Federated Computing for the Masses – Aggregating Resources to Tackle Large-scale Engineering Problems

Technical Report: TR-RDI2-20130515-0
May 15, 2013

Javier Diaz-Montes
Rutgers Discovery Informatics Institute
Rutgers University

Baskar Ganapathysubramanian
Department of Mechanical Engineering
Iowa State University

Manish Parashar
Rutgers Discovery Informatics Institute
Rutgers University

Ivan Rodero
Rutgers Discovery Informatics Institute
Rutgers University

Yu Xie
Department of Mechanical Engineering
Iowa State University

Jaroslaw Zola
Rutgers Discovery Informatics Institute
Rutgers University

Rutgers Discovery Informatics Institute (RDI^2)
Rutgers, The State University of New Jersey
96 Frelinghuysen Road, Piscataway, NJ 08854
http://rdi2.rutgers.edu/

Abstract

The complexity of many problems in science and engineering requires computational capacity greatly exceeding what average user can expect from a single computational center. While many of these problems can be viewed as a set of independent tasks, their collective complexity easily requires millions of core-hours on any state-of-the-art HPC resource, and throughput that cannot be sustained by a single multi-user queuing system. In this report we explore the use of aggregated heterogeneous HPC resources to solve large-scale engineering problems. We show that it is possible to build a computational federation that is easy to use by end-users, and at the same time is elastic, resilient and scalable. We argue that the fusion of federated computing and real-life engineering problems can be brought to average user if relevant middleware infrastructure is provided. To support our claims, we report on the use of federation of 10 geographically distributed heterogeneous HPC resources to perform a large-scale interrogation of the parameter space in the microscale fluid flow problem.

1 Introduction

The ever-growing complexity of scientific and engineering problems continues to pose new requirements and challenges for computing and data management. The analysis of high-dimensional parameter spaces, uncertainty quantification by stochastic sampling, or statistical significance assessment through resampling, are just few examples of a broad class of problems that are becoming increasingly important in a wide range of application domains. These "ensemble", also termed as many task computing, applications consist of a set of heterogeneous computationally intensive, and independent or loosely coupled tasks, and can easily consume millions of core-hours on any state-of-the-art HPC resource. While many of these problems are conveniently parallel, their collective complexity exceeds computational time and throughput that average user can obtain from a single computational center. For instance, the fluid flow problem we consider comprises more than ten thousand MPI tasks, and would require approximately 1.5 million core-hours to solve on the *Stampede* cluster at TACC – one of the most powerful machines within XSEDE [12]. Although XSEDE allocations of that size are not uncommon, the heavy utilization of *Stampede*, and its typical queue waiting times make it virtually impossible to execute that number of tasks within an acceptable time limit. The problem becomes even more complex if we take into account that individual tasks are heterogeneous, and add in the possibility of failures that are not uncommon in large-scale multi-user systems.

The above constraints are not unique to one particular problem or a system. Rather, they represent common obstacles that can limit the scale of problems that can be considered by an ordinary researcher on a single, even very powerful, system. What is important, this trend continues and one can only expect that more and more users will require computational throughput that cannot be delivered just by one resource. In order to overcome these limitations two important questions have to be addressed. First, how to empower a researcher with computational capability that is compatible to what currently is reserved for the "elite" problems. Second, how to deliver this capability in a user-centered way.

In this report, we argue that both questions can be answered by implementing a federation model in which a user, without any special privileges, can seamlessly aggregate multiple, globally distributed and heterogeneous HPC resources exploiting their intrinsic capabilities. Then, we show how federated computing can be used to solve in a user-centered way the actual large-scale computational problem in engineering. Our work is motivated by the need to understand flow behavior in microfluidic devices. Our focus is on empowering average user with aggregated computational capabilities typically reserved for selected high-profile problems. To achieve this, we aggregate heterogeneous HPC resources in the spirit of how volunteer computing assembles desktop computers. Specifically, we introduce a model of computational federation that i) is extremely easy to deploy and offers an intuitive API to meet expectations and needs of average user; ii) encapsulates cloud-like capabilities, e.g. on-demand resource provisioning, elasticity and resilience, to provide sustainable computational throughput; iii) provides strong fault-tolerance guarantees through constant monitoring of tasks and resources; iv) bridges multiple, highly heterogeneous resources, e.g. servers, clusters, supercomputers and clouds, to effectively exploit their intrinsic capabilities; and v) leverages existing hardware/software infrastructure.

To solve the computational problem we federate 10 machines from 3 countries and execute

12,845 MPI-based simulations, that collectively consume 2.5 million core-hours.

2 Problem Description

The ability to control fluid streams at microscale has significant applications in many domains, including biological processing [9], guiding chemical reactions [3], and creating structured materials [4], to name just a few. Two of the authors, henceforth referred to as the end-user, are part of a team that recently discovered that placing pillars of different dimensions, and at different offsets, allows "sculpting" the fluid flow in microchannels [1]. The design and placement of sequences of pillars allows a phenomenal degree of flexibility to program the flow for various bio-medical and manufacturing applications. However, to achieve such a control it is necessary to understand how flow is affected by different input parameters. This problem is highly representative for a broad category of parameter space interrogation techniques, which are essential for understanding how process variables affect behavior of the modeled system, to quantify uncertainty of the model when input data is incomplete or noisy, or to establish a ground on which inverse problems can be investigated. While these techniques are very diverse, typically they involve a large collection of computationally intensive tasks, with little or no synchronization between the tasks.

