2014 IEEE 28th International Parallel & Distributed Processing Symposium

A Coprocessor Sharing-Aware Scheduler for Xeon Phi-based Compute Clusters

Giuseppe Coviello, Srihari Cadambi and Srimat Chakradhar
NEC Laboratories America, Inc.
4 Independence Way, Suite 200
Princeton NJ 08540, USA.
{giuseppe.coviello, cadambi, chak}@nec-labs.com

***Abstract*—We propose a cluster scheduling technique for compute clusters with Xeon Phi coprocessors. Even though the Xeon Phi runs Linux which allows multiprocessing, cluster schedulers generally do not allow jobs to share coprocessors because sharing can cause oversubscription of coprocessor memory and thread resources. It has been shown that memory or thread oversubscription on a manycore like the Phi results in job crashes or drastic performance loss.**

We first show that such an exclusive device allocation policy causes severe coprocessor underutilization: for typical workloads, on average only 38% of the Xeon Phi cores are busy across the cluster. Then, to improve coprocessor utilization, we propose a scheduling technique that enables safe coprocessor sharing without resource oversubscription. Jobs specify their maximum memory and thread requirements, and our scheduler packs as many jobs as possible on each coprocessor in the cluster, subject to resource limits. We solve this problem using a greedy approach at the cluster level combined with a knapsack-based algorithm for each node. Every coprocessor is modeled as a knapsack and jobs are packed into each knapsack with the goal of maximizing job concurrency, i.e., as many jobs as possible executing on each coprocessor. Given a set of jobs, we show that this strategy of packing for high concurrency is a good proxy for (i) reducing makespan, without the need for users to specify job execution times and (ii) reducing coprocessor footprint, or the number of coprocessors required to finish the jobs without increasing makespan. We implement the entire system as a seamless add-on to Condor, a popular distributed job scheduler, and show makespan and footprint reductions of more than 50% across a wide range of workloads.

***Index Terms*—Middleware, coprocessors, processor scheduling, high performance computing**

I. Introduction

The Xeon Phi, is a PCIe device with 60 cores capable of supporting 240 hardware threads. It runs the Linux operating system and supports OpenMP, a popular parallel programming model. Consequently, the Xeon Phi is widely perceived to be more usable across a range of parallel applications, especially when compared to other manycore offerings in the recent past [1], [2]. Many OEM vendors are releasing Xeon Phi-based high-performance servers and several application kernels have achieved a speedup on these systems [3]. Clusters based on the Phi such as Stampede at TACC and a Xeon Phi-based data center by the DoE's NREL are also being commissioned. Such clusters already occupy several spots in the Top500 supercomputer list.

Despite these advances on the hardware side, specialized management software that take advantage of Xeon Phi's unique capabilities have not been proposed. Existing cluster managers such as Torque [4] and Condor [5] can be used for Xeon Phi clusters. However, they treat the Xeon Phi as yet another resource and do not take into consideration one its key differentiating factors: the Phi runs Linux and therefore can support execution of multiple jobs concurrently. Recent work [6] has shown that when multiple processes concurrently execute and share the Xeon Phi coprocessor on a single server, overall performance improves. At the same time, sharing can also cause coprocessor resource oversubscription. In contrast to a multicore processor, oversubscribing resources like threads and memory in a manycore coprocessor (like the Phi) can cause arbitrary process crashes and drastically reduce performance [6].

To avoid oversubscription, today's cluster managers generally dedicate Xeon Phi coprocessors to a job for its lifetime [7]. This approach is also prevalent in GPU- and other coprocessor-based clusters. Our experiments show that, for typical workloads, average Xeon Phi core utilization under this policy across a modestly-sized cluster is as low as 38%. This is because typical Xeon Phi offload jobs, which are launched on the host and intermittently offload to the Phi, do not continuously utilize all the Phi resources. Portions of a job run on the host leaving the Phi idle. Furthermore, offloads do not always use all 60 cores (240 threads) all the time [6]. Therefore, if each Xeon Phi is dedicated to a single offload job until its completion, idle periods and not using all cores result in decreased overall coprocessor utilization.

Low coprocessor utilization implies opportunities for sharing. With sharing, two offload jobs that together require 60 or fewer cores can run on one Xeon Phi device without oversubscription. Two jobs that need more than 60 cores can also run concurrently if their offloads do not overlap in time.

But such device sharing even without oversubscription is itself not useful unless it results in something more practical and measurable. In this paper, we investigate if safe Xeon Phi sharing can provide two tangible benefits at the cluster level: (i) makespan reduction and (ii) coprocessor footprint reduction across the cluster.

Specifically, we relax the exclusive coprocessor allocation policy and allow concurrent jobs to run on each Xeon Phi. To

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.44

achieve this, we augment existing cluster management software with a sharing-aware scheduler which uses memory and thread requirements specified by the user to safely share each Xeon Phi coprocessor across several jobs without overrunning resource limits, i.e., without resource oversubscription. We formulate this as a resource constrained packing problem, and use the knapsack algorithm to pack as many jobs as possible on each Xeon Phi. This approach directly increases concurrency. We find that *this is a good proxy for reducing makespan without knowledge of job execution times*. This is a key difference from earlier related work in makespan minimization. Reducing makespan for a set of jobs also allows us to use this approach to reduce coprocessor footprint in the cluster, i.e., reduce the number of Xeon Phi's required to process a set of jobs in a certain time. If all jobs are coprocessor-intensive, reducing the number of coprocessors directly reduces cluster size. We evaluate our method for different Xeon Phi job sets, both synthetic and real, across different resource requirement distributions, and find that makespan and coprocessor footprint can be reduced by more than 50%.

In summary, the key contribution of this paper is as follows. We explore how sharing coprocessors could result in makespan and footprint reduction at the cluster level. We build and evaluate a sharing-aware scheduler for Xeon Phi-based compute clusters. The scheduler runs on top of existing software, and is completely transparent to user applications and the underlying software. We devise a knapsack-based algorithm to pack jobs into coprocessors such that job concurrency is maximized under resource constraints. We show that this approach is a good proxy for reducing makespan as well as reducing footprint without knowledge of job execution times. We integrate our solution with Condor [5], COSMIC [6] and Intel's standard Xeon Phi software stack. We present detailed evaluations on real job sets as well as several controlled experiments on synthetic job sets where we study the effects of Xeon Phi job sharing across several distributions of job resource requirements, demonstrating significant makespan and footprint reductions.

