2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Platform and Co-runner Affinities for Many-Task Applications in Distributed Computing Platforms

Seontae Kim[1], Eunji Hwang[1], Tae-kyung Yoo[1], Jik-Soo Kim[2], Soonwook Hwang[2] and Young-ri Choi[1]
[1] *School of Electrical and Computer Engineering, UNIST, Ulsan, Korea*
Email: {stkim,hwangej88,tkyoo,ychoi}@unist.ac.kr
[2] *National Institute of Supercomputing and Networking, KISTI, Daejeon, Korea*
Email: {jiksoo.kim,hwang}@kisti.re.kr

***Abstract*—Recent emerging applications from a wide range of scientific domains often require a very large number of loosely coupled tasks to be efficiently processed. To support such applications effectively, all the available resources from different types of computing platforms such as supercomputers, grids, and clouds need to be utilized. However, exploiting heterogeneous resources from the platforms for multiple loosely coupled many-task applications is challenging, since the performance of an application can vary significantly depending on which platform is used to run it, and which applications co-run in the same node with it. In this paper, we analyze the platform and co-runner affinities of many-task applications in distributed computing platforms. We perform a comprehensive experimental study using four different platforms, and five many-task applications. We then present a two-level scheduling algorithm, which distributes the resources of different platforms to each application based on the platform affinity in the first level, and maps tasks of the applications to computing nodes based on the co-runner affinity for each platform in the second level. Finally, we evaluate the performance of our scheduling algorithm, using a trace-based simulator. Our simulation results demonstrate that our scheduling algorithm can improve the performance up to 30.0%, compared to a baseline scheduling algorithm.**

***Keywords*-Platform affinity; Co-runner affinity; Many-task applications; Performance analysis; Distributed computing platforms; Scheduling algorithms**

I. Introduction

Recent emerging applications from a wide range of scientific domains (e.g., astronomy, physics, pharmaceuticals, chemistry, etc.) often require a very large number of loosely coupled tasks (from tens of thousands to billions of tasks) to be efficiently processed with potentially large variances of task execution times and resource usage patterns. This makes existing computing paradigms such as High-Throughput Computing (HTC) [14] or Volunteer Computing [7] expand into Many-Task Computing (MTC) [19], and brings many research issues in the design and implementation of middleware systems that can effectively support these challenging scientific applications.

To support loosely coupled applications composed of many number of tasks effectively, all the available resources from different types of computing platforms such as *supercomputers*, *grids*, and *clouds* need to be utilized. However, exploiting heterogeneous resources from distributed computing platforms for multiple loosely coupled many-task applications with various resource usage patterns raises a challenging issue to effectively map the tasks of the applications to computing nodes in the different platforms.

The heterogeneous platforms have different characteristics, since different hardware, system software stack, middleware, network and storage configurations are used for each of the platforms. Thus, the runtime of a task for an application can vary dramatically depending on which platform is used to run it. Moreover, the runtime of the task can be significantly affected by co-running tasks, from the same application or different applications, in the same computing node. As a result, in order to minimize the total makespan of multiple (loosely coupled) many-task applications and consequently maximize the overall throughput, the *platform affinity* as well as the *co-runner affinity* of the applications must be fully analyzed, and considered for scheduling.

In this paper, we analyze the platform and co-runner affinities of many-task applications in distributed computing platforms with different types of supercomputers, grids, and clouds. We perform a comprehensive experimental study using four computing platforms, and five many-task applications from real scientific domains (except one). We then define two metrics, one for the platform affinity, and the other for the co-runner affinity, and present a two-level scheduling algorithm, aiming to minimize the system makespan. It distributes the resources of different platforms to each of applications based on the platform affinity in the first level, and for each platform, it maps tasks of the applications, which are allocated some resources of the platform in the first level, to computing nodes based on the co-runner affinity in the second level. Finally, we evaluate the performance of our scheduling algorithm, using a trace-based simulator.

In our simulations, we compare our scheduling algorithm with two other algorithms, fair-AllCore and worst-AllCore, which are ignorant of and/or contrary to the platform and co-runner affinities of applications, trying to show the maximum possible performance gain by being aware of the affinities. In the first level, the fair-AllCore algorithm allocates the

978-1-4799-8006-2/15 $31.00 © 2015 IEEE
DOI 10.1109/CCGrid.2015.129

resources of each platform equally to every application, while the worst-AllCore algorithm allocates the resources to an application with the worst platform affinity. In the second level, both of the algorithms basically map tasks from the same application to the same node for each platform. Our simulation results show that our algorithm can improve the performance up to 30.0% and 18.8%, compared to the worst-AllCore and fair-AllCore algorithms, respectively.

II. Target Platforms and Applications

In this section, we describe distributed computing platforms with three different types of supercomputers, grids and clouds, and five different many-task applications from various scientific domains such as pharmaceutic, bioinformatics, nuclear physics and astronomy that have been leveraged in our experiments.

A. Heterogeneous Computing Platforms

Table I shows detailed hardware specifications of our four different computing platforms. PLSI stands for "Partnership and Leadership for the nationwide Supercomputing Infrastructure" and consists of 8 geographically distributed supercomputing centers having 14 supercomputers connected via a dedicated 1Gbps network, resulting in total 87TFlops of computing power [6]. PLSI provides a common software stack for accounting, monitoring, global scheduling (based on IBM LoadLeveler) and a global shared storage system based on GPFS [23] which is mounted at every computing node for input/output data and executables. However, the shared storage system based on GPFS can be a performance bottleneck and affect the overall job completion rate whenever multiple simultaneous I/O operations are performed on top of it [29, 12]. For our experiments, two PLSI platforms of gene and cheetah were used.

The grid environment that has been used in our experiments consists of a single computing platform (*darthvader.kisti.re.kr*), operated by KISTI [4]. Specifically, the darthvader platform belongs to the France-Asia VO (Virtual Organization) which is a collaboration to share computing resources among multidisciplinary scientific projects between France and Asian countries [10]. As in a typical grid environment, the darthvader platform is composed of a computing element (CE) and a storage element (SE) so that input/output data and executables are stored at the SE and worker nodes in the CE process tasks by utilizing the SE. In order to process a computational task, first we have to copy/move required input data and executables from the SE to a worker node in the CE, and store back the results to the SE after the task processing. This additional file transfer overhead can be substantial as the size and number of required files increase.