The end-user developed a parallel, MPI-based Navier-Stokes equation solver, which can be used to simulate flows in a microchannel with an embedded pillar obstacle. Here, the microchannel with the pillar is a building block that implements a fluid transformation (see Fig. 1). For a given combination of microchannel height, pillar location, pillar diameter, and Reynolds number (four variables), the solver captures both qualitative and quantitative characteristics of flow. In order to reveal how the input parameters interplay, and how they impact flow, the end-user seeks to construct a phase diagram of possible flow behaviors. In addition, the end-user would like to create a library of single pillar transformations to enable analysis of sequences of pillars. This amounts to interrogating the resulting 4D parameter space, in which a single point is equivalent to a parallel Navier-Stokes simulation with a specific configuration.

The problem is challenging for several reasons. The search space consists of tens of thousands of points, and an individual simulation may take hundreds of core-hours, even when executed on a state-of-the-art HPC cluster. For example, the specific instance we consider requires 12,400 simulations. The individual tasks, although independent, are highly heterogeneous and their

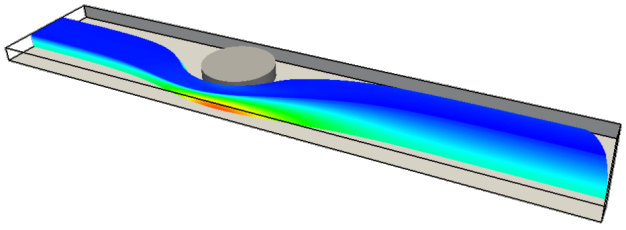

Figure 1: Example flow in a microchannel with a pillar. Four variables characterize the simulation: channel height, pillar location, pillar diameter, and Reynolds number. Please view in color.

cost of execution is very difficult to estimate *a priori*, owing to varying resolution and mesh density required for different configurations. In our case, the cost may range from 100 core-hours to 100,000 core-hours per task executed on the IBM Blue Gene/P. Consequently, scheduling and coordination of the execution cannot be performed manually, and a single system cannot support it. Finally, because the non-linear solver is iterative, it may fail to converge for some combinations of input parameters, in which case fault-tolerance mechanisms should be engaged. The above properties make the problem impossible for the end-user to solve using the standard computational resources (e.g. computational allocation from XSEDE).

3 Federation for the Masses

As we already mentioned, the presented problem is representative for a much broader class of parameter space interrogation techniques. Here, classic examples are Monte Carlo methods, stochastic sampling strategies (e.g. sparse grid collocation), or soft computing approaches (e.g. simulated annealing). These techniques constitute a significant fraction of all scientific codes in use today, and hence are of great practical importance to average user. Our goal is to develop a federation model that would be able to support resulting large-scale workloads, but at the same time would be user-centered. To build such a model it is imperative to understand two key elements. First, which specific properties of large-scale scientific and engineering applications must be taken into consideration to enable efficient execution in a large federated environment? Second, what kind of expectations must be addressed in order to achieve a user-centered design?

We start our considerations with answering the first question. A typical approach to investigate large search-spaces combines two elements: a master module that encapsulates a problem logic, e.g., to decide how the search-space should be navigated through, and a science-driver that implements the actual computational core. Usually, both elements are contained within separate software components, and problem logic can be implemented indirectly in the execution environment (e.g. as a script interacting with a queuing system). Individual instances of a science-driver are either independent or involve asynchronous communication. Naturally, complexity of both modules may vary drastically. However, in the vast majority of cases it is the science-driver that is represented by a complex parallel code, and requires HPC resources to execute. For instance, in our fluid flow challenge, the problem logic amounts to a simple enumeration of selected points in the search space, while the science-driver is a complex MPI-based fluid flow simulation. Although a science-driver is computationally challenging on its own, the actual complexity comes from the fact that the usual investigation involves large number of tasks (e.g. millions in any Monte Carlo analysis). Oftentimes a single resource is insufficient to execute the resulting workload either because of insufficient throughput or limited computational capability. Additionally, tasks might be heterogeneous and have diverse hardware requirements, or can be optimized for specific architectures. Moreover, except of simple scenarios, tasks are generated dynamically, based on partial or complete results delivered by previously completed tasks.

To answer the second question, we have to keep in mind that our focus is on a regular user with a need for large computational capacity. Such a user typically has access to several HPC resources which most likely are highly heterogeneous in terms of computational power, as well as underlying architecture, and are geographically distributed. The access is provided via a

standard environment, for example, a shell account. As a result, end-user is presented with a substantial computational power that nevertheless is scattered, highly diversified, lacks a unified exposition, and requires non-trivial coordination, ultimately hindering potential applicability.

Based on the above characteristic we can identify a several key features that a successful federation model must account for. If we consider workloads' properties then the federation must be elastic and scalable – the ability to scale up/out and down becomes essential to handle varying over time number of tasks. Additionally, the federation must be able to adapt to the diverse task requirements, and make optimal use of distinct features contributed by the heterogeneous federated resources. Consequently, capability, which we define as the ability of a federation to take advantage of particular hardware characteristics, must be the first-class citizen in the model. This requirement is synergistic with the concept of autonomic computing. To address user expectations, the federation must aggregate heterogeneous resources while operating completely in a user-space. After all, it is unrealistic to expect that a user will have an administrative privileges on any HPC resource. At the same time, the federation should hide low-level details, such as geographic location or hardware architecture, while offering a familiar programming interface, for example, supporting common parallel programming idioms like master/worker or MapReduce, which could be used directly by a user. Finally, a user must be able to deploy existing applications, i.e. science-drivers and sometimes a problem logic module, within the federation and without modifications.