The remainder of the document is organized as follows. We discuss background and motivational data in Sections II and III, our proposed framework and implementation in Section IV. Evaluation results are presented in Section V, related work in Section VI and the conclusion in Section VII.

II. Background

In this section, we introduce the Xeon Phi coprocessor, its offload programming model, thread and memory oversubscription in coprocessors, and finally Condor.

A. The Xeon Phi and Offload Programming

The Xeon Phi coprocessor has around 60 cores connected via a 512-bit bidirectional ring interconnect. It is packaged as a separate PCIe device, external to the host processor. Each Xeon Phi device has 8-16 GB of RAM that serves as the memory and file system storage for every user process, the Linux operating system, and ancillary daemon processes. A

```
#pragma offload target(mic:1) \
  in(a: length(SIZE)) \
  in(b: length(SIZE)) \
  inout(c: length(SIZE))
  for (int i = 0; i < SIZE; i++)
     c[i] = a[i] + b[i];
```

Fig. 1. Offload pragma example.

Xeon Phi core is dual-issue, in-order, and includes sixteen 32-bit vector lanes. Each core supports 4 hardware threads which, although individually slower than multicore threads, provide good aggregate performance for highly parallelized and vectorized kernels. This makes the offload model, where sequential code runs on the host processor and parallelizable kernels are offloaded to the Xeon Phi, an effective programming model.

Offload programming model: A programmer annotates code with offload pragmas [8] to identify regions to be offloaded to the Xeon Phi. An offload pragma can be placed before any statement, including compound statements such as loop nests or an OpenMP parallel block. Fig. 1 shows the use of an offload pragma to offload a `for loop` to the coprocessor. The pragma specifies the keyword `mic` followed by an optional integer to indicate a target Xeon Phi. The compiler builds the offload block (in this case, the `for loop`) to run on both the host and the coprocessor. The variables used by the code block are specified in the pragma statement as inputs, outputs or inouts. Lengths of pointer variables, such as the arrays a, b and c in Fig. 1, must be specified.

B. Xeon Phi software stack (MPSS)

The Xeon Phi software stack, called the MPSS (Many Integrated Core Platform Software Stack), consists of a portion that executes on the host, and a portion that executes on the coprocessor. The host portion includes stock Linux running along with PCI and card drivers. A Symmetric Communication Interface (SCIF) is provided by Intel for communicating between Xeon Phi devices or between the host and a Xeon Phi processor. On top of SCIF, the Coprocessor Offload Infrastructure (COI) [9] is a higher level framework providing a set of APIs to simplify development of applications using the offload model. COI includes APIs for launching device code, asynchronous execution and data transfer between the host and Xeon Phi.

The other part of the stack executes on the coprocessor, and includes a modified Linux kernel and drivers, the standard Linux `/proc` file system that can be used to query device state (for example, the load average), and the coprocessor side of the SCIF driver and the COI library.

For every host process that offloads work to the coprocessor, the COI middleware creates a process on the Xeon Phi that will execute offloaded code sections sent from the host process. Linux on the coprocessor is only aware of processes, but MPSS is aware of offloads within a COI process. However, while scheduling offloads, MPSS does not take into account thread or memory oversubscription.

C. Thread and Memory Oversubscription

Thread oversubscription occurs when the total number of threads across all jobs concurrently using the Xeon Phi exceeds the number of hardware threads. In a manycore processor, this results in performance loss due to the high cost of context switching [10] primarily owing to the large processor state (such as many wide vector registers).

Memory oversubscription occurs when the physical coprocessor memory is oversubscribed. This can result in process crashes since Linux's out-of-memory (OOM) killer randomly terminates processes when physical memory is oversubscribed. Memory oversubscription on the Xeon Phi is a bigger problem than it is on host processors since the coprocessors physical memory (maximum 16GB) is much smaller than the typical host physical memory in a server (100s GB to a few TB). Consequently, a large number of jobs can crash under memory oversubscription. Handling memory oversubscription by accurately monitoring available memory is nearly impossible since memory in Linux is not committed during allocation. In addition, thread stacks are also small when applications enter a parallel region, but can grow with time. This means that two jobs could concurrently run on the manycore without memory oversubscription, but once their stacks or committed memory blocks grow, they may oversubscribe memory and crash at some point in the future.

D. HTCondor ("Condor")

Condor [5] is a cluster job scheduler for compute-intensive jobs. Users submit their jobs to Condor which places them in a queue and chooses when and where to run them based on policies. Condor provides a framework for matching job resource requests with available resources. Its ClassAd mechanism allows each job to specify requirements (such as the amount of memory used) and preferences (such as a processor with more than 4 cores). It also allows cluster nodes to specify requirements and preferences about jobs they are willing to accept and run. Based on the ClassAds, Condor's *matchmaking* matches a pending job with an available machine.

A Condor pool comprises a single machine that serves as the central manager and all other cluster nodes. The central manager collects status information from all cluster nodes, and orchestrates matchmaking. To collect status information, it obtains ClassAd updates from each node. These updates include the state of the node such as currently available resources and load, and jobs that are executing on the node. The central manager then initiates a negotiation cycle during which all pending jobs are examined in FIFO order, and matched with machines. Negotiation cycles are triggered periodically. Once a match is made, a shadow process is started on the machine where the job was submitted, and a starter process on the target machine. The shadow process transfers the job and associated data files to the target machine, where the starter process spawns the user application. When the job completes, the starter process removes all processes spawned by the user job and frees any temporary scratch spaces, leaving the machine in a clean state.