Our last computing platform is a private cloud computing system. In our cloud computing platform, virtual machine (VM) image files are stored in the local disk of each host machine running the VMs, and all the worker VMs are created and started to run, before submitting tasks of applications to the platform.

Note that two PLSI (gene and cheetah) and grid (darth) platforms are production-level systems, where the resources are shared by multiple users, and submitted jobs are scheduled by some global schedulers. Therefore, we were unable to reserve an exclusive access to them or replace any part of their software stack.

B. Many-Task Applications

Table II shows the sizes of input/output data, and the average memory usage (over all the platforms) of a task for each of our five different many-task applications, AutoDock [1], Blast [2], CacheBench [3], Montage [5], and ThreeKaonOmega [22]. 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) [1]. In our case, we use this AutoDock to perform the docking of ligands to a set of target proteins to discover potential new drugs for several serious diseases such as SARS or Malaria. AutoDock is a mainly CPU-intensive application with small sizes of memory usage and input/output data files.

BLAST(Basic Local Alignment Search Tool) [2] is a tool to find regions of local similarity between genome sequences by comparing nucleotide or protein sequences to a database of sequences and calculating the statistical significance of matches. BLAST requires a relatively larger input file to be used for sequence matching process so that the data staging cost from the storage to each compute node can be substantial compared to other applications.

Montage [5], an Astronomical Image Mosaic Engine, is a toolkit for assembling Flexible Image Transport System (FITS) images into composite images called mosaics. Unlike the other applications, each task of Montage generates a substantial amount of intermediate files which can result in total hundreds of MBs. Therefore, Montage is a very I/O-intensive application which requires efficient support of simultaneous I/O operations.

ThreeKaonOmega is a simulation study of the multi-particle production scattering problem in the nuclear physics area [22]. Basically, it aims to analyze the experimental data from accelerator facilities and predict the physical observables in the scattering process, and the overall process consists of high dimensional matrix computations. ThreeKaonOmega is a pure CPU-intensive application with almost no memory usage and disk I/O operations.

Finally, in addition to our four real scientific applications having different resource usage patterns, we have used CacheBench [3] to study behaviors of our platforms under heavily memory-intensive operations. For CacheBench, a different amount of memory is used for each platform, based on the amount of memory and number of cores per node.

Table I
HETEROGENEOUS COMPUTING PLATFORMS USED IN OUR EXPERIMENTS[1]

Platform Type	Computing Platform	CPU	Memory Size	Total # of Nodes	Network
PLSI	kias.gene (gene) 2.0GHz	AMD Opteron 246 2.0GHz (2 Core)	8 GB	64	1Gbps
	unist.cheetah (cheetah) 2.53GHz	Intel Xeon 2.53GHz (8 Core)	12 GB	61	1Gbps
Grid (vo.france-asia.org)	darthvader.kisti.re.kr (darth) 2.0GHz	Intel Xeon 2.0GHz (8 Core)	16 GB	8	1Gbps
Cloud	Local cloud (lcloud) 2.0GHz	Intel Xeon 2.0GHz (12 Core)	32 GB (2.4GB per VM)	6	1Gbps

[1] Note that these experiments were conducted in May−July 2014.

Table II
SIZES OF INPUT AND OUTPUT DATA AND MEMORY USAGE OF A TASK FOR EACH OF THE APPLICATIONS

Application	Input data	Intermediate data	Output data	Memory Usage
AutoDock	8.0MB	-	3.1KB	157.5MB
Blast	1.5GB	-	1.9MB	410.1MB
CacheBench	-	-	1.4KB	1∼4GB
Montage	74.7MB	970.3MB	2.8MB	63.4MB
ThreeKaonOmega	-	-	6.3KB	0.6MB

III. EXPERIMENTAL ANALYSIS OF PLATFORM AND CO-RUNNER AFFINITIES

We investigated the performance of the five many-task applications, AutoDock(A), Blast(B), CacheBench(C), Montage(M), and ThreeKaonOmega(T) in the four different computing platforms. To understand the effect of performance interference among tasks running on the same computing node, i.e. *co-runners*, from either the same application or different applications, we measured the performance over various combinations of the applications in a node with n cores as follows:

- `OneCore`: only one task of an application runs in the node without any co-runners, regardless of available cores of the node.
- `AllCore`: n tasks from the same application run in the node concurrently.
- Combinations of two applications: for two different applications, $n/2$ tasks from each of the two applications run in the node concurrently with tasks from the other application. Such combinations are `AB`, `AC`, `AM`, `AT`, `BC`, `BM`, `BT`, `CM`, `CT` and `MT`.
- Combinations of four applications: for four different applications, $n/4$ tasks from each of the four applications run in the node concurrently with tasks from the other three applications. Such combinations are `ABCM`, `ABCT`, `ABMT`, `ACMT` and `BCMT`.

For `OneCore` and `AllCore`, the total numbers of tasks for AutoDock, Blast, Montage, and ThreeKaonOmega used in our experimental analysis were 512, 768, 1024, and 2304, respectively. However, for combinations of two or four applications, we have executed a subset of the tasks, which were selected randomly, from each application, due to the limit of enough resource allocations from production-level platforms where many other users are sharing the resources.

For a job that is a combination of applications, we requested that a whole node be allocated to it. Once the job was assigned to a node in a platform, multiple tasks of the applications from the combination were executed concurrently for a certain duration which varied case by case. The execution duration of a combination with co-runners was between 1 hour and 67 hours. By executing a combination of applications continuously at least for one hour, we believe that the effects of interference on each application were reflected in results.