3.1 A User-centered Approach to Federation

To deliver a federation model with properties highlighted in the previous section we focused on usability, elasticity and resilience as primary objectives. In our model, the underlying infrastructure is presented as a single pool of resources regardless of their physical location. The design is based on four layers, where the lowest layer is responsible for the interaction with physical resources, and the highest one is the actual user application. The appropriate provisioning of resources in accordance with user provided policies is realized by the cross-layer autonomic manager. The schematic representation of the design is presented in Figure 2.

The federation overlay is a self-organizing structure that provides a uniform view on top of physical resources. It allows users to add and remove resources dynamically as needed, and handles network and resource failures. It is also able to interact with heterogeneous resources ensuring interoperability. The layer provides a scalable content-based routing and messaging system that is based on Chord [8]. The routing engine addresses resources using attributes rather than of specific addresses, and supports flexible content-aware routing, and complex querying using partial keywords, wildcards, or ranges. It also guarantees that all peer nodes with data elements that match a query/message are located [6, 7]. The messaging system guarantees that messages, specified via flexible content descriptors, are served with bounded cost across resources [6, 7].

The service layer provides a range of services to support autonomics at the programming and application level. It includes a coordination service that handles the execution of applications, a discovery service to find resources based on their properties, and an associative object store service to manage tasks and data. This layer supports a Linda-like [2] tuple space coordination model, and provides a virtual shared-space abstraction that can be associatively accessed

by all system entities without knowledge of the physical locations of the hosts over which the space is distributed. In addition, the layer provides dynamically constructed transient spaces to allow applications to explicitly exploit context locality to improve system performance. Finally, it equips the programming layer with asynchronous (publish/subscribe or push/pull) messaging and event services to simplify communication between peer nodes.

The programming layer provides the basic framework for application development and management. As such, this layer acts as an interface between the federation and a user. The layer supports several common distributed programming paradigms, including the master/worker, workflow and MapReduce. These programming abstractions ease the development of applications by decoupling the application from the particularities of the infrastructure. Using the API offered by the layer, user can generate tasks and their associated properties, which next are managed by the service layer and autonomic manager. The task consistency service, included in the layer, handles lost and failed tasks. Different components of the application, for instance master and worker, or mapper and reducer, can communicate via virtual shared space, or using a direct connection.

The application layer technically represents the final application developed by a user on top of the programming layer. However, in many cases a user might be interested in benefiting from the federation to execute third-party, perhaps closed-source, software. In such cases the target software cannot or should not be modified, for example due to efficiency considerations. To accommodate for this, the programming layer can still be used in the standard way, however, the resulting application becomes a mere container that acts as a facade for the target software. This tremendously simplifies migration from traditional environments to our federation model. We should keep in mind however, that in this scenario the target application must be deployed on the federated resources beforehand.

Autonomic manager in the final and key ingredient of the federation. The manager enables the autonomic management and multi-objective optimization (including performance, energy, cost, and reliability criteria) of application execution through cross-layer application/infrastructure

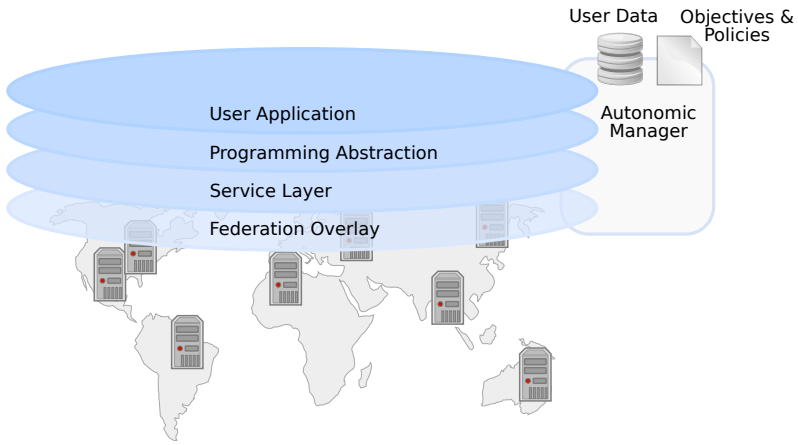

Figure 2: Multi-layer design of the proposed federation model. Here, the autonomic manager is a cross-layer component that based on user data and policies provisions appropriate resources.

adaptations. This component offers QoS by adapting the provisioned resources to the application's behavior as well as system configuration, which can change at run time, using the notion of elasticity at the application level. As a result, the federated infrastructure increases the opportunities to provision appropriate resources for given application based on user objectives or policies, and different resource classes can be mixed to achieve the user objectives. The manager can scale federation up/down/out based on the dynamic workload and provided user policies. For example, a user objective can be to accelerate the application execution within a given budget constraints, to complete the application within assumed deadline, or to use resources best matching to the application type (e.g computation vs. data intensive). Because application requirements and resource status may change, for example, due to workload surges, system failures or emergency system maintenance, the manager provisions resources adaptively to accommodate for these changes. Note, that the adaption ensures implicitly resilience of the federation.

3.2 Federation Architecture

The presented model allows the federation to dynamically evolve in terms of size and capabilities. In particular, the federation is created in a collaborative way, where sites talk with each other to identify themselves, negotiate the terms of adhesion, discover available resources, and advertise their own resources and capabilities. Therefore, a federated management space is created at runtime, while sites can join and leave at any point. As a part of the adhesion negotiation, sites may have to verify their identities using security mechanisms such as X.509 certificates, public/private key authentication, or others. The federation architecture is presented in Figure 3.

