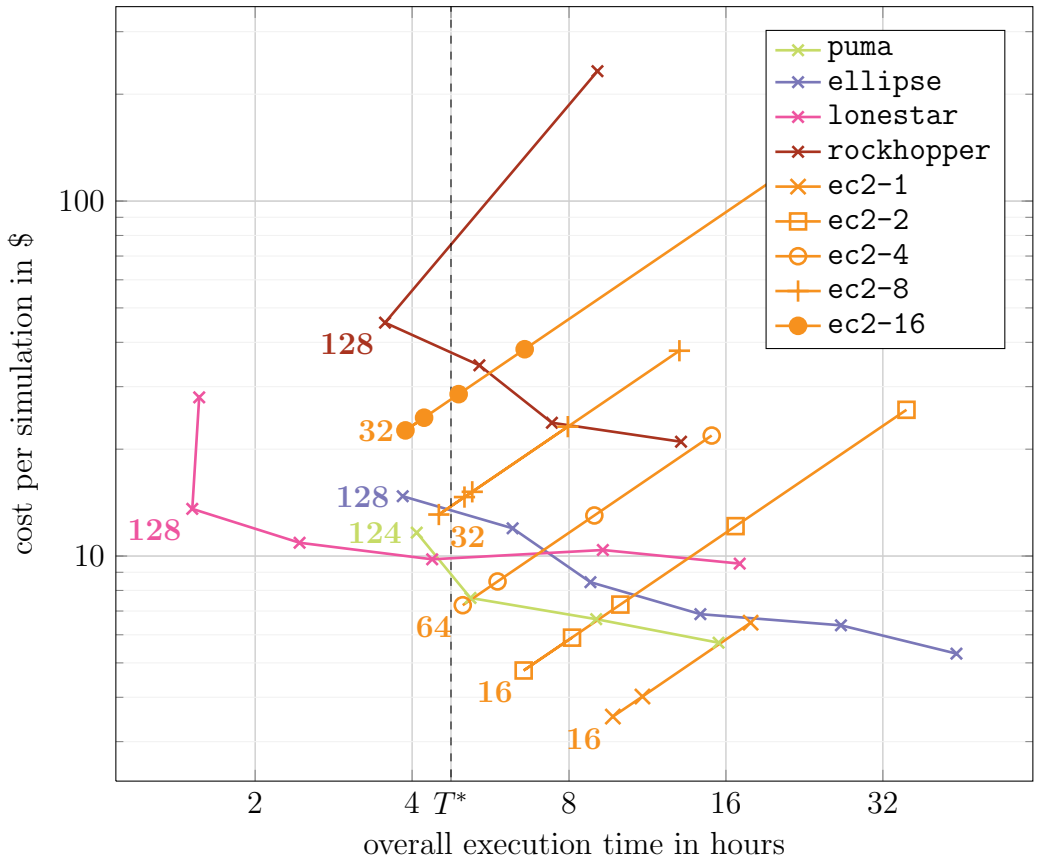

Figure 6: Relation between the cost and time of the simulation. The labels indicate the number of MPI processes in the fastest run. $T^* = 4h\ 44m$ corresponds to the average value among the fastest runs.

the user profiles the expected time to completion T^* is the average value of the times measured on the different architectures (cf. table 3), i.e., $T^* =$ 4h 44m. A user with an average delay tolerance is represented by a utility function that remains non-negative for a runtime up to twice the expected value (i.e., $T_0 = 2T^*$). A user with large delay tolerance accepts twice as much delay ($T_0 = 3T^*$), while a user with small delay tolerance accepts half as much (i.e., $T_0 = 1.5T^*$).

We plot in Figure 7 the user-specific utility functions together with the graphs shown in Figure 6. As discussed in previous sections, each platform was tested in several use cases (varying the number of computing cores); a use case is considered *useful* to the user if it is represented by a point on

the cost/time plot located below the graph of the user's utility function. For the sake of example, we reported on the plot the points corresponding to the cases discussed in detail in the previous sections. Point **A** corresponds to the fastest execution of the simulation in our experiment, obtained when using 128 cores on `lonestar`. This case is *useful* to user profiles 1 and 2, for which the *importance* of the simulation is greater than the actual cost. User profile 3 would not consider this case *useful* due to its high cost.