III. Motivation

In this section, we identify and quantify the opportunity for sharing. We set up an 8-node Xeon Phi-based cluster with each server containing 1 Xeon Phi coprocessor, and use a job set consisting of 1000 independent job instances from Table I. All jobs use a single server and offload to a single Xeon Phi on that server. Each job is guaranteed to fit within one Xeon Phi, i.e., each job uses less threads and memory than available on one device. The table shows the jobs, the number of Xeon Phi threads, the range of memory requests across all instances of that job, and the problem size.

We use Condor 7.8.7 and MPSS to process the 1000 job instances on the Xeon Phi cluster. If the system is Xeon Phi agnostic, thread and memory oversubscription of the Xeon Phi occur. [6] showed that performance is impacted by as much as 800% with thread oversubscription on the Phi. It also showed unpredictable job terminations and performance degradations of up to 650% under memory oversubscription. In our experiment with Condor and MPSS, we prevent oversubscription by allowing the servers to advertise Xeon Phi resources (devices and memory), and allowing jobs to request those Xeon Phi resources. Following the exclusive allocation policy, we allocate entire Xeon Phi devices to each job through its lifetime. Since there is no job concurrency on the coprocessors, there is no oversubscription-related crash or performance loss. Therefore at any given time, the maximum number of running jobs on the 8-node cluster with a total of 8 Xeon Phi coprocessors is 8.

We monitored the activity of each processing core on the manycore coprocessor during this experiment. Overall, average core utilization *was measured to be only around 50%*. That is, each coprocessor core was busy for only around half the time. There are two reasons for this. First, each job offloads only intermittently to the coprocessor. Second, and more importantly, a job may not use all 60 cores for all its offloads, thereby leaving some cores idle[1].

We also took synthetic Xeon Phi jobs (details in Section V) with different memory and thread requirements. For several different distributions of memory and thread requirements (e.g., most jobs having high memory and high thread requirements versus most jobs having low memory and thread requirements), we measured coprocessor core activity. We again observed low core utilizations ranging from 38% to 63%.

The fact that *the manycore processing unit is utilized at around half its capacity quantifies the opportunity for sharing*. If jobs share coprocessors, idle cores can be utilized better. We seek to explore this opportunity using our proposed sharing-aware cluster scheduler in order to reduce overall coprocessor footprint in the cluster.

IV. The Proposed Framework

In this section, we describe our framework by first illustrating the benefits of coprocessor sharing, then formulating the

[1]For many jobs, performance saturates at a lower level of parallelization with fewer than 60 cores [6]

Name	Description	Threads	Memory	Problem Size
KM	Computing K-means using Lloyd clustering algorithm	60	300-1250MB	4M points/3 dimensions/32 means
MC	Monte Carlo simulation of N paths and T time steps	180	400-650MB	N = 32M, T = 1000
MD	Molecular dynamics simulation	180	300-750MB	25000 particles, 5 time steps
SG	A series of matrix-matrix multiplications (SGEMM)	60	500-3400MB	8Kx8K matrices, 10 iterations
BT	A CFD application using block tri-diagonal solver [11]	240	300-1250MB	Grid: 162x162x162, 200 iterations
SP	A CFD application using scalar penta-diagonal solver [11]	180	300-1850MB	Grid: 162x162x162, 400 iterations
LU	A CFD application using lower-upper Gauss-Seidel solver [11]	180	400-1250MB	Grid: 162x162x162, 250 iterations

TABLE I
XEON PHI WORKLOADS USED FOR JOB GENERATION.

problem before presenting the solution. Finally, we describe the implementation of our module as a transparent add-on to Condor [5].

A. Benefits of Sharing Coprocessors

We focus on Xeon Phi offload jobs, which launch from the host and intermittently offload to the coprocessor. Sharing opportunities arise due to two reasons. First, a job might have completed an offload and is running on the host leaving the coprocessor free. Second, a job's offload may not be using all of the cores on the coprocessor, allowing another job to potentially use the free cores.

Fig. 2 shows the coprocessor usage profile of jobs J_1 and J_2. J_1 (top of figure) has two offloads both of which use 240 threads (all available hardware threads across 60 cores). Between the two offloads is a portion of time spent on the host during which the coprocessor is free. J_2, whose profile is shown in the middle chart of Fig. 2, has three offloads all using 240 threads. The bottom chart shows the profile when J_1 and J_2 run concurrently and share the Xeon Phi. Since each offload uses all available hardware threads, two offloads cannot run simultaneously due to thread oversubscription. However, "gaps" during which one job runs on the host can be utilized by the other job. Because of this, the overall makespan or execution time of the two jobs running concurrently is less than the sum of the execution times of the jobs (i.e., the sequential makespan). This assumes a simple scheduler at the node to ensure offloads from different jobs do not overlap when there is a possibility of thread oversubscription.

Fig. 3 shows the coprocessor usage profile of two jobs that do *not* use all coprocessor hardware threads, and whose offloads can overlap without thread oversubscription. J_3 has two offloads each of which uses 120 Xeon Phi threads (or 50% of the total available in hardware). J_4 has three offloads also using 120 threads each. The bottom chart in the figure shows J_3 and J_4 running concurrently. Unlike the previous case where an offload had to wait for the previous offload to finish in order to avoid thread oversubscription, these offloads can overlap without oversubscribing thread resources. Specifically, "Offload C" from J_3 can start even before "Offload 4" from J_4 has completed. Similarly, "Offload 5" and "Offload C", and "Offload 6" and "Offload D" can run simultaneously. This overlap not only takes advantage of idle periods between

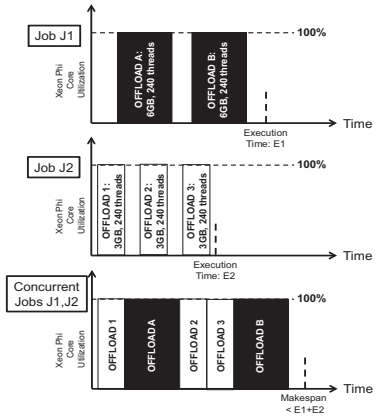

Fig. 2. Two offload jobs using maximal resources running concurrently.

