HTCaaS: Leveraging Distributed Supercomputing Infrastructures for Large-Scale Scientific Computing

Jik-Soo Kim, Seungwoo Rho, Seoyoung Kim, Sangwan Kim,
Seokkyoo Kim, and Soonwook Hwang
National Institute of Supercomputing and Networking
Korea Institute of Science and Technology Information
Daejeon, Republic of Korea
{jiksoo.kim,seungwoo0926,sssyyy77,sangwan,anemone,hwang}@kisti.re.kr

ABSTRACT

In this paper, we present the *HTCaaS* (High-Throughput Computing as a Service) which aims to provide researchers with ease of exploring large-scale and complex scientific problems by leveraging national supercomputing infrastructures in Korea. HTCaaS allows users to efficiently submit a large number of jobs *at once* by effectively managing and exploiting of all available computing resources. HTCaaS exploits a synthesis of well known techniques and its own intelligent scheduling algorithm to effectively support multiple users independently submitting large numbers of tasks to a collection of geographically distributed computing resources.

Throughout our micro-benchmark and protein docking experiments, we show that our HTCaaS can provide a single efficient job management system that can support the most challenging scientific applications.

Keywords

High-Throughput Computing, Many-Task Computing, HTCaaS, Multi-level Scheduling, Dynamic Fairnes

1. INTRODUCTION

Computing paradigms such as High-Throughput Computing or Volunteer Computing [1] mainly target compute-intensive, independent, and long-running applications consisting of many loosely-coupled tasks. Middleware systems such as Condor [11] or BOINC [1] have successfully achieved a tremendous computing power by harnessing a large number of computing resources consisting of either clusters of workstations or desktop machines over the Internet. However, recent emerging applications requiring millions or even billions of tasks to be processed with relatively short per task execution times have led the traditional HTC to expand into *Many-Task Computing* (MTC) [8]. These applications from a wide range of scientific domains (e.g., astronomy, physics, pharmaceuticals, chemistry, etc.) often require a very large number of tasks (from tens of thousands to billions of tasks), and have a large variance of task execution times (from hundreds of milliseconds to hours). This makes the existing middleware systems difficult to support challenging scientific applications due to lack of enough resources support, inefficiencies in task dispatching, unreliable and high-latency interconnects.

Therefore, to effectively support complex and demanding scientific applications consisting of many tasks, the goals of middleware systems for HTC/MTC applications must include the following:

- *Ease of Use*: User overhead for handling a large amount of jobs and computing resources should be minimized
- *Efficient Task Dispatching*: The overhead of task dispatching should be low enough to support a very large number of tasks
- *Adaptiveness*: The system should be able to adjust acquired resources according to changing load distribution due to heterogeneity in task execution times and computing resource capabilities
- *Fairness*: The system should be able to ensure fairness among multiple users submitting various numbers of tasks independently in order to reduce per user response time and improve the overall system throughput
- *Reliability*: By employing job monitoring technique, failed or suspended tasks (due to either job itself or computing resource) should be automatically resubmitted and managed.
- *Resource Integration*: The system should be able to effectively integrate as many computing resources as possible.

In this paper, we present the *HTCaaS* (High-Throughput Computing as a Service) which aims to provide researchers with ease of exploring large-scale and complex HTC/MTC problems by leveraging national PLSI Supercomputers [6] in Korea. HTCaaS allows users to efficiently submit a large number of jobs *at once* by effectively managing and exploiting of all available computing resources. HTCaaS exploits a synthesis of well known techniques and its own intelligent scheduling algorithm to effectively support multiple users with large numbers of tasks.

HTCaaS is currently running as a pilot service on top of PLSI computing resources and supporting a number of scientific applications from pharmaceutical domain and high-energy physics. The main contribution of our work for the research community is the evaluation of how different techniques can be effectively integrated to provide a single efficient job management system that can support challenging scientific applications.

2. HTCAAS: HIGH-THROUGHPUT COMPUTING AS A SERVICE

In this section, we introduce architectural details of HTCaaS and technologies effectively integrated to provide a single efficient job management system.