Despite the cost being relatively lower, the use case of `lonestar` represented by point **B** (8 computing cores) is not *useful* for any user profile, because for all of the profiles the time to completion of the simulation exceeds the time T_0 for which the utility function is zero. The use case of `ec2-1` corresponding to the cheapest execution in our experiment (16 computing cores) is represented by point **C**; because of the long time to completion, this use case is only *useful* to user profile 3. The fastest execution achieved on `ec2` resources (2 computing cores on each instance of `ec2-16`) is represented by point **D**. This use case has a cost exceeding the maximum value of the utility function for all the user profiles, so it is *useful* to none of them.

In our experiment, a variety of platforms can meet the requirements of user profile 1. Fast and expensive architectures (e.g., `lonestar`) can be chosen in alternative to slower and cheaper ones (e.g., `ec2`). However, because of a small delay tolerance, a cheap option (`ec2-2`) has to be ruled out, being penalized by high execution times. The second user profile has the largest pool of *useful* choices, including the cheaper (and slower) `ec2-2`. For user profile 3, because of the low priority assigned by the user to the job, most of the fastest options (`lonestar`, `ellipse`) have to be discarded. On the other hand, the on-premise cluster `puma` and some of Amazon's instances (most significantly the very cheap `ec2-1`) can meet the user's requests.

According to this model, one of the on-demand resources (`rockhopper`) is not *useful* to any of the considered user profiles. Amazon's diverse offering allows instead this service to be competitive for a wide range of user profiles, being able to provide reasonably small execution times (`ec2-8`) or extremely cheap solutions (`ec2-1`). On-premise resources do not perform well compared to HPC machines, being significantly slower, and in most cases they are also outperformed by cheaper on-demand resources. As a result, they are competitive only in specific execution cases (i.e., with the proper choice of the number of computing cores). Finally, `lonestar` is a very strong competitor in the first two user scenarios (average to high job priority), while its performance is matched and outperformed both by on-demand and on-premise

