Published in final edited form as:

Proc IEEE/ACM Int Conf Grid Computing. 2009 October 13; 2009: 34–41. doi:10.1109/GRID.2009.5353065.

Investigating the Use of Cloudbursts for High-Throughput Medical Image Registration

Hyunjoo Kim and Manish Parashar

NSF Center for Autonomic Computing Department of Electrical & Computer Engineering Rutgers, The State University of New Jersey 94 Brett Road, Piscataway, NJ, USA {hyunjoo@rutgers.edu, parashar@rutgers.edu}

David J. Foran and Lin Yang

The Center for Biomedical Imaging and Informatics Department of Radiology UMDNJ-Robert Wood Johnson Medical School 675 Hoes Ln West, Piscataway, NJ, USA {foran@umdnj.edu, yangli@umdnj.edu}

Abstract

This paper investigates the use of clouds and autonomic cloudbursting to support a medical image registration. The goal is to enable a virtual computational cloud that integrates local computational environments and public cloud services on-the-fly, and support image registration requests from different distributed researcher groups with varied computational requirements and QoS constraints. The virtual cloud essentially implements shared and coordinated task-spaces, which coordinates the scheduling of jobs submitted by a dynamic set of research groups to their local job queues. A policy-driven scheduling agent uses the QoS constraints along with performance history and the state of the resources to determine the appropriate size and mix of the public and private cloud resource that should be allocated to a specific request. The virtual computational cloud and the medical image registration service have been developed using the CometCloud engine and have been deployed on a combination of private clouds at Rutgers University and the Cancer Institute of New Jersey and Amazon EC2. An experimental evaluation is presented and demonstrates the effectiveness of autonomic cloudbursts and policy-based autonomic scheduling for this application.

Keywords

Cloud computing; Autonomic computing; Autonomic cloudbursts; Medical image registration

I. Introduction

Emerging cloud services represent a new paradigm for computing based on-demand access to computing utilities, an abstraction of unlimited computing resources, and a usage-based payment model made users essentially “rent” virtual resources and pay for what they use. Underlying these cloud services are consolidated and virtualize data centers that provide virtual machine (VM) containers hosting computation and data intensive applications from large numbers of distributed users. Furthermore, integrating these public cloud platforms with exiting computational Grids provides opportunities for on-demand scale-up and scale-down, i.e., cloudbursts. While such a paradigm can potentially have a significant impact on a wide range of application domains, various aspects of the existing applications and of current cloud infrastructure make the transition to clouds challenging.

The overall goal of the research is to investigate the use of clouds and autonomic cloudbursts for real-world scientific and engineering applications and specifically for high-throughput medical image registration. In this application, a set of image registration methods are used by different (geographically distributed) research groups to process their locally stored data. The set of images will be typically acquired at different time, or from different perspectives, and will be in different coordinate systems. It is therefore critical to align those images into the same coordinate system before applying any image analysis. Image registration is the process to determine the linear/nonlinear mapping T between two images of the same object or similar objects.

While conceptually this type of application is seemingly naturally suited for clouds, various aspects of the application and of current cloud infrastructure make its deployment challenging. For example, the workloads associated with the application can be quite dynamic, both in terms of the number of images processed as well as computation requirements of each image. Furthermore, each research group may have very different and dynamic QoS requirements, for example, one group may require high throughput while another may be constrained by a budget, and a third may have to balance both throughput and budget. It is also the case that researchers may want to use resources in their private cloud (or datacenter or Grid) first before bursting out onto a public cloud and may have preference for a particular cloud.

Current cloud offerings also have several limitations, especially in terms of being able to integrate private and public clouds as well as using multiple public clouds. Currently, an application must typically select a specific cloud when it is deployed. While there are several cloud offerings currently available (e.g., Amazon EC2 [1], Google App Engine [2], Microsoft Azure [3], and GoGrid [4], etc.), each with its own pricing policy, quality of service, and Service Level Agreements (SLA), users usually must select a particular service to run the application based on the application type, resource requirements and budgets. Furthermore, the performance of a cloud service can vary based on load, failures, network conditions, etc., resulting in different qualities of service to the application.