Table III
AVERAGE SLOWDOWN AGAINST `OneCore` (%)

Application	gene	cheetah	darth	lcloud
AutoDock	5.6	9.9	10.0	8.7
Blast	0.0	33.8	7.5	12.0
CacheBench	12.6	0.3	3.7	0.4
Montage	3.6	70.8	9.8	12.7
ThreeKaonOmega	4.7	5.9	6.4	4.3

In our experiments, we measured the runtime of each task for an application, and computed the average runtime of the application in a platform. We also measured CPU utilization, memory usage, and disk I/O performance. Note that in PLSI platforms, files for data and executables were stored in a global shared storage system based on GPFS, and so network I/O performance was measured.

A. Effects of Co-runners

We investigated the affinities of the five many-task applications to various combinations of co-runners. Figure 1 shows the *average runtime of tasks* for each of the applications, normalized by that without co-runners (`OneCore`) in the same platform, over various combinations of the applications, and Table III presents the average slowdowns of tasks for the applications against `OneCore` (across all combinations) in each of the platforms. Note that to investigate the pure effects of co-running tasks, the average task runtime used in this Section III-A for darth is computed based on times to execute a task on a computing node, not including the file transfer overhead.

From these results, we can observe the followings in general. First, interference between co-running tasks in gene has a modest or almost no effect on their performance (except CacheBench), since a node in gene has only two cores. Second, running a task with Montage tasks can slow down the execution of the task considerably, because Montage requires heavy file I/O.

For AutoDock, the maximum performance degradation (compared to `OneCore`) is 15.6% in darth with `AC`. For

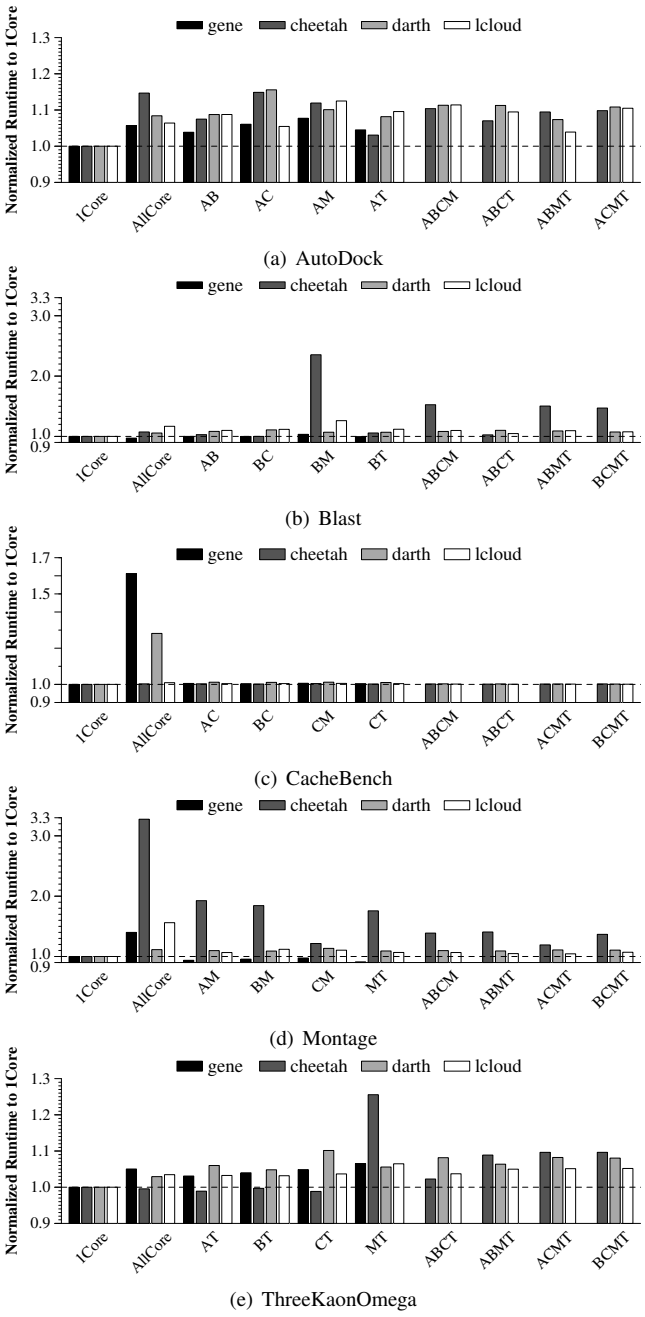

Figure 1. Runtimes over various combinations, normalized by `OneCore`[2]

different combinations, the effect of co-runners on the slowdown of AutoDock is similar to each other. For CacheBench, co-running tasks have no effect on its performance, except `AllCore` in gene and darth. In these two cases, the runtime increases because the size of memory of a computing node is not sufficient slightly to run 2 and 8 CacheBench tasks, respectively, ending up using the swap disk of the node. For ThreeKaonOmega, the effect of co-runners tends to be modest, except `MT` in cheetah, in which its runtime increases by 25.6%, due to the effects of co-running Montage tasks.

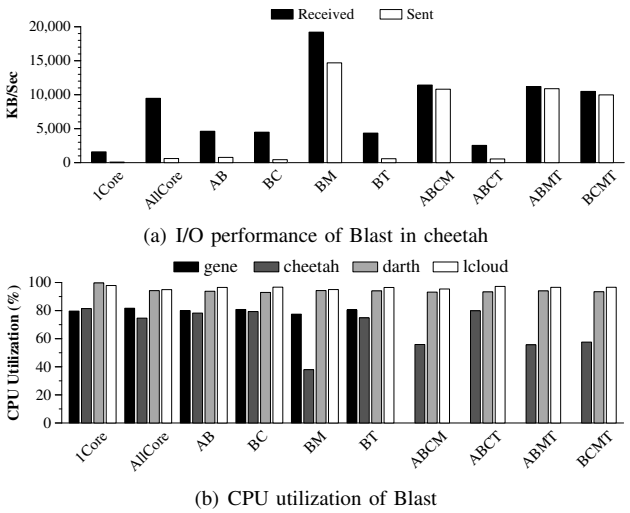

Figure 2. Resource usage patterns of Blast

For Blast, it reads a large amount of input data during its execution, and so its performance is significantly degraded in cheetah, when it is executed together with Montage that generates a large amount of intermediate data. Figure 2(a) shows the average amount of data to be received and sent for a node during the execution in cheetah, and Figure 2(b) shows the average CPU utilization of Blast over all the platforms. (Note that in Figure 2(a), the data accessed by other applications in a combination was included.) When a large amount of data access to GPFS occurs in cheetah, then GPFS becomes a performance bottleneck, and the CPU utilization drops significantly (especially for `BM`), degrading the performance of Blast.