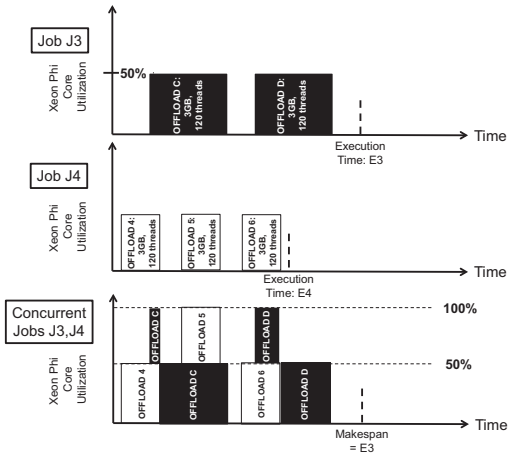

Fig. 3. Two offload jobs using partial resources running concurrently.

offloads, but also partial core occupancy. Consequently, the makespan of the concurrent run is significantly better than the sequential makespan as shown in the bottom of Fig. 3.

B. The Problem

Given a set of jobs and a cluster of Xeon Phi-based compute servers, we decide a schedule that attempts to get as much job concurrency on each Xeon Phi as possible. We show that

this is a good proxy for minimizing makespan, especially since job execution times are not given. Compared to prior work on minimal makespan scheduling such as [12], [13], our formulation has the following important differences:

- We specifically target clusters with coprocessor-based servers;
- We allow concurrent job execution by sharing the Xeon Phi coprocessor while classical makespan minimization scheduling is restricted to one job executing on a server at any given time;
- We impose coprocessor resource constraints (memory and number of threads) on concurrent jobs: no resource can be oversubscribed;
- We do not assume knowledge of job execution times since users usually cannot specify them accurately and therefore they cannot be relied upon;
- We use job concurrency as a proxy for minimizing makespan. Our approach is to schedule multiple jobs to a compute server concurrently and leverage coprocessor sharing in order to achieve low makespan.

We assume the user provides two pieces of information for each job: a maximum Xeon Phi memory requirement, and a maximum thread requirement. This could be relaxed this with tools that automatically estimate jobs' resource requirements. However that is outside the scope of this paper. Assuming users specify this information distills the contribution of this paper, namely sharing-aware coprocessor scheduling. Thus, given *n* jobs $J_1, J_2...J_n$ with Xeon Phi memory requirements $m_1, m_2...m_n$ and thread requirements $t_1, t_2...t_n$, and N identical compute servers each having D Xeon Phi coprocessors with memory limit M and thread limit T, we pack as many jobs as possible on each coprocessor subject to its memory and thread limits.

C. 0-1 Knapsack-based Approach

Traditional minimum makespan scheduling approaches use list scheduling, bin packing or their variants since they consider sequential jobs without resource constraints. We allow jobs to concurrently run on the servers with memory and thread resource constraints. The physical memory is a hard limit that concurrent jobs must not exceed since that will result in undesirable effects such as process crashes and extreme performance loss as reported in [6].

Given a set of jobs, the exhaustive approach would be construct all possible subsets that fit in each coprocessor's memory. Then the subset with most jobs that uses the maximum number of threads (under the coprocessor hardware limit) could be picked, and its jobs scheduled on the coprocessor. Afterwards the subsets must be reconstructed, and the process repeated.

Clearly, such an exhaustive approach would be prohibitively time consuming. We use a greedy knapsack-based approach to arrive at a schedule quickly. Given a job set and the cluster, we model each coprocessor as a knapsack and pack jobs into each knapsack one after another, with the goal of maximizing job concurrency, i.e., the number of jobs in the knapsack. Once we pack a knapsack, we move on to the next coprocessor and repeat the process with the remaining jobs.

```
for each Xeon Phi device D in cluster do
  pack jobs in D using knapsack algorithm
end for
while jobs remaining do
  for each Xeon Phi D with free memory do
    create knapsack: capacity = free memory in D
    pack jobs in D using knapsack algorithm
  end for
end while
```

Fig. 4. Greedy knapsack-based method for sharing-aware scheduling.

The knapsack-based approach allows us to consider both memory and thread constraints. We model the coprocessor-based cluster as a set of knapsacks each with a capacity, and schedule jobs such that the value of the filled knapsacks is maximized. Each Xeon Phi coprocessor in a compute server is a knapsack, and the items in it represent jobs that are concurrently running on that coprocessor. The knapsack capacity is the physical memory of the coprocessor. We set the value of each job such that it decreases with the number of its threads. The value of a knapsack is the sum of the values of the jobs packed in it. Therefore, in trying to maximize the value of the knapsack, the algorithm will try to pack many jobs with few threads into one coprocessor. This maximizes job concurrency, increasing core and device utilization. The value v_i of job J_i in our knapsack formulation is given by:

$$v_i = 1 - \left(\frac{t_i}{240}\right)^2 \tag{1}$$

where t_i is number of Xeon Phi threads requested by the job.

Our objective is to use concurrency to minimize makespan since we do not have knowledge of job execution times. In addition, we also do not know the profile of a job (such as the ones illustrated in Figs. 2 and 3). Knowledge of these could result in an optimal makespan, but is not realistic. Therefore, our method is to set the value of each job such that the knapsack approach tries to have as many concurrent jobs as possible subject to resource constraints. Having more jobs running at the same time on the same coprocessor will increase the chances of the Phi cores being well utilized and the gaps in Xeon Phi usage of any job being filled. In addition, having many concurrently executing jobs also improves chances that a long running job (which affects final makespan) will overlap with several other short jobs. In order to avoid oversubscription, the number of threads of all concurrent jobs must not exceed the number of hardware threads supported by the Xeon Phi. Unlike the memory however, this is not a hard limit. If the total number of threads exceed the coprocessor's hardware limit, we make the knapsack value zero.