In this research we envision a virtual computational cloud that integrates local computational environments (datacenters, Grids) and public cloud services on-the-fly, enables autonomic cloudbursts, and supports image registration requests from different distributed researcher groups with varied computational requirements and QoS constraints. The virtual cloud essentially implements shared and coordinated task-spaces, which coordinates the scheduling of jobs submitted by a dynamic set of research group to their local job queues. The jobs are then scheduled onto sets of workers that are dynamically provisioned on available private and/or public cloud resources based on their QoS constraints such as cost or performance. A policy-driven scheduling agent uses the QoS constraints along with performance history and the state of the resources to determine the appropriate size and mix of the public and private cloud resources that should be allocated to a specific application request, and policies can be defined and changed on-the-fly.

This paper has two specific objectives: (1) to demonstrate how such a virtual computational cloud can be effectively enabled by the CometCloud engine [5] and its support for autonomic cloud-bridging and cloudbursts; and (2) to experimentally demonstrate how the medical image registration can benefit from this cloud infrastructure. The virtual computational cloud and the medical image registration service have been developed using the CometCloud engine [5] and have been deployed on a combination of private clouds at Rutgers University, the Cancer Institute of New Jersey and Amazon EC2. An experiment evaluation is presented and demonstrates the effectiveness of autonomic cloudbursts and policy-based autonomic scheduling for this application.

The rest of this paper is organized as follows. Section II presents an overview of medical image registration, Section III presents the CometCloud framework for autonomic cloud bursts, and Section IV describes medical image registration using CometCloud. Experiment is presented in Section V and Section VI concludes this paper.

II. Application Description: Medical Image Registration

Nonlinear image registration [6] is the process to determine the mapping T between two images of the same object or similar objects acquired at different time, in different position or using different acquisition parameters or modalities. Both intensity/area based and landmark based methods have been reported to be effective in handling various registration tasks. Hybrid methods that integrate both techniques have demonstrated advantages in the literature [7].

In general, intensity/area based methods are widely accepted for fully automatic registration. But landmark based methods, though also commonly used, sometimes still rely on human intervention in selecting landmark points and/or performing point matching. Point matching in medical images is particularly challenging due to the variability in image acquisition and anatomical structures.

We developed alternative landmark point detection and matching method as a part of our hybrid image registration algorithm for both 2D and 3D images. The algorithm starts with automatic detection of a set of landmarks in both fixed and moving images, followed by a coarse to fine estimation of the nonlinear mapping using the landmarks. For 2D images, multiple resolution oriental histograms and intensity template are combined to obtain fast affine invariant local descriptor of the detected landmarks. For 3D volumes, considering both speed and accuracy, the global registration is first applied to pre-align two 3D images. Intensity template matching is further used to obtain the point correspondence between landmarks in the fixed and moving images. Because there is a large portion of outliers in the initial landmark correspondence, a robust estimator, RANSAC [8], is applied to reject outliers. The final refined inliers are used to robustly estimate a Thin Spline Transform (TPS) [9] to complete the final nonlinear registration. The proposed algorithm can handle much larger transformation and deformation compared with common image registration methods such as finite element method (FEM) or BSpline fitting, while still provide good registration results. The flowchart of the hybrid image registration algorithm is shown in Figure 1.

A. Automatic Landmark Detection and Matching

The automatic landmark detection is the procedure used to accurately detect the prominent and salient points in the image. Harris corner detector was applied to find the points with the large gradients in both directions (x and y for 2D images). The original computation in the Harris corner detector involves the computation of eigenvalues. Instead, the determinant and trace are used in our algorithm to find the corners using $F = \det(A) - \alpha \text{trace}(A)$ where α is chosen as 0.1.