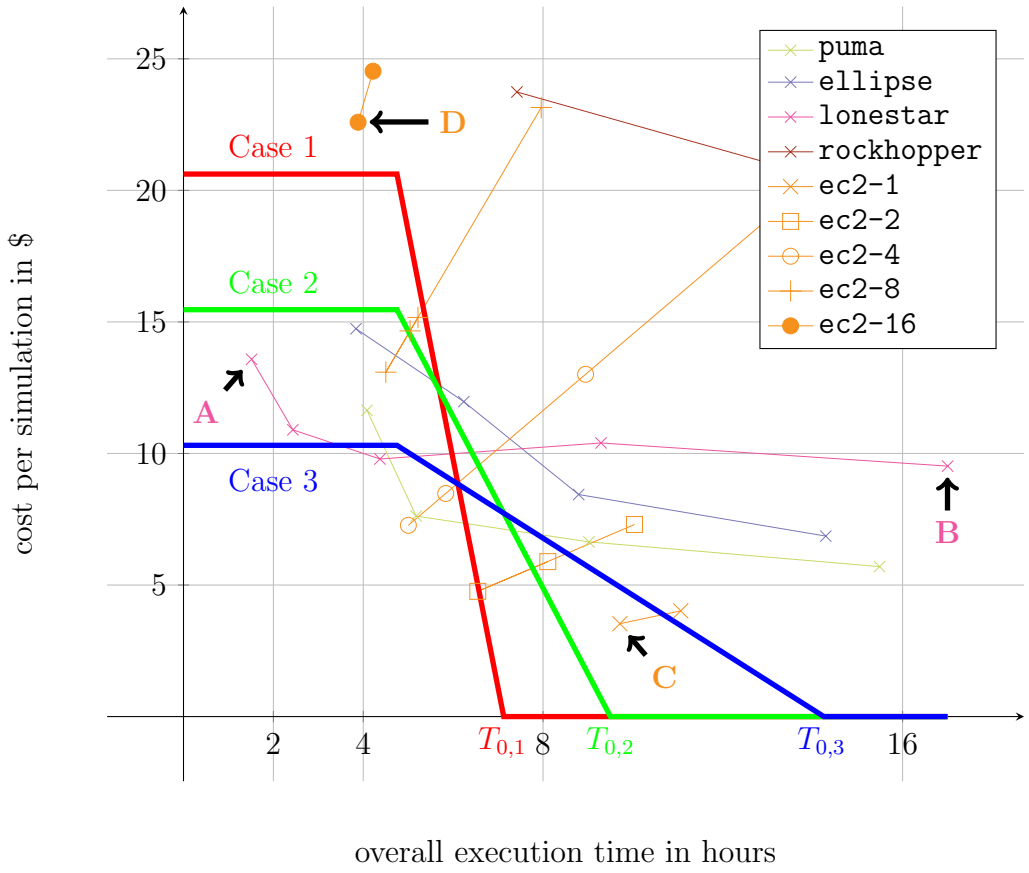

Figure 7: Evaluating the cost/time characteristics of the different platforms against the user-specific utility function. $T_{0,1}$, $T_{0,2}$ and $T_{0,3}$ are the times at which the utility function is zero for user profiles 1, 2 and 3, respectively.

resources in the third scenario (low job priority and high delay tolerance).

Notably, the on-demand cutting edge offering by Amazon EC2 has the advantage of availability. In fact, our analysis does not consider queue waiting times that may diminish the attractiveness of shorter execution time on grid resources. This feature would make the IaaS choice even more convenient. Moreover, the cost per simulation on the resources offered by Amazon can be optimized with a proper scheduling policy that takes into account the specific pricing policy of Amazon (per-node rather than per-core). Furthermore, if cost needs to be minimized, it is possible to select cheaper Amazon instances such as `cc2.4xlarge`.

8. Summary and future work

We have presented experiences and observations based on our exercise to deploy a production CFD code on five different target platforms characterized by heterogeneity in secondary attributes. Noteworthy are the difficulties that we encountered to simply provision the application, including package and library installation and other logistical hurdles.

Comparing execution time and cost of the application on on-premise and on-demand targets, we found some evidence to support the claim that IaaS resources may be utilized for scientific CFD simulations possibly at lower cost than incurred locally. In particular, our test with Amazon's spot-request feature coupled with availability of cutting edge resources (16-core nodes, 60GB RAM), suggests that small on-demand assemblies may be a viable alternative to local clusters. It is not without significance that IaaS's provide resources immediately, while local and grid resources are often subject to long queue wait times – an aspect that might offset any additional expense. Furthermore, while a modern local computing cluster with an efficient interconnection network will outperform an on-demand assembly (which is highly vulnerable to network performance) the cloud solution might be useful when cost needs to be minimized.

An important issue concerns the effort required for preparing the target platforms. In this study, we provisioned the runtime environment on all machines manually, by installing only the necessary and sufficient packages. We observe that the set up of a single machine took a few hours for an experienced member of the development team. As part of the ADAPT project, we are investigating schemes for automating these tasks. We are also exploring the use of third party software to address mundane, repeatable tasks (e.g. [7]) or the tailored generation of predefined images for IaaS ([40, 18]) that could significantly reduce the deployment effort on heterogeneous target platforms.

Acknowledgments

The authors gratefully acknowledge support by the Computational and Life Sciences (CLS) Strategic Initiative of Emory University. This research was supported in part by US National Science Foundation Grant OCI-1124418, and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

[1] Amazon Web Services/Developer Tools, http://aws.amazon.com/developertools, 2012.

[2] Boost C++ Libraries, http://www.boost.org, 2012.