For Montage, it requires to write a large amount of data. Thus, when multiple Montage tasks run together on a node of the PLSI platforms of gene and cheetah, its runtime increases dramatically because of contention in GPFS. With `AllCore`, its runtime in gene increases by 39.9%, which is much smaller than 227.6% in cheetah. It is because only two Montage tasks run together in gene, while 8 tasks run together in cheetah. In cheetah, the effect of co-runners varies over different combinations, but the effect tends to decrease as the number of concurrently running Montage tasks in a node decreases. The CPU utilization of Montage becomes low as 21.1%∼75.4% in gene and cheetah, resulting in the performance degradation, similar to Blast with `BM`.

As co-runners, the effect of CacheBench is stronger than that of Montage in darth. For example, in cheetah, the average runtime of Blast does not increase with `BC`, but increases by 135.4% with `BM`, while in grid, the average runtime increases by 11% with `BC` and 7% with `BM`. Figure 3 shows the cache performance evaluated by CacheBench, which measures the performance as it increases the length

[2]Note that we use various scales for y-axis to show small differences clearly.

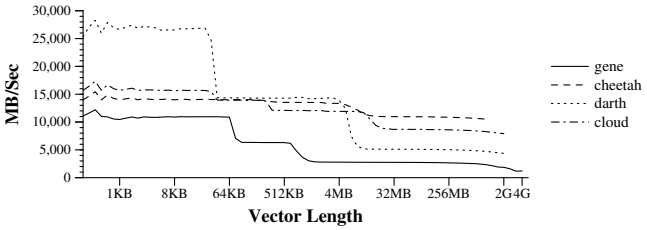

Figure 3. Cache performance of the four platforms

Table IV
OVERHEAD OF GRID COMPUTING PLATFORM

Application	Total File Transfer Time (Sec)	Actual Runtime (Sec)	Overhead (%)
AutoDock	5.14	347.11	1.48
Blast	3.82 (bundle)	46.35	8.24
CacheBench	3.09	357.93	0.86
Montage	7.07	142.41	4.97
ThreeKaonOmega	4.72	101.08	4.67

of a vector, for the platforms. When the length becomes larger than the size of the last-level cache, the performance degradation in darth is larger than that in lcloud and cheetah. Thus, the effect of CacheBench tasks as co-runners can be higher in darth than in the other platforms, specially with `AC` and `CT`.

B. Effects of Platforms

We investigated the affinities of the five applications to the different platforms. To investigate how the characteristics of the grid can affect the performance of an application, the file transfer overhead should be reflected in the performance of an application in darth. Table IV presents the file transfer overhead and actual runtime for a task of the applications on average. In case of Blast, the size of its input file is large as 1.5GB and so its transfer time is 76.36 seconds, but a runtime for each task is short as 42.54 seconds with `OneCore`. It is very inefficient to transfer the input file to a worker node in the CE from the SE for executing a single Blast task. To reduce this overhead, we assumed that 20 tasks are packaged as one bundle. Thus, the input file is transferred to the worker node with a bundle, and used to execute all the tasks in the bundle (as similar to [20]). For Blast, the total transfer time of a task is computed as the transfer time of 1.5 GB divided by 20, and overhead with bundling is computed in the table.

In Table V, the average runtimes and standard deviations of tasks for each of the applications with `OneCore` over the different platforms are presented, and in Figure 4, the average runtimes in Table V are normalized to that in the fastest platform for each of the applications. The runtime of an application with `OneCore` can vary considerably over the different platforms. For AutoDock, Blast and ThreeKaonOmega, the order of platforms with better performance, i.e. a higher affinity, is cheetah (the best), lcloud, darth, and gene (the worst). However, the degrees of the effects by the

Table V
RUNTIMES WITHOUT CO-RUNNERS OVER DIFFERENT PLATFORMS (SECONDS)

App.	gene		cheetah		darth		lcloud	
	Runtime	SD	Runtime	SD	Runtime	SD	Runtime	SD
A	480.42	108.4	244.05	48.6	347.11	69.3	294.16	56.3
B	63.97	42.6	37.9	27.1	46.35	32.6	38.47	29.0
C	376.35	6.5	336.11	0.1	357.93	0.6	354.83	0.4
M	309.21	16.0	150.77	9.7	142.41	12.5	106.63	1.8
T	200.18	8.9	71.02	3.0	101.08	10.3	86.88	0.6

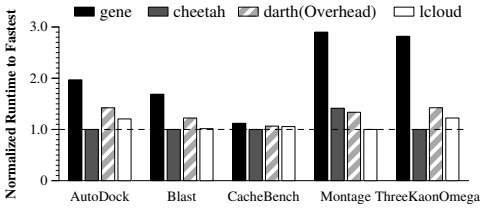

Figure 4. Runtimes over different platforms, normalized to that in the fastest platform

platforms for the applications are dissimilar. For Montage, its performance is degraded in the PLSI platforms significantly due to GPFS overhead, and so it has the best performance in lcloud. For CacheBench, its performance has almost no difference in the different platforms.

Therefore, the platform-level characteristics can also affect the overall performance of applications, but the level of impact can vary depending on the resource usage patterns of them.

C. Composite Analysis of Affinities

In this section, we analyze the effects of the platform and co-runner affinities in a comprehensive manner. Figure 5 shows the average runtimes of Blast, Montage, and ThreeKaonOmega with various combinations over the different platforms. For Blast, when it is executed with some applications other than Montage, it has a similar performance trend over the platforms as with `OneCore`. However, when it is executed with Montage, it has better performance in lcloud and darth than in cheetah. With Montage as co-runners, the impact of co-runners becomes dominant over that of platforms. For ThreeKaonOmega, the impact of platforms on its performance seems to be much stronger than that of co-runners. Although its runtime in a platform can be slightly affected by a different combination of co-runners, the major performance trend depends on which platform was used to run it. For Montage, its performance is strongly affected by both the platform and co-runner affinities.