After we detect the landmarks, we can extract features from the neighborhood of each landmark. The local orientation histograms are used as the features for landmark matching. The image is first convolved with the orientation filters. The filtering response in the neighborhood around the landmarks is computed to form the orientation histogram. Let $G_x(i, j)$ and $G_y(i, j)$ represent the gradients on pixel $p(i, j)$ along x and y direction, respectively. The orientation histogram h_k is defined as

$$h_k(i, j) = \begin{cases} C(i, j), & \theta(i, j) \in \text{bin}_k \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where

$$C(i, j) = \sqrt{G_x(i, j)^2 + G_y(i, j)^2} \quad (2)$$

and

$$\theta(i, j) = \arctan\left(\frac{G_y(i, j)}{G_x(i, j)}\right). \quad (3)$$

The orientation histogram encodes the directions of the edges at each landmark point. It is proven to be an effective feature descriptor when the training samples were small [10]. In order to achieve robust matching of the landmarks, the extensive searching in the image space is required. This step is time consuming and often creates the bottleneck for the landmark based image registration algorithm.

B. Robust Estimation and Nonlinear Image Registration

Because the original matching landmark sets contain missing landmarks. RANdom SAMple Consensus (RANSAC) [8] is used to reject outliers and robustly estimate the transformation. The algorithm is listed in Figure 2. The RANSAC robust estimator randomly selects the minimal subset of the landmarks to fit the model. Measured by a cost function, the points within a small distance are considered as a consensus set. The size of the consensus set is called the model support M . The algorithm is repeated multiple times and the model with largest support is called the robust fit. In Figure 3 we show the results of applying robust estimation to reject the outliers in the original matching landmarks. The Harris corner detector detected 32 landmark pairs in Figure 3(b). Based on the assumption of an affine transformation, the RANSAC found 8 inliers (shown in Figure 3(c)) and the rest 24 matching landmarks are rejected as outliers under the assumption for an Affine transformation.

The thin plate spline transform (TPS) is used to estimate the nonlinear transformation between the fixed and moving image based on the robust landmark correspondence. The TPS transformation T is calculated by minimizing the binding energy E_{TPS}

$$E_{TPS} = \sum_i \|T(w_i) - v_i\|^2 + \lambda I_f \quad (4)$$

where

$$I_f = \iint \left[\left(\frac{\partial^2 f}{\partial x^2} \right)^2 + 2 \left(\frac{\partial^2 f}{\partial x \partial y} \right)^2 + \left(\frac{\partial^2 f}{\partial y^2} \right)^2 \right] dx dy \quad (5)$$

with w_i and v_i denote the landmarks on the fixed and moving image, respectively.

TPS can provide a smooth matching function for each point in both images. The resulting nonlinear transformation is applied to map the moving image to the fixed image. For more details, we refer the readers to [9].

C. Performance and Speed Consideration

Some of experimental results for the registration algorithm are shown in Figure 4. The first row represents the fixed images and the second row is the moving images. The last row represents the results after image registration. The more similar the result image to the fixed image, the better the result would be. The rat lung image pair which represents a full respiration is shown in Figure 4(a). The Lenna image pair which contains a nonlinear transformation is shown in Figure 4(b). Two CT head image registration tests are shown in Figure 4(c). In Figure 4 (d) and (e), we show the image registration results using two misaligned pathology imaged specimens of human's breast tissue. In this set of experiments, we show that our method is generic and can be applied to different types of applications.

The algorithm can provide good registration results, but is also computationally expensive. While research effort continues to explore methods and strategies for more efficient and rapidly converging computational methods, increasing attention has been given to hardware architecture-based parallelization and optimization algorithms for the emerging high performance architectures, such as cluster and grid computing, advanced graphical processing units (GPU) and multicore computers. In [11] a parallel implementation of multimodal registration is reported for image guided neurosurgery using distributed and grid computing. The registration time was improved dramatically to near real-time. In [12] a distributed computation framework is developed, which allowed the parallelization of registration metrics and image segmentation/visualization algorithms while maximizing the performance and portability. In the following section, we will present the cloud computing framework which takes advantage of the CometCloud computing environment.

III. An Overview of CometCloud

CometCloud is an autonomic computing engine for cloud and Grid environments. It is based on the Comet [13] decentralized coordination substrate, and supports highly heterogeneous and dynamic cloud/Grid infrastructures, integration of public/private clouds and autonomic cloudbursts. A schematic overview of the architecture is presented in Figure 5.