The federation is based on the Comet coordination spaces concept [5], which is supported by

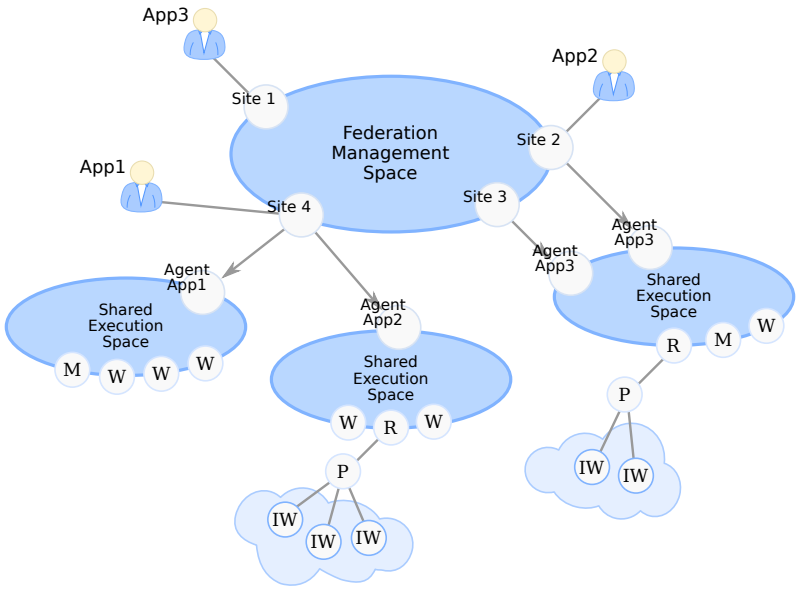

Figure 3: Architecture of the proposed federation model. Here, (M) denotes a master, (W) is a worker, (IW) an isolated worker, (P) a proxy, and (R) is a request handler. Arrows represent deployment of a computational site, while lines show communication channels.

the previously described model. The Comet spaces are used to coordinate different aspects of the federation. In particular, we decided to use two types of spaces. First, we have a single federated management space used to create the actual federation and orchestrate different resources. This space is used to interchange any operational message for discovering resources, announcing changes in a site, routing users' requests to appropriate sites, or initiating negotiations to create ad-hoc execution spaces. On the other hand, we can have multiple shared execution spaces that are created on demand to satisfy computational needs of the users. Execution spaces can be created in the context of a single site to provision local resources, and cloudburst to public clouds or external HPC systems. Moreover, they can be used to create a private sub-federation across several sites. This case can be useful when several sites have some common interest and they decide to jointly target certain type of tasks as a specialized community.

As shown in Figure 3, each shared execution space is controlled by an agent that creates the space, and coordinates the resources that execute a particular set of tasks. Agents can act as a master of the execution, or delegate this duty to a dedicated master (M) when some specific functionality is required. Moreover, agents deploy workers to perform the actual execution of tasks. These workers can be in a trusted network, be part of the shared execution space and store data, or they can be part of external resources such as public clouds, and therefore in a non-trusted network. The first type of workers is called secure (W) and can pull tasks directly from the space. Meanwhile, the second type is called isolated (IW) and cannot interact directly with the shared space. Instead, isolated workers have to interact with a proxy (P), and a request handler (R), to be able to pull tasks from the space. This distinction is important since it allows us to define specific boundaries in the way data is accessed. This fact in turn can be used to optimize the data storage and exploit data locality. Moreover, it also can be used to define security policies and decide who can access which data.

Users can access the federation and benefit from its capabilities from any participating site. Figure 4 presents the architectural details of a federated site. Here, we see two main components, namely resource manager and autonomic manager. The first one manages local resources, and

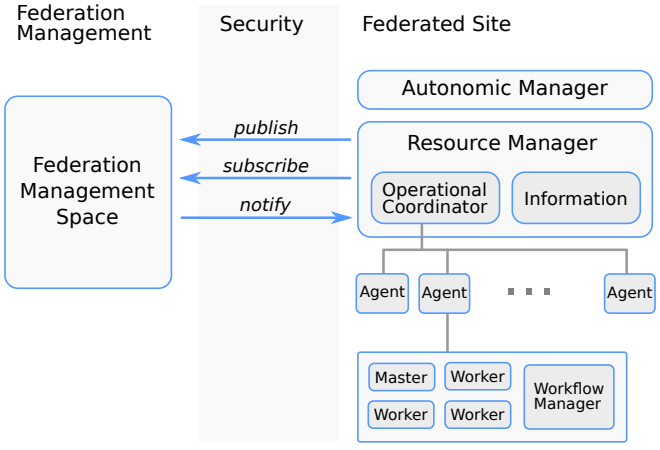

Figure 4: Main elements implementing a federated site. Each agent is responsible for executing tasks using one of the programming models (e.g. mater/worker, workflow).

elastically deploys agents to meet the computational requirements of the users. It also includes a monitoring system that collects status information of all federated resources. This information system can be used to announce the capabilities of the federation, to drive the execution of a user application, etc. On the other hand, the autonomic manager provides users with on-site autonomic capabilities. As we mentioned in Section 3.1, this component makes sure that user's application is executed within terms of the specified policies, and adapts the provisioned resources accordingly.

Federation sites interact with the rest of the federation though the federation management space in a publish/subscribe fashion. Each site publishes information about the status of its resources, the services they offer, or computational needs of its users. Additionally, each site creates subscriptions to be notified when there is some event of interest, such as for example that a user requests one of the offered services. Alternatively, the federation site is also able to work using a push/pull model.