2.1 Architecture

Figure 1 shows the overall architecture of our HTCaaS system. *Account Manager* manages user information and provides integrated authentication and authorization mechanisms to access various computing infrastructures. *User Data Manager* is responsible for managing user input and output data (uploads & downloads) during the course of job executions. *Job Manager* mostly performs job life-cycle management, i.e., from the job submission to the completion. Job Manager maintains *separate* job queues *per* user, receives a *Meta-Job* (written in JSDL [5]) which can be composed of multiple tasks from a user, validates the Meta-Job, automatically splits the Meta-Job into multiple tasks, and controls the execution of each task. *Monitoring Manager* periodically checks job executions and active *agents* by interacting with *DB Manager*, and if needed it initiates failure recovery mechanisms for agents (tasks).

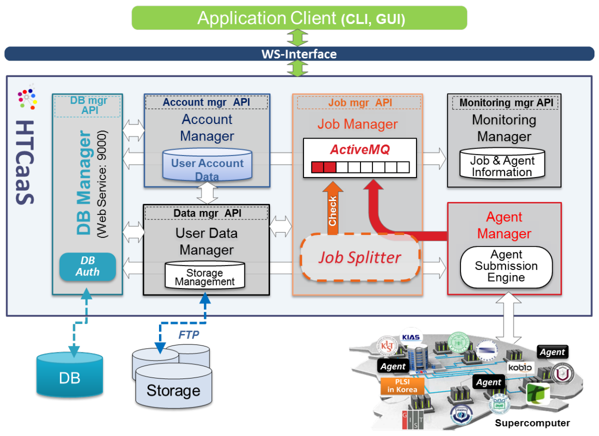


Figure 1: HTCaaS System Architecture

Since users may want to submit a large number of jobs by employing parameter sweeps or N-body calculations, HTCaaS introduces a concept of *Meta-Job* which specifies a higher-level job description based on the OGF JSDL standard [5]. Therefore, users of HTCaaS are able to easily submit and execute hundreds of thousands of jobs at once which can be simply expressed by a *single* JSDL script (*Ease of Use*). For those who are not familiar with XML style of scripting, we also provide an easy-to-use GUI tool which can automatically generate a JSDL script based on user's input so that it can be submitted into our system. Once a Meta-Job is submitted, HTCaaS automatically splits it into many tasks (by the Job Splitter component of the Job Manager in Figure 1) and inserts them into the job queue (implemented in ActiveMQ). While a Meta-Job (consisting of multiple tasks) is running, the Monitoring Manager periodically checks the status of agents and tasks. If some of agents or tasks fail, the Monitoring Manager informs the Agent Manager (or the Job Manager) to resubmit the failed agents (tasks) and manage them (addressing *Reliability*).

Once jobs are submitted into our HTCaaS, *agents* (implemented in Java) are dispatched from the Agent Manager and process tasks in geographically distributed supercomputing resources. HTCaaS employs agent-based *multi-level scheduling & streamlined task dispatching* (similar to the Falkon [9]) where existing batch schedulers (e.g., LoadLeveler in PLSI) are utilized for submitting agents and each agent bypasses the batch scheduler and directly contacts the job queue. Once deployed, each agent actively pulls the tasks, process them and record the statistics independently throughout the DB Manager which can be utilized for monitoring of tasks and agents by the Monitoring Manager. By employing agent-based multi-level scheduling mechanism consistently across heterogeneous computing resources, HTCaaS can effectively create a dedicated resource pool on the fly for fast dispatching of many tasks to circumvent the performance bottleneck of traditional batch schedulers (*Efficient Task Dispatching*, *Resource Integration*).