Conceptually, CometCloud is composed of a programming layer, service layer, and infrastructure layer. The infrastructure layer uses the Chord self-organizing overlay [14], and the Squid [15] information discovery and content-based routing substrate build on top of Chord. The routing engine supports flexible content-based 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 will be located. Note that resources (nodes) in the overlay have different roles (and accordingly, access privileges) based on their credentials and capabilities. This layer also provides replication and load balancing services, and handles dynamic joins and leaves of nodes as well as node failures. Every node maintains a replica of its successor node's state, and reflects changes to this replica whenever its successor notifies it of changes. If a node fails, the predecessor node merges the failed node's replica into its own state and then makes a new replica of its new successor. If a new node joins, the joining node's predecessor updates its replica to reflect the joining node's state, and the successor gives its state information to the joining node. Note that, to maintain load balancing, load should be redistributed among the nodes whenever a node joins and leaves.

The service layer provides a range of services to support autonomies at the programming and application level. This layer supports a Linda-like [16] tuple space coordination model, and provides a virtual shared-space abstraction as well as associative access primitives. Dynamically constructed transient spaces are also supported to allow applications to explicitly exploit context locality to improve system performance. Asynchronous (publish/subscribe) messaging and event services are also provided by this layer. Finally, online clustering services support autonomic management and enable self-monitoring and control. Events describing the status or behavior of system components are clustered and the clustering is used to detect anomalous behaviors.

The programming layer provides the basic framework for application development and management. It supports a range of paradigms including the master/worker/BOT. Masters generate tasks and workers consume them. Masters and workers can communicate via virtual shared space or using a direct connection. Scheduling and monitoring of tasks are supported by the application framework. The task consistency service handles lost/failed tasks. Even though replication is provided by the infrastructure layer, a task may be lost due to, for example, network congestion. In this case, since there is no failure, infrastructure level replication may not be able to handle it. These cases can be handled by the programming layer (e.g., the master), for example, by waiting for the result of each task for a pre-defined time interval, and if it does not receive the result back, regenerating the lost task. If the master receives duplicate results for a task, it selects one (the first) and ignores other (subsequent) results. Other supported paradigms include workflow-based applications as well as Mapreduce [17] (and Hadoop [18]).

A. Autonomic Cloud-bridging & Cloudbursts in CometCloud

The goal of autonomic cloudbursts is to seamlessly (and securely) bridge private enterprise clouds and datacenters with public utility clouds on-demand, to provide an abstraction of resizable computing capacity that is driven by user-defined high-level policies. It enables the dynamic deployment of application components, which typically run on internal organizational compute resources, onto a public cloud (i.e., cloudburst) to address dynamic workloads, spikes in demands, economic/budgetary issues, and other extreme requirements. Furthermore, given the increasing application and infrastructure scales, as well as their cooling, operation and management costs, typical over-provisioning strategies are no longer feasible. Autonomic cloudbursts can leverage utility clouds to provide on-demand scale-out and scale-in capabilities based on a range of metrics.

The overall support for autonomic cloudbursts in CometCloud is presented in Figure 6. CometCloud considers three types of clouds based on perceived security/trust and assigns capabilities accordingly. The first is a highly trusted, robust and secure cloud, usually composed of trusted/secure nodes within an enterprise, which is typically used to host masters and other key (management, scheduling, monitoring) roles. These nodes are also used to store state. In most applications, the privacy and integrity of critical data must be maintained, and as a result, tasks involving critical data should be limited to cloud nodes that have required credentials. The second type of cloud is one composed of nodes with such credentials, i.e., the cloud of secure workers. A privileged Comet space may span these two clouds and may contain critical data, tasks and other aspects of the application-logic/workflow. The final type of a cloud consists of casual workers. These workers are not part of the space but can access the space through the proxy to obtain (possibly encrypted) work units as long as they present required credentials and these credentials also define the nature of the access and type of data that can be accessed. Note that while nodes can be added or deleted from any of these clouds, autonomic cloudbursts primarily target worker nodes, and specifically worker nodes that do not host the Comet space as they are less expensive to add and delete.