4 Fluid Flow Analysis

To solve our fluid flow challenge we implemented the proposed federation model using the CometCloud framework [10]. CometCloud is an autonomic framework that enables dynamic, and on-demand federation of advanced cyberinfrastructures (ACI). It also provides a flexible programming platform, and API, to easily develop applications that can take advantage of federated ACIs. Furthermore, it enables dynamicity and fault-tolerance in the resulting infrastructure. As a result, sites can join and leave the federation at any moment without interrupting the execution.

We started the solving process by analyzing the simulation software, to decide how it should be integrated with the CometCloud infrastructure. Here, we concentrated on the following issues:

- *Input/output requirements* – What are the data formats used by the software? What is the scale of the data? Which output data has to be retained for a downstream analysis, and which can be discarded?
- *Error and exception handling* – How soft and hard errors are reported and handled in the software? What actions should be triggered by each error?
- *Scalability* – What is the minimal per-core memory requirement? In what core range the software exhibits malleable characteristics?
- *Software dependencies* – What third party packages are required by the software? How sensitive is the software to the choice of a compiler, and compiler options?

Based on the analysis we summarized the software requirements: The input data, that is finite element meshes, are stored in a custom format text files, with size varying from 1 MB to 5 MB. The main output is stored in the standard `PLT` format, with size varying from 25 MB to 100 MB. The software maintains weak scalability up to 2,048 cores across different interconnects, and scales linearly up to 256 cores. Furthermore, for many simulations the minimal configuration requires total 512 GB of RAM to execute. All errors in the software are reported through the

standard error output channel. Finally, the software is based on the PETSc library from the Argonne National Laboratory. This library is notoriously difficult to deploy properly, owing to a significant variability between different versions, and large battery of options.

We combined the MPI-based solver with the CometCloud infrastructure using the master/worker paradigm. In this scenario, the simulation software serves as a computational engine, while CometCloud is responsible for orchestrating the entire execution. The master/worker model is among several directly supported by CometCloud, and it perfectly matches problems with a large pool of independent tasks. The master component takes care of generating tasks, collecting results, verifying that all tasks executed properly, and keeping log of the execution. Here, each task is described by a simulation configuration (specific values of the input variables), and minimal hardware requirements. All tasks are automatically placed in the CometCloud-managed distributed task space for execution. In case of failed tasks the master recognizes the error and either directly resubmits task (in case of a hardware error or a resource leaving the federation), or regenerates it after first increasing the minimal hardware requirements and/or modifying solver parameters (in case of an application error and/or insufficient resources). In the proposed approach, a workers sole responsibility is to execute tasks pulled from the task space. To achieve this, each worker interacts with the respective queuing system and the native MPI library via a set of dedicated drivers implemented as simple shell scripts.

At this point, we should note that thanks to the space-based nature of CometCloud infrastructure, master/worker applications display a pull-based mechanism rather than the more prevalent push-based model. Each worker identifies tasks that best match its capabilities based on task properties (e.g. hardware requirements). Consequently, the heterogeneity of tasks and computational resources can be efficiently managed. Moreover, the entire platform becomes very flexible, for example, speculative execution and fault-tolerance can be naturally expressed.

The selected realization of computational federation has several important, and highly desired properties. The integration of the existing software with the CometCloud platform does not require any adjustments on the application side. The implementation of the master and worker components amounts to approximately 500 lines of a simple Java code, including the task generation logic, that would have to be implemented irrespective of the selected approach. The interaction with specific computational resources is based on drivers, that are actual shell scripts submitted to the queuing system. Finally, the entire infrastructure operates within shell accounts, without any special privileges. All this demonstrates the extreme ease of use and flexibility of the CometCloud-based solution, in tune with our goal of providing a user-centered solution.

4.1 Execution Environment

To interrogate the parameter space at the satisfactory precision level we identified 12,400 simulations (tasks) as essential. The estimated collective cost of these tasks is 1.5 million core-hours if executed on the *Stampede* cluster. While this number is already challenging, we note that approximately 300,000 tasks would be required to provide a fine-grained view of the parameter space. As we already mentioned, tasks are very heterogeneous in terms of hardware requirements and computational complexity. This is because of varying mesh density and size, as well as convergence rate of the solver. For instance, some tasks require minimum 512 GB of total RAM, while many can execute in 64 GB. To accommodate for this variability we classified tasks into

three groups (*small, medium, large*), based on their estimated minimal hardware requirements. Although this classification is necessarily error-prone, due to non-trivial dependencies between mesh size, and memory and time complexity, it serves as a good proxy based on which computational sites can decide which tasks to pull. At the same time, misclassified tasks can be handled by fault-tolerance mechanisms of CometCloud.

To execute the experiment we federated 10 different resources, provided by six institutions from three countries. The characteristics of the selected machines are outlined in Tables 1 and 2. As can be seen, utilized resources span different hardware architectures and queuing systems, ranging from the high-end supercomputers to small-scale servers. Depending on the hardware characteristics different machines accepted tasks from different classes (see Table 2). This was achieved by providing a simple configuration file to respective CometCloud worker. Our initial rough estimates indicated that the first seven machines (*Excalibur, Snake, Stampede, Lonestar, Hotel, India, Sierra*) would be sufficient to carry out the experiment, and conclude it within two weeks. However, during the experiment, as we explain later, we decided to integrate additional resources (*Carver, Hermes, Libra*). Because all machines were used within limits set by the hosting institutions no special arrangements were made with their system administrators, and both the-end users' software and CometCloud components were deployed using a basic `SSH` account.