HTCaaS maintains separate job queues and agents per user and this queue management policy aims to achieve two practical goals: reducing complexities of accounting and scheduling. Since computing resources in PLSI are shared among multiple users, it is important to track and meter the usage of these resources *per* user. This is because like other supercomputing infrastructure management systems (such as XSEDE [15] in U.S. or PRACE [7] in Europe), PLSI also provides allotment of computing time or data space based on the certified user account. Also, by maintaining separate job queues and agents per user, we can simplify the problem of scheduling computing resources among multiple users in the system. By carefully calibrating the number of agents per user, we can address the problem of fair resource sharing among multiple users (as described in Section 2.2). Therefore, each agent actively pulls the tasks from its dedicated job queue which corresponds to a specific *user*, and if there are no more tasks to be processed, it automatically releases the acquired resources and exits.

2.2 User-level Scheduling and Dynamic Fairness

As we mentioned in Section 2.1, HTCaaS maintains separate job queues and agents per user which effectively formulates the problem of ensuring fairness among multiple users as an agent-dispatching problem which decides and allocates the number of agents per user. However, initial allocation of agents to a user will not be sufficient since the overall load distribution can *change* due to the heterogeneity in task execution times and resource capabilities. Therefore, we need a mechanism that can consistently monitor the overall load distribution changes and effectively adapt to the changing load. Since the number of acquired resources are exactly matching with the number of agents per user in the HTCaaS, we implement our dynamic load balancing mechanisms by adjusting the number of agents allocated to a user.

We address *Fairness* and *Adaptiveness* by implementing the *dynamic fair resource sharing* algorithm which divides all available computing resources fairly across all *demanding* users in the system (when the system is heavily loaded) and exploits dynamic adjustment of acquired resources as free computing resources become available (as the overall system becomes lightly loaded). For this purpose, we use a *resource allotment* function, `RA(U)` (as seen from Equation 1), where U represents a user in the current system.

$$RA(U) = min\left(NumTasks(U), \frac{AvailableCores}{\sum_{p \in DU} Weight(p)} * Weight(U)\right) \tag{1}$$

In Equation 1, $NumTasks(U)$ is the length of the job queue assigned to the user U and initially it is same as the number of tasks submitted by the user U. $AvailableCores$ is the number of computing resources that can be acquired by the HTCaaS (including free CPUs and already assigned ones) and we assume a one-to-one mapping between an agent and a CPU core in the system. DU is a set of demanding users and a demanding user is defined as a user who has *more* tasks to be processed in his queue than the number of agents allocated to him due to lack of available computing resources. $Weight(p)$ represents the weight of a user p which can consider many different factors such as the number of tasks submitted by the user p, task running time, priority, etc. In our current implementation, we consistently set this weight function for all users as one (i.e., $Weight(p) = 1$) so that $\sum_{p \in DU} Weight(p)$ becomes the number of demanding users in the system. This implies that our current HTCaaS system divides the entire available computing resources *equally* across all demanding users. However, we can easily apply various factors such as user priority, expected task execution times, system administrator's policy as inputs to the weight function which can realize a form of *weighted* fairness.

Note that in Equation 1, $AvailableCores$ and the set DU can *change* over time as a new demanding user arrives at the system or existing demanding users become *satisfied*. A demanding user U becomes satisfied if and only if $NumTasks(U)$ becomes 0 which means that she is already allocated enough number of agents to process all of her tasks. If a demanding user U becomes satisfied, he is no longer counted as a demanding user which results in the increase of $\frac{AvailableCores}{\sum_{p \in DU} Weight(p)}$. Therefore, remaining demanding users in the system can benefit from newly available computing resources. HTCaaS keeps monitoring overall system load and sizes of job queues and adjusts them if needed according to the resource allotment function.