IV. Cloud-based Medical Image Registration Using CometCloud

A. Application Scenario

An overview of the operation of the CometCloud-based medical image recognition application scenario is presented in Figure 7. In this application scenario, there are multiple (possibly distributed) job queues from where users insert image registration requests to the CometCloud. Each of these entry points represents a research site in research collaboration, and maintains its own storage where medical images are stored. Each site generates its own requests with its own policies and QoS constraints. Note that a site can join the collaboration and CometCloud at anytime (provided it has the right credentials) and can submit requests. The requests (tasks) generated by the different sites are logged in the CometCloud virtual shared space that spans master nodes at each of the sites. These tasks are then consumed by workers, which may run on local computational nodes at the site, a shared datacenter or on a public cloud infrastructure. These workers can access the space using appropriate credentials, access authorized tasks (i.e., image registration request) and return results back to the appropriate master indicated in the task itself.

B. Autonomic Task Scheduling in CometCloud

The scheduling of tasks onto cloud resources for each site is managed by a scheduling agent, which uses policies and QoS constraints to determine the number, type/location, mix of workers to be used. The scheduling agent periodically monitors and evaluates the performance of cloud/datacenter resources and determines if the set of workers used should grow/shrink or change to ensure that the policies and QoS constraints are satisfied, for example, application throughput maximized while maintaining costs within a prescribed budget. Policies can be changed at anytime. The scheduling agent consisting of three component, viz., *Monitor*, *Cost analyzer* and *Cloudbursts manager*, which are described below.

Monitor—The monitor component collects and maintains statistics about scheduled tasks and returned results. This includes task information such as the data size and computation times, worker information such as its type and location, as well as aggregate information for each cloud/datacenter. The monitor obtains this information from the master.

Cost analyzer—The cost analyzer periodically evaluates the performance of each cloud and datacenter's, based on information gathered by the monitor, including the amount of computation performed and data transferred. It then uses this information along with the pricing models/policies for each resource set (e.g., cost for computing, data storage, data transfer, etc.) to estimate the cost for executing a task on each of these available resources. Note that in case of local computational resource a simple cost model based on operational and maintenance costs is used [19].

Cloudbursts manager—The cloudbursts manager determines scheduling strategies based on cost profiles it obtains from the cost analyzer component and the specified policies and constraints. Decision include growing, shrinking or changing (type and mix) the set of resources used. In this paper, we specifically experiment with two types of policies: *Time-based* and *Budget-based*. The time-based policy essentially dictates that application request should be completed as soon as possible, and assuming an adequate budget, the maximum required workers are allocated for the job. In the budget-based policy, a budget is enforced on the application, and workers must be allocated so that this budget is not violated. In this case the cloudbursts manager must adjust the number and mix (i.e., location and type) of workers so as to manage performance without violating the budget. To achieve this, it calculates the available budget and the remaining number of tasks during each scheduling

interval, and calculates the number of workers to be scheduled on each available resource type, going in order from high performance to low performance resources. Performance here refers to the number of tasks that can be processed by a node from the resource class during a scheduling period t . For example, even though a particular public cloud node configuration may have lower performance, the cloudbursts manager may schedule tasks due to budgetary constraints. However, if the budget allows, a maximum number of high performance nodes are allocated first. The number and mix of workers in each available cloud/datacenter is determined using the following formulation.

$$\begin{aligned} \sum_{i=1}^n C_i \times N_i &\leq B_r, \\ \sum_{i=1}^n N_i &= N_r \end{aligned} \quad (6)$$

where C_i is cost per task on a particular a node (on the cloud or datacenter) i , N_i is the number of tasks to be allocated to i , n is the number of available resource providers, i.e., public/private clouds or datacenters, N_r is the number of remaining tasks, and B_r is the remaining budget at the start of the scheduling iteration. If there were an unlimited number of nodes available across the different resource pools (i.e., clouds, datacenters), the scheduling agent would be able to allocate all nodes that were determined by Eq. 6, and all remaining tasks would be completed in next iteration. However, the number of available nodes across the resource pools will be limited. Assuming that the number of these available nodes is Max_i , the scheduling agent schedules $\min(N_i, Max_i)$ nodes and estimates the number of iterations it will take to complete remaining tasks as $N_i / \min(N_i, Max_i)$.