4.2 Experimental Results

The experiment lasted 16 days during which 10 different HPC resources were federated, and total of 12,845 tasks were executed. Together, all tasks consumed 2,897,390 core-hours, and generated 398 GB of the output data. The progress of the experiment is summarized in Figure 5.

The initial configuration of the federation included only five machines (*Excalibur, Snake, Stampede, Lonestar, Hotel*) out of seven planned. Two other machines, *India* and *Sierra*, joined with a delay caused by maintenance issues. After the first day of execution it became apparent that more computational resources were needed to finish the experiment within assumed deadline. This is because some machines were experiencing problems, and more importantly, our XSEDE allocation on *Stampede* was being exhausted rapidly. At that point, the first significant feature of our solution came into play – thanks to the extreme flexibility of the CometCloud platform temporal failures of individual resources did not interrupt the overall progress, and adding new resources was possible within few minutes from the moment the access to a new resource was acquired, and the simulation software was deployed. Indeed, on the second day *Hermes* from Spain was added to the execution pool, and soon after NERSC's *Carver*, and *Libra* from Singapore were federated. Consequently, the federation was able to sustain computational performance. Figure 5 shows that most of the time anywhere between 5 and 25 simulations were running, despite multiple idle periods scattered across the majority of the machines. These idle periods were caused by common factors, such as for example, hardware failures and long waiting times in system queues. All failures were handled by the CometCloud fault-tolerance mechanism. During the experiment 249 tasks had to be regenerated due to hardware errors, and 167 due to inability of the solver to converge. We note, that 29 additional tasks were run as a result of a speculative execution. All this demonstrates great robustness of the framework – depending on the availability of resources, and the rate of the execution, federation can be scaled up or down accordingly.

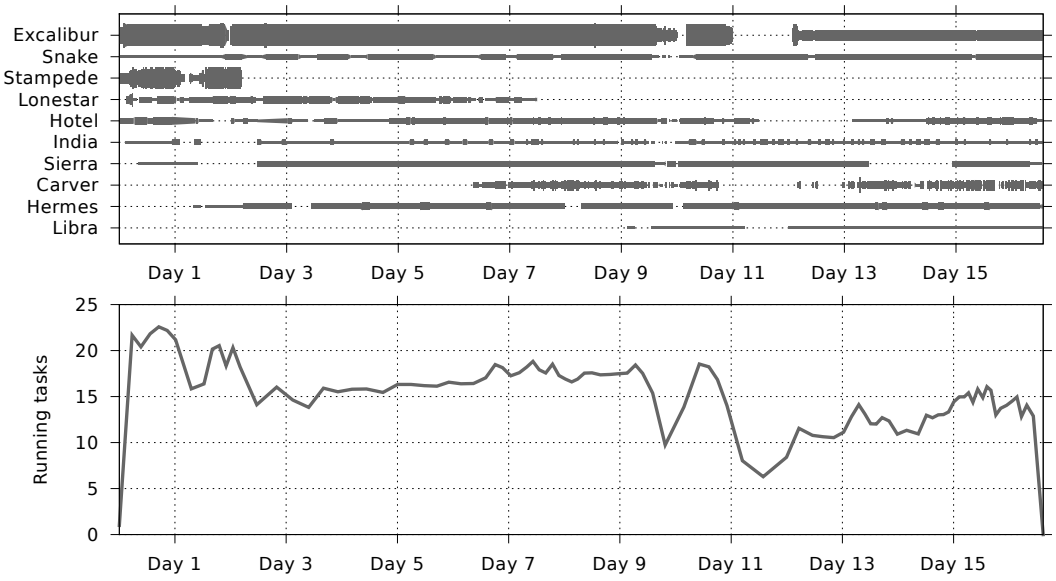

Figure 5: Summary of the experiment.