For example, suppose there are total 100 CPU cores available in the system and, at time t_0, all of them are free. At time t_1, user A arrives at the system and submits 200 tasks which is larger than the total amount of computing resources. According to the Equation 1, $RA(A)$ becomes 100 since there is only one demanding user in the system so that A is allocated 100 agents (CPU cores). At time t_2, user B arrives at the system and submits total 50 tasks and user A's tasks are still being processed. Now, $R(A)$ becomes 50 and $R(B)$ becomes 50. Because $R(A)$ has changed from 100 down to 50 due to the arrival of a new demanding user B, user A should *release* 50 agents to ensure the fairness. The way of realizing this mechanism is that rather than preempting current running tasks, agents will be expired *after* finishing the processing of *current* running tasks. This is because preempting running tasks requires complex mechanisms for state storage and resuming the task. Let's assume that user A's 100 agents were processing the first 100 tasks of user A at the time of t_2, 50 agents of user A will exit after processing the current tasks which result in total 100 tasks are already completed at the time of releasement. Then, user B acquires all of freely available 50 computing resources and becomes the satisfied user (# of demanding users decreases into 1). Now, let's add one more user in the system. User C arrives at the system at time t_3 and submits 20 tasks. Currently the number of demanding users is two (user A and user C) so that $R(A)$ becomes 50 (user A still has 50 tasks to be processed) and $R(C)$ becomes 20. Therefore, until the user C acquires 20 agents (from computing resources released as the overall processing of user B's tasks are completed), user A cannot claim more than currently assigned 50 agents. If tasks of user B are completed then, user C will be assigned 20 agents (now user C becomes a satisfied user) and remaining 30 agents (CPU cores) can be acquired by user A (if needed).

Therefore, HTCaaS can effectively ensure fairness among multiple users submitting varying numbers of tasks to a collection of computing resources and dynamically adjust the fairness according to the load changes.

3. EVALUATION

In this section, we present experimental results from running our HTCaaS on top of PLSI Supercomputers based on micro-benchmark and a real application from the pharmaceutical domain.

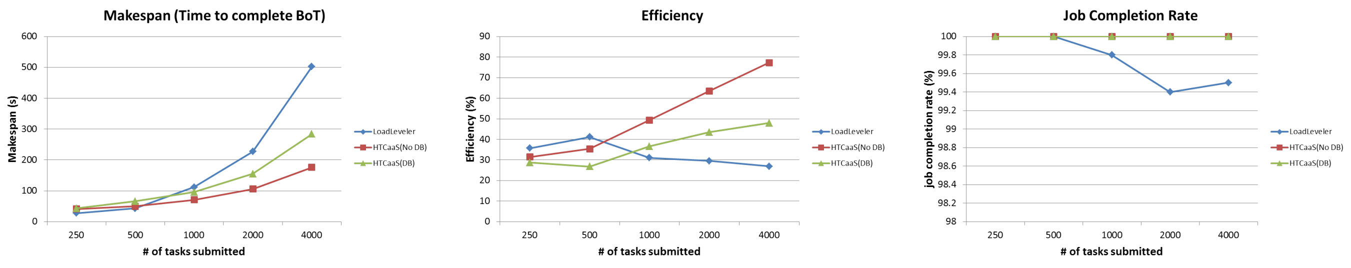
3.1 PLSI: Distributed Supercomputing Infrastructures in Korea

PLSI [6] stands for "Partnership & Leadership for the nationwide Supercomputing Infrastructure" and its main objective is to provide researchers with an integrated view of geographically distributed supercomputing infrastructures to solve complex and demanding scientific problems. PLSI is consisting of 10 supercomputing centers having 18 supercomputers connected via a dedicated 1Gbps network resulting in total 100TFLops of computing power (1,115 nodes with 8,508 CPU cores).

ORG	*SYSTEM*	*PROCESSOR*	*CORES*	OS
KIAS	helix (x86)	AMD Opteron 2GHz	128	CentOS 6.2
KIAS	gene (x86)	AMD Opteron 2GHz	128	CentOS 6.2
KOBIC	kobic (SUN)	AMD Opteron 2.1GHz	184	CentOS 5.4
KISTI	glory (SUN)	AMD Opteron 1.8GHz	514	CentOS 5.4