We use the standard dynamic programming method to solve the 0-1 knapsack problem. 0-1 indicates items are indivisible: jobs cannot be partially scheduled. Given n jobs (items) $J_1, J_2...J_n$ with Xeon Phi memory requirements (weights) $m_1, m_2...m_n$ and values $v_1, v_2...v_n$, and a knapsack of ca-

pacity M, the maximum value of the knapsack for n jobs $V(n, M)$ depends on the solution obtained for $n-1$ jobs. More generally for i jobs and knapsack capacity m:

$$\begin{aligned} V(i,m) &= 0 \; if \; i = 0 \; or \; m = 0 \\ &= MAX(V(i-1,m), v_i + V(i-1, m-m_i)) \end{aligned}$$

Complexity: Thus the solution to the knapsack of capacity m with i jobs is either the solution obtained for the knapsack without the i'th job, or the solution obtained for a knapsack whose size is reduced by the size of the i'th job m_i. For n jobs with w different sizes, the complexity is $O(nw)$. Typically w is not large: if jobs can request memory in increments of 50MB, then w is $8GB/50MB = 160$, which is smaller than the number of typical jobs on a large cluster. This makes the complexity nearly linear with the number of jobs.

The overall knapsack-based scheduling approach at the cluster-level is shown in Fig. 4. We start by creating a knapsack for each Xeon Phi device in each server and set the knapsack capacity to the full physical device memory. We fill all knapsacks one after another. While each knapsack is filled using the dynamic programming algorithm, filling one knapsack before starting another makes the approach greedy at the cluster level. When any device completes a job, we create a new knapsack whose capacity is set to the device memory that was freed up by the completed job. As long as unscheduled jobs exist, we fill each such new knapsack. This process continues until all jobs have been scheduled and completely executed.

D. System Implementation

Fig. 5 shows the overall implemented system. We integrate our scheduling module with the Condor distributed framework [5]. Many cluster users are already familiar with Condor. Our Xeon Phi sharing-aware scheduler is a transparent add-on to Condor. We use Condor's job submission formats as well as its mechanisms to negotiate with the cluster compute server nodes and dispatch jobs.

Our module interfaces with Condor at the cluster level, and uses COSMIC [6] at each node. *We require no changes to, or sources of, either Condor or COSMIC.*

1) Augmenting HTCondor: Here we describe how we integrate the proposed methods with Condor and COSMIC. As shown in Fig. 6, we set up Condor 7.8.7 in a master-worker configuration with a central manager and compute nodes. Each host processor on a compute node is represented as a slot that can be claimed to run a job. Only one job can run on one slot at a time. Each compute node obtains the number of Xeon Phi cards available as well as the card memory through the Xeon Phis `micinfo` utility, and advertises this in its ClassAd. Each job specifies its preferences for the number of Xeon Phi devices and memory in its job script.

Our cluster scheduler obtains the list of pending jobs from Condor. It then uses the knapsack-based method from Fig. 4 to create a job-to-node mapping. Using the utility condor_qedit,

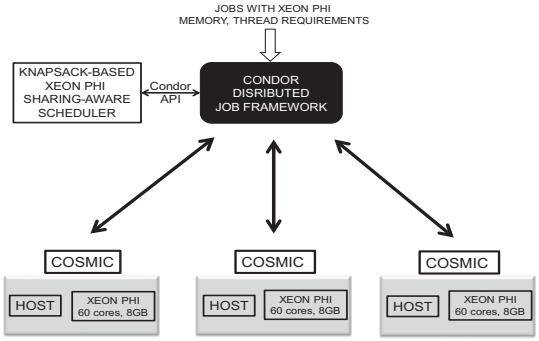

Fig. 5. Proposed overall Xeon Phi cluster middleware.

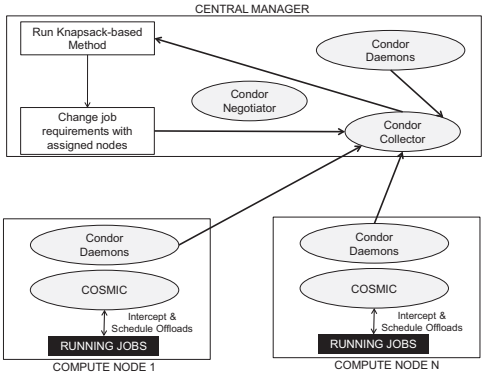

Fig. 6. Augmenting Condor with the proposed scheduler.

we then change each job's requirements by specifying the compute node obtained by the knapsack method as the only node on which the job can run. Specifically, we change the "Requirement" field in the job script and assign the selected compute node using "$Name ==< slotId > @ < NodeName >$". This forces Condor to subsequently dispatch the job to the compute node selected by our cluster component. Because the job is dispatched using Condor, we must wait for Condors next negotiation cycle which is triggered when the Condor collector obtains the changed job requirements. In order to reduce overheads, we submit the edited job requirements in a batch. That is, we process the entire list of pending jobs and submit the new requirements together to the Condor collector.

2) COSMIC on the Compute Node: At the node level, we use the COSMIC Xeon Phi middleware proposed in [6]. COSMIC is a transparent add-on to MPSS to handle thread and memory oversubscription when multiple processes compete for the Xeon Phi within a single server node. While not absolutely necessary, a middleware such as COSMIC on each node helps achieve the following:

- It terminates jobs whose memory exceeds the user specified limit. COSMIC does this using Linux containers [6]. Even though our knapsack formulation guarantees

adherence to user-specified memory limits, it cannot compensate for a user's mistakes such as underestimating the memory of a job.

- It manages accidental thread oversubscription by scheduling Phi offloads from different jobs in such a way that thread oversubscription is minimized, i.e., offloads that require more than the available threads in hardware will not be allowed to execute. The knapsack algorithm uses threads to determine the value of each job, and Xeon Phi memory as the knapsack capacity. COSMIC schedules Xeon Phi offloads within the jobs to prevent thread oversubscription.
- It balances load across the Xeon Phi cores. For instance, if two jobs require 120 threads (or 30 Xeon Phi cores supporting 4 threads per core) each, COSMIC automatically affinitizes threads to cores such that the jobs do not overlap and core utilization is maximized. Each job will run on its own set of 30 cores, utilizing all 60 cores on the device. Without COSMIC, such thread-to-core affinitization is not possible resulting in performance loss since two offloads with conflicting affinities may overlap and use the same cores leaving other cores idle.