From the above results, we can conclude that the impact of platforms and that of co-runners appear differently for each of the applications. For some applications, the affinity to a platform can be a main factor to determine the expected performance, but for some other applications, the affinity to co-runners can be more influential. In our results, the performance degradation due to the co-runner affinity is

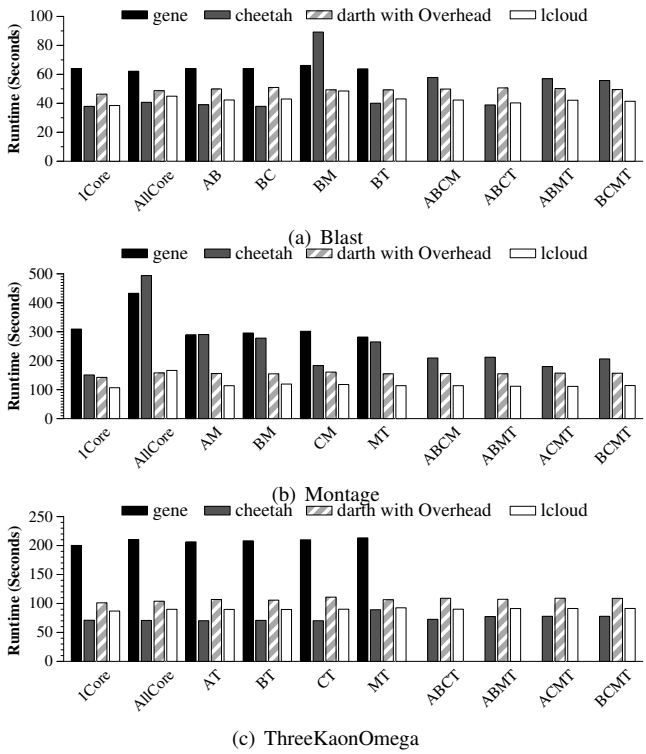

Figure 5. Runtimes with various combinations over different platforms

up to 3.28 times, and that due to the platform affinity is up to 2.9 times. Thus, we must take both of the platform and co-runner affinities into account on scheduling various applications over the heterogeneous platforms. Considering only one of the affinities is not sufficient to optimize the performance for the applications.

IV. Two-level Scheduling Mechanism

In this section, we first discuss two metrics, representing the platform and co-runner affinities of the applications. We then present a two-level scheduling algorithm based on the affinities for a system composed of multiple computing platforms.

A. Platform and Co-runner Affinity Metrics

To measure the effects of platforms and co-runners on the performance of many-task applications, we define two metrics, *platform affinity*, and *co-runner affinity*. The platform affinity of an application K to a platform P indicates how *suitable* platform P is to run application K. The platform affinity metric of K to P can be computed as follows. First, the average runtimes of tasks for K with `OneCore` in various platforms are normalized to the runtime in P. Then, the platform affinity of K to P is computed as the average of the normalized runtimes of the other platforms (not including P). For example, the average task runtimes of Montage on gene, cheetah, and darth, normalized to that of lcloud, are 2.90, 1.41, and 1.34, respectively, and so the platform affinity metric of Montage to lcloud is computed as $\frac{(2.90+1.41+1.34)}{3} = 1.883$.

Table VI
Platform affinity metric

	gene	cheetah	darth	lcloud
AutoDock	0.614	1.532	0.978	1.214
Blast	0.640	1.308	1.009	1.284
CacheBench	0.929	1.080	0.994	1.006
Montage	0.431	1.234	1.326	1.883
ThreeKaonOmega	0.431	1.822	1.181	1.428

Table VII
Co-runner affinity metric

	gene	cheetah	darth	lcloud
AutoDock	0.062	0.104	0.090	0.092
Blast	0.027	0.273	0.024	0.050
CacheBench	0.101	0.003	0.027	0.004
Montage	0.080	0.706	0.098	0.119
ThreeKaonOmega	0.042	0.058	0.161	0.043

This metric of the platform affinity represents the relative runtime of application K, if only the other platforms are equally used to run all the tasks of K. Thus, if the computed value is larger than 1, we can conclude that it is beneficial to run K on P, since if not, the runtime of K will increase. The higher a value of the platform affinity is, the more suitable P is to run K. If the computed value is less than 1, then it means that the runtime of K decreases if P is not used to execute it, and so P is not suitable to run K.

The co-runner affinity of an application K to a platform P indicates how strongly the performance of K is affected by any co-runners in platform P. To compute the co-runner affinity of K to P, we first compute the co-runner affinity of K to each combination C, which is composed of some number of applications including K, in P. It indicates how *suitable* the tasks of the applications in C are to run together with a task of K in the same node in P.

For each combination C (of `AllCore`, and two/four applications), the difference between the runtime of K with C and that with `OneCore` (i.e. runtime with C − runtime with `OneCore`) in platform P is computed. Then, the computed difference is normalized by the runtime with `OneCore`. Note that it is unlikely, but it is possible, that the difference is negative, and in this case, we replace the negative value to 0. The lower a value of the co-runner affinity of K to C is, the more suitable combination C is to run K in P. Finally, the co-runner affinity of K to P is computed as the average of the co-runner affinity values of K to all the combinations.

This metric of the co-runner affinity of K to P represents the performance degradation of K when a task of K runs together with other tasks, from the same application or other applications, in the same node in P. Therefore, if the value of this metric is small, then we can conclude that the performance of application K is not strongly affected by any co-running tasks in P.

Tables VI and VII show the platform and co-runner affinity metrics of the applications computed as the above for each of the platforms.