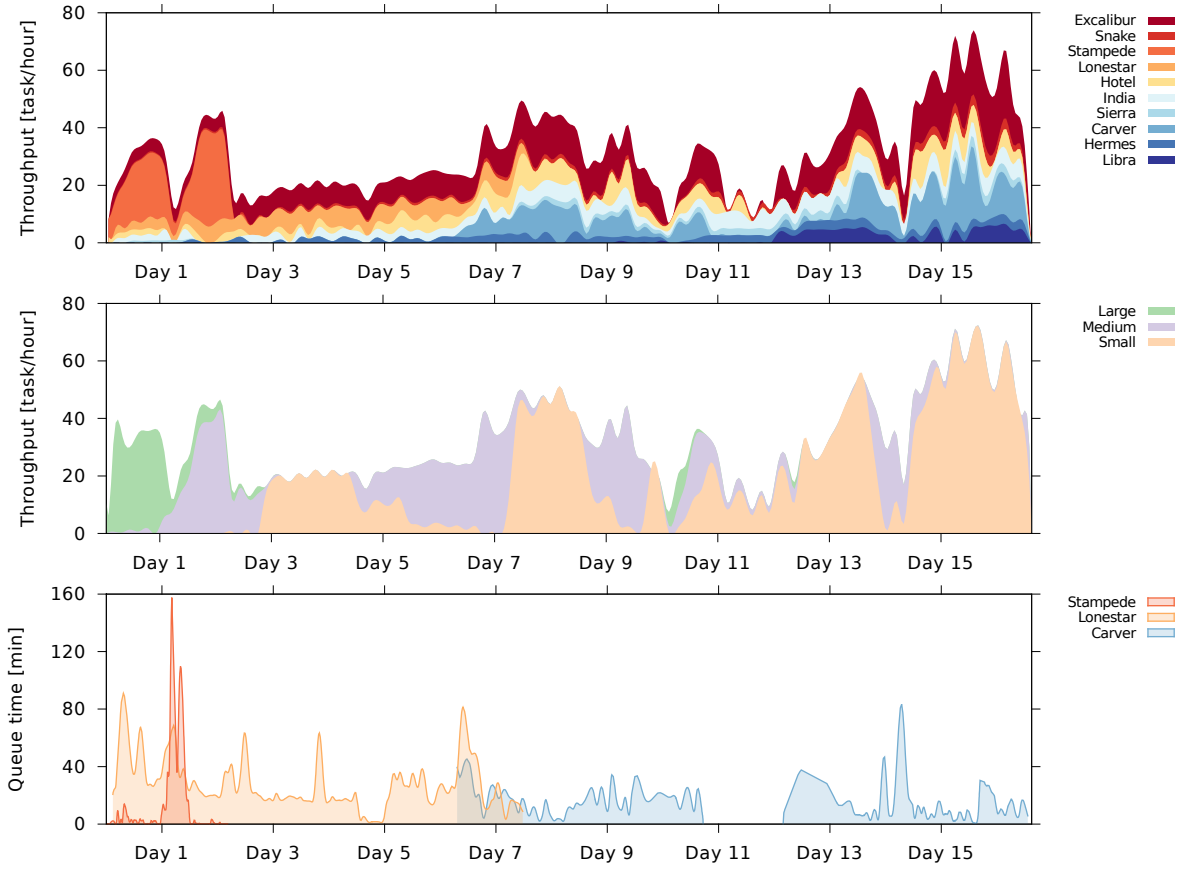

Figure 6: Throughput and queue waiting time. Please view in color.

Table 1: Computational resources used to execute the experiment.

Name	Provider	Type	Cores†	Memory‡	Network	Scheduler
Excalibur	RDI^2	IBM BG/P	8,192	512 MB	BG/P	LoadLeveler
Snake	RDI^2	Linux SMP	64	2 GB	N/A	N/A
Stampede	XSEDE	iDataPlex	1,024	4 GB	IB	SLURM
Lonestar	XSEDE	iDataPlex	480	2 GB	IB	SGE
Hotel	FutureGrid	iDataPlex	256	4 GB	IB	Torque
India	FutureGrid	iDataPlex	256	3 GB	IB	Torque
Sierra	FutureGrid	iDataPlex	256	4 GB	IB	Torque
Carver	DOE/NERSC	iDataPlex	512	4 GB	IB	Torque
Hermes	UCLM, Spain	Beowulf	256	4 GB	10 GbE	SGE
Libra	IHPC, Singapore	Beowulf	128	8 GB	1 GbE	N/A

Note: † – peak number of cores available to the experiment. ‡ – memory per core.

Table 2: Capability of each resource.

Name	Cores per task	Accepted tasks
Excalibur	1024	small/medium/large
Snake	64	small/medium
Stampede	128	small/medium/large
Lonestar	120	small/medium/large
Hotel	128	small/medium/large
India	128	small/medium/large
Sierra	128	small/medium/large
Carver	256	small/medium
Hermes	128	small/medium/large
Libra	128	small/medium

Figure 5: Summary of the experiment. Top: Utilization of different computational resources. Line thickness is proportional to the number of tasks being executed at given point of time. Gaps correspond to idle time, e.g. due to machine maintenance. Bottom: The total number of running tasks at given point of time.

Figure 6: Throughput and queue waiting time. Top: Dissection of throughput measured as the number of tasks completed per hour. Different colors represent component throughput of different machines. Middle: Throughput contribution by different task classes. Bottom: Queue waiting time on selected resources. Please view in color.

Figure 6 outlines how the computational throughput, measured as the number of tasks completed per hour, was shaped by different computational resources. Here, several interesting observations can be made. First, no single resource dominated the execution. Although *Stam-*

pede, the most powerful machine among all federated, provided a brief performance burst during the first two days, it was unable to deliver a sustained throughput. In fact, tasks on this machine were submitted to the "development" queue that limits the number of processors used by a job, but offers relatively high turnover rate. Yet, even this queue got saturated after the first day of execution, which caused a sudden drop in the throughput. This pattern can be observed on other systems as well (e.g., see *Lonestar* and *Carver*), and it confirms our earlier observation that no single system can offer a sufficient throughput. Another observation is related to how the throughput was distributed in time. The peak was achieved close to the end of the experiment, even though after twelfth day *Excalibur* was running at half its initial capacity (see Figure 5). This can be explained by the fact that the majority of tasks executing towards the end were small tasks. Consequently, all available resources were able to participate in execution, and short runtimes increased the overall throughput.

The last important element of the experiment was data management. Technically, the input data consisted of two components: already mentioned finite element mesh database, and a 4-tuple describing simulation parameters. The database was tightly integrated with the simulation software, and hence deployed together with the software. As a result, no special mechanisms were required to handle the input. The output data consisted of simulation results and several small auxiliary files. The size of the output varied between simulations. The data was compressed *in situ* and on-the-fly during the experiment, and then transferred using the `RSYNC` protocol to the central repository for a subsequent analysis.

The presented results clearly demonstrate feasibility and capability of our federation model. In our experiment, using just basic `SSH` access to several globally distributed and heterogeneous resources, we were able to solve a large-scale computational engineering problem, within just two weeks. What is important, this result was achieved in a few simple steps executed completely in a user-space. By providing a simple master/worker code we gained access to a unified and fault-tolerant platform able to sustain computational throughput.