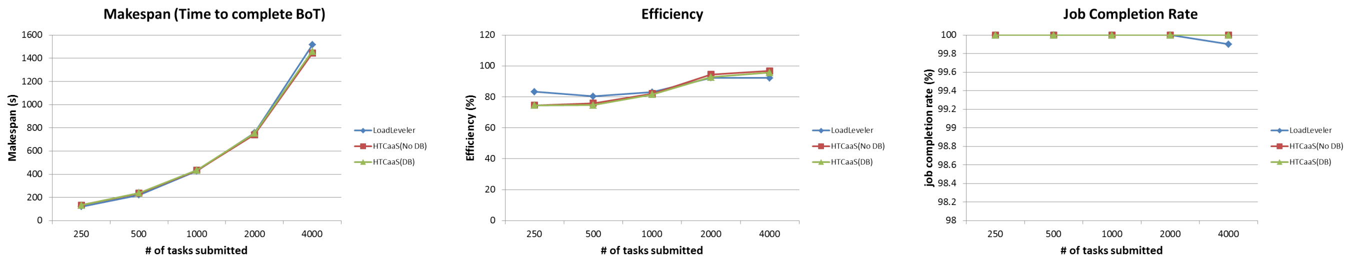
Table 1: PLSI Computing Resources leveraged by HTCaaS

PLSI provides a common software stack for accounting, monitoring, global scheduling (based on LoadLeveler) and a global shared storage system (based on GPFS [3]). Users of PLSI can utilize the LoadLeveler as a job submission system and exploit a total 400TB of global home/scratch directories mounted at every computing node as a shared storage system for input/output data and executables. The LoadLeveler scheduling system in the PLSI is configured as a *multi-cluster* environment consisting of two or more LoadLeveler clusters, grouped together through network connections that allow the clusters to share resources. Users of



(a) Makespan (b) Efficiency (c) Job Completion Rate

Figure 2: Micro-Benchmark Performance Results (Sleep 10)



(a) Makespan (b) Efficiency (c) Job Completion Rate

Figure 3: Micro-Benchmark Performance Results (Sleep 100)

PLSI can specify a list of multiple clusters in the job script and then, LoadLeveler is to decide which cluster is the best from the list of clusters, based on an administrator-defined metric (i.e., CLUSTER_METRIC). However, this approach has some drawbacks since even in the multi-cluster environment, LoadLeveler assigns all steps of a *multi-step* job (which corresponds to our Meta-Job consisting of multiple tasks) to the *same* cluster rather than intelligently partitioning overall tasks *across* multiple clusters (lack of *Resource Integration*).

Table 1 shows the list of computing resources currently connected with our HTCaaS system which consists of three different organizations having four different clusters. For now, HTCaaS is running as a pilot service on top of PLSI computing resources, however, as our system develop into a production-level service, we expect more computing resources will be integrated into the HTCaaS.

3.2 Micro-Benchmark Experiments

In this section, we present experimental results from performing micro-benchmark simulating a large number of short-running tasks (sleep 10) and relatively long-running tasks (sleep 100), however, both of tasks still fall inside the category of MTC in terms of task execution times [9]. In this experiments, we mainly used computing resources from the glory cluster in KISTI (Table 1) and at the time of our experiments, some other users were also participating in the cluster so that HTCaaS could use up to around 300 cores in the system.

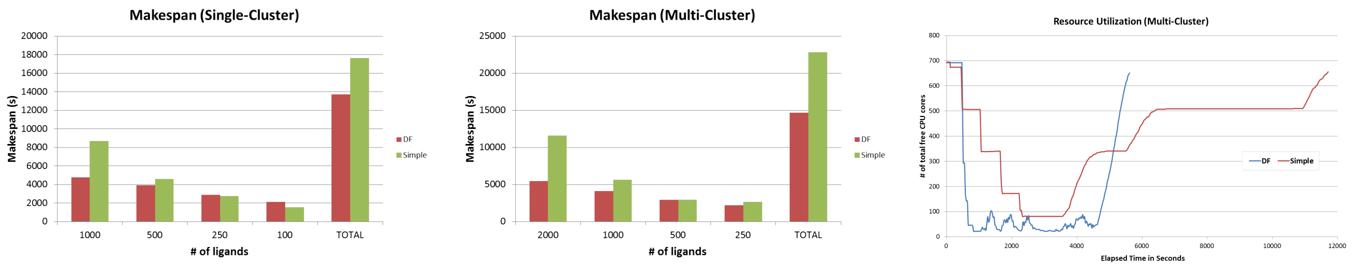
HTCaaS exploits the LoadLeveler as a first-level scheduler so that our comparison models include the LoadLeveler (labeled as **LoadLeveler** in the figures), HTCaaS with DB Manager connections from agents to update the status of tasks (**HTCaaS(DB)**), and HTCaaS without DB Manager interactions (**HTCaaS(No DB)**). DB Manager interactions play an important role in monitoring the progress of agents/tasks and dealing with failure recoveries, however, as the number of agents increases, the overhead of communicating with DB Manager can be substantial. To see the limits of our HTCaaS DB Manager, we tested HTCaaS without DB Manager connections and performance results were collected throughout post analysis of agent log files.