For example, assuming that we are using two clouds, A and B, the remaining number of tasks is 300 and the number of nodes on cloud A and B are 100 each. Let us assume that the scheduling agent decides to allocate 200 tasks to cloud A nodes and 100 tasks to cloud B nodes. If the scheduling agent can allocate 200 nodes on cloud A and 100 nodes on cloud B, then all 300 tasks will be completed in the next iteration. However since, the numbers of nodes is limited to 100 nodes on each cloud, the scheduling agent first allocates 100 nodes on cloud A. The estimated number of iterations to complete the 200 tasks allocated to cloud A is 2. As a result, even though up to 100 nodes can be allocated on cloud B, the agent allocates only 50 (=100 tasks/2 iterations) nodes, so that all tasks will be completed in next 2 iterations.

V. Experimental Evaluation

A. Experimental Environment

The virtual cloud environment used for the experiments consisted of two research sites located at Rutgers University and University of Medicine and Dentistry of New Jersey, one public cloud, i.e., Amazon Web Service (AWS) EC2 [1], and one private datacenter at Rutgers, i.e., TW. The two research sites hosted their own image servers and job queues, and workers running on EC2 or TW access these image servers to get the image described in the task assigned to them (see Figure 6). Each image server has 250 images resulting in a total of 500 tasks. Each image is two dimensional and its size is between 17KB and 65KB. The costs associated with running tasks on EC2 and TW nodes were computed based on costing models presented in [1] and [19] respectively. On EC2, we used standard small instances with a computing cost of \$0.10/hour, data transfer costs of \$0.10/GB for inward transfers and \$0.17/GB for outward transfers.

Costs for the TW datacenter included hardware investment, software, electricity etc., and were estimated based on the discussion in [19], which says that a datacenter costs \$120K/

lifecycle per rack and has a lifecycle of 10 years. Hence, we set the cost for TW to \$1.37/hour per rack. In the experiments we set the maximum number of available nodes to 25 for TW and 100 for EC2. Note that TW nodes outperform EC2 nodes, but are more expensive. We used budget-based policy for scheduling where the scheduling agent tries to complete tasks as soon as possible without violating the budget. We set the maximum available budget in the experiments to \$3 to complete all tasks. The motivation for this choice is as follows. If the available budget was sufficiently high, then all the available nodes on TW will be allocated, and tasks would be assigned until all the tasks were completed. If the budget is too small, the scheduling agent would not be able to complete all the tasks within the budget. Hence, we set the budget to an arbitrary value in between. Finally, the monitoring component of the scheduling agent evaluated the performance every 1 minute.

B. Results

The results from the experiments are plotted in Figure 8. Note that since the scheduling interval is 1 min, the X-axis corresponds to both time (in minutes) and the scheduling iteration number. Initially, the CometCloud scheduling agent does not know the cost of completing a task. Hence, it initially allocated 10 nodes each from TW and EC2. Figure 8 (a) shows the scheduled number of workers on TW and EC2 and (b) shows the average cost per task in each scheduling period for TW and EC2. In the beginning, since the budget is sufficient, the scheduling agent tries to allocate TW nodes even though they cost more than EC2 node. In the 2nd scheduling iteration, there are 460 tasks still remaining, and the agent attempts to allocate 180 TW nodes and 280 EC2 nodes to finish all tasks as soon as possible within the available budget. If TW and EC2 could provide the requested nodes, all the tasks would be completed by next iteration. However, since the maximum available TW node is only 25, it allocates these 25 TW nodes and estimates that a completion time of 7.2 iterations. The agent then decides on the number of EC2 workers to be used based on the estimated rounds.