Limitations: Our scheduling approach is static, and applies to a set of jobs waiting to execute. The set could represent a snapshot in a dynamic scenario with continously arriving jobs. Our approach is mainly beneficial if there are far more jobs than coprocessors: if the number of jobs is very small, naive scheduling approaches will likely perform equally well. We note however that our approach can also be used in a dynamic context, but that is outside the scope of this work.

V. Experimental Evaluation

In this section, we evaluate the effect of coprocessor sharing at the cluster-level. We use the workloads from Table I, as well as several synthetic workloads on an 8-node Xeon Phi-based cluster containing 1 Xeon Phi device per server node, each with 8GB of memory. All servers have 2 Xeon host processors with 8 cores each, and run RHEL 6.2 with Intel MPSS gold. We use Condor 7.8.7.

Our goal is to evaluate the proposed ideas in terms of makespan and coprocessor footprint reduction. In each experiment, we compare the following three cluster software configurations, abbreviated as **MC, MCC and MCCK**:

1) **MC (MPSS + Condor):** This is the baseline using MPSS and Condor, where jobs are run sequentially on each Xeon Phi with no concurrency or sharing.
2) **MCC (MPSS + Condor + COSMIC):** We add COSMIC to MPSS+Condor in order to allow job concurrency on the Phi's. COSMIC allows safe sharing without oversubscription, but is restricted to a compute node and unaware of the cluster level. Jobs are selected randomly at the cluster level: they are packed arbitrarily to Xeon Phi coprocessors and COSMIC prevents them from oversubscribing memory and threads.
3) **MCCK (MPSS + Condor + COSMIC + Knapsack Cluster Scheduler):** We add our proposed cluster scheduler to MPSS+Condor+COSMIC. The cluster scheduler decides which jobs should share Xeon Phi's for minimizing makespan thereby reducing footprint.

The rest of this evaluation section is organized as follows:

- *Makespan and Footprint Reduction*: We first show makespan and footprint reduction for a large set of real Xeon Phi jobs. We compare makespan obtained by MC, MCC and MCCK for the 8-node cluster. Then we show footprint reduction for MCC and MCCK, i.e., how much the cluster size can be reduced for MCC and MCCK and still achieve the same makespan that was obtained by MC on the 8-node cluster. Reduction in cluster size is obtained because the sharing-aware scheduler reduces the number of Xeon Phi's required. Since the jobs are coprocessor intensive, this directly results in overall cluster footprint reduction.
- *Sensitivity*: We show that our approach is beneficial across job sets with different memory and thread resource distributions. This is important since sharing a coprocessor is contingent on job resource requirements. Too many "big" jobs will force less sharing. Therefore we create synthetic job sets with different resource distributions, i.e., job sets skewed towards jobs with high memory and thread requirements to jobs sets skewed towards low memory and thread requirements. We perform controlled experiments to evaluate makespan and footprint reduction across these distributions for MC, MCC and MCCK.

A. Makespan and Footprint Reduction

We generate 1000 instances from the real Xeon Phi workloads listed in Table I, and measure the makespan for each of the above cases.

Table II shows the makespan on an 8-node cluster using MC, MCC and MCCK. With MC (MPSS+Condor) as the baseline, introducing COSMIC from [6] improves makespan by 27% because COSMIC allows jobs to share the Xeon Phi. However this uses random job selection at the cluster level. Introducing our knapsack-based job scheduling at the cluster level further improves the makespan to achieve a total improvement of 39% over the baseline.

Table II also shows cluster footprint reduction, i.e., for MCC and MCCK, the cluster size required to achieve the same makespan as the baseline (MC) on an 8-node cluster. We see that introducing COSMIC allows us to achieve the same makespan with a cluster of size 6 server nodes. The proposed cluster scheduling technique together with COSMIC can achieve the same makespan as an 8-node cluster with only 5 nodes. This is a 37.5% reduction in cluster size. This assumes coprocessor-intensive jobs and that there is no contention for the host by reducing cluster size: therefore reducing the number of coprocessors results in overall cluster size reduction.

Makespan and footprint reduction occur as a direct consequence of coprocessor sharing. The baseline case has no sharing while COSMIC allows sharing resulting in some improvement. But the sharing-aware cluster scheduler suitably

Configuration	MAKESPAN		FOOTPRINT	
	On 8-node cluster	Reduction compared to MC	Cluster size to achieve same makespan as 8-node cluster	Reduction from 8-node cluster
MC	3568	-	-	-
MCC	2611	27%	6	25%
MCCK	2183	39%	5	37.5%

TABLE II
MAKESPAN AND FOOTPRINT REDUCTION.

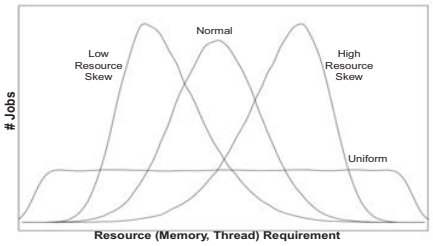

Fig. 7. Resource distributions of the synthetic job sets.

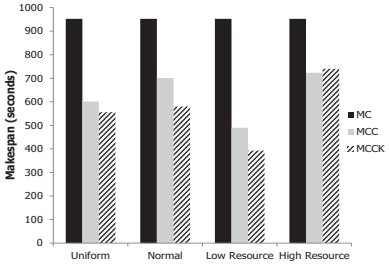

Fig. 8. Makespan reduction for different job distributions.

chooses jobs at the cluster level, and significantly increases coprocessor sharing.

B. Sensitivity to Job Resource Requirements

The ability to share jobs is based on job resource (memory and thread) requirements. If most jobs have small memory and thread requirements, significant concurrency can be achieved on the Xeon Phi. On the other hand, if most jobs have high memory and thread requirements, only limited concurrency is possible, resulting in smaller improvements.