Our metrics are *Makespan* (time to complete a bag of tasks) and *Efficiency*. The metric Efficiency is calculated as following Equation 2:

$$Efficiency(NT) = \frac{\frac{NT*PTE}{NCU}}{Makespan(NT)} * 100 \qquad (2)$$

In Equation 2, NT means the number of tasks in the experiment (in our case, from 250 to 4000), PTE is the per task execution time (10 or 100 seconds), NCU denotes the number of cores used to complete the bag of tasks. Therefore, $\frac{NT*PTE}{NCU}$ means the *ideal* performance to process NT tasks with NCU computing resources without any dispatching overhead, i.e., perfect parallelism. Efficiency can show how much each job scheduling system can approach to the ideal parallelism with given number of tasks (with execution times) and computing resources.

Figure 2 and 3 show the performance results of LoadLeveler, HTCaaS(No DB) and HTCaaS(DB) in terms of makespan and efficiency. Note that a certain number of tasks (from 250 to 4000) are submitted *at once*. HTCaaS can submit a bag of tasks simultaneously by using the concept of the Meta-Job and to rapidly acquire computing resources, *multi-step* functionality of the LoadLeveler was exploited (first-level scheduler specific optimization). As we can see from Figure 2a and 2b, HTCaaS (DB & No DB) clearly outperforms the LoadLeveler in terms of makespan and efficiency as the number of tasks increases. More specifically, when the number of tasks becomes 4000, HTCaaS(No DB) achieves only 35% of the makespan compared to that of the LoadLeveler,



(a) Makespan in Single Cluster (b) Makespan in Multiple Clusters (c) Resource Utilization in Multiple Clusters

Figure 4: Autodock Experimental Results

while `HTCaaS(DB)` gets the 57% of that. Interestingly, the efficiency of the `HTCaaS` are still being improved as we increase the number of tasks submitted, while the `LoadLeveler` shows that it already hit the maximum efficiency with given number of computing resources (about 300) when the number of tasks becomes 500. As we increase the per task execution time from 10s to 100s, all of three different job scheduling mechanisms show competitive results, however, `HTCaaS(No DB)` and `HTCaaS(DB)` still can outperform the `LoadLeveler` especially when the number of tasks becomes large. This means that for relatively long-running tasks, overheads of task dispatching can be effectively countervailed.

Another interesting results from micro-benchmarks are shown in Figure 2c and Figure 3c. As we increase the number of tasks from 250 to 4000, with high probability some jobs are *held* during the course of task dispatching from the LoadLeveler. This problem happens more often when the per task execution time is shorter as we can see from Figure 2c. Once a task is held during the dispatching process, the user should manually restart or cancel it to make a progress (lack of *Reliability*). We attribute the potential main reason of this problem to multiple simultaneous I/O operations on the shared storage file system (GPFS as we described in Section 3.1). When we were performing our micro-benchmarks, we intentionally generated output files of running tasks to the shared storage system mounted at every computing node in the PLSI because this is a typical scenario where many ordinary users are utilizing home/scratch directories as storing their application executables and data.