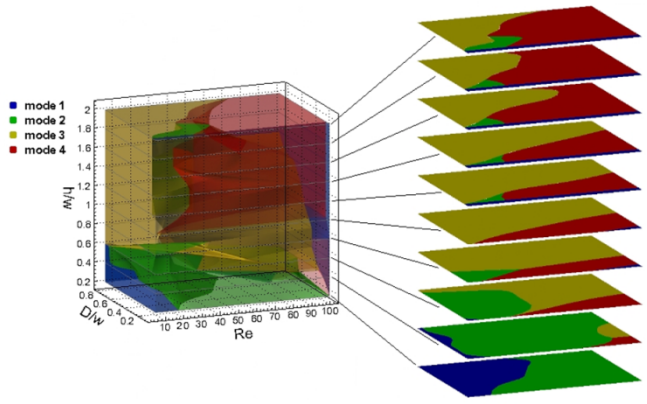

Figure 7: The phase diagram showing how different flow modes are distributed in the parameter space. Here, pillar offset is 0, D is a pillar diameter, h is a channel height, w is channel width, and Re is Reynolds number. Please view in color.

4.3 Science Outcomes

Thanks to the above experiment we obtained the most comprehensive data on the effect of pillars on microfluid channel flow. Although we are still in the process of analyzing this massive output, we already gained several interesting insights regarding fundamental features of the flow. Figure 7 shows how different flow modes are distributed in the parameter space. Here, each mode corresponds to one or two vortices generated, as proposed in [1]. When describing the problem we hinted that by arranging pillars into a specific sequence it is possible to perform basic flow transformations. Thanks to the library of flow configurations that we generated in this experiment, we can now investigate the inverse problem and, for example, ask questions about the optimal pillar arrangement to achieve a desired flow output. The implications of such capability are far-reaching, with potential applications in medical diagnostics and smart materials engineering.

5 Final Remarks

Providing an easy access to large-scale computational resources is one of the most important challenges facing the entire HPC community. In this report we described a federation model that addresses this challenge by empowering average user with computational capabilities typically reserved for high-profile computational problems. Using the federation we were able to solve the actual problem of constructing phase diagram of possible flow behaviors in the microscale devices.

Research work presented in this report was facilitated by the UberCloud experiment [11]. More information about this work can be found at `http://nsfcac.rutgers.edu/CometCloud/uff/`.

Acknowledgment

This work is supported in part by the National Science Foundation (NSF) via grants number IIP-0758566 and DMS-0835436 (RDI^2 group), and CAREER-1149365 and PHY-0941576 (Iowa State group). This project used resources provided by: the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the NSF grant number OCI-1053575, FutureGrid, which is supported in part by the NSF grant number OCI- 0910812, and the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy (DOE) under the contract number DE-AC02-05CH11231. The authors would like to thank the SciCom research group at the Universidad de Castillala Mancha, Spain (UCLM) for providing access to Hermes, and Distributed Computing research group at the Institute of High Performance Computing, Singapore (IHPC) for providing access to Libra. The authors would like to acknowledge the Consorzio Interuniversitario del Nord est Italiano Per il Calcolo Automatico, Italy (CINECA), Leibniz-Rechenzentrum, Germany (LRZ), Centro de Supercomputacion de Galicia, Spain (CESGA), and the National Institute for Computational Sciences (NICS) for willing to share their computational resources. The authors would like to thank Dr. Olga Wodo for discussion and help with development of the simulation software, and Dr. Dino

DiCarlo for discussions about the problem definition. The authors express gratitude to all administrators of systems used in this experiment, especially to Prentice Bisbal from RDI^2 and Koji Tanaka from FutureGrid, for their efforts to minimize downtime of computational resources, and a general support. The authors are grateful to Wolfgang Gentzsch and Burak Yenier for their overall support.

References

[1] H. Amini, E. Sollier, M. Masaeli, et al. Engineering fluid flow using sequenced microstructures. *Nature Communications*, 2013.

[2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[3] Y. Gambin, V. VanDelinder, F. A., et al. Visualizing a one-way protein encounter complex by ultrafast single-molecule mixing. *Nature Methods*, 8:239–241, 2011.

[4] H. Lee, J. Kim, H. Kim, et al. Colour-barcoded magnetic microparticles for multiplexed bioassays. *Nature Materials*, 9:745–749, 2010.

[5] Z. Li and M. Parashar. A computational infrastructure for grid-based asynchronous parallel applications. In *Proc. Int. Symp. on High Performance Distributed Computing (HPDC)*, pages 229–230, 2007.

[6] C. Schmidt. Flexible information discovery in decentralized distributed systems. In *Proc. Int. Symp. on High Performance Distributed Computing (HPDC)*, 2003.

[7] C. Schmidt and M. Parashar. Squid: Enabling search in DHT-based systems. *Journal of Parallel and Distributed Computing*, (7):962–975, 2008.

[8] I. Stoica, R. Morris, D. Liben-Nowell, et al. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 149–160, 2001.

[9] J. Wang, Y. Zhan, V. Ugaz, et al. Vortex-assisted DNA delivery. *Lab on a Chip*, 10:2057–2061, 2010.

[10] CometCloud Project. http://www.cometcloud.org/.

[11] The Uber-Cloud Experiment. http://www.hpcexperiment.com/.

[12] XSEDE Project. https://www.xsede.org/.