[3] boto - Python interface to Amazon Web Services, http://code.google.com/p/boto/, 2012.

[4] CAELinux, http://www.caelinux.com/CMS/, 2012.

[5] Code_Aster, http://www.code-aster.org/, 2012.

[6] Code_Saturne, http://research.edf.com/research-and-the-scientific-community/software/code-saturne/introduction-code-saturne-80058.html, 2012.

[7] DoIt Automation Tool, http://python-doit.sourceforge.net/, 2012.

[8] Elmer, http://www.csc.fi/english/pages/elmer, 2012.

[9] HPC Cloud Service, Penguin Computing, http://www.penguincomputing.com/Services/HPCCloud, 2012.

[10] LifeV Project, http://www.lifev.org, 2012.

[11] Matlab, www.mathworks.com, 2012.

[12] OpenFOAM, http://www.openfoam.com/, 2012.

[13] ParaView, http://www.paraview.org, 2012.

[14] Penguin Computing On Demand / Indiana University, https://podiu.penguincomputing.com/, 2012.

[15] R, http://www.r-project.org/, 2012.

[16] Salome, http://www.salome-platform.org/, 2012.

[17] SuiteSparse, http://www.cise.ufl.edu/research/sparse/SuiteSparse/, 2012.

[18] Using OpenFOAM with Amazon EC2, http://www.openfoam.com/resources/ec2.php, 2012.

[19] S. Akioka, Y. Muraoka, HPC benchmarks on Amazon EC2, in: 2010 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA), IEEE, pp. 1029–1034.

[20] AMD, AMD Core Math Library (ACML), http://www.amd.com/acml, 2012.

[21] J. Bourgeois, V. Sunderam, J. Slawinski, B. Cornea, Extending executability of applications on varied target platforms, in: 2011 13th International Conference on High Performance Computing and Communications (HPCC), IEEE, pp. 253–260.

[22] S. Brenner, L. Scott, The mathematical theory of finite element methods, volume 15, Springer Verlag, 2008.

[23] G. Cheliotis, C. Kenyon, R. Buyya, A. Melbourne, Grid economics: 10 lessons from finance, GRIDS Lab and IBM Research Zurich, Melbourne, Tech. Rep (2003).

[24] B. Chun, D. Culler, User-centric performance analysis of market-based cluster batch schedulers, in: 2002 2nd International Symposium on Cluster Computing and the Grid, IEEE/ACM, pp. 30–30.

[25] E. Deelman, G. Singh, M. Livny, B. Berriman, J. Good, The cost of doing science on the cloud: the montage example, in: 2008 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, pp. 1–12.

[26] H. Elman, D. Silvester, A. Wathen, Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics, Oxford University Press, USA, 2005.

[27] A. Ern, J. Guermond, Theory and practice of finite elements, volume 159, Springer Verlag, 2004.

[28] C. Evangelinos, C. Hill, Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon ec2., ratio 2 (2008) 2–34.

[29] L. Formaggia, K. Perktold, A. Quarteroni, Basic mathematical models and motivations, in: L. Formaggia, A. Quarteroni, A. Veneziani, A. Quarteroni, T. Hou, C. Bris, A.T. Patera, E. Zuazua (Eds.), Cardiovascular Mathematics, volume 1 of *MS&A*, Springer Milan, 2009, pp. 47–75.

[30] L. Formaggia, F. Saleri, A. Veneziani, Solving numerical PDE's: problems, applications, exercises, Springer Verlag, 2012.

[31] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, in: 2008 Grid Computing Environments Workshop (GCE '08), IEEE, pp. 1–10.

[32] C. Geuzaine, J. Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre-and post-processing facilities, Int J Numer Meth Engng 79 (2009) 1309–1331.

[33] D. Gottfrid, Self-service, prorated super computing fun!, The New York Times 1 (2007).

[34] C.M. Haggerty, D.A. de Zlicourt, M. Restrepo, J. Rossignac, T.L. Spray, K.R. Kanter, M.A. Fogel, A.P. Yoganathan, Comparing pre- and post-operative fontan hemodynamic simulations: Implications for the reliability of surgical planning, Ann Biomed Eng (2012).