B. Two-level Scheduling Algorithm

To demonstrate the importance of being aware of the platform and co-runner affinities, we present a two-level scheduling algorithm that optimizes the system performance in a basic way based on the affinities. Our goal is to minimize the system makespan by allocating the computing cores of a platform equally to some number of applications that have relatively high platform affinity values for the platform, and then finding combinations, using those applications, which have a small effect of co-runners in the platform. Therefore, our algorithm decides the number of cores to be allocated to an application for each of the platforms based on the platform affinity in the first level, and for each platform, it maps tasks of the applications to nodes in the platform based on the co-runner affinity in the second level.

Note that it can be advantageous to allocate the resources of a platform to several applications with diverse resource usage patterns, rather than one application with the highest platform affinity, for finding combinations that result in less performance degradation due to co-runners.

In this work, pilot jobs are used to reduce the overhead to schedule a large number of tasks for many-task applications. For a pilot job based system, there is a separate queue for each of submitted applications, and once a core of a platform is allocated to an application, a pilot job starts to execute on that core. It continuously fetches a task from the queue of the application, and executes until there are no tasks left in the queue. This mechanism is similarly implemented in [12, 21, 20].

For this affinity-aware scheduling algorithm, the following inputs are initially given.

- The number of nodes, and the number of cores per node for each of the platforms
- The number of (remaining) tasks, N_{task}, for each of the applications
- The platform and co-runner affinity metrics of each of the applications over all the platforms

In the first level scheduling, for each of the platforms, we first compute the average of the platform affinity values of all the applications, and the number of applications with a platform affinity value higher than the average. Basically, the available cores of a platform are distributed and allocated equally to the applications with a platform affinity value higher than the average. However, the number of cores allocated to each application should be limited by N_{task}. Thus, we allocate the cores of a platform to each of the applications in increasing order sorted by N_{task} of the applications, so that if an application needs less than the equal share of cores for the platform, the remaining cores will be allocated to the other applications.

Also, for an application K, we decide the number of cores to be allocated to K for each of the platforms in decreasing order sorted by its platform affinity. In this way, if N_{task} for K is less than the total number of cores which can be allocated to K from the different platforms, we can allocate cores from a platform with a higher value of the platform affinity first to application K.

After this process, it is possible that the available cores of some platforms have not been distributed yet, and some applications still need to get more cores. In this case, we repeat the above process. When we recompute the average value of the platform affinity for a platform with some available cores, we only consider the applications that still need to get more cores. Thus, the newly computed average can become larger than the old one. For example, initially the average of the platform affinity for gene is 0.609, but if ThreeKaonOmega is allocated all the needed cores and so it is not included to compute the average, the newly computed average becomes 0.653. Using the newly computed average, some application like Blast that has been allocated the resources from gene in the previous round cannot get the resources from gene any more. To avoid such cases, we remain to use the old average value.

Once the first level scheduling is done, for each platform, a set of the applications assigned to the platform and the number of cores allocated to each of the applications (i.e. the number of pilot jobs to be executed for the application) from the platform are computed. In the second level scheduling, for each of the platforms, we first search an application with the maximum co-runner affinity (whose performance can be most affected negatively by co-runner tasks) to the platform. For the selected application, we find a combination C with the smallest co-runner affinity. We then attempt to map this combination to a node as much as possible in the platform. If the remaining number of pilot jobs for some of the applications in C is not sufficient to create a whole combination any more, we repeat the above process to find a new combination from the remaining applications. If only a few number of pilot jobs, which are not enough to make a combination, from some of the applications remain, we make a random combination and assign it to a node.

In this scheduling algorithm, an application with low values of the platform affinity over all the platforms may suffer from starvation if applications with high values of the platform affinity are submitted continuously. In this case, an aging technique can be used to increase the values of the platform affinity periodically for such starving applications.

Note that the scheduler monitors and adapts to changes in the system, such as the submission and termination of applications, and the addition and removal of platforms. For a change, the scheduler redistributes the resources to the applications and re-map pilot jobs to nodes, but our system allows running tasks to complete without preemption as in [12].

Table VIII
DEFAULT RESOURCE CONFIGURATION

	gene	cheetah	darth	lcloud
# nodes	60	15	15	10
# cores per node	2	8	8	12
# total cores	120	120	120	120

Table IX
THE NUMBER OF TASKS IN DEFAULT WORKLOAD

	A	B	C	M	T
# Tasks	9,360	68,400	9,000	18,000	27,648

V. SIMULATION RESULTS

To evaluate the performance of our affinity-aware scheduling algorithm, we have implemented a trace-based simulator in C++. The trace of task runtimes obtained from the experiments discussed in Section III was used in the simulations. For `AllCore`, we have measured runtimes for all the tasks, but for the other combinations, we have measured runtimes for a subset of the tasks. For the simulations, to estimate the runtime of a task in these combinations, we used regression analysis based on the measured runtimes for a subset of the tasks. For an application K and a combination C in a platform, it estimates the relationship between the runtime of a task for K with `OneCore` and the runtime of the task with C.

In the simulations, we assume that all applications are submitted at the same time. Table VIII shows the default resource configuration used in our simulations. The total number of cores in the system was 480. Table IX shows the default workload. For this workload, we replicated tasks of the applications used in Section III. It was generated such that the five applications finish their execution around the same time, if the resources of every platform are allocated equally to each of the applications, and each task is executed with `OneCore`.

For each simulation run, a particular task of an application can be executed on different platforms with a different combination. This is because when two or more pilot jobs of the application from multiple platforms try to fetch a new task in the queue of the application at the same time, the task at the head of the queue is assigned to one of the pilot jobs randomly. For the result of each simulation, the average of 100 simulation runs was computed.

To analyze the maximum possible performance improvement by being aware of the platform and co-runner affinities, we implement two other algorithms, *fair-AllCore* and *worst-AllCore*, and compare our algorithm with them. In the fair-AllCore algorithm, the scheduler allocates the resources of each platform *fairly* (i.e. equally) to every application in the first level, and for each platform, it basically maps tasks of the applications with `AllCore` in the second level. For a system that is unaware of the platform and co-runner affinities, this algorithm may be commonly used. In the worst-AllCore algorithm, the scheduler allocates the resources of a platform to an application with the worst platform affinity in the first level, and then maps tasks to nodes with `AllCore` in the second level. The performance of this algorithm may be almost close to that of the worst case, and so we use its performance as a baseline.