In order to study sensitivity of our cluster scheduling technique to job resource requirements, we construct 4 sets of 400 synthetic Xeon Phi offload jobs with the following different distributions (Fig. 7):

- *Uniform*: Jobs are equally distributed across different resource requirements.
- *Normal*: The distribution curve is normal, i.e., most jobs are in the mid-resource range.
- *Low Resource Skew*: Jobs are skewed towards having low memory and thread resource requirements. The mean is 1 standard deviation from the normal mean towards lower resources.
- *High Resource Skew*: Jobs are skewed towards having high memory and thread resource requirements. The mean is 1 standard deviation from the normal mean towards higher resources.

Fig. 7 shows the 4 distributions, with the horizontal axis representing both memory and thread resources. We assume that jobs with low Xeon Phi memory requirements also have low thread requirements, and vice versa.

We now evaluate makespan for a cluster of size 8 nodes for the four different distributions. Fig. 8 shows the sensitivity of makespan improvements. The chart shows the distributions on the horizontal axis, and makespans for MC, MCC and MCCK on the vertical axis. For uniform, normal and low resource skew distributions, we see big improvements in makespan due to our scheduling approach. This is because in these cases, there are several jobs with low to medium resource requirements and therefore many jobs can be concurrently scheduled resulting in considerable Xeon Phi sharing. However, the high resource skew distribution has many big jobs which cannot concurrently execute without oversubscribing resources. Therefore, the improvement in makespan is smaller. But our scheduling approach does not degrade makespan significantly compared to MCC (which uses random job selection). The small degradation seen is due to our module having to externally integrate with Condor, and having to wait for Condor's scheduling cycle (see Section IV-D1). Furthermore, even with a distribution skewed towards high resource jobs, Xeon Phi sharing always improves makespan compared to the baseline MC where jobs run exclusively on the coprocessor.

Now we evaluate how our technique reduces cluster footprint across different job distributions, i.e., if our technique can produce the same makespan with a smaller cluster containing fewer Xeon Phi coprocessors for job sets with different resource demands. In order to do this, we measure makespan on clusters of progressively increasing sizes for the 400 synthetic jobs across the 4 resource usage distributions. The overall results are shown in Fig. 9. For large cluster sizes, we see that MCC and MCCK show smaller improvements than on small clusters when compared to the baseline MC where jobs run one after another. However, MCCK provides a large improvement over MCC for large clusters. This shows that the knapsack-based scheduling approach makes a difference when the cluster size is large, since a large cluster requires more decision making about where each job must be scheduled. For very

Configuration	Footprint (Cluster Size)			
	Uniform	Normal	Low Resource Skew	High Resource Skew
MC	8	8	8	8
MCC	6 (25%)	6 (25%)	4 (50%)	6 (25%)
MCCK	5 (37.5%)	5 (37.5%)	3 (67.5%)	6 (25%)

TABLE III
FOOTPRINT REDUCTION FOR DIFFERENT JOB DISTRIBUTIONS.

small clusters (e.g., 2 nodes), any form of sharing is beneficial over sequential, even random sharing provided by MCC. Therefore for such clusters, naive scheduling approaches are equally effective.

Table III shows the cluster footprint reduction summarized from the earlier experiments. MC has the same footprint regardless of distribution since it executes all jobs sequentially on each Xeon Phi. For the other distributions, we see an effect similar to makespan: for the uniform, normal and low resource skew distributions, MCCK reduces footprint by 37.5% to 67.5%, while for the high resource skew distribution, footprint is still reduced by 25%.

From Fig. 9, we observe that for very small clusters, any sharing is beneficial. So even MCC, in which random jobs share coprocessors, performs just as well as the knapsack-based approach. However, that is due to high "job pressure" or jobs per node for small clusters since we keep the number of jobs constant at 400. At high job pressure with very few nodes to decide between, intelligent job scheduling has little effect.

Therefore we now examine if our technique is effective at high job pressures but with more cluster nodes, so that cluster-level scheduling can make a difference. Therefore we examine what happens at high job pressures for larger clusters. Specifically, we increase the number of jobs from 400 to 1600 as we increase the cluster size from 2 to 8. Fig. 10 shows the makespan for this experiment under the normal resource usage distribution. We note that for large clusters, even at high job pressure, MCCK improves makespan by 11% over MCC and by 40% over MC. This demonstrates that cluster-level scheduling and coprocessor sharing together are beneficial

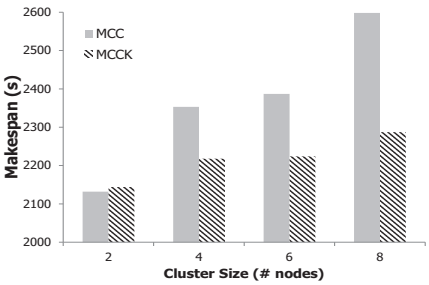

Fig. 10. Makespan with constant job pressure.

across different job distributions and at high job pressures.

VI. RELATED WORK

Considerable prior research exists in cluster scheduling. In [14], the authors propose delay scheduling to address the problem of scheduling for fairness and data locality in Hadoop clusters. When a job or its task that should be scheduled according to fairness cannot find a local node, it waits for a small amount of time, letting other jobs launch tasks instead. Quincy [15] also addresses the problem of scheduling with fairness and data locality for clusters where application data is stored on the compute nodes. They formulate the problem using a graph with associated costs and solve for min-cost flow. [16] presents a weighted max-min fair sharing approach in grids where tasks are assigned resources as close as possible to their fair shares. [17] describes a strict fairness technique for HPC systems: jobs used to backfill are preempted if their runtime predictions are incorrect and they delay high priority jobs. Since this guarantees fairness, their scheme aggressively starts jobs to backfill vacant slots in order reduce performance loss caused by the preemption. There is also considerable work in bag-of-tasks workloads [18]. In [19], the authors present a scheme to optimize cost when scheduling such applications on commercial clouds by learning to estimate task completion times. [20] proposes a scheduling method for bag-of-tasks applications on an auction-based grid where users bid to gain more resource shares.