[35] Z. Hill, M. Humphrey, A Quantitative Analysis of High Performance Computing with Amazons EC2 Infrastructure: The Death of the Local Cluster?, in: 2009 10th International Conference on Grid Computing (Grid 2009), IEEE/ACM, pp. 26–33.

[36] D.E. Irwin, L.E. Grit, J.S. Chase, Balancing Risk and Reward in a Market-Based Task Service, in: 2004 13th International Symposium on High Performance Distributed Computing (HPDC '04), IEEE, 2004.

[37] S. Joachim, G. Hannes, G. Robert, NETGEN - automatic mesh generator, `http://www.hpfem.jku.at/netgen`, 2012.

[38] G. Karypis, V. Kumar, MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, `http://www.cs.umn.edu/~metis`, 2009.

[39] C. Lee, A. Snavely, Precise and realistic utility functions for user-centric performance analysis of schedulers, in: 2007 16th International Symposium on High Performance Distributed Computing (HPDC '07), ACM, pp. 107–116.

[40] MIT, StarCluster - Software Tools for Academics and Researchers, `http://web.mit.edu/stardev/cluster/`, 2012.

[41] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, D. Epema, A performance analysis of ec2 cloud computing services for scientific computing, Cloud Computing (2010) 115–131.

[42] M. Piccinelli, A. Veneziani, D.A. Steinman, A. Remuzzi, L. Antiga, A framework for geometric analysis of vascular structures: application to cerebral aneurysms, IEEE Trans Med Imaging 28 (2009) 1141–55.

[43] L. Quartapelle, Numerical solution of the incompressible Navier-Stokes equations, volume 113, Birkhauser Basel, 1993.

[44] A. Quarteroni, R. Sacco, F. Saleri, Numerical mathematics, Text in Applied Mathematics, Springer, New York, 2007.

[45] S. Salsa, Partial differential equations in action: from modelling to theory, Springer Verlag, 2008.

[46] Sandia National Laboratories, The Trilinos Project, `http://trilinos.sandia.gov`, 2012.

[47] J. Slawinski, T. Passerini, U. Villa, A. Veneziani, V. Sunderam, Experiences with target-platform heterogeneity in clouds, grids, and on-premises resources, in: 2012 26th International Parallel and Distributed Processing Symposium (IPDPS-HCW), IEEE, pp. 41–52.

[48] J. Slawinski, M. Slawinska, V. Sunderam, The Unibus Approach to Provisioning Software Applications on Diverse Computing Resources, in: 2009 International Conference On High Performance Computing, 3rd International Workshop on Utility and Grid Computing (HIPCWUGC).

[49] P. Smith, A Cost-Benefit Analysis of a Campus Computing Grid, Master's thesis, Purdue University, 2011.

[50] S.F.C. Stewart, E.G. Paterson, G.W. Burgreen, P. Hariharan, M. Giarra, V. Reddy, S.W. Day, K.B. Manning, S. Deutsch, M.R. Berman, M.R. Myers, R.A. Malinauskas, Assessment of CFD Performance in Simulations of an Idealized Medical Device: Results of FDA's First Computational Interlaboratory Study, Cardiovascular Engineering and Technology 3 (2012) 139–160.

[51] The HDF Group, Hierarchical data format version 5, `http://www.hdfgroup.org/HDF5`, 2012.

[52] D.A. de Zélicourt, C.M. Haggerty, K.S. Sundareswaran, B.S. Whited, J.R. Rossignac, K.R. Kanter, J.W. Gaynor, T.L. Spray, F. Sotiropoulos, M.A. Fogel, A.P. Yoganathan, Individualized computer-based surgical planning to address pulmonary arteriovenous malformations in patients with a single ventricle with an interrupted inferior vena cava and azygous continuation, J. Thorac. Cardiovasc. Surg. 141 (2011) 1170–1177.