In case of the EC2, it takes around 1 minutes to launch (from the start of virtual machine to ready state for consuming tasks), and as a result, by the 4th iteration the cost per task for EC2 increases. At this point, the scheduling agent decides to decrease the number of TW nodes, which are expensive, and instead, decides to increase the number of EC2 nodes using the available budget. By the 9th iteration, 22 tasks are still remaining. The scheduling agent now decides to release 78 EC2 nodes because they will not have jobs to execute. The reason why the remaining jobs have not completed at the 10th iteration (i.e., 10 minutes) even though there are 22 nodes still working is that there was an unexplainable decrease in EC2 performance during our experiments. The variations in the cost per task in Figure 8 (b) are because the task completions are not uniformly distributed across the time intervals. Since the cost per interval is fixed (defined by AWS) the cost per task varies, depending on the number of task completed in a particular time interval. Figure 8 (c) shows the used budget over time. It shows all the tasks were completed within the budget and took around 13 minutes.

Figure 9 shows a comparison of execution time and used budget with/without the CometCloud scheduling agent. In the case where only EC2 nodes are used, when the number of EC2 nodes is decreased from 100 to 50 and 25, the execution time increases and the budget used decreases as shown (a) and (b). Comparing the same number of EC2 and TW nodes (25 EC2 and 25 TW), the execution time for 25 TW nodes is approximately half that for 25 EC2 nodes, however the costs for 25 TW nodes is significantly more than that for 25 EC2 nodes. When the CometCloud autonomic scheduling agent is used, the execution time is close to that obtained using 25 TW nodes, but the cost is much smaller and the tasks complete within the budget. The reason why the execution time in this case is larger than that for 100 EC2 node case is as follows: the cost peaks at time = 11 mins as seen in Figure

8 (b), and this causes the autonomic scheduler to reduce the number of EC2 nodes to approximately 20 (see Figure 8 (a)), causing the execution time to increase. An interesting observation from the plots is that if you don't have any limits on the number of EC2 nodes used, then a better solution is to allocate as many EC2 nodes as you can. However, if you only have limited number of EC2 nodes and want to be guaranteed that your job is completed within a limited budget, then the autonomic scheduling approach achieves an acceptable tradeoff. Since different cloud service will have different performance and cost profiles, the scheduling agent will have to use historical data and more complex models to compute schedules, as we extend CometCloud to include other service providers.

VI. Conclusion

In this paper, we investigated the use of clouds and autonomic cloudbursting to support a medical image registration using the CometCloud framework. CometCloud enables a virtual computational cloud that integrates local computational environments and public cloud services on-the-fly, and supports image registration requests from different distributed researcher groups with varied computational requirements and QoS constraints. The policy-driven scheduling agent uses the QoS constraints along with performance history and the state of the resources to determine the appropriate size and mix of the public and private cloud resources that should be allocated to a specific request. Specifically, cost models and users budgets were used to define policies. The virtual cloud infrastructures and the cloud-based medical image registration were deployed on a combination of private clouds at Rutgers University, the Cancer Institute of New Jersey and Amazon EC2 and an initial experimental evaluation is presented. The results demonstrated the effectiveness of autonomic cloudbursts and policy-based autonomic scheduling for this application.

Acknowledgments

The research presented in this paper is supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826, and CNS 0723594, by Department of Energy via the grant number DE-FG02-06ER54857, by a grant from UT Battelle, and by an IBM Faculty Award, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University. Experiments on the Amazon Elastic Compute Cloud (EC2) were supported by a grant from Amazon Web Services. Additional support was provided by grants from the NIH through contract 5R01EB003587-04 from the National Institute of Biomedical Imaging and Bioengineering and contract 5R01LM009239-02 from the National Library of Medicine.

References

- [1]. Amazon elastic compute cloud. <http://aws.amazon.com/ec2/>
- [2]. Google app engine. <http://code.google.com/appengine/>
- [3]. Azure service platform. <http://www.microsoft.com/azure/>
- [4]. Gogrid. <http://www.gogrid.com>
- [5]. Kim, H.; Chaudhari, S.; Parashar, M.; Marty, C. Online risk analytics on the cloud. Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on; May 2009; p. 484-489.
- [6]. Vercauteren, T.; Pennec, X.; Perchant, A.; Ayache, N. Symmetric log-domain diffeomorphic registration: A demons-based approach. Proc. International Conference on Medical Image Computing and Computer Assisted Intervention; 2008. p. 754-761.
- [7]. Azar, A.; Xu, C.; Pennec, X.; Ayache, N. An interactive hybrid non-rigid registration framework for 3D medical images. Proc. International Symposium on Biomedical Imaging; 2006. p. 824-827.
- [8]. Fischler MA, Bolles RC. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Comm. of the ACM 1981;vol. 24:381-395.