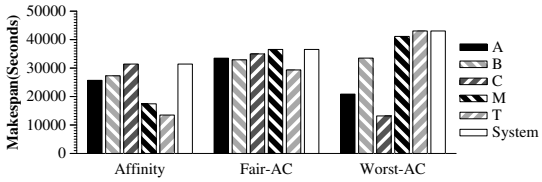

Figure 6. Makespans with the three algorithms

Figure 6 shows the makespans of the applications and the system when the default setting is used for our affinity-aware algorithm (Affinity), the fair-AllCore algorithm (Fair-AC), and the worst-AllCore algorithm (Worst-AC). The performance improvements of Affinity and Fair-AC algorithms against Worst-AC algorithm are 27.0% and 14.1%, respectively. In Affinity algorithm, ThreeKaonOmega and Montage complete their execution earlier than the other applications, since they have relatively higher values for the platform affinity. Thus, ThreeKaonOmega and Montage are allocated a larger number of cores than the other three applications. Also the performance of Montage can improve by using a combination other than `AllCore`. However, CacheBench is allocated 40 cores of gene, and continues to use only the cores until AutoDock is done. Thus, it finishes the execution at last. In Fair-AC algorithm, Montage has the longest makespan, because its performance is significantly degraded with `AllCore`.

Figure 7 shows the makespans over various workloads with the default resource configuration. In each *"Less K"* workload, the workload of application K is reduced to a half of the default workload. Figure 8 shows the makespans over various resource configurations of the system for the default workload. In each *"W/o P"* configuration, the resources from platform P are not used, resulting that the total number of cores in the system is 360. In these results, we can observe the similar trends for the three algorithms as in Figure 6.

In Less M and W/o cheetah, the improvement of Affinity algorithm against Worst-AC algorithm is reduced. Montage is affected considerably by the platform and co-runner affinities, and so in Less M, the workload that can be executed more efficiently by Affinity algorithm is reduced. When the resources of cheetah, which is a platform with a higher value of the platform affinity for all the applications except Montage, are not available, Affinity algorithm cannot use these resources for improving the performance. In W/o darth and W/o lcloud, the performance gap between Affinity and Fair-AC algorithms is a bit increased, compared to the default setting in Figure 6. Because Montage has the high platform affinity to lcloud and darth, the performance of Montage degrades in W/o darth and W/o lcloud. However,

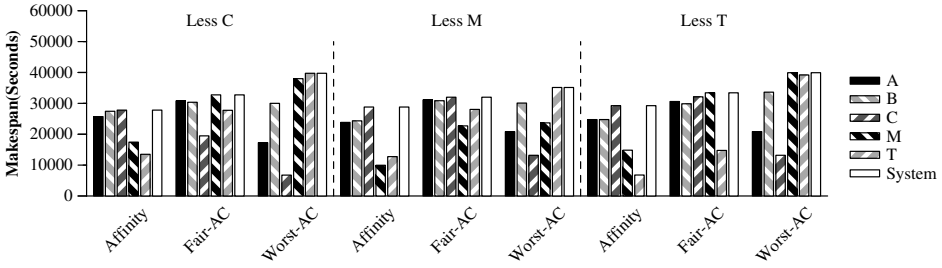

Figure 7. Makespans over various workloads

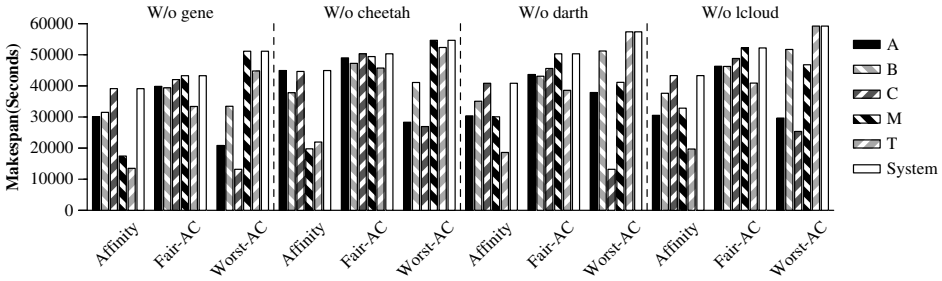

Figure 8. Makespans over various resource configurations

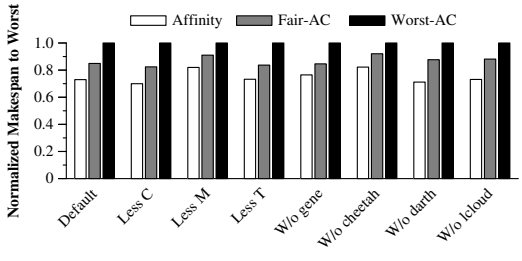

Figure 9. Makespans normalized by the worst-AllCore algorithm

with Affinity algorithm, Montage is assigned the resources of darth in W/o lcloud, and those of lcloud in W/o darth, and so the performance degradation of Montage is less severe, compared to Fair-AC algorithm.

Figure 9 presents the system makespans of the three algorithms normalized to those of Worst-AC algorithm for the simulations discussed above. The performance improvement of our algorithm against Worst-AC algorithm is 30.0% at maximum and 24.8% on average, and that against Fair-AC algorithm is 18.8% at maximum and 13.5% on average.

VI. Related Work

The performance of many-task applications has been analyzed in a certain platform such as supercomputers [26, 8] and clouds [11, 17]. However, the performance difference caused by unique characteristics of platforms as well as diverse combinations of co-runners has not been investigated. For virtualized environments, the performance interference has been analyzed, but the main focus was to understand the effects of virtualization [18, 13].