HTCaaS can encounter similar problems while it is launching multiple agents through the LoadLeveler (due to either the shared storage file system or the batch scheduler itself), however, HTCaaS can automatically retry the submission of failed agents and manage them (addressing *Reliability*). Also, to circumvent the performance bottleneck of concurrent I/O operations on the shared storage file system, each agent can leverage the *local* storage of the computing resource while it is processing the tasks (adopting some of Hadoop [12]'s philosophy where computations occur always *close* to the data). Since the User Data Manager can manage the overall input/output data staging, all of processed data can be effectively merged into a single data set that can be delivered to the user. Although it is difficult for us to support a very large data set such as Hadoop is targeting (terabytes or even petabytes with a very large block size) by using this approach, HTCaaS can effectively leverage local disks of geographically distributed computing resources (such as PLSI nodes) to support data-intensive HTC/MTC applications where typical size of a single input data is relatively small (from hundreds of KBs to MBs) [1, 8].

3.3 Protein Docking Experiments

In this section, we present experimental results from multiple users submitting various numbers of tasks simulating the protein docking throughout Autodock [2]. Autodock is a suite of automated docking tools to predict how small molecules (such as substrates or drug candidates) bind to a receptor of known 3D structure (docking) [2]. In our case, we use this Autodock tool to perform the docking of ligands to a set of target proteins to discover new drugs for several serious diseases such as SARS or Malaria with our collaborators from the School of Biological Sciences and Technology at the Chonnam National University in Korea (more than 500,000 ligand docking simulations have been performed throughout HTCaaS on PLSI).

In this experiment, we utilize all of computing resources connected to the HTCaaS (as seen from Table 1) and comparison models include HTCaaS employing the dynamic fair resource sharing algorithm presented in Section 2.2 (**DF**), and a simple resource partitioning mechanism where the entire available resources are equally divided to the users without adjustment (**Simple**). We assume that total four different users are sequentially arriving at our system and submit various numbers of tasks (i.e., ligands, from 1000 down to 100 for the single cluster, and from 2000 down to 250 for the multi-cluster) with an average inter-arrival time of 10 minutes (average running time of a single ligand is about 760 seconds). We also tested our HTCaaS schemes on two different cluster environments: single cluster where only the `glory` cluster was used for task processing and multi-cluster where all of four different clusters (`glory`, `kobic`, `gene` and `helix`) are utilized for protein docking experiments.

Figure 4 shows the performance results of `DF` and `Simple` on the single cluster (Figure 4a) and multi-cluster (Figure 4b) environments. At the time of our experiment for the single cluster, the `glory` cluster had total 360 free CPU cores. `Simple` divides these available free cores *equally* across four different users (i.e., each user has 90 cores for the protein docking process) to achieve a strict fairness, while `DF` dynamically adjusts the number of allocated cores to each user to achieve the dynamic fairness. As we can see from Figure 4a, `DF` outperforms `Simple` by effectively reducing the makespans for users with a large number of ligands (1000 or 500) and showing competitive performance for the smaller number of tasks (250 or 100).

`DF` can achieve even better performance compared to the

`Simple` (gets only 64.4% of total makespan) by seamlessly integrating multiple clusters (as seen from Figure 4b). With respect to the resource usage, `DF` can fully utilize available computing resources as they become available, while `Simple` inevitably wastes idle computing resources (as seen from Figure 4c). The (four) stairs in the `Simple` graph in Figure 4c are showing that at each time, a new user arrives at the system and submits a number of tasks to be processed. The gap between `Simple` and `DF` graphs are indicating the resource wastage during the course of experiments.

4. RELATED WORK

Middleware systems such as Falkon [4, 9] and MyCluster [14] employ multi-level scheduling mechanisms to support a large number of jobs. The dispatcher in the Falkon efficiently dispatches tasks to the executors (throughout GRAM4) on the computing resources which can be dynamically acquired by the provisioner. Falkon also implements data diffusion approach [4] by adding the data-aware scheduler in the dispatcher component which enables each executor to utilize the data cached in nearby neighbors. HTCaaS can leverage the local storage file system in each computing node during the course of computations. MyCluster creates personal clusters in user-space to support the submission and management of thousands of compute-intensive serial jobs to the resources on the TeraGrid by using the Condor or SGE clusters. Similarly, Sobie et al. created a HTCondor pool across distributed cloud computing systems for HTC scientific applications [10]. However, they still rely on heavy-weight schedulers to dispatch multiple tasks to the virtual cluster (especially compared to the commercial version of the LoadLeveler).