- [9]. Chui H, Rangarajan A. A new point matching algorithm for non-rigid registration. *Computer Vision and Image Understanding* 2003;vol. 89(no. 2):114–141.
- [10]. Levi, K.; Weiss, Y. Learning object detection from a small number of examples: the importance of good features. *Proc. IEEE International Conference on Computer Vision and Pattern Recognition*; Washington, DC. 2004. p. 53-60.
- [11]. Chrisochoides N, Fedorov A, Kot A, Archip N, Black P, Clatz O, Golby A, Kikinis R, Warfield S. Toward realtime, image guided neurosurgery using distributed and grid computing. *ACM/IEEE Super Computing* 2006:37–48.
- [12]. Warfield SK, Jolesz F, Kikinis R. A high performance approach to the registration of medical imaging data. *Parallel Computing* 1998;vol. 24(no. 9):1345–1368.
- [13]. Li, Z.; Parashar, M. A computational infrastructure for grid-based asynchronous parallel applications. *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*; New York, NY, USA: ACM; 2007. p. 229-230.
- [14]. Stoica, I.; Morris, R.; Liben-Nowell, D.; Karger, DR.; Kaashoek, MF.; Dabek, F.; Balakrishnan, H. Chord: A scalable peer-to-peer lookup protocol for internet applications. *ACM SIGCOMM*; 2001. p. 149-160.
- [15]. Schmidt C, Parashar M. Squid: Enabling search in dhtbased systems. *J. Parallel Distrib. Comput* 2008;vol. 68(no. 7):962–975.
- [16]. Carriero N, Gelernter D. Linda in context. *Commun. ACM* 1989;vol. 32(no. 4):444–458.
- [17]. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Commun. ACM* 2008;vol. 51(no. 1):107–113.
- [18]. Hadoop. <http://hadoop.apache.org/core/>
- [19]. APC. Determining total cost of ownership for data center and network room infrastructure. 2002. White paper

Figure 1.
The hybrid image registration flowchart.

Input: Given N m -tuples of data, threshold τ , and probability p

Algorithm:

- $K = \infty, k = 0, p^* = [], M^* = 0$
- while $K > k$ do
 - Choose a sample set S from N with size m
 - Compute the model parameter p from the sample set S
 - Count the number M of inliers using p and τ
 - if $M > M^*$
 - * $M^* = M$
 - * $p^* = p$
 - * $w = M/N$
 - * $K = \text{ceil} \left(\frac{\log(1-p)}{\log(1-w^m)} \right)$
 - * $k = k + 1$
 - end if
- end while

Output: parameter p^*

Figure 2.
The RANSAC algorithm.

Figure 3.
Apply the robust estimation to find the robust landmark correspondence. (a)The original fixed and moving image. (b)The original pairs of matching points. (c)The robust matching points after rejecting the outliers.

Figure 4.
 The image registration examples. (a) The Rat lung image transformation recovery example.
 (b) The Lenna image transformation recovery example. (c) The human head image
 transformation recovery example. (d) and (e) The tissue microarray image transformation
 recovery example.

Figure 5.
The CometCloud architecture.

Figure 6.
Support for autonomic cloud bursts in CometCloud.

Figure 7.
Overview of the medical image registration application scenario using CometCloud.

Figure 8.

Experimental evaluation of medical image registration using CometCloud - Results using the scheduling agent.

(a) Execution time varying the number of nodes for EC2 and TW

(b) Used budget over time varying the number of nodes for EC2 and TW

Figure 9.

Experimental evaluation of medical image registration using CometCloud - Comparison of performance and costs with/without autonomic scheduling.