The effects of different platforms and co-runners on the performance of applications have been investigated in the domain of large-scale datacenters and web-service workloads [16]. Also, several mapping heuristics for a set of independent tasks on heterogeneous computing systems have been studied [9, 15]. In the above work, the heterogeneity is mainly caused by different machine types like different microarchitectures. In [16], usually two long running web-service applications such as websearch are co-located in a node, due to core and memory requirements. Thus, the effects of microarchitectural heterogeneity are much stronger than those of co-runner heterogeneity. In their results, the performance degradation due to co-runners is usually less than 30%, but that due to platform affinity is as high as 3.5 times. However, in our work, we analyze the affinities of diverse many-task applications to heterogeneous platforms, which are different not only in microarchitectures of nodes, but also in software and middleware stack, and network and storage configurations. Furthermore, for many-task applications, a higher degree of co-runners need to be considered, since a task is usually assigned to each core in a node. In our results, the degradation due to co-runners can be higher than that due to platforms.

In [27], for a heterogeneous computing system with supercomputers, grids, and clouds, a proposed scheduling algorithm matches applications and resources, by considering the importance of a platform to an application (which is similar to the platform affinity), and the importance of an application to a platform. However, the effect of co-runners was not considered, and synthetic workloads were used for analysis. In [28], a scheduling algorithm was proposed to mitigate I/O contention for file transfer in grids for MTC.

Various metascheduling approaches to deal with multiple distributed computing environments such as HPC, grids,

and clouds were discussed in [24, 25], including a hierarchical scheduler. In our two-level scheduler, the first level metascheduler decides the number of cores to be allocated to an application for each platform, and the second level scheduler in each platform maps tasks of applications (i.e. pilot jobs) to nodes.

VII. Concluding Remarks

In this paper, we analyzed the platform and co-runner affinities of many-task applications in distributed computing platforms with different types of supercomputers, grids, and clouds. Our comprehensive experimental analysis showed that both platforms and co-runners can affect the performance of applications dramatically. We also demonstrated that it is critical to consider the platform and co-runner affinities on scheduling. The affinity-aware algorithm discussed in this paper is based on basic optimization. A sophisticated affinity-aware scheduling algorithm that can optimize the performance effectively over various scenarios is considered as our future work.

Acknowledgment

This work was partly supported by the KOREA INSTITUTE of SCIENCE and TECHNOLOGY INFORMATION (KISTI) as a subproject of project in 2015 Major Program. It was also partly supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2012R1A1A1043329), and the year of 2012 Research Fund(1.120008.01) of the UNIST(Ulsan National Institute of Science and Technology).

References

[1] Autodock. http://autodock.scripps.edu/.

[2] Blast. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[3] Cachebench. http://icl.cs.utk.edu/projects/llcbench/cachebench.html.

[4] Korea Institute of Science and Technology Information. http://en.kisti.re.kr/.

[5] Montage. http://montage.ipac.caltech.edu/.

[6] PLSI. http://www.plsi.or.kr/.

[7] David P. Anderson, Carl Christensen, and Bruce Allen. Designing a Runtime System for Volunteer Computing. In *Proceedings of the IEEE/ACM SC06 Conference*, November 2006.

[8] Timothy G Armstrong, Zhao Zhang, Daniel S Katz, Michael Wilde, and Ian T Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, 2010.

[9] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[10] Yonny CARDENAS. France-Asia Virtual Organization: Current Status. In *FJPPL Workshop*, 2012.

[11] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick HJ Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2011.

[12] Jik-Soo Kim, Seungwoo Rho, Seoyoung Kim, Sangwan Kim, Seokkyoo Kim, and Soonwook Hwang. HTCaaS: Leveraging Distributed Supercomputing Infrastructures for Large-Scale Scientific Computing. In *6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*. ACM, 2013.

[13] Younggyun Koh, Rob C Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *ISPASS*, pages 200–209, 2007.

[14] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP journal*, 11(1):36–40, 1997.

[15] Muthucumaru Maheswaran, Shoukat Ali, HJ Siegal, Debra Hensgen, and Richard F Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings Eighth Heterogeneous Computing Workshop*, pages 30–44. IEEE, 1999.

[16] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[17] Rafael Moreno-Vozmediano, Ruben S Montero, and Ignacio M Llorente. Multicloud deployment of computing clusters for loosely coupled mtc applications. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):924–930, 2011.

[18] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *IEEE 3rd International Conference on Cloud Computing*, pages 51–58, 2010.

[19] I. Raicu, I.T. Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *1st Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–11, 2008.

[20] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 43. ACM, 2007.

[21] Seungwoo Rho, Seoyoung Kim, Sangwan Kim, Seokkyoo Kim, Jik-Soo Kim, and Soonwook Hwang. HTCaaS: A Large-Scale High-Throughput Computing by Leveraging Grids, Supercomputers and Cloud. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1341–1342. IEEE, 2012.

[22] H.-Y. Ryu, A. I. Titov, A. Hosaka, and H.-C. Kim. ϕ photoproduction with coupled-channel effects. *Progress of Theoretical and Experimental Physics*, 2014(2):020003, February 2014.

[23] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[24] S. Sotiriadis, N. Bessis, F. Xhafa, and N. Antonopoulos. From meta-computing to interoperable infrastructures: A review of meta-schedulers for hpc, grid and cloud. In *IEEE 26th International Conference on Advanced Information Networking and Applications (AINA)*, pages 874–883, March 2012.

[25] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 359–366, 2002.

[26] Ke Wang, Zhangjie Ma, and Ioan Raicu. Modeling many-task computing workloads on a petaflop ibm blue gene/p supercomputer. *IEEE CloudFlow*, 2013.

[27] Jian Xiao, Yu Zhang, Shuwei Chen, and Huashan Yu. An application-level scheduling with task bundling approach for many-task computing in heterogeneous environments. In *Network and Parallel Computing*, volume 7513 of *LNCS*, pages 1–13. 2012.

[28] Yu Zhang, Shuwei Chen, and Ziqian Hu. A scheduling algorithm for many-task computing optimized for io contention in heterogeneous grid environment. In *Fifth International Conference on Computational and Information Sciences*, pages 1541–1544. IEEE, 2013.

[29] Zhao Zhang, Daniel S. Katz, Michael Wilde, Justin M. Wozniak, and Ian Foster. Mtc envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 37–48, 2013.