Closer to our core scheduling method exist many efforts in minimum makespan scheduling [12], [13]. Compared to these

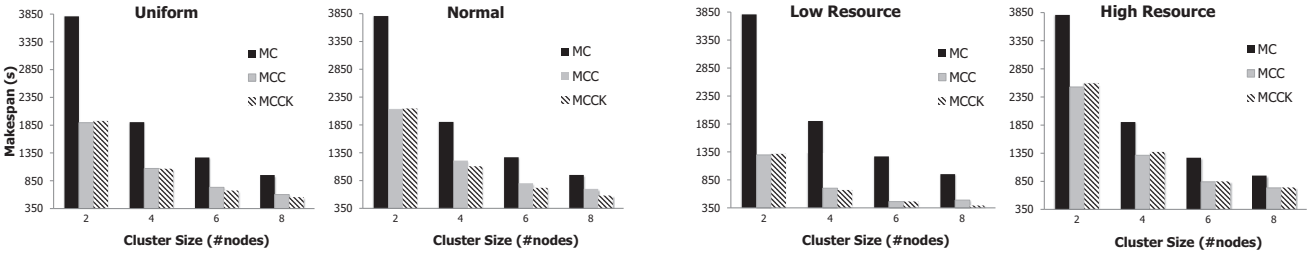

Fig. 9. Effect of cluster scheduling techniques on different sized clusters.

traditional approaches to solving this problem, ours has the following fundamental differences, as described earlier: (i) we specifically target coprocessor-based clusters, (ii) we do not assume jobs are sequential, but allow concurrency, (iii) we assume no knowledge of job execution times, while classical makespan minimization requires this information, which is often hard to specify accurately, (iv) we maximize concurrency as a proxy for minimizing makespan and (v) we consider coprocessor resource constraints.

VII. Conclusion

In this paper, we propose a scheduling method for Xeon Phi manycore coprocessor-based clusters. Existing cluster managers generally implement an exclusive allocation policy where coprocessors are not shared across jobs. This is primarily due to unpredictable effects of resource oversubscription in manycores.

We find that the exclusive allocation policy results in serious under-utilization of the Phi: for typical workloads, we measured average core utilization across the cluster to be only 38%. In order to address this, our framework relaxes the constraint that coprocessors cannot be shared across jobs. To avoid resource oversubscription, we use job resource requirements specified by the user, and propose a knapsack-based algorithm to pack as many jobs as possible on each Xeon Phi coprocessor while adhering to resource constraints. By increasing job concurrency, we find that we can reduce makespan for a set of jobs, or equivalently reduce cluster size for a set of jobs if makespan is constant. The knapsack framework simultaneously considers memory as well as thread constraints, and the proposed formulation is able to reduce makespan without requiring users to specify job execution times.

We implement and integrate our module with Condor, the distributed job scheduler, COSMIC, a recently proposed node-level Xeon Phi middleware that allows jobs to safely share a coprocessor, and MPSS, Intel's Xeon Phi software. Our integration is completely transparent to users and the underlying software, and requires no sources of Condor, COSMIC or MPSS. We test the end-to-end system on real Xeon Phi workloads, as well as synthetic workloads in controlled experiments. Makespan and footprint reductions of more than 50% due to coprocessor sharing are observed.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," vol. 28, no. 2, p. 3955, 2008.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - processor: A 64-core SoC with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb. 2008, p. 88 598.

[3] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione, "Early experiences with the intel many integrated cores accelerated computing technology," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, 2011, p. 21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016764

[4] *Torque Resource Manager*. Adaptive Computing. [Online]. Available: http://www.adaptivecomputing.com

[5] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, Oct. 2001.

[6] S. Cadambi, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar, "Cosmic: middleware for high performance and reliable multiprocessing on xeon phi coprocessors," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 215–226. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462921

[7] S. Suchyta, *Scheduling Jobs onto Intel Xeon Phi using PBS Professional*. [Online]. Available: http://www.pbsworks.com/Product.aspx?id=27

[8] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar, "Apricot: an optimizing compiler and productivity tool for x86-compatible many-core coprocessors," in *Proceedings of the International conference on Supercomputing*, 2012, p. 4758. [Online]. Available: http://dl.acm.org/citation.cfm?id=2304585

[9] C. J. Newburn, R. Deodhar, S. Dmitriev, R. Murty, R. Narayanaswamy, J. Wiegert, F. Chinchilla, and R. McGuire, "Offload compiler runtime for the intel xeon phi coprocessor," in *ISC*, 2013, pp. 239–254.

[10] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu, "A lightweight VMM on many core for high performance computing," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, p. 111120. [Online]. Available: http://doi.acm.org/10.1145/2451512.2451535

[11] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," Technical Report NAS-99-011, NASA Ames Research Center, Tech. Rep., 1999. [Online]. Available: http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf

[12] V. V. Vazirani, "Minimum makespan scheduling," in *Approximation Algorithms*. Springer, 2003, pp. 79–83.

[13] M. Englert, D. Ozmen, and M. Westermann, "The power of reordering for online minimum makespan scheduling," in *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. IEEE, 2008, pp. 603–612.

[14] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, 2010, p. 265278.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, p. 261276.

[16] N. D. Doulamis, A. D. Doulamis, E. A. Varvarigos, and T. A. Varvarigou, "Fair scheduling algorithms in grids," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 11, p. 16301648, 2007.

[17] Y. Yuan, G. Yang, Y. Wu, and W. Zheng, "PV-EASY: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, p. 240251.

[18] A. Iosup, O. Sonmez, S. Anoep, and D. Epema, "The performance of bags-of-tasks in large-scale distributed systems," in *Proceedings of the 17th international symposium on High performance distributed computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, p. 97108. [Online]. Available: http://doi.acm.org/10.1145/1383422.1383435

[19] A. Oprescu and T. Kielmann, "Bag-of-tasks scheduling under budget constraints," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, p. 351359.

[20] A. Sulistio and R. Buyya, "A time optimization algorithm for scheduling bag-of-task applications in auction-based proportional share systems," in *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, Oct., pp. 235–242.