Although Falkon and MyCluster show similar approaches with our HTCaaS system, neither of them can effectively address *Fairness* and *Resource Integration* to support multiple users submitting various numbers of tasks to a collection of distributed computing resources. DIRAC [13] is another type of multi-level scheduling system in the Grid Community to provide a complete solution for exploiting distributed computing resources of the LHCb experiment at CERN for data production and analysis. However, similar to Falkon or MyCluster, DIRAC cannot effectively leverage fairness among multiple users who want to actively share the common resources.

5. CONCLUSIONS

By employing the concept of Meta-Job (with easy-to-use client tools) and a fault-tolerant agent-based multi-level scheduling mechanism, HTCaaS can reduce the overheads of users from handling a large amount of jobs and computing resources (addressing *Ease of Use*, *Efficient Task Dispatching* and *Reliability*). Also, employing dynamic fair resource sharing mechanism enables HTCaaS to effectively address *Fairness* and *Adaptiveness*. To circumvent the performance bottleneck of the shared storage system employed in the PLSI, HTCaaS effectively leverages local disks of geographically distributed computing resources to support data-intensive HTC/MTC applications.

Our future work can include supporting more complex workloads consisting of HTC and HPC tasks, improving the scalability of HTCaaS, and applying job profiling technique to realize the weighted form of fairness.

6. REFERENCES

[1] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Nov. 2004.

[2] Autodock: a suite of automated docking tools. Available at http://autodock.scripps.edu/.

[3] General Parallel File System: Efficient storage management for big data applications. Available at http://www-03.ibm.com/systems/software/gpfs/.

[4] I. F. Ioan Raicu, Yong Zhao and A. Szalay. Accelerating Large-Scale Data Exploration through Data Diffusion. In *Proceedings of the 2008 ACM International workshop on Data-aware distributed computing (DADC'08)*, June 2008.

[5] Open Grid Forum Job Submission Description Language. Available at http://www.gridforum.org/documents/GFD.56.pdf.

[6] Partnership & Leadership for the nationwide Supercomputing Infrastructure. Available at http://www.plsi.or.kr/.

[7] PRACE: Partnership for Advanced Computing in Europe. Available at http://www.prace-ri.eu/.

[8] I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *Proceedings of the Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS'08)*, Nov. 2008.

[9] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Towards Loosely-Coupled Programming on Petascale Systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC'08)*, Nov. 2008.

[10] R. Sobie, A. Agarwal, I. Gable, C. Leavett-Brown, M. Paterson, R. Taylor, A. Charbonneau, R. Impey, and W. Podiama. HTC Scientific Computing in a Distributed Cloud Environment. In *Proceedings of the 4th Workshop on Scientific Cloud Computing (ScienceCloud) 2013*, June 2013.

[11] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[12] The Apache Hadoop project: open-source software for reliable, scalable, distributed computing. Available at http://hadoop.apache.org/.

[13] A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, C. Cioffi, J. Closier, R. G. Diaz, G. Kuznetsov, Y. Y. Li, R. Nandakumar, S. Paterson, R. Santinelli, A. C. Smith, M. S. Miguelez, and S. G. Jimenez. DIRAC: a community grid solution. *Journal of Physics: Conference Series*, 119:062048, 2008.

[14] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner. Creating Personal Adaptive Clusters for Managing Scientific Jobs in a Distributed Computing Environment. In *Proceedings of the Challenges of Large Applications in Distributed Environments (CLADE'06)*, June 2006.

[15] XSEDE: Extreme Science and Engineering Discovery Environment. Available at https://www.xsede.org/.